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 auto correlated 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 */
{
		/* Variables */
	int s;
	long int w, k;
		/* Code */
	for (s = 1; s < DATA /2; s +=2)
	{
		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 = w + 2147483563;
		cdat[s] = (float)w * 4.656613057392e-10;
	}
} /* End of ran2 */
```

```
                /* specavg4.cpp */
#include <stdio.h>
#include <math.h>
#define MAXLAYERS 3
#define DATA 2048
#define TWOPI 6.28318530717959
#define PHASE 0.785398163397448
#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;
};
                /* Global Variables */
        float tdat[DATA];

/* ---------------------------------------------------------------------------
 * The following is an implementation of the Fast Fourier Transform:
 * nn is a power of two, isign = 1 => DFT , isign = -1 => IDFT
 * Expects data[1] = Re{ x[0] } data[2] = Im{ x[0] }
 * so decrement data by 1 when calling
 * i.e., little_four(data - 1, 4096, 1);
 * data must be a one dimensional array of length 2 * nn
 *--------------------------------------------------------------------------- */
void little_four(float data[DATA], int nn, int isign)  /* Usually 2 * DATA */
{
        See Press et al. [32, pp. 404 - 414]. for code listing.
} /* End of little_four */

void imp(float datb[DATA], struct ray_path pdsi[MAXLAYERS], struct compute cdsi[1], int oo,
                double tau_kk, double ggg)
{
                /* Variables */
        int r, m;
        float xkm, ykm;
        double exparg, no_squirt, sine, cosine, timexx, summer, sind, cosind, expargfx;
                /* Code */
        for (r = 0; r < DATA; r++)
```

```
		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. NTIA Report 98-348	2. Gov't Accession No.	3. Recipient's Accession No.

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.	9. Project/Task/Work Unit No.
8. PERFORMING ORGANIZATION NAME AND ADDRESS National Telecommunications and Information Admin. Institute for Telecommunication Sciences 325 Broadway Boulder, CO 80303	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	20. Number of pages 115
	19. Security Class. *(This page)* unclassified	21. Price: