NTIA Report 98-348

Software Implementation of a Wideband HF Channel Transfer Function

David A. Sutherland Jr.

U.S. DEPARTMENT OF COMMERCE
William M. Daley, Secretary

Larry Irving, Assistant Secretary
for Communications and Information

April 1998

PREFACE

This work was performed by the Institute for Telecommunication Sciences, Boulder, Colorado for the National Communications System's Office of Standards and Technology under reimbursable order No: DNCR066007.

Programming language software, plotting software, and operating systems are mentioned in this report to adequately explain the content of the report and to allow readers to both understand and build upon this work. In no case does such identification imply recommendation or endorsement by the National Telecommunications and Information Administration, nor does it imply that the software and systems mentioned are necessarily the best available for this application.

ACKNOWLEDGMENTS

The author wishes to express his appreciation to Christopher J. Behm of the Wireless Networks Group at the Institute for Telecommunication Sciences (ITS), Boulder, Colorado for helping bridge the gap between past work and this present work; to Teresa L. Rusyn of Lockheed-Martin's Astronautics Division in Littleton, Colorado for her patient explanations of the mysteries of digital signal processing; and to James A. Hoffmeyer, former Chief of the Wireless Networks Group at ITS, for providing the opportunity to work in wideband HF radio simulation.

The author also wishes to extend his thanks to Dr. William A. Kissick, Timothy J. Riley, Natalie R. Sexton, and Jeffrey A. Wepman of ITS whose review comments and suggestions were not only extremely helpful in making this a more readable document but also in improving the technical clarity and accuracy of the manuscript. Chris Behm and Teresa Rusyn also provided many helpful comments and suggestions.

CONTENTS

Page

PREFACE iii

ACKNOWLEDGMENTS iv

FIGURES vii

TABLES viii

ABBREVIATIONS ix

1. INTRODUCTION 1

PART I. TECHNICAL AND BACKGROUND INFORMATION 3

2. THE PROPAGATION MODEL 3

3. THE TRANSFER FUNCTION 7

4. VERIFICATION 9

PART II. THE TRANSFER FUNCTION PROGRAM 19

5. TECHNICAL ASPECTS OF THE PROGRAM 19
- 5.1 Random Number Generation 19
- 5.2 Complex Number Arrays 19
- 5.3 Fast Fourier Transform 20
- 5.4 Computation of τ_l and α 20
- 5.5 Data Structures 20

6. USER'S GUIDE 21
- 6.1 The Input File 21
- 6.2 The Output Files 24
  - 6.2.1 The Transfer Function 24
  - 6.2.2 The Parameter Listing 24
- 6.3 Running the Program 25
  - 6.3.1 Windows 95 25
  - 6.3.2 Windows 3.1 / 3.0 25
- 6.4 Modifying the Program 26

CONTENTS (cont'd)

Page

7. CODE AND DOCUMENTATION 27
7.1 Purpose and General Description 27
7.2 Project File LEW1.CPP 28
7.2.1 Function **void main** 31
7.3 Project File LEW2.CPP 33
7.3.1 Function **void init** 35
7.3.2 Function **void input_data** 37
7.3.3 Function **void out1** 40
7.3.4 Function **void outit** 43
7.4 Project File LEW3.CPP 45
7.4.1 Function **void doit** 48
7.4.2 Function **void comp_arrays** 50
7.4.3 Function **double big_c** 53
7.4.4 Function **double little_el** 56
7.4.5 Function **double funvalue** 62
7.4.6 Function **void slicedo** 64
7.4.7 Function **pointer** to **float rvgexp** 68
7.4.8 Function **void get_2i_normals** 71
7.4.9 Function **double ran1** 73
7.4.10 Function **double ran2** 75
7.5 Project File LEW4.CPP 77
7.5.1 Function **void little_four** 79
7.5.2 Function **void imp** 80
7.6 Additional Information 83
7.6.1 Library Functions Used 83
7.6.2 Input Data File Format 84
7.6.3 Function Calling Hierarchy 85

8. SUMMARY 86

9. REFERENCES 87

APPENDIX: SCATTERING FUNCTION PROGRAM 91

FIGURES

Figure 1. Delay power profile 4

Figure 2. Scattering function above -10 dB for path 1 11

Figure 3. Scattering function contours above -3.1 dB for path 1 11

Figure 4. Spectral average contours above -3.1 dB of 80 scattering functions for path 1 12

Figure 5. Spectral average above -10 dB of 80 scattering functions for path 1 12

Figure 6. Spectral average contours above -3.1 dB of 80 scattering functions for path 2 14

Figure 7. Spectral average above -10 dB of 80 scattering functions for path 2 14

Figure 8. Spectral average contours above -3.1 dB of 80 scattering functions for path 3 16

Figure 9. Spectral average above -10 dB of 80 scattering functions for path 3 16

Figure 10. Spectral average above -10 dB of 80 scattering functions for paths 1, 2, and 3 17

Figure 11. Spectral average contours above the -10 dB level of 80 scattering functions for paths 1, 2, and 3 17

Figure 12. Spectral average above -5 dB of 160 scattering functions for path 4 18

Figure 13. Spectral average contours above -3.1 dB of 160 scattering functions for path 4 18

Figure 14. Horizontal and vertical asymptotes of function (26) above τ_U 57

Figure 15. Horizontal and vertical asymptotes of function (26) below τ_L. Inset: Zero crossing and function maximum illustrated at finer scale 57

TABLES

Table 1. Model Input: Skywave Path Parameters 9

Table 2. Verification Parameters 10

Table 3. Example Input Files 23

ABBREVIATIONS

ASCII American Standard Code for Information Interchange

DSP digital signal processing

FFT fast Fourier transform

HF high frequency (3-30 MHz)

IID independent and identically distributed

IIR infinite impulse response

ITS Institute for Telecommunication Sciences (Boulder, Colorado)

LCG linear congruential generator

MUF maximum useable frequency

NS/EP telecommunications National Security or Emergency Preparedness telecommunications

OTA over-the-air

PC personal computer

SOFTWARE IMPLEMENTATION OF A WIDEBAND HF CHANNEL TRANSFER FUNCTION

David A. Sutherland Jr.*

This report presents an analytic model implemented as the computer program of the transfer function of a wideband HF channel model for use in a hardware simulator. The transfer function is the basic input to the hardware simulator. The mathematical basis of the program and the propagation model is presented. Parameters that characterize the skywave paths of a particular HF ionospheric condition are inputs to the program. The program code is listed and documentation is provided. Graphical verification using spectrally averaged scattering functions indicates that the transfer function program performs well and should find use as both an engineering tool and as the basis for a new standard propagation model.

Key words: channel transfer function; HF channel model; HF propagation; scattering function; wideband HF

1. INTRODUCTION

Research for and development of a wideband high-frequency (HF) radio channel simulator in hardware has been in progress for several years at the Institute for Telecommunication Sciences (ITS), see, for example, Hoffmeyer and Nesenbergs [1], Vogler et al. [2], and Vogler and Hoffmeyer [3-5]. The need for such a simulator has been driven by the renewed interest in the HF band. In particular, the HF radio band is the primary band for emergency backup communications by both amateur radio operators and Government radio systems. Thus, the HF band is important to National Security or Emergency Preparedness (NS/EP) telecommunications systems.

The HF band is characterized by widely variable conditions depending on variations in long-term solar effects (the sunspot cycle) on the ionosphere, diurnal variations in the skywave channel, fading characteristics, man-made (ignition) and natural (lightning) noise, and interference due to the crowded worldwide spectrum. These variable characteristics of the channels in the HF band make it desirable to test radio systems and subsystems in the laboratory rather than incur the costs and complexity of actual point-to-point and network tests. For example, low-power spread spectrum and frequency-hopping systems may be effective in mitigating the harsh HF environment. Wideband simulation may be useful in testing the effectiveness of these systems in the laboratory, see, for example, Redding and Weddle [6]. Narrowband simulation, since it is usually limited to less than 12 kHz, is clearly insufficient in this regard.

---

*The author is with the Institute for Telecommunication Sciences, National Telecommunications and Information Administration, U.S. Department of Commerce, 325 Broadway, Boulder, CO 80303.

Unfortunately, HF channel simulation has mostly depended on narrowband simulators. The most widely recognized narrowband model is the Watterson model given in Watterson, Juroshek, and Bensema [7]. The Watterson model, although widely used, is extremely restricted, as Watterson himself made quite clear. The limitations include bandwidths less than 12 kHz, negligible delay dispersion, and a single skywave path. The Watterson model is also limited to channels with stationary behavior in delay and in frequency. As a result, hardware simulators depending on the Watterson model are valid only for bandwidth-limited, stable, and stationary channels. Hence, there is an ongoing need for a much wider bandwidth simulation model that more closely models the ionospheric effects of dispersion and multipath seen over HF channels. This is even more important now that digital signal-processing (DSP) technology is being developed and improved to handle the extreme effects seen over an HF channel.

This report describes an analytic model, implemented in a computer program, that supports such a desired wideband HF channel simulator with a 1-MHz bandwidth. The HF channel simulator in hardware emulates several aspects of an HF time-varying channel: Doppler shift, Doppler spread, delay spread, and delay offset. Future versions of the wideband simulator will also include noise and interference effects based on the models described in Lemmon and Behm [8,9].

The C language program runs on a personal computer (PC) in either the Windows 3.1 or the Windows 95 operating systems. It implements the discrete transfer function of the HF channel in question. The discrete transfer function is the basic input for the hardware simulator. Input for the program are the parameters that characterize each of the skywave paths of the HF channel being modeled plus additional parameters that specify numerical conditions for the computing run. The output consists of two ASCII files. The first is a listing of all the input parameters and all the computing parameters used by the program. The second output file is the transfer function. It consists of the complex Fourier coefficients that characterize the transfer function at each sampling point in time.

This report is divided into two parts. Part I (consisting of Sections 2 - 4) contains background information for the channel model and information on the transfer function. Section 2 is a description of the HF channel model. Section 3 provides the mathematical basis for the calculation of the transfer function. Section 4 presents graphical verification of the software by means of scattering functions.

Part II (consisting of Sections 5 - 7) contains the program listing and the documentation. Section 5 discusses and explains specific aspects of the program such as special techniques and random number generation. Section 6 is a user's guide that explains details of the input and output files and tells how to run the program. Section 7 contains the program documentation and the program code.

Summary and reference sections follow. An appendix with the code listing of the program used to produce the data for the scattering functions of Section 4 is also included.

PART I. TECHNICAL AND BACKGROUND INFORMATION

2. THE PROPAGATION MODEL

This wideband HF channel model is given as the impulse response function

$$h(t,\tau) = \sum_{n=1}^{N} h_n(t,\tau) \ , \tag{1}$$

where the independent variables are time t and delay τ, and $h_n(t,\tau)$ is the impulse response for one of N different ionospheric propagation paths (reflecting ionospheric layers) indexed by n. For each path n,

$$h_n(t,\tau) = \sqrt{P_n(\tau)}\, D_n(t,\tau)\, \psi_n(t,\tau) \ , \tag{2}$$

where $P_n(\tau)$ is the delay power profile, $D_n(t,\tau)$ is the deterministic phase function, and $\psi_n(t,\tau)$ is a stochastic modulating function.

For the nth propagation path, the delay power profile is given by

$$P_n(\tau) = A \frac{\alpha^{\alpha+1}}{\Delta \Gamma(\alpha+1)} x^{\alpha} e^{-\alpha x} \ , \tag{3}$$

where

$$x = \frac{\tau - \tau_c}{\Delta} + 1 \ , \tag{4}$$

A is the maximum received power, α is the shape parameter, τ_c is the mean delay at the center frequency (which fixes the delay offset), $\Delta = \tau_c - \tau_l$ (which serves as a scale parameter), τ_l is the point in delay such that, for (3), $P_n(\tau_l) \equiv 0$, and $\Gamma(\cdot)$ is the gamma function defined as

$$\Gamma(y) = \int_{s=0}^{\infty} s^{y-1} e^{-s} ds \ , \tag{5}$$

for y a positive integer. Equation 3 describes a form of the gamma probability distribution function. See Law and Kelton [10, p. 332] for details on the gamma distribution. An example delay power profile is shown in Figure 1 where A_{fl} is the receiver threshold, σ_τ is the delay spread at A_{fl}, and σ_c is the rise time to τ_c relative to A_{fl} with the restriction that

$$\sigma_c < \frac{\sigma_\tau}{2} . \tag{6}$$

The restriction is necessary for the existence of τ_l.

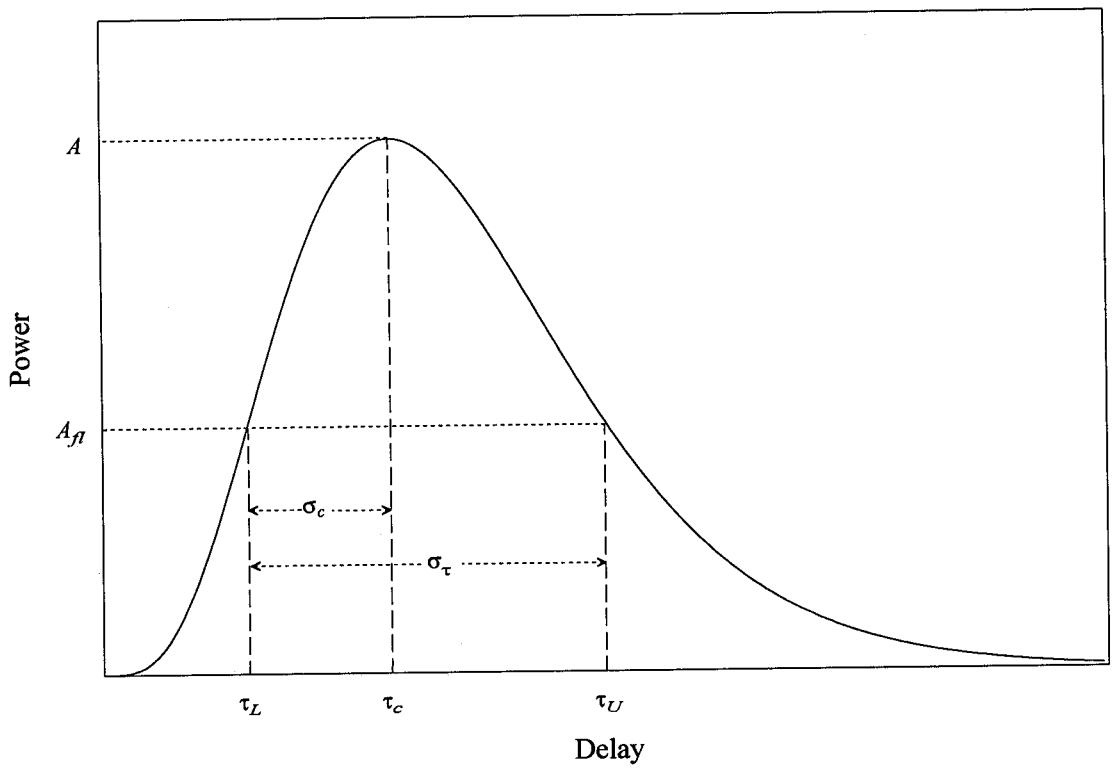

Figure 1. Delay power profile.

The deterministic phase function is given by

$$D_n(t,\tau) = \exp\{i2\pi[f_s + b(\tau - \tau_c)]t\} , \tag{7}$$

where f_s is the Doppler shift at τ_c and b is the slope or rate of change of Doppler shift with respect to τ. The dependence of phase on delay is the "slant" phenomenon described in the literature, see, for example, Wagner et al. [11].

The stochastic modulating function $\psi_n(t,\tau)$ is a set of random processes that models the effects of the constantly changing electron density in the ionospheric-reflecting layers that cause random variations in the received signals over HF channels. In this model, the processes will exhibit exponential autocorrelation in time as well as Rayleigh fading, a widely accepted characteristic of HF propagation, see Davies [12, pp. 242-245]. The random processes used in this model are described more completely in the next section.

The channel-scattering function relates the delay spread, the delay offset, the Doppler shift, and the Doppler frequency spread. It also provides a graphical method to display the signal energy distribution. The delay spread characterizes the phase variation of a received signal, while the Doppler shift and the Doppler spread characterize frequency variation. The scattering function $S(\tau, f_D)$ is the Fourier transform of the autocorrelation function of the impulse response given by

$$S(\tau,f_D) = \int_{\Delta t=-\infty}^{\infty} R(\tau,\Delta t)e^{-i2\pi f_D \Delta t} d\Delta t \ , \tag{8}$$

where f_D is the Doppler frequency, Δt is the autocorrelation time lag, and $R(\tau,\Delta t)$ is the autocorrelation function of the impulse response given by

$$R(\tau,\Delta t) = C_0 \int_{t=-\infty}^{\infty} h^*(\tau,t)h(\tau,t + \Delta t)dt \ , \tag{9}$$

where C_0 is a normalizing constant and * denotes complex conjugation, see Proakis [13, pp. 461-463]. For the Doppler spread having an exponential shape,

$$S(\tau,f_D) = P(\tau)\frac{\sigma_f}{\sigma_f + i2\pi(f_D - f_B)}e^{i2\pi\phi_0} \ , \tag{10}$$

where

$$f_B = f_s + b(\tau - \tau_c) \ , \tag{11}$$

ϕ_0 is a phase term, and σ_f is a function of the Doppler spread half-width σ_D given by

$$\sigma_f = \frac{2\pi\sigma_D s_v}{\sqrt{1 - s_v^2}} \ , \tag{12}$$

and where

$$s_v = \frac{A_{fl}}{A} \ . \tag{13}$$

More detailed explanations and descriptions of the model are given in Mastrangelo et al. [14]. The set of reports by Vogler and Hoffmeyer [3-5] also describes the development of the model. The scattering function, or ambiguity function, has been used to describe random channels in sonar and radar research as well as in HF radio. See Baggeroer [15] for an example or Middleton [16] for a general treatment.

3. THE TRANSFER FUNCTION

This section describes the discrete form of the transfer function of the channel model for implementation in a computer program. For this reason, the general mathematical definitions given in the previous section are rewritten in discrete form and are written to correspond with the functions, procedures, computations, and input parameters of the program. The channel transfer function is the discrete Fourier transform of the superimposed impulse responses given by (1) of each separate path with time held fixed.

The discrete form of the impulse response in terms of time t and delay τ is given by

$$h_{k,m} = \sqrt{P_k}\, c_{k,m}\, e^{i\eta_{k,m}} , \tag{14}$$

where the symbols and functions are explained in detail below. The independent variables delay τ and time t are indexed by the integers k, $m = 0, 1, 2, \ldots$, respectively. The discrete forms are defined as $\tau_k = \tau_0 + k\Delta\tau$ and $t_m = t_0 + m\Delta t$ where $\Delta\tau$, the delay increment, and Δt, the sampling interval, are arbitrary, but dependent on the hardware simulator's update rate, memory capacity, and bandwidth limitations.

The first factor, under the radical of (14), is the discrete form of the delay power profile given by (3). Hence, P_k determines how the impulse response behaves as a function of delay τ and is given by

$$P_k = Ae^{\alpha[\ln(g)+1-g]} , \tag{15}$$

where A is the power at the expected value of delay τ_c associated with the center frequency, α is the delay spread shape factor, ln is the natural logarithm function, and g is given by

$$g = \frac{\tau_k - \tau_l}{\sigma_l} , \tag{16}$$

where l is the largest k such that, for (15), $P_l = 0$, and $\sigma_l = \tau_c - \tau_l$, which is the rise time or Δ from (3) and (4). The parameters α and τ_l are functions of the received signal threshold A_{fl}, the overall delay spread σ_τ, and σ_l. The iterative method for computing α and τ_l is discussed in the program documentation on page 20. The convenient formula in (15) is derived by taking the natural logarithm of a normalized version of (3), raising the simplified expression of the normalization to the power of e, and then multiplying by A. The normalization of (3) is obtained by taking the ratio of $P_n(\tau_k)$ to $P_n(\tau_c)$.

The last two factors of (14) describe the shape of the Doppler spread. The exponential factor is the deterministic phase function mentioned in (2) and given by (7). In the argument of the exponent,

$$\eta_{k,m} = 2\pi\left[f_s + b\left(\tau_k - \tau_c\right)\right]t_m \ , \tag{17}$$

where f_s is the Doppler shift at τ_c, and b is the rate of change of the Doppler shift between τ_L and τ_c, or

$$b = \frac{f_s - f_{sL}}{\tau_c - \tau_L} \ , \tag{18}$$

where f_{sL} is the Doppler shift at the lower end of the delay spread τ_L. Equation 17 is a linear approximation to the relationship between delay and Doppler shift. The relationship is obtained from a truncated Taylor series expansion; see section 2.1.3 in Vogler and Hoffmeyer [4]. In this program, the constant part of the Taylor's expansion, $2\pi\phi_0$, given in Vogler and Hoffmeyer [5], is set to zero, i.e., $\phi_0 = 0$; however, this factor can be allowed to vary and may be used as a correction factor. This relationship is responsible for the slanted ridges seen in scattering function measurement and is often referred to as "slant."

The factor $c_{k,m}$ represents the stochastic modulating function from (2) and is used to model the variation in the received signals observed over HF channels. The generation of these sequences in k and m is accomplished by the following single-pole infinite impulse response (IIR) filtering process on the complex random sequence ρ_m

$$c_{k,m} = (1 - \lambda)\rho_m + \lambda c_{k,m-1} \ , \tag{19}$$

where $c_{k,-1} = 0$ for an initial condition and

$$\lambda = e^{-\Delta t \sigma_f} \ , \tag{20}$$

where the factor σ_f is given by (12). As a result, the $c_{k,m}$'s, with k held fixed, will have the desired exponential autocorrelation in the time direction.

The real and imaginary parts of the sequence ρ_m are independent and identically distributed (IID) Gaussian sequences with common mean 0 and common variance 1. This is equivalent to the bivariate Gaussian process of the Watterson model which accounts for long-term fading statistics following a Rayleigh distribution, see Watterson [7].

4. VERIFICATION

Program verification is accomplished by a graphical technique. The purpose of this verification is to ensure that the program's output fairly represents the expectations of the model in terms of both the input parameters and the computed parameters. The scattering function (8) relates the delay spread, the delay offset, the Doppler shift, and the Doppler spread. The scattering function is the Fourier transform of the autocorrelation function of the impulse response. The magnitude of the scattering function $|S(\tau,f_D)|$ is used as the basic graphical verification tool.

Table 1 presents the input data used to obtain scattering functions for demonstration of verification. The input parameters are the path distance D, the center frequency f_c, the penetration frequency f_p, the layer thickness σ, the height of maximum electron density h_0, the amplitude A, the delay spread σ_τ, the part of the delay spread below the mean delay σ_c, the Doppler spread half width σ_D, the Doppler shift at the mean delay f_s, and the Doppler shift at the lower end of the delay spread f_{sL}. Delay spreads are limited to less than 800 μs since this is a practical limitation for the channel simulator in hardware. This data is from the same set as used by Vogler and Hoffmeyer [4,5] which was obtained from Wagner, Goldstein, and Meyers [17], Basler et al. [18], and Wagner and Goldstein [19]. The reader may consult these sources for descriptions of the channel and techniques used to obtain the input parameters and the measured scattering functions. True validation of the transfer function must wait for the development of the hardware HF channel simulator. The input parameters that can be verified from the contour plots of the spectral averages at the -3-dB level are the Doppler spread $2\sigma_D$, the delay spread σ_τ, the Doppler shift at the mean delay f_s, and the Doppler shift at the lower end of the delay spread f_{sL}.

Table 1. Model Input: Skywave Path Parameters

Path	D (km)	f_c (MHz)	f_p (MHz)	σ (km)	h_0 (km)	A	σ_τ (μs)	σ_c (μs)	σ_D (Hz)	f_s (Hz)	f_{sL} (Hz)
1	126.0	5.5	13.0	30.0	265.0	1.0	70.0	34.0	0.05	0.2	0.1
2	126.0	5.5	13.0	28.0	270.0	1.0	20.0	9.0	0.05	-0.1	0.0
3	126.0	5.5	13.0	28.0	271.5	1.0	30.0	14.0	0.1	0.05	-0.05
4	88.0	2.8	5.87	30.0	240.0	0.25	350.0	170.0	5.0	1.1	0.8

In addition to the input parameters, several computed parameters may also be verified: the mean delay τ_c, the slant (the ratio of the Doppler frequency to the mean delay), and the lower and upper ends of the delay spread, τ_L and τ_U respectively. The mean delay is computed from the effective reflection height and the great circle distance between transmitter and receiver. The effective reflection height is a function of the input parameters f_c, f_p, σ, and h_0, see p. 53 in the program documentation. This calculation depends on a hyperbolic secant-squared (sech^2) model of electron

density, see Budden [20, p. 156]. The verification parameters for the paths in Table 1 that can be verified are given in Table 2.

Figures 2 - 13 contain sequential data of the averaged spectrum at particular points in delay. A surface is fitted over the data points for purposes of illustration. The contour plots are taken from the fitted surfaces. The scales in delay may seem inconsistent since the tick marks (and grid lines) in delay represent one of the data sequences. The delay value at that point in the graph is rounded from the actual value.

Table 2. Verification Parameters

Path	$2\sigma_D$ (Hz)	σ_τ (μs)	f_s (Hz)	f_{sL} (Hz)	τ_c (μs)	**slant** (Hz/μs)	τ_U (μs)	τ_L (μs)
1	0.1	70.0	0.2	0.1	1,831	0.0029	1,867	1,797
2	0.1	20.0	-0.1	0.0	1,863	-0.0111	1,874	1,854
3	0.2	30.0	0.05	-0.05	1,872	0.0071	1,888	1,858
4	10.0	350.0	1.1	0.8	1,637	0.0018	1,817	1,467

The first three paths in Tables 1 and 2 are from the ordinary mode and two extraordinary modes from 1-hop F-layer returns measured over a quiet 126-km path in California in the winter, see Wagner et al. [11].

One difficulty in studying the scattering function is in the stochastic construction of the impulse response. This makes it impossible to graphically measure parameters such as Doppler spread from the scattering function graphs, even with a quiet channel, as shown in Figure 2. Above the -3.1-dB level there is little energy exhibited in the scattering function as seen in Figure 3. This is all located in the small area at approximately 0.18 Hz and 1,810 μs. The -3-dB level is where the important parameters are displayed and graphically measured. The choice of level for verification is arbitrary, but in this model the input parameters are based at the -3-dB level. For example, the Doppler spread is measured at the mean delay, which, in this case, is 1,831 μs. As can be seen, there is no spectral energy at that point.

Smoothing techniques to rectify this problem do not suffice since there is no guarantee that the smoothed spectrum obtained from one stochastically constructed impulse response is representative of the true spectrum. Another possibility is to sample for a longer duration, but this will bring computing time and memory issues into play. Therefore, a spectral averaging technique is used to obtain a useable spectrum. Figures 4 and 5 appear more reasonable for obtaining visual verification. These figures are the average of 80 "runs" or the average of 80 separately constructed scattering functions. The program for the numerical construction of the spectrally averaged scattering function

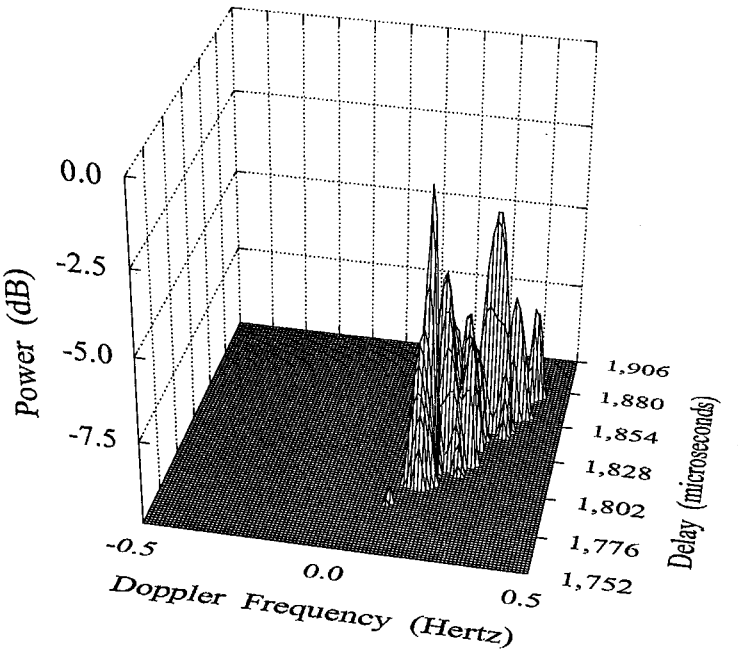

Figure 2. Scattering function above -10 dB for path 1.

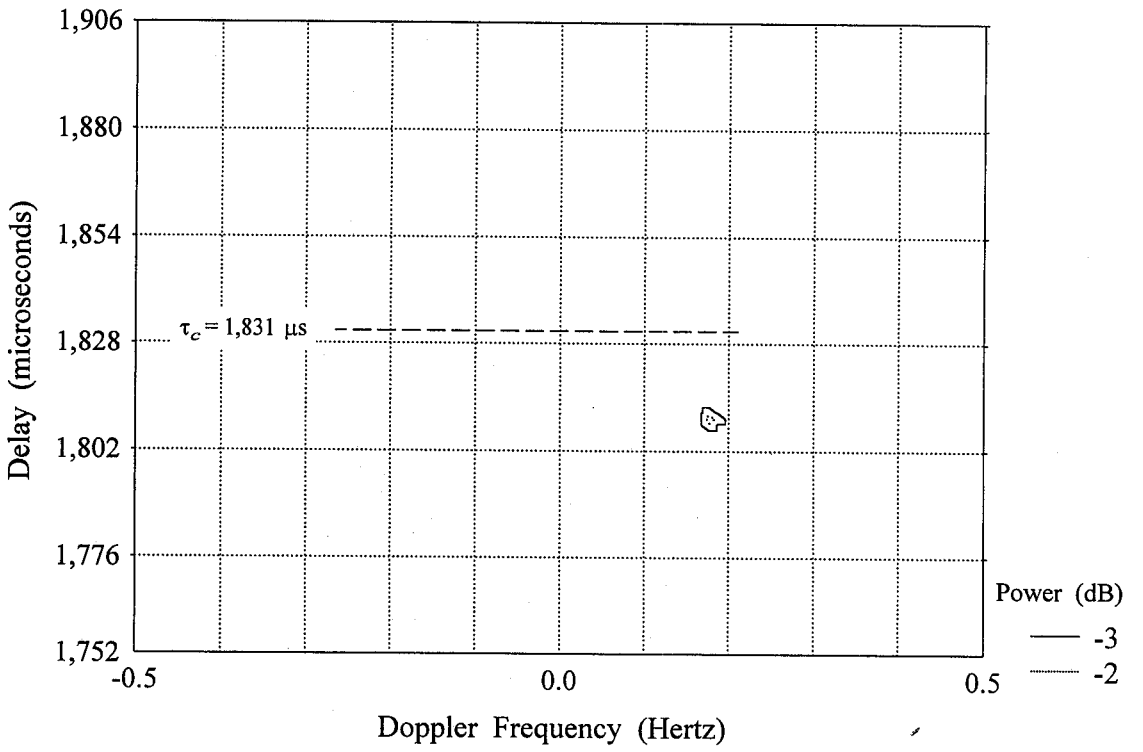

Figure 3. Scattering function contours above -3.1 dB for path 1.

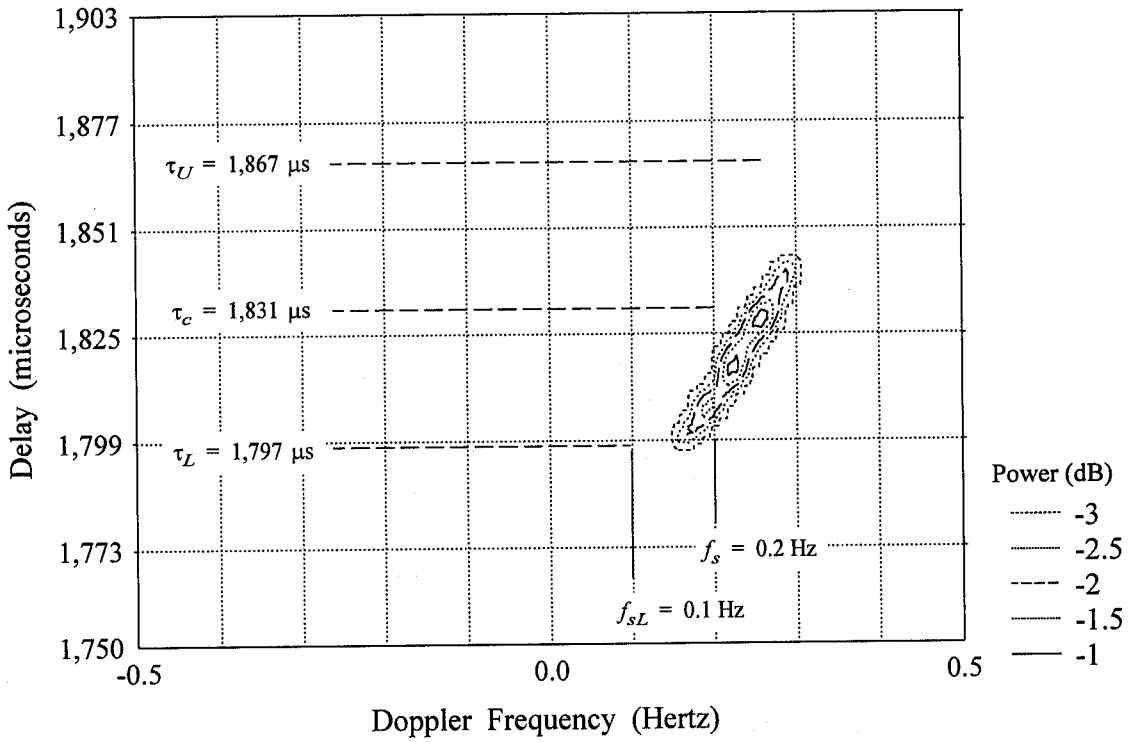

Figure 4. Spectral average contours above -3.1 dB of 80 scattering functions for path 1.

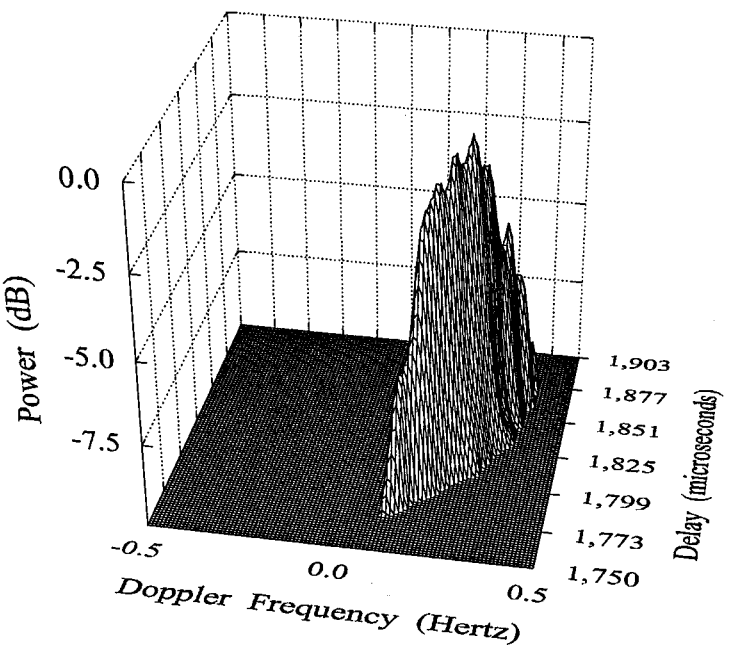

Figure 5. Spectral average above -10 dB of 80 scattering functions for path 1

is nearly identical to the transfer function program. The spectral averaging program is listed in the Appendix with documentation that describes the differences between the two programs.

The difficulty with spectral averaging is that it may take many computer runs to obtain a suitable average, especially for some ionospheric conditions, e.g., spread-F, which is defined as the phenomenon that is observed on ionograms displaying a wide range of echo pulse delay near the critical frequencies of the F layer, see, for example, Davies [12, p. 153].

Consider the first channel displayed in Figures 4 and 5. Notice that the averaged scattering function displayed in Figure 5, although considerably smoothed over that in Figure 3, still has some raggedness at the knife edge of the surface. For our purposes it still suffices for verification.

In Figure 4, the Doppler spread at the mean delay, $\tau_c = 1{,}831$ μs, measured across the -3-dB contour is approximately 0.06 Hz, which is smaller than expected by a factor of two-thirds. The delay spread across the -3-dB contour is also shorter than expected since the contours do not reach the upper end of the delay spread τ_U, although the -3-dB contour is close to the lower end of the delay spread τ_L. We estimated the slope of the line through the long axis of the -3-dB contour to be 250 μs/Hz or 0.004 Hz/μs, which is reasonably close to the predicted slant. It is believed that the discrepancies are due to the general filter design. The filter is a single pole IIR filter. Although the filter itself is a function of the sampling rate and the Doppler spread half width, it may be insufficient for accurate modeling of any one channel. Such discrepancies can be mitigated by design of digital filters to more carefully model a particular channel. This may be difficult since the raw data necessary may not be attainable due to the rapidly changing nature of the ionosphere.

The most noticeable discrepancy is that the Doppler shift at τ_c is about 0.05 Hz larger than expected. The discrepancy in Doppler shift is also noticeable at τ_L and appears to be the same offset as at τ_c. The shift appears to be constant through delay. This is also likely to be an effect caused by filtering. In this case, however, the effect may be mitigated by introducing a shift which is constant through delay. This constant shift term was mentioned in Section 3 as an arbitrary phase factor that had been set to zero. This is part of the deterministic phase function (7) which now becomes

$$D_n(t,\tau) = e^{i2\pi\{\phi_0 + [f_s + b(\tau - \tau_c)]\}} . \tag{21}$$

For path 2, there is even better graphical agreement with the parameters as seen in Figures 6 and 7. The mean delay $\tau_c = 1{,}863$ μs appears to cut the center of the -3.0-dB contour and intersects there with the expected Doppler shift; however, the maximum indicated by the -0.5-dB contour is at 1,860 μs and -0.05 Hz. The -3.0-dB contour extends approximately 16 μs, smaller than the 20-μs spread expected. The graphically measured slope of the line through the long axis of the -3.0-dB contour is -123.81 μs/Hz, which gives an estimated slant of -0.008 Hz/μs; this is close to the expected slant of -0.0111 Hz/μs. The Doppler spread at the mean delay is about 0.07 Hz, smaller than the 0.1-Hz spread expected at τ_c. The computed spectral average of scattering functions for this path is in fair agreement with the parameters without the discrepancies noted above for path 1.

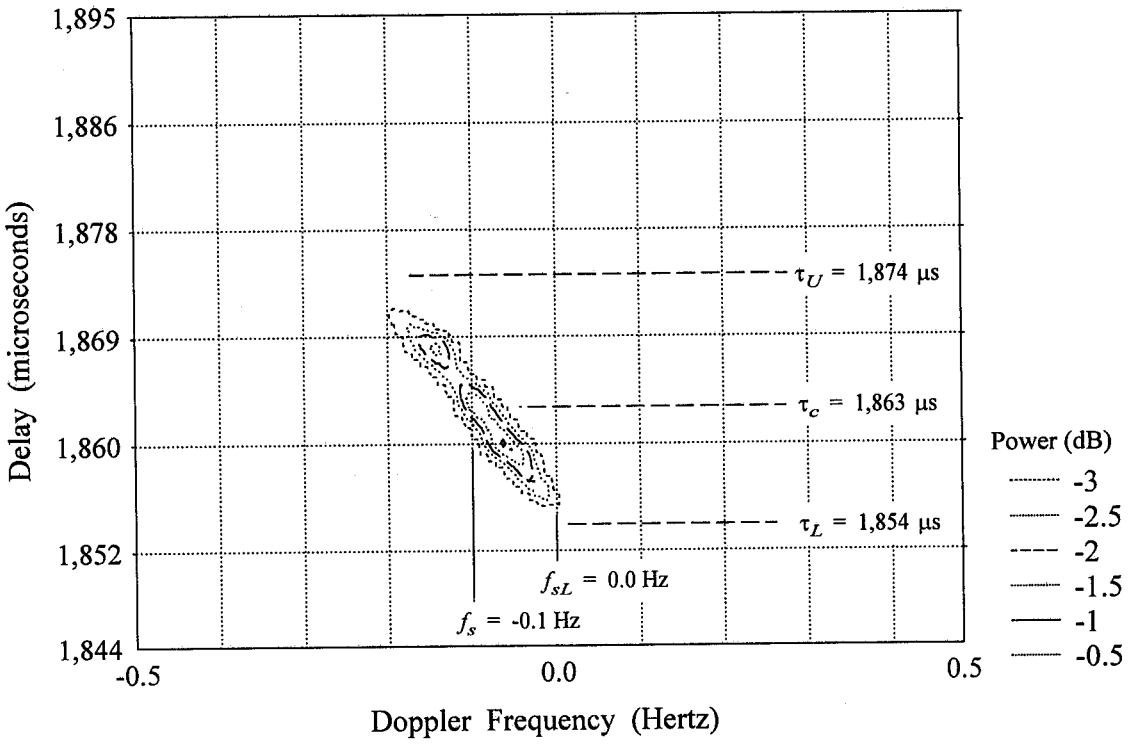

Figure 6. Spectral average contours above -3.1 dB of 80 scattering functions for path 2.

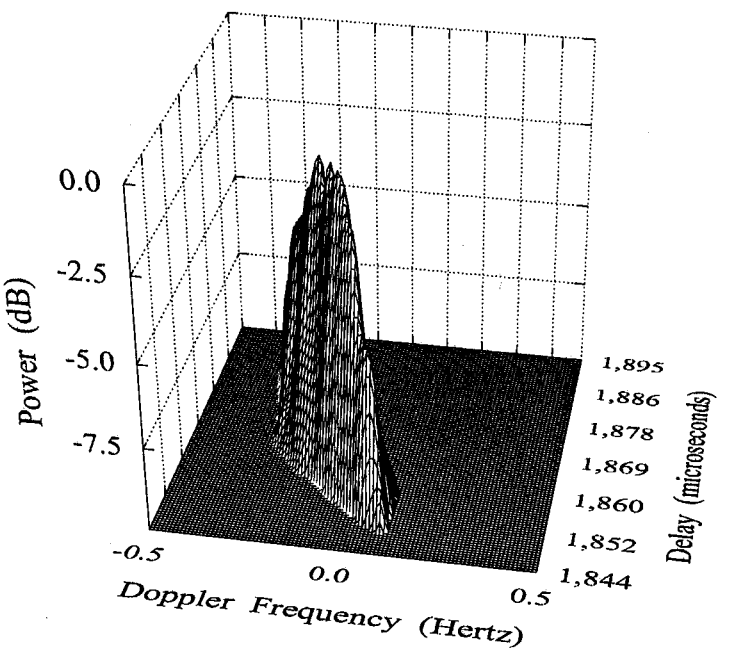

Figure 7. Spectral average above -10 dB of 80 scattering functions for path 2.

Figures 8 and 9 display the scattering function for path 3. There is a constant offset in Doppler frequency as seen before in path 1. The shift is about 0.04 Hz. The mean delay τ_c = 1,872 μs, touches the bottom of the -1.0-dB contour. The -3.0-dB contour extends for approximately 27 μs, which is short of the 30 μs expected. The -3.0-dB level is very close to τ_U = 1,888 μs, but τ_L = 1,858 μs is lower than the bottom edge of the -3.0-dB contour. Except for the shift in Doppler frequency the spectrally averaged result for this path is in fair agreement with the verification parameters.

An additional verification feature is the ability of the spectral average results to distinguish between separate paths. Figures 10 and 11 combine paths 1-3 into the actual observed situation of one ordinary mode and two extraordinary modes. Figure 10 shows two major divisions of energy. It is difficult to see the three paths on the three-dimensional rendering, but it is interesting to note that when the graphing software is "painting" the surface on the computer screen the three paths are dramatically distinct. The contour plot in Figure 11 clearly indicates the presence of three paths as the contours in the upper middle of the figure show two sections with distinctly different slopes.

Figures 12 and 13 present an intense spread-F situation over an 80-km Alaskan path, see Wagner and Goldstein [19]. The scattering function is the average of 160 runs. The mean delay of 1,637 μs appears to split the center of the contours. There is not enough resolution to clearly see the indication of slope, although the Doppler frequency at the mean delay looks reasonable; hence, there appears to be no unexpected offset in Doppler frequency as seen in two of the paths above. The Doppler spread appears to be about half of what is expected. The extent of the delay is about 65 μs short of the expected 350 μs. There appears to be good agreement with the verification parameters, but this situation probably requires more runs in order to obtain a smoother spectral average.

The model appears to agree well with the input and calculated parameters. An exception is the unexpected offset in overall Doppler frequency seen with two of the verification paths. This appears to be a function of the filter parameter λ given by (20). This parameter is itself a function of the sampling rate and the Doppler spread. This is a result of the general design of the model. The model is designed for a wide range of applications. To more carefully model a path with a particular ionospheric situation, it may be necessary to tailor the filter design to the situation. The filtering effect is also believed to be responsible for the delay shapes being shorter than expected. Another consideration is that it may be necessary to design the delay power profile (3) and (15) for the specific situation. The alternative is that the general model may need to be more complicated to fit more cases.

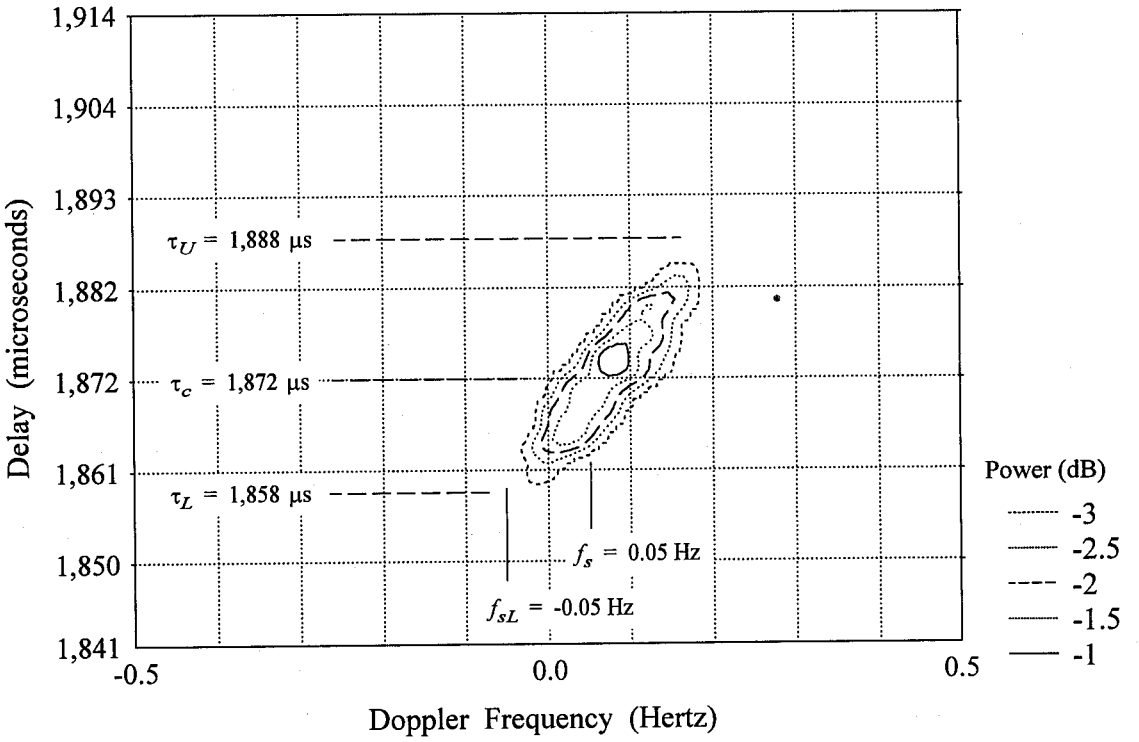

Figure 8. Spectral average contours above -3.1 dB of 80 scattering functions for path 3.

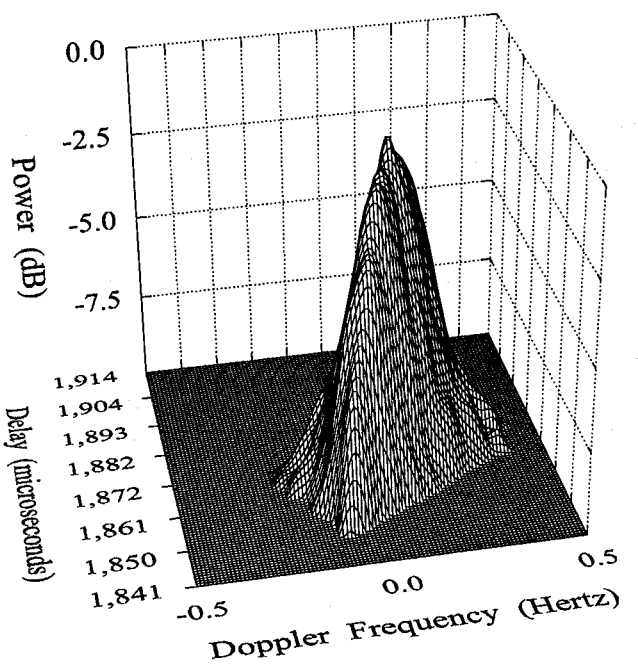

Figure 9. Spectral average above -10 dB of 80 scattering functions for path 3.

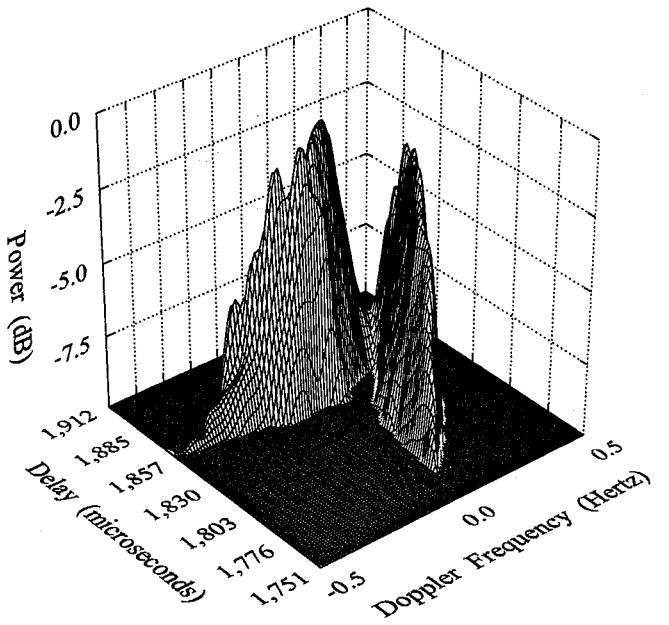

Figure 10. Spectral average above -10 dB of 80 scattering functions for paths 1, 2, and 3.

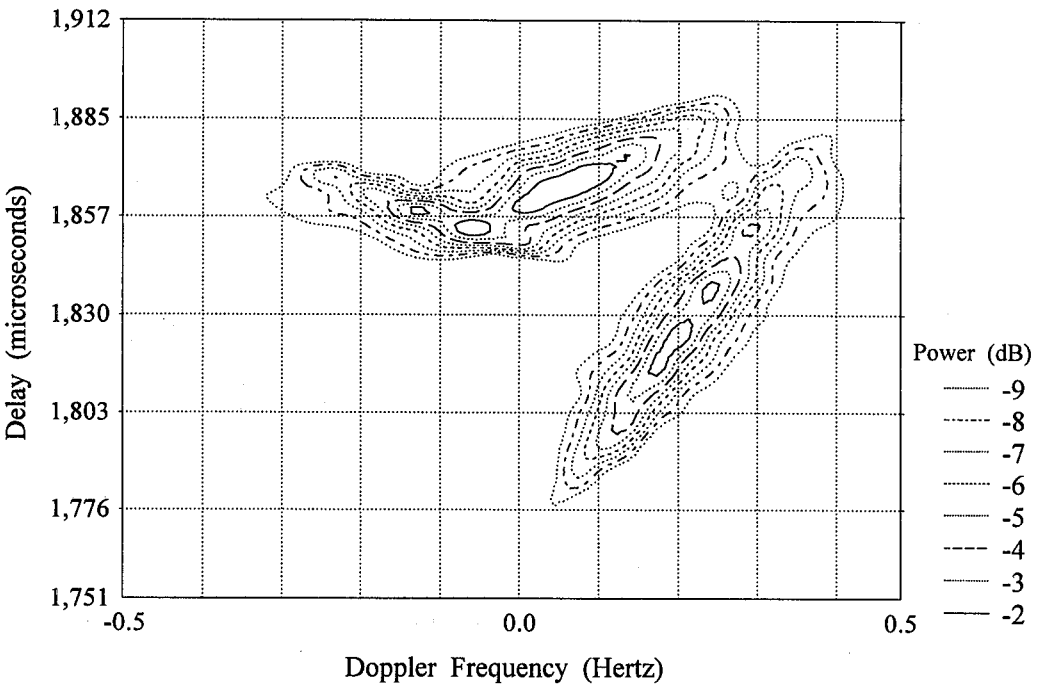

Figure 11. Spectral average contours above the -10 dB level of 80 scattering functions for paths 1, 2, and 3.

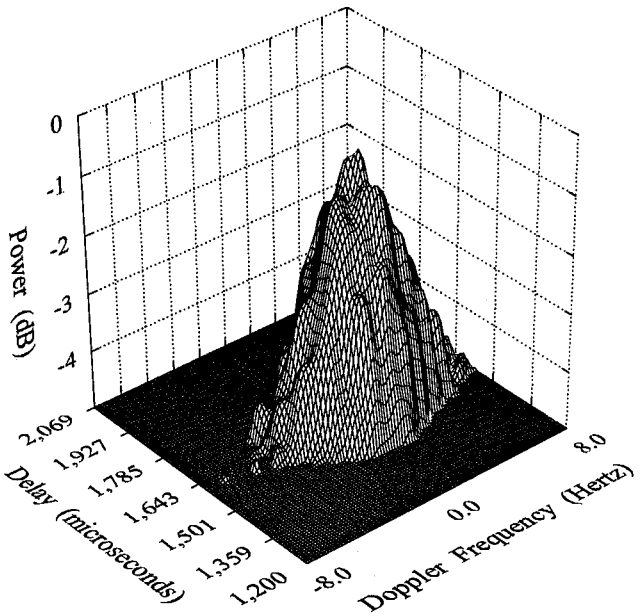

Figure 12. Spectral average above -5 dB of 160 scattering functions for path 4.

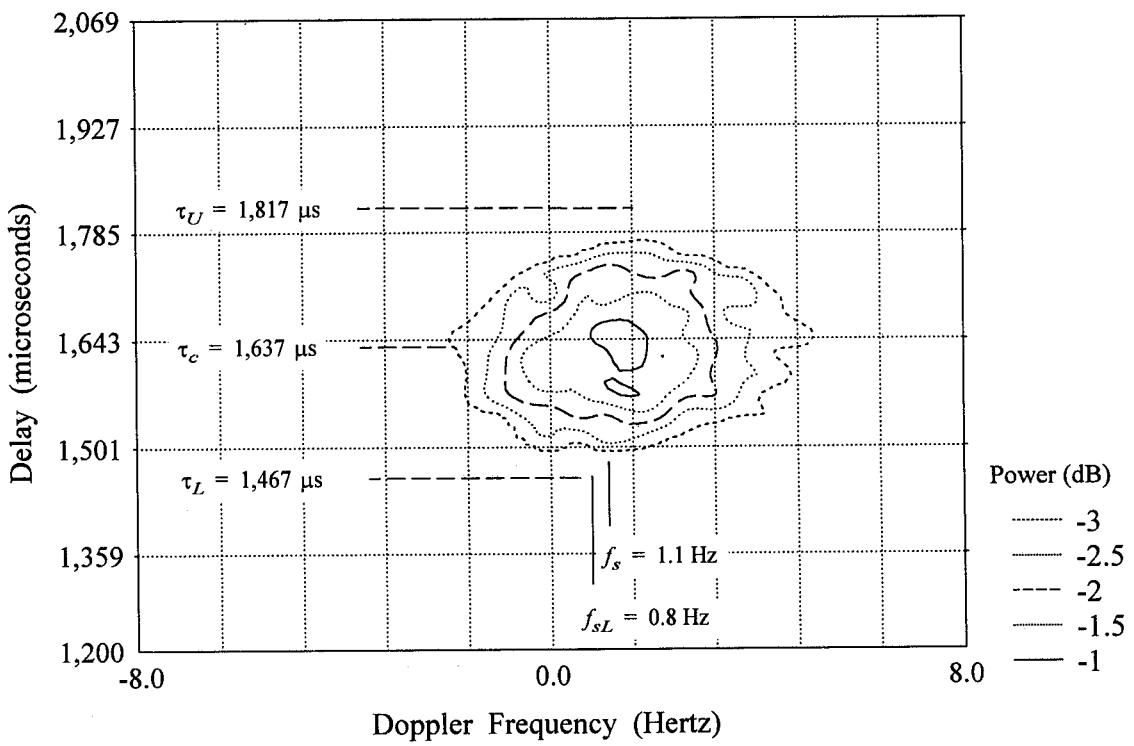

Figure 13. Spectral average contours above -3.1 dB of 160 scattering functions for path 4.

PART II. THE TRANSFER FUNCTION PROGRAM

5. TECHNICAL ASPECTS OF THE PROGRAM

This section discusses special techniques used in the design of the transfer function program. The larger data structures and the more complex or detailed routines are explained or justified. More details can be found in the documentation section, Section 7.

5.1. Random Number Generation

The assumption that long-term channel-fading characteristics follow a Rayleigh distribution requires two separate sequences from independent, normal distributions that have common zero mean and common unit variance. The polar method is used to generate two such sequences, see Law and Kelton [10, p. 491]. In turn, the polar method requires two separate sequences from independent uniform distributions on the interval (0,1) as input. To ensure independence, this program includes two different pseudorandom number generators that are each composites of several different linear congruential uniform random number generators (LCG).

The difficulty with a single LCG, which sometimes is used as input for the polar method, is that the normal distributions obtained may not be independent, see Brately, Fox, and Schrage [21, p. 223]. Example output sequences from the polar method, when plotted against one another, fall on a spiral! Clearly, these are not independent number streams.

5.2. Complex Number Arrays

An array of complex numbers is represented by an array of real numbers, that is, an array of floating point numbers. The even indices of the array, beginning with 0, indicate the real part of the complex number, while the odd indices indicate the imaginary part. The array elements with indices 0 and 1 are the real and imaginary part of the first complex number, respectively; the array elements with indices 2 and 3 are the second complex number, etc. The size of these arrays is driven by the channel simulation hardware requirements.

The random number input to the computation of the impulse response is held in a dynamically allocated array of floating point numbers. Stepping through this array is accomplished by pointer arithmetic. One section of this array is maintained for each ionospheric path. Dynamic memory is used since the program will support from one to three skywave paths. Hence, up to three separate segments of the array will be necessary. The array is used as both input and output for a digital filter.

5.3. Fast Fourier Transform

The fast Fourier transform (FFT) algorithm used returns the complex Fourier coefficients into the input array, hence only one array is necessary. The length of the input array is doubled by zero padding. This padding provides interpolation and helps avoid "wrap-around" effects.

5.4. Computation of τ_l and α

The parameters τ_l and α must be calculated iteratively since the equation for the gamma distribution has no closed form unless α is an integer. The computation of τ_l is accomplished by a bisection algorithm since the function being evaluated is not continuous. The presence of a natural logarithm in the function can produce complex results, see Section 7.4.4. The bisection algorithm is used since two estimates that lead to a result with the desired tolerance can be quickly found while avoiding the discontinuities. The shape parameter α is calculated directly from

$$\alpha = \frac{\ln s_v}{\ln Z_L + 1 - Z_L}, \tag{22}$$

where s_v is given by (13) and

$$Z_L = \frac{\tau_L - \tau_l}{\tau_c - \tau_l}, \tag{23}$$

see Vogler and Hoffmeyer [5, p. 32].

5.5. Data Structures

The first of the major data structures is a single element array containing one structure that holds all input and computed parameters that apply to all the paths. There is also an array of structures that holds all the input and calculated parameters particular to each sky wave path.

6. USER'S GUIDE

This section provides instructions to use the program. Several topics are discussed including input and output, and running the program under various environments.

6.1. The Input File

An input file contains a variable number of parameters depending on the number of ionospheric paths in the HF channel situation under consideration. The parameters are listed in order and separated by delimiters such as carriage return and line feed (CR-LF). The input files are ASCII files that consist of two types of information: data used by the program in the computing run and data descriptive of each path. The first five items in an input file are the "computing" data and are read into a single element array of type **compute** (see the documentation in Section 7). The italics here indicate a variable in the program while bold indicates a structure. Where a real number is indicated, a decimal point is required, for example, 326.0. The first five input elements are:

slices - an integer that indicates the number of time slices or samples in the time direction. The program computes the Fourier coefficients for a complete impulse response for each sample point.

delta_t - a real number that indicates the length of the sampling interval. This is given in microseconds for consistency with the units of the other time values in the input file. In Vogler and Hoffmeyer [5], this is indicated by the symbol t_m.

afl - a real number that indicates the receiver threshold. This is generally taken to be the half amplitude point (the 3-dB point).

paths - an integer that indicates the number of ionospheric paths modeled. The upper limit is three paths.

seed - an integer in the range 1 - 30,268, inclusive, that is used to initialize the random number seeds for the two uniform random number generators.

The parameters that follow serve to characterize the individual skywave paths. These parameters are all elements of the structure **ray_path**. For each path or layer, there are eleven parameters:

path_Distance - a real number that indicates the point-to-point distance, in kilometers, along the great circle between sender and receiver.

center_freq - a real number that indicates the center frequency, in megahertz, of the transmitting radio system.

penetrate_freq - a real number that indicates the penetration frequency, in megahertz, of the reflecting layer. Above this frequency there is no ionospheric reflection possible for this layer. The penetration frequency is generally higher than the maximum useable frequency (MUF) for the most highly ionized layer.

thick_scale - a real number that indicates the thickness, in kilometers, of the reflecting layer.

maxD_hgt - a real number indicating the height, in kilometers, of the maximum electron density of the layer.

peak_amplitude - a real number that indicates the maximum power A of the signal.

sigma_tau - a real number equal to the delay spread, in microseconds, at the half power point (3dB) A_{fl}.

sigma_c - a real number that indicates the rise time, in microseconds, of the impulse response with respect to the lower end of the delay spread τ_L.

sigma_D - a real number that gives the Doppler spread, in Hertz, at the half power point (3dB) A_{fl}.

fds - a real number that gives the Doppler frequency, in Hertz, indicating the Doppler shift at the mean delay τ_c.

fdl - a real number that equals the Doppler frequency, in Hertz, indicating the Doppler shift at the lower end of the delay spread τ_L.

If the last two parameters are equal, *fdl* = *fds*, then the transfer function will not be characteristic of the slant phenomenon. Slant characterizes Doppler frequency dependence on delay. This aspect of the channel is modeled as a linear relationship, see section 2.1.3. pp. 15-16 in Vogler and Hoffmeyer [4].

The input files can be generated with any editor that can save files in ASCII format. Examples of input files are given in Table 3. File 1 is an example input file for a situation with one layer. File 2 shows an example input file for two layers.

Table 3. Example Input Files

File 1	Parameter	File 2
512	*slices*	512
1,000,000	*delta_t*	500,000
0.5	*afl*	0.5
1	*paths*	2
1	*seed*	1
126.0	*path_Distance*	88.0
5.5	*center_freq*	5.3
13.0	*penetrate_freq*	7.07
30.0	*thick_scale*	20.0
265.0	*maxD_hgt*	225.0
1.0	*peak_amplitude*	0.8
70.0	*sigma_tau*	80.0
35.0	*sigma_c*	39.2
0.05	*sigma_D*	0.12
0.2	*fds*	-0.05
0.1	*fdl*	-0.05
	path_Distance	88.0
	center_freq	5.3
	penetrate_freq	5.56
	thick_scale	26.0
	maxD_hgt	228.0
	peak_amplitude	1.0
	sigma_tau	240.0
	sigma_c	117.8
	sigma_D	0.15
	fds	-0.05
	fdl	-0.05

6.2. The Output Files

6.2.1. The Transfer Function

This output file contains the complex Fourier coefficients that define the transfer function. The coefficients are written to the file as real floating point numbers without exponent and with a space between each field. There are 8,192 coefficients written for each time slice. Between the transfer functions for each time slice, a line feed is included; however, this should generally be seen as white space. The 8,192 coefficients represent 4,096 complex numbers with the real and imaginary parts alternating. The first number in the list is the real part of the first complex coefficient and the second number is the corresponding imaginary part, and so forth.

The present version of the HF channel hardware simulator expects coefficients for 299 time slices. Since the output file will be large, it is recommended that the transfer function software be run on the same machine driving the hardware simulator. Alternately, a network supporting large file transfers is recommended.

6.2.2. The Parameter Listing

This output file lists the parameters used by the program. All the input file parameters, as well as the important derived parameters are written to this file. The derived parameters are listed here in terms of their variable name in the program.

The computing parameters contained in the structure **compute** and derived by the program are as follows:

delta_tau - the delay interval in microseconds.

big_el - the earliest of the *tau_l* values for the layers in microseconds. If that value is negative, then *big_el* is set to 0.0 μs.

The following are the parameters computed for each layer contained in structure **ray_path** which are derived by the program:

tau_c - the mean delay associated with the carrier frequency, in microseconds.

sigma_f - value used to determine the Doppler spread shape.

slp - the slope of the linear relationship between the delay and the Doppler shift. The units are Hertz / microseconds.

tau_L - left end of the delay spread, in microseconds.

tau_U - right end of the delay spread, in microseconds.

tau_l - location parameter for the delay function in microseconds. This is not necessarily the same for every layer.

alpha - shape parameter for the delay power profile.

sigma_l - the rise time of the impulse response with respect to *tau_l* in microseconds.

lambda - exponential autocorrelation factor through time, for random input stream construction.

6.3. Running the Program

This section describes how to run the program in Windows 95 or Windows 3.0 / 3.1. It is assumed that the executable file, LEWS.EXE, is in the directory to which the output files will be written. The program writes the transfer function output file by appending to the file given on the command line. Hence, if the file name listed on the command line already exists, then the transfer function coefficients will be appended to that file. The program opens a new file for the parameter listing output file. Therefore, if the file name listed on the command line already exists, that file will be overwritten and all contents lost. The command line to start the program consists of the executing file, the input file, and the two output files:

< LEWS.EXE INPUT_FILENAME FIRST_OUTPUT_FILENAME SECOND_OUTPUT_FILENAME >

where FIRST_OUTPUT_FILENAME indicates the parameter listing output file and < > encloses the characters to be typed.

6.3.1. Windows 95

The executing program and the input file should be available in the same directory to which the output files will be written. Click the start button on the task bar. Click on Run from the start menu. Type the full command line, including the path to the directory (the browse capability may also be utilized). For example, type:

< D:\BORLANDC\BIN\LEWS.EXE INPUT.931 OUTPUTA.931 OUTPUTB1.931 >.

Click OK or hit the return key.

6.3.2. Windows 3.1 / 3.0

The executing program and the input file should be available in the same directory to which the output files will be written. From the program manager, click on File (or type Alt-F). Then click Run

(or type R). Type the full command line, including the path to the directory (the browse capability may also be used). For example, type:

< C:\BORLANDC\BIN\LEWS.EXE INPUT.931 OUTPUTA.931 OUTPUTB1.931 >.

Click OK or hit the return key.

6.4. Modifying the Program

The program may also be run directly from Borland C++ 3.1 using the standard procedures in the environment. This procedure is recommended if changes to the source code are made. Other C compiler/environments may also be used.

The program is written with the following constraints, limitations, and options:

- Target is a Windows 3.0 and above executable file .
- Large memory model.
- Full floating point is enabled.
- Source code is Borland C++ without objects.

7. CODE AND DOCUMENTATION

This program is a C program that ran on a main frame computer or a workstation, and has been updated and translated into a Borland C program capable of running under Windows 3.1 or Windows 95 on a PC. Attention has been given to make the code self-documenting. The program was developed under a C++ environment; however, no object-oriented code is involved.

7.1 Purpose and General Description

The program is an implementation of the transfer function for an HF propagation model developed and described in Vogler and Hoffmeyer [3-5]. The program reads a data file containing parameters that are descriptive of the several paths or rays of an HF radio skywave channel. The program writes the complex Fourier coefficients that define the channel transfer function to a file that will be used in a wideband HF channel simulator in hardware. The model simulates the time-varying characteristics of the HF channel and, in particular, models delay spread, delay offset, Doppler frequency shift and spread, and the relationship between delay and Doppler. The transfer function simulates the influence of the HF channel medium upon the transmitted signal. In the future, the hardware simulator will also include models for noise and interference described in Lemmon and Behm [8,9].

The program is a windows executing code with command line

< LEWS FILE1 FILE2 FILE3 >

where LEWS.EXE is the executing code file, FILE1 is the input file, FILE 2 is the output file for input and computed parameters, and FILE3 is the output file for the complex Fourier coefficients defining the transfer function.

The source code for the program is in four files, which are compiled separately and linked together to form the executing code: LEW1.CPP, LEW2.CPP, LEW3.CPP, LEW4.CPP.

The documentation is organized as follows: The documentation for a function or a file's list of global variables, defines, and includes, etc. will be immediately followed by the listing of that function's code. When documentation refers to code, the following conventions are used: The names of constants and files are indicated by all caps as in MAXLAYERS; bold indicates both default and defined data types and functions, for example **comp_arrays**; and variables are in italics, e.g., *peak_amplitude*.

7.2. Project File LEW1.CPP

The file LEW1.CPP contains the main program.

Defines:

MAXLAYERS - the maximum number of reflecting layers (or reflected rays seen by the receiver) in the ionosphere that the program will handle.

Structures:

ray_path - structure that contains all input and computed variables characteristic of a path.

.path_Distance (D) - **float**, point-to-point distance, in kilometers, between the transmitter and the receiver, used to compute *tau_c*, read from input data.

.center_freq (f_c) - **float**, the center frequency, in Hertz, used to compute *tau_c*, read from input data.

.penetrate_freq (f_p) - **float**, the penetration frequency, in Hertz, used to compute *tau_c*, read from input data. The penetration frequency must be greater than the center frequency, since frequencies above this point will not be reflected. The penetration frequency is above the maximum useable frequency (MUF). If *penetrate_freq* is less than *center_freq*, an error message is generated.

.thick_scale (σ) - **float**, thickness scale factor, indicates the thickness of the reflecting layer in kilometers, used to compute *tau_c*, read from input data.

.maxD_hgt (h_0) - **float**, maximum electron density height in the layer, used to compute *tau_c*, read from input data.

.peak_amplitude (A) - **float**, the amplitude at the mean delay τ_c, read from input data.

.sigma_tau ($\sigma_\tau = \tau_U - \tau_L$) - **float**, the delay spread, read from input data.

.sigma_c ($\sigma_c = \tau_c - \tau_L$) - **float**, the partial delay spread, distance between τ_c, the point of peak amplitude and the lower value of the delay spread τ_L, read from input data. Essentially, σ_c is the rise time to τ_c of the impulse response with respect to τ_L.

.sigma_D (σ_D) - **float**, the Doppler spread half-width at τ_c, read from input data.

.fds (f_s) - **float**, the Doppler shift at τ_c, read from input data.

.fdl (f_{sL}) - **float**, the Doppler shift at τ_L, read from input data.

.tau_c (τ_c) - **double**, expected delay associated with the carrier frequency, calculated from input data in **big_c**.

.sigma_f (σ_f) - **double**, used to determine the Doppler shape, computed in **comp_arrays**.

.slp (b) - **double**, the slope or "slant" of the linear relationship between delay and Doppler, computed in **comp_arrays**.

.tau_L ($\tau_L = \tau_c - \sigma_c$) - **double**, left end of the delay spread at the half power point A_{fl}, computed in **comp_arrays**.

.tau_U ($\tau_U = \tau_L + \sigma_\tau$) - **double**, right end of the delay spread, at the half power point A_{fl}, computed in **comp_arrays**.

.tau_l (τ_l), **double**, location parameter for the delay function, computed in **little_el**. This is not necessarily the same for every layer nor is it necessarily at zero delay for this model.

.alpha (α) - **double**, shape parameter for the delay function, computed in **comp_arrays**.

.sigma_l ($\sigma_l = \tau_c - \tau_l$) - **double**, the rise time of the impulse response with respect to τ_l.

.lambda (λ) - **double**, exponential autocorrelation factor through time for random input streams.

compute - structure that contains all the variables specific to the computations or not specific to an individual path.

.layers - **integer**, the number of reflective ionospheric layers or the number of reflected rays, read from input data.

.slices - **integer**, number of time slices, read from input data.

.seed - **integer**, primary seed for random number generation, between 0 and 30,268 inclusive, read from input data file.

.delta_t (Δt)- **float**, the sampling interval or the real time step, read from input data.

.afl (A_{fl}) - **float**, the receiver threshold from input data.

.delta_tau ($\Delta\tau$) - **double**, delay step, computed in **comp_arrays**.

.big_el - **double**, the least or earliest of the *tau_l* values for the layers, determined in **comp_arrays**.

String type:

STRING - used for handling file names of input and output files.

```
#include <stdio.h>

#define MAXLAYERS 3

typedef struct ray_path
{
	float path_Distance, center_freq, penetrate_freq, thick_scale, maxD_hgt;
	float peak_amplitude, sigma_tau, sigma_c, sigma_D, fds, fdl;
	double tau_c, sigma_f, slp, tau_L, tau_U, tau_l, alpha, sigma_l, lambda;
};

typedef struct compute
{
	int layers, slices, seed;
	float delta_t, afl;
	double delta_tau, big_el;
};

typedef char *STRING;
```

7.2.1. Function **void main**

Description:

The **main** function calls functions **init** and **doit** and also handles file names input from the command line. **Main** is in the file LEW1.CPP.

Parameters:

argc - **integer**, indicates the number of parameters on the command line; for this program there are four.

argv - array of **pointers** to the string arrays of the parameters, *argv*[1] points to the input file, *argv*[2] points to the first output file, *argv*[3] points to the second output file *argv*[0] points to the file with the executable code. Example command line for execution: LEWS FILE1.DAT FILE2.DAT FILE3.DAT

Structures:

p - an array of size MAXLAYERS of **ray_path**.
c - a one element array of **compute**.

Functions called:

init - type **void**, initialization procedure, reads and checks data from input file, writes parameter output file. **Main** passes *argc* by value, and **STRING** *argv*[1], *c* - an array of **compute**, and *p* - an array of **ray_path** by reference. These arrays are initialized here. **Init** is in LEW2.CPP.

doit - type **void**, the procedure that accomplishes everything but input/output. **Main** passes *c*, the single element array of **compute**, and *p*, an array of **ray_path**, and **STRING**s *argv*[2] and *argv*[3], by reference. **Doit** is in LEW3.CPP.

```
void main(int argc, char *argv[])
{
                /* Function prototypes */

        extern void init(int, STRING, compute[], ray_path[]);
        extern void doit(compute[], ray_path[], STRING, STRING);

                /* Structures */

        struct ray_path p[MAXLAYERS];
        struct compute c[1];

                /* Code */

        init(argc, argv[1], c, p);

        doit(c, p, argv[2], argv[3]);

} /* End of main */
```

7.3. Project File LEW2.CPP

This file contains the code that reads and checks the input data and prints the output files.

Includes:

STDIO.H
STDLIB.H

Defines:

MAXLAYERS - the maximum number of reflecting layers (or reflected rays seen by the receiver) in the ionosphere that the program will handle.
DATA - the number of real data points in the output data streams. Two successive data points represent a complex number. The first is the real part and the second is the imaginary part.

Structures:

ray_path - structure that contains all input and computed variables characteristic of a path. The elements of **ray_path** are given on p. 28.
compute - structure that contains all the variables specific to the computations or not specific to an individual path. The elements of **compute** are given on p. 29.

String type:

STRING - used for handling file names of input and output files.

Files:

innyfile - **pointer** to the input file.
datyfile - **pointer** to the first output file. This file will contain all the input and computed parameters.
bigfile - **pointer** to the second output file. This file will contain the transfer function.

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

#define MAXLAYERS 3
#define DATA 4096

typedef struct ray_path
{
	float path_Distance, center_freq, penetrate_freq, thick_scale, maxD_hgt;
	float peak_amplitude, sigma_tau, sigma_c, sigma_D, fds, fdl;
	double tau_c, sigma_f, slp, tau_L, tau_U, tau_l, alpha, sigma_l, lambda;
};

typedef struct compute
{
	int layers, slices, seed;
	float delth, afl;
	double delt, big_el;
};

typedef char *STRING;

FILE *innyfile, *datyfile, *bigfile;
```

7.3.1. Function **void init**

Description:

This function is called by **main**. **Init** opens the input file and calls **input_data** to read all the data into the variables and to check the data for compatibility with the model and for preventing division by zero, logarithms of nonpositive numbers, etc. Note that *delta_t* is multiplied by 10^{-6}. This is done to place *delta_t* in the proper units of seconds. For consistency, all times are input in microseconds. **Init** finishes by closing the input file. **Init** is in the file LEW2.CPP.

Variables passed to **init**:

arg_num - **integer**, the number of arguments in the command line, 4.
inny - **STRING**, the name of the input file.
ci - **compute**, structure containing the computation parameters.
pi - **ray_path**, structure containing the path parameters.

Functions called:

input_data - reads data from the input file and checks some input data. **Init** passes *ci*, a single element array of **compute**, and *pi*, an array of **ray_path**, by reference. **Input_data** is in LEW2.CPP.
exit - termination library function, requires STDLIB.H.
fopen - library function that opens files, requires STDIO.H.
fclose - library function that closes files, requires STDIO.H.

```
void init(int arg_num, STRING inny, struct compute ci[1], struct ray_path pi[MAXLAYERS])
{
            /* Function Prototype */

        void input_data(compute[], ray_path[]);

            /* Code */

        if (arg_num !=4)
        {
            printf("\n Error in function init! \n");
            printf("\n Is command line correct?: lews infile outfile1 outfile2 \n");
            printf("\n Program will terminate! \n");
            exit(0);
        }

        if ((innyfile = fopen(inny,"r")) == NULL)
        {
            printf("\n Error in function init! \n");
            printf("\n Input file cannot be opened! \n");
            printf("\n Terminating program! \n");
            exit(0);
        }

        input_data(ci, pi);

        ci[0].delta_t *= 1.0E-6;

        if (fclose(innyfile) == EOF)
        {
            printf("\n Error in function init! \n");
            printf("\n Cannot close the input file! \n");
            printf("\n Terminating program \n");
            exit(0);
        }

        return;

} /* End of init */
```

7.3.2. Function **void input_data**

Description:

> **Input_data** reads data from the file specified by the second argument on the command line. This function also checks input data values to prevent data that would violate the model and that would cause run time errors due to division by zero, logarithms of nonpositive numbers, square roots of negative numbers, etc. **Input_data** also checks that the pseudorandom number generator seed is in the proper range. The input file has already been opened by **init** and the input file name is a global variable.

Variables passed to **input_data**:

> *cii* - **compute**, structure containing the computation parameters, an array of length 1.
> *pii* - **ray_path**, structure containing the path parameters, an array of length MAXLAYERS.

Local variable:

> *j* - **integer**, used to count through the layers of the input data file.

Functions called:

> **fscanf** - library function reads from files, requires STDIO.H.
> **printf** - library function prints to the executing window, requires STDIO.H.
> **exit** - library termination function, requires STDLIB.H.

```
void input_data(struct compute cii[1], struct ray_path pii[MAXLAYERS])
{
        /* Variable */

    int j;

        /* Code */

    fscanf(innyfile, "%d%f%f%d%d", &cii[0].slices, &cii[0].delta_t, &cii[0].afl,
        &cii[0].layers, &cii[0].seed);

    for (j = 0; j < cii[0].layers; j++)
    {
        fscanf(innyfile, "%f%f%f%f%f%f%f%f%f%f%f", &pii[j].path_Distance,
            &pii[j].center_freq, &pii[j].penetrate_freq, &pii[j].thick_scale,
            &pii[j].maxD_hgt, &pii[j].peak_amplitude, &pii[j].sigma_tau,
            &pii[j].sigma_c, &pii[j].sigma_D, &pii[j].fds, &pii[j].fdl);

        /* Input data checking */

        if (pii[j].peak_amplitude == 0.0)
        {
            printf("\n Error in function input_data!");
            printf("\n Division by zero coming!");
            printf("\n Peak_amplitude, A, must be greater than 0!");
            printf("\n Program will terminate!");
            printf("\n Correct the input file!");
            exit(0);
        }

        if ((pii[j].sigma_c == 0.0) || (pii[j].sigma_c >= (pii[j].sigma_tau / 2)))
        {
            printf("\n Error in function input_data!");
            printf("\n Division by zero warning!");
            printf("\n Sigma_c must be greater than 0 and less than half sigma_tau!");
            printf("\n Program will terminate!");
            printf("\n Correct the input file!");
            exit(0);
        }
```

```
            if ((cii[0].afl <= 0.0) || (cii[0].afl >= 1.0))
            {
                    printf("\n Error in function input_data!");
                    printf("\n Square root of a negative number warning!");
                    printf("\n Afl must be between 0 and 1!");
                    printf("\n Program will terminate!");
                    printf("\n Correct the input file!");
                    exit(0);
            }

            if (pii[j].penetrate_freq <= pii[j].center_freq)
            {
                    printf("\n Error in function input_data!");
                    printf("\n Penetration frequency must be greater than the");
                    printf(" center frequency!");
                    printf("\n Program will terminate!");
                    printf("\n Correct the input file!");
                    exit(0);
            }

            if ((cii[0].seed < 1) || (cii[0].seed > 30268))
            {
                    printf("\n Error in function input_data!");
                    printf("\n The seed must be between 1");
                    printf("\n and 30268 inclusive!");
                    printf("\n Program will terminate!");
                    printf("\n Correct the input file!");
                    exit(0);
            }

    }

    return;

} /* End of input_data */
```

7.3.3. Function **void out1**

Description:

This function is called by **comp_arrays** and prints the input and calculated parameters for the computing run and all input and computed parameters for each layer to the file specified by the third argument on the command line. Essentially, **out1** prints out all of the elements of the arrays *cdco* and *pdco*. **Out1** is in file LEW2.CPP.

Variables passed to **out1**:

cdco - structure of type **compute**.
pdco - structure of type **ray_path**.
daout1 - **STRING**, output file name.

Local variable:

i - **integer**, counts through the number of layers.

Functions called:

fopen - library function, opens file for printing, requires STDIO.H.
printf - library function, prints to screen, requires STDIO.H.
fprintf - library function, prints to file, requires STDIO.H.
fclose - library function, closes file, requires STDIO.H.

```
void out1(struct compute cdco[1], struct ray_path pdco[MAXLAYERS], STRING daout1)
{
	/* Variable */

	int i;

	/* Code */

	if ((datyfile = fopen(daout1,"w")) == NULL)
	{
		printf("\n Error in function out1!");
		printf("\n First output file cannot be opened!");
		printf("\n Terminating Program! \n");
		exit(0);
	}

	fprintf(datyfile,"\n Computing Parameters \n");
	fprintf(datyfile,"\nInput parameters\n");
	fprintf(datyfile,"\n slices = %d", cdco[0].slices);
	fprintf(datyfile,"\n delta_t = %f", cdco[0].delta_t);
	fprintf(datyfile,"\n afl = %f", cdco[0].afl);
	fprintf(datyfile,"\n layers = %d", cdco[0].layers);
	fprintf(datyfile,"\n seed = %d", cdco[0].seed);
	fprintf(datyfile,"\n\nComputed parameter\n");
	fprintf(datyfile,"\n delta_tau = %lf", cdco[0].delta_tau);
	fprintf(datyfile,"\n big_el = %lf", cdco[0].big_el);
	fprintf(datyfile,"\n\n Individual Path Data \n");

	for (i = 0; i < cdco[0].layers; i++)
	{
		fprintf(datyfile,"\n Layer %d \n", i + 1);
		fprintf(datyfile,"\n Input parameters \n");
		fprintf(datyfile,"\n path distance = %f", pdco[i].path_Distance);
		fprintf(datyfile,"\n center frequency = %f", pdco[i].center_freq);
		fprintf(datyfile,"\n penetration frequency = %f", pdco[i].penetrate_freq);
		fprintf(datyfile,"\n Thickness scale factor = %f", pdco[i].thick_scale);
		fprintf(datyfile,"\n Height of the maximum density = %f", pdco[i].maxD_hgt);
		fprintf(datyfile,"\n peak amplitude = %f", pdco[i].peak_amplitude);
		fprintf(datyfile,"\n sigma_tau = %f", pdco[i].sigma_tau);
		fprintf(datyfile,"\n sigma_c = %f", pdco[i].sigma_c);
		fprintf(datyfile,"\n sigma_D = %f", pdco[i].sigma_D);
		fprintf(datyfile,"\n fds = %f", pdco[i].fds);
		fprintf(datyfile,"\n fdl = %f \n", pdco[i].fdl);
```

```
            fprintf(datyfile,"\n Computed parameters \n");
            fprintf(datyfile,"\n tau_c = %lf", pdco[i].tau_c);
            fprintf(datyfile,"\n sigma_f = %lf", pdco[i].sigma_f);
            fprintf(datyfile,"\n slp = %lf", pdco[i].slp);
            fprintf(datyfile,"\n tau_L = %lf", pdco[i].tau_L);
            fprintf(datyfile,"\n tau_U = %lf", pdco[i].tau_U);
            fprintf(datyfile,"\n tau_l = %lf", pdco[i].tau_l);
            fprintf(datyfile,"\n alpha = %lf", pdco[i].alpha);
            fprintf(datyfile,"\n sigma_l = %lf", pdco[i].sigma_l);
            fprintf(datyfile,"\n lambda = %lf\n", pdco[i].lambda);

        } /* End of  i loop. */

        if (fclose(datyfile) == EOF)
        {
            printf("\n Error in function out1!");
            printf("\n Cannot close the first output file!");
            printf("\n Terminating program! \n");
            exit(0);
        }

} /* End of out1 */
```

7.3.4. Function **void outit**

Description:

This function prints the complex coefficients of the transfer function to the file specified by the fourth argument in the command line. The complex numbers are represented by a list of floats that are successively real and imaginary. The white space separator is the space character. **Outit** appends the list of complex coefficients for each time slice to the same file. Separate time slices are separated by an extra line. **Outit** is in file LEW2.CPP.

Parameters passed to **outit**:

dat - array of **float**, the complex number array.
daout2 - **STRING**, the large file name.

Local variables:

q - **integer**, counts through *dat*.

Functions called:

fopen - library function, opens file for printing, in this case for appending to file, requires STDIO.H.
printf - library function, prints to screen, requires STDIO.H.
fprintf - library function, prints to file, requires STDIO.H.
fclose - library function, closes file, requires STDIO.H.

```
void outit(float dat[2 * DATA], STRING daout2)
{
                /* Variable */

        int q;

                /* Code */

        if ((bigfile = fopen(daout2,"a")) == NULL)
        {
                printf("\n Error in function outit! \n");
                printf("\n Second output file cannot be opened! \n");
                printf("\n Terminating program! \n");
                exit (0);
        }

        for (q = 0; q < 2 * DATA; q++)
                fprintf(bigfile,"%lf ", dat[q]);

        fprintf(bigfile,"\n ");

        if (fclose(bigfile) == EOF)
        {
                printf("\n Error in function outit! \n");
                printf("\n Cannot close the second output file! \n");
                printf("\n Terminating program! \n");
                exit(0);
        }

} /* End of outit */
```

7.4. Project File LEW3.CPP

This file contains the major computation of the program except for the FFT and the impulse response code.

Includes:

STDIO.H - library file containing the input/output routines.
STDLIB.H - standard library file needed for exit function.
MATH.H - library file containing the math functions.

Defines:

MAXLAYERS - the maximum number of reflecting layers (or reflected rays seen by the receiver) in the ionosphere that the program will handle.
DATA - the number of real data points in the output data streams. Two successive data points represent a complex number. The first is the real part and the second is the imaginary part.
TWOPI - definition of $2\pi = 6.28318530717959$.
C - speed of light in km/μs, C = 0.299792458.

Structures:

ray_path - structure that contains all input and computed variables characteristic of a path. The elements of **ray_path** are given on p. 28.
compute - structure that contains all the variables specific to the computations or not specific to an individual path. The elements of **compute** are given on p. 29.

String type:

STRING - used for handling file names of input and output files.

Global variables:

cdat - array of **float** of size 2 × DATA, holds the impulse response data in the first half (up to DATA) for each layer at a particular time slice, the second half is zero padding. Later *cdat* holds the complex coefficients of the FFT for printing to the output files. This is usually a structure of real variables, but it is used in this program as a complex structure. A consecutive pair of floats in *cdat* represent a complex number, the first number of the pair (the even index) represents the real part and the second (the odd index) is the imaginary part.

seed1 - **long integer**, random number seed for the Wichmann-Hill generator, initialized in **comp_arrays**, calculated and updated in **ran1**.

seed2 - **long integer**, random number seed for the Wichmann-Hill generator, initialized in **comp_arrays**, calculated and updated in **ran1**.

seed3 - **long integer**, random number seed for the Wichmann-Hill generator, initialized in **comp_arrays**, calculated and updated in **ran1**.

seed4 - **long integer**, random number seed for L'Ecuyer's generator, initialized in **comp_arrays**, calculated and updated in **ran2**.

seed5 - **long integer**, random number seed for L'Ecuyer's generator, initialized in **comp_arrays**, calculated and updated in **ran2**.

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

#define MAXLAYERS 3
#define DATA 4096
#define TWOPI 6.28318530717959
#define C 0.299792458

typedef struct ray_path
{
	float path_Distance, center_freq, penetrate_freq, thick_scale, maxD_hgt;
	float peak_amplitude, sigma_tau, sigma_c, sigma_D, fds, fdl;
	double tau_c, sigma_f, slp, tau_L, tau_U, tau_l, alpha, sigma_l, lambda;
};

typedef struct compute
{
	int layers, slices, seed;
	float delta_t, afl;
	double delta_tau, big_el;
};

typedef char *STRING;

	/* Global Variables */

long seed1, seed2, seed3, seed4, seed5;

float cdat[2 * DATA];
```

7.4.1. Function **void doit**

Description:

This function is called by **main**. **Doit** first calls **comp_arrays** to compute all necessary values for the model. Then it calls **slicedo** to make the transfer function for each time slice. **Doit** is in file LEW3.CPP.

Variables passed to **doit**:

cd - array of **compute**, structure containing the computation parameters.
pd - array of **ray_path**, structure containing the path parameters.
daty - **STRING** containing the name of the first output file.
daty2 - **STRING** containing the name of the second output file.

Functions called by **doit**:

comp_arrays - computes all the derived parameters from the input parameters. **Doit** passes *cd,* a single element array of **compute** and *pd,* an array of **ray_path** both by reference and also passes *daty,* a **STRING** containing the name of the first output file from the command line. **Comp_arrays** is in file LEW3.CPP.

slicedo - computes the transfer function for each time slice. **Doit** passes *cd,* a single element array of **compute** and *pd*, an array of **ray_path** by reference, and also passes *daty2,* a **STRING** containing the name of the second output file from the command line. **Slicedo** is in file LEW3.CPP.

```
void doit(struct compute cd[1], struct ray_path pd[MAXLAYERS], STRING daty,
                STRING daty2)
{
                /* Function prototypes */

        void comp_arrays(compute[], ray_path[], STRING);
        void slicedo(compute[], ray_path[], STRING);

                /* Code */

        comp_arrays(cd, pd, daty);

        slicedo(cd, pd, daty2);

        return;

} /* End of doit */
```

7.4.2. Function **void comp_arrays**

Description:

Comp_arrays is called by **doit** and computes all the derived parameters that define each layer. The path parameters computed are *sigma_f*, *lambda*, *tau_U*, *tau_L*, *slp*, and *alpha* which are parameters in the structure **ray_path**. The computing parameters that are calculated are *big_el* and *delta_tau*. *Delta_tau* is the largest *tau_U* less *big_el* divided by one fourth DATA. **Comp_arrays** also initializes the random number seeds *seed1*, *seed2*, *seed3*, *seed4*, and *seed5* by invoking the individual generators making up the Wichmann-Hill and L'Ecuyer's composite generators, see functions **ran1** and **ran2** below. **Comp_arrays** calls **big_c** to compute the parameter *tau_c*. **Comp_arrays** calls **little_el** to compute the parameter *tau_l*. Finally, this function calls **out1** to output the input and computed parameters for each layer and for the input and calculated computation parameters to a file. **Comp_arrays** is in file LEW3.CPP.

Parameters passed to **comp_arrays**:

cdc - array of **compute**, structure containing the computation parameters.
pdc - array of **ray_path**, structure containing the path parameters.
datout1 - **STRING**, contains the name of the first output file.

Local variables:

sv - **double**, the ratio of the receiver threshold to the amplitude.
Z_l - **double**, convenient holder for computing *alpha*, corresponds to (23).

k - **integer**, indexes the skywave paths or the reflective layers.

Functions called:

sqrt - library function takes the square root of a real non-negative number, requires MATH.H.
log - library function takes the natural logarithm of a positive real number, requires MATH.H.
big_c - type **double**, computes the value *tau_c* for each path, passes *cdc* and *pdc*, returns computed value of *tau_c*. **Big_c** is in file LEW3.CPP.
little_el - type **double,** function that computes the value *tau_l* for each path, passes *tau_c*, *tau_L*, and *tau_U*, returns *tau_l*. **Little_el** is in file LEW3.CPP.
out1 - type void, outputs computing and path information for each layer to file, passes *cdc*, *pdc*, and *datout1*, the file name character string. **Out1** is in file LEW2.CPP.

```
void comp_arrays(struct compute cdc[1], struct ray_path pdc[MAXLAYERS], STRING
                datout1)
{
                /* Function Prototypes */

        double big_c(ray_path[], int);
        double little_el(float, double, double);
        extern void out1(compute[], ray_path[], STRING);

                /* Local Variables */

        int k;
        double sv, Z_l, big_U;

                /* Initialize random number generator seeds */

        seed1 = (171 * cdc[0].seed) % 30269;
        seed2 = (172 * seed1) % 30307;
        seed3 = (170 * seed2) % 30323;
        k = seed3 / 52774;
        seed5 = 40692 * (seed3 - k * 52774) - k * 3791;

        if (seed5 < 0)
                seed5 += 2147483399;

        k = seed5 / 53668;
        seed4 = 40014 * (seed5 - k * 53668) - k * 12211;

        if (seed4 < 0)
                seed4 += 2147483563;

                /* Compute the layer parameters */

        for (k = 0; k < cdc[0].layers; k++)
        {
                sv = cdc[0].afl;
                pdc[k].sigma_f = TWOPI * pdc[k].sigma_D * sv / sqrt(1.0 - sv * sv);
                pdc[k].lambda = exp(-cdc[0].delta_t * pdc[k].sigma_f);

                /* Note that sv can't equal 1 above */

                pdc[k].tau_c = big_c(pdc, k);
```

```
            pdc[k].slp = (pdc[k].fds - pdc[k].fdl) / pdc[k].sigma_c;
            pdc[k].tau_L = pdc[k].tau_c - pdc[k].sigma_c;
            pdc[k].tau_U = pdc[k].tau_L + pdc[k].sigma_tau;

            pdc[k].tau_l = little_el(pdc[k].tau_c, pdc[k].tau_L,
                                     pdc[k].tau_U);
            Z_l = (pdc[k].tau_L - pdc[k].tau_l) / (pdc[k].tau_c - pdc[k].tau_l);
            pdc[k].alpha = (log(sv)) / (log(Z_l) + 1 - Z_l);
            pdc[k].sigma_l = pdc[k].tau_c - pdc[k].tau_l;
        } /* End of k-loop */
        /* Compute big_el and delta_tau */

        big_U = 0.0;
        cdc[0].big_el = 100000.0;

        for (k = 0; k < cdc[0].layers; k++)
        {
            if (pdc[k].tau_U > big_U)
                big_U = pdc[k].tau_U;

            if (pdc[k].tau_l < cdc[0].big_el)
                cdc[0].big_el = pdc[k].tau_l;
        }

        if (cdc[0].big_el < 0.0)
            cdc[0].big_el = 0.0;

        cdc[0].delta_tau = (big_U - cdc[0].big_el) / (DATA / 4);

        out1(cdc, pdc, datout1);

        return;

    } /* End of comp_arrays */
```

7.4.3. Function **double big_c**

Description:

This function, called by **comp_arrays**, calculates and returns the value of *tau_c*, the mean delay for the center frequency, *center_freq*, from the equation

$$\left(c\frac{\tau_c}{2}\right)^2 = h_e^2 + \left(\frac{D}{2}\right)^2 \tag{24}$$

where c is the speed of light (C in the program), D is the point-to-point distance between transmitter and receiver (*path_Distance*), and h_e is the effective reflection height (*effective_height*) given in Budden [20, p. 156] by

$$h_e = \sigma \ln\left[\sqrt{\frac{f_p^2}{f_c^2} - 1}\ \sinh\left(\frac{h_0}{\sigma}\right) + \sqrt{\frac{\sinh^2\left(\frac{h_0}{\sigma}\right)}{\frac{f_p^2}{f_c^2} - 1} - 1}\right] \tag{25}$$

where σ is the thickness scale factor (*thick_scale*), h_0 is the height of the maximum electron density (*maxD_hgt*), f_p is the penetration frequency (*penetrate_freq*), and f_c is the center frequency (*center_freq*).

The first equation is an application of the Pythagorean Theorem while the second is an evaluated integral expression for the effective reflection height in a hyperbolic secant squared (sech^2) electron density model. The functions are ln, the natural logarithm, and sinh, the hyperbolic sine. **Big_c** is in file LEW3.CPP.

This method of determining τ_c differs from the method presented in Vogler and Hoffmeyer [5, p. 6]. There, the method used to find τ_c requires an iterative scheme. The method above is a direct calculation of the effective reflection height followed by a direct calculation for the mean delay of the center frequency.

Parameters passed to **big_c**:

pdcb - array of **ray_path**, structure containing the path parameters.
t - **integer**, the index of the current path.

Parameter returned to **comp_arrays**:

tau_c, τ_c - mean delay for the center frequency.

Local variables:

comp1 - **double**, ratio of the penetration frequency to the center frequency, convenient variable to avoid divisions.
comp2 - **double**, square root of the quantity (*comp1* × *comp1* - 1), convenient variable to avoid unnecessary calculations.
comp3 - **double**, holds the hyperbolic sine of the height of the maximum electron density divided by the thickness scale factor, convenient variable to avoid recalculation.
effective_height - **double**, the effective reflection height of the layer.

Functions called by **big_c**:

sqrt - library function takes the square root of a real non-negative number, requires MATH.H.
log - library function takes the natural logarithm of a positive real number, requires MATH.H.
sinh - library function takes the hyperbolic sine of a real number, requires MATH.H.

```
double big_c(struct ray_path pdcb[MAXLAYERS], int t)
{
                /* Variables */

        double comp1, comp2, comp3, effective_height;

                /* Code */

        comp1 = pdcb[t].penetrate_freq / pdcb[t].center_freq;
        comp2 = sqrt((comp1 * comp1) - 1);
        comp3 = sinh(pdcb[t].maxD_hgt / pdcb[t].thick_scale);
        effective_height = pdcb[t].thick_scale * log(sqrt(comp2) * comp3 +
                                        sqrt((1 / comp2) * comp3 * comp3 - 1));
        return((2 / C) * sqrt(effective_height * effective_height +
                                        pdcb[t].path_Distance * pdcb[t].path_Distance / 4));

} /* End of big_c */
```

7.4.4. Function **double little_el**

Description:

Little_el is called by **comp_arrays**. **Little_el** finds the value of *tau_l* that zeros the function

$$f(\tau_l) = \ln\left(\frac{\tau_L - \tau_l}{\tau_U - \tau_l}\right) + \frac{\tau_U - \tau_L}{\tau_c - \tau_l} \tag{26}$$

where ln is the natural logarithm function. Equation 26 is a form of the equation $\ln Z_L - Z_L = \ln Z_U - Z_U$ from Vogler and Hoffmeyer [5, p. 32], but (26) is written in terms of the delay values. Note that Z_L is given by (23) and that

$$Z_U = \frac{\tau_U - \tau_l}{\tau_c - \tau_l} . \tag{27}$$

The function (26) is derived by evaluating (3) at the delay parameters τ_L, τ_U, and τ_c, see Figure 1 . Since $P_n(\tau_L) = P_n(\tau_U)$,

$$\ln\left[\frac{P_n(\tau_L)}{P_n(\tau_c)}\right] = \ln\left[\frac{P_n(\tau_U)}{P_n(\tau_c)}\right] , \tag{28}$$

which when simplified yields (26). **Little_el** is in file LEW3.CPP.

A bisection method was used to compute τ_l because (26) is complex valued when τ_l is between τ_L and τ_U. For values of τ_l greater than τ_U, (26) increases without bound as τ_l approaches τ_U from the right; the function approaches 0 asymptotically from above as τ_l increases. Thus, there is no value in this range that will zero the function (26), see Figure 14. To the left of τ_L, the function decreases without bound as τ_l approaches τ_L; however, (26) approaches 0 asymptotically from above. This implies that (26) crosses the x-axis, achieves zero value, to the left of τ_L, see Figure 15. The variable τ_l is a location parameter for the impulse response. For delay greater than τ_l, the delay power is positive otherwise the delay power is zero.

A Newton-Raphson method can result in evaluations in the complex part, which fail, or can throw successive approximations into the areas where the function approaches zero unless the initial guess is lucky. A bisection method was used to avoid those problems by keeping successive approximations in the "good" zone.

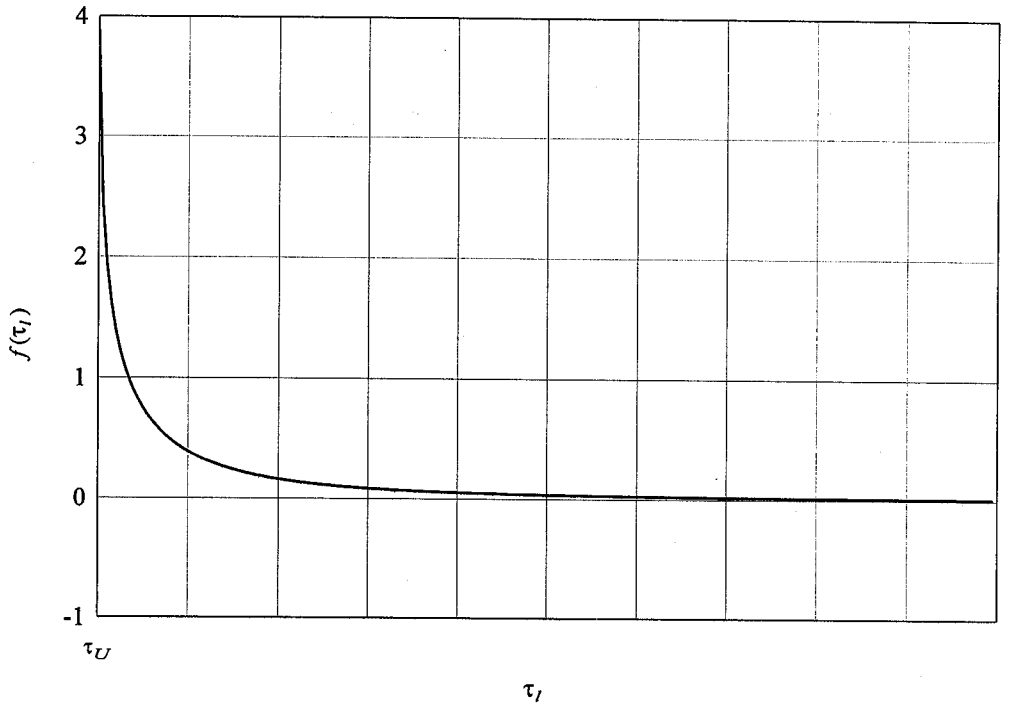

Figure 14. Horizontal and vertical asymptotes of function (26) above τ_U.

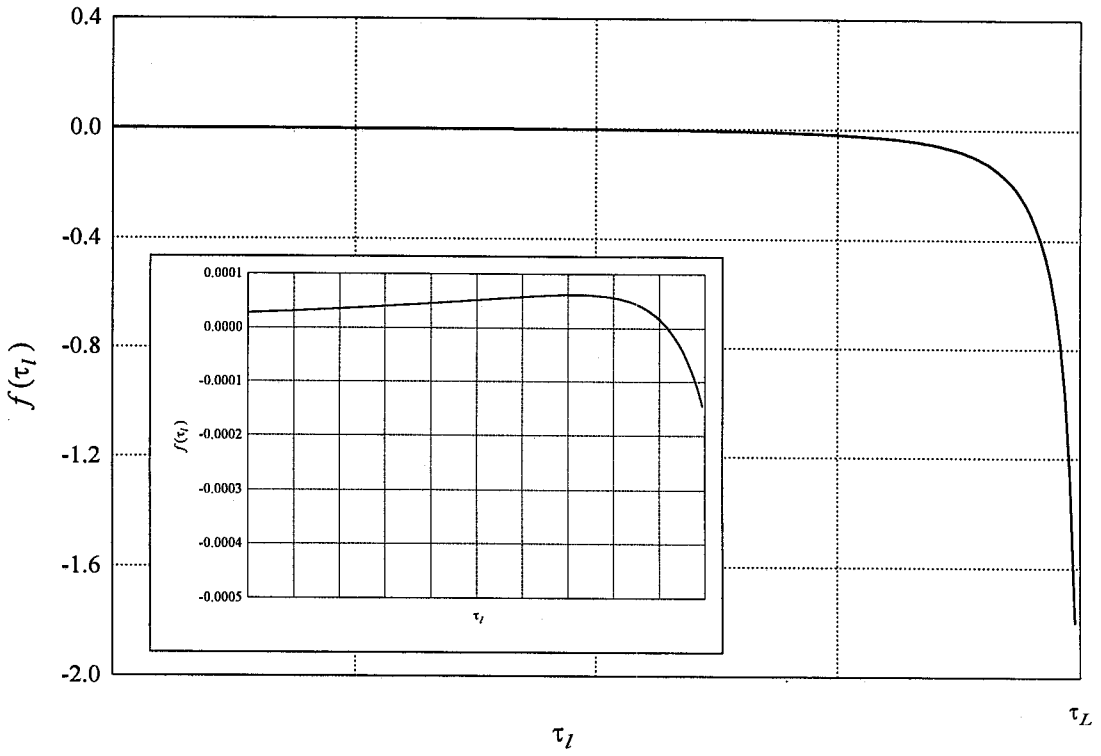

Figure 15. Horizontal and vertical asymptotes of function (26) below τ_L. Inset: Zero crossing and function maximum illustrated at finer scale.

Other fixed point methods, such as the secant method, may also be used. Initial approximations are found by searching for two points between τ_L and

$$\frac{\tau_L \tau_U - \tau_c^2}{\tau_L + \tau_U - 2\tau_c}, \tag{29}$$

which is the point at which (26) achieves its maximum value to the left of τ_L. The evaluation of (26) at the two points sought will have opposite sign. The bisection algorithm takes these two values and determines a value that zeros (26) with a tolerance of less than 0.0000001. The bisection method used is an implementation of the one given in Burden and Faires [22, pp. 28-33].

Parameters passed to **little_el**:

tau_c - **float**, delay associated with the carrier frequency for the current layer.
tau_L ($\tau_L = \tau_c - \sigma_c$) - **double**.
tau_U ($\tau_U = \tau_L + \sigma_\tau$) - **double**.

Parameter returned to **comp_arrays**:

searchpoint = *tau_l* - location parameter for the delay function, represents a location parameter for the delay shape. The values of *tau_l* for the different layers are not necessarily the same.

Local variables:

searchpoint - **double**, holds various approximations for the variable *tau_l*. The final approximation of *tau_l* is returned to **comp_arrays**.
negative_arg - **double**, holds approximation in bisection method that causes the function value to be negative, not necessarily itself negative.
positive_arg - **double**, holds approximation in bisection method that causes the function value to be positive, not necessarily itself positive.
holdval - **double**, convenient temporary variable.
halfdif - **double**, holds value in bisection algorithm that is used to keep from doing more than one division.

m - **integer**, counts passes through loops in **little_el**, compared to preset values to provide convenient stopping criteria for infinite loops, for example.

Functions called by **little_el**:

funvalue - computes and returns the value of (26). **Little_el** passes *tau_L*, *tau_U*, *tau_c*, and *searchpoint* (or *positive_arg*) to evaluate (26) at *searchpoint,* the current approximation to *tau_l*.

pow - library function returns x to the power of y, x^y, where x and y are type **double**. Requires MATH.H.

```
double little_el(float tau_c, double tau_L, double tau_U)
{
        /* Function prototype */

    double funvalue(double, double, float, double);

        /* Variables */

    int m;
    double searchpoint, negative_arg, positive_arg, holdval, halfdif;

        /* Code */

    positive_arg = (tau_L * tau_U - tau_c * tau_c) / (tau_L + tau_U - 2 * tau_c);

    searchpoint = (positive_arg + tau_L) / 2;

    /* Get two estimates for bisection algorithm */

    if ((holdval = funvalue(tau_L, tau_U, tau_c, searchpoint)) < 0)
        negative_arg = searchpoint;
    else
        if (holdval > 0)
        {
            positive_arg = searchpoint;

            while (1)
            {
                searchpoint = (searchpoint + tau_L) / 2;

                if ((holdval = funvalue(tau_L, tau_U, tau_c, searchpoint)) > 0)
                    positive_arg = searchpoint;
                else
                    if (holdval < 0)
                    {
                        negative_arg = searchpoint;
                        break;
                    }
                    else
                        return(searchpoint);  /* Holdval = 0 */
            }
        }
        else return(searchpoint);  /* Holdval = 0 */
```

```
        /* bisection algorithm with two appropriate estimates */

    for (m = 1; m <= 100; m++)
    {
        halfdif = (negative_arg - positive_arg) / 2;
        searchpoint = positive_arg + halfdif;

        if (((holdval = funvalue(tau_L, tau_U, tau_c, searchpoint)) == 0) ||
                    (halfdif < 0.0000001))
            return(searchpoint);

        if ((holdval * funvalue(tau_L, tau_U, tau_c, positive_arg)) > 0)
            positive_arg = searchpoint;
        else
            negative_arg = searchpoint;
    }
    printf("\n Error in function little_el!");
    printf("\n Bisection for tau_l failed after 100 iterations!");
    printf("\n Stopping program!");
    exit(0);

} /* End of little_el */
```

7.4.5. Function **double funvalue**

Description:

Funvalue is called by **little_el** and returns the value (**double**) of (26). **Funvalue** is located in file LEW3.CPP.

Parameters passed to **funvalue**:

c - **float**, corresponds to *tau_c*, passed by value.

L - **double**, corresponds to *tau_L*, passed by value.
U - **double**, corresponds to *tau_U*, passed by value.
point - **double**, corresponds to *tau_l*, passed by value.

Parameter returned to **little_el**:

The value of (26) evaluated at *point* - **double**.

Functions called by **funvalue**:

log - library function returns the natural logarithm of a real positive number, requires MATH.H.

```
double funvalue(double L, double U, float c, double point)
{
                /* Code */

        return(log((L - point) / (U - point)) + ((U - L) / (c - point)));

}   /* End of funvalue */
```

7.4.6. Function **void slicedo**

Description:

This function is called by **doit**. **Slicedo** starts by initializing a block of dynamic memory for the exponential autocorrelated random number streams that are used to compute the impulse response. There are DATA floats for each layer in the allocated block, enough for the computation of the nonzero padded portion of the impulse response array. **Slicedo** calls **rvgexp** to compute these number streams. For each time slice, **slicedo** calls **imp** to compute the impulse response by computing and superimposing the impulse responses for each reflective layer, then calls **little_four** to compute the FFT of the impulse response, then calls **outit** to print the complex Fourier coefficients to file. Finally, **slicedo** frees the allocated dynamic memory block. **Slicedo** is in file LEW3.CPP.

Parameters passed to **slicedo**:

cds - array of **compute**, structure containing the computation parameters.
pds - array of **ray_path**, structure containing the path parameters.
datout2 - **STRING**, contains the output file name, passed to **outit**.

Global variables used:

cdat - array of **float** of size 2 × DATA, holds the impulse response data in the first half (up to DATA) for each layer at a particular time slice, second half is zero padded, later holds the complex coefficients of the FFT for printing to the output files, although this is a structure of real variables it is used in this program as a complex structure. A consecutive pair of floats in this array represents a complex number, the first number of the pair (the even index) represents the real part and the second (the odd index) is the imaginary part. *Cdat* is initialized here in **slicedo**.

Local variables:

timex - **double**, value of time for each time slice.

n - **integer**, counts the time slices.
o - **integer**, counts through initialization of *cdat*, reused to count through layers in determining impulse response for a time slice.

front - **pointer** to a **float**, points to the first position of the dynamic block of floats, this is the location of the block.
starter - **pointer** to a **float**, points to the next starting place of the dynamic block, the place where the values for the impulse response for the next layer start.
nextest - pointer to **float**, points to the next starting place, is returned by **rvgexp** by reference.

Functions called by **slicedo**:

rvgexp - **pointer** to a **float**, creates and updates the random number array that has exponential autocorrelation in time. **Slicedo** passes *n*, *lambda,* and *starter*. **Rgvexp** returns a pointer to float, the next starting place in the dynamic array for subsequent layers. **Rgvexp** is in the file LEW3.CPP.

malloc - library function that allocates dynamic memory for the large random number arrays, needs STDLIB.H.

printf - prints a file, needs STDIO.H.

exit - library termination function, needs STDLIB.H.

free - library function that unallocates dynamic memory, needs STDLIB.H.

outit - prints the coefficients of the FFT to file, **slicedo** passes the complex array of coefficients, *cdat*, and the name of the output file, *datout2*. **Outit** is in the file LEW2.CPP.

imp - type void, creates the superimposed impulse response layer by layer for each time slice. **Slicedo** passes the data array *cdat*, *cds*, *pds*, *starter*, *timex*, and *n*. **Imp** is in the file LEW4.CPP.

little_four - type void, computes the FFT on the impulse response. **Slicedo** passes *cdat*, DATA (a global constant), and a +1 (indicates the direction of the FFT). **Little_four** returns the data array, *cdat*, which now contains the complex Fourier coefficients. **Little_four** is in the file LEW4.CPP.

```
void slicedo(struct compute cds[1], struct ray_path pds[MAXLAYERS], STRING datout2)
{
        /* Function prototypes */

    float * rvgexp(int, double, float *);
    extern void little_four(float[], int, int);
    extern void outit(float[], STRING);
    extern void imp(float[], ray_path[], compute[], float *, double, int);

        /* Variables */

    int n, o;
    float *front, *starter, *nextest;
    double timex;

        /* Code */

    if ((front = (float*) malloc(cds[0].layers * DATA * sizeof(float))) == NULL)
    {
        printf("\n Error in function slicedo!");
        printf("\n Not enough memory to allocate!");
        printf("\n Terminating program!");
        exit(0);
    }

    for (n = 0; n < cds[0].slices; n++)
    {
        for (o = 0; o < 2 * DATA; o++)
            cdat[o] = 0.0;

        /* Provides initialization for each slice and 0-padding */

        timex = n * cds[0].delta_t;

        for (o = 0; o < cds[0].layers; o++)
        {
            if (o == 0)
                starter = front;
            else
                starter = nextest;

            nextest = rvgexp(n, pds[o].lambda, starter);
```

```
                imp(cdat, pds, cds, starter, timex, o);

            } /*  End of layers loop */

            /* fast fourier xform */

            little_four(cdat - 1, DATA, 1);

            outit(cdat, datout2);

        } /* End of slices loop */

        free(front);
        return;

} /* End of slicedo */
```

7.4.7. Function **pointer** to **float** **rvgexp**

Description:

Rvgexp is called by **slicedo**. This function initializes and updates a dynamically allocated random floating point array that represents a complex array, which has exponential autocorrelation from time slice to time slice, see (19) in Section 3. The memory block for the array is initialized and freed in **slicedo**. The array is just large enough for the required data points in the impulse response function, **imp**, for each time slice. The array is updated for the next time slice with the current value for each float in the array used to compute the next value at that position. The autocorrelation quality is in the time direction, not in the delay direction. Rvgexp is located in file LEW3.CPP.

Parameters passed to **rvgexp**:

slice - **integer**, the current time slice.
lambduh - **double**, corresponds to *lambda* in slicedo, the exponential autocorrelation factor.
start - **pointer** to array of **floats**, points to the beginning of the dynamic block containing the array.

Local variables:

normal1 - **double**, variable from a normally distributed random number stream with 0 mean and variance equal to one. The notation N(0,1) is usually used for such a distribution.
normal2 - **double**, variable from an N(0,1) stream independent of *normal1*. These two N(0,1) distributions (*normal1* and *normal2*) are said to be independent and identically distributed (IID).
mult - **double**, used to hold computed factor 1 - λ to avoid recomputing.

p - **integer**, used to count through creation of array.

current - **pointer** to a **float**, points to the current position of the dynamic block. Used to run through the array.

Variable returned to **slicedo**:

current - pointer to **float**, points to the next position in the dynamically allocated array.

Function called by **rvgexp**:

get_2i_normals - produces two independent normal random variates. Nothing is passed by **rvgexp**, but two IID N(0,1) variates are returned by reference. **Get_2i_normals** is in the file LEW3.CPP.

```
float * rvgexp(int slice, double lambduh, float *start)
{
                /* Function prototype */

        void get_2i_normals(double *, double *);

                /* Variables */

        int p;
        float *current;
        double normal1, normal2, mult, squared;

                /* Code */

        current = start;
        mult = 1 - lambduh;

        if (slice == 0)
        {
                for (p = 0; p < DATA; p += 2)
                {
                        get_2i_normals(&normal1, &normal2);
                        *current = normal1 * mult;
                        current++;
                        *current = normal2 * mult;
                        current++;
                }
        }
        else
        {
                for (p = 0; p < DATA; p += 2)
                {
                        get_2i_normals(&normal1, &normal2);
                        *current = normal1 + lambduh * (*current - normal1);
                        current++;
                        *current = normal2 + lambduh *(*current - normal2);
                        current++;
                }
        }

    return (current);

} /* End of rvgexp */
```

7.4.8. Function **void get_2i_normals**

Description:

This function is an improvement of the Box-Muller algorithm that produces two independent standard normal variates. The algorithm is called the polar method and is an acceptance/rejection method given in Law and Kelton [10, pp. 490-492] with further reference to Marsaglia and Bray [23]. Atkinson and Pearce [24], and Ahrens and Dieter [25] report a 9 - 31% increase in speed over the standard Box-Muller method, see Box and Muller [26]. The algorithm requires two independent uniform pseudorandom number streams. To ensure this independence, two different random number generators are used. There are faster algorithms, see, for example, Kinderman and Ramage [27], but the polar method produces a pair of independent variates as required by the modulation function (19). **Get_2i_normals** is in file LEW3.CPP.

Parameters returned to **rvgexp**:

normy1 - **double**, N(0,1) distributed random variate passed back by reference.
normy2 - **double**, N(0,1) distributed random variate passed back by reference.

Local variables:

v1 - **double**.
v2 - **double**.
w - **double**.
y - **double**.

Functions called by **get_2i_normals**:

ran1 - Wichmann-Hill composite uniform random number generator, returns a **double** on the interval (0,1). **Ran1** is located in file LEW3.CPP.
ran2 - L'Ecuyer's composite uniform random number generator, returns a **double** on the interval (0,1). **Ran2** is located in file LEW3.CPP.
sqrt - returns the square root of a non-negative real number, must include MATH.H.
log - returns the natural logarithm of a positive real number, must include MATH.H.

```
void get_2i_normals(double *normy1, double *normy2)

        /* The polar method improvement of the Box-Muller method of producing two
         * independent N(0,1) variates.
         * Note:        Two random number generators are used to ensure that the two
         *              required random number streams are independent. */

{
                /* Function prototypes */

        double ran1();
        double ran2();

                /* Variables */

        double v1, v2, w, y;

                /* Code */

        do
        {
                v1 = 2.0 * ran1() - 1.0;
                v2 = 2.0 * ran2() - 1.0;
                w = v1 * v1 + v2 * v2;
        }
        while (w > 1.0);

        y = sqrt(-2.0 * log(w) / w);

        *normy1 = v1 * y;
        *normy2 = v2 * y;

        return;

} /* End of get_2i_normals */
```

7.4.9. Function **double ran1**

Description:

Ran1 is called by **get_2i_normals**. This function is an implementation of the Wichmann-Hill uniform random number generator. It requires three integer seeds. The random number generator is a composite of the three generators

$$\begin{aligned} U_{i+1} &= 171\, U_i \,(\text{mod } 30{,}269)\ , \\ V_{i+1} &= 172\, V_i \,(\text{mod } 30{,}307)\ , \text{ and} \\ W_{i+1} &= 170\, W_i \,(\text{mod } 30{,}323)\ , \end{aligned} \tag{30}$$

given in Jeruchim, Balaban, and Shanmugan [28, pp 273-275] with further reference to Coates, Janacek, and Lever [29], and Wichmann and Hill [30]. The period of the composite random stream is of the order 10^{13}. The function returns a **double** on the interval (0,1). The seeds are global variables and are initialized in **comp_arrays**. The seeds are updated with each call to **ran1**. **Ran1** is located in file LEW3.CPP.

Global variables used:

seed1 - **integer**, U in (30) above.
seed2 - **integer**, V in (30) above.
seed3 - **integer**, W in (30) above.

Variable returned to **get_2i_normals**:

Returns the fractional part of the sum of U, V, and W divided by 30,269, 30,307, and 30,323, respectively.

Functions called by **ran1**:

fmod - returns the fractional remainder of one **double** type divided by another. Need to include MATH.H.

```
double ran1()

        /* An implementation of the Wichmann-Hill composite algorithm
         * Note:        seed1, seed2, and seed3 are global variables initialized
         *              globally and retain value with each call */

{
                /* Code */

        seed1 = (171 * seed1) % 30269;
        seed2 = (172 * seed2) % 30307;
        seed3 = (170 * seed3) % 30323;

        return(fmod((double)seed1 / 30269.0 + (double)seed2 / 30307.0 +
                        (double)seed3 / 30323.0, 1.0));

} /* End of ran1 */
```

7.4.10. Function **double ran2**

Description:

This function is an implementation of L'Ecuyer's composite uniform random number generator. **Ran2** is located in LEW3.CPP. It requires two long integer seeds. The random number generator is a composite of the two generators

$$\begin{aligned} U_{i+1} &= 40{,}014\, U_i \,(\text{mod } 2{,}147{,}483{,}563) \text{ and} \\ V_{i+1} &= 40{,}692\, V_i \,(\text{mod } 2{,}147{,}483{,}399)\ , \end{aligned} \tag{31}$$

that, in turn, are inputs to the generator

$$W_{i+1} = U_{i+1} + V_{i+1} \ (\text{mod } 2{,}147{,}483{,}563)\ . \tag{32}$$

The algorithm is given in Brately, Fox, and Schrage, [21, pp. 204, 332] with further reference to L'Ecuyer [31]. The period is of the order 10^{18}. The seeds are global variables and are initialized in **comp_arrays**. The function returns a **double** on (0,1). The seeds are updated with each call.

Global variables used:

seed4 - **long integer**, U in (31) and (32) above.
seed5 - **long integer**, V in (31) and (32) above.

Variable returned to **get_2i_normals**:

w / 2,147,483,563 - the uniform random variate on the interval (0,1).

Local variables:

w - **long integer** - W in (32) above.
k - **long integer** - useful variable to avoid repeated divisions.

```
double ran2()

        /* An implementation of L'Ecuyer's composite algorithm
         * Note:        seed4 and seed5 are global variables initialized globally and
         *              retain value with each call */

{
                /* Variables */

        long int w, k;

                /* Code */

        k = seed4 / 53668;
        seed4 = 40014 * (seed4 - k * 53668) - k * 12211;

        if (seed4 < 0)
                seed4 += 2147483563;

        k = seed5 / 52774;
        seed5 = 40692 * (seed5 - k * 52774) - k * 3791;

        if (seed5 < 0)
                seed5 += 2147483399;

        w = seed5 - seed4;

        if (w <= 0)
                w += 2147483562;

        return((double)w * 4.656613057392e-10);

} /* End of ran2 */
```

7.5 Project File LEW4.CPP

This file contains code to calculate the impulse response and the FFT.

Includes:

STDIO.H - library file with the standard input/output routines.
STDLIB.H - standard library file needed for exit function.
MATH.H - library file with math functions.

Defines:

MAXLAYERS - the maximum number of reflecting layers (or reflected rays seen by the receiver) in the ionosphere that the program will handle.
DATA - the number of real data points in the output data streams. Two successive data points represent a complex number. The first is the real part and the second is the imaginary part.
TWOPI - definition of $2\pi = 6.28318530717959$.
SWAP - a data switching macro used by the FFT algorithm, **little_four**.

Structures:

ray_path - structure that contains all input and computed variables characteristic of a path. The elements of **ray_path** are given on p. 28.
compute - structure that contains all the variables specific to the computations or not specific to an individual path. The elements of **compute** are given on p. 29.

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

#define MAXLAYERS 3
#define DATA 4096
#define TWOPI 6.28318530717959

#define SWAP(a,b)    tempr = (a); \
                     (a) = (b); \
                     (b) = tempr

typedef struct ray_path
{
        float path_Distance, center_freq, penetrate_freq, thick_scale, maxD_hgt;
        float peak_amplitude, sigma_tau, sigma_c, sigma_D, fds, fdl;
        double tau_c, sigma_f, slp, tau_L, tau_U, tau_l, alpha, sigma_l, lambda;
};

typedef struct compute
{
        int layers, slices, seed;
        float delta_t, afl;
        double delta_tau, big_el;
};
```

7.5.1. Function **void little_four**

Description:

This function computes the complex FFT of a complex array and is called by **slicedo**. This is an implementation of the FFT found in Press et al. [32, pp. 404-414]. A single data array is passed in where the elements are alternating real and imaginary parts of complex numbers. This data array is replaced (passed back) with the complex coefficients of its Fourier transform. This function begins counting at 1, so the input array must be decremented by one in the function call, e.g., **little_four**(data - 1, 4096, 1). **Little_four** is contained in file LEW4.CPP.

Parameters passed to **little_four**:

data - array of **float**, size 2 × *nn*, passed by reference, contains the complex impulse response array.
nn - **integer**, must be a power of two, indicates size of the complex array.
isign - **integer**, a flag that indicates the desired direction of the FFT, 1 means the normal FFT will be run while -1 means the inverse FFT will be run. Normalization for the inverse case is not done within **little_four**.

Parameters returned by **little_four**:

data - array of **float**, size 2 × *nn*, contains the complex Fourier coefficients, returned by reference.

Macros used:

SWAP - this macro simply swaps the value of two variables. Used by **little_four** in data rearrangement.

Functions called:

sin - library function returns the sine of a real number. Needs MATH.H.

Code listing:

See Press et al. [32, pp. 404-414].

7.5.2. Function **void imp**

Description:

This function computes the impulse response for each time slice by computing and superimposing the impulse responses for each layer. See (2) and (14). **Imp** is located in LEW4.CPP.

Parameters passed to **imp**:

datb - array of **float** of size 2 × DATA, corresponds to *cdat* in slicedo.
pdsi - array of **ray_path** of size *layers*.
cdsi - array of **compute** of size 1.

start - pointer to **float**, current position in the random number array.

timexx - **double**, current slice time.

oo - **integer**, current time slice index.

Local variables:

tau_k - **double**, delay step.
gag - **double**, difference between *tau_k* and *tau_l* for current layer.
gg - **double**, *gag* divided by *sigma_l*.
exparg - **double**, argument of the combined exponential term.
squirt - **double**, result of the square root term.
sine - **double**, sine of *exparg*.
cosine - **double**, cosine of *exparg*.

xkm - **float**, real part of random variable term.
ykm - **float**, imaginary part of random variable term.

now - pointer to **float**, points to current term in the random number array, incremented within **imp**.

r - **integer** counter.

Functions called:

sqrt - library function that takes the square root of a non-negative real number, needs MATH.H.
exp - library function that raises *e* to a real number, requires MATH.H.
log - library function that takes the natural logarithm of a non-negative real number, needs MATH.H.
sin - library function that takes the sine of a real number, needs MATH.H.
cos - library function that takes the cosine of a real number, needs MATH.H.

```
void imp(float datb[2 * DATA], struct ray_path pdsi[MAXLAYERS],
                        struct compute cdsi[1], float *start, double timexx, int oo)
{
                /* Variables */

        int r;
        float xkm, ykm;
        float *now;
        double tau_k, gg, exparg, squirt, sine, cosine, gag;

                /* Code */

        now = start;
        tau_k = cdsi[0].big_el;

        for (r = 1; r < DATA / 2; r++)
        {
                tau_k += cdsi[0].delta_tau;

                if ((gag = tau_k - pdsi[oo].tau_l) <= 0)
                        continue;

                        /* Bypass since log(gg) in squirt computation below will be
                         * undefined when tau_k is less than or equal to .tau_l */
                else
                        gg = gag / pdsi[oo].sigma_l;

                exparg = TWOPI * (timexx * (pdsi[oo].fds + pdsi[oo].slp *
                                        (tau_k - pdsi[oo].tau_c)));
                squirt = sqrt((pdsi[oo].peak_amplitude * exp(pdsi[oo].alpha *
                                (log(gg) - gg + 1))));
                sine = sin(exparg);
                cosine = cos(exparg);
                xkm = *now;
                now++;
                ykm = *now;
                now++;
                datb[r + r] += (float)(squirt * (cosine * xkm - sine * ykm));
                datb[r + r + 1] += (float)(squirt * (cosine * ykm + sine * xkm));
        }
        return;

} /* End of imp */
```

7.6. Additional Information

This section contains additional information that may be of use to understanding, executing, manipulating, and changing the code.

7.6.1 Library Functions Used

exit - terminates the program, used to terminate for improper input arguments, and for unsuccessful input or output file openings and closings, must include STDLIB.H.
log - returns the natural logarithm of a positive real number, must include MATH.H.
sqrt - returns the square root of a non-negative real number, must include MATH.H.
fmod - returns the fractional remainder of one positive **double** divided by another, must include MATH.H.
pow - library function returns x to the power of y where x and y are type **double**, needs MATH.H.
cos - returns trigonometric cosine of a real number, must include MATH.H.
sin - returns trigonometric sine of a real number, must include MATH.H.
exp - returns e raised to the real argument, must include MATH.H.
fopen - opens files, requires STDIO.H.
fclose - closes files, requires STDIO.H.
fscanf - library function reads from files, requires STDIO.H.
printf - library function reads to files, requires STDIO.H.
fprintf - prints to file, requires STDIO.H.
malloc - allocates memory for the large data arrays, needs STDLIB.H.
free - unallocates memory block, needs STDLIB.H.
sinh - library function takes the hyperbolic sine of a real number, requires MATH.H.

7.6.2. Input Data File Format

This program reads input data from an ASCII file in the following order with white space between values.

slices (**integer**)
delta_t (**float**) [Δt]{microseconds}
afl (**float**)
layers (**integer** between 1 and 3 inclusive)
seed (**integer** between 1 and 30268 inclusive)

[For each layer]

path_Distance (**float**) [D] {kilometers}
center_freq (**float**) [f_c] {megaHertz}
penetrate_freq (**float**) [f_p] {megaHertz}
thick_scale (**float**) [σ] {kilometers}
maxD_hgt (**float**) [h_o] {kilometers}
peak_amplitude (**float**) [A]
sigma_tau (**float**) [σ_τ] {microseconds}
sigma_c (**float**) [σ_c] {microseconds}
sigma_D (**float**) [σ_D] {Hertz}
fsl (**float**) [f_s] {Hertz}
fdl (**float**) [f_{sL}] {Hertz}

[] Indicates variable symbol.
{} Indicates units.

7.6.3. Function Calling Hierarchy

This section contains a modified function calling tree that indicates the functions, including library functions that each function calls. A function calls the functions indented once immediately below it. The hierarchy also indicates the function that calls particular functions. For example, function **main** calls the functions **init** and **doit**. **Little_el** calls the function **funvalue** and the library function **pow**. **Ran1** is called by the function **get_2i_normals** and the library function **log** is called by the functions **comp_arrays**, **big_c**, **funvalue**, **get_2i_normals**, **slicedo**, and **imp**.

```
main
    init
        input_data
            {exit, fscanf, printf}
        {exit, fopen, fclose}
    doit
        comp_arrays
            {sqrt, log}
            big_c
                {sqrt, sinh, log}
            little_el
                funvalue
                    {log}
                {pow}
            {exit}
            out1
        slicedo
            rvgexp
                get_2i_normals
                    ran1
                        {fmod}
                    ran2
                    {sqrt, log}
            {cos, sin, log, sqrt}
            outit
            imp
                {sqrt, exp, log, sin, cos}
            little_four
                {sin}
```

{} - indicates library functions.

8. SUMMARY

This report presents and documents a computer program that calculates the transfer function for a wideband HF channel simulator with a 1-MHz bandwidth. The program outputs the transfer function which is intended as input to the hardware HF channel simulator under development. The program's input is a set of parameters that characterizes the conditions of a particular skywave path for the HF channel being modeled.

Mathematical descriptions of the model are provided in the report. Code listing of the program is provided and complete documentation and a user's guide are included.

Graphical verification of the model is provided through plots of scattering functions for each path presented. Scattering functions relate several important parameters of the model, including delay spread, delay offset, Doppler shift, and Doppler spread. Scattering functions are also a popular method of presenting measured channels. The scattering functions allow visual verification of several of the input and computed parameters of the program. The verification indicates that the model agrees well with the expected representation of the input parameters. The exception is that an unexpected constant offset in overall Doppler frequency was found that appears to be related to filtering parameters for the stochastic input of the model. The offset is related to the sampling rate and to the Doppler spread input to the program.

A solution to the offset problem is to carefully design the digital filters for each channel rather than to use a general filter for all situations. This could be a considerable effort; however, the result would be a set of standard channels that could be used for laboratory testing and evaluation of radio systems and radio networks. In particular, the channels could be used to prove the efficacy and reliability of communications systems in support of NS/EP goals and missions in the laboratory and without the need for expensive and time-consuming over-the-air (OTA) testing.

Even with the unexpected shift problem, the verification results are in good enough agreement with the model to indicate that the hardware simulator will still be an excellent engineering tool. As such, it could be used in research and development of radio systems and networks.

9. REFERENCES

[1] J. Hoffmeyer and M. Nesenbergs, "Wideband HF modeling and simulation," NTIA Report 87-221, July 1987. (NTIS Order No. PB88-116116/AS).

[2] L. Vogler, J. Hoffmeyer, J. Lemmon, and M. Nesenbergs, "Progress and remaining issues in the development of the wideband HF channel model and simulator," *NATO AGARD Conference Proceedings, Propagation Effects, and Circuit Performance of Modern Millitary Radio Systems with Particular Emphasis on Those Employing Bandspreading*, Paris, France, October,1988, Paper No. 6.

[3] L.E. Vogler and J.A. Hoffmeyer, "A new approach to HF channel modeling and simulation - Part I: Deterministic model," NTIA Report 88-240, December, 1988. (NTIS Order No. PB89-203962/AS).

[4] L.E. Vogler and J.A. Hoffmeyer, "A new approach to HF channel modeling and simulation - Part II: Stochastic model," NTIA Report 90-255, February, 1990. (NTIS Order No. PB90-200338/AS).

[5] L.E. Vogler and J.A. Hoffmeyer, "A new approach to HF channel modeling and simulation - Part III: Transfer function," NTIA Report 92-284, March, 1992.

[6] C. Redding and D. Weddle, "Adaptive HF radio test using real-time channel evaluation systems," *Proceedings IEEE Military Communications Conference*, October,1994.

[7] C.C. Watterson, J.R. Juroshek, and W.D. Bensema, "Experimental confirmation of an HF channel model," *IEEE Transactions on Communication Technology*, COM-18, pp. 792-803, 1970.

[8] J.J. Lemmon and C.J. Behm, "Wideband HF noise/interference modeling Part I: First order statistics," NTIA Report 91-277, May, 1991.

[9] J.J. Lemmon and C.J. Behm, "Wideband HF noise/interference modeling Part II: Higher order statistics," NTIA Report 93-293, January, 1993.

[10] A.M. Law and W.D. Kelton, *Simulation Modeling and Analysis*, 2nd Ed, NY: McGraw-Hill, 1991.

[11] L.S. Wagner, J.A. Goldstein, W.D. Meyers, and P.A. Bello, "The HF skywave channel: Measured scattering functions for midlatitude and auroral channels and estimates for short-term wideband HF rake modem performance," *1989 Military Communications Conference*, Boston, 3, pp. 48.1-48.2.

[12] K. Davies, *Ionospheric Radio Propagation*, National Bureau of Standards Monograph 80, 1965.

[13] J.G. Proakis, *Digital Communications*, NY: McGraw Hill, 1983.

[14] J.F. Mastrangelo, J.J. Lemon, L.E. Vogler, J.A. Hoffmeyer, L.E. Pratt, and C.J. Behm, "A new wideband high frequency channel simulation system," *IEEE Transactions on Communications*, 45, No. 1, pp. 26-34, 1997.

[15] A.B. Baggeroer, "Sonar signal processing," *Applications of Digital Signal Processing*, Alan V. Oppenheim Ed., Englewood Cliffs, N.J.: Prentice Hall, 1978, pp. 331-437.

[16] D. Middleton, "A statistical theory of reverberation and similar first ordered scattered fields, Parts I and II," *IEEE Transactions of Information Theory*, 13, pp. 372-414, 1967.

[17] L.S. Wagner, J.A. Goldstein, and W.D. Meyers, "Wideband probing of the transauroral channel: solar minimum," *Radio Science*, 23, No. 4, pp. 555-568, 1988.

[18] R.P. Basler, P.B. Bentley, G.H. Price, R.T. Tsunoda, and T.L. Wong, "Ionospheric distortion of HF signals," Technical Report DNA-TR-87-246, SRI Project 8675, SRI International, Menlo Park, CA. 94025, 1987.

[19] L.S. Wagner, and J.A. Goldstein, "Response of the high latitude HF skywave channel to an isolated magnetic disturbance," *IEE Conference Publication No. 339*, pp. 233-237, 1991.

[20] K.G. Budden, *Radio Waves in the Ionosphere*, NY: Cambridge University Press, 1961.

[21] P. Brately, B.L. Fox, and L.E. Schrage, *A Guide to Simulation*, 2nd Ed., NY: Springer Verlag, 1987.

[22] R.L. Burden and J.D. Faires, *Numerical Analysis* 3rd Ed., Boston: Prindle, Weber, and Schmidt, 1985.

[23] G. Marsaglia and T.A. Bray, "A convenient method for generating normal variables," *SIAM Review*, 6, pp. 260-264, 1964.

[24] A.C. Atkinson and M.C. Pearce, "The computer generation of beta, gamma, and normal random variables," *Journal of the Royal Statistical Society*, A139, pp. 431-448, 1976.

[25] J.H. Ahrens and U. Dieter, "Computer methods for sampling from the exponential and normal distributions," *Communications of the Association of Computing Machinery*, 15, pp. 873- 882, 1972.

[26] G.E.P. Box and M.E. Muller, "A note on the generation of random normal deviates," *Annals of Mathematical Statistics*, 29, pp. 610-611, 1958.

[27] A.J. Kinderman and J.G. Ramage, "Computer generation of normal random variables," *Journal of the American Statistical Association*, 71, pp. 893-896, 1976.

[28] M.C. Jeruchim, P. Balaban, and K.S. Shanmugan, *Simulation of Communications Systems*, NY: Plenum Press, 1992.

[29] R.F. Coates, G.F. Janacek, and K.V. Lever, "Monte Carlo simulation and random number generation," *IEEE Journal on Selected Areas of Communication*, 6, pp. 58-66, 1988.

[30] B.A. Wichmann and I.D. Hill, "An efficient and portable pseudo random number generator," *Applied Statistics,* AS-183, pp. 188-190, 1982.

[31] P. L'Ecuyer, "Efficient and portable combined pseudorandom number generators," *Communications of the Association of Computing Machinery*, 31, pp. 742-749 and 774, 1988.

[32] W.H. Press, B.P. Flannery, S.P. Teukolsky, and W.T. Vetterling, *Numerical Recipes in C: The Art of Scientific Computing*, NY: Cambridge University Press, 1988.

APPENDIX: SCATTERING FUNCTION PROGRAM

This program is essentially the same as the transfer function program and hence is presented with little additional documentation. The program produces the scattering function rather than a transfer function. The scattering function can be graphically presented with the proper plotting software.

The major difference is that this program does the FFT of the impulse response in the time direction at each increment of delay. For that reason, the dynamic memory arrangement is no longer necessary and the global array *cdat* carries all the information at every step. Thus, *cdat* is initialized and filled with the uniform random number streams that are used to compute the N(0,1) random streams which replace the uniform streams in *cdat*. This is repeated in **rvgexp**, where the random streams are replaced with the exponentially autocorrelated sequences in *cdat*. Finally, *cdat* contains the impulse response in the time direction. This is done for each reflecting layer with the array *fcdat* accumulating the superimposed impulse responses. The FFT is performed on *fcdat*. The magnitude of *fcdat* is placed in the array *outdat* and is printed to file by **outit**.

The function **outit** normalizes the data by setting maximum and minimum values of amplitude and converts the values to dB. The minimum and maximum are set arbitrarily. If the amplitude values are desired, the code which is commented out should replace the for loop immediately following.

```
                /* specavg1.cpp */
#include <stdio.h>
#define MAXLAYERS 3
typedef struct ray_path
{
        float path_Distance, center_freq, penetrate_freq, thick_scale, maxD_hgt;
        float peak_amplitude, sigma_tau, sigma_c, sigma_D, fds, fdl;
        double tau_c, sigma_f, slp, tau_L, tau_U, tau_l, alpha, sigma_l, lambda;
};
typedef struct compute
{
        int layers, slices, seed;
        float delta_t, afl;
        double delta_tau, big_el;
};
typedef char *STRING;

void main(int argc, char *argv[])
{
                /* Function prototypes */
        extern void init(int, STRING, compute[], ray_path[]);
        extern void doit(compute[], ray_path[], STRING, STRING);
                /* Structures */
        struct ray_path p[MAXLAYERS];
        struct compute c[1];
                /* Code */
        init(argc, argv[1], c, p);
        doit(c, p, argv[2], argv[3]);
 } /* End of main */
```

```
/* specavg2.cpp */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define MAXLAYERS 3
#define DATA 2048
typedef struct ray_path
{
	float path_Distance, center_freq, penetrate_freq, thick_scale, maxD_hgt;
	float peak_amplitude, sigma_tau, sigma_c, sigma_D, fds, fdl;
	double tau_c, sigma_f, slp, tau_L, tau_U, tau_l, alpha, sigma_l, lambda;
};
typedef struct compute
{
	int layers, slices, seed;
	float delta_t, afl;
	double delta_tau, big_el;
};
typedef char *STRING;
FILE *innyfile, *datyfile, *bigfile;

void init(int arg_num, STRING inny, struct compute ci[1], struct ray_path pi[MAXLAYERS])
{
	/* Function prototype */
	void input_data(compute[], ray_path[]);
	/* Code */
	if (arg_num !=4)
	{
		printf("\n Error in function init! \n");
		printf("\n Is command line correct?: lewsblue infile outfile1 outfile2 \n");
		printf("\n Program will terminate! \n");
		exit(0);
	}
	if ((innyfile = fopen(inny,"r")) == NULL)
	{
		printf("\n Error in function init! \n");
		printf("\n Input file cannot be opened! \n");
		printf("\n Terminating program! \n");
		exit(0);
	}
	input_data(ci, pi);
	ci[0].delta_t *= 1.0E-6;
	if (fclose(innyfile) == EOF)
	{
		printf("\n Error in function init! \n");
		printf("\n Cannot close the input file! \n");
		printf("\n Program will terminate! \n");
```

```
                exit(0);
        }
        return;
} /* End of init */

void input_data(struct compute cii[1], struct ray_path pii[MAXLAYERS])
{
                /* Variable */
        int j;
                /* Code */
        fscanf(innyfile, "%d%f%f%d%d", &cii[0].slices, &cii[0].delta_t, &cii[0].afl, &cii[0].layers,
                                &cii[0].seed);
        for (j = 0; j < cii[0].layers; j++)
        {
                fscanf(innyfile, "%f%f%f%f%f%f%f%f%f%f%f", &pii[j].path_Distance,
                &pii[j].center_freq, &pii[j].penetrate_freq, &pii[j].thick_scale, &pii[j].maxD_hgt,
                        &pii[j].peak_amplitude, &pii[j].sigma_tau, &pii[j].sigma_c, &pii[j].sigma_D,
                        &pii[j].fds, &pii[j].fdl);
                        /* Input data checking */
                if (pii[j].peak_amplitude == 0.0)
                {
                        printf("\n Error in function input_data!");
                        printf("\n Division by zero coming!");
                        printf("\n Peak_amplitude, A, must be greater than 0!");
                        printf("\n Program will terminate!");
                        printf("\n Correct the input file!");
                        exit(0);
                }
                if (pii[j].sigma_c == 0.0)
                {
                        printf("\n Error in function input_data!");
                        printf("\n Division by zero warning!");
                        printf("\n Sigma_c must be greater than 0!");
                        printf("\n Program will terminate!");
                        printf("\n Correct the input file!");
                        exit(0);
                }
                if ((cii[0].afl <= 0.0) || (cii[0].afl >= 1.0))
                {
                        printf("\n Error in function input_data!");
                        printf("\n Square root of a negative number warning!");
                        printf("\n Afl must be between 0 and 1! ");
                        printf("\n Program will terminate!");
                        printf("\n Correct the input file!");
                        exit(0);
                }
                if (pii[j].penetrate_freq <= pii[j].center_freq)
```

```
                {
                        printf("\n Error in function input_data!");
                        printf("\n Penetration frequency must be greater than the");
                        printf(" center frequency!");
                        printf("\n Program will terminate!");
                        printf("\n Correct the input file!");
                        exit(0);
                }
                if ((cii[0].seed < 1) || (cii[0].seed > 30268))
                {
                        printf("\n Error in function input_data!");
                        printf("\n The seed must be between 1");
                        printf("\n and 30268 inclusive!");
                        printf("\n Program will terminate!");
                        printf("\n Correct the input file!");
                        exit(0);
                }
        }
        return;
} /* End of input_data */

void out1(struct compute cdco[1], struct ray_path pdco[MAXLAYERS], STRING daout1)
{
                /* Variable */
        int i;
                /* Code */
        if ((datyfile = fopen(daout1,"w")) == NULL)
        {
                printf("\n Error in function out1! \n");
                printf("\n First output file cannot be opened \n");
                printf("\n Terminating program! \n");
                exit(0);
        }
        fprintf(datyfile,"\n Computing Parameters \n");
        fprintf(datyfile,"\nInput parameters\n");
        fprintf(datyfile,"\n slices = %d", cdco[0].slices);
        fprintf(datyfile,"\n delta_t = %f", cdco[0].delta_t);
        fprintf(datyfile,"\n afl = %f", cdco[0].afl);
        fprintf(datyfile,"\n layers = %d", cdco[0].layers);
        fprintf(datyfile,"\n seed = %d", cdco[0].seed);
        fprintf(datyfile,"\n\nComputed parameter\n");
        fprintf(datyfile,"\n delta_tau = %lf", cdco[0].delta_tau);
        fprintf(datyfile,"\n big_el = %lf", cdco[0].big_el);
        fprintf(datyfile,"\n\n Individual Path Data \n");
        for (i = 0; i < cdco[0].layers; i++)
        {
                fprintf(datyfile,"\n Layer %d \n", i + 1);
```

```
            fprintf(datyfile,"\n Input parameters \n");
            fprintf(datyfile,"\n path distance = %f", pdco[i].path_Distance);
            fprintf(datyfile,"\n center frequency = %f", pdco[i].center_freq);
            fprintf(datyfile,"\n penetration frequency = %f", pdco[i].penetrate_freq);
            fprintf(datyfile,"\n Thickness scale factor = %f", pdco[i].thick_scale);
            fprintf(datyfile,"\n Height of the maximum density = %f", pdco[i].maxD_hgt);
            fprintf(datyfile,"\n peak amplitude = %f", pdco[i].peak_amplitude);
            fprintf(datyfile,"\n sigma_tau = %f", pdco[i].sigma_tau);
            fprintf(datyfile,"\n sigma_c = %f", pdco[i].sigma_c);
            fprintf(datyfile,"\n sigma_D = %f", pdco[i].sigma_D);
            fprintf(datyfile,"\n fds = %f", pdco[i].fds);
            fprintf(datyfile,"\n fdl = %f \n", pdco[i].fdl);
            fprintf(datyfile,"\n Computed parameters \n");
            fprintf(datyfile,"\n tau_c = %lf", pdco[i].tau_c);
            fprintf(datyfile,"\n sigma_f = %lf", pdco[i].sigma_f);
            fprintf(datyfile,"\n slp = %lf", pdco[i].slp);
            fprintf(datyfile,"\n tau_L = %lf", pdco[i].tau_L);
            fprintf(datyfile,"\n tau_U = %lf", pdco[i].tau_U);
            fprintf(datyfile,"\n tau_l = %lf", pdco[i].tau_l);
            fprintf(datyfile,"\n alpha = %lf", pdco[i].alpha);
            fprintf(datyfile,"\n sigma_l = %lf", pdco[i].sigma_l);
            fprintf(datyfile,"\n lambda = %lf\n", pdco[i].lambda);
        } /* End of i loop */
        if (fclose(datyfile) == EOF)
        {
            printf("\n Error in function out1! \n");
            printf("\n Cannot close the first output file! \n");
            printf("\n Terminating program! \n");
            exit(0);
        }
} /* End of out1 */

void outit(float fdat[DATA], STRING daout2)
{
            /* Variables */
        int q;
        float max, quot, min;
            /* Code */
        if ((bigfile = fopen(daout2,"a")) == NULL)
        {
            printf("\n Error in function outit! \n");
            printf("\n Second output file cannot be opened! \n");
            printf("\n Terminating program! \n");
            exit (0);
        }
        max = 0.2979;
        min = -20.0;
```

```
//      for (q = (DATA / 2) - 1; q >= 0; q--)
//              fprintf(bigfile, "%lf ", fdat[q]);
        for (q = (DATA / 2) - 1; q >= 0; q--)
        {
                if ((quot = fdat[q] / max) > 0.01)
                        fprintf(bigfile, "%lf ", 10 * log10(quot));
                else
                        fprintf(bigfile, "%lf ", min);
        }
        fprintf(bigfile,"\n");
        if (fclose(bigfile) == EOF)
        {
                printf("\n Error in function outit! \n");
                printf("\n Cannot close the second output file! \n");
                printf("\n Terminating program! \n");
                exit(0);
        }
} /* End of outit */
```

```
                /* specavg3.cpp */
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <alloc.h>
#define MAXLAYERS 3
#define DATA 2048
#define TWOPI 6.28318530717959
#define C 0.299792458
typedef struct ray_path
{
        float path_Distance, center_freq, penetrate_freq, thick_scale, maxD_hgt;
        float peak_amplitude, sigma_tau, sigma_c, sigma_D, fds, fdl;
        double tau_c, sigma_f, slp, tau_L, tau_U, tau_l, alpha, sigma_l, lambda;
};
typedef struct compute
{
        int layers, slices, seed;
        float delta_t, afl;
        double delta_tau, big_el;
};
typedef char *STRING;
                /* Global Variables */
        long seed1, seed2, seed3, seed4, seed5;
        float cdat[DATA];  /* DATA is 4 x slices value */
        float fcdat[DATA];
        float outdat[DATA / 2];

void doit(struct compute cd[1], struct ray_path pd[MAXLAYERS], STRING daty, STRING daty2)
{
                /* Function prototypes */
        void comp_arrays(compute[], ray_path[], STRING);
        void slicedo(compute[], ray_path[], STRING);
                /* Code */
        comp_arrays(cd, pd, daty);
        slicedo(cd, pd, daty2);
        return;
} /* End of doit */

void comp_arrays(struct compute cdc[1], struct ray_path pdc[MAXLAYERS], STRING datout1)
{
                /* Function prototypes */
        double big_c(ray_path[], int);
        double little_el(float, double, double);
        extern void out1(compute[], ray_path[], STRING);
                /* Variables */
        int k, jump;
```

```
        double sv, Z_l, big_U;
                /* Initialize random number generator seeds */
        seed1 = (171 * cdc[0].seed) % 30269;
        seed2 = (172 * seed1) % 30307;
        seed3 = (170 * seed2) % 30323;
        k = seed3 / 52774;
        seed5 = 40692 * (seed3 - k * 52774) - k * 3791;
        if (seed5 < 0)
                seed5 += 2147483399;
        k = seed5 / 53668;
        seed4 = 40014 * (seed5 - k * 53668) - k * 12211;
        if (seed4 < 0)
                seed4 += 2147483563;
                /* Compute the layer parameters */
        for (k = 0; k < cdc[0].layers; k++)
        {
                sv = cdc[0].afl;
                pdc[k].sigma_f = TWOPI * pdc[k].sigma_D * sv / sqrt(1.0 - sv * sv);
                pdc[k].lambda = exp(-cdc[0].delta_t * pdc[k].sigma_f);
                /* Note that sv can't equal 1 */
                pdc[k].tau_c = big_c(pdc, k);
                pdc[k].slp = (pdc[k].fds - pdc[k].fdl) / pdc[k].sigma_c;
                pdc[k].tau_L = pdc[k].tau_c - pdc[k].sigma_c;
                pdc[k].tau_U = pdc[k].tau_L + pdc[k].sigma_tau;
                pdc[k].tau_l = little_el(pdc[k].tau_c, pdc[k].tau_L, pdc[k].tau_U);
                Z_l = (pdc[k].tau_L - pdc[k].tau_l) / (pdc[k].tau_c - pdc[k].tau_l);
                pdc[k].alpha = (log(sv)) / (log(Z_l) + 1 - Z_l);
                pdc[k].sigma_l = pdc[k].tau_c - pdc[k].tau_l;
        }   /* End of k-loop */
        /* Compute big_el and delta_tau */
        big_U = 0.0;
        cdc[0].big_el = 100000.0;
        for (k = 0; k < cdc[0].layers; k++)
        {
                if (pdc[k].tau_U > big_U)
                        big_U = pdc[k].tau_U;
                if (pdc[k].tau_l < cdc[0].big_el)
                        cdc[0].big_el = pdc[k].tau_l;
        }
        if (cdc[0].big_el < 0.0)
                cdc[0].big_el = 0.0;
        cdc[0].delta_tau = (big_U - cdc[0].big_el) / (DATA / 4);
        out1(cdc, pdc, datout1);
        return;
} /* End of comp_arrays */

double big_c(struct ray_path pdcb[MAXLAYERS], int t)
```

```
{
        /* Variables */
    double comp1, comp2, comp3, effective_height;
        /* Code */
    comp1 = pdcb[t].penetrate_freq / pdcb[t].center_freq;
    comp2 = sqrt((comp1 * comp1) - 1);
    comp3 = sinh(pdcb[t].maxD_hgt / pdcb[t].thick_scale);
    effective_height = pdcb[t].thick_scale * log(sqrt(comp2) * comp3 +
                                sqrt((1 / comp2) * comp3 * comp3 - 1));
    return((2 / C) * sqrt(effective_height * effective_height +
                                pdcb[t].path_Distance * pdcb[t].path_Distance / 4));
} /* End of big_c */

double little_el(float tau_c, double tau_L, double tau_U)
{
        /* Function Prototype */
    double funvalue(double, double, float, double);
        /* Variables */
    int m;
    double searchpoint, negative_arg, positive_arg, holdval, halfdif;
        /* Code */
    searchpoint = tau_L - 1;
    /* Get two estimates for bisection algorithm */
    if ((holdval = funvalue(tau_L, tau_U, tau_c, searchpoint)) < 0)
        negative_arg = searchpoint;
    else
        if (holdval > 0)
        {
            positive_arg = searchpoint;
            while (1)
            {
                searchpoint = (searchpoint + tau_L) / 2;
                if ((holdval = funvalue(tau_L, tau_U, tau_c, searchpoint)) > 0)
                    positive_arg = searchpoint;
                else
                    if (holdval < 0)
                    {
                        negative_arg = searchpoint;
                        break;
                    }
                    else
                        return(searchpoint);
            }
        }
        else return(searchpoint);  /* Holdval = 0 */
    /* bisection algorithm with two appropriate estimates */
    for (m = 1; m <= 100; m++)
```

```
        {
                halfdif = (negative_arg - positive_arg) / 2;
                searchpoint = positive_arg + halfdif;
                if (((holdval = funvalue(tau_L, tau_U, tau_c, searchpoint)) == 0) ||
                                                        (halfdif < 0.0000001))
                        return(searchpoint);
                if ((holdval * funvalue(tau_L, tau_U, tau_c, positive_arg)) > 0)
                        positive_arg = searchpoint;
                else
                        negative_arg = searchpoint;
        }
                printf("\n Error in function little_el!");
                printf("\n Bisection for tau_l failed after 100 iterations!");
                printf("\n Stopping program!");
                exit(0);
} /* End of little_el */

double funvalue(double L, double U, float c, double point)
{
                /* Code */
        return(log((L - point) / (U - point)) + ((U - L) / (c - point)));
}   /* End of funvalue */

void slicedo(struct compute cds[1], struct ray_path pds[MAXLAYERS], STRING datout2)
{
                /* Function prototypes */
        void rvgexp(double);
        extern void little_four(float[], int, int);
        extern void outit(float[], STRING);
        extern void imp(float[], ray_path[], compute[], int, double, double);
                /* Variables */
        int m, n, o, p;
        double gag, gg, tau_k;
                /* Code */
        tau_k = cds[0].big_el;
        for (n = 0; n < 1024; n++)   /* Up to 1024 counting delay*/
        {
                for (o = 0; o < DATA / 2; o++)
                        outdat[o] = 0.0;
                tau_k += cds[0].delta_tau;
                for (m = 0; m < 160; m++)
                {
                        for (o = 0; o < DATA; o++)
                                fcdat[o] = 0.0;
                        for (o = 0; o < cds[0].layers; o++)
                        {
                                for (p = 0; p < DATA; p++)
```

```
                        cdat[p] = 0.0;
                if ((gag = tau_k - pds[o].tau_l) <= 0.0)
                        continue;
                else
                {
                        rvgexp(pds[o].lambda);
                        gg = gag / pds[o].sigma_l;
                        imp(cdat, pds, cds, o, tau_k, gg);
                }
                for (p = 0; p < DATA; p++)
                        fcdat[p] += cdat[p];
        } /* End of layers loop */
                little_four(fcdat - 1, DATA / 2, 1);
        for (o = 0; o < DATA / 2; o++)
                outdat[o] += sqrt(fcdat[o + o] * fcdat[o + o] + fcdat[o + o + 1] *
                                                        fcdat[o + o + 1]);
    } /* End of m-loop */
    for (m = 0; m < DATA / 2; m++)
            outdat[o] /= 160.0;
    outit(outdat, datout2);
  } /* End of delay slice loop */
  return;
} /* End of slicedo */

void rvgexp(double lambduh)
{
        /* Function prototype */
  void get_2i_normals();
        /* Variables */
  int s;
  double mult;
        /* Code */
  mult = 1 - lambduh;
  get_2i_normals();
  cdat[0] = cdat[0] * mult;
  cdat[1] = cdat[1] * mult;
  for (s = 2; s < DATA / 2; s += 2)
  {
        cdat[s] = cdat[s] + lambduh * (cdat[s - 2] - cdat[s]);
        cdat[s + 1] = cdat[s + 1] + lambduh * (cdat[s - 1] - cdat[s + 1]) ;
  }
  return;
} /* End of rvgexp */

void get_2i_normals()
  /* The polar method improvement of the Box-Muller method of producing two
   * independent N(0,1) variates.
```

```
 * Note:        Two random number generators are used to ensure that the two
 *              required random number streams really are independent. */
{
        /* Function prototypes */
    void ran1();
    void ran2();
        /* Variables */
    int s;
    double arg, mult, v1, v2, w, y;
        /* Code */
    ran1();
    ran2();
    for (s = 0; s < DATA / 2; s +=2)
    {
        mult = sqrt(-2 * log(cdat[s]));
        arg = TWOPI * cdat[s + 1];
        cdat[s] = mult * cos(arg);
        cdat[s + 1] = mult * sin(arg);
    }
    return;
} /* End of get_2i_normals */

void ran1()
    /* An implementation of the Wichmann-Hill composite algorithm
     * Note:        seed1, seed2, and seed3 are global variables initialized
     *              globally and retain value with each call */
{

        /* Variable */
    int s;
        /* Code */
    for (s = 0; s < DATA / 2; s += 2)
    {
        seed1 = (171 * seed1) % 30269;
        seed2 = (172 * seed2) % 30307;
        seed3 = (170 * seed3) % 30323;
        cdat[s] = (float)fmod((double)seed1 / 30269.0 + (double)seed2 / 30307.0 +
                                (double)seed3 / 30323.0, 1.0);
    }
    return;
} /* End of ran1 */

void ran2()
    /* An implementation of L'Ecuyer's composite algorithm
     * Note:        seed4 and seed5 are global variables initialized globally and
     *              retain value with each call */
{
```

```
            tdat[r] = datb[r];
        no_squirt = (pdsi[oo].peak_amplitude * exp(pdsi[oo].alpha * (log(ggg) - ggg + 1)));
        expargfx = TWOPI * (pdsi[oo].fds + pdsi[oo].slp * (tau_kk - pdsi[oo].tau_c));
        timexx = 0.0;
        for (r = 0; r < cdsi[0].slices; r++)
        {
            timexx += cdsi[0].delta_t;
            exparg = timexx * expargfx;
            sine = sin(exparg);
            cosine = cos(exparg);
            sind = (no_squirt / cdsi[0].slices) * sine;
            cosind = (no_squirt / cdsi[0].slices) * cosine;
            summer = 0.0;
            for(m = 0; m < cdsi[0].slices - r; m++)
                summer = summer + (tdat[m + m] * tdat[m + m + r + r]) + (tdat[m + m +
        1] *                                    tdat[m + m + r + r + 1]);
            datb[r + r] = (float)(cosind * summer / (cdsi[0].slices - r));
            datb[r + r + 1] = (float)(sind * summer / (cdsi[0].slices - r));
        }
        return;
} /* End of imp */
```

FORM NTIA-29
(4-80)

U.S. DEPARTMENT OF COMMERCE
NAT'L. TELECOMMUNICATIONS AND INFORMATION ADMINISTRATION

BIBLIOGRAPHIC DATA SHEET

	1. PUBLICATION NO.	2. Gov't Accession No.	3. Recipient's Accession No.
	NTIA Report 98-348		

4. TITLE AND SUBTITLE
Software Implementation of a Wideband HF Channel Transfer Function

5. Publication Date
April 1998

6. Performing Organization Code
NTIA/ITS.N1

7. AUTHOR(S)
David A. Sutherland, Jr.

8. PERFORMING ORGANIZATION NAME AND ADDRESS
National Telecommunications and Information Admin.
Institute for Telecommunication Sciences
325 Broadway
Boulder, CO 80303

9. Project/Task/Work Unit No.

10. Contract/Grant No.

11. Sponsoring Organization Name and Address
National Communications System
NCS-N6
701 South Court House Road
Arlington, VA 22204-2198

12. Type of Report and Period Covered

13.

14. SUPPLEMENTARY NOTES

15. ABSTRACT *(A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here.)*

This Report presents an analytic model implemented as the computer program of the transfer function of a wideband HF Channel model for use in a hardware simulator. The transfer function is the basic input to the hardware simulator. The mathematical basis of the program and the propagation model is presented. Parameters that characterize the skywave paths of a particular HF ionospheric condition are inputs to the program. The program code is listed and documentation is provided. Graphical verification using spectrally averaged scattering functions indicates that the transfer function program performs well and should find use as both an engineering tool and as the basis for a new standard propagation model.

16. Key Words *(Alphabetical order, separated by semicolons)*

channel transfer function; HF channel model; HF propagation; scattering function; wideband HF

17. AVAILABILITY STATEMENT

☒ UNLIMITED.

☐ FOR OFFICIAL DISTRIBUTION.

18. Security Class. *(This report)*
unclassified

19. Security Class. *(This page)*
unclassified

20. Number of pages
115

21. Price:

*U.S. GOVERNMENT PRINTING OFFICE 1998-0-673-018/40087