1. **AD Global Variables.** Global Routines and Variables. Changed version to reflect bug fix in the Fresnel routine section.
   Revised in May 1995 to allow slides to absorb and various modifications to improve the way that the file looks.
   Revision May 1996 to remove uninitialized tfluence
   Revision May 1998 to improve wrarray.

   \(\text{ad}_\text{globl}.\text{c}\ 1) \equiv\ \\
   \#include \langle \text{math.h}\rangle\ \\
   \#include \langle \text{stdio.h}\rangle\ \\
   \#include \langle \text{stdlib.h}\rangle\ \\
   \#include "\text{ad}_\text{globl}.\text{h}\"\ \\
   \#include "\text{ad}_\text{frsnl}.\text{h}\"\ \\
   \text{(Global variables for adding-doubling 10)}\ \\
   \text{(Definition for Zero} \_ \text{Layer 13)}\ \\
   \text{(Definition for AD} \_ \text{error 15)}\ \\
   \text{(Definition for URU} \_ \text{and} \_ \text{UR1 17)}\ \\
   \text{(Definition for UFU} \_ \text{and} \_ \text{UF1 19)}\ \\
   \text{(Definition for wrmatrix 21)}\ \\
   \text{(Definition for wrarray 23)}\ \\

2. \(\text{ad}_\text{globl}.\text{h}\ 2) \equiv\ \\
   \text{(Preprocessor definitions)}\ \\
   \text{(Types to export from AD Globals 8)}\ \\
   \text{(External variables to export from AD Globals 11)}\ \\
   \text{(Prototype for Zero} \_ \text{Layer 12);}\ \\
   \text{(Prototype for AD} \_ \text{error 14);}\ \\
   \text{(Prototype for URU} \_ \text{and} \_ \text{UR1 16);}\ \\
   \text{(Prototype for UFU} \_ \text{and} \_ \text{UF1 18);}\ \\
   \text{(Prototype for wrmatrix 20);}\ \\
   \text{(Prototype for wrarray 22);}
3. Constants.
   We begin with a version number for this just the adding-doubling part of the program. (The
   inverse adding-doubling code may have a different version number.)
   
   #define AD_VERSION 1.50

4. The number of quadrature points determines how accurately the integrals are performed. Larger
   numbers of quadrature points lead to more accurate solutions. Fewer points yield much
   faster computations since the computation time is proportional to $n^3$ or $n^2 \ln n$ because an $n \times n$
   matrix must be inverted.

   For most practical purposes four quadrature points is plenty. However, if you need very accurate
   reflection and transmission values, then increase the number of quadrature points. For example,
   if you want to verify a Monte Carlo implementation, then just crank the number up to 16 or 32
   and you are almost certain to get 5 significant digits in your answer.

   The number of quadrature points does not need to be a power of 2, but it should be an even
   number. If it isn’t then somewhere in the bowels of this program it will get changed. Finally, if
   you are unsure of how accurate a solution is, then increase the number of quadrature points and
   repeat the algorithm.

   There is no intrinsic reason that the maximum number of quadrature points is limited to 64.
   If you have enough memory then this number can be increased. But if you have read the stuff
   above, my feeling is, why bother?
   
   #define MAX_QUAD_PTS 64
   #define DEFAULT_QUAD_PTS 4

5. The two permissible phase functions are isotropic and Henyey-Greenstein.
   
   #define ISOTROPIC 0
   #define HENYEY_GREENSTEIN 1

6. The last two constants are related to the details of how the initial adding-doubling layer is
   generated. It is very unlikely that these will ever be used by anyone.
   
   #define DIAMOND 0
   #define INFINITESIMAL_GENERATOR 1

7. This last define is so that intermediate values can be generated during the calculation of the
   initial layer matrices. It is named after Martin Hammer who requested it.
   
   #define MARTIN_HAMMER 1
8. Types.

The fundamental structure for an adding-doubling calculation keeps all the details of the optical properties of the sample together. The sample is bounded by a glass slide above and below. The glass slides have indices of refraction $n_{\text{top slide}}$ and $n_{\text{bottom slide}}$. The glass slides may absorb light, in which case $b_{\text{top slide}}$ or $b_{\text{bottom slide}}$ may be non-zero.

The albedo of the slab is denoted $a$, the optical thickness of the slab by $b = (\mu_a + \mu_s)d$, and the average cosine of the phase function by $g$. The phase function of the slab is restricted to just isotropic and Henyey-Greenstein phase functions at the moment.

\[
\langle \text{Types to export from AD Globals} \rangle \equiv
\begin{align*}
\text{typedef struct AD_slab_type} & \{
\text{double } a; \\
\text{double } b; \\
\text{double } g; \\
\text{int } \text{phase\_function}; \\
\text{double } n_{\text{slab}}; \\
\text{double } n_{\text{top slide}}; \\
\text{double } n_{\text{bottom slide}}; \\
\text{double } b_{\text{top slide}}; \\
\text{double } b_{\text{bottom slide}};
\} \text{slab\_type};
\end{align*}
\]

See also section 9.

This code is used in section 2.

9. \langle \text{Types to export from AD Globals} \rangle +\equiv

\[
\begin{align*}
\text{typedef struct AD_method_type} & \{
\text{int } \text{quad\_pts}; \\
\text{double } a_{\text{calc}}, b_{\text{calc}}, g_{\text{calc}}, b_{\text{thinnest}};
\} \text{method\_type};
\end{align*}
\]

10. The $\text{Martin\_Hammer}$ variable only exists to print internal results when testing. Its only an integer and doesn’t take up much space so here it is.

\[
\langle \text{Global variables for adding-doubling} \rangle \equiv
\begin{align*}
\#define \text{AD\_GLOBAL\_SOURCE} \\
\text{double } \text{angle}[\text{MAX\_QUAD\_PTS} + 1]; \\
\text{double } \text{weight}[\text{MAX\_QUAD\_PTS} + 1]; \\
\text{double } \text{twoaw}[\text{MAX\_QUAD\_PTS} + 1]; \\
\text{int } \text{Martin\_Hammer} = 0;
\end{align*}
\]

This code is used in section 1.

11. \langle \text{External variables to export from AD Globals} \rangle \equiv

\[
\begin{align*}
\#ifndef \text{AD\_GLOBAL\_SOURCE} \\
\text{extern double } \text{angle}[\text{MAX\_QUAD\_PTS} + 1]; \\
\text{extern double } \text{weight}[\text{MAX\_QUAD\_PTS} + 1]; \\
\text{extern double } \text{twoaw}[\text{MAX\_QUAD\_PTS} + 1]; \\
\text{extern int } \text{Martin\_Hammer};
\#endif
\end{align*}
\]

This code is used in section 2.
12. **Global routines.** My standard error handler

(Prototype for Zero_Layer 12) ≡

```c
void AD_error(char error_text[])
```

This code is used in sections 2 and 13.

13. (Definition for Zero_Layer 13) ≡

(Prototype for Zero_Layer 12)

```c
{ 
    fprintf(stderr,"Adding-Doubling_error\n");
    fprintf(stderr,"%s\n", error_text);
    fprintf(stderr,"...now_exiting_to_system...\n");
    exit(1);
}
```

This code is used in section 1.

14. (Prototype for AD_error 14) ≡

```c
void Zero_Layer(int n, double **r, double **t)
```

This code is used in sections 2 and 15.

15. (Definition for AD_error 15) ≡

(Prototype for AD_error 14)

```c
{ 
    int i, j;
    for (i = 1; i ≤ n; i++)
        for (j = 1; j ≤ n; j++)
            t[i][j] = 0.0;
    r[i][j] = 0.0;
    for (i = 1; i ≤ n; i++)
        t[i][i] = 1/twoaw[i];
}
```

This code is used in section 1.

16. (Prototype for URU_and_UR1 16) ≡

```c
void URU_and_UR1(int n, double n_slab, double **R, double *URU, double *UR1)
```

This code is used in sections 2 and 17.

---

`AD_GLOBAL_SOURCE`: ???, §0.

`fprintf`, `<stdio.h>`

`stderr`, `<stdio.h>`

`exit`, `<stdlib.h>`

`MAX_QUAD_PTS = 64`, §4.
17.  ⟨Definition for URU and UR1 17⟩ ≡  
(Prototype for URU and UR1 16)  
{  
    int i, j;  
    double temp, critical_mu;  
    int last_j;  
    critical_mu = Cos_Critical_Angle(n_slab, 1.0);  
    last_j = 1;  
    while (angle[last_j] < critical_mu) last_j ++;  
    *URU = 0.0;  
    for (i = last_j; i ≤ n; i++) {  
        temp = 0.0;  
        for (j = 1; j ≤ n; j++) temp += R[i][j] * twoaw[j];  
        *URU += temp * twoaw[i];  
    }  
    *UR1 = temp;  
    *URU *= n_slab * n_slab;  
}  
This code is used in section 1.  

18.  ⟨Prototype for UFU and UF1 18⟩ ≡  
void UFU_andUF1 (int n, double n_slab, double **Lup, double **Ldown, double *UFU, double *UF1)  
This code is used in sections 2 and 19.  

19.  ⟨Definition for UFU_andUF1 19⟩ ≡  
(Prototype for UFU_andUF1 18)  
{  
    int i, j;  
    double temp;  
    *UFU = 0.0;  
    for (j = 1; j ≤ n; j++) {  
        temp = 0.0;  
        for (i = 1; i ≤ n; i++) temp += (Lup[i][j] + Ldown[i][j]) * 2 * weight[i];  
        *UFU += twoaw[j] * temp;  
    }  
    *UF1 = temp * n_slab * n_slab;  
    *UFU *= n_slab * n_slab / 2;  
}  
This code is used in section 1.  

20.  ⟨Prototype for wrmatrix 20⟩ ≡  
void wrmatrix (int n, double **a)  
This code is used in sections 2 and 21.  

21.  ⟨Definition for wrmatrix 21⟩ ≡  
(Prototype for wrmatrix 20)  
{
int i, j;
double tflux, flux;
char *title = "flux";

printf("%9d", 0.0);
for (i = 1; i <= n; i++)
    printf("%9.5f", angle[i]);

flux = 0.0;
for (i = 1; i <= n; i++){
    flux = 0.0;
    for (j = 1; j <= n; j++)
        if ((a[i][j] > 10) || (a[i][j] < -10))
            printf("*****");
        else
            printf("%9.5f", a[i][j]);
    flux += a[i][j] * twoaw[j];
    tflux += flux * twoaw[i];
}

printf("%9s", "flux");
for (i = 1; i <= n; i++)
    printf("%9.5f", flux);

printf("%9.5f
", tflux);

This code is used in section 1.

22. ⟨Prototype for wrarray 22⟩≡

void wrarray(int n, double *a)

This code is used in sections 2 and 23.

---

angle: double [], §10.  
Cos_Critical_Angle: double, §137.  
n: int, §16.  
n_slab: double, §16.  
printf, <stdio.h>.  
R: double **, §16.  
twoaw: double [], §10.  
URU_and_UR1: void, §16.  
weight: double [], §10.  
UR1: double *, §16.  

23. (Definition for `swarray 23`) ≡
(Prototype for swarray 22)
{
    int i;
    double sum;
    for (i = 1; i ≤ n; i++) printf("%9.5f", angle[i]);
    printf("%9s
", "angles");
    sum = 0.0;
    for (i = 1; i ≤ n; i++) {
        if (a[i] > 10 ∨ a[i] < -10) printf("*********");
        else printf("%9.5f", a[i]);
        if (a[i] < 10 ∧ a[i] < -10) sum += a[i];
    }
    printf("%9.5f", sum);
    printf("%9s
", "(natural) ");
    sum = 0.0;
    for (i = 1; i ≤ n; i++) {
        if (a[i] > 10 ∨ a[i] < -10) printf("*********");
        else printf("%9.5f", a[i]/twoaw[i]);
        if (a[i] < 10 ∧ a[i] < -10) sum += a[i];
    }
    printf("%9.5f", sum);
    printf("%9s
", "*2aw ");
    for (i = 1; i ≤ (n + 2); i++) printf("**********");
    printf("\n\n");
}

This code is used in section 1.

24. Just print out an array without mucking
(Prototype for swarray 24) ≡
    void swarray(int n, double *a)
This code is used in section 25.

25. (Definition for swarray 25) ≡
(Prototype for swarray 24)
{
    int i;
    double sum;
    for (i = 1; i ≤ n; i++) printf("%9.5f", angle[i]);
    printf("%9s
", "*2aw ");
    sum = 0.0;
    for (i = 1; i ≤ n; i++) {
        if (a[i] > 10 ∨ a[i] < -10) printf("*********");
        else printf("%9.5f", a[i]/twoaw[i]);
        if (a[i] < 10 ∧ a[i] < -10) sum += a[i];
    }
    printf("%9.5f\n", sum);
    for (i = 1; i ≤ (n + 2); i++) printf("**********");
printf (*"\n\n");
}
26. **AD Prime.** This has the rather stupid name prime because I was at a loss for another. Currently this is very poorly commented. The fluence routine has not even been checked. There may or may not be errors associated with the $n^2$ law in there. It just needs to be checked.

```c
#include <math.h>
#include "nr_util.h"
#include "ad_globl.h"
#include "ad_bound.h"
#include "ad_start.h"
#include "ad_doubl.h"
#include "ad_prime.h"
#include "ad_matrx.h"

⟨Definition for RT Matrices 29⟩
⟨Definition for RT 31⟩
⟨Definition for RTabs 45⟩
⟨Definition for Flux_Fluence 55⟩
```

27. ```c
#include "ad_prime.h 27⟩
⟨Preprocessor definitions⟩
⟨Prototype for RT Matrices 28⟩;
⟨Prototype for RT 30⟩;
⟨Prototype for RTabs 44⟩;
⟨Prototype for Flux_Fluence 54⟩;
```
28. **R and T Matrix routines.** This section contains the routine to calculate the reflection and transmission matrix for a scattering and absorbing slab. Basically you just need to set the number of quadrature points \( \vec{quad} \) pts and the optical properties (the albedo, anisotropy, optical thickness, and choice of phase function) in \( \text{slab} \). Call this routine and get back matrices filled with cool numbers.

(Prototype for \( RT \) Matrices 28) \( \equiv \)

```c
void RT_Matrices(int n, struct AD_slab_type *slab, struct AD_method_type *method, double **R, double **T)
```

This code is used in sections 27 and 29.

29. **Definition for \( RT \) Matrices 29** \( \equiv \)

(Prototype for \( RT \) Matrices 28) \{ 
```
double d;
if (n < 3) method-quad_pts = DEFAULT_QUAD_PTS;
else if (n > MAX_QUAD_PTS) method-quad_pts = MAX_QUAD_PTS;
else if (n & 1) method-quad_pts = n/2 * 2;
else method-quad_pts = n; 
Choose_Method(+slab, method);
if (slab-b ≤ 0) { 
    Zero_Layer(n, R, T);
    return;
}
```
```
n = method-quad_pts;
Init_Layer(+slab, +method, R, T);
if (slab-b = HUGE_VAL) d = 1.0; /* Ignored ... just set it something. */
else d = method-b_thinnest * slab-b/method-b_calc;
```
```
Double_Until(n, R, T, d, slab-b);
```
```
}
```
```
This code is used in section 26.
```

\( \text{AD_method_type: struct, } \S 9. \)
\( \text{AD_slab_type: struct, } \S 8. \)
\( b: \text{double, } \S 8. \)
\( b_calc: \text{double, } \S 9. \)
\( b_thinnest: \text{double, } \S 9. \)
\( \text{Choose_Method: void, } \S 71. \)
\( \text{DEFAULT_QUAD_PTS = 4, } \S 4. \)
\( \text{Double_Until: void, } \S 102. \)
\( \text{Flux_Fluence: void, } \S 54. \)
\( \text{HUGE_VAL, } \text{<math.h>}. \)
\( \text{Init_Layer: void, } \S 87. \)
\( \text{MAX_QUAD_PTS = 64, } \S 4. \)
\( \text{quad_pts: int, } \S 9. \)
\( \text{R: void, } \S 30. \)
\( \text{RTabs: void, } \S 44. \)
\( \text{Zero_Layer: void, } \S 14. \)
30. Total reflection and transmission.

RT is the top level routine for accessing the adding-doubling algorithm. By passing the optical parameters characteristic of the slab, this routine will do what it must to return the total reflection and transmission for collimated and diffuse irradiance.

This routine has three different components based on if zero, one, or two boundary layers must be included. If the index of refraction of the slab and the top and bottom slides are all one, then no boundaries need to be included. If the top and bottom slides are identical, then some simplifications can be made and some time saved as a consequence. If the top and bottom slides are different, then the full red carpet treatment is required.

Since the calculation time increases for each of these cases we test for matched boundaries first. If the boundaries are matched then don’t bother with boundaries for the top and bottom. Just calculate the integrated reflection and transmission. Similarly, if the top and bottom slides are similar, then quickly calculate these.

⟨Prototype for RT 30⟩ ≡

\[
\text{void RT(int } n, \text{ struct AD_slab_type } *slab, \text{ double } *UR1, \text{ double } *UT1, \text{ double } *URU, \text{ double } *UTU)\]

This code is used in sections 27 and 31.

31. ⟨Definition for RT 31⟩ ≡

⟨Prototype for RT 30⟩

\{
  ⟨Declare variables for RT 32⟩
  ⟨Allocate and calculate R and T for homogeneous slab 33⟩
  if (slab_n_slab \equiv 1 \land slab_n_top_slide \equiv 1 \land slab_n_bottom_slide \equiv 1 \land slab_b_top_slide \equiv 0 \land slab_b_bottom_slide \equiv 0)
    {  ⟨Do slab with no boundaries 34⟩
    }
  else if (slab_n_top_slide \equiv slab_n_bottom_slide \land slab_b_top_slide \equiv 0 \land slab_b_bottom_slide \equiv 0)
    {  ⟨Allocate and generate top boundary 35⟩
      ⟨Do slab with matched top and bottom boundaries 36⟩
      ⟨Free top boundary 37⟩
    }
  else
    {  ⟨Allocate and generate top boundary 35⟩
      ⟨Allocate and generate bottom boundary 38⟩
      ⟨Allocate misc matrices 39⟩
      ⟨Do slab with mismatched boundaries 40⟩
      ⟨Free misc matrices 41⟩
      ⟨Free bottom boundary 42⟩
      ⟨Free top boundary 37⟩
    }

  ⟨Free R and T 43⟩
\}

This code is used in section 26.

32. ⟨Declare variables for RT 32⟩ ≡

double **R, **T, **R2, **T2;
double *R01, *R10, +T01, +T10;
double *R23, *R32, +T23, +T32;
double **R02, **R20, **T02, **T20;
double **R03, **R30, **T03, **T30;
double **atemp, **btemp;
struct AD_method_type method;

This code is used in section 31.

33. Find the R and T for a homogeneous slab without boundaries
\langle Allocate and calculate R and T for homogeneous slab \rangle \equiv
\begin{align*}
R &= \text{dmatrix}(1,n,1,n); \\
T &= \text{dmatrix}(1,n,1,n); \\
RT\text{Matrices}(n, \text{slab}, &\text{method}, R, T);
\end{align*}
This code is used in sections 31 and 45.

34. \langle Do slab with no boundaries \rangle \equiv
\begin{align*}
URU\_and\_UR1(n, \text{slab}-n, \text{slab}, R, URU, UR1); \\
URU\_and\_UR1(n, \text{slab}-n, \text{slab}, T, UTU, UT1);
\end{align*}
This code is used in section 31.

35. \langle Allocate and generate top boundary \rangle \equiv
\begin{align*}
R01 &= \text{dvector}(1,n); \\
R10 &= \text{dvector}(1,n); \\
T01 &= \text{dvector}(1,n); \\
T10 &= \text{dvector}(1,n); \\
\text{Init\_Boundary}(\ast\text{slab}, \text{method}.quad\_pts, R01, R10, T01, T10, \text{TOP\_BOUNDARY});
\end{align*}
This code is used in sections 31 and 49.

36. \langle Do slab with matched top and bottom boundaries \rangle \equiv
\begin{align*}
\text{atemp} &= \text{dmatrix}(1,n,1,n); \\
\text{btemp} &= \text{dmatrix}(1,n,1,n); \\
R2 &= \text{dmatrix}(1,n,1,n); \\
T2 &= \text{dmatrix}(1,n,1,n); \\
\text{Add\_Slides}(n, R01, R10, T01, T10, R, T, R2, T2, \ast\text{atemp}, \ast\text{btemp}); \\
URU\_and\_UR1(n, \text{slab}-n, \text{slab}, R2,URU, UR1); \\
URU\_and\_UR1(n, \text{slab}-n, \text{slab}, T2,UTU, UT1); \\
\text{free\_dmatrix}(\ast\text{atemp}, 1,n,1,n); \\
\text{free\_dmatrix}(\ast\text{btemp}, 1,n,1,n); \\
\text{free\_dmatrix}(R2, 1,n,1,n); \\
\text{free\_dmatrix}(T2, 1,n,1,n);
\end{align*}
This code is used in section 31.
37.  \(\text{Free top boundary}\) \(\equiv\)
\[
\text{free\_dvector}(R01, 1, n); \\
\text{free\_dvector}(R10, 1, n); \\
\text{free\_dvector}(T01, 1, n); \\
\text{free\_dvector}(T10, 1, n);
\]
This code is used in sections 31 and 45.

38.  \(\text{Allocate and generate bottom boundary}\) \(\equiv\)
\[
R23 = \text{dvector}(1, n); \\
R32 = \text{dvector}(1, n); \\
T23 = \text{dvector}(1, n); \\
T32 = \text{dvector}(1, n); \\
\text{Init\_Boundary}(*\text{slab}, \text{method\_quad\_pts}, R23, R32, T23, T32, \text{BOTTOM\_BOUNDARY});
\]
This code is used in sections 31 and 50.

39.  \(\text{Allocate misc matrices}\) \(\equiv\)
\[
R02 = \text{dmatrix}(1, n, 1, n); \\
R20 = \text{dmatrix}(1, n, 1, n); \\
T02 = \text{dmatrix}(1, n, 1, n); \\
T20 = \text{dmatrix}(1, n, 1, n); \\
R03 = \text{dmatrix}(1, n, 1, n); \\
R30 = \text{dmatrix}(1, n, 1, n); \\
T03 = \text{dmatrix}(1, n, 1, n); \\
T30 = \text{dmatrix}(1, n, 1, n); \\
\text{atemp} = \text{dmatrix}(1, n, 1, n); \\
\text{btemp} = \text{dmatrix}(1, n, 1, n);
\]
This code is used in sections 31 and 45.

40.  \(\text{Do slab with mismatched boundaries}\) \(\equiv\)
\[
\text{Add\_Top}(n, R01, R10, T01, T10, R02, R20, T02, T20, \text{atemp}, \text{btemp}); \\
\text{Add\_Bottom}(n, R02, R20, T02, T20, R23, R32, T23, T32, R03, R30, T03, T30, \text{atemp}, \text{btemp}); \\
\text{URU\_and\_UR1}(n, \text{slab}\_n\_slab, R03, \text{URU}, \text{UR1}); \\
\text{Transpose\_Matrix}(n, T03); \\
\text{URU\_and\_UR1}(n, \text{slab}\_n\_slab, T03, \text{UTU}, \text{UT1});
\]
This code is used in section 31.

41.  \(\text{Free misc matrices}\) \(\equiv\)
\[
\text{free\_dmatrix}(R02, 1, n, 1, n); \\
\text{free\_dmatrix}(R20, 1, n, 1, n); \\
\text{free\_dmatrix}(T02, 1, n, 1, n); \\
\text{free\_dmatrix}(T20, 1, n, 1, n); \\
\text{free\_dmatrix}(R03, 1, n, 1, n); \\
\text{free\_dmatrix}(R30, 1, n, 1, n); \\
\text{free\_dmatrix}(T03, 1, n, 1, n); \\
\text{free\_dmatrix}(T30, 1, n, 1, n); \\
\text{free\_dmatrix}(\text{atemp}, 1, n, 1, n); \\
\text{free\_dmatrix}(\text{btemp}, 1, n, 1, n);
\]
This code is used in sections 31 and 45.
42. \( \{ \text{Free bottom boundary} \ \ 42 \} \equiv \)
\[
\text{free\_dvector(R23, 1, n)}; \\
\text{free\_dvector(R32, 1, n)}; \\
\text{free\_dvector(T23, 1, n)}; \\
\text{free\_dvector(T32, 1, n)};
\]
This code is used in sections 31 and 45.

43. \( \{ \text{Free R and T} \ \ 43 \} \equiv \)
\[
\text{free\_dmatrix}(R, 1, n, 1, n); \\
\text{free\_dmatrix}(T, 1, n, 1, n);
\]
This code is used in sections 31 and 45.

---

Add\_Bottom: void, §129.
Add\_Top: void, §127.
atemp: double **, §32.
BOTTOM\_BOUNDARY = 1, §116.
btemp: double **, §32.
dmatrix, <nr\_util.h>.
dvector, <nr\_util.h>.
free\_dmatrix, <nr\_util.h>.
free\_dvector, <nr\_util.h>.
Inst\_Boundary: void, §116.
method: struct AD\_method\_type, §32.
n: int, §30.
n\_slab: double, §18.
quad\_pts: int, §9.
R: double **, §32.
R01: double *, §32.
R02: double **, §32.
R03: double **, §32.
R10: double *, §32.
R20: double **, §32.
R23: double *, §32.
R30: double **, §32.
R32: double *, §32.
R02: double **, §32.
T02: double **, §32.
T03: double **, §32.
T10: double *, §32.
T20: double **, §32.
T23: double *, §32.
T30: double **, §32.
T32: double *, §32.
Transpose\_Matrix: void, §164.
UR1: double *, §30.
URU: double *, §30.
URU\_and\_UR1: void, §16.
UT1: double *, §30.
UTU: double *, §30.
44. Including absorbing slides.

The idea is to create a function that includes absorption in the top and bottom slides. This is done by creating two extra layers, finding the full reflection and transmission matrices for these layers and adding them to the slab. Of course this only works when all the indices of refraction are the same. Yikes!

This routine returns \( UR1 \) and \( UT1 \) for light incident from the top of the slab. The values for light incident from the bottom will be different when the slides on the top and bottom are different. Caveat emptor!

\[
\langle \text{Prototype for } RTabs \rangle \equiv \\
\text{void } RTabs(\text{int } n, \text{struct AD slab type } \star \star \text{slab, double } \star UR1, \text{double } \star UT1, \text{double } \star URU, \text{double } \star UTU)
\]

This code is used in sections 27 and 45.

45. \( \langle \text{Definition for } RTabs \rangle \equiv \)

\( \langle \text{Prototype for } RTabs \rangle \equiv \)

\{ \\
\text{double } \star \star Rtop, \star \star Ttop, \star \star Rbottom, \star \star Tbottom; \\
\text{struct AD slab type } \text{slab1}; \\
\text{double } \text{btop, bbottom}; \\
\langle \text{Allocate and calculate R and T for homogeneous slab } \rangle \\
\langle \text{Allocate and calculate top absorbing slide } \rangle \\
\langle \text{Allocate and calculate bottom absorbing slide } \rangle \\
\langle \text{Allocate misc matrices } \rangle \\
\langle \text{Allocate and calculate top non-absorbing boundary } \rangle \\
\langle \text{Allocate and calculate bottom non-absorbing boundary } \rangle \\
\langle \text{Add all the stuff together } \rangle \\
\langle \text{Free misc matrices } \rangle \\
\langle \text{Free bottom boundary } \rangle \\
\langle \text{Free top boundary } \rangle \\
\langle \text{Free R and T } \rangle \\
\langle \text{Free matrices for the top and bottom absorbing slides } \rangle
\}

This code is used in section 26.

46. \( \langle \text{Declare variables for } RTabs \rangle \equiv \)

\text{double } \star \star R, \star \star T; \\
\text{double } \star R01, \star R10, \star T01, \star T10; \\
\text{double } \star R23, \star R32, \star T23, \star T32; \\
\text{double } \star \star R02, \star \star R03, \star \star R10, \star \star R20, \star \star R23, \star \star T02, \star \star T20; \\
\text{double } \star \star R03, \star \star R03, \star \star R03, \star \star T03, \star \star T30; \\
\text{double } \star \star atemp, \star \star btemp; \\
\text{struct AD method type } \text{method};

This code is used in section 45.

47. \( \langle \text{Allocate and calculate top absorbing slide } \rangle \equiv \)

\text{slab1.b = slab-b_top_slide;}

This code is used in sections 27 and 45.
Adding-Doubling (Version 1.5): INCLUDING ABSORBING SLIDES

```c
slab1.a = 0;
slab1.g = 0;
slab1.phase_function = HENYEY_GREENSTEIN;
slab1.n_slab = slab.n_slab;
slab1.n_top_slide = 1.0;
slab1.n_bottom_slide = 1.0;
slab1.b_top_slide = 0.0;
slab1.b_bottom_slide = 0.0;
Rtop = dmatrix(1, n, 1, n);
Ttop = dmatrix(1, n, 1, n);
RT_Matrices(n, &slab1, &method, Rtop, Ttop);
```

This code is used in section 45.

48. ⟨Allocate and calculate bottom absorbing slide 48⟩ ≡
```
slab1.b = slab.b_bottom_slide;
Rbottom = dmatrix(1, n, 1, n);
Tbottom = dmatrix(1, n, 1, n);
RT_Matrices(n, &slab1, &method, Rbottom, Tbottom);
```

This code is used in section 45.

49. ⟨Allocate and calculate top non-absorbing boundary 49⟩ ≡
```
btop = slab.b_top_slide;
slab.b_top_slide = 0;
⟨Allocate and generate top boundary 35⟩
slab.b_top_slide = btop;
```

This code is used in section 45.

50. ⟨Allocate and calculate bottom non-absorbing boundary 50⟩ ≡
```
bbottom = slab.b_bottom_slide;
slab.b_bottom_slide = 0;
⟨Allocate and generate bottom boundary 38⟩
slab.b_bottom_slide = bbottom;
```

This code is used in section 45.

---

a: double *, §24.
AD_method_type: struct, §9.
AD_slab_type: struct, §8.
b: double, §8.
b_bottom_slide: double, §8.
b_top_slide: double, §8.
dmatrix, <nr_util.h>.
g: double, §8.
HENYEY_GREENSTEIN = 1, §5.
n_slab: double, §18.
n_top_slide: double, §8.
phase_function: int, §8.
RT_Matrices: void, §28.
51. \( \langle \text{Add all the stuff together} \rangle \equiv \)
\[
\begin{align*}
&\text{Add}\(n, R_{top}, T_{top}, R_{top}, R, R, T, R02, R20, T02, T20)\); \\
&\text{Add}\(n, R02, R20, T02, T20, R_{bottom}, R_{bottom}, T_{bottom}, T_{bottom}, R03, R30, T03, T30)\); \\
&\text{Add}_\text{Top}\(n, R01, R10, T01, T10, R03, R30, T03, R02, R20, T02, T20, \text{atemp}, \text{btemp})\); \\
&\text{Add}_\text{Bottom}\(n, R02, R20, T02, T20, R23, R32, T23, T32, R03, R30, T03, T30, \text{atemp}, \text{btemp})\); \\
&\text{URU and UR1}\(n, \text{slab} \vec{n}_{\text{slab}}, R03, \text{URU}, \text{UR1})\); \\
&\text{Transpose Matrix}\(n, T03)\); \\
&\text{URU and UR1}\(n, \text{slab} \vec{n}_{\text{slab}}, T03, \text{UTU}, \text{UT1})\);
\end{align*}
\]
This code is used in section 45.

52. \( \langle \text{Free matrices for the top and bottom absorbing slides} \rangle \equiv \)
\[
\begin{align*}
&\text{free_dmatrix}\(R_{top}, 1, n, 1, n)\); \\
&\text{free_dmatrix}\(T_{top}, 1, n, 1, n)\); \\
&\text{free_dmatrix}\(R_{bottom}, 1, n, 1, n)\); \\
&\text{free_dmatrix}\(T_{bottom}, 1, n, 1, n)\);
\end{align*}
\]
This code is used in section 45.
53. Flux and Fluence.
Calculates the flux and fluence at various depths between the optical depths \( z_{\text{min}} \) and \( z_{\text{max}} \) for a slab. The number of values is \( \text{intervals} + 1 \) times...i.e. it calculates at \( z_{\text{min}}, z_{\text{min}} + (z_{\text{max}} - z_{\text{min}})/\text{intervals}, \ldots, z_{\text{max}} \).

The fluence and fluxes at 0 and slab.b are calculated just inside the boundary, i.e. beneath any existing glass slide or just below a mismatched boundary.

This routine could be improved dramatically. I just have not had the need so far.

This has not been adequately tested.

#define \( \text{MAX\_FLUENCE\_INTERVALS} \) 200

54. (Prototype for \textit{Flux\_Fluence} 54) ≡

\textbf{void} \textit{Flux\_Fluence} (\textit{int} \textit{n}, \textbf{struct AD\_slab\_type} *\textit{slab}, \textbf{double} \textit{zmin}, \textbf{double} \textit{zmax}, \textbf{int} \textit{intervals}, \textbf{double} *\textit{UF1\_array}, \textbf{double} *\textit{UFU\_array}, \textbf{double} *\textit{flux\_up}, \textbf{double} *\textit{flux\_down})

This code is used in sections 27 and 55.

55. (Definition for \textit{Flux\_Fluence} 55) ≡
(Prototype for \textit{Flux\_Fluence} 54)

\{ 
  (Declare variables for \textit{Flux\_Fluence} 56)
  \textbf{if} (\textit{intervals} > \textit{MAX\_FLUENCE\_INTERVALS})
    \textit{AD\_error}("\textit{too\_many\_intervals\_requested}\ 1\ \textit{increase}\ 1\ \textit{the\_const\_max\_fluence\_intervals}");
  \textbf{for} (\textit{i} = 0; \textit{i} \leq \textit{intervals}; \textit{i}++) \{ 
    (Find radiance at each depth 60)
    (Calculate Fluence and Flux 61)
  \}
  (Free all those intermediate matrices 62)
\}

This code is used in section 26.

---

\textit{AD\_error}: \textbf{void}, §12.

\textit{AD\_slab\_type}: \textbf{struct}, §8.

\textit{Add\_Top}: \textbf{void}, §129.

\textit{Add\_Bottom}: \textbf{void}, §129.

\textit{atemp}: \textbf{double **}, §46.

\textit{b}: \textbf{double}, §8.

\textit{btemp}: \textbf{double **}, §46.

\textit{freem\_d\_matrix}, \textit{\langle ar\_util\_h\rangle}.

\textit{i}: \textbf{int}, §25.

\textit{n}: \textbf{int}, §44.

\textit{n\_slab}: \textbf{double}, §18.

\textit{R}: \textbf{double **}, §46.

\textit{R01}: \textbf{double}, §46.

\textit{R02}: \textbf{double **}, §46.

\textit{R03}: \textbf{double **}, §46.

\textit{R10}: \textbf{double *}, §46.

\textit{R12}: \textbf{double **}, §46.

\textit{R20}: \textbf{double **}, §46.

\textit{R21}: \textbf{double}, §46.

\textit{R23}: \textbf{double **}, §46.

\textit{R30}: \textbf{double **}, §46.

\textit{R32}: \textbf{double **}, §46.

\textit{Rbottom}: \textbf{double **}, §45.

\textit{Rtop}: \textbf{double **}, §45.

\textit{slab}: \textbf{struct AD\_slab\_type *}, §44.

\textit{T}: \textbf{double **}, §46.

\textit{T01}: \textbf{double **}, §46.

\textit{T02}: \textbf{double **}, §46.

\textit{T03}: \textbf{double **}, §46.

\textit{T10}: \textbf{double *}, §46.

\textit{T20}: \textbf{double **}, §46.

\textit{T23}: \textbf{double *}, §46.

\textit{T30}: \textbf{double **}, §46.

\textit{T32}: \textbf{double *}, §46.

\textit{Top}: \textbf{double **}, §45.

\textit{Top\_array\_matrix}: \textbf{void}, §164.

\textit{Top\_d\_matrix}, \textit{\langle ar\_util\_h\rangle}.

\textit{UR1}: \textbf{double *}, §44.

\textit{URU}: \textbf{double *}, §44.

\textit{URU\_and\_UR1}: \textbf{void}, §16.

\textit{UT1}: \textbf{double *}, §44.

\textit{UTU}: \textbf{double *}, §44.
56. (Declare variables for Flux_Fluence 56) \equiv
define double +R01, +R10, +T01, +T10;
define double +R56, +R65, +T56, +T65;
define double ++R12, ++T12;
define double ++R23, ++T23;
define double ++R34, ++T34;
define double ++R85, ++T45;
define double ++R02, ++R20, ++T02, ++T20;
define double ++R46, ++R64, ++T46, ++T64;
define double ++R03, ++R30, ++T03, ++T30;
define double ++R36, ++R63, ++T36, ++T63;
define double ++Lup, ++Ldown;
define double **a, **b;
define double flx_down, flx_up, UFU, UF1;
define double slab_thickness;
define struct AD_method_type method;
define int i, j;

This code is used in section 55.

57. (Find the 02 matrix for the slab above all layers 57) \equiv

\[ \text{slab}\_\text{thickness} = \text{slab-b}; \quad / * \text{save it for later} */ \]

\[ \text{slab-b} = \text{zmin}; \]

\begin{align*}
\text{R12} &= \text{dmatrix}(1, n, 1, n); \\
\text{T12} &= \text{dmatrix}(1, n, 1, n); \\
\text{RT\_Matrices}(n, \text{slab}, \&\text{method}, \text{R12}, \text{T12}); \\
\text{R01} &= \text{dvector}(1, n); \\
\text{R10} &= \text{dvector}(1, n); \\
\text{T01} &= \text{dvector}(1, n); \\
\text{T10} &= \text{dvector}(1, n); \\
\text{Init\_Boundary}(\ast\text{slab}, \&\text{method}, \&\text{quad\_pts}, \text{R01}, \text{R10}, \text{T01}, \text{T10}, \text{TOP\_BOUNDARY}); \\
\text{R20} &= \text{dmatrix}(1, n, 1, n); \\
\text{T20} &= \text{dmatrix}(1, n, 1, n); \\
\text{R02} &= \text{dmatrix}(1, n, 1, n); \\
\text{T02} &= \text{dmatrix}(1, n, 1, n); \\
\text{a} &= \text{dmatrix}(1, 1, n); \\
\text{b} &= \text{dmatrix}(1, n, 1, n); \\
\text{Add\_Top}(n, \text{R01}, \text{R10}, \text{T01}, \text{T10}, \text{R12}, \text{T12}, \text{R20}, \text{T20}, \text{a}, \text{b}); \\
\text{free\_dmatrix}(\text{R12}, 1, n, 1, n); \\
\text{free\_dmatrix}(\text{T12}, 1, n, 1, n); \\
\text{free\_dvector}(\text{R01}, 1, n); \\
\text{free\_dvector}(\text{R10}, 1, n); \\
\text{free\_dvector}(\text{T01}, 1, n); \\
\text{free\_dvector}(\text{T10}, 1, n); \\
\end{align*}

This code is used in section 55.

58. (Find the 46 matrix for the slab below all layers 58) \equiv

\[ \text{slab-b} = \text{slab\_thickness} - \text{zmax}; \]

\[ \text{R45} = \text{dmatrix}(1, n, 1, n); \]
Adding-Doubling (Version 1.5): FLUX AND FLUENCE

\[ T_{45} = \text{dmatrix}(1, n, 1, n); \]
\[ RT\text{-Matrices}(n, \text{slab, } &\text{method}, R_{45}, T_{45}); \]
\[ R_{56} = \text{dvector}(1, n); \]
\[ R_{65} = \text{dvector}(1, n); \]
\[ T_{56} = \text{dvector}(1, n); \]
\[ T_{65} = \text{dvector}(1, n); \]
\[ \text{Init\_Boundary}(*\text{slab, } &\text{method, quad\_pts}, R_{56}, R_{65}, T_{56}, T_{65}, \text{BOTTOM\_BOUNDARY}); \]
\[ R_{46} = \text{dmatrix}(1, n, 1, n); \]
\[ T_{46} = \text{dmatrix}(1, n, 1, n); \]
\[ R_{64} = \text{dmatrix}(1, n, 1, n); \]
\[ T_{64} = \text{dmatrix}(1, n, 1, n); \]
\[ \text{Add\_Bottom}(n, R_{45}, R_{45}, R_{45}, T_{45}, T_{45}, T_{45}, T_{45}, R_{56}, R_{65}, T_{56}, T_{65}, R_{64}, R_{64}, T_{46}, T_{46}, a, b); \]
\[ \text{free\_dmatrix}(R_{45}, 1, n, 1, n); \]
\[ \text{free\_dmatrix}(T_{45}, 1, n, 1, n); \]
\[ \text{free\_dvector}(R_{56}, 1, n); \]
\[ \text{free\_dvector}(R_{65}, 1, n); \]
\[ \text{free\_dvector}(T_{56}, 1, n); \]
\[ \text{free\_dvector}(T_{65}, 1, n); \]
\[ \text{free\_dmatrix}(a, 1, n, 1, n); \]
\[ \text{free\_dmatrix}(b, 1, n, 1, n); \]

This code is used in section 55.

59. 〈Allocate intermediate matrices 59〉 ≡

\[ R_{23} = \text{dmatrix}(1, n, 1, n); \]
\[ T_{23} = \text{dmatrix}(1, n, 1, n); \]
\[ R_{03} = \text{dmatrix}(1, n, 1, n); \]
\[ T_{03} = \text{dmatrix}(1, n, 1, n); \]
\[ R_{30} = \text{dmatrix}(1, n, 1, n); \]
\[ T_{30} = \text{dmatrix}(1, n, 1, n); \]
\[ R_{34} = \text{dmatrix}(1, n, 1, n); \]
\[ T_{34} = \text{dmatrix}(1, n, 1, n); \]
\[ R_{63} = \text{dmatrix}(1, n, 1, n); \]
\[ T_{63} = \text{dmatrix}(1, n, 1, n); \]
\[ R_{36} = \text{dmatrix}(1, n, 1, n); \]
\[ T_{36} = \text{dmatrix}(1, n, 1, n); \]
\[ \text{Lup} = \text{dmatrix}(1, n, 1, n); \]
\[ \text{Ldown} = \text{dmatrix}(1, n, 1, n); \]

This code is used in section 55.

\begin{itemize}
  \item \textbf{AD\_method\_type}: \texttt{struct}, §9.
  \item \textbf{Add\_Bottom}: \texttt{void}, §29.
  \item \textbf{Add\_Top}: \texttt{void}, §127.
  \item \textbf{BOTTOM\_BOUNDARY} = 1, §116.
  \item \textbf{dmatrix}, <\texttt{nr\_util.h}>
  \item \textbf{dvector}, <\texttt{nr\_util.h}>
  \item \textbf{Flux\_Fluence}: \texttt{void}, §54.
  \item \texttt{free\_dmatrix}, <\texttt{nr\_util.h}>
  \item \texttt{free\_dvector}, <\texttt{nr\_util.h}>
  \item \textbf{Init\_Boundary}: \texttt{void}, §116.
  \item \textbf{n}: \texttt{int}, §54.
  \item \texttt{quad\_pts}: \texttt{int}, §9.
  \item \textbf{RT\_Matrices}: \texttt{void}, §28.
  \item \texttt{slab}: \texttt{struct AD\_slab\_type*}, §54.
  \item \textbf{TOP\_BOUNDARY} = 0, §116.
  \item \texttt{zmax}: \texttt{double}, §54.
  \item \texttt{zmin}: \texttt{double}, §54.
\end{itemize}
60. (Find radiance at each depth 60) \(\equiv\)
\[
\text{slab}_b = (z_{\text{max}} - z_{\text{min}})/\text{intervals} \times i;
\]
\[
\text{RTMatrices}(n, \text{slab}, \text{&method}, R_{23}, T_{23});
\]
\[
\text{Add}(n, R_{02}, R_{20}, T_{02}, T_{20}, R_{23}, R_{23}, T_{23}, T_{23}, R_{03}, T_{03}, T_{30});
\]
\[
\text{slab}_b = (z_{\text{max}} - z_{\text{min}}) - \text{slab}_b;
\]
\[
\text{RTMatrices}(n, \text{slab}, \text{&method}, R_{34}, T_{34});
\]
\[
\text{Add}(n, R_{34}, R_{34}, T_{34}, T_{34}, R_{46}, R_{46}, T_{46}, T_{46}, R_{36}, R_{63}, T_{36}, T_{63});
\]
\[
\text{Between}(n, R_{03}, R_{30}, T_{03}, T_{30}, R_{30}, R_{36}, R_{63}, T_{36}, T_{63}, L_{up}, L_{down});
\]
This code is used in section 55.

61. (Calculate Fluence and Flux 61) \(\equiv\)
\[
\text{UFU} \text{and UF1}(n, \text{slab−n_slab}, L_{up}, L_{down}, \&\text{UFU}, \&\text{UF1});
\]
\[
\text{UF1_array}[i] = \text{UF1};
\]
\[
\text{UFU_array}[i] = \text{UFU};
\]
\[
\text{flx_down} = 0.0;
\]
\[
\text{flx_up} = 0.0;
\]
\[
\text{for} \ (j = 1; \ j \leq n; \ j++) \ \{
\]
\[
\text{flx_down} += \text{twoaw}[j] \times L_{down}[j][n];
\]
\[
\text{flx_up} += \text{twoaw}[j] \times L_{up}[j][n];
\]
\[
\}
\]
\[
\text{flx_down}[i] = \text{flx_down} \times \text{slab−n_slab} \times \text{slab−n_slab};
\]
\[
\text{flx_up}[i] = \text{flx_up} \times \text{slab−n_slab} \times \text{slab−n_slab};
\]
This code is used in section 55.

62. (Free all those intermediate matrices 62) \(\equiv\)
\[
\text{free_dmatrix}(R_{02}, 1, n, 1, n);
\]
\[
\text{free_dmatrix}(T_{02}, 1, n, 1, n);
\]
\[
\text{free_dmatrix}(R_{20}, 1, n, 1, n);
\]
\[
\text{free_dmatrix}(T_{20}, 1, n, 1, n);
\]
\[
\text{free_dmatrix}(R_{23}, 1, n, 1, n);
\]
\[
\text{free_dmatrix}(T_{23}, 1, n, 1, n);
\]
\[
\text{free_dmatrix}(R_{30}, 1, n, 1, n);
\]
\[
\text{free_dmatrix}(T_{30}, 1, n, 1, n);
\]
\[
\text{free_dmatrix}(R_{34}, 1, n, 1, n);
\]
\[
\text{free_dmatrix}(T_{34}, 1, n, 1, n);
\]
\[
\text{free_dmatrix}(R_{46}, 1, n, 1, n);
\]
\[
\text{free_dmatrix}(T_{46}, 1, n, 1, n);
\]
\[
\text{free_dmatrix}(L_{up}, 1, n, 1, n);
\]
\[
\text{free_dmatrix}(L_{down}, 1, n, 1, n);
\]
This code is used in section 55.
63. **AD Start.** This has the routines for forming the initial matrix to start off an adding-doubling calculation.

Added printing of intermediate results for Martin Hammer.

```c
#include <math.h>
#include <float.h>
#include <stdio.h>
#include <math.h>
#include "nr_util.h"
#include "nr_gaulg.h"
#include "ad_start.h"
#include "ad_radau.h"
#include "ad_phase.h"
#include "ad_matrx.h"
#include "ad_frsnl.h"
#include "ad_globl.h"
```

Added printing of intermediate results for Martin Hammer.

```c
Add: void, §94.
b: double **, §56.
Between: void, §106.
Choose_Method: void, §71.
flux_down: double *, §54.
flux_up: double, §56.
free_dmatrix, <nr_util.h>,
Get_JGLayer: static void, §75.
Get_Start_Depth: double, §66.
i: int, §56.
Inst_Layer: void, §87.
intervals: int, §54.
j: int, §56.
Ldown: double **, §56.
Lup: double **, §56.
method: struct AD_method_type, §56.
n: int, §54.

n_slab: double, §18.
Quadrature: void, §69.
R02: double **, §56.
R03: double **, §56.
R20: double **, §56.
R21: double **, §56.
R30: double **, §56.
R34: double **, §56.
R35: double **, §56.
R46: double **, §56.
R53: double **, §56.
R64: double **, §56.
RT_Matrices: void, §28.
slab: struct AD_slab_type *, §54.
U1: double, §56.
U1_array: double *, §54.
UFU: double, §56.
UFU_and_U1: void, §18.
UFU_array: double *, §54.
max: double, §54.
min: double, §54.
```
65. **Basic routines.**

This file contains the three procedures which must be called before any doubling may take place. They should be called in the following order:

- **Choose Method** — to fill the method record
- **Quadrature** — to calculate the quad angles and weights
code to initialize angle, weight, and twoaw
- **Init Layer** — to calculate the thin layer R and T
- **Double Until** — to obtain R and T for the desired thickness

66. **Get Start Depth** selects the best minimum starting thickness to start the doubling process. The criterion is based on an assessment of the (1) round-off error, (2) the angular initialization error, and (3) the thickness initialization error. Wiscombe concluded that an optimal starting thickness depends on the smallest quadrature angle, and recommends that when either the infinitesimal generator or diamond initialization methods are used then the initial thickness is optimal when type 2 and 3 errors are comparable, or when

\[ d \approx \mu \]

Note that round-off is important when the starting thickness is less than \(1 \cdot 10^{-4}\) for diamond initialization and less than \(1 \cdot 10^{-8}\) for infinitesimal generator initialization assuming about 14 significant digits of accuracy.

Since the final thickness is determined by repeated doubling, the starting thickness is found by dividing by 2 until the starting thickness is less than \(\mu\). Also we make checks for a layer with zero thickness and one that infinitely thick.

\[
\text{Prototype for Get Start Depth} 66) \equiv \\
\text{double Get Start Depth (double mu, double d)}
\]

This code is used in sections 64 and 67.

67. **Definition for Get Start Depth 67** \(\equiv\)

\[
\text{(Prototype for Get Start Depth 66)}
\]

\[
\text{if (d} \leq 0) \text{ return 0.0;}
\]

\[
\text{if (d} \equiv \text{HUGE_VAL) return (mu/2.0);}
\]

\[
\text{while (d} > \text{mu) d /= 2;}
\]

\[
\text{return d;}
\]

This code is used in section 63.
68. **Quadrature.**

69. This returns the quadrature angles using Radau quadrature over the interval 0 to 1 if there is no critical angle for total internal reflection in the slab. If there is a critical angle whose cosine is $\mu_c$ then Gaussian quadrature points are chosen from 0 to $\mu_c$ and Radau quadrature points over the interval $\mu_c$ to 1.

(Prototype for Quadrature 69) $\equiv$

```c
void Quadrature(int n, double n_slab, double *x, double *w)
```

This code is used in sections 64 and 70.

70. (Definition for Quadrature 70) $\equiv$

(Prototype for Quadrature 69)

```c
{
    int i, nby2;
    double *x1, *w1;
    double mu_c;
    if (n_slab == 1) {
        Radau(0.0, 1.0, x, w, n);
        return;
    }
    mu_c = Cos_Critical_Angle(n_slab, 1.0);
    nby2 = n/2;
    gauleg(0.0, mu_c, x, w, nby2);
    x1 = dvector(1, nby2);
    w1 = dvector(1, nby2);
    Radau(mu_c, 1.0, x1, w1, nby2);
    for (i = 1; i <= nby2; i++) {
        x[nby2 + i] = x1[i];
        w[nby2 + i] = w1[i];
    }
    free_dvector(x1, 1, nby2);
    free_dvector(w1, 1, nby2);
}
```

This code is used in section 63.

---

angle: double [], §10.
Choose_Method: void, §71.
Cos_Critical_Angle: double, §137.
Double_Until: void, §102.
dvector, <nr_util.h>.
free_dvector, <nr_util.h>.
Rad: double **, §46.
Radau: void, §211.
T: double **, §46.
twoaw: double [], §10.
weight: double [], §10.
71. Choose Method fills the method structure with correct values for $a_{\text{calc}}$, $b_{\text{calc}}$, $g_{\text{calc}}$, and $b_{\text{thinnest}}$ based on the delta-M method. Furthermore, the quadrature angles and weights are also calculated. Before calling this routine $\text{method.quad}_\text{pts}$ must be set to some multiple of 2. If this routine is not called then it is up to you to
1. to fill the method record appropriately
2. call Quadrature
3. fill global arrays $\text{angle}$, $\text{weight}$, and $\text{twoaw}$
4. determine the thickness of the thinnest layer

Prototype for Choose Method 71 ≡

\begin{verbatim}
void Choose_Method(struct AD_slab_type slab, struct AD_method_type *method)
\end{verbatim}

This code is used in sections 64 and 72.

72. Definition for Choose Method 72 ≡

Prototype for Choose Method 71

\begin{verbatim}
{ double af;
  int i, n;
  n = method->quad._pts;
  af = pow(slab.g, n) * slab.a;
  method->a_calc = (slab.a - af) / (1 - af);
  method->b_calc = (1 - af) * slab.b;
  method->g_calc = slab.g;
  Quadrature(n, slab.n_slab, angle, weight);
  for (i = 1; i <= n; i++) twoaw[i] = 2 * angle[i] * weight[i];
  method->b_thinnest = Get_Start_Depth(angle[1], method->b_calc);
}
\end{verbatim}

This code is used in section 63.
**73. Initialization.**

The basic idea behind diamond initialization is to rewrite the time-independent, one-dimensional, azimuthally averaged, radiative transport equation

\[ \nu \frac{\partial L(\tau, \nu)}{\partial \tau} + L(\tau, \nu) = \frac{a}{2} \int_{-1}^{1} h(\nu, \nu') L(\tau, \nu') d\nu' \]

in a discrete form as

\[ \pm \nu_i \frac{\partial L(\tau, \pm \nu_i)}{\partial \tau} + L(\tau, \pm \nu_i) = \frac{a}{2} \sum_{j=1}^{M} w_j \left[ h(\nu_i, \nu_j) L(\tau, \pm \nu_i) + h(\nu_i, -\nu_j) L(\tau, \mp \nu_i) \right] \]

When this equation is integrated over a thin layer from \( \tau_0^* \) to \( \tau_1^* \) then get

\[ \pm \nu_i \left[ L(\tau_1^*, \pm \nu_i) - L(\tau_0^*, \pm \nu_i) \right] + dL_{1/2}(\pm \nu_i) \]

\[ = \frac{a}{2} \sum_{j=1}^{M} w_j d \left[ h(\nu_i, \nu_j) L_{1/2}(\pm \nu_i) + h(\nu_i, -\nu_j) L_{1/2}(\mp \nu_i) \right] \]

where \( d = \tau_1^* - \tau_0^* \). The integrated radiance \( L_{1/2}(\nu) \) is

\[ L_{1/2}(\nu) \equiv \frac{1}{\Delta \tau} \int_{\tau_0^*}^{\tau_1^*} L(\tau, \nu) d\tau \]

Exactly how this integral is approximated determines the type of initialization. Wiscombe evaluated a number of initialization methods and found two that were useful. These are the infinitesimal generator and the diamond methods. The infinitesimal generator initialization makes the approximation

\[ L_{1/2}(-\nu) = L(\tau_1^*, -\nu) \quad L_{1/2}(\nu) = L(\tau_0^*, \nu) \]

and the diamond initialization assumes

\[ L_{1/2}(\nu) = \frac{1}{2} \left[ L(\tau_0^*, \nu) + L(\tau_1^*, \nu) \right] \]
74. Infinitesimal Generator Initialization.

75. \( \text{Get}_\text{IGI}\text{Layer} \) generates the starting matrix with the infinitesimal generator method. The accuracy is \( O(d) \) and assumes that the average irradiance upwards is equal to that travelling downwards at the top and the average radiance upwards equals that moving upwards from the bottom.

\[
L_{1/2}(-\nu) = L(\tau^*_1, -\nu) \quad L_{1/2}(\nu) = L(\tau^*_0, \nu)
\]

After manipulation, Wiscombe obtains these basic formulas for the infinitesimal generator method,

\[
R = \hat{R}d \quad T = I - \hat{T}d
\]

where \( d \) is the optical thickness of the layer and \( I \) is the identity matrix. The values for \( \hat{R} \) and \( \hat{T} \) are given by

\[
\hat{R}_{ij} = \frac{a}{2\mu_i} h^{+\nu}_{ij} w_j \quad \hat{T}_{ij} = \frac{\delta_{ij}}{\mu_i} - \frac{a}{2\mu_i} h^{++}_{ij} w_j
\]

and

\[
R_{ij} = \frac{ad}{4\mu_i\mu_j} h_{ij}^{-\nu} \quad T_{ij} = \frac{ad}{2\mu_i} h_{ij}^{++} + \delta_{ij} \left[ 1 - \frac{d}{\mu_i} \right]
\]

This would be fine, but the way that the reflection and transmission matrices are set-up requires that each we multiply each matrix on the right by \( 1/(2\mu_j w_j) \). Putting things together we get

\[
R_{ij} = \frac{ad}{4\mu_i\mu_j} h_{ij}^{-\nu}
\]

and

\[
T_{ij} = \frac{ad}{4\mu_i\mu_j} h_{ij}^{++} + \delta_{ij} \left[ 1 - \frac{d}{\mu_i} \right]
\]

(Definition for \( \text{Get}_\text{IGI}\text{Layer} \) 75)

```c
static void Get_IGLayer(struct AD_method_type method, double **h, double **R, double **T) {
    int i, j, n;
    double a, c, d, temp;
    a = method.a_calc;
    d = method.b_thinest;
    n = method.quad_pts;
    for (j = 1; j <= n; j++) {
        temp = a * d/4/angle[j];
        for (i = 1; i <= n; i++) {
            c = temp/angle[i];
            ```}
\[ R[i][j] = c \times h[i][j]; \]
\[ T[i][j] = c \times h[i][j]; \]
\}
\[ T[j][j] += (1 - d/angle[j])/twoaw[j]; \]
\}

This code is used in section 63.
76. Diamond Initialization.

It should be noted up front that the implementation contained herein is somewhat cryptic. Much of the complexity comes from using the tricks in the appendix A of Wiscombe's paper ("On initialization, error and flux conservation in the doubling method.") After spending a whole day tracking down a small error in the calculation of the reflection matrix, I will spend a few moments trying to improve the documentation for this whole section. It should be apparent that this is no substitute for reading the paper.

The advantage of the diamond initialization method is that its accuracy is of the order of the square of the optical thickness \( O(d^2) \). This means that much thicker starting layers and retain good starting accuracy. This reduces the number of doubling steps that are required. However, if the layer thickness is too thin then the accuracy gets much worse because errors in the numerical precision start to affect the results.

Get_Diamond_Layer generates the starting matrix with the diamond method. This implies that the integral can be replaced by a simple average of the radiances at the top and bottom of the layer,

\[
L_{1/2}(\nu) = \frac{1}{2} [L(\tau_0^*, \nu) + L(\tau_1^*, \nu)]
\]

(Definition for Get_Diamond_Layer 76) \( \equiv \)

```c
static void Get_Diamond_Layer(struct AD_method_type method, double **h, double **R, double **T)
{
    (Local variables and initialization 84)
    (Find r and t 77)
    (Find C = r/(1 + t) 78)
    (Find G = 0.5(1 + t - Cr) 79)
    (print r, t, and g for Martin Hammer 80)
    (Calculate R and T 81)
    (Free up memory 85)
}
```

This code is used in section 63.

77. This diamond initialization method uses the same \( \hat{R} \) and \( \hat{T} \) as was used for infinitesimal generator method. However, we want to form the \( r \) and \( t \)

\[
r = \frac{d}{2} \hat{R} \quad t = \frac{d}{2} \hat{T}
\]

Recall that

\[
\hat{R}_{ij} = \frac{a}{2\mu_i} h_{ij}^{++} w_j \quad \hat{T}_{ij} = \frac{\delta_{ij}}{\mu_i} - \frac{a}{2\mu_i} h_{ij}^{++} w_j
\]

therefore

\[
r_{ij} = \frac{a w_j}{4\mu_i} h_{ij}^{+-} \quad t_{ij} = \delta_{ij} \frac{d}{2\mu_i} - \frac{a w_j}{4\mu_i} h_{ij}^{++}
\]

If you happen to be wondering why right multiplication by \( 1/(2\mu_j w_j) \) is not needed, you would be a thinking sort of person. Division by \( 1/(2\mu_j w_j) \) is not needed until the final values for \( R \) and \( T \) are formed.
Adding-Doubling (Version 1.5): DIAMOND INITIALIZATION

⟨Find \( r \) and \( t \) ⟩ \( r \equiv \frac{a \times d \times \text{weight}[j]/4}{\text{for} (j = 1; j \leq n; j++) \{ \text{temp} = a \times d \times \text{weight}[j]/4; \text{for} (i = 1; i \leq n; i++) \{ \text{c} = \text{temp}/\text{angle}[i]; \text{R}[i][j] = \text{c} \times \text{h}[i][j]; \text{T}[i][j] = -\text{c} \times \text{h}[i][j]; \}} \text{T}[j][j] += d/(2 \times \text{angle}[j]); \}} \text{This code is used in section 76.} \)

78. Wiscombe points out (in Appendix A), that the matrix inversions can be avoided by noting that if we want \( C \) from the combination

\[ C = r(I + t)^{-1} \]

then one needs only solve the system

\[ (I + t)^T C T = r T \]

for \( C \). This is done in the routine \texttt{Left\_Inverse\_Multiply}. We just need to create \( A = I + T \) and fire it off to \texttt{Left\_Inverse\_Multiply}. Actually, Wiscombe goes on to suggest a faster method that takes advantage of the column oriented structure of storage on the computer. Since we are using the Numerical Recipes scheme, I don’t think that his refinement will prove faster because it involves more multiplications and divisions. (Actually, that improvement was exactly what the bug in the program was. I included the required multiplications and voilà! It worked.)

\langle Find \( C = r/(1 + t) \rangle \equiv \text{for} (i = 1; i \leq n; i++) \{ \text{for} (j = 1; j \leq n; j++) \text{A}[i][j] = \text{T}[i][j]; \text{A}[i][i] += 1.0; \} \text{Left\_Inverse\_Multiply}(n, A, R, C); \text{This code is used in section 76.} \)

\( a: \text{double}, \S 75. \)
\( A: \text{double **}, \S 206. \)
\( \text{AD\_method\_type: struct}, \S 9. \)
\( \text{angle: double [],} \S 10. \)
\( c: \text{double}, \S 75. \)
\( C: \text{double **}, \S 206. \)
\( d: \text{double}, \S 75. \)
\( g: \text{double}, \S 8. \)
\( G: \text{double **}, \S 84. \)
\( i: \text{int}, \S 75. \)
\( j: \text{int}, \S 75. \)
\( \text{Left\_Inverse\_Multiply: void}, \S 204. \)
\( n: \text{int}, \S 75. \)
\( r: \text{double **}, \S 14. \)
\( t: \text{double **}, \S 14. \)
\( \text{temp: double}, \S 75. \)
\( \text{weight: double []}, \S 10. \)
79. Here the matrix

\[ G = \frac{1}{2}(I + t - Cr) \]

is formed.

(Find \( G = 0.5(1 + t - Cr) \) 79) \[ \equiv \]

\textit{Matrix} \_\textit{Multiply}(n, C, R, G);

\textbf{for} \( (i = 1; \ i \leq n; \ i++) \) \{ \n\textbf{for} \( (j = 1; \ j \leq n; \ j++) \) \( G[i][j] = (T[i][j] - G[i][j])/2; \)
\( G[i][i] += 0.5; \}
\}

This code is used in section 76.

80. To print intermediate results for Chapter 4 of AJ’s book, then it is necessary to print things from within \textit{Get-Diamond-Layer}. Martin Hammer requested that I provide these results. Since this is the only time that they are of interest, they are only printed when both the compiler define \texttt{MARTIN_HAMMER} is defined, and when the variable \texttt{Martin-Hammer \neq 0}.

(\textbf{print} \( r, \ t, \ \text{and} \ g \textbf{\ for Martin Hammer 80}) \equiv \]
\textbf{#ifndef MARTIN_HAMMER}
\{ \n\textbf{double} **Ginv, **G2;
\textbf{if} (\texttt{Martin-Hammer} \neq 0) \{ \n\textbf{printf}("A from equation 5.55\n");
\textbf{wrmatrix}(n, T);
\textbf{printf}("B from equation 5.55\n");
\textbf{wrmatrix}(n, R);
\textbf{Ginv = dmatrix}(1, n, 1, n);
\textbf{G2 = dmatrix}(1, n, 1, n);
\textbf{for} \( (i = 1; \ i \leq n; \ i++) \) \{ \n\textbf{for} \( (j = 1; \ j \leq n; \ j++) \) \{ \n\textbf{G2}[i][j] = G[i][j] \* 2.0;
\}
\}\textbf{Matrix}_\textit{Inverse}(n, G2, Ginv);
\textbf{printf}("Inverse of G from equation 5.56\n");
\textbf{wrmatrix}(n, G2);
\textbf{printf}("G from equation 5.56\n");
\textbf{wrmatrix}(n, Ginv);
\textbf{free_matrix}(Ginv, 1, n, 1, n);
\textbf{free_matrix}(G2, 1, n, 1, n);
\}
\textbf{#endif}

This code is used in section 76.
Adding-Doubling (Version 1.5): DIAMOND INITIALIZATION

C: double **, §206.
dmatrix, <nr_util.h>.
free_matrix, <nr_util.h>.
g: double, §8.
G: double **, §84.
Get_Diamond_Layer: static void, §76.

r: int, §75.
j: int, §75.
Martin_Hammer: int, §10.
MARTIN_HAMMER = 1, §7.
Matrix_Multiply: void, §177.
n: int, §75.

printf, <stdio.h>.
r: double **, §14.
R: double **, §76.
t: double **, §14.
T: double **, §76.
wrmatrix: void, §20.
81. Now we get the part that I really don’t understand. However, I know that this works. There are a couple of confusing transposes and bizarre incorporation of twoaw, but everything hangs together. Now since the single layer matrices R and T are the solutions to the systems of equations

\[ GR = C \quad G(t + I) = I \]

We do the little shuffle and only find the LU decomposition of G once and use it to find both R and T + 1.

(Calculate R and T s1) \(\equiv\)

\[ \text{Transpose} \_\text{Matrix} (n, G); \]
\[ \text{Decomp} (n, G, \& \text{condition}, \& \text{ipvt}); \]
\[ \text{if} (\text{condition} \equiv 1 \cdot 10^{32}) \text{ nrerror} ("\text{Singular Matrix...failed in diamond_init}"); \]

for \((i = 1; i \leq n; i++) \) \{ 
  \(\text{Solve for row of } R \) s2 \}
\(\text{Solve for row of } T \) s3 \}

\#ifdef MARTIN_HAMMER

\{ \n  \text{double} **\text{T2}, **\text{Ginv}; \n  \text{if} (\text{Martin_Hammer} \equiv 5) \{ \n      \text{T2} = \text{dmatrix}(1, n, 1, n); \n      \text{Ginv} = \text{dmatrix}(1, n, 1, n); \n      \text{Copy} \_\text{Matrix} (n, T, \text{T2}); \n      \text{for} (i = 1; i \leq n; i++) \{ \n          \text{T2}[i][i] += 1/twoaw[i]; \n      \} \n      \text{for} (i = 1; i \leq n; i++) \{ \n          \text{for} (j = 1; j \leq n; j++) \{ \n              \text{T2}[i][j] *= twoaw[j] * 0.5; \n          \} \n      \} \n      \text{printf} ("G=(T−1)/2\text{from equation }5.55\text{\n}""); \n      \text{wrmatrix} (n, \text{T2}); \n      \text{Matrix} \_\text{Inverse} (n, \text{T2}, \text{Ginv}); \n      \text{printf} ("1/G\text{\n}""); \n      \text{wrmatrix} (n, \text{Ginv}); \n      \text{free} \_\text{matrix} (\text{T2}, 1, n, 1, n); \n      \text{free} \_\text{matrix} (\text{Ginv}, 1, n, 1, n); \n  \} \n\} \n\#endif

This code is used in section 76.

82. We use the decomposed form of G to find R. Since G is now the LU decomposition of \(GT\), we must pass rows of the \(C\) to \(\text{Solve}\) and get rows back. Note the finess with

\[ \text{work}_j = C_j \frac{a_j w_j}{a_i w_i} \]
To get everything in the right place. This is discussed in Wiscombe’s appendix. Finally, we dutifully put these values back in $R$ and divide by $1/(2\mu_j w_j)$ so that $R$ will be symmetric and have the proper form.

\[
\text{(Solve for row of } R \text{ s2) } \equiv \\
\text{for (} j = 1; \ j \leq n; \ j++ \text{) } \text{work}[j] = C[j][i] \ast \text{twoaw}[j]/\text{twoaw}[i]; \\
\text{Solve} (n, G, \text{work, ipvt}); \\
\text{for (} j = 1; \ j \leq n; \ j++ \text{) } R[i][j] = \text{work}[j]/\text{twoaw}[j];
\]

This code is used in section 81.

83. We again use the decomposed form of $G$ to find $T$. This is much simpler since we only need to pass rows of the identity matrix back and forth. We again carefully put these values back in $T$ and divide by $1/(2\mu_j w_j)$ so that $T$ is properly formed. Oh yes, we can’t forget to subtract the identity matrix!

\[
\text{(Solve for row of } T \text{ s3) } \equiv \\
\text{for (} j = 1; \ j \leq n; \ j++ \text{) } \text{work}[j] = 0; \\
\text{work}[i] = 1.0; \\
\text{Solve} (n, G, \text{work, ipvt}); \\
\text{for (} j = 1; \ j \leq n; \ j++ \text{) } T[i][j] = \text{work}[j]/\text{twoaw}[j]; \\
T[i][i] -= 1.0/\text{twoaw}[i]; \quad /\!* \text{Subtract Identity Matrix } */
\]

This code is used in section 81.
84. Pretty standard stuff here. Allocate memory and print a warning if the thickness is too small.

```c
(int) local variables and initialization 84) ≡
  int i, j, n;
double **A, **G, **C;
double a, c, d, temp;
double *work;
double condition;
int *ipvt;
d = method.b.thinnest;
a = method.a.calc;
n = method.quad.pnts;
A = dmatrix(1, n, 1, n);
G = dmatrix(1, n, 1, n);
C = dmatrix(1, n, 1, n);
work = dvector(1, n);
ipvt = ivector(1, n);
if (d < 1 \cdot 10^{-4}) fprintf(stderr, "****Roundoff error is a problem--Use IGI_method\n")
```

This code is used in section 76.

85. (Free up memory 85) ≡
```c
free_dvector(work, 1, n);
free_ivector(ipvt, 1, n);
free_dmatrix(A, 1, n, 1, n);
free_dmatrix(G, 1, n, 1, n);
free_dmatrix(C, 1, n, 1, n);
```

This code is used in section 76.
86. Layer Initialization.

87. \textit{Init\_Layer} returns reflection and transmission matrices for a thin layer. Space must previously been allocated for \( R \) and \( T \).

\[
\langle \text{Prototype for } \text{Init\_Layer } \rangle \equiv \\
\text{void } \text{Init\_Layer (struct AD\_slab\_type slab, struct AD\_method\_type method, double } \star\star R, \star\star T) \\
\]

This code is used in sections 64 and 88.

88. \[
\langle \text{Definition for } \text{Init\_Layer } \rangle \equiv \\
\langle \text{Prototype for } \text{Init\_Layer } \rangle \\
\{
\quad \text{double } \star\star h; \\
\quad \text{int } n; \\
\quad n = \text{method}\_\text{quad}\_\text{pts}; \\
\quad \text{if (slab.} b \leq 0) \\
\quad \\
\quad \quad \text{Zero\_Layer (n, R, T);} \\
\quad \quad \text{return}; \\
\quad \}
\]
\[
\quad h = \text{dmatrix (} -n, -n, -n, n); \\
\quad \text{Get}\_\text{Phi (n, slab.} \text{phase\_function, method.g\_calc, h);} \\
\quad \text{if (method.b\_thinnest < } 1 \cdot 10^{-4} \lor \text{method.b\_thinnest < 0.09 } \times \text{angle[1])} \\
\quad \quad \text{Get}\_\text{IGI\_Layer (method, h, R, T);} \\
\quad \text{else } \text{Get}\_\text{Diamond\_Layer (method, h, R, T);} \\
\quad \text{free\_dmatrix (h, } -n, -n, -n, n); \\
\}
\]

This code is used in section 63.

\begin{itemize}
\item \textit{a\_calc}: double, §9.
\item \textit{AD\_method\_type}: struct, §9.
\item \textit{AD\_slab\_type}: struct, §8.
\item \textit{angle}: double [], §10.
\item \textit{b\_thinnest}: double, §9.
\item \textit{dmatrix}: \textit{dvector}, §9.
\item \textit{free\_dmatrix}, \textit{free\_dvector}, §239.
\item \textit{fprintf}, \textit{stdout.h}.
\item \textit{Get}\_\text{Phi}: void, §239.
\item \textit{get\_printf}, \textit{stdio.h}.
\item \textit{Get}\_\text{Diamond\_Layer}: static void, §76.
\item \textit{Get}\_\text{IGI\_Layer}: static void, §75.
\item \textit{g\_calc}: double, §9.
\item \textit{free\_vector}, \textit{nr\_util.h}.
\item \textit{phase\_function}: int, §8.
\item \textit{stderr}, \textit{stdio.h}.
\item \textit{Zero\_Layer}: void, §14.
\item \textit{printf}, \textit{stdio.h}.
\item \textit{vector}, \textit{nr\_util.h}.
\item \textit{method}: struct AD\_method\_type, §76.
\item \textit{quad\_pts}: int, §9.
\end{itemize}
89. **AD Double.** This has the routines needed to add layers together in various combinations.

\[
\langle \text{ad_doubl.c 89} \rangle \equiv
\]

```c
#include <math.h>
#include <float.h>
#include "nr_util.h"
#include "ad_matrx.h"
#include "ad_globl.h"
#include "ad_doubl.h"
```

(Definition for Star_Multiply 109)
(Definition for Star_One_Minus 110)
(Definition for Basic_Add_Layers 91)
(Definition for Basic_Add_Layers_With_Sources 92)
(Definition for Add 95)
(Definition for Add_With_Sources 97)
(Definition for Add_Homogeneous 99)
(Definition for Double_Once 101)
(Definition for Double_Until 103)
(Definition for Double_Until_Infinite 105)
(Definition for Between 107)

90. \(
\langle \text{ad_doubl.h 90} \rangle \equiv
\)

```c
(Prototype for Add 94);
(Prototype for Add_With_Sources 96);
(Prototype for Add_Homogeneous 98);
(Prototype for Double_Once 100);
(Prototype for Double_Until 102);
(Prototype for Double_Until_Infinite 104);
(Prototype for Between 106);
```
Adding-Doubling (Version 1.5): BASIC ROUTINE TO ADD LAYERS WITHOUT SOURCES

Add: void, §94.
Add_Homogeneous: void, §98.
Add_With_Sources: void, §96.
Basic_Add_Layers: static void, §91.

Basic_Add_Layers_With_Sources:
static void, §92.

Between: void, §106.
Double_Once: void, §100.
Double_Until: void, §102.

Double_Until_Infinite: void, §104.
Star_One_Minus: static void, §110.
91. Basic Routine to Add Layers Without Sources.

The basic equations for the adding-doubling method (neglecting sources) are

\[ T_{02}^0 = T_{12}(E - R^{10}R^{12})^{-1}T_{01} \]
\[ R_{20}^0 = T_{12}(E - R^{10}R^{12})^{-1}R^{10}T_{21} + R_{21} \]
\[ T_{20}^0 = T_{10}(E - R^{12}R^{10})^{-1}T_{21} \]
\[ R_{02}^0 = T_{10}(E - R^{12}R^{10})^{-1}R^{12}T_{01} + R_{01} \]

Upon examination it is clear that the two sets of equations have the same form. Therefore if I implement the first two equations, then the second set can be obtained by suitable switching of the parameters. Furthermore, these equations assume some of the multiplications are star multiplications. Explicitly,

\[ T_{02}^0 = T_{12}(E - R^{10} \star R^{12})^{-1}T_{01} \]

and

\[ R_{20}^0 = T_{12}(E - R^{10} \star R^{12})^{-1}R^{10} \star T_{21} + R_{21} \]

where the identity matrix \( E \) is then

\[ E_{ij} = \frac{1}{2\mu_i w_i} \delta_{ij} \]

where \( \delta_{ij} \) is the usual Kronecker delta. It is noteworthy that if say \( R_{10} \equiv 0 \), then \( E^{-1} \equiv c \) and so

\[ T_{02}^0 = T_{12} \star T_{01} = T_{12} \star T_{01} \]

One goal of this routine was to make it efficient and easy to use. It is possible to call this routine with the same pointer for all the different reflection matrices and the pointer for the transmission matrices may be the same also. (The reflection and transmission pointers may need to be distinct. The temporary memory pointers \( a \) and \( b \) must be distinct from each other and distinct from the reflection and transmission matrices.)

Note: it should be possible to eliminate the need for the matrix \( b \) if Inverse_Multiply could be called with an argument list like Inverse_Multiply\( (n, A, B, A) \). A quick glance at the code suggests that this would just force the allocation of the matrix into the Inverse_Multiply routine and no net gain would result.

(Definition for Basic_Add_Layers 91) \equiv

```c
static void Basic_Add_Layers(int n, double **R10, double **T01, double **R12, double **R21, double **T12, double **T21, double **R20, double **T02, double **a, double **b)
{
    Star_Multiply(n, R10, R12, a); /* a = R_{10} \star R_{12} */
    Star_One_Minus(n, a);         /* a = E - R_{10} \star R_{12} */
    Left_Inverse_Multiply(n, a, T12, b); /* b = T_{12}(E - R^{10}R^{12})^{-1} */
    Matrix_Multiply(n, b, R10, a); /* a = T_{12}(E - R^{10} \star R^{12})^{-1}R_{10} */
    Star_Multiply(n, a, T21, a); /* a = T_{12}(E - R^{10} \star R^{12})^{-1}R^{10} \star T_{21} */
    Matrix_Sum(n, R21, a, R20);
    Copy_Matrix(n, T01, a);
    Matrix_Multiply(n, b, a, T02);
}
```

This code is used in section 89.
A: `double **`, §84.
B: `double *`, §198.
`Copy_Matrix`: void, §160.
`Inverse_Multiply`: ???, §0.
`Left_Inverse_Multiply`: void, §204.
`Matrix_Multiply`: void, §177.
`Matrix_Sum`: void, §172.
`Star_One_Minus`: static void, §110.
92. Basic Routine to Add Layers With Sources.

The adding-doubling equations including source terms \( J \) are identical to those given above for the reflection and transmission. The only difference is that the source terms must be kept track of separately according to

\[
J_{j+}^{02} = J_{j+}^{12} + T^{12}(E - R^{10}R^{12})^{-1}(J_{j+}^{01} + R^{10}J^{21})
\]

and

\[
J_{j+}^{20} = J_{j+}^{10} + T^{10}(E - R^{12}R^{10})^{-1}(J_{j+}^{21} + R^{12}J^{01})
\]

where the + subscript indicates the downward direction and − indicates the upward direction. Note that these subscripts are not needed. Thus we have

\[
J_{j+}^{02} = J_{j+}^{12} + T^{12}(E - R^{10}R^{12})^{-1}(J_{j+}^{01} + R^{10}J^{21})
\]

and

\[
J_{j+}^{20} = J_{j+}^{10} + T^{10}(E - R^{12}R^{10})^{-1}(J_{j+}^{21} + R^{12}J^{01})
\]

Again, it is apparent that clever switching of the arguments requires that only one set of equations needs to be calculated. These equations assume some of the multiplications are star multiplications. Explicitly,

\[
J_{j+}^{02} = J_{j+}^{12} + T^{12}(E - R^{10} \ast R^{12})^{-1}(J_{j+}^{01} + R^{10} \ast J^{21})
\]

(Definiton for *Basic_Add_Layers_With_Sources* 92) ≡

```
static void Basic_Add_Layers_With_Sources(int n, double **R10, double **T01, double **R12, double **R21, double **T12, double **T21, double **R20, double **T02, double **J01, double **J12, double **J21, double **J02, double **a, double **b)
{
    Star_Multiply(n, R10, R12, a);  /* a = R^{10} \ast R^{12} */
    Star_One_Minus(n, a);          /* a = E - R^{10} \ast R^{12} */
    Left_Inverse_Multiply(n, a, T12, b); /* b = T^{12}(E - R^{10}R^{12})^{-1} */
    Matrix_Multiply(n, b, R10, a); /* a = T^{12}(E - R^{10} \ast R^{12})^{-1}R^{10} */
    Matrix_Multiply(n, a, T21, a); /* a = T^{12}(E - R^{10} \ast R^{12})^{-1}R^{10} \ast T^{21} */
    Matrix_Sum(n, R21, a, R20);
    Copy_Matrix(n, T01, a);
    Matrix_Multiply(n, a, T02);
    Star_Multiply(n, R10, J21, a); /* a = R^{10} \ast J^{21} */
    Matrix_Sum(n, J01, a, a);      /* a = J^{01} + R^{10} \ast J^{21} */
    Matrix_Multiply(n, a, J02);    /* a = J^{01} + R^{10} \ast J^{21} */
    Matrix_Sum(n, J02, J12, J02);
}
```

This code is used in section 89.
93. Higher level routines.

94. (Prototype for Add 94) ≡

\[
\text{void Add} (\text{int } n, \text{double } **R01, \text{double } **R10, \text{double } **T01, \text{double } **T10, \text{double } **R12, \text{double } **T12, \text{double } **T21, \text{double } **R20, \text{double } **T02, \text{double } **T20)
\]

This code is used in sections 90 and 95.

95. Add returns the reflection and transmission matrices for a two different layers added together. These matrices do not have to be homogeneous. The output matrices \(R20\), \(R02\), \(T20\), and \(T02\) should be distinct from the input matrices.

(Definition for Add 95) ≡

(Prototype for Add 94)

\[
\begin{align*}
&\{ \langle \text{Allocate memory for } a \text{ and } b \rangle \} \\
&\text{Basic\_Add\_Layers} (n, R10, T01, R12, T12, T21, R20, T02, a, b); \\
&\text{Basic\_Add\_Layers} (n, R12, T21, R10, R01, T10, T01, R02, T20, a, b); \\
&\{ \langle \text{Free Memory for } a \text{ and } b \rangle \}
\end{align*}
\]

This code is used in section 89.

96. (Prototype for Add\_With\_Sources 96) ≡

\[
\text{void Add\_With\_Sources} (\text{int } n, \text{double } **R01, \text{double } **R10, \text{double } **T01, \text{double } **T10, \text{double } **R12, \text{double } **T12, \text{double } **T21, \text{double } **R20, \text{double } **T02, \text{double } **T20, \text{double } **J01, \text{double } **J10, \text{double } **J12, \text{double } **J21, \text{double } **J02, \text{double } **J20)
\]

This code is used in sections 90 and 97.

97. Add\_With\_Sources returns the reflection and transmission matrices for two different layers added together. These matrices do not have to be homogeneous. The output matrices \(R20\), \(R02\), \(T20\), \(T20\), \(J20\), and \(J02\) should be distinct from the input matrices.

(Definition for Add\_With\_Sources 97) ≡

(Prototype for Add\_With\_Sources 96)

\[
\begin{align*}
&\{ \langle \text{Allocate memory for } a \text{ and } b \rangle \} \\
&\text{Basic\_Add\_Layers\_With\_Sources} (n, R10, T01, R12, R21, T12, T21, R20, T02, J01, J12, J21, J02, a, b); \\
&\text{Basic\_Add\_Layers\_With\_Sources} (n, R12, T21, R10, R01, T10, R02, T20, J21, J10, J01, J20, a, b); \\
&\{ \langle \text{Free Memory for } a \text{ and } b \rangle \}
\end{align*}
\]

This code is used in section 89.

---

Basic\_Add\_Layers: static void, §91.  
Left\_Inverse\_Multiply: void, §204.  
Star\_Multiply: static void, §109.  
Matrix\_Multiply: void, §177.  
Star\_One\_Minus: static void, §110.  
Copy\_Matrix: void, §160.  
Matrix\_Sum: void, §172.
98. \{Prototype for \textit{Add}$_{\text{Homogeneous}}$ 98\} \equiv
\begin{verbatim}
void \textit{Add}$_{\text{Homogeneous}}$(int \textit{n}, double **R01, double **T01, double **R12, double **T12, double **R02, double **T02)
\end{verbatim}
This code is used in sections 90 and 99.

99. \{Definition for \textit{Add}$_{\text{Homogeneous}}$ 99\} \equiv
\begin{verbatim}
\{Prototype for \textit{Add}$_{\text{Homogeneous}}$ 98\}
\{
    \{Allocate memory for \(a\) and \(b\) 111\}
    \textit{Basic}$_{\text{Add}}$\textit{Layers}(n, R01, T01, R12, T12, R02, T02, a, b);
    \{Free Memory for \(a\) and \(b\) 112\}
\}
\}
This code is used in section 89.

100. This just adds a layer to itself. Couldn’t \textit{Basic}$_{\text{Add}}$\textit{Layers} be used? It would mean that there would be no restriction on the use of variables — i.e., \(R\) could be used as both a factor and as a result.

\begin{verbatim}
\{Prototype for \textit{Double}$_{\text{Once}}$ 100\} \equiv
void \textit{Double}$_{\text{Once}}$(int \textit{n}, double **\textit{R}, double **\textit{T})
\end{verbatim}
This code is used in sections 90 and 101.

101. \{Definition for \textit{Double}$_{\text{Once}}$ 101\} \equiv
\begin{verbatim}
\{Prototype for \textit{Double}$_{\text{Once}}$ 100\}
\{
    \{Allocate memory for \(a\) and \(b\) 111\}
    \textit{Basic}$_{\text{Add}}$\textit{Layers}(n, R, T, R, T, R, T, a, b);
    \{Free Memory for \(a\) and \(b\) 112\}
\}
\}
This code is used in section 89.

102. \textit{Double}$_{\text{Until}}$ and \textit{Double}$_{\text{Until Infinit}}$ are the only ones that really take advantage of the external allocation of memory from the routine. I was kind of careful to make sure that this routine terminates if bad start and end values are given i.e., \(end \neq start \cdot 2^k\). Furthermore, it should work correctly if the target thickness is infinite. I suppose that I could put some error warnings in...but right now I don’t want to take the time.

\begin{verbatim}
\{Prototype for \textit{Double}$_{\text{Until}}$ 102\} \equiv
void \textit{Double}$_{\text{Until}}$(int \textit{n}, double **\textit{r}, double **\textit{t}, double \textit{start}, double \textit{end})
\end{verbatim}
This code is used in sections 90 and 103.

103. \{Definition for \textit{Double}$_{\text{Until}}$ 103\} \equiv
\begin{verbatim}
\{Prototype for \textit{Double}$_{\text{Until}}$ 102\}
\{
    \{if (\textit{end} \equiv \text{HUGE} \text{VAL}) \{
        \textit{Double}$_{\text{Until Infinit}}$(n, r, t);
        return;
    \}
    \{Allocate memory for \(a\) and \(b\) 111\}
\}
\}
while (fabs(end - start) > 0.000001 && end > start) {
    Basic_Add_Layers(n, r, t, r, r, t, t, r, t, a, b);
    start *= 2;
} (Free Memory for a and b 112)
}

This code is used in section 89.

104. Double_Until_Infinite continues doubling until the thickness of the slab is essentially infinite. Originally I had defined infinite as a diffuse transmission less than $10^{-6}$. However, when the albedo is unity, then this is kind of impractical and I changed the definition of infinity to be that the diffuse transmission changes by less than one part in $10^{-6}$ after one doubling step. The more I think about this, the less sense it makes....

(Prototype for Double_Until_Infinite 104) ≡

void Double_Until_Infinite(int n, double **r, double **t)

This code is used in sections 90 and 105.

105. (Definition for Double_Until_Infinite 105) ≡

(Prototype for Double_Until_Infinite 104)
{
    double oldutu, UTU, UT1;
    (Allocate memory for a and b 111)
    UTU = 0.0;
    do {
        oldutu = UTU;
        Basic_Add_Layers(n, r, t, r, t, r, t, t, r, t, a, b);
        URU_and_UR1(n, 1.0, t, &UTU, &UT1);
    } while (fabs(UTU - oldutu) >= 0.000001);
    (Free Memory for a and b 112)
}

This code is used in section 89.

a: double **, §92.  
§91.  
HUGE_VAL, <math.h>.

b: double **, §92.  
fabs, <math.h>.

Basic_Add_Layers: static void.  

URU_and_UR1: void, §16.
106. Internal Radiance.

*Between* finds the radiance between two slabs. This equation for the upward radiance at the interface between two layers is

\[
L_+ = (E - R^{12} \star R^{10})^{-1} (R^{12} \star T^{01} \star L^0 + T^{21} \star L^2)
\]

where \(L^0\) is the downward radiance on the top layer and \(L^2\) is the upward radiance on the bottom layer. The equation for the downward mid-layer radiance can be obtained similarly using

\[
L_- = (E - R^{10} \star R^{12})^{-1} (T^{01} \star L^0 + R^{10} \star T^{21} \star L^2)
\]

Now assume that \(L^2\) is zero. Then the matrix

\[
L_- = (E - R^{12} \star R^{10})^{-1} R^{12} \star T^{01}
\]

can be used to find the downward fluence by simply star multiplying with the downward irradiance. Similarly,

\[
L_+ = (E - R^{10} \star R^{12})^{-1} T^{01}
\]

(Prototype for *Between* 106) \equiv

\[
\text{void Between}(\text{int } n, \text{double } *^*R01, \text{double } *^*R10, \text{double } *^*T01, \text{double } *^*T10, \text{double } *^*R12, \text{double } *^*R21, \text{double } *^*T12, \text{double } *^*T21, \text{double } *^*Lup, \text{double } *^*Ldown)
\]

This code is used in sections 90 and 107.

107. (Definition for *Between* 107) \equiv

(Prototype for *Between* 106)

\[
\{
\text{Allocate memory for } a \text{ and } b
\text{ Star_Multiply}(n, R10, R12, a);
\text{ Star_One_Minus}(n, a);
\text{ Right_Inverse_Multiply}(n, a, T01, Ldown);
\text{ Star_Multiply}(n, R12, R10, a);
\text{ Star_One_Minus}(n, a);
\text{ Right_Inverse_Multiply}(n, a, R12, b);
\text{ Star_Multiply}(n, b, T01, Lup);
\text{ Free Memory for } a \text{ and } b
\}
\]

This code is used in section 89.
108. **Utility routines.**

109. Star matrix multiplication $A \ast B$ is defined to directly correspond to an integration, i.e.

$$A \ast B = \int_{0}^{1} A(\mu, \mu') B(\mu', \mu'') 2 \mu d\mu$$

then

$$A \ast B = \sum_{j} A_{ij} 2 \mu_{j} w_{j} B_{jk}$$

where $\mu_{j}$ is the $j$th quadrature angle and $w_{j}$ is its corresponding weight. It is sometimes useful to consider these matrix “star multiplications” as normal matrix multiplications which include a diagonal matrix $c$

$$c_{ij} = 2 \mu_{i} w_{i} \delta_{ij}$$

Thus a matrix star multiplication may be written

$$A \ast B = A c B$$

where the multiplications on the RHS of the above equation are usual matrix multiplications.

Since the routine *Matrix_Multiply* that multiplies the matrices $A$ and $B$ to get $C$, allows $A$ and $C$ to be coincident. I first find $C = Ac$ and then do $C = C \cdot B$. This allows us to avoid allocating a temporary matrix. $A$ may occupy the same memory as $C$, but $B$ and $C$ must be distinct.

(Definition for *Star_Multiply* 109) \equiv

```c
static void Star_Multiply(int n, double **A, double **B, double **C)
{
    Right_Diagonal_Multiply(n, A, twoaw, C);
    Matrix_Multiply(n, C, B, C);
}
```

This code is used in section 89.

110. This subtracts the matrix $A$ from the unit matrix for star multiplication.

(Definition for *Star_One_Minus* 110) \equiv

```c
static void Star_One_Minus(int n, double **A)
{
    int i, j;
    for (i = 1; i <= n; i++)
    {
        for (j = 1; j <= n; j++)
        {
            A[i][j] *= -1;
            A[i][i] += 1.0/twoaw[i];
        }
    }
}
```

This code is used in section 89.

---

$a$: double **, §92.  
$Right_Diagonal_Multiply$: void, §168.  
$Right_Inverse_Multiply$: void, §206.  
$twoaw$: double [], §10.  
$Matrix_Multiply$: void, §177.
111. <Allocate memory for \( a \) and \( b \)>
\[
\text{double} ~~ **a, **b; \\
a = dmatrix(1, n, 1, n); \\
b = dmatrix(1, n, 1, n);
\]
This code is used in sections 95, 97, 99, 101, 103, 105, and 107.

112. <Free Memory for \( a \) and \( b \)>
\[
\text{free_dmatrix}(a, 1, n, 1, n); \\
\text{free_dmatrix}(b, 1, n, 1, n);
\]
This code is used in sections 95, 97, 99, 101, 103, 105, and 107.
113. AD Boundary.
This section has routines associated with incorporating boundary conditions into the adding-doubling algorithm.

\{ad_bound.c\ 113\} ≡
#include <math.h>
#include "ad_globl.h"
#include "ad_bound.h"
#include "ad_frsnl.h"
#include "ad_matrx.h"
#include "nr_util.h"

Prototype for A\_Add\_Slide 122; 
Prototype for B\_Add\_Slide 124; 
Definition for Init\_Boundary 117 
Definition for Boundary\_RT 120 
Definition for Add\_Top 128 
Definition for Add\_Bottom 130 
Definition for A\_Add\_Slide 123 
Definition for B\_Add\_Slide 125 
Definition for Add\_Slides 132 
Definition for Sp\_RT 134

114. \{ad_bound.h\ 114\} ≡
(Preprocessor definitions)
(Prototype for Init\_Boundary 116); 
(Prototype for Boundary\_RT 119); 
(Prototype for Add\_Top 127); 
(Prototype for Add\_Bottom 129); 
(Prototype for Add\_Slides 131); 
(Prototype for Sp\_RT 133);

\_Add\_Slide: static void, §122. 
Add\_Bottom: void, §129. 
Add\_Slides: void, §131. 
Add\_Top: void, §127. 
B\_Add\_Slide: static void, §124. 
Boundary\_RT: void, §119. 
dmatrix, <nr\_util.h>. 
Inst\_Boundary: void, §116. 
n: int, §110. 
Sp\_RT: void, §133. 
free\_dmatrix, <nr\_util.h>. 

115. Boundary Initialization.

116. *Init* _Boundary* creates reflection and transmission matrices to simulate a boundary. If boundary \( \equiv \) TOP_BOUNDARY then the arrays returned are for the top surface and the labels are as expected i.e. T01 is the reflection for light from air passing to the slab. Otherwise the calculations are made for the bottom surface and the labels are backwards i.e. T01 \( \equiv \) T32 and T10 \( \equiv \) T23, where 0 is the first air slide surface, 1 is the slide/slab surface, 2 is the second slide/slab surface, and 3 is the bottom slide/air surface.

```c
#define TOP_BOUNDARY 0
#define BOTTOM_BOUNDARY 1
```

(Prototype for *Init* _Boundary* 116) \( \equiv \)

```c
void Init_Boundary(struct AD_slab_type slab, int n, double *R01, double *R10, double *T01, double *T10, char boundary)
```

This code is used in sections 114 and 117.

117. (Definition for *Init* _Boundary* 117) \( \equiv \)

(Prototype for *Init* _Boundary* 116)

```c
{
    if (boundary \( \equiv \) TOP_BOUNDARY) {
        Boundary_RT(1.0, slab.n_top_slide, slab.n_slab, n, slab.b_top_slide, R01, T01);
        Boundary_RT(slab.n_slab, slab.n_top_slide, 1.0, n, slab.b_top_slide, R10, T10);
    }
    else {
        Boundary_RT(1.0, slab.n_bottom_slide, slab.n_slab, n, slab.b_bottom_slide, R10, T10);
        Boundary_RT(slab.n_slab, slab.n_bottom_slide, 1.0, n, slab.b_bottom_slide, R01, T01);
    }
}
```

This code is used in section 113.

118. *Boundary* _RT* computes the diagonal matrix (represented as an array) that characterizes reflection and transmission at an air (0), absorbing glass (1), slab (2) boundary. The reflection matrix is the same entering or exiting the slab. The transmission matrices should differ by a factor of \( (\frac{n_{slab}}{n_{outside}})^4 \), due to \( n^2 \) law of radiance, but there is some inconsistency in the program and if I use this principle then regular calculations for R and T don’t work and the fluence calculations still don’t work. So punted and took all that code out.

The important point that must be considered. All the angles in this program assume that the angles are those actually in the sample. This allows angles greater that the critical angle to be used. Everything is fine as long as the index of refraction of the incident medium is 1.0. If this is not the case then the angle inside the medium must be figured out.

119. (Prototype for *Boundary* _RT* 119) \( \equiv \)

```c
void Boundary_RT(double n_i, double n_g, double n_t, int n, double b,
                 double *R, double *T)
```

This code is used in sections 114 and 120.
120. (Definition for Boundary_{RT} 120) \equiv
(Prototype for Boundary_{RT} 119)
{
    int i;
    double refl, trans;
    double mu;
    for (i = 1; i \leq n; i++) {
        if (n_{i} \equiv 1.0) mu = \cosSnell(n_{t}, \angle[i], n_{i});
        else mu = \angle[i];
        Absorbing_Glass_{RT}(n_{i}, n_{g}, n_{t}, mu, b, &refl, &trans);
        R[i] = refl \times twoaw[i];
        T[i] = trans;
    }
}
This code is used in section 113.
121. **Boundary incorporation algorithms.**

The next two routines \texttt{A\_Add\_Slide} and \texttt{B\_Add\_Slide} are modifications of the full addition algorithms for dissimilar layers. They are optimized to take advantage of the diagonal nature of the boundary matrices. There are two algorithms below to facilitate adding slides below and above the sample.

122. \texttt{A\_Add\_Slide} computes the resulting \( R_{20} \) and \( T_{02} \) matrices for a glass slide on top of an inhomogeneous layer characterized by \( R_{12} \), \( R_{21} \), \( T_{12} \), \( T_{21} \). It is ok if \( R_{21} \equiv R_{12} \) and \( T_{12} \equiv T_{21} \). But I do not think that it is required by this routine. The result matrices \( R_{20} \) and \( T_{02} \) should be independent of the input matrices None of the input matrices are changed.

The critical quantities are

\[
T_{02} = T_{12}(E - R_{10}R_{12})^{-1}T_{01}
\]

and

\[
R_{20} = T_{12}(E - R_{10}R_{12})^{-1}R_{10}T_{21} + R_{21}
\]

(Prototype for \texttt{A\_Add\_Slide} 122) \equiv

\[
\text{static void } \texttt{A\_Add\_Slide}(\text{int } n, \text{double } **R_{12}, \text{double } **R_{21}, \text{double } **T_{12}, \text{double } **T_{21}, \text{double } *R_{10}, \text{double } *T_{01}, \text{double } **R_{20}, \text{double } **T_{02}, \text{double } **\text{atemp}, \text{double } **\text{btemp})
\]

This code is used in sections 113 and 123.

123. (Definition for \texttt{A\_Add\_Slide} 123) \equiv

(Prototype for \texttt{A\_Add\_Slide} 122)

\[
\text{double } **\text{ctemp};
\]

\[
\text{ctemp = } R_{20};
\]

\[
\text{Left\_Diagonal\_Multiply}(n, R_{10}, R_{12}, \text{atemp});
\]

\[
\text{One\_Minus}(n, \text{atemp});
\]

\[
\text{Left\_Inverse\_Multiply}(n, \text{atemp}, T_{12}, \text{ctemp});
\]

\[
\text{Right\_Diagonal\_Multiply}(n, \text{ctemp}, T_{01}, T_{02});
\]

\[
\text{Right\_Diagonal\_Multiply}(n, \text{ctemp}, R_{10}, \text{btemp});
\]

\[
\text{Matrix\_Multiply}(n, \text{btemp}, T_{21}, \text{atemp});
\]

\[
\text{Matrix\_Sum}(n, R_{21}, \text{atemp}, \text{R}_{20});
\]

This code is used in section 113.

124. \texttt{B\_Add\_Slide} computes the resulting \( R_{02} \) and \( T_{20} \) matrices for a glass slide on top of an inhomogeneous layer characterized by \( R_{12} \), \( R_{21} \), \( T_{12} \), \( T_{21} \). It is ok if \( R_{21} \equiv R_{12} \) and \( T_{12} \equiv T_{21} \). But I do not think that it is required by this routine. The result matrices \( R_{02} \) and \( T_{20} \) should be independent of the input matrices None of the input matrices are changed.

The critical equations are

\[
T_{20} = T_{10}(E - R_{12}R_{10})^{-1}T_{21}
\]

and

\[
R_{02} = T_{10}(E - R_{12}R_{10})^{-1}R_{12}T_{01} + R_{01}
\]

(Prototype for \texttt{B\_Add\_Slide} 124) \equiv
static void B_Add_Slide(int n, double **R12, double **T21, double *R01, double *R10, double *T01, double *T10, double **R02, double **T20, double **atemp, double **btemp)

This code is used in sections 113 and 125.

125. (Definition for B_Add_Slide 125) ≡
   (Prototype for B_Add_Slide 124)
   {
   double **ctemp;
   int i;
   ctemp = R02;
   Right_Diagonal_Multiply(n, R12, R10, atemp);
   One_Minus(n, atemp);
   Diagonal_To_Matrix(n, T10, btemp);
   Left_Inverse_Multiply(n, atemp, btemp, ctemp);
   Matrix_Multiply(n, ctemp, T21, T20);
   Matrix_Multiply(n, ctemp, R12, btemp);
   Right_Diagonal_Multiply(n, btemp, T01, R02);
   for (i = 1; i ≤ n; i++) R02[i][i] += R01[i]/DSQR(twoaw[i]);
   }

This code is used in section 113.

---

Diagonal_To_Matrix: void, §166.
Matrix_Multiply: void, §177.
Right_Diagonal_Multiply: void, §168.
DSQR, <nr_util.h>: void, §162.
Left_Diagonal_Multiply: void, §170.
One_Minus: void, §162.
Matrix_Sum: void, §168.
Left_Inverse_Multiply: void, §162.
twoaw: double [], §10.
126. Routines to incorporate slides.

127. *Add_Top* calculates the reflection and transmission matrices for a slab with a boundary placed on top of it.

\[
\begin{align*}
\text{n} & \quad \text{size of matrix} \\
R01, R10, T01, T10 & \quad 	ext{R, T for slide assuming 0=air and 1=slab} \\
R12, R21, T12, T21 & \quad 	ext{R, T for slab assuming 1=slide and 2=?} \\
R02, R20, T02, T20 & \quad \text{calc R, T for both assuming 0=air and 2=?} \\
atemp, btemp & \quad \text{previously allocated temporary storage matrices}
\end{align*}
\]

\[(Prototype for Add_Top 127) \equiv \]
\[
\text{void Add_Top(int n, double } \ast \text{R01, double } \ast \text{R10, double } \ast \text{T01, double } \ast \text{T10, double } \ast \text{R12, double } \ast \text{R21, double } \ast \text{T12, double } \ast \text{T21, double } \ast \text{R02, double } \ast \text{T02, double } \ast \text{T20, double } \ast \text{atemp, double } \ast \text{btemp})
\]

This code is used in sections 114 and 128.

128. \[(Definition for Add_Top 128) \equiv (Prototype for Add_Top 127) \]
\[
\begin{align*}
\text{A_Add_Slide}(n, R12, R21, T12, T21, R10, T01, R01, T02, T20, atemp, btemp); \\
\text{B_Add_Slide}(n, R10, T01, R21, R12, T12, T21, R02, T02, T20, atemp, btemp);
\end{align*}
\]

This code is used in section 113.

129. *Add_BOTTOM* calculates the reflection and transmission matrices for a slab with a boundary placed beneath it

\[
\begin{align*}
\text{n} & \quad \text{size of matrix} \\
R01, R10, T01, T10 & \quad 	ext{R, T for slab assuming 0=slab top and 1=slab bottom} \\
R12, R21, T12, T21 & \quad 	ext{R, T for slide assuming 1=slab bottom and 2=slide bottom} \\
R02, R20, T02, T20 & \quad \text{calc R, T for both assuming 0=slab top and 2=slide bottom} \\
atemp, btemp & \quad \text{previously allocated temporary storage matrices}
\end{align*}
\]

\[(Prototype for Add_BOTTOM 129) \equiv \]
\[
\text{void Add_BOTTOM(int n, double } \ast \text{R01, double } \ast \text{R10, double } \ast \text{T01, double } \ast \text{T10, double } \ast \text{R12, double } \ast \text{R21, double } \ast \text{T12, double } \ast \text{T21, double } \ast \text{R02, double } \ast \text{T02, double } \ast \text{T20, double } \ast \text{atemp, double } \ast \text{btemp})
\]

This code is used in sections 114 and 130.

130. \[(Definition for Add_BOTTOM 130) \equiv (Prototype for Add_BOTTOM 129) \]
\[
\begin{align*}
\text{A_Add_Slide}(n, R10, R01, T10, R12, T21, R02, T20, atemp, btemp); \\
\text{B_Add_Slide}(n, R10, T01, R21, R12, T21, T12, R20, T02, atemp, btemp);
\end{align*}
\]

This code is used in section 113.
Adding-Doubling (Version 1.5): ROUTINES TO INCORPORATE SLIDES

A\textunderscore Add\_Slide: \texttt{static void}, §122. \quad B\textunderscore Add\_Slide: \texttt{static void}, §124.
131. Including identical slides. Add_Slides is optimized for a slab with equal boundaries on each side. Add_Slides calculates the reflection and transmission matrices for a slab with the same boundary placed above and below it. It is assumed that the slab is homogeneous. in this case the resulting R and T matrices are independent of direction. There are no constraints on R01, R10, T01, and T10. The handles for R and T cannot be equal to those for R_total and T_total.

- n: size of matrix
- R01, R10, T01, T10: R, T for slide assuming 0=air and 1=slab
- R, T: R, T for homogeneous slab
- R_total, T_total: R, T for all 3 with top = bottom boundary
- atemp, btemp: temporary storage matrices

If equal boundary conditions exist on both sides of the slab then, by symmetry, the transmission and reflection operator for light travelling from the top to the bottom are equal to those for light propagating from the bottom to the top. Consequently only one set need be calculated. This leads to a faster method for calculating the reflection and transmission for a slab with equal boundary conditions on each side. Let the top boundary be layer 01, the medium layer 12, and the bottom layer 23. The boundary conditions on each side are equal: R01 = R32, R10 = R23, T01 = T32, and T10 = T23. For example the light reflected from layer 01 (travelling from boundary 0 to boundary 1) will equal the amount of light reflected from layer 32, since there is no physical difference between the two cases. The switch in the numbering arises from the fact that light passes from the medium to the outside at the top surface by going from 1 to 0, and from 2 to 3 on the bottom surface. The reflection and transmission for the slab with boundary conditions are R30 and T03 respectively. These are given by

\[ T_{02} = T_{12}(E - R_{10} R_{12})^{-1} T_{01} \]

and

\[ R_{20} = T_{12}(E - R_{10} R_{12})^{-1} R_{10} T_{21} + R_{21} \]

and

\[ T_{03} = T_{10}(E - R_{20} R_{10})^{-1} T_{02} \]

and

\[ R_{30} = T_{10}(E - R_{20} R_{10})^{-1} R_{20} T_{01} + R_{01} \]

Further increases in efficiency may be made by exploiting the diagonal nature of the reflection and transmission operators for an interface, since most matrix/matrix multiplications above become vector/matrix multiplications.

(Prototype for Add_Slides 131) ≡

```c
void Add_Slides(int n, double *R01, double *R10, double *T01, double *T10,
                 double **R, double **T,
                 double **R_total, double **T_total,
                 double **atemp, double **btemp)
```

This code is used in sections 114 and 132.
132.
(Definition for Add_Slides 132) ≡
(Prototype for Add_Slides 131)
{
    int i;
    double **R12, **R21, **T12, **T21;
    double temp;
    R12 = R;
    R21 = R;
    T21 = T;
    T12 = T;
    Left_Diagonal_Multiply(n, R10, R12, atemp);
    One_Minus(n, atemp);
    Left_Inverse_Multiply(n, atemp, T12, T_total);
    Right_Diagonal_Multiply(n, T_total, R10, btemp);
    Matrix_Multiply(n, btemp, T21, R_total);
    Matrix_Sum(n, R_total, R21, R_total);
    Right_Diagonal_Multiply(n, R_total, R10, atemp);
    One_Minus(n, atemp);
    Matrix_Inverse(n, atemp, btemp);
    Left_Diagonal_Multiply(n, T10, btemp, atemp);
    Matrix_Multiply(n, atemp, T_total, btemp);
    Right_Diagonal_Multiply(n, btemp, T01, T_total);
    Matrix_Multiply(n, atemp, R_total, btemp);
    Right_Diagonal_Multiply(n, btemp, T01, R_total);
    for (i = 1; i ≤ n; i++) {
        temp = twoaw[i];
        R_total[i][i] += R01[i]/(temp * temp);
    }
}
This code is used in section 113.

Left_Diagonal_Multiply: void, §170.
Left_Inverse_Multiply: void, §204.
Matrix_Multiply: void, §177.
Matrix_Sum: void, §172.
One_Minus: void, §162.
Right_Diagonal_Multiply: void, §168.
twoaw: double [], §10.
133. Specular R and T.

$Sp_{RT}$ calculates the specular reflection and transmission for light normally incident on a slide-slab-slide sandwich. The sample is characterized by the record $slab$. The total unscattered reflection and transmission for normal irradiance ($ur1$ and $ut1$) together with their companions $uru$ and $utu$ for diffuse irradiance.

The way that this routine calculates the diffuse unscattered quantities based on the global quadrature angles previously set-up. Consequently, these estimates are not exact. In fact if $n = 4$ then only two quadrature points will actually be used to figure out the diffuse reflection and transmission (assuming mismatched boundaries).

This algorithm is pretty simple. Since the quadrature angles are all chosen assuming points inside the medium, I must calculate the corresponding angle for light entering from the outside. If the cosine of this angle is greater than zero then the angle does not correspond to a direction in which light is totally internally reflected. For this ray, I find the unscattered that would be reflected or transmitted from the slab. I multiply this by the quadrature angle and weight $twoaw[i]$ to get the total diffuse reflectance and transmittance.

A bit of bookkeeping is necessary to make sure that the reflection and transmission for normal irradiance is not calculated twice. If I was sure which quadrature angle corresponded to normal irradiance then this bookkeeping could be eliminated. However, $\mu = 1$ would always have to be a quadrature angle.

Oh, yes. The mysterious multiplication by a factor of $n_{slab} \cdot n_{slab}$ is required to account for the $n^2$-law of radiance.

(Prototype for $Sp_{RT}$ 133) $\equiv$

```c
void Sp_RT(int n, struct AD_slab_type slab, double *ur1, double *ut1, double *uru, double *utu)
```

This code is used in sections 114 and 134.

134. (Definition for $Sp_{RT}$ 134) $\equiv$

(Prototype for $Sp_{RT}$ 133)

```c
{
    double mu_outside, r, t;
    int i, do_normal;
    *uru = 0;
    *utu = 0;
    do_normal = 1;
    for (i = 1; i <= n; i++) {
        mu_outside = Cos_Snell(slab.n_slab, angle[i], 1.0);
        if (mu_outside != 0) {
            Sp_mu_RT(slab.n_top_slide, slab.n_slab, slab.n_bottom_slide, slab.b_top_slide, slab.b,
                     slab.b_bottom_slide, mu_outside, &r, &t);
            *uru += twoaw[i] * r;
            *utu += twoaw[i] * t;
        }
        if (mu_outside == 1.0) {
            do_normal = 0;
            *ut1 = t;
            *ur1 = r;
        }
    }
}
```
if (do_normal)  
  Sp_mu_RT(slab.n_top_slide, slab.n_slab, slab.n_bottom_slide, slab.b_top_slide, slab.b,  
  slab.b_bottom_slide, 1.0, ur1, ut1);
  *uru *= slab.n_slab * slab.n_slab;
  *utu *= slab.n_slab * slab.n_slab;
}

This code is used in section 113.

---

**AD_slab_type**: struct, §8.

- angle: double [], §10.
- b: double, §119.
- b_bottom_slide: double, §8.

**b_top_slide**: double, §8.

**Cos_Snell**: double, §139.

**i**: int, §132.

**n_bottom_slide**: double, §8.

**n_slab**: double, §69.

**n_top_slide**: double, §8.

**Sp_mu_RT**: void, §148.

**twoaw**: double [], §10.
135. AD Fresnel. This is a part of the core suite of files for the adding-doubling program. Not surprisingly, this program includes routines to calculate Fresnel reflection.

Change 3/3/95 in \textit{Cos\_Snell} to special case normal incidence.
Change 3/3/95 in \textit{Sp\_mu\_RT} to fix a bug.
Changes 5/21/95 to added routine for absorbing slides and improved documentation to work a bit better with \texttt{ctwill}.

\begin{verbatim}
#include <math.h>
#include <float.h>
#include <stdio.h>
#include "ad_frsnl.h"

Prototype for \texttt{Fresnel} 142;
Prototype for \texttt{Glass} 144;
Prototype for \texttt{R1} 153;
Definition for \texttt{Cos\_Critical\_Angle} 138
Definition for \texttt{Cos\_Snell} 140
Definition for \texttt{Fresnel} 143
Definition for \texttt{Glass} 145
Definition for \texttt{Absorbing\_Glass\_RT} 147
Definition for \texttt{R1} 154
Definition for \texttt{Sp\_mu\_RT} 149
Definition for \texttt{Diffuse\_Glass\_R} 156
\end{verbatim}

136. \begin{verbatim}
#include "ad_frsnl.h"
Prototype for \texttt{Cos\_Critical\_Angle} 137;
Prototype for \texttt{Cos\_Snell} 139;
Prototype for \texttt{Absorbing\_Glass\_RT} 146;
Prototype for \texttt{Sp\_mu\_RT} 148;
Prototype for \texttt{Diffuse\_Glass\_R} 155;
\end{verbatim}
137. The critical angle.

$\text{Cos}_{\text{Critical Angle}}$ calculates the cosine of the critical angle. If there is no critical angle then 0.0 is returned (i.e., cos(\(\pi/2\))). Note that no trigonmetric functions are required. Recalling Snell’s law

\[ n_i \sin \theta_i = n_t \sin \theta_t \]

To find the critical angle, let \( \theta_t = \pi/2 \) and then

\[ \theta_c = \sin^{-1} \frac{n_t}{n_i} \]

The cosine of this angle is then

\[ \cos \theta_c = \cos \left( \sin^{-1} \frac{n_t}{n_i} \right) = \sqrt{n_i^2 - n_t^2} \]

or more simply

\[ \cos \theta_c = \sqrt{1 - n^2} \]

where \( n = n_t/n_i \).

(Prototype for $\text{Cos}_{\text{Critical Angle}}$ 137) \(\equiv\)

\begin{verbatim}
  double Cos_Critical_Angle(double ni, double nt)
  {
    double x;
    if (nt >= ni) return 0.0;
    else {
      x = nt/ni;
      x = sqrt(1.0 - x * x);
      return x;
    }
  }
\end{verbatim}

This code is used in sections 136 and 138.

138. (Definition for $\text{Cos}_{\text{Critical Angle}}$ 138) \(\equiv\)

(Prototype for $\text{Cos}_{\text{Critical Angle}}$ 137)

\begin{verbatim}
  double x;
  if (nt >= ni) return 0.0;
  else {
    x = nt/ni;
    x = sqrt(1.0 - x * x);
    return x;
  }
\end{verbatim}

This code is used in section 135.
139. Snell’s Law.

\texttt{Cos\_Snell} returns the cosine of the angle that the light propagates through a medium given the cosine of the angle of incidence and the indices of refraction. Let the cosine of the angle of incidence be \( \mu_i \), the transmitted cosine as \( \mu_t \), the index of refraction of the incident material \( n_i \) and that of the transmitted material be \( n_t \).

Snell’s law states

\[ n_i \sin \theta_i = n_t \sin \theta_t \]

but if the angles are expressed as cosines, \( \mu_i = \cos \theta_i \) then

\[ n_i \sin(\cos^{-1} \mu_i) = n_t \sin(\cos^{-1} \mu_t) \]

Solving for \( \mu_t \) yields

\[ \mu_t = \cos\{\sin^{-1}\left[(n_i/n_t) \sin(\cos^{-1} \mu_i)\right]\} \]

which is pretty ugly. However, note that \( \sin(\cos^{-1} \mu) = \sqrt{1-\mu^2} \) and the above becomes

\[ \mu_t = \sqrt{1-(n_i/n_t)^2(1-\mu_i^2)} \]

and no trigonmetric calls are necessary. Hooray!

A few final notes. I check to make sure that the index of refraction of changes before calculating a bunch of stuff. This routine should not be passed incident angles greater than the critical angle, but I shall program defensively and test to make sure that the argument of the \texttt{sqrt} function is non-negative. If it is, then I return \( \mu_t = 0 \) i.e., \( \theta_t = 90^\circ \).

I also pretest for the common but trivial case of normal incidence.

\begin{verbatim}
(Prototype for \texttt{Cos\_Snell 139} ) ≡
double Cos\_Snell(double n_i, double mu_i, double n_t, double n_t)
\end{verbatim}

This code is used in sections 136 and 140.

140. (Definition for \texttt{Cos\_Snell 140} ) ≡
(Prototype for \texttt{Cos\_Snell 139} )
\begin{verbatim}
{ double temp;
  if (mu_i ≡ 1.0) return 1.0;
  if (n_i ≡ n_t) return mu_i;
  temp = n_i/n_t;
  temp = 1.0 - temp * temp * (1.0 - mu_i * mu_i);
  if (temp < 0) return 0.0;
  else return (sqrt(temp));
}
\end{verbatim}

This code is used in section 135.
141. Fresnel Reflection.

Fresnel calculates the specular reflection for light incident at an angle $\theta_i$ from the normal (having a cosine equal to $\mu_i$) in a medium with index of refraction $n_i$ onto a medium with index of refraction $n_t$.

The usual way to calculate the total reflection for unpolarized light is to use the Fresnel formula

$$R = \frac{1}{2} \left[ \frac{\sin^2(\theta_i - \theta_t)}{\sin^2(\theta_i + \theta_t)} + \frac{\tan^2(\theta_i - \theta_t)}{\tan^2(\theta_i + \theta_t)} \right]$$

where $\theta_i$ and $\theta_t$ represent the angle (from normal) that light is incident and the angle at which light is transmitted. There are several problems with calculating the reflection using this formula. First, if the angle of incidence is zero, then the formula results in division by zero. Furthermore, if the angle of incidence is near zero, then the formula is the ratio of two small numbers and the results can be inaccurate. Second, if the angle of incidence exceeds the critical angle, then the calculation of $\theta_t$ results in an attempt to find the arcsine of a quantity greater than one. Third, all calculations in this program are based on the cosine of the angle. This routine forces the calling routine to find $\theta_i = \cos^{-1} \mu$. Fourth, the routine also gives problems when the critical angle is exceeded.

Closer inspection reveals that this is the wrong formulation to use. The formulas that should be used for parallel and perpendicular polarization are

$$R_\parallel = \left[ \frac{n_t \cos \theta_i - n_i \cos \theta_t}{n_t \cos \theta_i + n_i \cos \theta_t} \right]^2, \quad R_\perp = \left[ \frac{n_i \cos \theta_i - n_t \cos \theta_t}{n_i \cos \theta_i + n_t \cos \theta_t} \right]^2.$$

The formula for unpolarized light, written in terms of $\mu_i = \cos \theta_i$ and $\mu_t = \cos \theta_t$ is

$$R = \frac{1}{2} \left[ \frac{n_i \mu_i - n_t \mu_t}{n_i \mu_i + n_t \mu_t} \right]^2 + \frac{1}{2} \left[ \frac{n_t \mu_i - n_i \mu_t}{n_t \mu_i + n_i \mu_t} \right]^2$$

This formula has the advantage that no trig routines need to be called and that the case of normal irradiance does not cause division by zero. Near normal incidence remains numerically well-conditioned. In the routine below, I test for matched boundaries and normal incidence to eliminate unnecessary calculations. I also test for total internal reflection to avoid possible division by zero. I also find the ratio of the indices of refraction to avoid an extra multiplication and several intermediate variables.

142. (Prototype for Fresnel 142) ≡

static double Fresnel(double n_i, double n_t, double mu_i)

This code is used in sections 135 and 143.

\[ \text{sqrt, <math.h>}. \]
143. (Definition for Fresnel 143) ≡
(Prototype for Fresnel 142)
{
    double mu_t, ratio, temp, temp1;
    if (n_i ≡ n_t) return 0.0;
    if (mu_i ≡ 1.0) {
        temp = (n_i - n_t)/(n_i + n_t);
        return (temp * temp);
    }
    if (mu_i ≡ 0.0) return 1.0;
    mu_t = Cos_Snell(n_i, mu_i, n_t);
    if (mu_t ≡ 0.0) return 1.0;
    ratio = n_i/n_t;
    temp = ratio * mu_t;
    temp1 = (mu_i - temp)/(mu_i + temp);
    temp = ratio * mu_i;
    temp = (mu_t - temp)/(mu_t + temp);
    return ((temp1 * temp1 + temp * temp)/2);
}
This code is used in section 135.
144. Reflection from a glass slide.

Glass calculates the total specular reflection (i.e., including multiple internal reflections) based on the indices of refraction of the incident medium \( n_\text{i} \), the glass \( n_\text{g} \), and medium into which the light is transmitted \( n_\text{t} \) for light incident at an angle from the normal having cosine \( \mu_\text{i} \).

In many tissue optics problems, the sample is constrained by a piece of glass creating an air-glass-tissue sequence. The adding-doubling formalism can calculate the effect that the layer of glass will have on the radiative transport properties by including a layer for the glass-tissue interface and a layer for the air-glass interface. However, it is simpler to find net effect of the glass slide and include only one layer for the glass boundary.

The first time I implemented this routine, I did not include multiple internal reflections. After running test cases, it soon became apparent that the percentage errors were way too big for media with little absorption and scattering. It is not hard to find the result for the reflection from a non-absorbing glass layer (equation A2.21 in my dissertation) in which multiple reflections are properly accounted for

\[
gr = r_1 + r_2 - 2r_1r_2 \]

Here \( r_1 \) is the reflection at the air-glass interface and \( r_2 \) is the reflection at the glass-sample interface.

There is one pitfall in calculating \( r_g \). When the angle of incidence exceeds the critical angle then the formula above causes division by zero. If this is the case then \( r_1 = 1 \) and can easily be tested for.

To eliminate unnecessary computation, I check to make sure that it really is necessary to call the Fresnel routine twice. It is noteworthy that the formula for \( r_g \) works correctly if the first boundary is not totally reflecting but the second one is. Note that \( \mu_g \) gets calculated twice (once in the first call to Fresnel and once directly).

(Prototype for Glass 144) ≡

\[
\text{static double Glass(} \text{double n}_\text{i}, \text{double n}_\text{g}, \text{double n}_\text{t}, \text{double } \mu_\text{i})
\]

This code is used in sections 135 and 145.
145. (Definition for Glass 145) \equiv
(Prototype for Glass 144)
{
    double r1, r2, mu_g, temp;
    if (n_i \equiv n_g) return (Fresnel(n_g, n_t, mu_i));
    r1 = Fresnel(n_i, n_g, mu_i);
    if (r1 \geq 1.0 \lor n_g \equiv n_t) return r1;
    mu_g = Cos_Snell(n_i, mu_i, n_g);
    r2 = Fresnel(n_g, n_t, mu_g);
    temp = r1 * r2;
    temp = (r1 + r2 - 2 * temp) / (1 - temp);
    return temp;
}

This code is used in section 135.
146. Reflection from an absorbing slide.

*Absorbing Glass RT* calculates the total specular reflection and transmission (i.e., including multiple internal reflections) based on the indices of refraction of the incident medium $n_i$, the glass $n_g$, and medium into which the light is transmitted $n_t$ for light incident at an angle from the normal having cosine $\mu_i$. The optical thickness of the glass $b = \mu_a d$ is measured normal to the glass.

This routine was generated to help solve a problem with the inverse adding-doubling program associated with samples with low absorbances. A particular situation arises when the slides have significant absorption relative to the sample absorption. Anyway, it is not hard to extend the result for non-absorbing slides to the absorbing case

$$r = \frac{r_1 + (1 - 2r_1)r_2\exp(-2b/\mu_g)}{1 - r_1r_2\exp(-2b/\mu_g)}$$

Here $r_1$ is the reflection at the sample-glass interface and $r_2$ is the reflection at the glass-air interface and $\mu_g$ is the cosine of the angle inside the glass. Note that if $b \neq 0$ then the reflection depends on the order of the indices of refraction, otherwise $n_i$ and $n_t$ can be switched and the result should be the same.

The corresponding result for transmission is

$$t = \frac{(1 - r_1)(1 - r_2)\exp(-b/\mu_g)}{1 - r_1r_2\exp(-2b/\mu_g)}$$

There are two potential pitfalls in the calculation. The first is when the angle of incidence exceeds the critical angle then the formula above causes division by zero. If this is the case, *Fresnel* will return $r_1 = 1$ and this routine responds appropriately. The second case is when the optical thickness of the slide is too large.

I don’t worry too much about optimal coding, because this routine does not get called all that often and also because *Fresnel* is pretty good at avoiding unnecessary computations. At worst this routine just has a couple of extra function calls and a few extra multiplications.

I also check to make sure that the exponent is not too small.

(Prototype for *Absorbing Glass RT* 146) ≡

```c
void Absorbing_Glass_RT(double n_i, double n_g, double n_t, double mu_i, double b, double *r, double *t)
```

This code is used in sections 136 and 147.
(Definition for \texttt{Absorbing\_Glass\_RT 147}) \equiv
(Prototype for \texttt{Absorbing\_Glass\_RT 146})
\
\{ double \ r1, \ r2, \ mu.g, \ expo, \ denom; \\
* t = 0; \\
* r = Fresnel(n_i, n_g, mu.i); \\
if \ (r ≥ 1.0 \lor b ≡ \texttt{HUGE\_VAL} \lor mu.i ≡ 0.0) \ return; \\
mu.g = Cos\_Shell(n_i, mu.i, n_g); \\
r1 = *r; \\
r2 = Fresnel(n_g, n_t, mu.g); \\
if \ (b ≡ 0.0) \ { \\
* r = (r1 + r2 - 2.0 * r1 * r2) / (1 - r1 * r2); \\
* t = 1.0 - (*r); \\
}
else \ { \\
expo = -b/mu.g; \\
if \ (2 * expo ≤ \texttt{DBL\_MIN\_10\_EXP} \lor 2.3025851) \ return; \\
expo = \exp(expo); \\
denom = 1.0 - r1 * r2 * expo * expo; \\
* r = (r1 + (1.0 - 2.0 * r1) * r2 * expo * expo) / denom; \\
* t = (1.0 - r1) * (1.0 - r2) * expo / denom; \\
}
\}

This code is used in section 135.
148. Unscattered refl and trans for a sample.

`Sp_mu_RT` calculates the unscattered reflection and transmission (i.e., specular) for light incident at an angle having a cosine `mu` from air onto a non-absorbing glass plate with index `n_top` on a sample with index `n_slab` resting on another non-absorbing glass plate with index `n_bottom` and then exiting into air again.

(Prototype for `Sp_mu_RT` 148) ≡
```c
void Sp_mu_RT(double n_top, double n_slab, double n_bottom, double tau_top, double tau_slab, double tau_bottom, double mu, double *r, double *t)
```

This code is used in sections 136 and 149.

149. (Definition for `Sp_mu_RT` 149) ≡

(Prototype for `Sp_mu_RT` 148)
```c
{ double r_top, r_bottom, t_top, t_bottom, mu_slab, beer, denom, temp;
  *r = 0;
  *t = 0;
  Absorbing_Glass_RT(1.0, n_top, n_slab, mu, tau_top, &r_top, &t_top);
  Absorbing_Glass_RT(n_slab, n_bottom, 1.0, mu, tau_bottom, &r_bottom, &t_bottom);
  (Calculate beer 151)
  (Calculate r and t 152)
}
```

This code is used in section 135.

150. Nothing tricky here except a check to make sure that the reflection for the top is not equal to that on the bottom before calculating it again. I also drop out of the routine if the top surface is totally reflecting.

151. I am careful here not to cause an underflow error and to avoid division by zero.

It turns out that I found a small error in this code fragment. Basically I misunderstood what one of the values in `float.h` represented. This version is now correct

(Calculate beer 151) ≡
```c
mu_slab = Cos_Snell(1.0, mu, n_slab);
if (mu_slab ≡ 0) beer = 0.0;
else if (tau_slab ≡ HUGE_VAL) beer = 0.0;
else {
  temp = −tau_slab/mu_slab;
  if (2 * temp ≤ DBL_MIN_10_EXP * 2.3025851) beer = 0.0;
  else beer = exp(temp);
}
```

This code is used in section 149.

---

`Absorbing_Glass_RT`: void, §146.
`b`: double, §146.
`Cos_Snell`: double, §139.
`DBL_MIN_10_EXP`, `<float.h>`.
`exp`, `<math.h>`.

`Fresnel`: static double, §142.
`HUGE_VAL`, `<math.h>`.
`mu_i`: double, §146.
`n_j`: double, §146.
`n_i`: double, §146.
`r`: double *, §146.
`t`: double *, §146.
152. If $r_{\text{top}}$ is the reflection for the top and $r_{\text{bottom}}$ is that for the bottom surface then the total reflection will be

$$r = r_{\text{top}} + \frac{r_{\text{bottom}} t_{\text{top}}^2 \exp(-2\tau/\mu)}{1 - r_{\text{top}} r_{\text{bottom}} \exp(-2\tau/\mu)}$$

and the transmission is

$$t = \frac{t_{\text{top}} t_{\text{bottom}} \exp(-\tau/\mu)}{1 - r_{\text{top}} r_{\text{bottom}} \exp(-2\tau/\mu)}$$

where $\mu$ is the angle inside the slab and $\tau$ is the optical thickness of the slab.

I have already calculated the reflections and the exponential attenuation, so I can just plug into the formula after making sure that it is really necessary. The denominator cannot be zero since I know $r_{\text{top}} < 1$ and that $r_{\text{bottom}}$ and $\text{beer}$ are less than or equal to one.

The bug that was fixed was in the calculated reflection. I omitted a $r_{\text{bottom}}$ in the numerator of the fraction used to calculate the reflection.

\[
\begin{cases}
\text{(Calculate } r \text{ and } t \text{ 152) } \equiv \\
\quad \text{if } (\text{beer} \equiv 0.0) \{ \\
\quad \quad \ast r = r_{\text{top}}; \\
\quad \} \\
\quad \text{else } \{ \\
\quad \quad \text{temp} = t_{\text{top}} \ast \text{beer}; \\
\quad \quad \text{denom} = 1 - r_{\text{top}} \ast r_{\text{bottom}} \ast \text{beer} \ast \text{beer}; \\
\quad \quad \ast r = r_{\text{top}} + r_{\text{bottom}} \ast \text{temp} \ast \text{temp} \ast \text{denom}; \\
\quad \quad \ast t = t_{\text{bottom}} \ast \text{temp} \ast \text{denom}; \\
\quad \} \\
\}
\]

This code is used in section 149.
153. Total diffuse reflection.

\( R_1 \) calculates the first moment of the Fresnel reflectance using the analytic solution of Walsh. The integral of the first moment of the Fresnel reflection \( (R_1) \) has been found analytically by Walsh, [see Ryde 1931]

\[
R_1 = \frac{1}{2} + \frac{(m - 1)(3m + 1)}{6(m + 1)^2} + \frac{m^2(m^2 - 1)^2}{(m^2 + 1)^3} \log \left( \frac{m - 1}{m + 1} \right)
\]

\[
- \frac{2m^3(m^2 + 2m - 1)}{(m^2 + 1)(m^3 - 1)} + \frac{8m^4(m^2 + 1)}{(m^2 + 1)(m^4 - 1)^2} \log m
\]

where Walsh’s parameter \( m = n_i/n_t \). This equation is only valid when \( n_i < n_t \). If \( n_i > n_t \) then using (see Egan and Hilgeman 1973),

\[
\frac{1 - R_1(n_i/n_t)}{n_t^2} = \frac{1 - R_1(n_t/n_i)}{n_i^2}
\]

or

\[
R(1/m) = 1 - m^2[1 - R(m)]
\]

(Prototype for \( R_1 \) 153) \( \equiv \)

static double \( R_1(\)double \( n_i, \)double \( n_t) \)

This code is used in sections 135 and 154.

\begin{itemize}
\item \textbf{beer}: double, §149.
\item \textbf{denom}: double, §149.
\item \textbf{r}: double *, §148.
\item \textbf{r_bottom}: double, §149.
\item \textbf{r_top}: double, §149.
\item \textbf{t}: double *, §148.
\item \textbf{t_bottom}: double, §149.
\item \textbf{t_top}: double, §149.
\item \textbf{temp}: double, §149.
\end{itemize}
154. (Definition for $R_1$ 154) \(\equiv\)
(Prototype for $R_1$ 153)
{
    \textit{double} m, m2, m4, mm1, mp1, r, temp;
    \textit{if} (ni \equiv nt) \textit{return} 0.0;
    \textit{if} (ni < nt) m = nt/ni;
    \textit{else} m = ni/nt;
    m2 = m * m;
    m4 = m2 * m2;
    mm1 = m - 1;
    mp1 = m + 1;
    temp = (m2 - 1)/(m2 + 1);
    r = 0.5 + mm1 * (3 * m + 1)/6/mp1/mp1;
    r += m2 * temp * temp/(m2 + 1) * log(mm1/mp1);
    r -= 2 * m * m2 * (m2 + 2 * m - 1)/(m2 + 1)/(m4 - 1);
    r += 8 * m4 * (m4 + 1)/(m2 + 1)/(m4 - 1)/(m4 - 1) * log(m);
    \textit{if} (ni < nt) \textit{return} r;
    \textit{else return} (1 - (1 - r)/m2);
}

This code is used in section 135.
155. Diffusion reflection from a glass slide.

\textit{Diffuse\textunderscore Glass\textunderscore R} returns the total diffuse specular reflection from the air-glass-tissue interface

\begin{verbatim}
Prototype for Diffuse\textunderscore Glass\textunderscore R 155) \equiv
double Diffuse\textunderscore Glass\textunderscore R(double nair, double nslide, double nslab)
\end{verbatim}

This code is used in sections 136 and 156.

156. (Definition for Diffuse\textunderscore Glass\textunderscore R 156) \equiv

\begin{verbatim}
Prototype for Diffuse\textunderscore Glass\textunderscore R 155)
{
double raiglass, rglasstissue, rtemp;
raiglass = R1(nair, nslide);
rglasstissue = R1(nslabel, nslab);
rtemp = raiglass \ast rglasstissue;
if (rtemp \geq 1) return 1.0;
else return ((raiglass + rglasstissue - 2 \ast rtemp)/(1 - rtemp));
}
\end{verbatim}

This code is used in section 135.
157. AD Matrix.

This is a part of the core suite of files for the adding-doubling program. Not surprisingly, this program includes routines to manipulate matrices. These routines require that the matrices be stored using the allocation scheme outlined in *Numerical Recipes* by Press et al. I have spent some time optimizing the matrix multiplication routine *Matrix_Multiply* because roughly half the time in any adding-doubling calculation is spent doing matrix multiplication. Lastly, I should mention that all the routines assume a square matrix of size \( n \) by \( n \).

\[
\text{ad_matrx.c 157} \equiv
\begin{align*}
\# \text{ include } & < \text{stddef.h} > \# \text{ include } < \text{math.h} > \# \text{ include } "\text{ad_matrx.h}" \\
\text{include } & "\text{nr_util.h}" \ \text{(Definition for Copy_Matrix 161)} \\
\text{(Definition for One_Minus 163)} \\
\text{(Definition for Transpose_Matrix 165)} \\
\text{(Definition for Diagonal_To_Matrix 167)} \\
\text{(Definition for Right_Diagonal_Multiply 169)} \\
\text{(Definition for Left_Diagonal_Multiply 171)} \\
\text{(Definition for Matrix_Multiply 178)} \\
\text{(Definition for Matrix_Sum 173)} \\
\text{(Definition for Solve 199)} \\
\text{(Definition for Decomp 189)} \\
\text{(Definition for Matrix_Inverse 203)} \\
\text{(Definition for Left_Inverse_Multiply 205)} \\
\text{(Definition for Right_Inverse_Multiply 207)}
\end{align*}
\]

158. In this module I collect up information that needs to be written to the header file \text{ad_matrx.h} so that other source files that want to make use of the function defined here will have the necessary declarations available.

\[
\text{ad_matrx.h 158} \equiv
\begin{align*}
\text{(Prototype for Copy_Matrix 160)}; \\
\text{(Prototype for One_Minus 162)}; \\
\text{(Prototype for Transpose_Matrix 164)}; \\
\text{(Prototype for Diagonal_To_Matrix 166)}; \\
\text{(Prototype for Right_Diagonal_Multiply 168)}; \\
\text{(Prototype for Left_Diagonal_Multiply 170)}; \\
\text{(Prototype for Matrix_Multiply 177)}; \\
\text{(Prototype for Matrix_Sum 172)}; \\
\text{(Prototype for Solve 198)}; \\
\text{(Prototype for Decomp 188)}; \\
\text{(Prototype for Matrix_Inverse 202)}; \\
\text{(Prototype for Left_Inverse_Multiply 204)}; \\
\text{(Prototype for Right_Inverse_Multiply 206)};
\end{align*}
\]
159. Simple Matrix Routines.

160. `Copy_Matrix` replaces the matrix $B$ by $A$

Prototype for `Copy_Matrix 160` ≡

```c
void Copy_Matrix(int n, double **A, double **B)
```

This code is used in sections 158 and 161.

161. (Definition for `Copy_Matrix 161`) ≡

Prototype for `Copy_Matrix 160` ≡

```c
{ 
  double *a_ptr, *b_ptr, *a_last;
  a_last = &A[n][n];
  a_ptr = &A[1][1];
  b_ptr = &B[1][1];
  while (a_ptr ≤ a_last) *b_ptr ++ = *a_ptr ++;
}
```

This code is used in section 157.

162. `One_Minus` replaces the matrix $A$ by $1-A$

Prototype for `One_Minus 162` ≡

```c
void One_Minus(int n, double **A)
```

This code is used in sections 158 and 163.

163. (Definition for `One_Minus 163`) ≡

Prototype for `One_Minus 162` ≡

```c
{ 
  int i, j;
  for (i = 1; i ≤ n; i++) {
    for (j = 1; j ≤ n; j++) A[i][j] *= -1;
    A[i][i] += 1.0;
  }
}
```

This code is used in section 157.

164. `Transpose_Matrix` transposes a matrix.

Prototype for `Transpose_Matrix 164` ≡

```c
void Transpose_Matrix(int n, double **a)
```

This code is used in sections 158 and 165.

---

*Decomp*: void, §188.

*Diagonal_To_Matrix*: void, §166.

*b*: double **, §88.

*Left_Diagonal_Multiply*: void, §170.

*Left_Inverse_Multiply*: void, §204.

*math*: ???, §0.


*Matrix_Multiply*: void, §177.

*Matrix_Sum*: void, §172.

*n*: int, §133.

*Right_Diagonal_Multiply*: void, §168.

*Right_Inverse_Multiply*: void, §206.

*Solve*: void, §198.

*stddef*: ???, §0.
165. (Definition for Transpose_Matrix 165) \equiv
(Prototype for Transpose_Matrix 164)
{
    int i, j;
    double swap;
    for (i = 1; i \leq n; i++) {
        for (j = i + 1; j \leq n; j++) {
            swap = a[i][j];
            a[i][j] = a[j][i];
            a[j][i] = swap;
        }
    }
}

This code is used in section 157.

166. Diagonal_To_Matrix converts a diagonal array to a matrix
(Prototype for Diagonal_To_Matrix 166) \equiv
void Diagonal_To_Matrix(int n, double *Diag, double **Mat)

This code is used in sections 158 and 167.

167. (Definition for Diagonal_To_Matrix 167) \equiv
(Prototype for Diagonal_To_Matrix 166)
{
    int i, j;
    for (i = 1; i \leq n; i++) {
        for (j = 1; j \leq n; j++) Mat[i][j] = 0.0;
        Mat[i][i] = Diag[i];
    }
}

This code is used in section 157.

168. Right_Diagonal_Multiply multiplies the matrix A by the diagonal matrix B, puts the result in C. A and C can be the same matrix

\[ C \leftarrow A \cdot B \]

Note that B is stored as a vector.
(Prototype for Right_Diagonal_Multiply 168) \equiv
void Right_Diagonal_Multiply(int n, double **A, double *B, double **C)

This code is used in sections 158 and 169.

169. (Definition for Right_Diagonal_Multiply 169) \equiv
(Prototype for Right_Diagonal_Multiply 168)
{
    int i, j;
    for (i = 1; i \leq n; i++) {
        for (j = 1; j \leq n; j++) C[i][j] = A[i][j] * B[j];
    }
}

This code is used in section 157.
170. \textit{Left\_Diagonal\_Multiply} multiplies the diagonal matrix $a$ by the matrix $B$, puts the result in $C$. $B$ and $C$ can be the same matrix

\begin{verbatim}
Prototype for Left\_Diagonal\_Multiply
\end{verbatim}

\begin{verbatim}
void Left\_Diagonal\_Multiply(int n, double *A, double **B, double **C)
\end{verbatim}

This code is used in sections 158 and 171.

171. \textit{Definition for Left\_Diagonal\_Multiply}

\begin{verbatim}
Prototype for Left\_Diagonal\_Multiply
\end{verbatim}

\begin{verbatim}
{
    int i, j;
    for (i = 1; i \leq n; i++)
        for (j = 1; j \leq n; j++)
            C[i][j] = A[i][i] \ast B[i][j];
\}
\end{verbatim}

This code is used in section 157.

172. \textit{Matrix\_Sum} adds the two matrices $A$ and $B$, puts the result in $C$. The matrices need not be distinct

\begin{verbatim}
Prototype for Matrix\_Sum
\end{verbatim}

\begin{verbatim}
void Matrix\_Sum(int n, double **A, double **B, double **C)
\end{verbatim}

This code is used in sections 158 and 173.

173. \textit{Definition for Matrix\_Sum}

\begin{verbatim}
Prototype for Matrix\_Sum
\end{verbatim}

\begin{verbatim}
{
    int i, j;
    for (i = 1; i \leq n; i++)
        for (j = 1; j \leq n; j++)
            C[i][j] = A[i][j] + B[i][j];
\}
\end{verbatim}

This code is used in section 157.

\begin{itemize}
    \item \texttt{a: double **, §164.}
    \item \texttt{n: int, §164.}
    \item \texttt{Transpose\_Matrix: void, §164.}
\end{itemize}
174. Matrix Multiplication. This is the crux of this whole unit at present. Most of the time in the adding-doubling algorithm is spent doing matrix multiplication and this implementation has been optimized using pointers.

Matrix\_Multiply multiplies the two matrices \( A \) and \( B \) and puts the result in \( C \). The following routine requires that \( C \) does not occupy the same space as \( B \), but it can be coincident with \( A \). However, all the matrices in the adding-doubling method are square and I did not want to pass three separate dimensions to this routine.

The usual way matrix multiplication uses an algorithm something similar to:

\[
\text{for } (i = 1; i \leq n; i++)
\{ \\
\quad \text{for } (j = 1; j \leq n; j++) \\
\quad \quad C[i][j] = 0.0; \\
\quad \text{for } (k = 1; k \leq n; k++)
\quad \quad C[i][j] += A[i][k] \times B[k][j]; \\
\}\}
\]

175. This has the unfortunate problem that the innermost loop indexes successive columns of \( A \) and successive rows of \( B \). Because indexing successive rows requires something other than a unit increment of the matrix pointer, a different algorithm is used. In this case,

\[
\text{for } (i = 1; i \leq n; i++)
\{ \\
\quad \text{for } (j = 1; j \leq n; j++) \\
\quad \quad C[i][j] = 0.0; \\
\quad \text{for } (k = 1; k \leq n; k++)
\quad \quad C[i][j] += A[i][k] \times B[k][j]; \\
\}\}
\]

176. This particular form of indexing was chosen to take advantage of the row storage of matrices designated by the Numerical Recipes scheme. The innermost loop of the matrix multiplication routine now only requires unit increments of the matrix pointers \( C \) and \( B \).

Explicitly using pointers to the entries in the salient matrices makes this routine roughly 20% faster than when the above implementation is used. Profiling of the code indicates that roughly 45% of the time spent in an adding-doubling calculation is spent in this one routine. Therefore even a modest 20% increase will translate to a ten percent improvement in performance.

Finally, the algorithm can be improved to allow the pointers to \( A \) and \( C \) to be the same. This is sufficient to allow us to avoid allocating an extra matrix here and there. It can easily be adapted to work with “star” multiplication by premultiplying using Right\_Diagonal\_Multiply. The drawbacks are that a vector \( D \) must be allocated on each call. It is also necessary to copy the data from the vector \( D \) to the output matrix \( C \).

177. (Prototype for Matrix\_Multiply 177) \equiv

\[
\text{void Matrix\_Multiply}\!(\text{int } n, \text{double } **A, \text{double } **B, \text{double } **C)
\]

This code is used in sections 158 and 178.
178. (Definition for Matrix_Multiply 178) \equiv
   (Prototype for Matrix_Multiply 177)
   
   \{ 
   (Local variables for Matrix_Multiply 179)
   (Do awkward cases 180)
   (Allocate memory for D 181)
   (Initialization for Matrix_Multiply 182)
   (Multiplying A and B 185)
   (Free memory for D 186)
   \}

This code is used in section 157.

179. (Local variables for Matrix_Multiply 179) \equiv
   double *a_ptr, *a_start;
   double *b_start, *b_last;
   double *c_start, *c_very_last, *c_ptr;
   double *D;
   double *d_start, *d_last;
   register double t, *d_ptr, *b_ptr;
   ptrdiff_t row;

This code is used in section 178.

180. (Do awkward cases 180) \equiv
   if (n \leq 0) {
      nrerror("Non-positive\_dimension\_passed\_to\_Matrix\_Multiply");
   } else if (n \equiv 1) {
      C[1][1] = A[1][1] * B[1][1];
      return;
   }

This code is used in section 178.

181. I need a temporary vector equal to the row length of C to hold intermediate calculations.
    This will allow A and C to point to the same matrix and still yield the correct results.
    (Allocate memory for D 181) \equiv
    \hspace{1cm} D = dvector(1, n);

This code is used in section 178.

\begin{itemize}
\item[A:] double **, §172.
\item[B:] double **, §172.
\item[C:] double **, §172.
\item[D:] double **, §206.
\item[dvector, <nr_util.h>]:
\item[nrerror, <nr_util.h>]:
\item[i: int, §173.]:
\item[j: int, §173.]:
\item[k: int, §241.]:
\item[n: int, §172.]:
\item[Right_Diagonal_Multiply: void, §168.]:
\end{itemize}
Adding-Doubling (Version 1.5): \textsc{Matrix Multiplication}

During the initialization phase, I need to know how far it is from one row to the next row. Because of the peculiar way that \textsc{Numerical Recipes} allocates the matrices, this may and probably is not equal to \( n \). The number of entries is found explicitly by subtracting a pointer to the first entry in row one from the first entry in row two. This assumes that the size of the matrix is at least two. To make this routine bulletproof, this would need to be changed—but I do not think it is really necessary.

\[
\begin{align*}
\text{Initialization for Matrix Multiply} \equiv & \quad a_{\text{start}} = &A[1][1]; \\
b_{\text{last}} = &B[n][1]; \\
row = &A[2][1] - a_{\text{start}}; \\
c_{\text{very last}} = &C[n][n]; \\
d_{\text{start}} = &D[1]; \\
d_{\text{last}} = &D[n];
\end{align*}
\]

This code is used in section 178.

There may be a better way of doing this, but I bet it would depend on specific knowledge about how zero is stored in the computer.

\[
\begin{align*}
\text{Zero D} \equiv & \quad d_{\text{ptr}} = d_{\text{start}}; \\
& \quad \text{while } (d_{\text{ptr}} \leq d_{\text{last}}) \ast d_{\text{ptr}}++ = 0.0;
\end{align*}
\]

This code is used in section 185.

Copy the contents of \( D \) to \( C \). This could potentially be sped up using \texttt{memmove()} but I just want it to work for now.

\[
\begin{align*}
\text{Copy D into C} \equiv & \quad d_{\text{ptr}} = d_{\text{start}}; \\
& \quad c_{\text{ptr}} = c_{\text{start}}; \\
& \quad \text{while } (d_{\text{ptr}} \leq d_{\text{last}}) \ast c_{\text{ptr}}++ = \ast d_{\text{ptr}}++;
\end{align*}
\]

This code is used in section 185.

Here is the heart of the routine. The first row of \( C \) is filled completely, then the routine goes on to the second row and so on. The inner loop is responsible for multiplying \( A[i][k] \) (represented by \( t = \ast a_{\text{ptr}} \)) by every element in row \( i \) and adding it to the appropriate element in row \( i \) of \( C \).

\[
\begin{align*}
\text{Multiplying A and B} \equiv & \quad \text{for } (c_{\text{start}} = \& C[1][1]; \; c_{\text{start}} \leq c_{\text{very last}}; \; c_{\text{start}} == \text{row}) \{ \\
& \quad a_{\text{ptr}} = a_{\text{start}}; \\
& \quad (\text{Zero D}) \; \text{for } (b_{\text{start}} = \& B[1][1]; \; b_{\text{start}} \leq b_{\text{last}}; \; b_{\text{start}} == \text{row}) \{ \\
& \quad \quad t = \ast a_{\text{ptr}}++; \\
& \quad \quad b_{\text{ptr}} = b_{\text{start}}; \\
& \quad \quad d_{\text{ptr}} = d_{\text{start}}; \\
& \quad \quad \text{while } (d_{\text{ptr}} \leq d_{\text{last}}) \ast d_{\text{ptr}}++ += t \ast \ast b_{\text{ptr}}++;
\}
\end{align*}
\]

This code is used in section 178.
186. Dump the memory that was allocated.

\[
\text{Free memory for } D \text{ 186) } \equiv \\
\text{free_dvector}(D, 1, n);
\]

This code is used in section 178.
187. Matrix Decomposition.

188. (Prototype for Decomp 188) ≡
   void Decomp(int n, double **A, double *condition, int *ipvt)
This code is used in sections 158 and 189.

189. Decomp decomposes a double matrix by Gaussian elimination and estimates the condition of the matrix.

   Use solve to compute solutions to linear systems

   On input n is the order of the matrix and A is the matrix to be triangularized.

   On output A contains an upper triangular matrix U and a permuted version of a lower triangular matrix I - L so that (permutation matrix)*A=L*U. condition is an estimate of the condition of A. For the linear system AX = B, changes in A and B may cause changes condition times as large in X. If condition + 1.0 = condition, A is singular to working precision. condition is set to 1.0 · 10^{32} if exact singularity is detected. ipet is the pivot vector ipet(k) is the index of the kth pivot row ipet(n) = \(-1\)^{(numberofinterchanges)}

   (Definition for Decomp 189) ≡
   (Prototype for Decomp 188)
   {
      double t, anorm;
      int i, j, k, m;
      (Do n ≡ 1 case 190)
      (Compute 1-norm of A 191)
      (Gaussian elimination with partial pivoting 192)
      (Check for singularity 196)
   }
   This code is used in section 157.

190. This should probably be fixed to compute the inverse of a non-zero 1by 1 matrix.

   (Do n ≡ 1 case 190) ≡
   ipet[n] = 1;
   if (n ≡ 1) {
      if (A[1][1] ≡ 0) {
         nrerror("1×1 Matrix is Singular, i.e., zero");
         return;
      }
   }
   This code is used in section 189.

191. (Compute 1-norm of A 191) ≡
   anorm = 0.0;
   for (j = 1; j ≤ n; j++) {
      t = 0.0;
      for (i = 1; i ≤ n; i++) t += fabs(A[i][j]);
      if (t > anorm) anorm = t;
   }
   This code is used in section 189.
192. (Gaussian elimination with partial pivoting 192) \(\equiv\)
\[
\begin{aligned}
&\text{for } (k = 1; \ k < n; \ k++) \{ \\
&\quad \langle \text{Find pivot 193} \rangle \\
&\quad \langle \text{Compute multipliers 194} \rangle \\
&\quad \langle \text{Interchange and eliminate by columns 195} \rangle \\
\}
\end{aligned}
\]
This code is used in section 189.

193. (Find pivot 193) \(\equiv\)
\[
\begin{aligned}
m &= k; \\
&\text{for } (i = k + 1; \ i \leq n; \ i++) \\
&\quad \text{if } (\text{fabs}(A[i][k]) > \text{fabs}(A[m][k])) \ m = i; \\
&\quad ipet[k] = m; \\
&\quad \text{if } (m \neq k) \ ipet[n] *= -1; \\
&\quad t = A[m][k]; \\
&\quad A[m][k] = A[k][k]; \\
&\quad A[k][k] = t; \quad /* \text{skip step if pivot is zero */} \\
&\quad \text{if } (t \equiv 0) \ \text{continue};
\end{aligned}
\]
This code is used in section 192.

194. (Compute multipliers 194) \(\equiv\)
\[
\begin{aligned}
&\text{for } (i = k + 1; \ i \leq n; \ i++) \ A[i][k] /= -t;
\end{aligned}
\]
This code is used in section 192.

195. (Interchange and eliminate by columns 195) \(\equiv\)
\[
\begin{aligned}
&\text{for } (j = k + 1; \ j \leq n; \ j++) \{ \\
&\quad t = A[m][j]; \\
&\quad A[m][j] = A[k][j]; \\
&\quad A[k][j] = t; \\
&\quad \text{if } (t \equiv 0) \ \text{continue}; \\
&\quad \text{for } (i = k + 1; \ i \leq n; \ i++) \ A[i][j] += A[i][k] * t;
\}
\end{aligned}
\]
This code is used in section 192.

196. (Check for singularity 196) \(\equiv\)
\[
\begin{aligned}
&\ast \text{condition } = 1.0; \\
&\text{for } (k = 1; \ k \leq n; \ k++) \{ \\
&\quad \text{if } (A[k][k] \equiv 0.0) \{ \\
&\quad\quad \ast \text{condition } = 1 \cdot 10^{32}; \\
&\quad\quad \text{return}; \\
&\quad\} \\
\}
\end{aligned}
\]
This code is used in section 189.

\[B: \text{double }**, \text{§177.} \quad \text{fabs, <math.h>.} \quad \text{nrerror, <nr_util.h>.}\]
197. Solving systems of equations.

198.  (Prototype for Solve 198) ≡
void Solve(int n, double **A, double *B, int *ipet)

This code is used in sections 158 and 199.

199.  This procedure finds the solution of the linear system $AX = B$ Don't use if Decomp has found a singularity

On input $n$ is the order of matrix, $A$ is the triangularized matrix obtained form Decomp. $B$ is the right hand side vector and $ipet$ is the pivot vector obtained from Decomp

On output $B$ is the solution vector $X$.

(Definition for Solve 199) ≡
(Prototype for Solve 198)
{
  int i, k, m;
  double t;
  (Forward elimination 200)
  (Back substitution 201)
}

This code is used in section 157.

200.  (Forward elimination 200) ≡
for (k = 1; k < n; k++) {
  m = ipet[k];
  t = B[m];
  B[m] = B[k];
  B[k] = t;
  for (i = k + 1; i ≤ n; i++) B[i] += A[i][k] * t;
}
This code is used in section 199.

201.  (Back substitution 201) ≡
for (k = n; k > 1; k--) {
  B[k] /= A[k][k];
  t = -B[k];
  for (i = 1; i < k; i++) B[i] += A[i][k] * t;
}  
B[1] /= A[1][1];
This code is used in section 199.

202.  Finds the inverse of the matrix $A$ (of order $n$) and stores the answer in $Ainv$.
(Prototype for Matrix_Inverse 202) ≡
void Matrix_Inverse(int n, double **A, double **Ainv)

This code is used in sections 158 and 203.

203.  (Definition for Matrix_Inverse 203) ≡
(Prototype for Matrix_Inverse 202)
{
Adding-Doubling (Version 1.5): SOLVING SYSTEMS OF EQUATIONS

int *ipvt;
int i, j;
double *work;
double condition;
ipvt = ivector(1, n);
Decomp(n, A, &condition, ipvt);
if (condition ≡ (condition + 1) ∨ condition ≡ 1 · 10^{32}) {
    free_ivector(ipvt, 1, n);
nrerror("Singular Matrix, ... failed in Inverse_Multiply
")
}
work = dvector(1, n);
for (i = 1; i ≤ n; i++) {
    for (j = 1; j ≤ n; j++) work[j] = 0.0;
    work[i] = 1.0;
    Solve(n, A, work, ipvt);
    for (j = 1; j ≤ n; j++) Ainv[j][i] = work[j];
}
free_dvector(work, 1, n);
free_ivector(ipvt, 1, n);
}

This code is used in section 157.

204. ⟨Prototype for Left_Inverse_Multiply 204⟩ ≡

void Left_Inverse_Multiply(int n, double **D, double **C, double **A)

This code is used in sections 158 and 205.
205. \textit{Left\textunderscore Inverse\textunderscore Multiply} computes $A = C \cdot D^{-1}$ where $A$, $C$ and $D$ are all $n$ by $n$ matrices. This is faster than inverting and then multiplying by a factor of six. Space for $A$ should be allocated before calling this routine.

\texttt{(Definition for Left\textunderscore Inverse\textunderscore Multiply 205) \equiv}
\texttt{(Prototype for Left\textunderscore Inverse\textunderscore Multiply 204) \{ int \cdot ipvt; int i, j; double \cdot work; double condition; Transpose\_Matrix(n, D); ipvt = ivector(1, n); Decomp(n, D, &condition, ipvt); /* Check for singular result */ if \{ condition \equiv (condition + 1) \lor condition \equiv 1 \cdot 10^{32} \} \{ nrerror("Singular \_Matrix_\ldots failed in Left\_Inverse\_Multiply\n"); \} work = dvector(1, n); for \{ i = 1; i \leq n; i++ \} /* Cycle through all the row in \_C */ for \{ j = 1; j \leq n; j++ \} /* put a row of \_C into \_work */ work[j] = C[i][j]; /* and avoid a Transpose Matrix */ Solve(n, D, work, ipvt); /* Again avoiding a Transpose Matrix */ A[i][j] = work[j]; /* stuff the results into a row of \_A */ \} free_dvector(work, 1, n); free_ivector(ipvt, 1, n); \}}

This code is used in section 157.

206. \texttt{(Prototype for Right\textunderscore Inverse\textunderscore Multiply 206) \equiv}
\texttt{void Right\textunderscore Inverse\textunderscore Multiply(int n, double **D, double **C, double **A) \}

This code is used in sections 158 and 207.

207. \textit{Right\textunderscore Inverse\textunderscore Multiply} computes $A = D^{-1} \cdot C$ where $A$, $C$ and $D$ are all $n$ by $n$ matrices. This is faster than inverting and then multiplying by a factor of six. Space for $A$ should be allocated before calling this routine.

\texttt{(Definition for Right\textunderscore Inverse\textunderscore Multiply 207) \equiv}
\texttt{(Prototype for Right\textunderscore Inverse\textunderscore Multiply 206) \{ int \cdot ipvt; int i, j; double \cdot work; double condition; ipvt = ivector(1, n); Decomp(n, D, &condition, ipvt); /* Check for singular result */ if \{ condition \equiv (condition + 1) \lor condition \equiv 1 \cdot 10^{32} \} \{ \}
Adding-Doubling (Version 1.5): SOLVING SYSTEMS OF EQUATIONS

```c
free_ivector(ipvt, 1, n);
nrerror("Singular_Matrix,...failed_in_Right_Inverse_Multiply\n");
}
work = dvector(1, n);
for (i = 1; i <= n; i++) {
    /* Cycle through all the rows */
    for (j = 1; j <= n; j++)
        /* put a column of C into work */
        work[j] = C[j][i];
    Solve(n, D, work, ipvt);
    for (j = 1; j <= n; j++)
        /* stuff the results into a column of A */
        A[j][i] = work[j];
}
free_dvector(work, 1, n);
free_ivector(ipvt, 1, n);
```

This code is used in section 157.

---

A: double **, §204.
C: double **, §204.
D: double **, §204.
Decomp: void, §188.
dvector, <nr_util.h>.
free_dvector, <nr_util.h>.
free_ivector, <nr_util.h>.
ivector, <nr_util.h>.
Left_Inverse_Multiply: void, §204.
Solve: void, §198.
Transpose_Matrix: void, §164.

n: int, §204.
nrerror, <nr_util.h>.
208. AD Radau Quadrature.

This global variable is needed because the degree of the Legendre Polynomial must be known.

The routine \textit{Radau} stores the correct value in this.

\texttt{\#define NSLICES 128}
\texttt{\#define EPS 1 \cdot 10^{-16}}

\texttt{\langle ad_radau.c \rangle \equiv}

\{ Preprocessor definitions \}
\# include "ad_radau.h" \# include "nr_rtsaf.h" \# include "nr_util.h" \# include "nr_zbrak.h" static int local_n_size;

\{ Prototype for \textit{Pn} and \textit{Pnm1} \}
\{ Prototype for \textit{Pnd} \}
\{ Prototype for \textit{phi} \}
\{ Definition for \textit{Pn} and \textit{Pnm1} \}
\{ Definition for \textit{Pnd} \}
\{ Definition for \textit{phi} \}
\{ Definition for \textit{Radau} \}

209. \texttt{\langle ad_radau.h \rangle \equiv}

\{ Prototype for \textit{Radau} \}
210. Introduction.

The adding-doubling method is based on numerical integration of functions using quadrature,

\[ \int_0^1 f(\nu, \nu') d\nu' = \sum_{k=1}^{N} w_k f(x_k) \]

The values of the quadrature points \( x_k \) and the weights \( w_k \) are chosen in such a way that the integral is evaluated exactly for a polynomial of order \( 2N - 1 \) (or possibly \( 2N - 2 \) depending on the quadrature method). Using \( N \) quadrature points (Gaussian) is equivalent to the spherical harmonic method of order \( P_{N-1} \), i.e. four quadrature points corresponds to the \( P_3 \) method. The specific choice of quadrature methods for samples with mismatched boundaries is described in the next section.

Total internal reflection causes problems by changing the effective range of integration. Usually, adding-doubling integrals range from 0 to 1, since the angle varies from \( \pi/2 \) to 0 and therefore the cosine varies from 0 to 1. The integrations are calculated using numerical quadrature, and the quadrature angles are optimized for this range. If the cosine of the critical angle is denoted by \( \nu_c \) for a boundary layer with total internal reflection, then the effective range of integration is reduced to \( \nu_c \) to 1 (because the rest of the integration range is now zero). To maintain integration accuracy, the integral is broken into two parts and each is evaluated by quadrature over the specified subrange,

\[ \int_0^1 A(\nu, \nu')B(\nu', \nu'') d\nu' = \int_0^{\nu_c} A(\nu, \nu')B(\nu', \nu'') d\nu' + \int_{\nu_c}^1 A(\nu, \nu')B(\nu', \nu'') d\nu'. \]

Here \( A(\nu, \nu') \) and \( B(\nu, \nu') \) represent reflection or transmission functions, and clearly if either is identically zero for values of \( \nu \) less than \( \nu_c \), the integration range is reduced. The calculations in this paper used Gaussian quadrature for the range from 0 to \( \nu_c \), thereby avoiding calculations at both endpoints (in particular, the angle \( \nu = 0 \) is avoided, which may cause division by zero). Radau quadrature is used for the range from \( \nu_c \) to 1, so \( \nu = 1 \) could be specified as a quadrature point. Each part of the integration range gets half of the quadrature points; when no critical angle exists, Radau quadrature is used over the entire range.

Radau quadrature requires finding the \( n \) roots of the following equation

\[ P_{n-1}(x_i) + \frac{x_i - 1}{n} P'_{n-1}(x_i) = 0 \]

Here \( P_n(x) \) is the \( n \)th Legendre polynomial of order zero and \( P'_{n-1}(x_i) \) is the first derivative of the \( n-1 \) Legendre polynomial. These roots are the required quadrature points for the integration range -1 to 1. The \( n \)th integration angle \( \nu_n \) corresponds with \( x_n = -1 \) (normal incidence).
211. **Radau.** *Radau* calculates the $n$ quadrature points $x_i$ and weights $w_i$.

(Prototype for *Radau* 211) $\equiv$

```c
void Radau(double x1, double x2, double *x, double *w, int n)
```

This code is used in sections 209 and 212.

212. (Definition for *Radau* 212) $\equiv$

(Prototype for *Radau* 211)

```c
{x[n] = -1.0;
 w[n] = 2.0/(n * n);
 switch (n) {
 case 2: (Values for $n \equiv 2$ 230)
 case 4: (Values for $n \equiv 4$ 231)
 case 8: (Values for $n \equiv 8$ 232)
 case 16: (Values for $n \equiv 16$ 233)
 default: (Values for arbitrary $n$ 214)
 }
 (Scale values 213)
}
```

This code is used in section 208.

213. The code to scale values is easy. *Radau* quadrature is defined over the range $-1$ to $1$. Here we just linearly scale the width of each interval and weight as appropriate. To modify for the range $\nu_c$ to $1$ the following relations are needed to find the necessary integration angles $\nu_i$ and weights $w_i$

$$\nu_i = \frac{1 + \nu_c - (1 - \nu_c)x_i}{2}$$

and

$$w_i = \frac{1 - \nu_c}{(1 - x_i)\sqrt{P_n'}(x_i)}$$

(Protocol values 213) $\equiv$

```c
{ double xm, xl;
 int i;
 xm = (x2 + x1) * 0.5;
 xl = (x2 - x1) * 0.5;
 for (i = 1; i <= n; i++) {
 x[i] = xm - xl * x[i];
 w[i] = xl * w[i];
 }
```

This code is used in section 212.

214. Here is the method for finding *Radau* quadrature points for non-tabulated values.

(Values for arbitrary $n$ 214) $\equiv$

```c
{
int i, nb, ndiv;
double z;
double *xb1, *xb2;

(Allocate memory for Radau 215)
(Bracket roots 216)
(Find roots and weights 217)
(Free memory for Radau 218)
break;
}

This code is used in section 212.

215. (Allocate memory for Radau 215) ≡
xb1 = dvector(1, NSLICES);
xb2 = dvector(1, NSLICES);
This code is used in section 214.

216. Bracket \( n - 1 \) roots, double \( ndiv \) if not enough roots are found.
(Bracket roots 216) ≡
local n_size = n;
if (2 * n > NSLICES) \( ndiv = \text{NSLICES} \);
else \( ndiv = 2 * n \);
do {
    nb = n - 1;
    zbrak(phi, -1.0, 1.0, ndiv, xb1, xb2, \&nb);
    ndiv *= 2;
} while (nb < n - 1 \&\& ndiv ≤ NSLICES);
if (nb < n - 1) nrerror("Cannot find enough roots for Radau quadrature");
This code is used in section 214.

217. Find the roots with an accuracy \( \text{EPS} \) and store them in the array \( x \). Put them in backwards so that \( x[n] = -1 \) is in the correct spot.
(Find roots and weights 217) ≡
for (i = 1; i < n; i++) {
    z = rtsafe(phi_and_phiprime, xb1[i], xb2[i], EPS);
    x[n - i] = z;
    w[n - i] = 1/((1 - z) * DSQR(Pnd(n - 1, z)));
}

This code is used in section 214.

218. (Free memory for Radau 218) ≡
free_dvector(xb1, 1, NSLICES);
free_dvector(xb2, 1, NSLICES);
This code is used in section 214.

---

`dvector`, `<nr_util.h>`
`EPS` = \(1 \cdot 10^{-16}\), §208.
`free_dvector`, `<nr_util.h>`

`local_n_size`: static int, §208.

`nrerror`, `<nr_util.h>`

`NSLICES` = 128, §208.

`phi_and_phiprime`: static double, §227.
`rtsafe`, `<nr_rtsaf.h>`.
`zbrak`, `<nr_zbrak.h>`.

`DSQR`, `<nr_util.h>`.
`Pnd`: static double, §221.
`rtsafe`, `<nr_rtsaf.h>`.
219. \( P_{n\text{and}P_{n-1}} \) returns \( P_n(x) \) and \( P_{n-1}(x) \)

\[
\langle \text{Prototype for } P_{n\text{and}P_{n-1}} \rangle 
\equiv
\text{static void } P_{n\text{and}P_{n-1}}(\text{int } n, \text{double } x, \text{double } *P_{n-1}, \text{double } *P_n)
\]

This code is used in sections 208 and 220.

220. \( \langle \text{Definition for } P_{n\text{and}P_{n-1}} \rangle \equiv \langle \text{Prototype for } P_{n\text{and}P_{n-1}} \rangle \)

\[
\{ 
\text{int } k;
\text{double } Pk, Pkp1;
\text{double } Pkm1 = 1.0;
*P_{n-1} = 1.0;
*P_n = 1.0;
\text{if } (x \geq 1.0) \text{ return; }
\text{if } (x \leq -1.0) \text{ } x = -1;
Pk = x;
\text{for } (k = 1; \ k < n; \ k++) \{ 
\text{Pkp1} = ((2* k + 1) * x * Pk - k * Pkm1)/(k + 1);
Pkm1 = Pk;
Pk = Pkp1;
\}
\text{Pkm1} = Pkm1;
\text{*P_n} = Pk;
\}
\]

This code is used in section 208.

221. To calculate the weights for the quadrature points we need to evaluate the first derivative of the Legendre polynomial. To do this we use a recurrence relation given by H. H. Michels, in “Abscissas and weigh coefficients for Lobatto quadrature,” \textit{Math Comp}, 17, 237-244 (1963).

\[
\langle \text{Prototype for } P_{n\text{d}} \rangle \equiv
\text{static double } P_{n\text{d}}(\text{int } n, \text{double } x)
\]

This code is used in sections 208 and 222.

222. \( \langle \text{Definition for } P_{n\text{d}} \rangle \equiv \langle \text{Prototype for } P_{n\text{d}} \rangle \)

\[
\{ 
\text{double } p, \text{pminus, pplus};
\text{int } i;
\text{if } (x > 1.0) \{ 
x = 1;
\}
\text{else if } (x < -1.0) \{ 
x = -1;
\}
pminus = 0;
p = 1;
\text{if } (n \leq 0) \text{ return } pminus;
\}
\]
Adding-Doubling (Version 1.5): RADAU

for (i = 1; i < n; i++) {
    pplus = ((2 * i + 1) * x * p - (i + 1) * pminus) / i;
    pminus = p;
    p = pplus;
} return p;

This code is used in section 208.

223. To use Newton’s method to find the roots of

\[ \phi_{n-1}(x) = \frac{P_{n-1}(x) + P_n(x)}{1 + x} \]

we need to find the derivative. This is

\[ \phi'_{n-1}(x) = \frac{P'_{n-1}(x) + P'_n(x)}{1 + x} - \frac{P_{n-1}(x) + P_n(x)}{(1 + x)^2} \]

Now we can use our recurrence relation

\[ (1 - x^2)P'_{n-1}(x) = nxP_{n-1}(x) - nP_n(x) \]

To eliminate the derivative terms in the above equation to get

\[ \phi'_{n-1} = \frac{(nx + x - 1)P_{n-1}(x) + (nx + 2x - n - 1)P_n(x) - (n + 1)P_{n+1}(x)}{(1 - x)(1 + x)^2} \]

The higher order Legendre Polynomial can be eliminated using

\[ (n + 1)P_{n+1}(x) = (2n + 1)xP_n(x) - nP_{n-1}(x) \]

to get

\[ \phi'_{n-1} = \frac{(nx + x + n - 1)P_{n-1}(x) + (-nx + x - n - 1)P_n(x)}{(1 - x)(1 + x)^2} \]

And therefore we just call the routine that will return \( P_n(x) \) and \( P_{n-1}(x) \) and multiply by the appropriate factors to obtain both terms.

The only problem is when \( x = 1 \) or \( x = -1 \). Then we get this spurious division by zero. So we special case these and evaluate them elsewhere.

(Prototype for \( \phi_{\text{and} \text{phiprime}} \) 223) \( \equiv \)

\[ \text{static void phi_and_phiprime(double x, double *phi, double *phiprime)} \]

This code is used in sections 208 and 224.
224. \( \langle \text{Definition for } \phi_{\text{and } \phi'} \ 224 \rangle \equiv \langle \text{Prototype for } \phi_{\text{and } \phi'} \ 223 \rangle \) \\\n\{ \\\n\hspace{0.5em} \textbf{double} \ Pn, \ Pnm1; \\\n\hspace{0.5em} \textbf{int} \ n; \\\n\hspace{0.5em} n = \text{local}_n.size; \\\n\hspace{0.5em} \textbf{if} \ (x \geq 1.0) \ \{ \\\n\hspace{1.5em} \langle \text{Phi and phi prime at } x = 1 \ 225 \rangle \} \\\n\hspace{0.5em} \textbf{else if} \ (x \leq -1.0) \ \{ \\\n\hspace{1.5em} \langle \text{Phi and phi prime at } x = -1 \ 225 \rangle \} \\\n\hspace{0.5em} \textbf{else} \ { \\\n\hspace{1.5em} \ Pn_{\text{and } Pnm1} \ (n, x, \& Pnm1, \& Pn); \\\n\hspace{1.5em} \phi = (Pn + Pnm1)/(1 + x); \\\n\hspace{1.5em} \phi' = ((n + x + x + n) * Pnm1 + (-n * x + x - n - 1) * Pn)/(1 + x)/(1 + x)/(1 - x); \} \\\n\} \\\nThis code is used in section 208.

225. To find \( \phi(1) \) and \( \phi'(1) \) we need to recall a few facts about Legendre polynomials. First, \( P_n(1) = 1 \).

Therefore \( \phi(1) = 1 \).

The value of the first derivative is somewhat trickier. Recall that the Legendre polynomials are solutions to
\[(1 - x^2)P''_n(x) - 2xP'_n(x) + n(n+1)P_n(x) = 0\]
Now if \( x = 1 \) then the first term on the left hand side will be zero. Therefore
\[P'_n(1) = \frac{n(n+1)}{2}\]

Therefore \( \phi'_{n-1}(1) = \frac{n^2 - 1}{2} \).

\langle \text{Phi and phi prime at } x = 1 \ 225 \rangle \equiv \\\n\{ \\\n\hspace{0.5em} \phi = 1; \\\n\hspace{0.5em} \phi' = \frac{n^2 - 1}{2}; \} \\\nThis code is used in section 224.
226. To evaluate $\phi(-1)$ we must return to the original definition, i.e. So

$$\phi_{n-1}(x) = P_{n-1}(x) + \frac{x-1}{n} P'_{n-1}(x)$$

To evaluate this we need to remember some stuff, namely that

$$P_n(-x) = (-1)^n P_n(x) \quad \text{so} \quad P_n(-1) = (-1)^n$$

The value of the first derivative is again obtained from the differential equation and

$$P'_n(-1) = -\frac{n(n+1)}{2} P_n(-1) = (-1)^n \frac{n(n+1)}{2}$$

Now we just substitute to get

$$\phi_{n-1}(-1) = (-1)^{n-1} \cdot n$$

The first derivative is more difficult. Mathematica says that it is

$$\phi'_{n-1}(-1) = (-1)^n \frac{n(1-n^2)}{4}$$

⟨ Phi and phi at $x = -1$ 226⟩ ≡

*phi = n;
*phiprime = -n * (1 - n * n)/4;
if (n % 2 ≠ 1) {
    *phi *= -1;
    *phiprime *= -1;
}

This code is used in section 224.
Adding-Doubling (Version 1.5): RADAU

\section*{227.} For Radau quadrature, we want to find the \(n - 1\) roots of

\[\phi_{n-1}(x) = P_{n-1}(x) + \frac{x-1}{n} P'_{n-1}(x)\]

F. B. Hildebrand notes that by using a recurrence formula this becomes

\[\phi_{n-1}(x) = \frac{P_{n-1}(x) + P_n(x)}{1 + x}\]

This is particularly convenient, because we must find \(P_{n-1}(x)\) before we can find \(P_n(x)\) and this is exactly what \texttt{Pn and Pnm1} does.

It is noteworthy that this routine uses the recurrence formula

\[P_{n+1}(x) = \frac{(2n + 1)xP_n(x) - nP_{n-1}(x)}{n + 1}\]

to calculate the Legendre polynomial \(P_n(x)\). This recurrence relation is given in H. H. Michels, “Abscissas and weight coefficients for Lobatto quadrature,” \textit{Math Comp}, \textbf{17}, 237-244 (1963).

\texttt{Prototype for phi 227} \equiv \texttt{static double phi(double x)}

This code is used in sections 208 and 228.

\section*{228.} \texttt{(Definition for phi 228) \equiv}

\texttt{(Prototype for phi 227)}

\{\texttt{double Pn, Pnm1;} \}

\texttt{if} \ ((x \leq -1.0))

\texttt{if} \ ((\texttt{local}._\texttt{n}._\texttt{size} \% 2 \neq 1) \texttt{return} \ (-\texttt{local}._\texttt{n}._\texttt{size});}

\texttt{else} \texttt{return} \ ((\texttt{local}._\texttt{n}._\texttt{size});}

\texttt{Pn and Pnm1(\texttt{local}._\texttt{n}._\texttt{size}, x, \&Pnm1, \&Pn);}

\texttt{return} \ ((\texttt{Pn} + \texttt{Pnm1})/(1 + x));

\}

This code is used in section 208.
229. **Radau Tables.**

Here is a selection of commonly used number of quadrature points.

230. \( \langle \text{Values for } n \equiv 2 \ 230 \rangle \equiv \)
\[
\begin{align*}
  x[1] &= 0.3333333333333334; \\
  w[1] &= 1.500000000000000; \\
  \text{break};
\end{align*}
\]

This code is used in section 212.

231. \( \langle \text{Values for } n \equiv 4 \ 231 \rangle \equiv \)
\[
\begin{align*}
  x[3] &= -0.5753189235216942; \\
  x[2] &= 0.1810662711185306; \\
  x[1] &= 0.8228240897459921; \\
  w[3] &= 0.657688639601182; \\
  w[2] &= 0.776389376863437; \\
  w[1] &= 0.4409244223535367; \\
  \text{break};
\end{align*}
\]

This code is used in section 212.

232. \( \langle \text{Values for } n \equiv 8 \ 232 \rangle \equiv \)
\[
\begin{align*}
  x[7] &= -0.8874748789261557; \\
  x[6] &= -0.6395186165262152; \\
  x[5] &= -0.2947505657736607; \\
  x[4] &= 0.0943072526611108; \\
  x[3] &= 0.4684203544308211; \\
  x[2] &= 0.7706418936781916; \\
  x[1] &= 0.9550412271225750; \\
  w[7] &= 0.1853581548029793; \\
  w[6] &= 0.3041306206467856; \\
  w[5] &= 0.3765175453891186; \\
  w[4] &= 0.3915721674524935; \\
  w[3] &= 0.3470147956345014; \\
  w[2] &= 0.2496479013298649; \\
  w[1] &= 0.1145088147442572; \\
  \text{break};
\end{align*}
\]

This code is used in section 212.

---

**local_n_size**: `static int`, §208.

**Pn_and_Pnm1**: `static void`, §219.

**w**: `double *`, §211.

**n**: `int`, §224.
233. (Values for $n \equiv 16 \ 233$) \equiv

\begin{align*}
x[15] &= -0.9714610905263484; \\
x[14] &= -0.9054008198116666; \\
x[13] &= -0.8045734013587561; \\
x[12] &= -0.6728619212112202; \\
x[11] &= -0.5153294780626855; \\
x[10] &= -0.33803039000599197; \\
x[9] &= -0.1477783218133717; \\
x[8] &= 0.0481153830735303; \\
x[7] &= 0.2421226227060438; \\
x[6] &= 0.4267878274849459; \\
x[5] &= 0.5950144898997919; \\
x[4] &= 0.7403379488928179; \\
x[3] &= 0.8571740937696823; \\
x[2] &= 0.9410354027041150; \\
x[1] &= 0.988716220549766; \\
&w[15] &= 0.0477022269476863; \\
w[14] &= 0.0839852814444965; \\
w[13] &= 0.1170203531038591; \\
w[12] &= 0.145555452202026; \\
w[11] &= 0.1684963978499219; \\
w[10] &= 0.184961781486653; \\
w[10] &= 0.184961781486653; \\
w[9] &= 0.1943190897115679; \\
w[8] &= 0.1962087882390318; \\
w[7] &= 0.1905582942553547; \\
w[6] &= 0.1775847927527395; \\
w[5] &= 0.1577869218042020; \\
w[4] &= 0.1319256999330681; \\
w[3] &= 0.1009956796217840; \\
w[2] &= 0.066189586101364; \\
w[1] &= 0.0288971390168143; \\
\text{break};
\end{align*}

This code is used in section 212.
234. **AD Phase Function.** This section contains all the routines associated with generating the necessary matrices for Henyey-Greenstein phase functions. This is the place to put code to implement other phase functions.

\begin{verbatim}
#include <stdlib.h>
#include <math.h>
#include "nr_util.h"
#include "ad_globl.h"
#include "ad_phase.h"
\end{verbatim}

(Definition for Get Phi 240)

235. \begin{verbatim}
\end{verbatim}

(Prototype for Get Phi 239);
236. **Redistribution function.** The single scattering phase function $p(\nu)$ for a tissue determines the amount of light scattered at an angle $\nu = \cos \theta$ from the direction of incidence. The subtended angle $\nu$ is the dot product of the unit vectors $\hat{s}_i$ and $\hat{s}_j$

$$\nu = \hat{s}_i \cdot \hat{s}_j = \nu_i \nu_j + \sqrt{1 - \nu_i^2} \sqrt{1 - \nu_j^2} \cos \phi$$

where $\hat{s}_i$ is the incident and $\hat{s}_j$ is the scattered light directions.

The redistribution function $h_{ij}$ determines the fraction of light scattered from an incidence cone with angle $\nu_i$ into a cone with angle $\nu_j$. The redistribution function is calculated by averaging the phase function over all possible azimuthal angles for fixed angles $\nu_i$ and $\nu_j$:

$$h(\nu_i, \nu_j) = \frac{1}{2\pi} \int_0^{2\pi} p(\nu_i \nu_j + \sqrt{1 - \nu_i^2} \sqrt{1 - \nu_j^2} \cos \phi) \, d\phi$$

Note that the angles $\nu_i$ and $\nu_j$ may also be negative (light travelling in the opposite direction). The full redistribution matrix may be expressed in terms a $2 \times 2$ matrix of $n \times n$ matrices

$$h = \begin{bmatrix} h^{--} & h^{-+} \\ h^{++} & h^{+-} \end{bmatrix}$$

The first plus or minus sign indicates the sign in front of the incident angle and the second is the sign of the direction of the scattered light.

When the cosine of the angle of incidence or exitance is unity ($\nu_i = 1$ or $\nu_j = 1$), then the redistribution function $h(1, \nu_j)$ is equivalent to the phase function $p(\nu_j)$. In the case of isotropic scattering, the redistribution function is a constant

$$h(\nu_i, \nu_j) = p(\nu) = \frac{1}{4\pi}.$$  

For Henyey-Greenstein scattering, the redistribution function can be expressed in terms of the complete elliptic integral of the second kind $E(x)$

$$h(\nu_i, \nu_j) = \frac{2}{\pi} \frac{1 - g^2}{(\alpha - \gamma)\sqrt{\alpha + \gamma}} E\left(\sqrt{\frac{2\gamma}{\alpha + \gamma}}\right)$$

where $g$ is the average cosine of the Henyey-Greenstein phase function and

$$\alpha = 1 + g^2 - 2g\nu_i\nu_j \quad \text{and} \quad \gamma = 2g\sqrt{1 - \nu_i^2} \sqrt{1 - \nu_j^2}$$

The function $E(x)$ may be calculated using algorithms found in Press et al. This method of calculating the phase function is slower than the method that is used in this program.

Other phase functions require numerical integration of the phase function. If the phase function is highly anisotropic, then the integration over the azimuthal angle is particularly difficult and care must be taken to ensure that the integration is accurate. This is important because errors in the redistribution function enter directly into the reflection and transmission matrices for thin layers. Any errors will be doubled with each successive addition of layers and small errors will rapidly increase.
An alternate way to calculate the redistribution function is the $\delta-M$ method of Wiscombe. This method works especially well for highly anisotropic phase functions. The number of quadrature points is specified by $M$. The $\delta-M$ method approximates the true phase function by a phase function consisting of a Dirac delta function and $M-1$ Legendre polynomials

$$p^\ast(\nu) = 2g^M\delta(1-\nu) + (1-g^M) \sum_{k=0}^{M-1} (2k+1)\chi_k^\ast P_k(\nu)$$

where

$$\chi_k^\ast = \frac{\chi_k - g^M}{1 - g^M} \quad \text{and} \quad \chi_k = \frac{1}{2} \int_0^1 p(\nu)P_k(\nu)\,d\nu.$$

When the $\delta-M$ method substitutes $p^\ast(\nu) \rightarrow p(\nu)$, then both the albedo and optical thickness must also be changed, $a^\ast \rightarrow a$ and $\tau^\ast \rightarrow \tau$. This approximation is analogous to the similarity transformation often used to improve the diffusion approximation by moving a part $(g^M)$ of the scattered light into the unscattered component. The new optical thickness and albedo are

$$\tau^\ast = (1 - ag^M)\tau \quad \text{and} \quad a^\ast = a \frac{1 - g^M}{1 - ag^M}.$$

This is equivalent transforming the scattering coefficient as $\mu^\ast_s = \mu_s(1 - g^M)$. The redistribution function can now be written as

$$h^\ast(\nu_i,\nu_j) = \sum_{k=0}^{M-1} (2k+1)\chi_k^\ast P_k(\nu_i)P_k(\nu_j)$$

For the special case of a Henyey-Greenstein phase function,

$$\chi_k^\ast = \frac{g^k - g^M}{1 - g^M}.$$

Calculate the renormalization matrix for a Henyey-Greenstein phase function using the delta-M method. This version has been optimized for isotropic and Henyey-Greenstein phase functions.

\noindent\textbf{238.} \textit{(Prototype for \texttt{Get\_Phi})} \equiv \textbf{void \texttt{Get\_Phi(int n, int phase\_function, double g, double **h)}}

This code is used in sections 235 and 240.

\noindent\textit{n: int, \S 224.}
240. (Definition for \textit{Get\_Phi} 240) \equiv

\text{Prototype for \textit{Get\_Phi} 239}

\{
\text{Local variables for \textit{Get\_Phi} 241}
\text{Test for bad calling parameters 242}
\text{Initialize the phase function matrix 243}
\text{We’re done if phase function is isotropic 244}
\text{Calculate the quadrature coefficients 245}
\text{Create Legendre Polynomial matrix 246}
\text{Calculate the coefficients 250}
\text{Add the symmetric part of the matrix 251}
\text{(Free } p \text{ and chi 252)}
\}

This code is used in section 234.

241. (Local variables for \textit{Get\_Phi} 241) \equiv

\text{int } i, j, k;
\text{double } g2M, gk, x;
\text{double } *\text{chi} ;
\text{double } **\text{p} ;

This code is used in section 240.

242. (Test for bad calling parameters 242) \equiv

\text{if } (g \neq 0 \land \text{phase\_function } \neq \text{HENYEY\_GREENSTEIN})
\text{AD\_error("Only the Henyey–Greenstein phase\_function has been implemented\n")};
\text{if } (\text{fabs}(g) \geq 1) \text{ AD\_error("Get\_Phi was called with a bad g\_calc value");}

This code is used in section 240.

243. (Initialize the phase function matrix 243) \equiv

\text{for } (i = -n; i \leq n; i++)
\text{for } (j = -n; j \leq n; j++)
\text{h}[i][j] = 1; /* zero the zero column and zero row */
\text{for } (i = -n; i \leq n; i++)
\text{h}[i][0] = 0.0;
\text{h}[0][i] = 0.0;
\}

This code is used in section 240.

244. (We’re done if phase function is isotropic 244) \equiv

\text{if } (g \equiv 0) \text{ return;}

This code is used in section 240.

245. To avoid extra calculation let’s define

\text{chi}[k] \equiv (2k + 1)\chi_k^*

This will slightly simplify things later on

\text{(Calculate the quadrature coefficients 245) \equiv}
\text{chi = \textit{devector}(1, n);
Adding-Doubling (Version 1.5): REDISTRIBUTION FUNCTION

\[ \text{g2M} = \text{pow}(g, n); \]
\[ g_k = 1.0; \]
\[ \text{for } (k = 1; k < n; k++) \{ \]
\[ \quad g_k *= g; \]
\[ \quad \text{chi}[k] = (2 * k + 1) * (g_k - \text{g2M}) / (1 - \text{g2M}); \]
\[ \} \]

This code is used in section 240.

246. Allocate the matrix for the Legendre values this is much more efficient than calculating them as they are needed. Since the Legendre polynomial \( P_n(x) \) is generated using recurrence relations, all Legendre polynomials \( P_k(x) \), where \( 0 \leq k \leq n \) must also be calculated. Now the formula

\[ h^*(\nu_i, \nu_j) = \sum_{k=0}^{n-1} (2k + 1) \chi_k^* P_k(\nu_i) P_k(\nu_j) \]

requires all those to be found as well. There are \( 2n + 1 \) values that must be calculated for \( -\mu_n \ldots 0 \ldots \mu_n \) different arguments. A simple way is just to put all of the necessary values in a two-dimensional array and define \( p[i][j] \equiv P(\mu_j) \).

( Create Legendre Polynomial matrix 246 )
( Allocate the polynomial matrix 247 )
( Fill in all the unique values 248 )
( Fill in the symmetric values 249 )

This code is used in section 240.

247. It is not at all clear that zeroing is needed.

( Allocate the polynomial matrix 247 )
\[ p = \text{dmatrix}(0, n, -n, n); \]

This code is used in section 246.

---

\( AD\text{err}or \): void, §12.
dmatrix, <nr_util.h>.
dvector, <nr_util.h>.
fabs, <math.h>.
g: double, §239.
Get_Phi: void, §239.
h: double **, §239.
HENYEY_GREENSTEIN = 1, §5.
n: int, §239.
p: double, §222.
phase_function: int, §239.
pow, <math.h>.
Here I use the recurrence relation

\[ P_{k+1}(\mu_j) = \frac{(2k + 1)xP_k(\mu_j) - kP_{k-1}(\mu_j)}{k + 1} \]

(which should be stable) to find all the values for all the positive angles.

\[ \langle \text{Fill in all the unique values} \rangle \equiv \]

\[ \text{for} \ (j = 1; j \leq n; j++) \ { \}
  p[0][j] = 1;
  x = \text{angle}[j];
  p[1][j] = x;
  \text{for} \ (k = 1; k < n; k++) \ { \}
  p[k][j] = ((2k + 1) \times x \times p[k][j] - k \times p[k - 1][j])/(k + 1);
\]

This code is used in section 246.

I make use of the fact that

\[ P_k(-\nu_j) = (-1)^k P_k(\nu_j) \]

to fill in all the negative angles in the phase function matrix. This eliminates half the calculation. I do two at a time. This way there does not need to be a flag. Since I know that the dimension of the matrix will be even, this should not be a problem. If the matrix is not then you have problems.

\[ \langle \text{Fill in the symmetric values} \rangle \equiv \]

\[ \text{for} \ (j = 1; j \leq n; j++) \ { \}
  \text{for} \ (k = 1; k < n; k++) \ { \}
  p[k][-j] = -p[k][j];
  p[k][j] = p[k][j];
\]

This code is used in section 246.

Just a straightforward calculation of

\[ h^*(\nu_i, \nu_j) = \sum_{k=0}^{n-1} (2k + 1)\chi_k^*P_k(\nu_i)P_k(\nu_j) \]

and since \( \chi_0^* = 1 \) and \( P_0(x) = 1 \) this is

\[ h^*(\nu_i, \nu_j) = 1 + \sum_{k=1}^{n-1} (2k + 1)\chi_k^*P_k(\nu_i)P_k(\nu_j) \]

Since \( h \) has many symmetries, there are only about \( n^2/4 \) unique entries. We only need to calculate those. Oh yeah, recall that \( \chi[k] \) includes the factor \( 2k + 1 \) for speed.

\[ \langle \text{Calculate the coefficients} \rangle \equiv \]
for (i = 1; i ≤ n; i++) {
    for (j = i; j ≤ n; j++) {
        for (k = 1; k < n; k++) {
            h[i][j] += ch[i][k] * p[k][i] * p[k][j];
            h[−i][j] += ch[i][k] * p[k][−i] * p[k][j];
        }
    }
}

This code is used in section 240.

251. Several symmetries in the redistribution matrix are used to fill in some entries that begin with a negative angle

\[ h(−ν_i, ν_j) = h(ν_j, −ν_i) \]

and secondly

\[ h(−ν_i, −ν_j) = h(ν_j, ν_i) \]

Next, some entries along the diagonal are filled in using

\[ h(−ν_i, −ν_i) = h(ν_i, ν_i) \]

Finally, the lower triangle is filled in using the values from the upper half using

\[ h(ν_i, ν_j) = h(ν_j, ν_i) \]

This could probably be more elegant, but it hurts my brain to think about it. This works and should take advantage of all the symmetries present.

(Add the symmetric part of the matrix 251) \equiv

for (i = n; i ≥ 2; i--) {
    for (j = 1; j < i; j++) {
        h[−i][j] = h[−j][i];
        h[−i][−j] = h[j][i];
    }
}

for (i = 1; i ≤ n; i++) h[−i][−i] = h[i][i];

for (i = −n; i ≤ n; i++)
    for (j = i + 1; j ≤ n; j++) h[j][i] = h[i][j];

This code is used in section 240.

252. (Free p and chi 252) \equiv

free_dmatrix(p, 0, n, −n, n);
free_dvector(chi, 1, n);

This code is used in section 240.

<table>
<thead>
<tr>
<th>angle: double [], §10.</th>
<th>h: double **, §239.</th>
</tr>
</thead>
<tbody>
<tr>
<td>chi: double *, §241.</td>
<td>n: int, §239.</td>
</tr>
<tr>
<td>free_dmatrix, &lt;nr_util.h&gt;</td>
<td>p: double **, §241.</td>
</tr>
<tr>
<td>free_dvector, &lt;nr_util.h&gt;</td>
<td>x: double, §241.</td>
</tr>
</tbody>
</table>
253. **Index.** Here is a cross-reference table for the adding-doubling program. All sections in which an identifier is used are listed with that identifier, except that reserved words are indexed only when they appear in format definitions, and the appearances of identifiers in section names are not indexed. Underlined entries correspond to where the identifier was declared. Error messages and a few other things like “ASCII code dependencies” are indexed here too.