The Adding-Doubling Program

(Version 3-16-1)

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*.

```
⟨ ad_globl.c 1 ⟩ ≡
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "ad_globl.h"
#include "ad_frsnl.h"
  ⟨ Global variables for adding-doubling 11 ⟩
  ⟨ Definition for Zero_Layer 16 ⟩
  ⟨ Definition for AD_error 14 ⟩
  ⟨ Definition for URU_and_UR1 22 ⟩
  ⟨ Definition for URU_and_UR1_Cone 18 ⟩
  ⟨ Definition for URU_and_URx_Cone 20 ⟩
  ⟨ Definition for UFU_and_UF1 24 ⟩
  ⟨ Definition for wrmatrix 26 ⟩
  ⟨ Definition for wrarray 28 ⟩
```

2.

```
⟨ ad_globl.h 2 ⟩ ≡
  ⟨ Preprocessor definitions ⟩
  ⟨ Types to export from AD Globals 9 ⟩
  ⟨ External variables to export from AD Globals 12 ⟩
  ⟨ Prototype for Zero_Layer 15 ⟩;
  ⟨ Prototype for AD_error 13 ⟩;
  ⟨ Prototype for URU_and_UR1 21 ⟩;
  ⟨ Prototype for URU_and_UR1_Cone 17 ⟩;
  ⟨ Prototype for URU_and_URx_Cone 19 ⟩;
  ⟨ Prototype for UFU_and_UF1 23 ⟩;
  ⟨ Prototype for wrmatrix 25 ⟩;
  ⟨ Prototype for wrarray 27 ⟩;
```

3. Constants.

This is Version 2.0.0 of the adding-doubling code. (The inverse adding-doubling code may have a different version number.)

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 128. 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 128
#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. And finally something for whether the light is conical or oblique

```
#define CONE 1
#define OBLIQUE 0
```

9. 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 indicies of refraction *n_top_slide* and *n_bottom_slide*. The glass slides may absorb light, in which case *b_top_slide* or *b_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.

⟨ Types to export from AD Globals 9 ⟩ ≡

```
typedef struct AD_slab_type {
    double a;
    double b;
    double g;
    int phase_function;
    double n_slab;
    double n_top_slide;
    double n_bottom_slide;
    double b_top_slide;
    double b_bottom_slide;
    double cos_angle;
} slab_type;
```

See also section 10.

This code is used in section 2.

10. ⟨ Types to export from AD Globals 9 ⟩ +≡

```
typedef struct AD_method_type {
    int quad_pts;
    double a_calc, b_calc, g_calc, b_thinnest;
} method_type;
```

11. The *Martin_Hammer* variable only exists to print internal results when testing. Its only a integer and doesn't take up much space so here it is.

⟨ Global variables for adding-doubling 11 ⟩ ≡

```
#define AD_GLOBAL_SOURCE
    double angle[MAX_QUAD_PTS + 1];
    double weight[MAX_QUAD_PTS + 1];
    double twoaw[MAX_QUAD_PTS + 1];
    int Martin_Hammer = 0;
```

This code is used in section 1.

12. ⟨ External variables to export from AD Globals 12 ⟩ ≡

```
#ifndef AD_GLOBAL_SOURCE
    extern double angle[MAX_QUAD_PTS + 1];
    extern double weight[MAX_QUAD_PTS + 1];
    extern double twoaw[MAX_QUAD_PTS + 1];
    extern int Martin_Hammer;
#endif
```

This code is used in section 2.

13. Global routines. My standard error handler

⟨ Prototype for *AD_error* 13 ⟩ ≡
```
void AD_error(char error_text[])
```
This code is used in sections 2 and 14.

14. ⟨ Definition for *AD_error* 14 ⟩ ≡
```
⟨ Prototype for AD_error 13 ⟩
{
  fprintf(stderr, "Adding-Doubling␣error\n");
  fprintf(stderr, "%s\n", error_text);
  fprintf(stderr, "...now␣exiting␣to␣system...\n");
  exit(EXIT_FAILURE);
}
```
This code is used in section 1.

15. ⟨ Prototype for *Zero_Layer* 15 ⟩ ≡
```
void Zero_Layer(int n, double **r, double **t)
```
This code is used in sections 2 and 16.

16. ⟨ Definition for *Zero_Layer* 16 ⟩ ≡
```
⟨ Prototype for Zero_Layer 15 ⟩
{
  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.

17. Figure out the reflection for collimated irradiance returning within a right angle cone whose cosine of the half-apex angle is *mu*. Thus when *mu* ≡ 0 then the total light over all angles is returned. Furthermore *mu* is defined on the air side of the slab,

$$\mathrm{UR1} \equiv \int_{\mu}^{1} R(\nu', 1)\, 2\nu' d\nu'$$

Similarly for irradiance characterized by diffuse light within a cone one can calculate the amount of reflectance returing within that cone as

$$\mathrm{URU} \equiv n^2 \int_{\mu}^{1} \int_{0}^{1} R(\nu', \nu'') 2\nu' d\nu'\, 2\nu''\, d\nu''$$

where, n^2 term is to account for the n^2 law of radiance.

⟨ Prototype for *URU_and_UR1_Cone* 17 ⟩ ≡
```
void URU_and_UR1_Cone(int n, double n_slab, double mu, double **R, double *URU, double *UR1)
```
This code is used in sections 2 and 18.

18. ⟨ Definition for *URU_and_UR1_Cone* 18 ⟩ ≡
⟨ Prototype for *URU_and_UR1_Cone* 17 ⟩
```
{
  int i, j, last_j;
  double mu_slab;
  double temp = 0.0;
  if (n_slab ≡ 1) mu_slab = mu;
  else mu_slab = sqrt(n_slab * n_slab − 1 + mu * mu)/n_slab;
  last_j = 1;
  while (angle[last_j] ≤ mu_slab) last_j++;
  *URU = 0.0;
  for (i = 1; i ≤ n; i++) {
    temp = 0.0;
    for (j = last_j; j ≤ n; j++) temp += R[i][j] * twoaw[j];
    *URU += temp * twoaw[i];
  }
  *UR1 = temp;
  *URU *= n_slab * n_slab/(1 − mu * mu);
}
```
This code is used in section 1.

19. Figure out the reflection for oblique irradiance returning from a layer Note that *mu* is the cosine of the angle that the cone makes with the normal to the slab in air,

$$\mathrm{URx} = \int_{\mu}^{1} R(\nu', \mu) 2\nu' \, d\nu'$$

For diffuse irradiance over the cone, the total flux back `URU` is somewhat arbitrarily chosen as the that flux returning in the same cone. Specifically as

$$\mathrm{URU} = n^2 \int_{\mu}^{1} \int_{\mu}^{1} R(\nu', \nu'') 2\nu' \, d\nu' 2\nu'' \, d\nu''$$

where, n^2 term is to account for the n^2 law of radiance. (If you want the total flux returning within a cone for uniform diffuse illumination then use *URU_and_UR1_Cone*.)

⟨ Prototype for *URU_and_URx_Cone* 19 ⟩ ≡
```
void URU_and_URx_Cone(int n, double n_slab, double mu, double **R, double *URU, double *URx)
```
This code is used in sections 2 and 20.

20. ⟨ Definition for *URU_and_URx_Cone* 20 ⟩ ≡

```
⟨ Prototype for URU_and_URx_Cone 19 ⟩
{
  int i, j, cone_index;
  double mu_slab, urx, delta, closest_delta;
  double degrees = 180.0/M_PI;
  mu_slab = sqrt(n_slab * n_slab - 1 + mu * mu)/n_slab;
  closest_delta = 1;
  cone_index = n;
  for (i = n; i ≥ 1; i--) {
    delta = fabs(angle[i] - mu_slab);
    if (delta < closest_delta) {
      closest_delta = delta;
      cone_index = i;
    }
  }
  if (fabs(angle[cone_index] - mu_slab) > 1 · 10^-5) {
    fprintf(stderr, "Something is wrong with the quadrature\n");
    fprintf(stderr, "theta_i = %5.2f degrees or ", acos(mu) * degrees);
    fprintf(stderr, "cos(theta_i) = %8.5f\n", mu);
    fprintf(stderr, "theta_t = %5.2f degrees or ", acos(mu_slab) * degrees);
    fprintf(stderr, "cos(theta_t) = %8.5f\n", mu_slab);
    fprintf(stderr, " index  degrees cosine\n");
    for (i = n; i ≥ 1; i--) {
      fprintf(stderr, " %5d   %5.2f ", i, acos(angle[i]) * degrees);
      fprintf(stderr, " %8.5f\n", angle[i]);
    }
    fprintf(stderr, "Closest quadrature angle is i=%5d ", cone_index);
    fprintf(stderr, "or cos(theta)=%8.5f\n", angle[cone_index]);
    fprintf(stderr, "Assuming normal incidence\n");
  }
  *URU = 0.0;
  for (i = 1; i ≤ n; i++) {
    urx = 0.0;
    for (j = 1; j ≤ n; j++) urx += R[i][j] * twoaw[j];
    *URU += urx * twoaw[i];
    if (i ≡ cone_index) *URx = urx;
  }
  *URU *= n_slab * n_slab;
}
```

This code is used in section 1.

21. Just add up all the angles up to the critical angle. This is a commonly used convenience function to easily calculate UR1 and URU. We select the entire range of angles by passing $\cos(\pi/2) = 0$ to the *URU_and_UR1_Cone* routine.

⟨ Prototype for *URU_and_UR1* 21 ⟩ ≡

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

This code is used in sections 2 and 22.

22. ⟨ Definition for *URU_and_UR1* 22 ⟩ ≡
⟨ Prototype for *URU_and_UR1* 21 ⟩
```
{
  URU_and_UR1_Cone(n, n_slab, 0.0, R, URU, UR1);
}
```
This code is used in section 1.

23. ⟨ Prototype for *UFU_and_UF1* 23 ⟩ ≡
void *UFU_and_UF1* (**int** *n*, **double** *n_slab*, **double** ∗∗*Lup*, **double** ∗∗*Ldown*, **double** ∗UFU, **double** ∗UF1)

This code is used in sections 2 and 24.

24. ⟨ Definition for *UFU_and_UF1* 24 ⟩ ≡
⟨ Prototype for *UFU_and_UF1* 23 ⟩
```
{
  int i, j;
  double temp = 0.0;
  *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.

25. ⟨ Prototype for *wrmatrix* 25 ⟩ ≡
void *wrmatrix* (**int** *n*, **double** ∗∗*a*)

This code is used in sections 2 and 26.

26. ⟨ Definition for *wrmatrix* 26 ⟩ ≡

```
⟨ Prototype for wrmatrix 25 ⟩
{
  int i, j;
  double tflux, flux;

  printf("%9.5f", 0.0);
  for (i = 1; i ≤ n; i++) printf("%9.5f", angle[i]);
  printf("␣␣␣␣␣flux\n");
  tflux = 0.0;
  for (i = 1; i ≤ n; i++) {
    printf("%9.5f", angle[i]);
    for (j = 1; j ≤ n; j++)
      if ((a[i][j] > 10) ∨ (a[i][j] < −10)) printf("␣␣␣␣*****");
      else printf("%9.5f", a[i][j]);
    flux = 0.0;
    for (j = 1; j ≤ n; j++)
      if ((a[i][j] < 10) ∧ (a[i][j] > −10)) flux += a[i][j] ∗ twoaw[j];
    printf("%9.5f\n", flux);
    tflux += flux ∗ twoaw[i];
  }
  printf("%9s", "flux␣␣␣");
  for (i = 1; i ≤ n; i++) {
    flux = 0.0;
    for (j = 1; j ≤ n; j++)
      if ((a[j][i] < 10) ∧ (a[j][i] > −10)) flux += a[j][i] ∗ twoaw[j];
    printf("%9.5f", flux);
  }
  printf("%9.5f\n", tflux);
  for (i = 1; i ≤ (n + 2); i++) printf("*********");
  printf("\n\n");
}
```

This code is used in section 1.

27. ⟨ Prototype for *wrarray* 27 ⟩ ≡

```
void wrarray(int n, double *a)
```

This code is used in sections 2 and 28.

28. ⟨ Definition for *wrarray* 28 ⟩ ≡

```
⟨ Prototype for wrarray 27 ⟩
{
  int i;
  double sum;
  for (i = 1; i ≤ n; i++) printf("%9.5f", angle[i]);
  printf("%9s\n", "␣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\n", "␣(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\n", "*2aw");
  for (i = 1; i ≤ (n + 2); i++) printf("*********");
  printf("\n\n");
}
```

This code is used in section 1.

29. Just print out an array without mucking

⟨ Prototype for *swrarray* 29 ⟩ ≡

```
  void swrarray(int n, double *a)
```

This code is used in section 30.

30. ⟨ Definition for *swrarray* 30 ⟩ ≡

```
⟨ Prototype for swrarray 29 ⟩
{
  int i;
  double sum;
  for (i = 1; i ≤ n; i++) printf("%9.5f", angle[i]);
  printf("%9s\n", "*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");
}
```

31. AD Prime. This has the rather stupid name prime because I was at a loss for another. Currently this is 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.

⟨ ad_prime.c 31 ⟩ ≡
#include <math.h>
#include <float.h>
#include <stdio.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"
#include "ad_cone.h"
⟨ Definition for *RT_Matrices* 35 ⟩
⟨ Definition for RT 37 ⟩
⟨ Definition for *ez_RT* 52 ⟩
⟨ Definition for *RTabs* 56 ⟩
⟨ Definition for *Flux_Fluence* 66 ⟩
⟨ Definition for *ez_RT_unscattered* 54 ⟩

32. ⟨ ad_prime.h 32 ⟩ ≡
⟨ Preprocessor definitions ⟩
⟨ Prototype for *RT_Matrices* 34 ⟩;
⟨ Prototype for RT 36 ⟩;
⟨ Prototype for *ez_RT* 51 ⟩;
⟨ Prototype for *RTabs* 55 ⟩;
⟨ Prototype for *Flux_Fluence* 65 ⟩;
⟨ Prototype for *ez_RT_unscattered* 53 ⟩;

33. ⟨ lib_ad.h 33 ⟩ ≡
⟨ Prototype for *ez_RT* 51 ⟩;
⟨ Prototype for *ez_RT_unscattered* 53 ⟩;

34. 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 *method→quad_pts* and the optical properties (the albedo, anisotropy, optical thickness, and choice of phase function) in *slab*. Call this routine and get back matrices filled with cool numbers.

⟨ Prototype for *RT_Matrices* 34 ⟩ ≡

```
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 32 and 35.

35. ⟨ Definition for *RT_Matrices* 35 ⟩ ≡

```
⟨ Prototype for RT_Matrices 34 ⟩
{
  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) ≡ 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 31.

36. Total reflection and transmission.
RT is the top level routine for accessing the adding-doubling algorithm. By passing the optical paramters 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 36 ⟩ ≡
```
void RT(int n, struct AD_slab_type *slab, double *UR1, double *UT1, double *URU, double *UTU)
```
This code is used in sections 32 and 37.

37. ⟨ Definition for RT 37 ⟩ ≡
```
⟨ Prototype for RT 36 ⟩
{
  ⟨ Declare variables for RT 38 ⟩
  if (slab→cos_angle ≠ 1.0) {
    RT_Cone(n, slab, OBLIQUE, UR1, UT1, URU, UTU);
    return;
  }
  ⟨ Validate input parameters 39 ⟩
  ⟨ Allocate and calculate R and T for homogeneous slab 40 ⟩
  if (slab→b ≡ 0) {
    Sp_RT(n, *slab, UR1, UT1, URU, UTU);
  }
  else if (slab→n_slab ≡ 1 ∧ slab→n_top_slide ≡ 1 ∧ slab→n_bottom_slide ≡ 1 ∧ slab→b_top_slide ≡
        0 ∧ slab→b_bottom_slide ≡ 0) {
    ⟨ Do slab with no boundaries 41 ⟩
  }
  else if (slab→n_top_slide ≡ slab→n_bottom_slide ∧ slab→b_top_slide ≡ 0 ∧ slab→b_bottom_slide ≡ 0) {
    ⟨ Allocate and generate top boundary 42 ⟩
    ⟨ Do slab with matched top and bottom boundaries 43 ⟩
    ⟨ Free top boundary 44 ⟩
  }
  else {
    ⟨ Allocate and generate top boundary 42 ⟩
    ⟨ Allocate and generate bottom boundary 45 ⟩
    ⟨ Allocate misc matrices 46 ⟩
    ⟨ Do slab with mismatched boundaries 47 ⟩
    ⟨ Free misc matrices 48 ⟩
    ⟨ Free bottom boundary 49 ⟩
    ⟨ Free top boundary 44 ⟩
  }
  ⟨ Free R and T 50 ⟩
}
```
This code is used in section 31.

38. ⟨ Declare variables for RT 38 ⟩ ≡

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

$*\mathtt{UR1} = -1$;
$*\mathtt{URU} = -1$;
$*\mathtt{UT1} = -1$;
$*\mathtt{UTU} = -1$;

This code is used in section 37.

39.

⟨ Validate input parameters 39 ⟩ ≡

if $(slab\to n\_slab < 0)$ **return**;
if $(slab\to n\_top\_slide < 0)$ **return**;
if $(slab\to n\_bottom\_slide < 0)$ **return**;
if $(slab\to a < 0 \lor slab\to a > 1)$ **return**;
if $(slab\to g < -1 \lor slab\to g > 1)$ **return**;
if $(slab\to b < 0)$ **return**;

This code is used in section 37.

40. Find the R and T for a homogeneous slab without boundaries

⟨ Allocate and calculate R and T for homogeneous slab 40 ⟩ ≡

$R = dmatrix(1, n, 1, n)$;
$T = dmatrix(1, n, 1, n)$;
$RT\_Matrices(n, slab, \&method, R, T)$;

This code is used in sections 37 and 56.

41. ⟨ Do slab with no boundaries 41 ⟩ ≡

$URU\_and\_UR1(n, slab\to n\_slab, R, \mathtt{URU}, \mathtt{UR1})$;
$URU\_and\_UR1(n, slab\to n\_slab, T, \mathtt{UTU}, \mathtt{UT1})$;

This code is used in section 37.

42. ⟨ Allocate and generate top boundary 42 ⟩ ≡

$\mathtt{R01} = dvector(1, n)$;
$\mathtt{R10} = dvector(1, n)$;
$\mathtt{T01} = dvector(1, n)$;
$\mathtt{T10} = dvector(1, n)$;
$Init\_Boundary(*slab, method.quad\_pts, \mathtt{R01}, \mathtt{R10}, \mathtt{T01}, \mathtt{T10}, \mathtt{TOP\_BOUNDARY})$;

This code is used in sections 37 and 60.

43. ⟨ Do slab with matched top and bottom boundaries 43 ⟩ ≡

```
atemp = dmatrix(1, n, 1, n);
btemp = dmatrix(1, n, 1, n);
R2 = dmatrix(1, n, 1, n);
T2 = dmatrix(1, n, 1, n);
Add_Slides(n, R01, R10, T01, T10, R, T, R2, T2, atemp, btemp);
URU_and_UR1(n, slab→n_slab, R2, URU, UR1);
URU_and_UR1(n, slab→n_slab, T2, UTU, UT1);
free_dmatrix(atemp, 1, n, 1, n);
free_dmatrix(btemp, 1, n, 1, n);
free_dmatrix(R2, 1, n, 1, n);
free_dmatrix(T2, 1, n, 1, n);
```

This code is used in section 37.

44. ⟨ Free top boundary 44 ⟩ ≡

```
free_dvector(R01, 1, n);
free_dvector(R10, 1, n);
free_dvector(T01, 1, n);
free_dvector(T10, 1, n);
```

This code is used in sections 37 and 56.

45. ⟨ Allocate and generate bottom boundary 45 ⟩ ≡

```
R23 = dvector(1, n);
R32 = dvector(1, n);
T23 = dvector(1, n);
T32 = dvector(1, n);
Init_Boundary(*slab, method.quad_pts, R23, R32, T23, T32, BOTTOM_BOUNDARY);
```

This code is used in sections 37 and 61.

46. ⟨ Allocate misc matrices 46 ⟩ ≡

```
R02 = dmatrix(1, n, 1, n);
R20 = dmatrix(1, n, 1, n);
T02 = dmatrix(1, n, 1, n);
T20 = dmatrix(1, n, 1, n);
R03 = dmatrix(1, n, 1, n);
R30 = dmatrix(1, n, 1, n);
T03 = dmatrix(1, n, 1, n);
T30 = dmatrix(1, n, 1, n);
atemp = dmatrix(1, n, 1, n);
btemp = dmatrix(1, n, 1, n);
```

This code is used in sections 37 and 56.

47. ⟨ Do slab with mismatched boundaries 47 ⟩ ≡

```
Add_Top(n, R01, R10, T01, T10, R, R, T, T, R02, R20, T02, T20, atemp, btemp);
Add_Bottom(n, R02, R20, T02, T20, R23, R32, T23, T32, R03, R30, T03, T30, atemp, btemp);
URU_and_UR1(n, slab→n_slab, R03, URU, UR1);
Transpose_Matrix(n, T03);
URU_and_UR1(n, slab→n_slab, T03, UTU, UT1);
```

This code is used in section 37.

48. ⟨ Free misc matrices 48 ⟩ ≡
$free\_dmatrix(\mathtt{R02}, 1, n, 1, n)$;
$free\_dmatrix(\mathtt{R20}, 1, n, 1, n)$;
$free\_dmatrix(\mathtt{T02}, 1, n, 1, n)$;
$free\_dmatrix(\mathtt{T20}, 1, n, 1, n)$;
$free\_dmatrix(\mathtt{R03}, 1, n, 1, n)$;
$free\_dmatrix(\mathtt{R30}, 1, n, 1, n)$;
$free\_dmatrix(\mathtt{T03}, 1, n, 1, n)$;
$free\_dmatrix(\mathtt{T30}, 1, n, 1, n)$;
$free\_dmatrix(atemp, 1, n, 1, n)$;
$free\_dmatrix(btemp, 1, n, 1, n)$;

This code is used in sections 37 and 56.

49. ⟨ Free bottom boundary 49 ⟩ ≡
$free\_dvector(\mathtt{R23}, 1, n)$;
$free\_dvector(\mathtt{R32}, 1, n)$;
$free\_dvector(\mathtt{T23}, 1, n)$;
$free\_dvector(\mathtt{T32}, 1, n)$;

This code is used in sections 37 and 56.

50. ⟨ Free R and T 50 ⟩ ≡
$free\_dmatrix(R, 1, n, 1, n)$;
$free\_dmatrix(T, 1, n, 1, n)$;

This code is used in sections 37 and 56.

51. Simple interfaces for Perl, Python, or Mathematica.

ez_RT is a top level routine for accessing the adding-doubling algorithm. This routine was originally created so that I could make a Perl .xs module. Since I did not know how to mess around with passing structures, I changed the interface to avoid using structures.

⟨ Prototype for *ez_RT* 51 ⟩ ≡

void *ez_RT*(**int** *n*, **double** *nslab*, **double** *ntopslide*, **double** *nbottomslide*, **double** *a*, **double** *b*, **double** *g*, **double** *UR1, **double** *UT1, **double** *URU, **double** *UTU)

This code is used in sections 32, 33, and 52.

52. ⟨ Definition for *ez_RT* 52 ⟩ ≡

```
⟨ Prototype for ez_RT 51 ⟩
{
  struct AD_slab_type slab;
  slab.n_slab = nslab;
  slab.n_top_slide = ntopslide;
  slab.n_bottom_slide = nbottomslide;
  slab.b_top_slide = 0;
  slab.b_bottom_slide = 0;
  slab.a = a;
  slab.b = b;
  slab.g = g;
  slab.phase_function = HENYEY_GREENSTEIN;
  slab.cos_angle = 1.0;
  RT(n, &slab, UR1, UT1, URU, UTU);
}
```

This code is used in section 31.

53. Unscattered reflection and transmission.

ez_RT_unscattered is a top level routine for accessing the adding-doubling algorithm. This routine was created so that I could make a Perl module. Since I did not know how to mess around with passing structures, I changed the interface to avoid using structures.

⟨ Prototype for *ez_RT_unscattered* 53 ⟩ ≡

```
void ez_RT_unscattered(int n, double nslab, double ntopslide, double nbottomslide, double a, double
    b, double g, double *UR1, double *UT1, double *URU, double *UTU)
```

This code is used in sections 32, 33, and 54.

54. ⟨ Definition for *ez_RT_unscattered* 54 ⟩ ≡

```
⟨ Prototype for ez_RT_unscattered 53 ⟩
{
  struct AD_slab_type slab;
  slab.n_slab = nslab;
  slab.n_top_slide = ntopslide;
  slab.n_bottom_slide = nbottomslide;
  slab.b_top_slide = 0;
  slab.b_bottom_slide = 0;
  slab.a = a;
  slab.b = b;
  slab.g = g;
  slab.phase_function = HENYEY_GREENSTEIN;
  slab.cos_angle = 1.0;
  Sp_RT(n, slab, UR1, UT1, URU, UTU);
}
```

This code is used in section 31.

55. 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!*

⟨ Prototype for *RTabs* 55 ⟩ ≡

void *RTabs*(**int** *n*, **struct AD_slab_type** ∗*slab*, **double** ∗`UR1`, **double** ∗`UT1`, **double** ∗`URU`, **double** ∗`UTU`)

This code is used in sections 32 and 56.

56. ⟨ Definition for *RTabs* 56 ⟩ ≡
⟨ Prototype for *RTabs* 55 ⟩
{
⟨ Declare variables for *RTabs* 57 ⟩
double ∗∗*Rtop*, ∗∗*Ttop*, ∗∗*Rbottom*, ∗∗*Tbottom*;
struct AD_slab_type *slab1*;
double *btop*, *bbottom*;
⟨ Allocate and calculate R and T for homogeneous slab 40 ⟩
⟨ Allocate and calculate top absorbing slide 58 ⟩
⟨ Allocate and calculate bottom absorbing slide 59 ⟩
⟨ Allocate misc matrices 46 ⟩
⟨ Allocate and calculate top non-absorbing boundary 60 ⟩
⟨ Allocate and calculate bottom non-absorbing boundary 61 ⟩
⟨ Add all the stuff together 62 ⟩
⟨ Free misc matrices 48 ⟩
⟨ Free bottom boundary 49 ⟩
⟨ Free top boundary 44 ⟩
⟨ Free R and T 50 ⟩
⟨ Free matrices for the top and bottom absorbing slides 63 ⟩
}

This code is used in section 31.

57. ⟨ Declare variables for *RTabs* 57 ⟩ ≡
double ∗∗*R*, ∗∗*T*;
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 56.

58. ⟨ Allocate and calculate top absorbing slide 58 ⟩ ≡

```
slab1.b = slab→b_top_slide;
slab1.cos_angle = slab→cos_angle;
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 56.

59. ⟨ Allocate and calculate bottom absorbing slide 59 ⟩ ≡

```
slab1.b = slab→b_bottom_slide;
slab1.cos_angle = slab→cos_angle;
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 56.

60.

⟨ Allocate and calculate top non-absorbing boundary 60 ⟩ ≡

```
btop = slab→b_top_slide;
slab→b_top_slide = 0;
⟨ Allocate and generate top boundary 42 ⟩
slab→b_top_slide = btop;
```

This code is used in section 56.

61.

⟨ Allocate and calculate bottom non-absorbing boundary 61 ⟩ ≡

```
bbottom = slab→b_bottom_slide;
slab→b_bottom_slide = 0;
⟨ Allocate and generate bottom boundary 45 ⟩
slab→b_bottom_slide = bbottom;
```

This code is used in section 56.

62.

⟨ Add all the stuff together 62 ⟩ ≡

```
Add(n, Rtop, Rtop, Ttop, Ttop, R, R, T, T, R02, R20, T02, T20);
Add(n, R02, R20, T02, T20, Rbottom, Rbottom, Tbottom, Tbottom, R03, R30, T03, T30);
Add_Top(n, R01, R10, T01, T10, R03, R30, T03, T30, R02, R20, T02, T20, atemp, btemp);
Add_Bottom(n, R02, R20, T02, T20, R23, R32, T23, T32, R03, R30, T03, T30, atemp, btemp);
URU_and_UR1(n, slab→n_slab, R03, URU, UR1);
Transpose_Matrix(n, T03);
URU_and_UR1(n, slab→n_slab, T03, UTU, UT1);
```

This code is used in section 56.

63.

⟨ Free matrices for the top and bottom absorbing slides 63 ⟩ ≡

$free\_dmatrix(Rtop, 1, n, 1, n);$
$free\_dmatrix(Ttop, 1, n, 1, n);$
$free\_dmatrix(Rbottom, 1, n, 1, n);$
$free\_dmatrix(Tbottom, 1, n, 1, n);$

This code is used in section 56.

64. Flux and Fluence.
Calculates the flux and fluence at various depths between the optical depths *zmin* and *zmax* for a slab. The number of values is $intervals + 1$ times...i.e. it calculates at *zmin*, $zmin + (zmax - zmin)/intervals$, ... , *zmax*

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 MAX_FLUENCE_INTERVALS 200
```

65. ⟨ Prototype for *Flux_Fluence* 65 ⟩ ≡

```
void Flux_Fluence(int n, struct AD_slab_type *slab, double zmin, double zmax, int
    intervals, double *UF1_array, double *UFU_array, double *flux_up, double *flux_down)
```

This code is used in sections 32 and 66.

66. ⟨ Definition for *Flux_Fluence* 66 ⟩ ≡

```
⟨ Prototype for Flux_Fluence 65 ⟩
{
  ⟨ Declare variables for Flux_Fluence 67 ⟩
  if (intervals > MAX_FLUENCE_INTERVALS)
    AD_error("too␣many␣intervals␣requested.␣␣increase␣the␣const␣max_fluence_intervals\n");
  ⟨ Find the 02 matrix for the slab above all layers 68 ⟩
  ⟨ Find the 46 matrix for the slab below all layers 69 ⟩
  ⟨ Allocate intermediate matrices 70 ⟩
  for (i = 0; i ≤ intervals; i++) {
    ⟨ Find radiance at each depth 71 ⟩
    ⟨ Calculate Fluence and Flux 72 ⟩
  }
  ⟨ Free all those intermediate matrices 73 ⟩
}
```

This code is used in section 31.

67. ⟨ Declare variables for *Flux_Fluence* 67 ⟩ ≡

```
double *R01, *R10, *T01, *T10;
double *R56, *R65, *T56, *T65;
double **R12, **T12;
double **R23, **T23;
double **R34, **T34;
double **R45, **T45;
double **R02, **R20, **T02, **T20;
double **R46, **R64, **T46, **T64;
double **R03, **R30, **T03, **T30;
double **R36, **R63, **T36, **T63;
double **Lup, **Ldown;
double **a, **b;
double flx_down, flx_up, UFU, UF1;
double slab_thickness;
struct AD_method_type method;
int i, j;
```

This code is used in section 66.

68.

⟨ Find the 02 matrix for the slab above all layers 68 ⟩ ≡

```
slab_thickness = slab→b;    /* save it for later */
slab→b = zmin;
R12 = dmatrix(1, n, 1, n);
T12 = dmatrix(1, n, 1, n);
RT_Matrices(n, slab, &method, R12, T12);
R01 = dvector(1, n);
R10 = dvector(1, n);
T01 = dvector(1, n);
T10 = dvector(1, n);
Init_Boundary(*slab, method.quad_pts, R01, R10, T01, T10, TOP_BOUNDARY);
R20 = dmatrix(1, n, 1, n);
T20 = dmatrix(1, n, 1, n);
R02 = dmatrix(1, n, 1, n);
T02 = dmatrix(1, n, 1, n);
a = dmatrix(1, n, 1, n);
b = dmatrix(1, n, 1, n);
Add_Top(n, R01, R10, T01, T10, R12, R12, T12, T12, R02, R20, T02, T20, a, b);
free_dmatrix(R12, 1, n, 1, n);
free_dmatrix(T12, 1, n, 1, n);
free_dvector(R01, 1, n);
free_dvector(R10, 1, n);
free_dvector(T01, 1, n);
free_dvector(T10, 1, n);
```

This code is used in section 66.

69. ⟨ Find the 46 matrix for the slab below all layers 69 ⟩ ≡

```
slab→b = slab_thickness − zmax;
R45 = dmatrix(1, n, 1, n);
T45 = dmatrix(1, n, 1, n);
RT_Matrices(n, slab, &method, R45, T45);
R56 = dvector(1, n);
R65 = dvector(1, n);
T56 = dvector(1, n);
T65 = dvector(1, n);
Init_Boundary(*slab, method.quad_pts, R56, R65, T56, T65, BOTTOM_BOUNDARY);
R46 = dmatrix(1, n, 1, n);
T46 = dmatrix(1, n, 1, n);
R64 = dmatrix(1, n, 1, n);
T64 = dmatrix(1, n, 1, n);
Add_Bottom(n, R45, R45, T45, T45, R56, R65, T56, T65, R46, R64, T46, T64, a, b);
free_dmatrix(R45, 1, n, 1, n);
free_dmatrix(T45, 1, n, 1, n);
free_dvector(R56, 1, n);
free_dvector(R65, 1, n);
free_dvector(T56, 1, n);
free_dvector(T65, 1, n);
free_dmatrix(a, 1, n, 1, n);
free_dmatrix(b, 1, n, 1, n);
```

This code is used in section 66.

70. ⟨ Allocate intermediate matrices 70 ⟩ ≡

```
R23 = dmatrix(1, n, 1, n);
T23 = dmatrix(1, n, 1, n);
R03 = dmatrix(1, n, 1, n);
T03 = dmatrix(1, n, 1, n);
R30 = dmatrix(1, n, 1, n);
T30 = dmatrix(1, n, 1, n);
R34 = dmatrix(1, n, 1, n);
T34 = dmatrix(1, n, 1, n);
R63 = dmatrix(1, n, 1, n);
T63 = dmatrix(1, n, 1, n);
R36 = dmatrix(1, n, 1, n);
T36 = dmatrix(1, n, 1, n);
Lup = dmatrix(1, n, 1, n);
Ldown = dmatrix(1, n, 1, n);
```

This code is used in section 66.

71. ⟨ Find radiance at each depth 71 ⟩ ≡

```
slab→b = (zmax − zmin)/intervals ∗ i;
RT_Matrices(n, slab, &method, R23, T23);
Add(n, R02, R20, T02, T20, R23, R23, T23, T23, R03, R30, T03, T30);
slab→b = (zmax − zmin) − slab→b;
RT_Matrices(n, slab, &method, R34, T34);
Add(n, R34, R34, T34, T34, R46, R64, T46, T64, R36, R63, T36, T63);
Between(n, R03, R30, T03, T30, R36, R63, T36, T63, Lup, Ldown);
```

This code is used in section 66.

72. ⟨ Calculate Fluence and Flux 72 ⟩ ≡

```
UFU_and_UF1(n, slab→n_slab, Lup, Ldown, &UFU, &UF1);
UF1_array[i] = UF1;
UFU_array[i] = UFU;
flx_down = 0.0;
flx_up = 0.0;
for (j = 1; j ≤ n; j++) {
   flx_down += twoaw[j] ∗ Ldown[j][n];
   flx_up += twoaw[j] ∗ Lup[j][n];
}
flux_down[i] = flx_down ∗ slab→n_slab ∗ slab→n_slab;
flux_up[i] = flx_up ∗ slab→n_slab ∗ slab→n_slab;
```

This code is used in section 66.

73. ⟨ Free all those intermediate matrices 73 ⟩ ≡

```
free_dmatrix(R02, 1, n, 1, n);
free_dmatrix(T02, 1, n, 1, n);
free_dmatrix(R20, 1, n, 1, n);
free_dmatrix(T20, 1, n, 1, n);
free_dmatrix(R23, 1, n, 1, n);
free_dmatrix(T23, 1, n, 1, n);
free_dmatrix(R03, 1, n, 1, n);
free_dmatrix(T03, 1, n, 1, n);
free_dmatrix(R30, 1, n, 1, n);
free_dmatrix(T30, 1, n, 1, n);
free_dmatrix(R34, 1, n, 1, n);
free_dmatrix(T34, 1, n, 1, n);
free_dmatrix(R63, 1, n, 1, n);
free_dmatrix(T63, 1, n, 1, n);
free_dmatrix(R36, 1, n, 1, n);
free_dmatrix(T36, 1, n, 1, n);
free_dmatrix(R64, 1, n, 1, n);
free_dmatrix(T64, 1, n, 1, n);
free_dmatrix(R46, 1, n, 1, n);
free_dmatrix(T46, 1, n, 1, n);
free_dmatrix(Lup, 1, n, 1, n);
free_dmatrix(Ldown, 1, n, 1, n);
```

This code is used in section 66.

74. AD Layers. This file provides routines to obtain reflection and transmission values for normal illumination of several multiple scattering and absorbing layers.

⟨ ad_layers.c 74 ⟩ ≡
```
#include <math.h>
#include <float.h>
#include "nr_util.h"
#include "ad_globl.h"
#include "ad_bound.h"
#include "ad_doubl.h"
#include "ad_prime.h"
#include "ad_matrx.h"
#include "ad_prime.h"
```
⟨ Definition for *RT_Layers_All* 77 ⟩
⟨ Definition for *RT_Layers* 88 ⟩

75. ⟨ ad_layers.h 75 ⟩ ≡
⟨ Preprocessor definitions ⟩
⟨ Prototype for *RT_Layers* 87 ⟩;
⟨ Prototype for *RT_Layers_All* 76 ⟩;

76. RT Layers. Sometimes you just need to know the total reflection and transmission from a target consisting of multiple layers. This is the routine for you. It adds a bunch of scattering and absorbing layers together which have the same index of refraction together. The top and bottom are possibly bounded by glass slides. This is not particularly fast, but it should get the job done.

nlayers specifies the number of different layers (not including possible glass slides above and below the composite sample. The optical properties are passed in three zero-based arrays of doubles. For example $a[1]$ is the albedo of the second layer.

⟨ Prototype for *RT_Layers_All* 76 ⟩ ≡

```
void RT_Layers_All(int n, double nslab, double ntopslide, double nbottomslide, int nlayers, double
    a[], double b[], double g[], double *dUR1, double *dUT1, double *dURU, double
    *dUTU, double *uUR1, double *uUT1, double *uURU, double *uUTU)
```

This code is used in sections 75 and 77.

77. ⟨ Definition for *RT_Layers_All* 77 ⟩ ≡

```
⟨ Prototype for RT_Layers_All 76 ⟩
{
  ⟨ Declare variables for RT_Layers 79 ⟩
  ⟨ Validate layer properties 78 ⟩
  ⟨ Allocate slab memory 80 ⟩
  ⟨ Initialize slab structure 82 ⟩
  ⟨ Initialize composite layer 83 ⟩
  ⟨ Allocate and generate top and bottom boundaries 81 ⟩
  ⟨ Add all composite layers together 84 ⟩
  ⟨ Add top and bottom boundaries 85 ⟩
  ⟨ Free memory for RT_Layers 86 ⟩
}
```

This code is used in section 74.

78. Simple sanity checks to ensure values are reasonable.

⟨ Validate layer properties 78 ⟩ ≡

```
if (nlayers < 1) return;
if (nslab < 0) return;
if (ntopslide < 0) return;
if (nbottomslide < 0) return;
for (i = 0; i < nlayers; i++) {
  if (a[i] < 0 ∨ a[i] > 1) return;
  if (b[i] < 0) return;
  if (g[i] < −1 ∨ g[i] > 1) return;
}
```

This code is used in section 77.

79. ⟨ Declare variables for *RT_Layers* 79 ⟩ ≡

```
struct AD_slab_type slab;
struct AD_method_type method;
double *R01, *R10, *T01, *T10;
double *R34, *R43, *T34, *T43;
double **R12, **R21, **T12, **T21;
double **R23, **R32, **T23, **T32;
double **R13, **R31, **T13, **T31;
double **atemp, **btemp;
int i;
*dUR1 = -1;
*dUT1 = -1;
*dURU = -1;
*dUTU = -1;
*uUR1 = -1;
*uUT1 = -1;
*uURU = -1;
*uUTU = -1;
```

This code is used in section 77.

80. ⟨ Allocate slab memory 80 ⟩ ≡

```
R12 = dmatrix(1, n, 1, n);
R21 = dmatrix(1, n, 1, n);
T12 = dmatrix(1, n, 1, n);
T21 = dmatrix(1, n, 1, n);
R23 = dmatrix(1, n, 1, n);
R32 = dmatrix(1, n, 1, n);
T23 = dmatrix(1, n, 1, n);
T32 = dmatrix(1, n, 1, n);
R13 = dmatrix(1, n, 1, n);
R31 = dmatrix(1, n, 1, n);
T13 = dmatrix(1, n, 1, n);
T31 = dmatrix(1, n, 1, n);
atemp = dmatrix(1, n, 1, n);
btemp = dmatrix(1, n, 1, n);
```

This code is used in section 77.

81. Create the matrices needed for the top and bottom. This needs to be done after a call to *RT_Matrices*( ) so that quadrature angles are chosen.

⟨ Allocate and generate top and bottom boundaries 81 ⟩ ≡

```
R01 = dvector(1, n);
R10 = dvector(1, n);
T01 = dvector(1, n);
T10 = dvector(1, n);
Init_Boundary(slab, n, R01, R10, T01, T10, TOP_BOUNDARY);
R34 = dvector(1, n);
R43 = dvector(1, n);
T34 = dvector(1, n);
T43 = dvector(1, n);
Init_Boundary(slab, n, R34, R43, T34, T43, BOTTOM_BOUNDARY);
```

This code is used in section 77.

82. We set this to be a clear layer so that the composite layer will be created properly. The index of refraction of the slab is important so that the quadrature angles will be chosen correctly.

⟨ Initialize slab structure 82 ⟩ ≡

```
slab.n_slab = nslab;
slab.n_top_slide = ntopslide;
slab.n_bottom_slide = nbottomslide;
slab.b_top_slide = 0;
slab.b_bottom_slide = 0;
slab.a = 0.0;
slab.b = 0.0;
slab.g = 0.0;
slab.phase_function = HENYEY_GREENSTEIN;
slab.cos_angle = 1.0;
```

This code is used in section 77.

83. The composite layer initially has 0% reflection and 100% transmission. We fob the details on how this layer is created to the *RT_Matrices* which goes to the trouble to initialize *method* and call *Zero_Layer* for us. Finally, since this optical problem is not reversible (illumination from below gives a different answer), we need to initialize the upward matrices as well. This simplifies the code when adding successive layers.

⟨ Initialize composite layer 83 ⟩ ≡

```
RT_Matrices(n, &slab, &method, R23, T23);
Copy_Matrix(n, R23, R32);
Copy_Matrix(n, T23, T32);
```

This code is used in section 77.

84. Now add the layers together. Since the composite layer has been initialized to be a clear layer, we can just add layers to it. We start from the bottom. Find the transport matrices for this layer. Add this layer to the top of the composite layer. This is repeated for each of the layers.

⟨ Add all composite layers together 84 ⟩ ≡

```
while (nlayers ≥ 1) {
  nlayers--;
  slab.a = a[nlayers];
  slab.b = b[nlayers];
  slab.g = g[nlayers];
  RT_Matrices(n, &slab, &method, R12, T12);
  Add(n, R12, R12, T12, T12, R23, R32, T23, T32, R13, R31, T13, T31);
  Copy_Matrix(n, R13, R23);
  Copy_Matrix(n, R31, R32);
  Copy_Matrix(n, T13, T23);
  Copy_Matrix(n, T31, T32);
}
```

This code is used in section 77.

85. The only confusing part about this piece of code is that the layer numbering gets all messed up. The composite layer is in the 23 matrices. This gets added to the top 01 boundary and should be labeled the 03 matrix. Instead I use the already allocated 13 matrices. This layer is then added to the bottom 34 matrices and should result in 04 matrices, but once again I use the 23 matrices. Finally, the total reflectances and transmittances are calculated, so that all the remains is to free the allocated memory! Not so hard after all.

⟨ Add top and bottom boundaries 85 ⟩ ≡

```
Add_Top(n, R01, R10, T01, T10, R23, R32, T23, T32, R13, R31, T13, T31, atemp, btemp);
Add_Bottom(n, R13, R31, T13, T31, R34, R43, T34, T43, R23, R32, T23, T32, atemp, btemp);
URU_and_UR1(n, slab.n_slab, R23, dURU, dUR1);
URU_and_UR1(n, slab.n_slab, R32, uURU, uUR1);
Transpose_Matrix(n, T23);
Transpose_Matrix(n, T32);
URU_and_UR1(n, slab.n_slab, T23, dUTU, dUT1);
URU_and_UR1(n, slab.n_slab, T32, uUTU, uUT1);
```

This code is used in section 77.

86. ⟨ Free memory for *RT_Layers* 86 ⟩ ≡

```
free_dvector(R01, 1, n);
free_dvector(R10, 1, n);
free_dvector(T01, 1, n);
free_dvector(T10, 1, n);
free_dmatrix(R12, 1, n, 1, n);
free_dmatrix(R21, 1, n, 1, n);
free_dmatrix(T12, 1, n, 1, n);
free_dmatrix(T21, 1, n, 1, n);
free_dmatrix(R23, 1, n, 1, n);
free_dmatrix(R32, 1, n, 1, n);
free_dmatrix(T23, 1, n, 1, n);
free_dmatrix(T32, 1, n, 1, n);
free_dmatrix(R13, 1, n, 1, n);
free_dmatrix(R31, 1, n, 1, n);
free_dmatrix(T13, 1, n, 1, n);
free_dmatrix(T31, 1, n, 1, n);
free_dmatrix(atemp, 1, n, 1, n);
free_dmatrix(btemp, 1, n, 1, n);
free_dvector(R34, 1, n);
free_dvector(R43, 1, n);
free_dvector(T34, 1, n);
free_dvector(T43, 1, n);
```

This code is used in section 77.

87. This just returns the reflection and transmission for light travelling downwards. This is most often what is desired.

⟨ Prototype for *RT_Layers* 87 ⟩ ≡

```
void RT_Layers(int n, double nslab, double ntopslide, double nbottomslide, int nlayers, double
    a[], double b[], double g[], double *UR1, double *UT1, double *URU, double *UTU)
```

This code is used in sections 75 and 88.

88. ⟨ Definition for *RT_Layers* 88 ⟩ ≡

```
⟨ Prototype for RT_Layers 87 ⟩
{
  double uUR1, uUT1, uURU, uUTU;
  RT_Layers_All(n, nslab, ntopslide, nbottomslide, nlayers, a, b, g, 3 9UR1, UT1, URU, UTU, &uUR1, &uUT1,
      &uURU, &uUTU);
}
```

This code is used in section 74.

89. AD Cone. This file provides routines to obtain reflection and transmission values returning within a cone assuming normal illumination.

```
⟨ ad_cone.c 89 ⟩ ≡
#include <math.h>
#include <float.h>
#include <stdio.h>
#include "nr_util.h"
#include "ad_globl.h"
#include "ad_matrx.h"
#include "ad_bound.h"
#include "ad_doubl.h"
#include "ad_start.h"
  ⟨ Definition for RT_Cone 93 ⟩
  ⟨ Definition for ez_RT_Cone 103 ⟩
  ⟨ Definition for ez_RT_Oblique 105 ⟩
```

90.
```
⟨ ad_cone.h 90 ⟩ ≡
  ⟨ Preprocessor definitions ⟩
  ⟨ Prototype for RT_Cone 92 ⟩;
  ⟨ Prototype for ez_RT_Cone 102 ⟩;
  ⟨ Prototype for ez_RT_Oblique 104 ⟩;
```

91.
```
⟨ ad_cone_ez.h 91 ⟩ ≡
  ⟨ Preprocessor definitions ⟩
  ⟨ Prototype for ez_RT_Cone 102 ⟩;
  ⟨ Prototype for ez_RT_Oblique 104 ⟩;
```

92. RT Cone. Sometimes you just need to know the total reflection and transmission from a target within a specified cone of angles. For example, you might want to test a Monte Carlo implementation of fiber illumination. The way that this works is to divide the integration over angles into two or three pieces. A separate quadrature is done over each integration range. For example if $\nu_{\rm cone}$ is the cosine of the cone angle and there are no index of refraction changes that need to accounted for, then

$$\int_0^1 A(\nu,\nu')B(\nu',\nu'')\,d\nu' = \int_0^{\nu_{\rm cone}} A(\nu,\nu')B(\nu',\nu'')\,d\nu' + \int_{\nu_{\rm cone}}^1 A(\nu,\nu')B(\nu',\nu'')\,d\nu'.$$

otherwise one needs to include the critical angle as a special point in the integration and the integration becomes

$$\int_0^1 A(\nu,\nu')B(\nu',\nu'')\,d\nu' = \int_0^{\nu_{\rm crit}} A(\nu,\nu')B(\nu',\nu'')\,d\nu' + \int_{\nu_{\rm crit}}^{\nu_{\rm cone}} A(\nu,\nu')B(\nu',\nu'')\,d\nu' + \int_{\nu_{\rm cone}}^1 A(\nu,\nu')B(\nu',\nu'')\,d\nu'.$$

Radau quadrature is chosen for the integration range from $\nu_{\rm cone}$ to 1. The other two use Gaussian quadrature.

⟨ Prototype for *RT_Cone* 92 ⟩ ≡
void *RT_Cone*(**int** *n*, **struct AD_slab_type** ∗*slab*, **int** *use_cone*, **double** ∗UR1, **double** ∗UT1, **double** ∗URU, **double** ∗UTU)

This code is used in sections 90 and 93.

93. ⟨ Definition for *RT_Cone* 93 ⟩ ≡
⟨ Prototype for *RT_Cone* 92 ⟩
{
⟨ *RT_Cone* Declare variables 94 ⟩
⟨ *RT_Cone* Check inputs 95 ⟩
⟨ *RT_Cone* Allocate slab memory 97 ⟩
⟨ *RT_Cone* Initialize homogeneous layer 98 ⟩
⟨ *RT_Cone* Allocate and generate top and bottom boundaries 99 ⟩
⟨ *RT_Cone* Add top and bottom boundaries 100 ⟩
⟨ *RT_Cone* Free memory 101 ⟩
}

This code is used in section 89.

94. ⟨ *RT_Cone* Declare variables 94 ⟩ ≡
struct AD_method_type *method*;
double ∗R01, ∗R10, ∗T01, ∗T10;
double ∗R23, ∗R32, ∗T23, ∗T32;
double ∗∗R12, ∗∗T12;
double ∗∗R02, ∗∗T02, ∗∗T20, ∗∗R20;
double ∗∗R03, ∗∗T03, ∗∗T30, ∗∗R30;
double ∗∗*atemp*, ∗∗*btemp*;
double *d*;
∗UR1 = −1;
∗URU = −1;
∗UT1 = −1;
∗UTU = −1;

This code is used in section 93.

95.

⟨ *RT_Cone* Check inputs 95 ⟩ ≡

```
if (slab→n_slab < 0) return;
if (slab→n_top_slide < 0) return;
if (slab→n_bottom_slide < 0) return;
if (slab→a < 0 ∨ slab→a > 1) return;
if (slab→g < −1 ∨ slab→g > 1) return;
if (slab→b < 0) return;
if (slab→cos_angle < 0 ∨ slab→cos_angle > 1) return;
```

See also section 96.

This code is used in section 93.

96. The number of quadrature points must be fixed before starting to allocate memory. We want the number of points to be at least twelve so that each of the three integrals will have four quadrature points.

⟨ *RT_Cone* Check inputs 95 ⟩ +≡

```
n = 12 ∗ (n/12);
if (n < 12) n = 12;
method.quad_pts = n;
```

97. ⟨ *RT_Cone* Allocate slab memory 97 ⟩ ≡

```
R12 = dmatrix(1, n, 1, n);
T12 = dmatrix(1, n, 1, n);
R02 = dmatrix(1, n, 1, n);
T02 = dmatrix(1, n, 1, n);
R20 = dmatrix(1, n, 1, n);
T20 = dmatrix(1, n, 1, n);
R03 = dmatrix(1, n, 1, n);
T03 = dmatrix(1, n, 1, n);
R30 = dmatrix(1, n, 1, n);
T30 = dmatrix(1, n, 1, n);
atemp = dmatrix(1, n, 1, n);
btemp = dmatrix(1, n, 1, n);
```

This code is used in section 93.

98. The homogeneous layer initially has 0% reflection and 100% transmission. We cannot fob the details on how this layer is created to *RT_Matrices* because we need to (1) set the quadrature angles to a multiple of three, and (2) explicitly make a call to *Choose_Cone_Method* so that the quadrature angles will get chosen appropriately.

This code is directly lifted from the *RT_Matrices* routine.

⟨ *RT_Cone* Initialize homogeneous layer 98 ⟩ ≡

```
Choose_Cone_Method(slab, &method);
if (slab→b ≤ 0) {
  Zero_Layer(n, R12, T12);
  return;
}
n = method.quad_pts;
Init_Layer(∗slab, method, R12, T12);
d = 1.0;
if (slab→b ≠ HUGE_VAL) d = method.b_thinnest ∗ slab→b/method.b_calc;
Double_Until(n, R12, T12, d, slab→b);
```

This code is used in section 93.

99. Create the matrices needed for the top and bottom

⟨ *RT_Cone* Allocate and generate top and bottom boundaries 99 ⟩ ≡

```
R01 = dvector(1, n);
R10 = dvector(1, n);
T01 = dvector(1, n);
T10 = dvector(1, n);
Init_Boundary(*slab, n, R01, R10, T01, T10, TOP_BOUNDARY);
R23 = dvector(1, n);
R32 = dvector(1, n);
T23 = dvector(1, n);
T32 = dvector(1, n);
Init_Boundary(*slab, n, R23, R32, T23, T32, BOTTOM_BOUNDARY);
```

This code is used in section 93.

100. Here the layer numbering is pretty consistent. The top slide is 01, the scattering layer is 12, and the bottom slide is 23. Light going from the top of the slide to the bottom of the scattering layer is 02 and similarly light going all the way through is 03.

The only tricky part is that the definitions of UR1 and URU have changed from their usual definitions. When *use_cone* ≡ OBLIQUE then UR1 refers to the light reflected back into the specified cone for normal irradiance and URU is for light reflected back into the cone for light incident uniformly at all angles within that cone. Otherwise, assume that the incidence is oblique. UR1 then refers to the total amount of light reflected back for light incident only at the cone angle.

⟨ *RT_Cone* Add top and bottom boundaries 100 ⟩ ≡

```
Add_Top(n, R01, R10, T01, T10, R12, R12, T12, T12, R02, R20, T02, T20, atemp, btemp);
Add_Bottom(n, R02, R20, T02, T20, R23, R32, T23, T32, R03, R30, T03, T30, atemp, btemp);
if (use_cone ≡ CONE) {
  URU_and_UR1_Cone(n, slab→n_slab, slab→cos_angle, R03, URU, UR1);
  Transpose_Matrix(n, T03);
  URU_and_UR1_Cone(n, slab→n_slab, slab→cos_angle, T03, UTU, UT1);
}
else {
  {
    double unused;
    if (use_cone ≠ OBLIQUE)
      fprintf(stderr, "Unknown␣type␣for␣use_cone.␣␣Assuming␣oblique␣incidence.\n");
    URU_and_URx_Cone(n, slab→n_slab, slab→cos_angle, R03, URU, UR1);
    URU_and_UR1(n, slab→n_slab, R03, URU, &unused);
    Transpose_Matrix(n, T03);
    URU_and_URx_Cone(n, slab→n_slab, slab→cos_angle, T03, UTU, UT1);
    URU_and_UR1(n, slab→n_slab, T03, UTU, &unused);
  }
}
```

This code is used in section 93.

101. ⟨ *RT_Cone* Free memory 101 ⟩ ≡

```
free_dvector(R01, 1, n);
free_dvector(R10, 1, n);
free_dvector(T01, 1, n);
free_dvector(T10, 1, n);
free_dmatrix(R12, 1, n, 1, n);
free_dmatrix(T12, 1, n, 1, n);
free_dmatrix(R03, 1, n, 1, n);
free_dmatrix(R30, 1, n, 1, n);
free_dmatrix(T03, 1, n, 1, n);
free_dmatrix(T30, 1, n, 1, n);
free_dmatrix(R02, 1, n, 1, n);
free_dmatrix(R20, 1, n, 1, n);
free_dmatrix(T02, 1, n, 1, n);
free_dmatrix(T20, 1, n, 1, n);
free_dmatrix(atemp, 1, n, 1, n);
free_dmatrix(btemp, 1, n, 1, n);
free_dvector(R32, 1, n);
free_dvector(R23, 1, n);
free_dvector(T32, 1, n);
free_dvector(T23, 1, n);
```

This code is used in section 93.

102. Simple wrapper that avoids data structures

⟨ Prototype for *ez_RT_Cone* 102 ⟩ ≡

```
void ez_RT_Cone(int n, double nslab, double ntopslide, double nbottomslide, double a, double
    b, double g, double cos_cone_angle, double *UR1, double *UT1, double *URU, double *UTU)
```

This code is used in sections 90, 91, and 103.

103. ⟨ Definition for *ez_RT_Cone* 103 ⟩ ≡

```
⟨ Prototype for ez_RT_Cone 102 ⟩
{
  struct AD_slab_type slab;

  slab.n_slab = nslab;
  slab.n_top_slide = ntopslide;
  slab.n_bottom_slide = nbottomslide;
  slab.b_top_slide = 0;
  slab.b_bottom_slide = 0;
  slab.a = a;
  slab.b = b;
  slab.g = g;
  slab.cos_angle = cos_cone_angle;
  slab.phase_function = HENYEY_GREENSTEIN;
  RT_Cone(n, &slab, CONE, UR1, UT1, URU, UTU);
}
```

This code is used in section 89.

104. This routine calculates reflection and transmission for oblique incidence. *URx* and *UTx* are the total light reflected and transmitted for light incident at *cos_oblique_angle*. `URU` and `UTU` are the same thing for diffuse incident light.

⟨ Prototype for *ez_RT_Oblique* 104 ⟩ ≡

void *ez_RT_Oblique*(**int** *n*, **double** *nslab*, **double** *ntopslide*, **double** *nbottomslide*, **double** *a*, **double** *b*, **double** *g*, **double** *cos_oblique_angle*, **double** **URx*, **double** **UTx*, **double** *`URU`, **double** *`UTU`)

This code is used in sections 90, 91, and 105.

105. ⟨ Definition for *ez_RT_Oblique* 105 ⟩ ≡

```
⟨ Prototype for ez_RT_Oblique 104 ⟩
{
  struct AD_slab_type slab;
  slab.n_slab = nslab;
  slab.n_top_slide = ntopslide;
  slab.n_bottom_slide = nbottomslide;
  slab.b_top_slide = 0;
  slab.b_bottom_slide = 0;
  slab.a = a;
  slab.b = b;
  slab.g = g;
  slab.cos_angle = cos_oblique_angle;
  slab.phase_function = HENYEY_GREENSTEIN;
  RT_Cone(n, &slab, OBLIQUE, URx, UTx, URU, UTU);
}
```

This code is used in section 89.

106. 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.

```
#include <math.h>
#include <float.h>
#include <stdio.h>
#include "ad_frsnl.h"
#include "ad_globl.h"
#include "ad_matrx.h"
#include "ad_phase.h"
#include "ad_radau.h"
#include "ad_start.h"
#include "nr_gaulg.h"
#include "nr_util.h"
  ⟨ Definition for Get_Start_Depth 110 ⟩
  ⟨ Definition for Quadrature 113 ⟩
  ⟨ Definition for Choose_Method 115 ⟩
  ⟨ Definition for Choose_Cone_Method 117 ⟩
  ⟨ Definition for Get_Diamond_Layer 126 ⟩
  ⟨ Definition for Init_Layer 138 ⟩
```

107. ⟨ ad_start.h 107 ⟩ ≡

```
  ⟨ Prototype for Get_Start_Depth 109 ⟩;
  ⟨ Prototype for Choose_Method 114 ⟩;
  ⟨ Prototype for Choose_Cone_Method 116 ⟩;
  ⟨ Prototype for Init_Layer 137 ⟩;
  ⟨ Prototype for Quadrature 112 ⟩;
```

108. 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

109. *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 μ. Also we make checks for a layer with zero thickness and one that infinitely thick.

⟨ Prototype for *Get_Start_Depth* 109 ⟩ ≡
  double *Get_Start_Depth*(**double** *mu*, **double** *d*)

This code is used in sections 107 and 110.

110. ⟨ Definition for *Get_Start_Depth* 110 ⟩ ≡

```
⟨ Prototype for Get_Start_Depth 109 ⟩
{
  if (d ≤ 0) return 0.0;
  if (d ≡ HUGE_VAL) return (mu/2.0);
  while (d > mu) d /= 2;
  return d;
}
```

This code is used in section 106.

111. Quadrature.

112. 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 μ_c then Radau quadrature points are chosen from 0 to μ_c and Radau quadrature points over the interval μ_c to 1.

⟨ Prototype for *Quadrature* 112 ⟩ ≡
```
void Quadrature(int n, double n_slab, double *x, double *w)
```
This code is used in sections 107 and 113.

113. ⟨ Definition for *Quadrature* 113 ⟩ ≡
```
⟨ Prototype for Quadrature 112 ⟩
{
  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 106.

114. *Choose_Method* fills the method structure with correct values for *a_calc*, *b_calc*, *g_calc*, and *b_thinnest* based on the delta-M method. Furthermore, the quadrature angles and weights are also calculated. Before calling this routines *method.quad_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 *angle*, *weight*, and *twoaw*
4. determine the thickness of the thinnest layer

⟨ Prototype for *Choose_Method* 114 ⟩ ≡
```
void Choose_Method(struct AD_slab_type *slab, struct AD_method_type *method)
```
This code is used in sections 107 and 115.

115. ⟨ Definition for *Choose_Method* 115 ⟩ ≡

```
⟨ Prototype for Choose_Method 114 ⟩
{
  double af;
  int i, n;
  if (0 < slab→cos_angle ∧ slab→cos_angle < 1) {
    Choose_Cone_Method(slab, method);
    return;
  }
  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);
}
```

This code is used in section 106.

116. *Choose_Cone_Method* adds the ability to specify a specific quadrature angle so that accurate estimates of the reflection and transmission might be made for when the light returning in a particular cone is of interest. This code mimicks the usual *Choose_Method* above, and in fact explicitly uses it for a couple of special cases.

⟨ Prototype for *Choose_Cone_Method* 116 ⟩ ≡

```
void Choose_Cone_Method(struct AD_slab_type ∗slab, struct AD_method_type ∗method)
```

This code is used in sections 107 and 117.

117. ⟨ Definition for *Choose_Cone_Method* 117 ⟩ ≡

```
⟨ Prototype for Choose_Cone_Method 116 ⟩
{
  double af, ∗angle1, ∗weight1, cos_crit_angle, mu;
  int i, n, nby2, nby3;
  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;
  ⟨ Special case when cosine is zero 120 ⟩
  ⟨ Special case when no index of refraction change 121 ⟩
  ⟨ Gaussian quadrature from 0 to the critical angle 122 ⟩
  ⟨ Radau quadrature from the critical angle to the cone angle 123 ⟩
  ⟨ Radau quadrature from the cone angle to 1 124 ⟩
}
```

This code is used in section 106.

118. ⟨ print angles 118 ⟩ ≡

This code is used in sections 120, 121, and 124.

119. ⟨ debug print angles 119 ⟩ ≡

```
{
  printf("****Cone␣Angle␣␣␣␣␣␣␣␣␣␣=␣%6.2f␣degrees,␣Cosine()=%6.4f\n",
      acos(slab→cos_angle) * 180.0/M_PI, slab→cos_angle);
  double sum = 0;
  for (i = 1; i ≤ n; i++) {
    sum += twoaw[i];
    printf("%02d␣theta=%6.2f␣cos(theta)=%6.4f␣w=%6.4f␣2aw=%6.4f\n", i,
        acos(angle[i])/M_PI * 180.0, angle[i], weight[i], twoaw[i]);
  }
  printf("twoaw␣sum␣=␣%8.4f\n", sum);
}
```

120. When the cone angle is zero or ninety degrees then we can just use the standard method for choosing the quadrature points.

⟨ Special case when cosine is zero 120 ⟩ ≡

```
if (slab→cos_angle ≡ 0 ∨ slab→cos_angle ≡ 1) {
  Choose_Method(slab, method);
  ⟨ print angles 118 ⟩
  return;
}
```

This code is used in section 117.

121. When there is no index of refraction change, there is no critical angle to worry about. Since we want the cone angle to be included as one of our angles, we use Radau quadrature. That way both the cone angle and perpendicular angles are included.

⟨ Special case when no index of refraction change 121 ⟩ ≡

```
if (slab→n_slab ≡ 1 ∧ slab→n_top_slide ≡ 1 ∧ slab→n_bottom_slide ≡ 1) {
  nby2 = n/2;
  Radau(0.0, slab→cos_angle, angle, weight, nby2);
  angle1 = dvector(1, nby2);
  weight1 = dvector(1, nby2);
  Radau(slab→cos_angle, 1.0, angle1, weight1, nby2);
  for (i = 1; i ≤ nby2; i++) {
    angle[nby2 + i] = angle1[i];
    weight[nby2 + i] = weight1[i];
  }
  free_dvector(angle1, 1, nby2);
  free_dvector(weight1, 1, nby2);
  for (i = 1; i ≤ n; i++) twoaw[i] = 2 * angle[i] * weight[i];
  method→b_thinnest = Get_Start_Depth(angle[1], method→b_calc);
  ⟨ print angles 118 ⟩
  return;
}
```

This code is used in section 117.

122. Now we need to include three angles, the critical angle, the cone angle, and perpendicular. Now the important angles are the ones in the slab. So we calculate the cosine of the critical angle in the slab and cosine of the cone angle in the slab.

The critical angle will always be greater than the cone angle in the slab and therefore the cosine of the critical angle will always be less than the cosine of the cone angle. Thus we will integrate from zero to the cosine of the critical angle (using Gaussian quadrature to avoid either endpoint) then from the critical angle to the cone angle (using Radau quadrature so that the cosine angle will be included) and finally from the cone angle to 1 (again using Radau quadrature so that 1 will be included).

⟨ Gaussian quadrature from 0 to the critical angle 122 ⟩ ≡

```
cos_crit_angle = Cos_Critical_Angle(slab→n_slab, 1.0);
nby3 = n/3;
gauleg(0.0, cos_crit_angle, angle, weight, nby3);
```

This code is used in section 117.

123. ⟨ Radau quadrature from the critical angle to the cone angle 123 ⟩ ≡

```
mu = sqrt(slab→n_slab * slab→n_slab − 1 + slab→cos_angle * slab→cos_angle)/slab→n_slab;
angle1 = dvector(1, nby3);
weight1 = dvector(1, nby3);
Radau(cos_crit_angle, mu, angle1, weight1, nby3);
for (i = 1; i ≤ nby3; i++) {
    angle[nby3 + i] = angle1[i];
    weight[nby3 + i] = weight1[i];
}
```

This code is used in section 117.

124. ⟨ Radau quadrature from the cone angle to 1 124 ⟩ ≡

```
Radau(mu, 1.0, angle1, weight1, nby3);
for (i = 1; i ≤ nby3; i++) {
    angle[nby3 * 2 + i] = angle1[i];
    weight[nby3 * 2 + i] = weight1[i];
}
free_dvector(angle1, 1, nby3);
free_dvector(weight1, 1, nby3);
for (i = 1; i ≤ n; i++) twoaw[i] = 2 * angle[i] * weight[i];
method→b_thinnest = Get_Start_Depth(angle[1], method→b_calc);
⟨ print angles 118 ⟩
```

This code is used in section 117.

125. 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

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

When this equation is integrated over a thin layer from τ_0^* to τ_1^* then get

$$\pm\nu_i[L(\tau_1^*,\pm\nu_i) - L(\tau_0^*,\pm\nu_i)] + dL_{1/2}(\pm\nu_i)$$
$$= \frac{a}{2}\sum_{j=1}^{M} w_j d\left[h(\nu_i,\nu_j)L_{1/2}(\pm\nu_i) + h(\nu_i,-\nu_j)L_{1/2}(\mp\nu_i)\right]$$

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 infinitesmal generator initialization makes the approximation

$$L_{1/2}(-\nu) = L(\tau_1^*,-\nu) \qquad\qquad L_{1/2}(\nu) = L(\tau_0^*,\nu)$$

and the diamond initialization assumes

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

126. 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* 126 ⟩ ≡

```
static void Get_Diamond_Layer(struct AD_method_type method, double **h, double **R, double
        **T)
{
  ⟨ Local variables and initialization 134 ⟩
  ⟨ Find r and t 127 ⟩
  ⟨ Find C = r/(1 + t) 128 ⟩
  ⟨ Find G = 0.5(1 + t − Cr) 129 ⟩
  ⟨ print r, t, and g for Martin Hammer 130 ⟩
  ⟨ Calculate R and T 131 ⟩
  ⟨ Free up memory 135 ⟩
}
```

This code is used in section 106.

127. 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

$$r = \frac{d}{2}\hat{R} \qquad t = \frac{d}{2}\hat{T}$$

Recall that

$$\hat{R}_{ij} = \frac{a}{2\mu_i} h_{ij}^{+-} w_j \qquad \hat{T}_{ij} = \frac{\delta_{ij}}{\mu_i} - \frac{a}{2\mu_i} h_{ij}^{++} w_j$$

therefore

$$r_{ij} = \frac{adw_j}{4\mu_i} h_{ij}^{+-} \qquad t_{ij} = \delta_{ij}\frac{d}{2\mu_i} - \frac{adw_j}{4\mu_i} h_{ij}^{++}$$

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.

⟨ Find r and t 127 ⟩ ≡

```
for (j = 1; j ≤ n; j++) {
  temp = a * d * weight[j]/4;
  for (i = 1; i ≤ n; i++) {
    c = temp/angle[i];
    R[i][j] = c * h[i][−j];
    T[i][j] = −c * h[i][j];
  }
  T[j][j] += d/(2 * angle[j]);
}
```

This code is used in section 126.

128. 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 *Left_Inverse_Multiply*. We just need to create $A = I + T$ and fire it off to *Left_Inverse_Multiply*. Actually, Wiscome 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 voilá! It worked.)

⟨ Find $C = r/(1 + t)$ 128 ⟩ ≡

```
for (i = 1; i ≤ n; i++) {
  for (j = 1; j ≤ n; j++) A[i][j] = T[i][j];
  A[i][i] += 1.0;
}
Left_Inverse_Multiply(n, A, R, C);
```

This code is used in section 126.

129. Here the matrix

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

is formed.

⟨ Find $G = 0.5(1 + t - Cr)$ 129 ⟩ ≡

```
Matrix_Multiply(n, C, R, G);
for (i = 1; i ≤ n; i++) {
  for (j = 1; j ≤ n; j++) G[i][j] = (T[i][j] − G[i][j])/2;
  G[i][i] += 0.5;
}
```

This code is used in section 126.

130. To print intermediate results for Chapter 4 of AJ's book, then it is necessary to print things from within *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 `MARTIN_HAMMER` is defined, and when the variable *Martin_Hammer* $\neq 0$.

⟨ print r, t, and g for Martin Hammer 130 ⟩ ≡

```
#ifdef MARTIN_HAMMER
  {
    double **Ginv, **G2;
    if (Martin_Hammer ≠ 0) {
      printf("A␣from␣equation␣5.55\n");
      wrmatrix(n, T);
      printf("B␣from␣equation␣5.55\n");
      wrmatrix(n, R);
      Ginv = dmatrix(1, n, 1, n);
      G2 = dmatrix(1, n, 1, n);
      for (i = 1; i ≤ n; i++) {
        for (j = 1; j ≤ n; j++) {
          G2[i][j] = G[i][j] * 2.0;
        }
      }
      Matrix_Inverse(n, G2, Ginv);
      printf("Inverse␣of␣G␣from␣equation␣5.56\n");
      wrmatrix(n, G2);
      printf("G␣from␣equation␣5.56\n");
      wrmatrix(n, Ginv);
      free_matrix(Ginv, 1, n, 1, n);
      free_matrix(G2, 1, n, 1, n);
    }
  }
#endif
```

This code is used in section 126.

131. 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

$$GR = C \qquad\qquad G(t+I) = I$$

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 131 ⟩ ≡
  Transpose_Matrix(n, G);
  Decomp(n, G, &condition, ipvt);
  if (condition ≡ 1 · 10^32) AD_error("Singular␣Matrix␣...␣failed␣in␣diamond_init\n");
  for (i = 1; i ≤ n; i++) {
    ⟨ Solve for row of R 132 ⟩
    ⟨ Solve for row of T 133 ⟩
  }
#ifdef MARTIN_HAMMER
  {
    double **T2, **Ginv;
    if (Martin_Hammer ≡ 5) {
      T2 = dmatrix(1, n, 1, n);
      Ginv = dmatrix(1, n, 1, n);
      Copy_Matrix(n, T, T2);
      for (i = 1; i ≤ n; i++) {
        T2[i][i] += 1/twoaw[i];
      }
      for (i = 1; i ≤ n; i++) {
        for (j = 1; j ≤ n; j++) {
          T2[i][j] *= twoaw[j] * 0.5;
        }
      }
      printf("G=(T-1)/2␣from␣equation␣5.55\n");
      wrmatrix(n, T2);
      Matrix_Inverse(n, T2, Ginv);
      printf("1/G\n");
      wrmatrix(n, Ginv);
      free_matrix(T2, 1, n, 1, n);
      free_matrix(Ginv, 1, n, 1, n);
    }
  }
#endif
```

This code is used in section 126.

132. We use the decomposed form of G to find R. Since G is now the LU decomposition of G^T, we must pass rows of the C to *Solve* and get rows back. Note the finess with

$$\mathrm{work}_j = C_{ji}\frac{a_j w_j}{a_i w_i}$$

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.

⟨ Solve for row of R 132 ⟩ ≡

```
for (j = 1; j ≤ n; j++) work[j] = C[j][i] * twoaw[j]/twoaw[i];
Solve(n, G, work, ipvt);
for (j = 1; j ≤ n; j++) R[i][j] = work[j]/twoaw[j];
```

This code is used in section 131.

133. 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!

⟨ Solve for row of T 133 ⟩ ≡

```
for (j = 1; j ≤ n; j++) work[j] = 0;
work[i] = 1.0;
Solve(n, G, work, ipvt);
for (j = 1; j ≤ n; j++) T[i][j] = work[j]/twoaw[j];
T[i][i] −= 1.0/twoaw[i];    /* Subtract Identity Matrix */
```

This code is used in section 131.

134. Pretty standard stuff here. Allocate memory and print a warning if the thickness is too small.

⟨ Local variables and initialization 134 ⟩ ≡

```
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_pts;
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);
```

This code is used in section 126.

135.

⟨ Free up memory 135 ⟩ ≡

```
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 126.

136. Layer Initialization.

137. *Init_Layer* returns reflection and transmission matrices for a thin layer. Space must previously been allocated for R and T.

⟨ Prototype for *Init_Layer* 137 ⟩ ≡
  void *Init_Layer*(**struct AD_slab_type** *slab*, **struct AD_method_type** *method*, **double** $**R$, **double** $**T$)

This code is used in sections 107 and 138.

138. ⟨ Definition for *Init_Layer* 138 ⟩ ≡

```
⟨ Prototype for Init_Layer 137 ⟩
{
  static double **h = Λ;
  static double current_g = 10.0;
  int n;

  n = method.quad_pts;
  if (slab.b ≤ 0) {
    Zero_Layer(n, R, T);
    return;
  }    /* allocate space for redistribution function */
  if (h ≡ Λ) h = dmatrix(−n, n, −n, n);
  if (current_g ≠ method.g_calc) {
    current_g = method.g_calc;
    Get_Phi(n, slab.phase_function, method.g_calc, h);
  }
  Get_Diamond_Layer(method, h, R, T);
}
```

This code is used in section 106.

139. AD Double. This has the routines needed to add layers together in various combinations.

```
⟨ ad_doubl.c 139 ⟩ ≡
#include <math.h>
#include <float.h>
#include <stdio.h>
#include "nr_util.h"
#include "ad_matrx.h"
#include "ad_globl.h"
#include "ad_doubl.h"
  ⟨ Definition for Star_Multiply 159 ⟩
  ⟨ Definition for Star_One_Minus 160 ⟩
  ⟨ Definition for Basic_Add_Layers 141 ⟩
  ⟨ Definition for Basic_Add_Layers_With_Sources 142 ⟩
  ⟨ Definition for Add 145 ⟩
  ⟨ Definition for Add_With_Sources 147 ⟩
  ⟨ Definition for Add_Homogeneous 149 ⟩
  ⟨ Definition for Double_Once 151 ⟩
  ⟨ Definition for Double_Until 153 ⟩
  ⟨ Definition for Double_Until_Infinite 155 ⟩
  ⟨ Definition for Between 157 ⟩
```

140. ⟨ ad_doubl.h 140 ⟩ ≡

```
  ⟨ Prototype for Add 144 ⟩;
  ⟨ Prototype for Add_With_Sources 146 ⟩;
  ⟨ Prototype for Add_Homogeneous 148 ⟩;
  ⟨ Prototype for Double_Once 150 ⟩;
  ⟨ Prototype for Double_Until 152 ⟩;
  ⟨ Prototype for Double_Until_Infinite 154 ⟩;
  ⟨ Prototype for Between 156 ⟩;
```

141. Basic Routine to Add Layers Without Sources.
The basic equations for the adding-doubling method (neglecting sources) are

$$\mathbf{T}^{02} = \mathbf{T}^{12}(\mathbf{E} - \mathbf{R}^{10}\mathbf{R}^{12})^{-1}\mathbf{T}^{01}$$

$$\mathbf{R}^{20} = \mathbf{T}^{12}(\mathbf{E} - \mathbf{R}^{10}\mathbf{R}^{12})^{-1}\mathbf{R}^{10}\mathbf{T}^{21} + \mathbf{R}^{21}$$

$$\mathbf{T}^{20} = \mathbf{T}^{10}(\mathbf{E} - \mathbf{R}^{12}\mathbf{R}^{10})^{-1}\mathbf{T}^{21}$$

$$\mathbf{R}^{02} = \mathbf{T}^{10}(\mathbf{E} - \mathbf{R}^{12}\mathbf{R}^{10})^{-1}\mathbf{R}^{12}\mathbf{T}^{01} + \mathbf{R}^{01}$$

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,

$$\mathbf{T}^{02} = \mathbf{T}^{12}(\mathbf{E} - \mathbf{R}^{10} \star \mathbf{R}^{12})^{-1}\mathbf{T}^{01}$$

and

$$\mathbf{R}^{20} = \mathbf{T}^{12}(\mathbf{E} - \mathbf{R}^{10} \star \mathbf{R}^{12})^{-1}\mathbf{R}^{10} \star \mathbf{T}^{21} + \mathbf{R}^{21}$$

where the identity matrix $\mathbf{E}$ is then

$$\mathbf{E}^{ij} = \frac{1}{2\mu_i w_i}\delta_{ij}$$

where δ_{ij} is the usual Kronecker delta. It is noteworthy that if say $R^{10} \equiv 0$, then $\mathbf{E}^{-1} \equiv \mathbf{c}$ and so

$$\mathbf{T}^{02} = \mathbf{T}^{12}\mathbf{c}\mathbf{T}^{01} = \mathbf{T}^{12} \star \mathbf{T}^{01}$$

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 *Left_Inverse_Multiply* could be called with an argument list like *Left_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 *Left_Inverse_Multiply* routine and no net gain would result.

⟨ Definition for *Basic_Add_Layers* 141 ⟩ ≡

```
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 ⋆ R^12 */
  Star_One_Minus(n, a);      /* a = E − R^10 ⋆ 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 ⋆ R^12)^−1 R^10 */
  Star_Multiply(n, a, T21, a);      /* a = T^12(E − R^10 ⋆ R^12)^−1 R^10 ⋆ 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 139.

142. 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

$$\mathbf{J}^{02}_{+} = \mathbf{J}^{12}_{+} + \mathbf{T}^{12}(\mathbf{E} - \mathbf{R}^{10}\mathbf{R}^{12})^{-1}(\mathbf{J}^{01}_{+} + \mathbf{R}^{10}\mathbf{J}^{21}_{-})$$

and

$$\mathbf{J}^{20}_{+} = \mathbf{J}^{10}_{-} + \mathbf{T}^{10}(\mathbf{E} - \mathbf{R}^{12}\mathbf{R}^{10})^{-1}(\mathbf{J}^{21}_{-} + \mathbf{R}^{12}\mathbf{J}^{01}_{+})$$

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

$$\mathbf{J}^{02} = \mathbf{J}^{12} + \mathbf{T}^{12}(\mathbf{E} - \mathbf{R}^{10}\mathbf{R}^{12})^{-1}(\mathbf{J}^{01} + \mathbf{R}^{10}\mathbf{J}^{21})$$

and

$$\mathbf{J}^{20} = \mathbf{J}^{10} + \mathbf{T}^{10}(\mathbf{E} - \mathbf{R}^{12}\mathbf{R}^{10})^{-1}(\mathbf{J}^{21} + \mathbf{R}^{12}\mathbf{J}^{01})$$

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,

$$\mathbf{J}^{02} = \mathbf{J}^{12} + \mathbf{T}^{12}(\mathbf{E} - \mathbf{R}^{10} \star \mathbf{R}^{12})^{-1}(\mathbf{J}^{01} + \mathbf{R}^{10} \star \mathbf{J}^{21})$$

⟨ Definition for *Basic_Add_Layers_With_Sources* 142 ⟩ ≡

```
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 ⋆ R^12 */
  Star_One_Minus(n, a);   /* a = E − R^10 ⋆ 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 ⋆ R^12)^−1 R^10 */
  Star_Multiply(n, a, T21, a);   /* a = T^12(E − R^10 ⋆ R^12)^−1 R^10 ⋆ T^21 */
  Matrix_Sum(n, R21, a, R20);
  Copy_Matrix(n, T01, a);
  Matrix_Multiply(n, b, a, T02);
  Star_Multiply(n, R10, J21, a);   /* a = R^10 ⋆ J^21 */
  Matrix_Sum(n, J01, a, a);   /* a = J^01 + R^10 ⋆ J^21 */
  Matrix_Multiply(n, b, a, J02);   /* J02 = T^12(E − R^10 ⋆ R^12)^−1(J^01 + R^10 ⋆ J^21) */
  Matrix_Sum(n, J02, J12, J02);
}
```

This code is used in section 139.

143. Higher level routines.

144.

⟨ Prototype for *Add* 144 ⟩ ≡

void *Add*(**int** n, **double** **R01, **double** **R10, **double** **T01, **double** **T10, **double** **R12, **double** **R21, **double** **T12, **double** **T21, **double** **R02, **double** **R20, **double** **T02, **double** **T20)

This code is used in sections 140 and 145.

145. *Add* 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, and T02 should be distinct from the input matrices.

⟨ Definition for *Add* 145 ⟩ ≡
```
⟨ Prototype for Add 144 ⟩
{
  ⟨ Allocate memory for a and b 161 ⟩
  Basic_Add_Layers(n, R10, T01, R12, R21, T12, T21, R20, T02, a, b);
  Basic_Add_Layers(n, R12, T21, R10, R01, T10, T01, R02, T20, a, b);
  ⟨ Free Memory for a and b 162 ⟩
}
```
This code is used in section 139.

146.

⟨ Prototype for *Add_With_Sources* 146 ⟩ ≡

void *Add_With_Sources*(**int** n, **double** **R01, **double** **R10, **double** **T01, **double** **T10, **double** **J01, **double** **J10, **double** **R12, **double** **R21, **double** **T12, **double** **T21, **double** **J12, **double** **J21, **double** **R02, **double** **R20, **double** **T02, **double** **T20, **double** **J02, **double** **J20)

This code is used in sections 140 and 147.

147. *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, T02, J20, and J02 should be distinct from the input matrices.

⟨ Definition for *Add_With_Sources* 147 ⟩ ≡
```
⟨ Prototype for Add_With_Sources 146 ⟩
{
  ⟨ Allocate memory for a and b 161 ⟩
  Basic_Add_Layers_With_Sources(n, R10, T01, R12, R21, T12, T21, R20, T02, J01, J12, J21, J02, a, b);
  Basic_Add_Layers_With_Sources(n, R12, T21, R10, R01, T10, T01, R02, T20, J21, J10, J01, J20, a, b);
  ⟨ Free Memory for a and b 162 ⟩
}
```
This code is used in section 139.

148.

⟨ Prototype for *Add_Homogeneous* 148 ⟩ ≡

void *Add_Homogeneous*(**int** n, **double** **R01, **double** **T01, **double** **R12, **double** **T12, **double** **R02, **double** **T02)

This code is used in sections 140 and 149.

149.

⟨ Definition for *Add_Homogeneous* 149 ⟩ ≡
  ⟨ Prototype for *Add_Homogeneous* 148 ⟩
  {
    ⟨ Allocate memory for a and b 161 ⟩
    Basic_Add_Layers(n, R01, T01, R12, R12, T12, T12, R02, T02, a, b);
    ⟨ Free Memory for a and b 162 ⟩
  }

This code is used in section 139.

150. This just adds a layer to itself. Couldn't *Basic_Add_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.

⟨ Prototype for *Double_Once* 150 ⟩ ≡
  void *Double_Once*(**int** n, **double** $**R$, **double** $**T$)

This code is used in sections 140 and 151.

151.

⟨ Definition for *Double_Once* 151 ⟩ ≡
  ⟨ Prototype for *Double_Once* 150 ⟩
  {
    ⟨ Allocate memory for a and b 161 ⟩
    Basic_Add_Layers($n, R, T, R, R, T, T, R, T, a, b$);
    ⟨ Free Memory for a and b 162 ⟩
  }

This code is used in section 139.

152. *Double_Until* and *Double_Until_Infinite* 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., $end \neq start \cdot 2^k$. Futhermore, 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.

⟨ Prototype for *Double_Until* 152 ⟩ ≡
  void *Double_Until*(**int** n, **double** $**r$, **double** $**t$, **double** *start*, **double** *end*)

This code is used in sections 140 and 153.

153.

⟨ Definition for *Double_Until* 153 ⟩ ≡
```
⟨ Prototype for Double_Until 152 ⟩
{
  if (end ≡ HUGE_VAL) {
    Double_Until_Infinite(n, r, t);
    return;
  }
  {
    ⟨ Allocate memory for a and b 161 ⟩
    while (fabs(end − start) > 0.00001 ∧ end > start) {
      Basic_Add_Layers(n, r, t, r, r, t, t, r, t, a, b);
      start ∗= 2;
    }
    ⟨ Free Memory for a and b 162 ⟩
  }
}
```
This code is used in section 139.

154. *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* 154 ⟩ ≡
```
void Double_Until_Infinite(int n, double **r, double **t)
```
This code is used in sections 140 and 155.

155.

⟨ Definition for *Double_Until_Infinite* 155 ⟩ ≡
```
⟨ Prototype for Double_Until_Infinite 154 ⟩
{
  double oldutu, UTU, UT1;
  ⟨ Allocate memory for a and b 161 ⟩
  UTU = 0.0;
  do {
    oldutu = UTU;
    Basic_Add_Layers(n, r, t, r, 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 162 ⟩
}
```
This code is used in section 139.

156. Internal Radiance.

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

$$\mathbf{L}_{-} = (\mathbf{E} - \mathbf{R}^{12} \star \mathbf{R}^{10})^{-1}(\mathbf{R}^{12} \star \mathbf{T}^{01} \star \mathbf{L}^{0}_{+} + \mathbf{T}^{21} \star \mathbf{L}^{2}_{-})$$

where $\mathbf{L}^{0}_{+}$ is the downward radiance on the top layer and $\mathbf{L}^{2}_{-}$ is the upward radiance on the bottom layer. The equation for the downward mid-layer radiance can be obtained similarly using

$$\mathbf{L}_{+} = (\mathbf{E} - \mathbf{R}^{10} \star \mathbf{R}^{12})^{-1}(\mathbf{T}^{01} \star \mathbf{L}^{0}_{+} + \mathbf{R}^{10} \star \mathbf{T}^{21} \star \mathbf{L}^{2}_{-})$$

Now assume that $\mathbf{L}^{2}_{-}$ is zero. Then the matrix

$$\mathbf{L}_{-} = (\mathbf{E} - \mathbf{R}^{12} \star \mathbf{R}^{10})^{-1}\mathbf{R}^{12} \star \mathbf{T}^{01}$$

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

$$\mathbf{L}_{+} = (\mathbf{E} - \mathbf{R}^{10} \star \mathbf{R}^{12})^{-1}\mathbf{T}^{01}$$

⟨ Prototype for *Between* 156 ⟩ ≡

```
void Between(int n, double **R01, double **R10, double **T01, double **T10, double **R12, double
    **R21, double **T12, double **T21, double **Lup, double **Ldown)
```

This code is used in sections 140 and 157.

157. ⟨ Definition for *Between* 157 ⟩ ≡

```
⟨ Prototype for Between 156 ⟩
{
  (void) R01;
  (void) T10;
  (void) R21;
  (void) T12;
  (void) T21;
  ⟨ Allocate memory for a and b 161 ⟩
  Star_Multiply(n, R10, R12, a);
  Star_One_Minus(n, a);
  Right_Inverse_Multiply(n, a, T01, Ldown);
  Star_Multiply(n, R12, R10, a);
  Star_One_Minus(n, a);
  Right_Inverse_Multiply(n, a, R12, b);
  Star_Multiply(n, b, T01, Lup);
  ⟨ Free Memory for a and b 162 ⟩
}
```

This code is used in section 139.

158. Utility routines.

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

$$A \star B = \int_0^1 A(\mu,\mu')B(\mu',\mu'')\,2\mu d\mu$$

then

$$A \star B = \sum_j A^{ij} 2\mu_j w_j B^{jk}$$

where μ_j is the jth 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

$$\mathbf{c}_{ij} = 2\mu_i w_i \delta_{ij}$$

Thus a matrix star multiplication may be written

$$A \star B = A\,\mathbf{c}\,B$$

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* 159 ⟩ ≡

```
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 139.

160. This subtracts the matrix A from the unit matrix for star multiplication.

⟨ Definition for *Star_One_Minus* 160 ⟩ ≡

```
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 139.

161. ⟨ Allocate memory for a and b 161 ⟩ ≡

```
double **a, **b;
a = dmatrix(1, n, 1, n);
b = dmatrix(1, n, 1, n);
```

This code is used in sections 145, 147, 149, 151, 153, 155, and 157.

162.

⟨ Free Memory for a and b 162 ⟩ ≡

```
free_dmatrix(a, 1, n, 1, n);
free_dmatrix(b, 1, n, 1, n);
```

This code is used in sections 145, 147, 149, 151, 153, 155, and 157.

163. AD Boundary.
This section has routines associated with incorporating boundary conditions into the adding-doubling algorithm.

```
⟨ad_bound.c 163⟩ ≡
#include <math.h>
#include <stdio.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 172⟩;
  ⟨Prototype for B_Add_Slide 174⟩;
  ⟨Definition for Init_Boundary 167⟩
  ⟨Definition for Boundary_RT 170⟩
  ⟨Definition for Add_Top 178⟩
  ⟨Definition for Add_Bottom 180⟩
  ⟨Definition for A_Add_Slide 173⟩
  ⟨Definition for B_Add_Slide 175⟩
  ⟨Definition for Add_Slides 182⟩
  ⟨Definition for Sp_RT 184⟩
```

164. ⟨ad_bound.h 164⟩ ≡

```
  ⟨Preprocessor definitions⟩
  ⟨Prototype for Init_Boundary 166⟩;
  ⟨Prototype for Boundary_RT 169⟩;
  ⟨Prototype for Add_Top 177⟩;
  ⟨Prototype for Add_Bottom 179⟩;
  ⟨Prototype for Add_Slides 181⟩;
  ⟨Prototype for Sp_RT 183⟩;
```

165. Boundary Initialization.

166. *Init_Boundary* creates reflection and transmission matrices to simulate a boundary. If *boundary* ≡ `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` ≡ `T32` and `T10` ≡ `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

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

```
⟨ Prototype for Init_Boundary 166 ⟩ ≡
  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 164 and 167.

167.
```
⟨ Definition for Init_Boundary 167 ⟩ ≡
  ⟨ Prototype for Init_Boundary 166 ⟩
  {
    if (boundary ≡ 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 163.

168. *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 $(n_{\text{slab}}/n_{\text{outside}})^4$, 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 remembered is that 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.

169.
```
⟨ Prototype for Boundary_RT 169 ⟩ ≡
  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 164 and 170.

170. ⟨ Definition for *Boundary_RT* 170 ⟩ ≡

```
⟨ Prototype for Boundary_RT 169 ⟩
{
  int i;
  double refl, trans;
  double mu;

  for (i = 1; i ≤ n; i++) {
    if (n_i ≡ 1.0) mu = Cos_Snell(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 * twoaw[i];
    T[i] = trans;
  }
}
```

This code is used in section 163.

171. Boundary incorporation algorithms.

The next two routines *A_Add_Slide* and *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.

172. *A_Add_Slide* computes the resulting `R20` and `T02` matrices for a glass slide on top of an inhomogeneous layer characterized by `R12`, `R21`, `T12`, `T21`. It is ok if `R21` ≡ `R12` and `T12` ≡ `T21`. But I do not think that it is required by this routine. The result matrices `R20` and `T02` should be independent of the input matrices None of the input matrices are changed

The critical quantites 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 *A_Add_Slide* 172 ⟩ ≡

```
static void A_Add_Slide(int n, double **R12, double **R21, double **T12, double **T21,
      double *R10, double *T01, double **R20, double **T02,
      double **atemp, double **btemp)
```

This code is used in sections 163 and 173.

173. ⟨ Definition for *A_Add_Slide* 173 ⟩ ≡

```
⟨ Prototype for A_Add_Slide 172 ⟩
{
   double **ctemp;

   ctemp = R20;
   Left_Diagonal_Multiply(n, R10, R12, atemp);
   One_Minus(n, atemp);
   Left_Inverse_Multiply(n, atemp, T12, ctemp);
   Right_Diagonal_Multiply(n, ctemp, T01, T02);
   Right_Diagonal_Multiply(n, ctemp, R10, btemp);
   Matrix_Multiply(n, btemp, T21, atemp);
   Matrix_Sum(n, R21, atemp, R20);
}
```

This code is used in section 163.

174. *B_Add_Slide* computes the resulting `R02` and `T20` matrices for a glass slide on top of an inhomogeneous layer characterized by `R12`, `R21`, `T12`, `T21`. It is ok if `R21` ≡ `R12` and `T12` ≡ `T21`. But I do not think that it is required by this routine. The result matrices `R02` and `T20` 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 *B_Add_Slide* 174 ⟩ ≡

```
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 163 and 175.

175. ⟨ Definition for *B_Add_Slide* 175 ⟩ ≡

```
⟨ Prototype for B_Add_Slide 174 ⟩
{
  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]/twoaw[i]/twoaw[i];
}
```

This code is used in section 163.

176. Routines to incorporate slides.

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

n	size of matrix
R01, R10, T01, T10	R, T for slide assuming 0=air and 1=slab
R12, R21, T12, T21	R, T for slab assuming 1=slide and 2=?
R02, R20, T02, T20	calc R, T for both assuming 0=air and 2=?
atemp, *btemp*	previously allocated temporary storage matrices

⟨ Prototype for *Add_Top* 177 ⟩ ≡

```
void Add_Top(int n, double *R01, double *R10, double *T01, double *T10,
    double **R12, double **R21, double **T12, double **T21,
    double **R02, double **R20, double **T02, double **T20,
    double **atemp, double **btemp)
```

This code is used in sections 164 and 178.

178.

⟨ Definition for *Add_Top* 178 ⟩ ≡

```
⟨ Prototype for Add_Top 177 ⟩
{
  A_Add_Slide(n, R12, R21, T12, T21, R10, T01, R20, T02, atemp, btemp);
  B_Add_Slide(n, R12, T21, R01, R10, T01, T10, R02, T20, atemp, btemp);
}
```

This code is used in section 163.

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

n	size of matrix
R01, R10, T01, T10	R, T for slab assuming 0=slab top and 1=slab bottom
R12, R21, T12, T21	R, T for slide assuming 1=slab bottom and 2=slide bottom
R02, R20, T02, T20	calc R, T for both assuming 0=slab top and 2=slide bottom
atemp, *btemp*	previously allocated temporary storage matrices

⟨ Prototype for *Add_Bottom* 179 ⟩ ≡

```
void Add_Bottom(int n, double **R01, double **R10, double **T01, double **T10,
    double *R12, double *R21, double *T12, double *T21,
    double **R02, double **R20, double **T02, double **T20,
    double **atemp, double **btemp)
```

This code is used in sections 164 and 180.

180.

⟨ Definition for *Add_Bottom* 180 ⟩ ≡

```
⟨ Prototype for Add_Bottom 179 ⟩
{
  A_Add_Slide(n, R10, R01, T10, T01, R12, T21, R02, T20, atemp, btemp);
  B_Add_Slide(n, R10, T01, R21, R12, T21, T12, R20, T02, atemp, btemp);
}
```

This code is used in section 163.

181. 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 Tcannot 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: $R_{01} = R_{32}$, $R_{10} = R_{23}$, $T_{01} = T_{32}$, and $T_{10} = T_{23}$. 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 R_{30} and T_{03} 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* 181 ⟩ ≡
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 164 and 182.

182.

⟨ Definition for *Add_Slides* 182 ⟩ ≡

```
⟨ Prototype for Add_Slides 181 ⟩
{
    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 163.

183. Specular R and T.

Sp_RT calculates the specular reflection and transmission for light incident on a slide-slab-slide sandwich. The sample is characterized by the record *slab*. The total unscattered reflection and transmission for oblique irradiance (*urx* and *utx*) together with their companions *uru* and *utu* for diffuse irradiance. The cosine of the incident angle is specified by *slab.cos_angle*.

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 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.

Oh, yes. The mysterious multiplication by a factor of *n_slab* ∗ *n_slab* is required to account for the n^2-law of radiance.

⟨ Prototype for *Sp_RT* 183 ⟩ ≡

```
void Sp_RT(int n, struct AD_slab_type slab, double *urx, double *utx, double *uru, double *utu)
```

This code is used in sections 164 and 184.

184. ⟨ Definition for *Sp_RT* 184 ⟩ ≡

```
⟨ Prototype for Sp_RT 183 ⟩
{
  double mu_outside, r, t;
  int i;

  *uru = 0;
  *utu = 0;
  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;
    }
  }
  Sp_mu_RT(slab.n_top_slide, slab.n_slab, slab.n_bottom_slide, slab.b_top_slide, slab.b, slab.b_bottom_slide,
      slab.cos_angle, urx, utx);
  *uru *= slab.n_slab * slab.n_slab;
  *utu *= slab.n_slab * slab.n_slab;
}
```

This code is used in section 163.

185. 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.

⟨ ad_frsnl.c 185 ⟩ ≡
#include <math.h>
#include <float.h>
#include <stdio.h>
#include "ad_frsnl.h"
⟨ Prototype for *Fresnel* 192 ⟩;
⟨ Prototype for R1 206 ⟩;
⟨ Definition for *Cos_Critical_Angle* 188 ⟩
⟨ Definition for *Cos_Snell* 190 ⟩
⟨ Definition for *Fresnel* 193 ⟩
⟨ Definition for *Glass* 195 ⟩
⟨ Definition for *Absorbing_Glass_RT* 197 ⟩
⟨ Definition for R1 207 ⟩
⟨ Definition for *Sp_mu_RT* 202 ⟩
⟨ Definition for *Sp_mu_RT_Flip* 200 ⟩
⟨ Definition for *Diffuse_Glass_R* 209 ⟩

186. ⟨ ad_frsnl.h 186 ⟩ ≡
⟨ Prototype for *Cos_Critical_Angle* 187 ⟩;
⟨ Prototype for *Cos_Snell* 189 ⟩;
⟨ Prototype for *Absorbing_Glass_RT* 196 ⟩;
⟨ Prototype for *Sp_mu_RT* 201 ⟩;
⟨ Prototype for *Sp_mu_RT_Flip* 199 ⟩;
⟨ Prototype for *Diffuse_Glass_R* 208 ⟩;
⟨ Prototype for *Glass* 194 ⟩;

187. The critical angle.
Cos_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 trigonometric 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

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

or more simply

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

where $n = n_t/n_i$.

⟨ Prototype for *Cos_Critical_Angle* 187 ⟩ ≡
```
double Cos_Critical_Angle(double ni, double nt)
```
This code is used in sections 186 and 188.

188. ⟨ Definition for *Cos_Critical_Angle* 188 ⟩ ≡
```
⟨ Prototype for Cos_Critical_Angle 187 ⟩
{
  double x;
  if (nt ≥ ni) return 0.0;
  else {
    x = nt/ni;
    x = sqrt(1.0 − x ∗ x);
    return x;
  }
}
```
This code is used in section 185.

189. Snell's Law.

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 μ_t, the transmitted cosine as μ_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 μ_t yields

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

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 *sqrt* function is non-negative. If it is, then I return $\mu_t = 0$ i.e., $\theta_t = 90°$.

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

⟨ Prototype for *Cos_Snell* 189 ⟩ ≡

double *Cos_Snell*(**double** *n_i*, **double** *mu_i*, **double** *n_t*)

This code is used in sections 186 and 190.

190. ⟨ Definition for *Cos_Snell* 190 ⟩ ≡

```
⟨ Prototype for Cos_Snell 189 ⟩
{
  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));
}
```

This code is used in section 185.

191. Fresnel Reflection.

Fresnel calculates the specular reflection for light incident at an angle θ_i from the normal (having a cosine equal to μ_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 θ_i and θ_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 θ_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

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

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

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

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.

192. ⟨ Prototype for *Fresnel* 192 ⟩ ≡

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

This code is used in sections 185 and 193.

193. ⟨ Definition for *Fresnel* 193 ⟩ ≡

```
⟨ Prototype for Fresnel 192 ⟩
{
  double mu_t, ratio, temp, temp1;
  if (mu_i ≡ 0.0) return 1.0;
  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);
  }
  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 185.

194. 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_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.

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

$$r_g = \frac{r_1 + r_2 - 2r_1r_2}{1 - r_1r_2}$$

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 the first boundary is not totally reflecting but the second one is. Note that μ_g gets calculated twice (once in the first call to *Fresnel* and once directly).

⟨ Prototype for *Glass* 194 ⟩ ≡

```
double Glass(double n_i, double n_g, double n_t, double mu_i)
```

This code is used in sections 186 and 195.

195. ⟨ Definition for *Glass* 195 ⟩ ≡

```
⟨ Prototype for Glass 194 ⟩
{
  double r1, r2, mu_g, temp;
  if (n_i ≡ n_g) return (Fresnel(n_g, n_t, mu_i));
  r1 = Fresnel(n_i, n_g, mu_i);
  if (r1 ≥ 1.0 ∨ n_g ≡ 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 185.

196. 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

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

Here r_1 is the reflection at the sample-glass interface and r_2 is the reflection at the glass-air interface and μ_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

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

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* 196 ⟩ ≡

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 186 and 197.

197. ⟨ Definition for *Absorbing_Glass_RT* 197 ⟩ ≡

```
⟨ Prototype for Absorbing_Glass_RT 196 ⟩
{
  double r1, r2, mu_g, expo, denom;
  *t = 0;
  *r = Fresnel(n_i, n_g, mu_i);
  if (*r ≥ 1.0 ∨ b ≡ HUGE_VAL ∨ mu_i ≡ 0.0) return;
  mu_g = Cos_Snell(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 ≤ DBL_MIN_10_EXP * 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 185.

198. Unscattered refl and trans for a sample.

199. *Sp_mu_RT_Flip* finds the reflectance to incorporate flipping of the sample. This is needed when the sample is flipped between measurements.

⟨ Prototype for *Sp_mu_RT_Flip* 199 ⟩ ≡
void *Sp_mu_RT_Flip*(**int** *flip*, **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 186 and 200.

200. ⟨ Definition for *Sp_mu_RT_Flip* 200 ⟩ ≡

```
⟨ Prototype for Sp_mu_RT_Flip 199 ⟩
{
  Sp_mu_RT(n_top, n_slab, n_bottom, tau_top, tau_slab, tau_bottom, mu, r, t);
  if (flip ∧ n_top ≠ n_bottom ∧ tau_top ≠ tau_bottom) {
    double correct_r = *r;
    Sp_mu_RT(n_bottom, n_slab, n_top, tau_bottom, tau_slab, tau_top, mu, r, t);
    *r = correct_r;
  }
}
```

This code is used in section 185.

201. *Sp_mu_RT* calculates the unscattered reflection and transmission (i.e., specular) through a glass-slab-glass sandwich. Light is incident at an angle having a cosine *mu* from air onto a possibly absorbing glass plate with index *n_top* on a sample with index *n_slab* resting on another possibly absorbing glass plate with index *n_bottom* and then exiting into air again.

The optical thickness of the slab is *tau_slab*.

⟨ Prototype for *Sp_mu_RT* 201 ⟩ ≡
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 186 and 202.

202. ⟨ Definition for *Sp_mu_RT* 202 ⟩ ≡

```
⟨ Prototype for Sp_mu_RT 201 ⟩
{
  double r_top, r_bottom, t_top, t_bottom, mu_slab, beer, denom, temp, mu_in_slab;
  *r = 0;
  *t = 0;
  Absorbing_Glass_RT(1.0, n_top, n_slab, mu, tau_top, &r_top, &t_top);
  mu_in_slab = Cos_Snell(1.0, mu, n_slab);
  Absorbing_Glass_RT(n_slab, n_bottom, 1.0, mu_in_slab, tau_bottom, &r_bottom, &t_bottom);
  ⟨ Calculate beer 204 ⟩
  ⟨ Calculate r and t 205 ⟩
}
```

This code is used in section 185.

203. 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.

204. 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* 204 ⟩ ≡

```
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 202.

205. If $r_{\rm top}$ is the reflection for the top and $r_{\rm bottom}$ is that for the bottom surface then the total reflection will be

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

and the transmission is

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

where μ is the angle inside the slab and τ 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_top* < 1 and that *r_bottom* and *beer* are less than or equal to one.

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

⟨ Calculate r and t 205 ⟩ ≡

```
if (beer ≡ 0.0) {
  ∗r = r_top;
}
else {
  temp = t_top ∗ beer;
  denom = 1 − r_top ∗ r_bottom ∗ beer ∗ beer;
  ∗r = r_top + r_bottom ∗ temp ∗ temp/denom;
  ∗t = t_bottom ∗ temp/denom;
}
```

This code is used in section 202.

206. Total diffuse reflection.

R1 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]

$$R_1 = \frac{1}{2} + \frac{(m-1)(3m+1)}{6(m+1)^2} + \left[\frac{m^2(m^2-1)^2}{(m^2+1)^3}\right]\log\left(\frac{m-1}{m+1}\right) - \frac{2m^3(m^2+2m-1)}{(m^2+1)(m^4-1)} + \left[\frac{8m^4(m^4+1)}{(m^2+1)(m^4-1)^2}\right]\log m$$

where Walsh's parameter $m = n_t/n_i$. 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 R1 206 ⟩ ≡
  static double R1(**double** *ni*, **double** *nt*)

This code is used in sections 185 and 207.

207. ⟨ Definition for R1 207 ⟩ ≡

```
⟨ Prototype for R1 206 ⟩
{
  double m, m2, m4, mm1, mp1, r, temp;
  if (ni ≡ nt) return 0.0;
  if (ni < nt) m = nt/ni;
  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);
  if (ni < nt) return r;
  else return (1 − (1 − r)/m2);
}
```

This code is used in section 185.

208. Diffusion reflection from a glass slide.

Diffuse_Glass_R returns the total diffuse specular reflection from the air-glass-tissue interface

⟨ Prototype for *Diffuse_Glass_R* 208 ⟩ ≡

```
double Diffuse_Glass_R(double nair, double nslide, double nslab)
```

This code is used in sections 186 and 209.

209. ⟨ Definition for *Diffuse_Glass_R* 209 ⟩ ≡

```
⟨ Prototype for Diffuse_Glass_R 208 ⟩
{
  double rairglass, rglasstissue, rtemp;
  rairglass = R1(nair, nslide);
  rglasstissue = R1(nslide, nslab);
  rtemp = rairglass * rglasstissue;
  if (rtemp ≥ 1) return 1.0;
  else return ((rairglass + rglasstissue − 2 * rtemp)/(1 − rtemp));
}
```

This code is used in section 185.

210. 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 outline 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.

⟨ ad_matrx.c 210 ⟩ ≡
#include <stddef.h>
#include <math.h>
#include "ad_globl.h"
#include "ad_matrx.h"
#include "nr_util.h"
⟨ Definition for *Copy_Matrix* 214 ⟩
⟨ Definition for *One_Minus* 216 ⟩
⟨ Definition for *Transpose_Matrix* 218 ⟩
⟨ Definition for *Diagonal_To_Matrix* 220 ⟩
⟨ Definition for *Right_Diagonal_Multiply* 222 ⟩
⟨ Definition for *Left_Diagonal_Multiply* 224 ⟩
⟨ Definition for *Matrix_Multiply* 231 ⟩
⟨ Definition for *Matrix_Sum* 226 ⟩
⟨ Definition for *Solve* 252 ⟩
⟨ Definition for *Decomp* 242 ⟩
⟨ Definition for *Matrix_Inverse* 256 ⟩
⟨ Definition for *Left_Inverse_Multiply* 258 ⟩
⟨ Definition for *Right_Inverse_Multiply* 260 ⟩

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

⟨ ad_matrx.h 211 ⟩ ≡
⟨ Prototype for *Copy_Matrix* 213 ⟩;
⟨ Prototype for *One_Minus* 215 ⟩;
⟨ Prototype for *Transpose_Matrix* 217 ⟩;
⟨ Prototype for *Diagonal_To_Matrix* 219 ⟩;
⟨ Prototype for *Right_Diagonal_Multiply* 221 ⟩;
⟨ Prototype for *Left_Diagonal_Multiply* 223 ⟩;
⟨ Prototype for *Matrix_Multiply* 230 ⟩;
⟨ Prototype for *Matrix_Sum* 225 ⟩;
⟨ Prototype for *Solve* 251 ⟩;
⟨ Prototype for *Decomp* 241 ⟩;
⟨ Prototype for *Matrix_Inverse* 255 ⟩;
⟨ Prototype for *Left_Inverse_Multiply* 257 ⟩;
⟨ Prototype for *Right_Inverse_Multiply* 259 ⟩;

212. Simple Matrix Routines.

213. *Copy_Matrix* replaces the matrix B by A

⟨ Prototype for *Copy_Matrix* 213 ⟩ ≡
void *Copy_Matrix*(**int** n, **double** $**A$, **double** $**B$)

This code is used in sections 211 and 214.

214. ⟨ Definition for *Copy_Matrix* 214 ⟩ ≡

```
⟨ Prototype for Copy_Matrix 213 ⟩
{
  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 210.

215. *One_Minus* replaces the matrix A by 1-A

⟨ Prototype for *One_Minus* 215 ⟩ ≡
void *One_Minus*(**int** n, **double** $**A$)

This code is used in sections 211 and 216.

216. ⟨ Definition for *One_Minus* 216 ⟩ ≡

```
⟨ Prototype for One_Minus 215 ⟩
{
  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 210.

217. *Transpose_Matrix* transposes a matrix.

⟨ Prototype for *Transpose_Matrix* 217 ⟩ ≡
void *Transpose_Matrix*(**int** n, **double** $**a$)

This code is used in sections 211 and 218.

218. ⟨ Definition for *Transpose_Matrix* 218 ⟩ ≡
⟨ Prototype for *Transpose_Matrix* 217 ⟩
{
  int i, j;
  double *swap*;
  for $(i = 1;\ i \le n;\ i{+}{+})$ {
    for $(j = i + 1;\ j \le n;\ j{+}{+})$ {
      $swap = a[i][j]$;
      $a[i][j] = a[j][i]$;
      $a[j][i] = swap$;
    }
  }
}

This code is used in section 210.

219. *Diagonal_To_Matrix* converts a diagonal array to a matrix

⟨ Prototype for *Diagonal_To_Matrix* 219 ⟩ ≡
  void *Diagonal_To_Matrix*(**int** n, **double** $*Diag$, **double** $**Mat$)

This code is used in sections 211 and 220.

220. ⟨ Definition for *Diagonal_To_Matrix* 220 ⟩ ≡
⟨ Prototype for *Diagonal_To_Matrix* 219 ⟩
{
  int i, j;
  for $(i = 1;\ i \le n;\ i{+}{+})$ {
    for $(j = 1;\ j \le n;\ j{+}{+})$ $Mat[i][j] = 0.0$;
    $Mat[i][i] = Diag[i]$;
  }
}

This code is used in section 210.

221. *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* 221 ⟩ ≡
  void *Right_Diagonal_Multiply*(**int** n, **double** $**A$, **double** $*B$, **double** $**C$)

This code is used in sections 211 and 222.

222. ⟨ Definition for *Right_Diagonal_Multiply* 222 ⟩ ≡
⟨ Prototype for *Right_Diagonal_Multiply* 221 ⟩
{
  int i, j;
  for $(i = 1;\ i \le n;\ i{+}{+})$
    for $(j = 1;\ j \le n;\ j{+}{+})$ $C[i][j] = A[i][j] * B[j]$;
}

This code is used in section 210.

223. *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

⟨ Prototype for *Left_Diagonal_Multiply* 223 ⟩ ≡

void *Left_Diagonal_Multiply*(**int** n, **double** $*A$, **double** $**B$, **double** $**C$)

This code is used in sections 211 and 224.

224. ⟨ Definition for *Left_Diagonal_Multiply* 224 ⟩ ≡

```
⟨ Prototype for Left_Diagonal_Multiply 223 ⟩
{
  int i, j;
  for (i = 1; i ≤ n; i++)
    for (j = 1; j ≤ n; j++) C[i][j] = A[i] * B[i][j];
}
```

This code is used in section 210.

225. *Matrix_Sum* adds the two matrices A and B, puts the result in C The matrices need not be distinct

⟨ Prototype for *Matrix_Sum* 225 ⟩ ≡

void *Matrix_Sum*(**int** n, **double** $**A$, **double** $**B$, **double** $**C$)

This code is used in sections 211 and 226.

226. ⟨ Definition for *Matrix_Sum* 226 ⟩ ≡

```
⟨ Prototype for Matrix_Sum 225 ⟩
{
  int i, j;
  for (i = 1; i ≤ n; i++)
    for (j = 1; j ≤ n; j++) C[i][j] = A[i][j] + B[i][j];
}
```

This code is used in section 210.

227. 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. There is no inherent reason that A, B, and C must all be $n \times n$ matrices. 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:

⟨unused fragment one 227⟩ ≡

```
for (i = 1; i ≤ n; i++) {
  for (j = 1; j ≤ n; j++) {
    C[i][j] = 0.0;
    for (k = 1; k ≤ n; k++) C[i][j] += A[i][k] * B[k][j];
  }
}
```

228. 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,

⟨unused fragment two 228⟩ ≡

```
for (i = 1; i ≤ n; i++)
  for (j = 1; j ≤ n; j++) C[i][j] = 0.0;
for (i = 1; i ≤ n; i++) {
  for (k = 1; k ≤ n; k++) {
    for (j = 1; j ≤ n; j++) C[i][j] += A[i][k] * B[k][j];
  }
}
```

229. 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.

Explictly 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.

230. ⟨Prototype for *Matrix_Multiply* 230⟩ ≡

```
void Matrix_Multiply(int n, double **A, double **B, double **C)
```

This code is used in sections 211 and 231.

231. ⟨ Definition for *Matrix_Multiply* 231 ⟩ ≡

```
⟨ Prototype for Matrix_Multiply 230 ⟩
{
  ⟨ Local variables for Matrix_Multiply 232 ⟩
  ⟨ Do awkward cases 233 ⟩
  ⟨ Allocate memory for D 234 ⟩
  ⟨ Initialization for Matrix_Multiply 235 ⟩
  ⟨ Multiplying A and B 238 ⟩
  ⟨ Free memory for D 239 ⟩
}
```

This code is used in section 210.

232. ⟨ Local variables for *Matrix_Multiply* 232 ⟩ ≡

```
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 231.

233. ⟨ Do awkward cases 233 ⟩ ≡

```
if (n ≤ 0) {
  AD_error("Non-positive dimension passed to Matrix_Multiply");
}
else if (n ≡ 1) {
  C[1][1] = A[1][1] * B[1][1];
  return;
}
```

This code is used in section 231.

234. 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 234 ⟩ ≡

```
D = dvector(1, n);
```

This code is used in section 231.

235. 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 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.

⟨ Initialization for *Matrix_Multiply* 235 ⟩ ≡

```
a_start = &A[1][1];
b_last = &B[n][1];
row = &A[2][1] − a_start;
c_very_last = &C[n][n];
d_start = &D[1];
d_last = &D[n];
```

This code is used in section 231.

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

$\langle$ Zero D 236 $\rangle \equiv$
$d\_ptr = d\_start;$
while $(d\_ptr \le d\_last)$ $*d\_ptr{+}{+} = 0.0;$

This code is used in section 238.

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

$\langle$ Copy D into C 237 $\rangle \equiv$
$d\_ptr = d\_start;$
$c\_ptr = c\_start;$
while $(d\_ptr \le d\_last)$ $*c\_ptr{+}{+} = *d\_ptr{+}{+};$

This code is used in section 238.

238. 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 $t = *a\_ptr$) by every element in row i and adding it to the appropriate element in row i of C.

$\langle$ Multiplying A and B 238 $\rangle \equiv$
for $(c\_start = \&C[1][1];\ c\_start \le c\_very\_last;\ c\_start \mathrel{+}= row)$ {
  $a\_ptr = a\_start;$
  $\langle$ Zero D 236 $\rangle$
  for $(b\_start = \&B[1][1];\ b\_start \le b\_last;\ b\_start \mathrel{+}= row)$ {
    $t = *a\_ptr{+}{+};$
    $b\_ptr = b\_start;$
    $d\_ptr = d\_start;$
    while $(d\_ptr \le d\_last)$ $*d\_ptr{+}{+} \mathrel{+}= t * (*b\_ptr{+}{+});$
  }
  $\langle$ Copy D into C 237 $\rangle$
  $a\_start \mathrel{+}= row;$
}

This code is used in section 231.

239. Dump the memory that was allocated.

$\langle$ Free memory for D 239 $\rangle \equiv$
free_dvector$(D, 1, n);$

This code is used in section 231.

240. Matrix Decomposition.

241. ⟨ Prototype for *Decomp* 241 ⟩ ≡
```
void Decomp(int n, double **A, double *condition, int *ipvt)
```
This code is used in sections 211 and 242.

242. *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 $\mathbf{I} - \mathbf{L}$ so that (permutation matrix)*A=L*U. *condition* is an estimate of the condition of A. For the linear system $\mathbf{AX} = \mathbf{B}$, 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 $1.0 \cdot 10^{32}$ if exact singularity is detected. *ipvt* is the pivot vector $ipvt(k)$ is the index of the kth pivot row $ipvt(n) = (-1)^{(\mathrm{numberofinterchanges})}$

⟨ Definition for *Decomp* 242 ⟩ ≡
```
⟨ Prototype for Decomp 241 ⟩
{
  double t, anorm;
  int i, j, k, m;
  ⟨ Do n ≡ 1 case 243 ⟩
  ⟨ Compute 1-norm of A 244 ⟩
  ⟨ Gaussian elimination with partial pivoting 245 ⟩
  ⟨ Check for singularity 249 ⟩
}
```
This code is used in section 210.

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

⟨ Do $n \equiv 1$ case 243 ⟩ ≡
```
ipvt[n] = 1;
if (n ≡ 1) {
  if (A[1][1] ≡ 0) {
    AD_error("1␣X␣1␣Matrix␣is␣Singular␣---␣i.e.␣zero");
    return;
  }
}
```
This code is used in section 242.

244. ⟨ Compute 1-norm of A 244 ⟩ ≡
```
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 242.

245. ⟨ Gaussian elimination with partial pivoting 245 ⟩ ≡

```
for (k = 1; k < n; k++) {
  ⟨ Find pivot 246 ⟩
  ⟨ Compute multipliers 247 ⟩
  ⟨ Interchange and eliminate by columns 248 ⟩
}
```

This code is used in section 242.

246. ⟨ Find pivot 246 ⟩ ≡

```
m = k;
for (i = k + 1; i ≤ n; i++)
  if (fabs(A[i][k]) > fabs(A[m][k])) m = i;
ipvt[k] = m;
if (m ≠ k) ipvt[n] *= −1;
t = A[m][k];
A[m][k] = A[k][k];
A[k][k] = t;    /* skip step if pivot is zero */
if (t ≡ 0) continue;
```

This code is used in section 245.

247. ⟨ Compute multipliers 247 ⟩ ≡

```
for (i = k + 1; i ≤ n; i++) A[i][k] /= −t;
```

This code is used in section 245.

248. ⟨ Interchange and eliminate by columns 248 ⟩ ≡

```
for (j = k + 1; j ≤ n; j++) {
  t = A[m][j];
  A[m][j] = A[k][j];
  A[k][j] = t;
  if (t ≡ 0) continue;
  for (i = k + 1; i ≤ n; i++) A[i][j] += A[i][k] * t;
}
```

This code is used in section 245.

249. ⟨ Check for singularity 249 ⟩ ≡

```
*condition = 1.0;
for (k = 1; k ≤ n; k++) {
  if (A[k][k] ≡ 0.0) {
    *condition = 1 · 10^32;
    return;
  }
}
```

This code is used in section 242.

250. Solving systems of equations.

251.

⟨ Prototype for *Solve* 251 ⟩ ≡
  void *Solve*(**int** n, **double** $**A$, **double** $*B$, **int** $*ipvt$)

This code is used in sections 211 and 252.

252. 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 *ipvt* is the pivot vector obtained from *Decomp*

On output B is the solution vector X.

```
⟨ Definition for Solve 252 ⟩ ≡
  ⟨ Prototype for Solve 251 ⟩
  {
    int i, k, m;
    double t;
    ⟨ Forward elimination 253 ⟩
    ⟨ Back substitution 254 ⟩
  }
```

This code is used in section 210.

253. ⟨ Forward elimination 253 ⟩ ≡

```
  for (k = 1; k < n; k++) {
    m = ipvt[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 252.

254. ⟨ Back substitution 254 ⟩ ≡

```
  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 252.

255. Finds the inverse of the matrix A (of order n) and stores the answer in *Ainv*.

⟨ Prototype for *Matrix_Inverse* 255 ⟩ ≡
  void *Matrix_Inverse*(**int** n, **double** $**A$, **double** $**Ainv$)

This code is used in sections 211 and 256.

256. ⟨ Definition for *Matrix_Inverse* 256 ⟩ ≡

```
⟨ Prototype for Matrix_Inverse 255 ⟩
{
  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);
    AD_error("Singular␣Matrix␣...␣failed␣in␣Inverse_Multiply\n");
  }
  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 210.

257. ⟨ Prototype for *Left_Inverse_Multiply* 257 ⟩ ≡

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

This code is used in sections 211 and 258.

258. *Left_Inverse_Multiply* computes $\mathbf{A} = \mathbf{C} \cdot \mathbf{D}^{-1}$ 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.

⟨ Definition for *Left_Inverse_Multiply* 258 ⟩ ≡

```
⟨ Prototype for Left_Inverse_Multiply 257 ⟩
{
  int *ipvt;
  int i, j;
  double *work;
  double condition;

  Transpose_Matrix(n, D);
  ipvt = ivector(1, n);
  Decomp(n, D, &condition, ipvt);    /* Check for singular result */
  if (condition ≡ (condition + 1) ∨ condition ≡ 1 · 10^32) {
    free_ivector(ipvt, 1, n);
    AD_error("Singular␣Matrix␣...␣failed␣in␣Left_Inverse_Multiply\n");
  }
  work = dvector(1, n);
  for (i = 1; i ≤ n; i++) {    /* Cycle through all the row in C */
    for (j = 1; j ≤ n; j++)    /* put a row of C into work */
      work[j] = C[i][j];    /* and avoid a Transpose Matrix */
    Solve(n, D, work, ipvt);
    for (j = 1; j ≤ n; j++)    /* 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 210.

259. ⟨ Prototype for *Right_Inverse_Multiply* 259 ⟩ ≡

```
void Right_Inverse_Multiply(int n, double **D, double **C, double **A)
```

This code is used in sections 211 and 260.

260. *Right_Inverse_Multiply* computes $\mathbf{A} = \mathbf{D}^{-1} \cdot \mathbf{C}$ 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.

⟨Definition for *Right_Inverse_Multiply* 260⟩ ≡

```
⟨Prototype for Right_Inverse_Multiply 259⟩
{
  int *ipvt;
  int i, j;
  double *work;
  double condition;

  ipvt = ivector(1, n);
  Decomp(n, D, &condition, ipvt);      /* Check for singular result */
  if (condition == (condition + 1) || condition == 1 · 10^32) {
    free_ivector(ipvt, 1, n);
    AD_error("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 210.

261. AD Radau Quadrature.
This global variable is needed because the degree of the Legendre Polynomial must be known. The routine *Radau* stores the correct value in this.

#define NSLICES 512
#define EPS $1 \cdot 10^{-16}$

⟨ ad_radau.c 261 ⟩ ≡
⟨ Preprocessor definitions ⟩
#include "ad_globl.h"
#include "ad_radau.h"
#include "nr_rtsaf.h"
#include "nr_util.h"
#include "nr_zbrak.h"
static int *local_n_size*;
⟨ Prototype for *Pn_and_Pnm1* 272 ⟩;
⟨ Prototype for *Pnd* 274 ⟩;
⟨ Prototype for *phi* 280 ⟩;
⟨ Prototype for *phi_and_phiprime* 276 ⟩;
⟨ Definition for *Pn_and_Pnm1* 273 ⟩
⟨ Definition for *Pnd* 275 ⟩
⟨ Definition for *phi* 281 ⟩
⟨ Definition for *phi_and_phiprime* 277 ⟩
⟨ Definition for *Radau* 265 ⟩

262. ⟨ ad_radau.h 262 ⟩ ≡
⟨ Prototype for *Radau* 264 ⟩;

263. 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 $\frac{\pi}{2}$ 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 ν_c for a boundary layer with total internal reflection, then the effective range of integration is reduced to ν_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 ν less than ν_c, the integration range is reduced. The calculations in this paper used Gaussian quadrature for the range from 0 to ν_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 ν_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 nth 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 nth integration angle ν_n corresponds with $x_n = -1$ (normal incidence).

264. Radau. *Radau* calculates the n quadrature points x_i and weights w_i.

⟨ Prototype for *Radau* 264 ⟩ ≡

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

This code is used in sections 262 and 265.

265. ⟨ Definition for *Radau* 265 ⟩ ≡

```
⟨ Prototype for Radau 264 ⟩
{
  x[n] = −1.0;
  w[n] = 2.0/(n ∗ n);
  switch (n) {
  case 2: ⟨ Values for n ≡ 2 283 ⟩
  case 4: ⟨ Values for n ≡ 4 284 ⟩
  case 8: ⟨ Values for n ≡ 8 285 ⟩
  case 16: ⟨ Values for n ≡ 16 286 ⟩
  default: ⟨ Values for arbitrary n 267 ⟩
  }
  ⟨ Scale values 266 ⟩
}
```

This code is used in section 261.

266. 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 ν_c to 1 the following relations are needed to find the necessary integration angles ν_i and weights w_i

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

and

$$w_i = \frac{1-\nu_c}{(1-x_i)\sqrt{P'_{n-1}(x_i)}}$$

⟨ Scale values 266 ⟩ ≡

```
{
  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 265.

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

⟨ Values for arbitrary n 267 ⟩ ≡
```
{
  int i, nb, ndiv;
  double z;
  double *xb1, *xb2;
  ⟨ Allocate memory for Radau 268 ⟩
  ⟨ Bracket roots 269 ⟩
  ⟨ Find roots and weights 270 ⟩
  ⟨ Free memory for Radau 271 ⟩
  break;
}
```
This code is used in section 265.

268. ⟨ Allocate memory for Radau 268 ⟩ ≡
```
xb1 = dvector(1, NSLICES);
xb2 = dvector(1, NSLICES);
```
This code is used in section 267.

269. Bracket $n - 1$ roots, double *ndiv* if not enough roots are found.

⟨ Bracket roots 269 ⟩ ≡
```
local_n_size = n;
if (2 * n > NSLICES) ndiv = 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) AD_error("Cannot␣find␣enough␣roots␣for␣Radau␣quadrature");
```
This code is used in section 267.

270. Find the roots with an accuracy 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 270 ⟩ ≡
```
for (i = 1; i < n; i++) {
  double tmp;
  z = rtsafe(phi_and_phiprime, xb1[i], xb2[i], EPS);
  x[n - i] = z;
  tmp = Pnd(n - 1, z);
  w[n - i] = 1/((1 - z) * tmp * tmp);
}
```
This code is used in section 267.

271. ⟨ Free memory for Radau 271 ⟩ ≡
```
free_dvector(xb1, 1, NSLICES);
free_dvector(xb2, 1, NSLICES);
```
This code is used in section 267.

272. *Pn_and_Pnm1* returns $P_n(x)$ and $P_{n-1}(x)$

⟨ Prototype for *Pn_and_Pnm1* 272 ⟩ ≡

```
static void Pn_and_Pnm1(int n, double x, double *Pnm1, double *Pn)
```

This code is used in sections 261 and 273.

273. ⟨ Definition for *Pn_and_Pnm1* 273 ⟩ ≡

```
⟨ Prototype for Pn_and_Pnm1 272 ⟩
{
  int k;
  double Pk, Pkp1;
  double Pkm1 = 1.0;
  *Pnm1 = 1.0;
  *Pn = 1.0;
  if (x ≥ 1.0) return;
  if (x ≤ −1.0) x = −1;
  Pk = x;
  for (k = 1; k < n; k++) {
    Pkp1 = ((2 * k + 1) * x * Pk − k * Pkm1)/(k + 1);
    Pkm1 = Pk;
    Pk = Pkp1;
  }
  *Pnm1 = Pkm1;
  *Pn = Pk;
}
```

This code is used in section 261.

274. 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," *Math Comp*, **17**, 237-244 (1963).

⟨ Prototype for *Pnd* 274 ⟩ ≡

```
static double Pnd(int n, double x)
```

This code is used in sections 261 and 275.

275. ⟨ Definition for *Pnd* 275 ⟩ ≡

```
⟨ Prototype for Pnd 274 ⟩
{
  double p, pminus, pplus;
  int i;
  if (x > 1.0) {
    x = 1;
  }
  else if (x < −1.0) {
    x = −1;
  }
  pminus = 0;
  p = 1;
  if (n ≤ 0) return pminus;
  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 261.

276. 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

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

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_and_phiprime* 276 ⟩ ≡
  static void *phi_and_phiprime*(**double** x, **double** ∗*phi*, **double** ∗*phiprime*)

This code is used in sections 261 and 277.

277. ⟨ Definition for *phi_and_phiprime* 277 ⟩ ≡

```
⟨ Prototype for phi_and_phiprime 276 ⟩
{
  double Pn, Pnm1;
  int n;
  n = local_n_size;
  if (x ≥ 1.0) {
    ⟨ Phi and phiprime at x = 1 278 ⟩
  }
  else if (x ≤ −1.0) {
    ⟨ Phi and phiprime at x = −1 279 ⟩
  }
  else {
    Pn_and_Pnm1(n, x, &Pnm1, &Pn);
    *phi = (Pn + Pnm1)/(1 + x);
    *phiprime = ((n * x − 1 + x + n) * Pnm1 + (−n * x + x − n − 1) * Pn)/(1 + x)/(1 + x)/(1 − x);
  }
}
```

This code is used in section 261.

278. 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

$$\phi_{n-1}'(1) = \frac{n^2 - 1}{2}$$

⟨ Phi and phiprime at $x = 1$ 278 ⟩ ≡

```
{
  *phi = 1;
  *phiprime = (n * n - 1)/2;
}
```

This code is used in section 277.

279. 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

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

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

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

Now we just substitute to get

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

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

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

⟨ Phi and phiprime at $x = -1$ 279 ⟩ ≡

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

This code is used in section 277.

280. 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 *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," *Math Comp*, **17**, 237-244 (1963).

⟨ Prototype for *phi* 280 ⟩ ≡

```
static double phi(double x)
```

This code is used in sections 261 and 281.

281. ⟨ Definition for *phi* 281 ⟩ ≡

```
⟨ Prototype for phi 280 ⟩
{
  double Pn, Pnm1;
  if (x ≤ −1.0) {
    if (local_n_size % 2 ≠ 1) return (−local_n_size);
    else return (local_n_size);
  }
  Pn_and_Pnm1(local_n_size, x, &Pnm1, &Pn);
  return ((Pn + Pnm1)/(1 + x));
}
```

This code is used in section 261.

282. Radau Tables.
Here is a selection of commonly used number of quadrature points.

283. ⟨ Values for $n \equiv 2$ 283 ⟩ ≡
$x[1] = 0.3333333333333334;$
$w[1] = 1.5000000000000000;$
break;

This code is used in section 265.

284. ⟨ Values for $n \equiv 4$ 284 ⟩ ≡
$x[3] = -0.5753189235216942;$
$x[2] = 0.1810662711185306;$
$x[1] = 0.8228240809745921;$
$w[3] = 0.6576886399601182;$
$w[2] = 0.7763869376863437;$
$w[1] = 0.4409244223535367;$
break;

This code is used in section 265.

285. ⟨ Values for $n \equiv 8$ 285 ⟩ ≡
$x[7] = -0.8874748789261557;$
$x[6] = -0.6395186165262152;$
$x[5] = -0.2947505657736607;$
$x[4] = 0.0943072526611108;$
$x[3] = 0.4684203544308211;$
$x[2] = 0.7706418936781916;$
$x[1] = 0.9550412271225750;$
$w[7] = 0.1853581548029793;$
$w[6] = 0.3041306206467856;$
$w[5] = 0.3765175453891186;$
$w[4] = 0.3915721674524935;$
$w[3] = 0.3470147956345014;$
$w[2] = 0.2496479013298649;$
$w[1] = 0.1145088147442572;$
break;

This code is used in section 265.

286. $\langle$ Values for $n \equiv 16$ 286 $\rangle \equiv$

$x[15] = -0.9714610905263484;$
$x[14] = -0.9054008198116666;$
$x[13] = -0.8045734013587561;$
$x[12] = -0.6728619212112202;$
$x[11] = -0.5153294780626855;$
$x[10] = -0.3380303900599197;$
$x[9] = -0.1477783218133717;$
$x[8] = 0.0481153830735303;$
$x[7] = 0.2421226227060438;$
$x[6] = 0.4267878274849459;$
$x[5] = 0.5950144898997919;$
$x[4] = 0.7403379488928179;$
$x[3] = 0.8571740937696823;$
$x[2] = 0.9410354027041150;$
$x[1] = 0.9887186220549766;$
$w[15] = 0.0477022269476863;$
$w[14] = 0.0839852814449645;$
$w[13] = 0.1170203531038591;$
$w[12] = 0.1455555452202026;$
$w[11] = 0.1684963978499219;$
$w[10] = 0.1849617814886653;$
$w[10] = 0.1849617814886653;$
$w[9] = 0.1943190897115679;$
$w[8] = 0.1962087882390318;$
$w[7] = 0.1905582942553547;$
$w[6] = 0.1775847927527395;$
$w[5] = 0.1577869218042020;$
$w[4] = 0.1319256999330681;$
$w[3] = 0.1009956796217840;$
$w[2] = 0.0661895086101364;$
$w[1] = 0.0288971390168143;$
break;

This code is used in section 265.

287. 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.

```
⟨ ad_phase.c 287 ⟩ ≡
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "nr_util.h"
#include "ad_globl.h"
#include "ad_phase.h"
  ⟨ Definition for Get_Phi 293 ⟩
```

288. ⟨ ad_phase.h 288 ⟩ ≡

```
  ⟨ Prototype for Get_Phi 292 ⟩;
```

289. 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 ν is the dot product of the unit vectors $\hat{\mathbf{s}}_i$ and $\hat{\mathbf{s}}_j$

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

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

The redistribution function $\mathbf{h}_{ij}$ determines the fraction of light scattered from an incidence cone with angle ν_i into a cone with angle ν_j. The redistribution function is calculated by averaging the phase function over all possible azimuthal angles for fixed angles ν_i and ν_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 ν_i and ν_j may also be negative (light travelling in the opposite direction). The full redistribution matrix may be expressed in terms a 2×2 matrix of $n{\times}n$ matrices

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

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

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

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.

290. An alternate way to calculate the redistribution function is the δ–M method of Wiscombe. This method works especially well for highly anisotropic phase functions. The number of quadrature points is specified by M. The δ–M method approximates the true phase function by a phase function consisting of a Dirac delta function and $M-1$ Legendre polynomials

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

where

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

When the δ–M method substitutes $p^*(\nu) \to p(\nu)$, then both the albedo and optical thickness must also be changed, $a^* \to a$ and $\tau^* \to \tau$. 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

$$\tau^* = (1-ag^M)\tau \qquad \text{and} \qquad a^* = a\frac{1-g^M}{1-ag^M}$$

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

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

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

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

291. 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.

292. ⟨ Prototype for *Get_Phi* 292 ⟩ ≡
void *Get_Phi*(**int** *n*, **int** *phase_function*, **double** *g*, **double** ∗∗*h*)

This code is used in sections 288 and 293.

293. ⟨ Definition for *Get_Phi* 293 ⟩ ≡
⟨ Prototype for *Get_Phi* 292 ⟩
{
⟨ Local variables for *Get_Phi* 294 ⟩
⟨ Test for bad calling parameters 295 ⟩
⟨ Initialize the phase function matrix 296 ⟩
⟨ We're done if phase function is isotropic 297 ⟩
⟨ Calculate the quadrature coefficients 298 ⟩
⟨ Create Legendre Polynomial matrix 299 ⟩
⟨ Calculate the coefficients 303 ⟩
⟨ Add the symmetric part of the matrix 304 ⟩
⟨ Free *p* and *chi* 305 ⟩
}

This code is used in section 287.

294. ⟨ Local variables for *Get_Phi* 294 ⟩ ≡

```
int i, j, k;
double g2M, gk, x;
double *chi;
double **p;
```

This code is used in section 293.

295. ⟨ Test for bad calling parameters 295 ⟩ ≡

```
if (g ≠ 0 ∧ phase_function ≠ HENYEY_GREENSTEIN)
  AD_error("Only the Henyey-Greenstein phase function has been implemented\n");
if (fabs(g) ≥ 1) AD_error("Get_Phi was called with a bad g_calc value");
```

This code is used in section 293.

296. ⟨ Initialize the phase function matrix 296 ⟩ ≡

```
for (i = −n; i ≤ n; i++)
  for (j = −n; j ≤ n; j++) h[i][j] = 1;    /* zero the zero column and zero row */
for (i = −n; i ≤ n; i++) {
  h[i][0] = 0.0;
  h[0][i] = 0.0;
}
```

This code is used in section 293.

297. ⟨ We're done if phase function is isotropic 297 ⟩ ≡

```
if (g ≡ 0) return;
```

This code is used in section 293.

298. To avoid extra calculation let's define

$$chi[k] \equiv (2k+1)\chi_k^*$$

This will slighly simplify things later on

⟨ Calculate the quadrature coefficients 298 ⟩ ≡

```
chi = dvector(1, n);
g2M = pow(g, n);
gk = 1.0;
for (k = 1; k < n; k++) {
  gk *= g;
  chi[k] = (2 * k + 1) * (gk − g2M)/(1 − g2M);
}
```

This code is used in section 293.

299. 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 $0 \le k \le n$ 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 $p[i][j] \equiv P_i(\mu_j)$.

⟨ Create Legendre Polynomial matrix 299 ⟩ ≡
  ⟨ Allocate the polynomial matrix 300 ⟩
  ⟨ Fill in all the unique values 301 ⟩
  ⟨ Fill in the symmetric values 302 ⟩

This code is used in section 293.

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

⟨ Allocate the polynomial matrix 300 ⟩ ≡

```
p = dmatrix(0, n, -n, n);
```

This code is used in section 299.

301. 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.

⟨ Fill in all the unique values 301 ⟩ ≡

```
for (j = 1; j ≤ n; j++) {
  p[0][j] = 1;
  x = angle[j];
  p[1][j] = x;
  for (k = 1; k < n; k++) p[k + 1][j] = ((2 * k + 1) * x * p[k][j] - k * p[k - 1][j])/(k + 1);
}
```

This code is used in section 299.

302. 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.

⟨ Fill in the symmetric values 302 ⟩ ≡

```
for (j = 1; j ≤ n; j++)
  for (k = 1; k < n; k++) {
    p[k][-j] = -p[k][j];
    k++;
    p[k][-j] = p[k][j];
  }
```

This code is used in section 299.

303. 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.

⟨ Calculate the coefficients 303 ⟩ ≡

```
for (i = 1; i ≤ n; i++) {
  for (j = i; j ≤ n; j++) {
    for (k = 1; k < n; k++) {
      h[i][j] += chi[k] * p[k][i] * p[k][j];
      h[−i][j] += chi[k] * p[k][−i] * p[k][j];
    }
  }
}
```

This code is used in section 293.

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

$$h(-\nu_i,\nu_j)=h(\nu_j,-\nu_i)$$

and secondly

$$h(-\nu_i,-\nu_j)=h(\nu_j,\nu_i)$$

Next, some entries along the diagonal are filled in using

$$h(-\nu_i,-\nu_i)=h(\nu_i,\nu_i)$$

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

$$h(\nu_i,\nu_j)=h(\nu_j,\nu_i)$$

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 304 ⟩ ≡

```
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 293.

305. ⟨ Free *p* and *chi* 305 ⟩ ≡

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

This code is used in section 293.

306. Main Program.

Here is a quick program that I put together on the 18th of July 1996 to calculate the change in reflection and transmission when a small change in the absorption coefficient is made. Specifically, the absorption coefficient will change from μ_a to $\mu_a + \mu_a\Delta$.

The program reads and input file that contains the optical properties of the slab. The output file will have the same name, but appended by ".out" and contain the change in the reflection and transmission calculated for normal irradiance using 8 quadrature points.

Note that the streams get redirected so that I can use the standard streams for reading, writing, and error messages. This makes interactive stuff problematic, but this whole thing is a batch sort of problem.

All the output for this web file goes into `ad_main.c` but to simplify the Makefile, I create an empty `ad_main.h`.

⟨ `ad_main.h` 306 ⟩ ≡

307. The program begins here

```
⟨ ad_main.c 307 ⟩ ≡
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <unistd.h>
#include <errno.h>
#include "ad_globl.h"
#include "ad_prime.h"
#include "ad_cone.h"
#include "version.h"
  ⟨ print version function 315 ⟩
  ⟨ print usage function 316 ⟩
  ⟨ stringdup together function 317 ⟩
  ⟨ mystrtod function 318 ⟩
  ⟨ validate slab function 320 ⟩
  int main(int argc, char **argv)
  {
    ⟨ Declare variables for main 308 ⟩
    if (argc ≡ 1) {
      print_usage();
      exit(EXIT_FAILURE);
    }
    ⟨ Handle options 312 ⟩
    if (argc ≥ 1) {
      ⟨ Prepare file for reading 313 ⟩
      ⟨ Prepare file for writing 314 ⟩
      while (feof(stdin) ≡ 0) {
        slab.phase_function = HENYEY_GREENSTEIN;
        ⟨ Read line from input file 310 ⟩
        ⟨ Calculate and Print the Results 311 ⟩
      }
    }
    else {
      ⟨ Put optical properties into slab 309 ⟩
      ⟨ Calculate and Print the Results 311 ⟩
    }
    return EXIT_SUCCESS;
  }
```

308. ⟨ Declare variables for *main* 308 ⟩ ≡

```
struct AD_slab_type slab;
int nstreams = 24;
double anisotropy = 0;
double albedo = 0.5;
double index_of_refraction = 1.0;
double index_of_slide1 = 1.0;
double index_of_slide2 = 1.0;
double optical_thickness = 100;
char *g_out_name = Λ;
double g_incident_cosine = 1;
int machine_readable_output = 0;
double R1, T1, URU, UTU;
```

This code is used in section 307.

309. I assume that the optical properties are in the following order — albedo, optical thickness, anisotropy, the index of refraction of the slab, the index of refraction of the top slide, the index of refraction of the bottom slide. The slides are assumed to have no absorption.

⟨ Put optical properties into *slab* 309 ⟩ ≡

```
slab.phase_function = HENYEY_GREENSTEIN;
slab.a = albedo;
slab.b = optical_thickness;
slab.g = anisotropy;
slab.n_slab = index_of_refraction;
slab.n_top_slide = index_of_slide1;
slab.n_bottom_slide = index_of_slide2;
slab.b_top_slide = 0.0;
slab.b_bottom_slide = 0.0;
slab.cos_angle = g_incident_cosine;
```

This code is used in section 307.

310.

⟨ Read line from input file 310 ⟩ ≡

```
{
  int fileflag;
  fileflag = scanf("%lf", &slab.a);
  slab.cos_angle = g_incident_cosine;
  if (fileflag ≠ EOF) fileflag = scanf("%lf", &slab.b);
  if (fileflag ≠ EOF) fileflag = scanf("%lf", &slab.g);
  if (fileflag ≠ EOF) fileflag = scanf("%lf", &slab.n_slab);
  if (fileflag ≠ EOF) fileflag = scanf("%lf", &slab.n_top_slide);
  if (fileflag ≠ EOF) fileflag = scanf("%lf", &slab.n_bottom_slide);
  if (fileflag ≠ EOF) fileflag = scanf("%lf", &slab.b_top_slide);
  if (fileflag ≠ EOF) fileflag = scanf("%lf", &slab.b_bottom_slide);
  if (fileflag ≠ EOF) fileflag = scanf("%d", &nstreams);
}
```

This code is used in section 307.

311. ⟨ Calculate and Print the Results 311 ⟩ ≡

```
validate_slab(slab, nstreams, machine_readable_output);
RT(nstreams, &slab, &R1, &T1, &URU, &UTU);
if (machine_readable_output) printf("%9.5f␣\t%9.5f␣\t%9.5f␣\t%9.5f\n", R1, T1, URU, UTU);
else {
  printf("UR1␣=␣Total␣Reflection␣␣␣for␣Normal␣␣Illumination\n");
  printf("UT1␣=␣Total␣Transmission␣for␣Normal␣␣Illumination\n");
  printf("URU␣=␣Total␣Reflection␣␣␣for␣Diffuse␣Illumination\n");
  printf("UTU␣=␣Total␣Transmission␣for␣Diffuse␣Illumination\n\n");
  printf("␣␣␣UR1␣␣␣␣\t␣␣␣UT1␣␣␣␣\t␣␣␣URU␣␣␣␣\t␣␣␣UTU\n");
  printf("%9.5f␣\t%9.5f␣\t%9.5f␣\t%9.5f\n", R1, T1, URU, UTU);
}
```

This code is used in section 307.

312. use the *getopt* to process options.

⟨ Handle options 312 ⟩ ≡

```
{
  int c;
  double x;
  while ((c = getopt(argc, argv, "hvma:b:g:i:n:o:q:s:t:")) ≠ −1) {
    switch (c) {
    case 'a': albedo = my_strtod(optarg);
      break;
    case 'i': x = my_strtod(optarg);
      if (x < 0 ∨ x > 90) fprintf(stderr, "Incident␣angle␣must␣be␣between␣0␣and␣90␣degrees\n");
      else g_incident_cosine = cos(x * M_PI/180.0);
      break;
    case 'o': g_out_name = strdup(optarg);
      break;
    case 'n': index_of_refraction = my_strtod(optarg);
      break;
    case 's': index_of_slide1 = my_strtod(optarg);
      index_of_slide2 = index_of_slide1;
      break;
    case 't': index_of_slide2 = my_strtod(optarg);
      break;
    case 'm': machine_readable_output = 1;
      break;
    case 'q': nstreams = (int) my_strtod(optarg);
      break;
    case 'b': optical_thickness = my_strtod(optarg);
      break;
    case 'g': anisotropy = my_strtod(optarg);
      break;
    case 'v': print_version();
      exit(EXIT_SUCCESS);
    default: fprintf(stderr, "unknown␣option␣'%c'\n", c);      /* fall through */
    case 'h': print_usage();
      exit(EXIT_SUCCESS);
    }
  }
  argc −= optind;
  argv += optind;
}
```

This code is used in section 307.

313. Make sure that the file is not named '-' and warn about too many files

⟨ Prepare file for reading 313 ⟩ ≡

```
if (argc > 1) {
  fprintf(stderr, "Only␣a␣single␣file␣can␣be␣processed␣at␣a␣time\n");
  fprintf(stderr, "try␣'apply␣ad␣file1␣file2␣...␣fileN'\n");
  exit(EXIT_FAILURE);
}
if (argc ≡ 1 ∧ strcmp(argv[0], "-") ≠ 0) {    /* filename exists and != "-" */
  if (freopen(argv[0], "r", stdin) ≡ Λ) {
    fprintf(stderr, "Could␣not␣open␣file␣'%s'\n", argv[0]);
    exit(EXIT_FAILURE);
  }
  if (g_out_name ≡ Λ) g_out_name = strdup_together(argv[0], ".rt");
}
```

This code is used in section 307.

314. Take care of all the output files

⟨ Prepare file for writing 314 ⟩ ≡

```
if (g_out_name ≠ Λ) {
  if (freopen(g_out_name, "w", stdout) ≡ Λ) {
    fprintf(stderr, "Could␣not␣open␣file␣<%s>␣for␣output", g_out_name);
    exit(EXIT_FAILURE);
  }
}
```

This code is used in section 307.

315. ⟨ print version function 315 ⟩ ≡

```
static void print_version(void)
{
  fprintf(stdout, "ad␣%s\n", Version);
  fprintf(stdout, "Copyright␣1993-2024␣Scott␣Prahl,␣scott.prahl@oit.edu\n");
  fprintf(stdout, "␣␣␣␣␣␣␣␣␣␣(see␣Applied␣Optics,␣32:559-568,␣1993)\n\n");
  fprintf(stdout, "This␣is␣free␣software;␣see␣the␣source␣for␣copying␣conditions.\n");
  fprintf(stdout, "There␣is␣no␣warranty;␣not␣even␣for␣MERCHANTABILITY␣or␣FITNESS.\n");
  fprintf(stdout, "FOR␣A␣PARTICULAR␣PURPOSE.\n");
}
```

This code is used in section 307.

316. ⟨ print usage function 316 ⟩ ≡

```
static void print_usage(void)
{
  fprintf(stdout, "ad␣%s\n\n", Version);
  fprintf(stdout, "ad␣finds␣the␣reflection␣and␣transmission␣from␣optical␣properties\n\n");
  fprintf(stdout, "Usage:␣␣ad␣[options]␣input\n\n");
  fprintf(stdout, "Options:\n");
  fprintf(stdout, "␣␣-a␣#␣␣␣␣␣␣␣␣␣␣␣␣␣albedo␣(0-1)\n");
  fprintf(stdout, "␣␣-b␣#␣␣␣␣␣␣␣␣␣␣␣␣␣optical␣thickness␣(>0)\n");
  fprintf(stdout, "␣␣-g␣#␣␣␣␣␣␣␣␣␣␣␣␣␣scattering␣anisotropy␣(-1␣to␣1)\n");
  fprintf(stdout, "␣␣-h␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣display␣help\n");
  fprintf(stdout, "␣␣-i␣theta␣␣␣␣␣␣␣␣␣oblique␣incidence␣at␣angle␣theta\n");
  fprintf(stdout, "␣␣-m␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣machine␣readable␣output\n");
  fprintf(stdout, "␣␣-n␣#␣␣␣␣␣␣␣␣␣␣␣␣␣specify␣index␣of␣refraction␣of␣slab\n");
  fprintf(stdout, "␣␣-o␣filename␣␣␣␣␣␣explicitly␣specify␣filename␣for␣output\n");
  fprintf(stdout, "␣␣-q␣#␣␣␣␣␣␣␣␣␣␣␣␣␣quadrature␣points␣4,␣8,␣16,␣32\n");
  fprintf(stdout, "␣␣-s␣#␣␣␣␣␣␣␣␣␣␣␣␣␣specify␣index␣of␣refraction␣of␣slide\n");
  fprintf(stdout, "␣␣-v␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣version␣information\n");
  fprintf(stdout, "Examples:\n");
  fprintf(stdout, "␣␣ad␣data␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣UR1,␣UT1,␣URU,␣UTU␣in␣data.rt\n");
  fprintf(stdout, "␣␣ad␣-m␣data␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣data.rt␣in␣machine␣readable␣format\n");
  fprintf(stdout, "␣␣ad␣data␣-o␣out.txt␣␣␣␣␣␣␣␣␣␣␣␣␣out.txt␣is␣the␣\n");
  fprintf(stdout, "␣␣ad␣-a␣0.3␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣a=0.3,␣b=inf,␣g=0.0,␣n=1.0\n");
  fprintf(stdout, "␣␣ad␣-a␣0.3␣-b␣0.4␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣a=0.3,␣b=0.4,␣g=0.0,␣n=1.0\n");
  fprintf(stdout, "␣␣ad␣-a␣0.3␣-b␣0.4␣-g␣0.5␣␣␣␣␣␣␣␣a=0.3,␣b=0.4,␣g=0.5,␣n=1.0\n");
  fprintf(stdout, "␣␣ad␣-a␣0.3␣-b␣0.4␣-n␣1.5␣␣␣␣␣␣␣␣a=0.3,␣b=0.4,␣g=0.0,␣n=1.5\n\n");
  fprintf(stdout, "inputfile␣has␣lines␣of␣the␣form:\n");
  fprintf(stdout, "␣␣␣␣a␣b␣g␣nslab␣ntopslide␣nbottomlslide␣btopslide␣bbottomslide␣q\n");
  fprintf(stdout, "where:\n");
  fprintf(stdout, "␣␣␣␣1)␣a␣=␣albedo\n");
  fprintf(stdout, "␣␣␣␣2)␣b␣=␣optical␣thickness\n");
  fprintf(stdout, "␣␣␣␣3)␣g␣=␣anisotropy\n");
  fprintf(stdout, "␣␣␣␣4)␣nslab␣=␣index␣of␣refraction␣of␣slab\n");
  fprintf(stdout, "␣␣␣␣5)␣ntopslide␣=␣index␣of␣refraction␣of␣glass␣slide␣on␣top\n");
  fprintf(stdout, "␣␣␣␣6)␣nbottomslide␣=␣index␣of␣refraction␣of␣glass␣slide␣on␣bottom\n");
  fprintf(stdout, "␣␣␣␣7)␣btopslide␣=␣optical␣depth␣of␣top␣slide␣(for␣absorbing␣slides)\n");
  fprintf(stdout,
      "␣␣␣␣8)␣bbottomslide␣=␣optical␣depth␣of␣bottom␣slide␣(for␣absorbing␣slides)\n");
  fprintf(stdout, "␣␣␣␣9)␣q␣=␣number␣of␣quadrature␣points\n\n");
  fprintf(stdout, "Report␣bugs␣to␣<scott.prahl@oit.edu>\n\n");
}
```

This code is used in section 307.

317. returns a new string consisting of s+t

⟨ stringdup together function 317 ⟩ ≡

```
static char *strdup_together(char *s, char *t)
{
  char *both;
  if (s ≡ Λ) {
    if (t ≡ Λ) return Λ;
    return strdup(t);
  }
  if (t ≡ Λ) return strdup(s);
  both = malloc(strlen(s) + strlen(t) + 1);
  if (both ≡ Λ) fprintf(stderr, "Could␣not␣allocate␣memory␣for␣both␣strings.\n");
  strcpy(both, s);
  strcat(both, t);
  return both;
}
```

This code is used in section 307.

318. catch parsing errors in strtod

⟨ mystrtod function 318 ⟩ ≡

```
static double my_strtod(const char *str)
{
  char *endptr;
  errno = 0;
  double val = strtod(str, &endptr);
  if (endptr ≡ str) {   /* No digits were found */
    fprintf(stderr, "Error:␣No␣conversion␣could␣be␣performed␣for␣`%s`.\n", str);
    exit(EXIT_FAILURE);
  }
  if (*endptr ≠ '\0') {   /* String contains extra characters after the number */
    printf("Partial␣conversion:␣converted␣value␣=␣%f,␣remaining␣string␣=␣%s\n", val, endptr);
    exit(EXIT_FAILURE);
  }
  if (errno ≡ ERANGE) {
      /* The converted value is out of range of representable values by a double */
    printf("Error:␣The␣value␣is␣out␣of␣range␣of␣double.\n");
    exit(EXIT_FAILURE);
  }
  return val;
}
```

This code is used in section 307.

319. ⟨ print short version function 319 ⟩ ≡

```
static void print_short_version(void)
{
  fprintf(stdout, "%s", VersionShort);
}
```

320. Make sure that the input values are correct

⟨ validate slab function 320 ⟩ ≡

```
static void validate_slab(struct AD_slab_type slab, int nstreams, int machine)
{
  if (slab.a < 0 ∨ slab.a > 1) {
    fprintf(stderr, "Bad␣Albedo␣a=%f\n", slab.a);
    exit(EXIT_FAILURE);
  }
  if (slab.b < 0) {
    fprintf(stderr, "Bad␣Optical␣Thickness␣b=%f\n", slab.b);
    exit(EXIT_FAILURE);
  }
  if (slab.g ≤ −1 ∨ slab.g ≥ 1) {
    fprintf(stderr, "Bad␣Anisotropy␣g=%f\n", slab.g);
    exit(EXIT_FAILURE);
  }
  if (slab.n_slab < 0 ∨ slab.n_slab > 10) {
    fprintf(stderr, "Bad␣Slab␣Index␣n=%f\n", slab.n_slab);
    exit(EXIT_FAILURE);
  }
  if (slab.n_top_slide < 1 ∨ slab.n_top_slide > 10) {
    fprintf(stderr, "Bad␣Top␣Slide␣Index␣n=%f\n", slab.n_top_slide);
    exit(EXIT_FAILURE);
  }
  if (slab.n_bottom_slide < 1 ∨ slab.n_bottom_slide > 10) {
    fprintf(stderr, "Bad␣Top␣Slide␣Index␣n=%f\n", slab.n_bottom_slide);
    exit(EXIT_FAILURE);
  }
  if (slab.b_top_slide < 0 ∨ slab.b_top_slide > 10) {
    fprintf(stderr, "Bad␣Top␣Slide␣Optical␣Thickness␣b=%f\n", slab.b_top_slide);
    exit(EXIT_FAILURE);
  }
  if (slab.b_bottom_slide < 0 ∨ slab.b_bottom_slide > 10) {
    fprintf(stderr, "Bad␣Bottom␣Slide␣Optical␣Thickness␣b=%f\n", slab.b_bottom_slide);
    exit(EXIT_FAILURE);
  }
  if (nstreams < 4 ∨ nstreams % 4 ≠ 0) {
    fprintf(stderr, "Bad␣Number␣of␣Quadrature␣Points␣npts=%d\n", nstreams);
    fprintf(stderr, "Should␣be␣a␣multiple␣of␣four!\n");
    exit(EXIT_FAILURE);
  }
}
```

This code is used in section 307.

321. 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.