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.

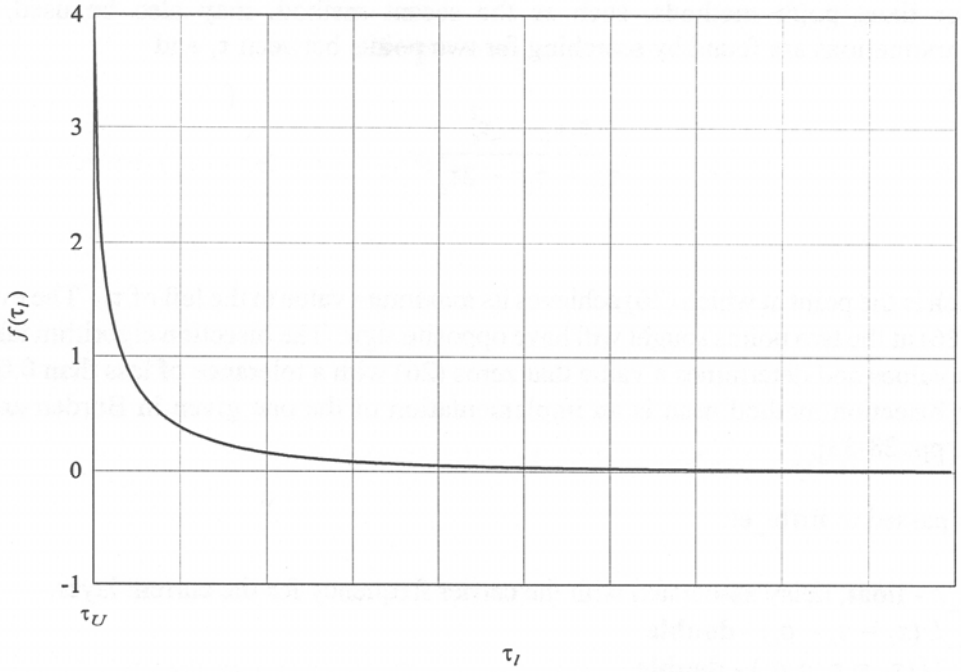

Figure 14. Horizontal and vertical asymptotes of function (26) above τ_U.

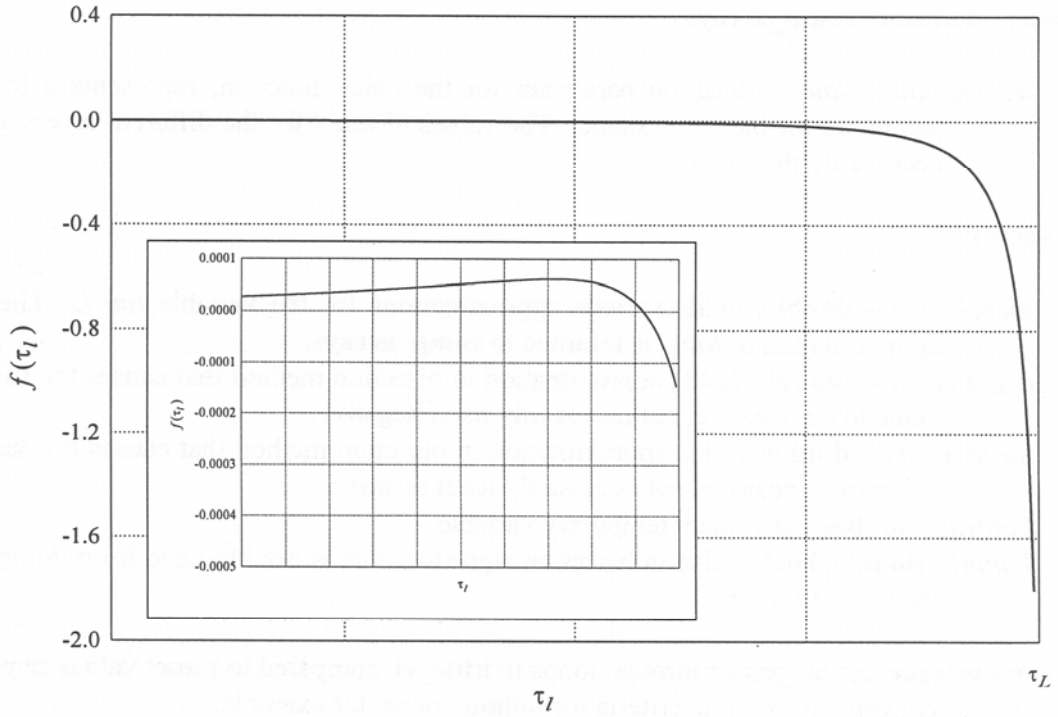

Figure 15. Horizontal and vertical asymptotes of function (26) below τ_L. Inset: Zero crossing and function maximum illustrated at finer scale.

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

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

that, in turn, are inputs to the generator

$$W_{i+1} = U_{i+1} + V_{i+1} \pmod{2{,}147{,}483{,}563}\ . \tag{32}$$

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 */
```