1. Main Program.

Here is a relatively robust command line utility that shows how the iad and ad subroutines might be called. It suffers because it is written in CWEB and I used the macro expansion feature instead of creating separate functions. Oh well.

I create an empty file `iad_main.h` to simplify the Makefile

⟨ `iad_main.h` 1 ⟩ ≡

2. All the actual output for this web file goes into `iad_main.c`

```
⟨ iad_main.c 2 ⟩ ≡
  ⟨ Include files for main 3 ⟩⟨ print version function 17 ⟩⟨ print usage function 18 ⟩⟨ stringdup together
        function 24 ⟩⟨ seconds elapsed function 25 ⟩⟨ print error legend function 23 ⟩⟨ print dot
        function 27 ⟩⟨ calculate coefficients function 19 ⟩⟨ parse string into array function 26 ⟩⟨ print
        results header function 21 ⟩⟨ Print results function 22 ⟩int main(int argc, char **argv){
        ⟨ Declare variables for main 4 ⟩⟨ Handle options 5 ⟩Initialize_Measure(&m);
    ⟨ Command-line changes to m 15 ⟩Initialize_Result(m, &r); ⟨ Command-line changes to r 10 ⟩
    if (cl_forward_calc ≠ UNINITIALIZED) {
      ⟨ Calculate and Print the Forward Calculation 6 ⟩return 0;
    }
    if (process_command_line) {
      ⟨ Count command-line measurements 16 ⟩⟨ Calculate and write optical properties 8 ⟩return 0;
    }
    ⟨ prepare file for reading 7 ⟩if (Read_Header(stdin, &m, &params) ≡ 0) { start_time = clock( );
        ⟨ Command-line changes to m 15 ⟩
    while (Read_Data_Line(stdin, &m, params) ≡ 0) {
      ⟨ Calculate and write optical properties 8 ⟩first_line = 0;
    }
    }
    if (cl_verbosity > 0) fprintf(stderr, "\n\n");
    if (any_error ∧ cl_verbosity > 1) print_error_legend( );
    return 0; }
```

3. The first two defines are to stop Visual C++ from silly complaints

⟨ Include files for *main* 3 ⟩ ≡

```
#define _CRT_SECURE_NO_WARNINGS
#define _CRT_NONSTDC_NO_WARNINGS
#define NO_SLIDES 0
#define ONE_SLIDE_ON_TOP 1
#define TWO_IDENTICAL_SLIDES 2
#define ONE_SLIDE_ON_BOTTOM 3
#define MR_IS_ONLY_RD 1
#define MT_IS_ONLY_TD 2
#define NO_UNSCATTERED_LIGHT 3
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <ctype.h>
#include "ad_globl.h"
#include "ad_prime.h"
#include "iad_type.h"
#include "iad_pub.h"
#include "iad_io.h"
#include "iad_calc.h"
#include "iad_util.h"
#include "mygetopt.h"
#include "version.h"
#include "mc_lost.h"
#include "ad_frsnl.h"
  extern char *optarg;
  extern int optind;
```

This code is used in section 2.

4. ⟨ Declare variables for *main* 4 ⟩ ≡

```
struct measure_type m;
struct invert_type r;
double m_r, m_t;
char *g_out_name = Λ;
char c;
int first_line = 1;
long n_photons = 100000;
int MC_iterations = 19;
int any_error = 0;
int process_command_line = 0;
int params = 0;
int cl_quadrature_points = 8;
int cl_verbosity = 2;
double cl_forward_calc = UNINITIALIZED;
double cl_default_a = UNINITIALIZED;
double cl_default_g = UNINITIALIZED;
double cl_default_b = UNINITIALIZED;
double cl_default_mua = UNINITIALIZED;
double cl_default_mus = UNINITIALIZED;
double cl_tolerance = UNINITIALIZED;
double cl_no_unscat = UNINITIALIZED;
double cl_slide_OD = UNINITIALIZED;
double cl_cos_angle = UNINITIALIZED;
double cl_beam_d = UNINITIALIZED;
double cl_sample_d = UNINITIALIZED;
double cl_sample_n = UNINITIALIZED;
double cl_slide_d = UNINITIALIZED;
double cl_slide_n = UNINITIALIZED;
double cl_slides = UNINITIALIZED;
double cl_default_fr = UNINITIALIZED;
double cl_rstd_t = UNINITIALIZED;
double cl_rstd_r = UNINITIALIZED;
double cl_mus0 = UNINITIALIZED;
double cl_musp0 = UNINITIALIZED;
double cl_mus0_pwr = UNINITIALIZED;
double cl_mus0_lambda = UNINITIALIZED;
double cl_UR1 = UNINITIALIZED;
double cl_UT1 = UNINITIALIZED;
double cl_Tc = UNINITIALIZED;
double cl_method = UNINITIALIZED;
double cl_num_spheres = UNINITIALIZED;
double cl_sphere_one[5] = {UNINITIALIZED, UNINITIALIZED, UNINITIALIZED, UNINITIALIZED,
    UNINITIALIZED};
double cl_sphere_two[5] = {UNINITIALIZED, UNINITIALIZED, UNINITIALIZED, UNINITIALIZED,
    UNINITIALIZED};
clock_t start_time = clock( );
```

This code is used in section 2.

5. use the *mygetop* to process options. We only handle help at the moment

```
⟨ Handle options 5 ⟩ ≡
  while ((c = my_getopt(argc, argv,
        "?1:2:a:A:b:B:c:d:D:e:E:f:F:g:G:hi:n:N:M:o:p:q:r:R:S:t:T:u:vV:x:Xz")) ≠ EOF) {
    int n;
    char cc;
    switch (c) {
    case '1': parse_string_into_array(optarg, cl_sphere_one, 5);
      break;
    case '2': parse_string_into_array(optarg, cl_sphere_two, 5);
      break;
    case 'a': cl_default_a = strtod(optarg, Λ);
      break;
    case 'A': cl_default_mua = strtod(optarg, Λ);
      break;
    case 'b': cl_default_b = strtod(optarg, Λ);
      break;
    case 'B': cl_beam_d = strtod(optarg, Λ);
      break;
    case 'c': cl_no_unscat = strtod(optarg, Λ);
      break;
    case 'd': cl_sample_d = strtod(optarg, Λ);
      break;
    case 'D': cl_slide_d = strtod(optarg, Λ);
      break;
    case 'e': cl_tolerance = strtod(optarg, Λ);
      break;
    case 'E': cl_slide_OD = strtod(optarg, Λ);
      break;
    case 'f': cl_default_fr = strtod(optarg, Λ);
      break;
    case 'F':      /* initial digit means this is mus is constant */
      if (isdigit(optarg[0])) {
        cl_default_mus = strtod(optarg, Λ);
        break;
      }      /* should be a string like 'R 1000 1.2 -1.8' */
      n = sscanf(optarg, "%c %lf %lf %lf", &cc, &cl_mus0_lambda, &cl_mus0, &cl_mus0_pwr);
      if (n ≠ 4) {
        fprintf(stderr, "Screwy argument for -F option\n");
        exit(1);
      }
      if (cc ≡ 'R' ∨ cc ≡ 'r') {
        cl_musp0 = cl_mus0;
        cl_mus0 = UNINITIALIZED;
      }
      break;
    case 'g': cl_default_g = strtod(optarg, Λ);
      break;
    case 'G':
      if (optarg[0] ≡ '0') cl_slides = NO_SLIDES;
      else if (optarg[0] ≡ '2') cl_slides = TWO_IDENTICAL_SLIDES;
      else if (optarg[0] ≡ 't' ∨ optarg[0] ≡ 'T') cl_slides = ONE_SLIDE_ON_TOP;
```

```
    else if (optarg[0] ≡ 'b' ∨ optarg[0] ≡ 'B') cl_slides = ONE_SLIDE_ON_BOTTOM;
    else {
      fprintf(stderr, "Argument for -G option must be 't' (for top)");
      fprintf(stderr, " or 'b' for bottom or '0' or '2'\n");
      exit(1);
    }
    break;
  case 'i': cl_cos_angle = strtod(optarg, Λ);
    if (cl_cos_angle < 0 ∨ cl_cos_angle > 90)
      fprintf(stderr, "Incident angle must be between 0 and 90 degrees\n");
    else cl_cos_angle = cos(cl_cos_angle * 3.1415926535/180.0);
    break;
  case 'M': MC_iterations = (int) strtod(optarg, Λ);
    break;
  case 'n': cl_sample_n = strtod(optarg, Λ);
    break;
  case 'N': cl_slide_n = strtod(optarg, Λ);
    break;
  case 'o': g_out_name = strdup(optarg);
    break;
  case 'p': n_photons = (int) strtod(optarg, Λ);
    break;
  case 'q': cl_quadrature_points = (int) strtod(optarg, Λ);
    if (cl_quadrature_points % 4 ≠ 0) {
      fprintf(stderr, "Number of quadrature points must be a multiple of 4\n");
      exit(1);
    }
    if ((cl_cos_angle ≠ UNINITIALIZED) ∧ (cl_quadrature_points % 12 ≠ 0)) {
      fprintf(stderr, "Quadrature must be 12, 24, 36, ... for oblique incidence\n");
      exit(1);
    }
    break;
  case 'r': cl_UR1 = strtod(optarg, Λ);
    process_command_line = 1;
    break;
  case 'R': cl_rstd_r = strtod(optarg, Λ);
    break;
  case 'S': cl_num_spheres = (int) strtod(optarg, Λ);
    break;
  case 't': cl_UT1 = strtod(optarg, Λ);
    process_command_line = 1;
    break;
  case 'T': cl_rstd_t = strtod(optarg, Λ);
    break;
  case 'u': cl_Tc = strtod(optarg, Λ);
    process_command_line = 1;
    break;
  case 'v': print_version( );
    break;
  case 'V': cl_verbosity = strtod(optarg, Λ);
    break;
  case 'x': Set_Debugging((int) strtod(optarg, Λ));
```

```
      break;
    case 'X': cl_method = COMPARISON;
      break;
    case 'z': cl_forward_calc = 1;
      process_command_line = 1;
      break;
    default: fprintf(stderr, "unknown␣option␣'%c'\n", c);      /* fall through */
    case 'h': case '?': print_usage( );
      break;
    }
  }
  argc -= optind;
  argv += optind;
```

This code is used in section 2.

6. ⟨ Calculate and Print the Forward Calculation 6 ⟩ ≡

```
{
  double mu_sp, mu_a;
  if (cl_default_a ≡ UNINITIALIZED) {
    if (cl_default_mua ≠ UNINITIALIZED ∧ cl_default_mus ≠ UNINITIALIZED)
      r.a = cl_default_mus/(cl_default_mua + cl_default_mus);
    else r.a = 0;
  }
  else r.a = cl_default_a;
  if (cl_default_b ≡ UNINITIALIZED) {
    if (cl_default_mua ≠ UNINITIALIZED ∧ cl_default_mus ≠ UNINITIALIZED ∧ cl_sample_d ≠
        UNINITIALIZED) r.b = cl_sample_d * (cl_default_mua + cl_default_mus);
    else r.b = HUGE_VAL;
  }
  else r.b = cl_default_b;
  if (cl_default_g ≡ UNINITIALIZED) r.g = 0;
  else r.g = cl_default_g;
  r.slab.a = r.a;
  r.slab.b = r.b;
  r.slab.g = r.g;
  Calculate_MR_MT(m, r, MC_iterations, &m_r, &m_t);
  Calculate_Mua_Musp(m, r, &mu_sp, &mu_a);
  if (cl_verbosity > 0) {
    Write_Header(m, r, -1);
    print_results_header(stdout);
  }
  print_optical_property_result(stdout, m, r, m_r, m_t, mu_a, mu_sp, 0, 0);
}
```

This code is used in section 2.

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

```
⟨ prepare file for reading 7 ⟩ ≡
  if (argc > 1) {
    fprintf(stderr, "Only␣a␣single␣file␣can␣be␣processed␣at␣a␣time\n");
    fprintf(stderr, "try␣'apply␣iad␣file1␣file2␣...␣fileN'\n");
    exit(1);
  }
  if (argc ≡ 1 ∧ strcmp(argv[0], "-") ≠ 0) {      /* filename exists and != "-" */
    int n;
    char *base_name, *rt_name;
    base_name = strdup(argv[0]);
    n = (int)(strlen(base_name) − strlen(".rxt"));
    if (n > 0 ∧ strstr(base_name + n, ".rxt") ≠ Λ) base_name[n] = '\0';
    rt_name = strdup_together(base_name, ".rxt");
    if (freopen(argv[0], "r", stdin) ≡ Λ ∧ freopen(rt_name, "r", stdin) ≡ Λ) {
      fprintf(stderr, "Could␣not␣open␣either␣'%s'␣or␣'%s'\n", argv[0], rt_name);
      exit(1);
    }
    if (g_out_name ≡ Λ) g_out_name = strdup_together(base_name, ".txt");
    free(rt_name);
    free(base_name);
  }
  if (g_out_name ≠ Λ) {
    if (freopen(g_out_name, "w", stdout) ≡ Λ) {
      fprintf(stderr, "Could␣not␣open␣file␣'%s'␣for␣output\n", g_out_name);
      exit(1);
    }
  }
```

This code is used in section 2.

8. Need to explicitly reset *r.search* each time through the loop, because it will get altered by the calculation process. We want to be able to let different lines have different constraints. In particular consider the file *newton.tst*. In that file the first two rows contain three real measurements and the last two have the collimated transmission explicitly set to zero — in other words there are really only two measurements.

```
⟨ Calculate and write optical properties 8 ⟩ ≡
  { ⟨ Local Variables for Calculation 9 ⟩
  Initialize_Result(m, &r);
  ⟨ Command-line changes to r 10 ⟩
  if (cl_method ≡ COMPARISON ∧ m.d_sphere_r ≠ 0 ∧ m.as_r ≡ 0) {
    fprintf(stderr, "A␣dual-beam␣measurement␣is␣specified,␣but␣no␣port␣sizes.\n");
    fprintf(stderr, "You␣might␣forsake␣the␣-X␣option␣and␣use␣zero␣spheres␣(which␣gives\n");
    fprintf(stderr, "the␣same␣result␣except␣lost␣light␣is␣not␣taken␣into␣account).\n");
    fprintf(stderr, "Alternatively,␣bite␣the␣bullet␣and␣enter␣your␣sphere␣parameters,\n");
    fprintf(stderr, "with␣the␣knowledge␣that␣only␣the␣beam␣diameter␣and␣sample␣port\n");
    fprintf(stderr, "diameter␣are␣worth␣obsessing␣over.\n");
    exit(0);
  }
  ⟨ Write Header 11 ⟩
  Inverse_RT(m, &r);
  calculate_coefficients(m, r, &LR, &LT, &mu_sp, &mu_a);
  ⟨ Improve result using Monte Carlo 12 ⟩
  print_optical_property_result(stdout, m, r, LR, LT, mu_a, mu_sp, mc_iter, rt_total);
  if (Debug(DEBUG_LOST_LIGHT)) fprintf(stderr, "\n");
  else print_dot (start_time, r . error , mc_total, rt_total, 99, cl_verbosity, &any_error ) ; }
```

This code is used in section 2.

9.

```
⟨ Local Variables for Calculation 9 ⟩ ≡
  static int rt_total = 0;
  static int mc_total = 0;
  int mc_iter = 0;
  double ur1 = 0;
  double ut1 = 0;
  double uru = 0;
  double utu = 0;
  double mu_a = 0;
  double mu_sp = 0;
  double LR = 0;
  double LT = 0;
  rt_total++;
```

This code is used in section 8.

10. ⟨ Command-line changes to r 10 ⟩ ≡

```
r.method.quad_pts = cl_quadrature_points;
if (cl_default_a ≠ UNINITIALIZED) r.default_a = cl_default_a;
if (cl_default_mua ≠ UNINITIALIZED) {
  r.default_mua = cl_default_mua;
  if (cl_sample_d ≠ UNINITIALIZED) r.default_ba = cl_default_mua * cl_sample_d;
  else r.default_ba = cl_default_mua * m.slab_thickness;
}
if (cl_default_b ≠ UNINITIALIZED) r.default_b = cl_default_b;
if (cl_default_g ≠ UNINITIALIZED) r.default_g = cl_default_g;
if (cl_tolerance ≠ UNINITIALIZED) {
  r.tolerance = cl_tolerance;
  r.MC_tolerance = cl_tolerance;
}
if (cl_musp0 ≠ UNINITIALIZED)
  cl_mus0 = (r.default_g ≠ UNINITIALIZED) ? cl_musp0/(1 − r.default_g) : cl_musp0;
if (cl_mus0 ≠ UNINITIALIZED ∧ m.lambda ≠ 0)
  cl_default_mus = cl_mus0 * pow(m.lambda/cl_mus0_lambda, cl_mus0_pwr);
if (cl_default_mus ≠ UNINITIALIZED) {
  r.default_mus = cl_default_mus;
  if (cl_sample_d ≠ UNINITIALIZED) r.default_bs = cl_default_mus * cl_sample_d;
  else r.default_bs = cl_default_mus * m.slab_thickness;
}
```

This code is used in sections 2 and 8.

11. ⟨ Write Header 11 ⟩ ≡

```
if (rt_total ≡ 1 ∧ cl_verbosity > 0) {
  Write_Header(m, r, params);
  if (MC_iterations > 0) {
    if (n_photons ≥ 0)
      fprintf(stdout, "#␣␣Photons␣used␣to␣estimate␣lost␣light␣=␣␣␣%ld\n", n_photons);
    else fprintf(stdout, "#␣␣␣␣␣␣Time␣used␣to␣estimate␣lost␣light␣=␣␣␣%ld␣ms\n", −n_photons);
  }
  else fprintf(stdout, "#␣␣Photons␣used␣to␣estimate␣lost␣light␣=␣␣␣0\n");
  fprintf(stdout, "#\n");
  print_results_header(stdout);
}
```

This code is used in section 8.

12. Use Monte Carlo to figure out how much light leaks out. We use the sphere corrected values as the starting values and only do try Monte Carlo when spheres are used, the albedo unknown or non-zero, and there has been no error. The sphere parameters must be known because otherwise the beam size and the port size are unknown.

```
⟨ Improve result using Monte Carlo 12 ⟩ ≡
  if (m.as_r ≠ 0 ∧ r.default_a ≠ 0 ∧ m.num_spheres > 0) { double mu_sp_last = mu_sp;
  double mu_a_last = mu_a;
  if (Debug(DEBUG_LOST_LIGHT)) {
    print_results_header(stderr);
    print_optical_property_result(stderr, m, r, LR, LT, mu_a, mu_sp, mc_iter, rt_total);
  }
  while (mc_iter < MC_iterations) { MC_Lost(m, r, −1000, &ur1, &ut1, &uru, &utu, &m.ur1_lost,
      &m.ut1_lost, &m.uru_lost, &m.utu_lost);
  mc_total++;
  mc_iter++;
  Inverse_RT(m, &r);
  calculate_coefficients(m, r, &LR, &LT, &mu_sp, &mu_a);
  if (fabs(mu_a_last−mu_a)/(mu_a+0.0001) < r.MC_tolerance∧fabs(mu_sp_last−mu_sp)/(mu_sp+0.0001) <
      r.MC_tolerance) break;
  mu_a_last = mu_a;
  mu_sp_last = mu_sp;
  if (Debug(DEBUG_LOST_LIGHT))
    print_optical_property_result(stderr, m, r, LR, LT, mu_a, mu_sp, mc_iter, rt_total);
  else print_dot (start_time, r . error , mc_total, rt_total, mc_iter, cl_verbosity, &any_error ) ; if ( r .
      error ≠ IAD_NO_ERROR ) break; } }
```

This code is used in section 8.

13. ⟨ Testing MC code 13 ⟩ ≡

```
{
  struct AD_slab_type s;
  double ur1, ut1, uru, utu;
  double adur1, adut1, aduru, adutu;
  s.a = 0.0;
  s.b = 0.5;
  s.g = 0.0;
  s.phase_function = HENYEY_GREENSTEIN;
  s.n_slab = 1.0;
  s.n_top_slide = 1.0;
  s.n_bottom_slide = 1.0;
  s.b_top_slide = 0;
  s.b_bottom_slide = 0;
  MC_RT(s, &ur1, &ut1, &uru, &utu);
  RT(32, &s, &adur1, &adut1, &aduru, &adutu);
  fprintf(stderr, "\na=%5.4f␣b=%5.4f␣g=%5.4f␣n=%5.4f␣ns=%5.4f\n", s.a, s.b, s.g, s.n_slab,
      s.n_top_slide);
  fprintf(stderr, "␣␣␣␣UR1␣␣␣␣␣␣␣␣␣␣␣␣␣UT1␣␣␣␣␣␣␣␣␣␣␣␣␣URU␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣UTU\n");
  fprintf(stderr, "␣␣AD␣␣␣␣MC␣␣␣␣␣␣␣␣␣AD␣␣␣␣MC␣␣␣␣␣␣␣␣␣AD␣␣␣␣MC␣␣␣␣␣␣␣␣␣␣␣AD␣␣␣␣MC␣\n");
  fprintf(stderr, "%5.4f␣%5.4f␣␣␣␣%5.4f␣%5.4f␣␣␣␣", adur1, ur1, adut1, ut1);
  fprintf(stderr, "%5.4f␣%5.4f␣␣␣␣%5.4f␣%5.4f\n␣", aduru, uru, adutu, utu);
  s.b = 100.0;
  s.n_slab = 1.5;
  fprintf(stderr, "\na=%5.4f␣b=%5.4f␣g=%5.4f␣n=%5.4f␣ns=%5.4f\n", s.a, s.b, s.g, s.n_slab,
      s.n_top_slide);
  MC_RT(s, &ur1, &ut1, &uru, &utu);
  RT(32, &s, &adur1, &adut1, &aduru, &adutu);
  fprintf(stderr, "%5.4f␣%5.4f␣␣␣␣%5.4f␣%5.4f␣␣␣␣", adur1, ur1, adut1, ut1);
  fprintf(stderr, "%5.4f␣%5.4f␣␣␣␣%5.4f␣%5.4f\n␣", aduru, uru, adutu, utu);
  s.n_slab = 2.0;
  fprintf(stderr, "\na=%5.4f␣b=%5.4f␣g=%5.4f␣n=%5.4f␣ns=%5.4f\n", s.a, s.b, s.g, s.n_slab,
      s.n_top_slide);
  MC_RT(s, &ur1, &ut1, &uru, &utu);
  RT(32, &s, &adur1, &adut1, &aduru, &adutu);
  fprintf(stderr, "%5.4f␣%5.4f␣␣␣␣%5.4f␣%5.4f␣␣␣␣", adur1, ur1, adut1, ut1);
  fprintf(stderr, "%5.4f␣%5.4f␣␣␣␣%5.4f␣%5.4f\n␣", aduru, uru, adutu, utu);
  s.n_slab = 1.5;
  s.n_top_slide = 1.5;
  s.n_bottom_slide = 1.5;
  fprintf(stderr, "\na=%5.4f␣b=%5.4f␣g=%5.4f␣n=%5.4f␣ns=%5.4f\n", s.a, s.b, s.g, s.n_slab,
      s.n_top_slide);
  MC_RT(s, &ur1, &ut1, &uru, &utu);
  RT(32, &s, &adur1, &adut1, &aduru, &adutu);
  fprintf(stderr, "%5.4f␣%5.4f␣␣␣␣%5.4f␣%5.4f␣␣␣␣", adur1, ur1, adut1, ut1);
  fprintf(stderr, "%5.4f␣%5.4f␣␣␣␣%5.4f␣%5.4f\n␣", aduru, uru, adutu, utu);
  s.n_slab = 1.3;
  s.n_top_slide = 1.5;
  s.n_bottom_slide = 1.5;
  fprintf(stderr, "\na=%5.4f␣b=%5.4f␣g=%5.4f␣n=%5.4f␣ns=%5.4f\n", s.a, s.b, s.g, s.n_slab,
      s.n_top_slide);
  MC_RT(s, &ur1, &ut1, &uru, &utu);
```

```
    RT(32, &s, &adur1, &adut1, &aduru, &adutu);
    fprintf(stderr, "%5.4f␣%5.4f␣␣␣%5.4f␣%5.4f␣␣␣", adur1, ur1, adut1, ut1);
    fprintf(stderr, "%5.4f␣%5.4f␣␣␣%5.4f␣%5.4f\n␣", aduru, uru, adutu, utu);
    s.a = 0.5;
    s.b = 1.0;
    s.n_slab = 1.0;
    s.n_top_slide = 1.0;
    s.n_bottom_slide = 1.0;
    fprintf(stderr, "\na=%5.4f␣b=%5.4f␣g=%5.4f␣n=%5.4f␣ns=%5.4f\n", s.a, s.b, s.g, s.n_slab,
        s.n_top_slide);
    MC_RT(s, &ur1, &ut1, &uru, &utu);
    RT(32, &s, &adur1, &adut1, &aduru, &adutu);
    fprintf(stderr, "%5.4f␣%5.4f␣␣␣%5.4f␣%5.4f␣␣␣", adur1, ur1, adut1, ut1);
    fprintf(stderr, "%5.4f␣%5.4f␣␣␣%5.4f␣%5.4f\n␣", aduru, uru, adutu, utu);
    s.g = 0.5;
    fprintf(stderr, "\na=%5.4f␣b=%5.4f␣g=%5.4f␣n=%5.4f␣ns=%5.4f\n", s.a, s.b, s.g, s.n_slab,
        s.n_top_slide);
    MC_RT(s, &ur1, &ut1, &uru, &utu);
    RT(32, &s, &adur1, &adut1, &aduru, &adutu);
    fprintf(stderr, "%5.4f␣%5.4f␣␣␣%5.4f␣%5.4f␣␣␣", adur1, ur1, adut1, ut1);
    fprintf(stderr, "%5.4f␣%5.4f␣␣␣%5.4f␣%5.4f\n␣", aduru, uru, adutu, utu);
    s.n_slab = 1.5;
    fprintf(stderr, "\na=%5.4f␣b=%5.4f␣g=%5.4f␣n=%5.4f␣ns=%5.4f\n", s.a, s.b, s.g, s.n_slab,
        s.n_top_slide);
    MC_RT(s, &ur1, &ut1, &uru, &utu);
    RT(32, &s, &adur1, &adut1, &aduru, &adutu);
    fprintf(stderr, "%5.4f␣%5.4f␣␣␣%5.4f␣%5.4f␣␣␣", adur1, ur1, adut1, ut1);
    fprintf(stderr, "%5.4f␣%5.4f␣␣␣%5.4f␣%5.4f\n␣", aduru, uru, adutu, utu);
}
```

14. ⟨ old formatting 14 ⟩ ≡

```
if (cl_verbosity > 0 ∧ count % 100 ≡ 0) fprintf(stderr, "\n");
if (cl_verbosity > 0) printf(format2, m.m_r, m.m_t, m.m_u, r.a, r.b, r.g, r.final_distance);
else printf("%9.5f\t%9.5f\t%9.5f\t%9.5f\n", r.a, r.b, r.g, r.final_distance);
```

15. Stuff the command line arguments that should be constant over the entire inversion process into the measurement record and set up the result record to handle the arguments properly so that the optical properties can be determined.

```
⟨ Command-line changes to m 15 ⟩ ≡
  if (cl_cos_angle ≠ UNINITIALIZED) {
    m.slab_cos_angle = cl_cos_angle;
    cl_quadrature_points = 12 * (cl_quadrature_points/12);
    if (cl_quadrature_points < 12) cl_quadrature_points = 12;
  }
  if (cl_sample_n ≠ UNINITIALIZED) m.slab_index = cl_sample_n;
  if (cl_slide_n ≠ UNINITIALIZED) {
    m.slab_bottom_slide_index = cl_slide_n;
    m.slab_top_slide_index = cl_slide_n;
  }
  if (cl_slide_OD ≠ UNINITIALIZED) {
    m.slab_bottom_slide_b = cl_slide_OD;
    m.slab_top_slide_b = cl_slide_OD;
  }
  if (cl_sample_d ≠ UNINITIALIZED) m.slab_thickness = cl_sample_d;
  if (cl_beam_d ≠ UNINITIALIZED) m.d_beam = cl_beam_d;
  if (cl_slide_d ≠ UNINITIALIZED) {
    m.slab_bottom_slide_thickness = cl_slide_d;
    m.slab_top_slide_thickness = cl_slide_d;
  }
  if (cl_slides ≡ NO_SLIDES) {
    m.slab_bottom_slide_index = 1.0;
    m.slab_bottom_slide_thickness = 0.0;
    m.slab_top_slide_index = 1.0;
    m.slab_top_slide_thickness = 0.0;
  }
  if (cl_slides ≡ ONE_SLIDE_ON_TOP) {
    m.slab_bottom_slide_index = 1.0;
    m.slab_bottom_slide_thickness = 0.0;
  }
  if (cl_slides ≡ ONE_SLIDE_ON_BOTTOM) {
    m.slab_top_slide_index = 1.0;
    m.slab_top_slide_thickness = 0.0;
  }
  if (cl_method ≠ UNINITIALIZED) m.method = (int) cl_method;
  if (cl_rstd_t ≠ UNINITIALIZED) m.rstd_t = cl_rstd_t;
  if (cl_rstd_r ≠ UNINITIALIZED) m.rstd_r = cl_rstd_r;
  if (cl_sphere_one[4] ≠ UNINITIALIZED) {
    double d_sample_r, d_entrance_r, d_detector_r;
    m.d_sphere_r = cl_sphere_one[0];
    d_sample_r = cl_sphere_one[1];
    d_entrance_r = cl_sphere_one[2];
    d_detector_r = cl_sphere_one[3];
    m.rw_r = cl_sphere_one[4];
    m.as_r = (d_sample_r/m.d_sphere_r/2) * (d_sample_r/m.d_sphere_r/2);
    m.ae_r = (d_entrance_r/m.d_sphere_r/2) * (d_entrance_r/m.d_sphere_r/2);
    m.ad_r = (d_detector_r/m.d_sphere_r/2) * (d_detector_r/m.d_sphere_r/2);
    m.aw_r = 1.0 - m.as_r - m.ae_r - m.ad_r;
```

```
    m.d_sphere_t = m.d_sphere_r;
    m.as_t = m.as_r;
    m.ae_t = m.ae_r;
    m.ad_t = m.ad_r;
    m.aw_t = m.aw_r;
    m.rw_t = m.rw_r;
    if (cl_num_spheres ≡ UNINITIALIZED) m.num_spheres = 1;
  }
  if (cl_sphere_two[4] ≠ UNINITIALIZED) {
    double d_sample_t, d_entrance_t, d_detector_t;
    m.d_sphere_t = cl_sphere_two[0];
    d_sample_t = cl_sphere_two[1];
    d_entrance_t = cl_sphere_two[2];
    d_detector_t = cl_sphere_two[3];
    m.rw_t = cl_sphere_two[4];
    m.as_t = (d_sample_t/m.d_sphere_t/2) * (d_sample_t/m.d_sphere_t/2);
    m.ae_t = (d_entrance_t/m.d_sphere_t/2) * (d_entrance_t/m.d_sphere_t/2);
    m.ad_t = (d_detector_t/m.d_sphere_t/2) * (d_detector_t/m.d_sphere_t/2);
    m.aw_t = 1.0 - m.as_t - m.ae_t - m.ad_t;
    if (cl_num_spheres ≡ UNINITIALIZED) m.num_spheres = 2;
  }
  if (cl_num_spheres ≠ UNINITIALIZED) m.num_spheres = (int) cl_num_spheres;
  if ((cl_no_unscat ≡ MR_IS_ONLY_RD) ∨ (cl_no_unscat ≡ NO_UNSCATTERED_LIGHT)) m.sphere_with_rc = 0.0;
  if ((cl_no_unscat ≡ MT_IS_ONLY_TD) ∨ (cl_no_unscat ≡ NO_UNSCATTERED_LIGHT)) m.sphere_with_tc = 0.0;
  if (cl_UR1 ≠ UNINITIALIZED) m.m_r = cl_UR1;
  if (cl_UT1 ≠ UNINITIALIZED) m.m_t = cl_UT1;
  if (cl_Tc ≠ UNINITIALIZED) m.m_u = cl_Tc;
  if (cl_default_fr ≠ UNINITIALIZED) m.f_r = cl_default_fr;
```

This code is used in section 2.

16. put the values for command line reflection and transmission into the measurement record.

```
⟨ Count command-line measurements 16 ⟩ ≡
  m.num_measures = 3;
  if (m.m_t ≡ 0) m.num_measures--;
  if (m.m_u ≡ 0) m.num_measures--;
  params = m.num_measures;
  if (m.num_measures ≡ 3) {      /* need to fill slab entries to calculate the optical thickness */
    struct AD_slab_type s;
    s.n_slab = m.slab_index;
    s.n_top_slide = m.slab_top_slide_index;
    s.n_bottom_slide = m.slab_bottom_slide_index;
    s.b_top_slide = m.slab_top_slide_b;
    s.b_bottom_slide = m.slab_bottom_slide_b;
    s.cos_angle = m.slab_cos_angle;
    cl_default_b = What_Is_B(s, m.m_u);
  }
```

This code is used in section 2.

17. ⟨ print version function 17 ⟩ ≡

```
static void print_version(void)
{
  fprintf(stderr, "iad␣%s\n", Version);
  fprintf(stderr, "Copyright␣2011␣Scott␣Prahl,␣prahls@ohsu.edu\n");
  fprintf(stderr, "␣␣␣␣␣␣␣␣␣␣(see␣Applied␣Optics,␣32:559-568,␣1993)\n");
  exit(0);
}
```

This code is used in section 2.

18. ⟨ print usage function 18 ⟩ ≡

```
static void print_usage(void)
{
  fprintf(stderr, "iad␣%s\n\n", Version);
  fprintf(stderr, "iad␣finds␣optical␣properties␣from␣measurements\n\n");
  fprintf(stderr, "Usage:␣␣iad␣[options]␣input\n\n");
  fprintf(stderr, "Options:\n");
  fprintf(stderr, "␣␣-1␣'#␣#␣#␣#␣#'␣␣␣reflection␣sphere␣parameters␣\n");
  fprintf(stderr, "␣␣-2␣'#␣#␣#␣#␣#'␣␣␣transmission␣sphere␣parameters␣\n");
  fprintf(stderr, "␣␣␣␣␣␣'sphere␣d,␣sample␣d,␣entrance␣d,␣detector␣d,␣wall␣r'\n");
  fprintf(stderr, "␣␣-a␣#␣␣␣␣␣␣␣␣␣␣␣␣use␣this␣albedo␣\n");
  fprintf(stderr, "␣␣-A␣#␣␣␣␣␣␣␣␣␣␣␣␣use␣this␣absorption␣coefficient␣\n");
  fprintf(stderr, "␣␣-b␣#␣␣␣␣␣␣␣␣␣␣␣␣use␣this␣optical␣thickness␣\n");
  fprintf(stderr, "␣␣-B␣#␣␣␣␣␣␣␣␣␣␣␣␣beam␣diameter␣\n");
  fprintf(stderr, "␣␣-c␣#␣␣␣␣␣␣␣␣␣␣␣␣measurements␣have␣unscattered␣light?\n");
  fprintf(stderr, "␣␣-C␣␣␣␣␣␣␣␣␣␣␣␣␣␣do␣not␣correct␣for␣diffuse␣light␣loss\n");
  fprintf(stderr, "␣␣-d␣#␣␣␣␣␣␣␣␣␣␣␣␣thickness␣of␣sample␣\n");
  fprintf(stderr, "␣␣-D␣#␣␣␣␣␣␣␣␣␣␣␣␣thickness␣of␣slide␣\n");
  fprintf(stderr, "␣␣-e␣#␣␣␣␣␣␣␣␣␣␣␣␣error␣tolerance␣(default␣0.0001)␣\n");
  fprintf(stderr, "␣␣-E␣#␣␣␣␣␣␣␣␣␣␣␣␣optical␣depth␣(=mua*D)␣for␣slides\n");
  fprintf(stderr,
      "␣␣-f␣#␣␣␣␣␣␣␣␣␣␣␣␣allow␣a␣fraction␣0.0-1.0␣of␣light␣to␣hit␣sphere␣wall␣first\n");
  fprintf(stderr, "␣␣-F␣#␣␣␣␣␣␣␣␣␣␣␣␣use␣this␣scattering␣coefficient␣\n");
  fprintf(stderr, "␣␣-F␣'P␣lambda0␣mus0␣gamma'␣␣␣mus=mus0*(lambda/lambda0)^gamma\n");
  fprintf(stderr, "␣␣-F␣'R␣lambda0␣musp0␣gamma'␣␣musp=musp0*(lambda/lambda0)^gamma\n");
  fprintf(stderr, "␣␣-g␣#␣␣␣␣␣␣␣␣␣␣␣␣scattering␣anisotropy␣(default␣0)␣\n");
  fprintf(stderr, "␣␣-G␣#␣␣␣␣␣␣␣␣␣␣␣␣'t'␣(one␣top)␣or␣'b'␣(one␣bottom)␣slide\n");
  fprintf(stderr, "␣␣-h␣␣␣␣␣␣␣␣␣␣␣␣␣␣display␣help\n");
  fprintf(stderr, "␣␣-i␣#␣␣␣␣␣␣␣␣␣␣␣␣light␣is␣incident␣at␣this␣angle␣in␣degrees\n");
  fprintf(stderr, "␣␣-M␣#␣␣␣␣␣␣␣␣␣␣␣␣number␣of␣Monte␣Carlo␣iterations\n");
  fprintf(stderr, "␣␣-n␣#␣␣␣␣␣␣␣␣␣␣␣␣specify␣index␣of␣refraction␣of␣slab\n");
  fprintf(stderr, "␣␣-N␣#␣␣␣␣␣␣␣␣␣␣␣␣specify␣index␣of␣refraction␣of␣slides\n");
  fprintf(stderr, "␣␣-o␣filename␣␣␣␣␣␣explicitly␣specify␣filename␣for␣output\n");
  fprintf(stderr, "␣␣-p␣#␣␣␣␣␣␣␣␣␣␣␣␣#␣of␣Monte␣Carlo␣photons␣(default␣100000)\n");
  fprintf(stderr, "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣a␣negative␣number␣is␣max␣MC␣time␣in␣milliseconds\n");
  fprintf(stderr, "␣␣-q␣#␣␣␣␣␣␣␣␣␣␣␣␣number␣of␣quadrature␣points␣(default=8)\n");
  fprintf(stderr, "␣␣-r␣#␣␣␣␣␣␣␣␣␣␣␣␣total␣reflection␣measurement\n");
  fprintf(stderr, "␣␣-R␣#␣␣␣␣␣␣␣␣␣␣␣␣actual␣reflectance␣for␣100%%␣measurement␣\n");
  fprintf(stderr, "␣␣-S␣#␣␣␣␣␣␣␣␣␣␣␣␣number␣of␣spheres␣used\n");
  fprintf(stderr, "␣␣-t␣#␣␣␣␣␣␣␣␣␣␣␣␣total␣transmission␣measurement\n");
  fprintf(stderr, "␣␣-T␣#␣␣␣␣␣␣␣␣␣␣␣␣actual␣transmission␣for␣100%%␣measurement␣\n");
  fprintf(stderr, "␣␣-u␣#␣␣␣␣␣␣␣␣␣␣␣␣unscattered␣transmission␣measurement\n");
  fprintf(stderr, "␣␣-v␣␣␣␣␣␣␣␣␣␣␣␣␣␣version␣information\n");
  fprintf(stderr, "␣␣-V␣0␣␣␣␣␣␣␣␣␣␣␣␣verbosity␣low␣---␣no␣output␣to␣stderr\n");
  fprintf(stderr, "␣␣-V␣1␣␣␣␣␣␣␣␣␣␣␣␣verbosity␣moderate␣\n");
  fprintf(stderr, "␣␣-V␣2␣␣␣␣␣␣␣␣␣␣␣␣verbosity␣high\n");
  fprintf(stderr, "␣␣-x␣#␣␣␣␣␣␣␣␣␣␣␣␣set␣debugging␣level\n");
  fprintf(stderr, "␣␣-X␣␣␣␣␣␣␣␣␣␣␣␣␣␣dual␣beam␣configuration\n");
  fprintf(stderr, "␣␣-z␣␣␣␣␣␣␣␣␣␣␣␣␣␣do␣forward␣calculation\n");
  fprintf(stderr, "Examples:\n");
  fprintf(stderr, "␣␣iad␣data␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Optical␣values␣put␣in␣data.txt\n");
  fprintf(stderr, "␣␣iad␣-c␣1␣data␣␣␣␣␣␣␣␣␣␣␣␣␣Assume␣M_R␣has␣no␣unscattered␣reflectance\n");
```

```
    fprintf(stderr,
        "␣␣iad␣-c␣2␣data␣␣␣␣␣␣␣␣␣␣␣␣␣␣Assume␣M_T␣has␣no␣unscattered␣transmittance\n");
    fprintf(stderr,
        "␣␣iad␣-c␣3␣data␣␣␣␣␣␣␣␣␣␣␣␣␣␣Assume␣M_R␣&␣M_T␣have␣no␣unscattered␣light\n");
    fprintf(stderr,"␣␣iad␣-e␣0.0001␣data␣␣␣␣␣␣␣␣Better␣convergence␣to␣R␣&␣T␣values\n");
    fprintf(stderr,
        "␣␣iad␣-f␣1.0␣data␣␣␣␣␣␣␣␣␣␣␣␣All␣light␣hits␣reflectance␣sphere␣wall␣first\n");
    fprintf(stderr,"␣␣iad␣-o␣out␣data␣␣␣␣␣␣␣␣␣␣␣Calculated␣values␣in␣out\n");
    fprintf(stderr,"␣␣iad␣-r␣0.3␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣R_total=0.3,␣b=inf,␣find␣albedo\n");
    fprintf(stderr,"␣␣iad␣-r␣0.3␣-t␣0.4␣␣␣␣␣␣␣␣␣␣R_total=0.3,␣T_total=0.4,␣find␣a,b,g\n");
    fprintf(stderr,"␣␣iad␣-r␣0.3␣-t␣0.4␣-n␣1.5␣␣␣R_total=0.3,␣T_total=0.4,␣n=1.5,␣find␣a,b\n");
    fprintf(stderr,"␣␣iad␣-r␣0.3␣-t␣0.4␣␣␣␣␣␣␣␣␣␣R_total=0.3,␣T_total=0.4,␣find␣a,b\n");
    fprintf(stderr,"␣␣iad␣-p␣1000␣data␣␣␣␣␣␣␣␣␣␣Only␣1000␣photons\n");
    fprintf(stderr,"␣␣iad␣-p␣-100␣data␣␣␣␣␣␣␣␣␣␣Allow␣only␣100ms␣per␣iteration\n");
    fprintf(stderr,"␣␣iad␣-q␣4␣data␣␣␣␣␣␣␣␣␣␣␣␣Four␣quadrature␣points\n");
    fprintf(stderr,"␣␣iad␣-M␣0␣data␣␣␣␣␣␣␣␣␣␣␣␣No␣MC␣␣␣␣(iad)\n");
    fprintf(stderr,"␣␣iad␣-M␣1␣data␣␣␣␣␣␣␣␣␣␣␣␣MC␣once␣␣(iad␣->␣MC␣->␣iad)\n");
    fprintf(stderr,"␣␣iad␣-M␣2␣data␣␣␣␣␣␣␣␣␣␣␣␣MC␣twice␣(iad␣->␣MC␣->␣iad␣->␣MC␣->␣iad)\n");
    fprintf(stderr,"␣␣iad␣-M␣0␣-q␣4␣data␣␣␣␣␣␣␣␣Fast␣and␣crude␣conversion\n");
    fprintf(stderr,
        "␣␣iad␣-G␣t␣data␣␣␣␣␣␣␣␣␣␣␣␣␣One␣top␣slide␣with␣properties␣from␣data.rxt\n");
    fprintf(stderr,
        "␣␣iad␣-G␣b␣-N␣1.5␣-D␣1␣data␣Use␣1␣bottom␣slide␣with␣n=1.5␣and␣thickness=1\n");
    fprintf(stderr,"␣␣iad␣-x␣␣␣␣1␣data␣␣␣␣␣␣␣␣␣␣␣Show␣sphere␣and␣MC␣effects\n");
    fprintf(stderr,"␣␣iad␣-x␣␣␣␣2␣data␣␣␣␣␣␣␣␣␣␣␣DEBUG_GRID\n");
    fprintf(stderr,"␣␣iad␣-x␣␣␣␣4␣data␣␣␣␣␣␣␣␣␣␣␣DEBUG_ITERATIONS\n");
    fprintf(stderr,"␣␣iad␣-x␣␣␣␣8␣data␣␣␣␣␣␣␣␣␣␣␣DEBUG_LOST_LIGHT\n");
    fprintf(stderr,"␣␣iad␣-x␣␣␣16␣data␣␣␣␣␣␣␣␣␣␣␣DEBUG_SPHERE_EFFECTS\n");
    fprintf(stderr,"␣␣iad␣-x␣␣␣32␣data␣␣␣␣␣␣␣␣␣␣␣DEBUG_BEST_GUESS\n");
    fprintf(stderr,"␣␣iad␣-x␣␣␣64␣data␣␣␣␣␣␣␣␣␣␣␣DEBUG_EVERY_CALC\n");
    fprintf(stderr,"␣␣iad␣-x␣128␣data␣␣␣␣␣␣␣␣␣␣␣DEBUG_SEARCH\n");
    fprintf(stderr,"␣␣iad␣-x␣255␣data␣␣␣␣␣␣␣␣␣␣␣All␣debugging␣output\n");
    fprintf(stderr,
        "␣␣iad␣-X␣-i␣8␣data␣␣␣␣␣␣␣␣␣Dual␣beam␣spectrometer␣with␣8␣degree␣incidence\n\n");
    fprintf(stderr,
        "␣␣iad␣-z␣-a␣0.9␣-b␣1␣-i␣45␣␣Forward␣calc␣assuming␣45␣degree␣incidence\n\n");
    fprintf(stderr,"␣␣apply␣iad␣data1␣data2␣␣␣␣␣Process␣multiple␣files\n\n");
    fprintf(stderr,"Report␣bugs␣to␣<prahls@ohsu.edu>\n\n");
    exit(0);
  }
```

This code is used in section 2.

19. Just figure out the damn scattering and absorption

```
⟨ calculate coefficients function 19 ⟩ ≡
  static void Calculate_Mua_Musp(struct measure_type m, struct invert_type r, double *musp, double
      *mua)
  {
    if (r.default_b ≡ HUGE_VAL ∨ r.b ≡ HUGE_VAL) {
      if (r.a ≡ 0) {
        *musp = 0.0;
        *mua = 1.0;
        return;
      }
      *musp = 1.0 − r.g;
      *mua = (1.0 − r.a)/r.a;
      return;
    }
    *musp = r.a * r.b/m.slab_thickness * (1.0 − r.g);
    *mua = (1 − r.a) * r.b/m.slab_thickness;
  }
```

See also section 20.

This code is used in section 2.

20. This can only be called immediately after *Invert_RT* You have been warned! Notice that *Calculate_Distance* does not pass any slab properties.

```
⟨ calculate coefficients function 19 ⟩ +≡
  static void calculate_coefficients(struct measure_type m, struct invert_type r, double *LR, double
      *LT, double *musp, double *mua)
  {
    double delta;
    *LR = 0;
    *LT = 0;
    Calculate_Distance(LR, LT, &delta);
    Calculate_Mua_Musp(m, r, musp, mua);
  }
```

21. ⟨ print results header function 21 ⟩ ≡

```
static void print_results_header(FILE *fp)
{
  fprintf(fp,"#␣␣␣␣␣\tMeasured␣\t␣␣␣M_R␣␣␣\tMeasured␣\t␣␣␣M_T␣␣␣\tEstimat\
      ed\tEstimated\tEstimated");
  if (Debug(DEBUG_LOST_LIGHT)) fprintf(fp,
      "\t␣␣Lost␣␣␣\t␣␣Lost␣␣␣\t␣␣Lost␣␣␣\t␣␣Lost␣␣␣\t␣␣␣MC␣␣␣␣\t␣␣␣IAD␣␣␣\t␣␣Error␣␣");
  fprintf(fp,"\n");
  fprintf(fp,"##wave\t␣␣␣M_R␣␣␣\t␣␣␣fit␣␣␣\t␣␣␣M_T␣␣␣\t␣␣␣fit␣␣␣\t␣␣mu_a␣\
      ␣␣\t␣␣mu_s'␣␣\t␣␣␣␣g␣␣␣␣");
  if (Debug(DEBUG_LOST_LIGHT)) fprintf(fp,
      "\t␣␣␣UR1␣␣␣\t␣␣␣URU␣␣␣\t␣␣␣UT1␣␣␣\t␣␣␣UTU␣␣␣\t␣␣␣␣#␣␣␣␣\t␣␣␣␣#␣␣␣␣\t␣␣State␣␣");
  fprintf(fp,"\n");
  fprintf(fp,"#␣[nm]\t␣␣[---]␣␣\t␣␣[---]␣␣\t␣␣[---]␣␣\t␣␣[---]␣␣\t␣␣1/mm␣\
      ␣␣\t␣␣1/mm␣␣␣\t␣␣[---]␣␣");
  if (Debug(DEBUG_LOST_LIGHT)) fprintf(fp,
      "\t␣␣[---]␣␣\t␣␣[---]␣␣\t␣␣[---]␣␣\t␣␣[---]␣␣\t␣␣[---]␣␣\t␣␣[---]␣␣\t␣␣[---]␣␣");
  fprintf(fp,"\n");
}
```

This code is used in section 2.

22. When debugging lost light, it is handy to see how each iteration changes the calculated values for the optical properties. We do that here if we are debugging, otherwise we just print a number or something to keep the user from wondering what is going on.

⟨ Print results function 22 ⟩ ≡

```
void print_optical_property_result (FILE *fp, struct measure_type m, struct invert_type r, double
    LR, double LT, double mu_a, double mu_sp, int mc_iter, int line ) {
if (m.lambda ≠ 0) fprintf(fp,"%6.1f\t",m.lambda);
else fprintf (fp,"%6d\t", line ) ;
if (mu_a > 200) mu_a = 199.9999;
if (mu_sp > 1000) mu_sp = 999.9999;
fprintf(fp,"%0.3e\t%0.3e\t",m.m_r,LR);
fprintf(fp,"%0.3e\t%0.3e\t",m.m_t,LT);
fprintf(fp,"%0.3e\t",mu_a);
fprintf(fp,"%0.3e\t",mu_sp);
fprintf(fp,"%0.3e\t",r.g);
if (Debug(DEBUG_LOST_LIGHT)) {
  fprintf(fp,"%0.3e\t%0.3e\t",m.ur1_lost,m.uru_lost);
  fprintf(fp,"%0.3e\t%0.3e\t",m.ut1_lost,m.utu_lost);
  fprintf(fp,"␣%2d␣␣\t",mc_iter);
  fprintf(fp,"␣%4d\t",r.iterations);
}
fprintf (fp,"#␣%c␣\n", what_char ( r . error ) ) ;
fflush(fp); }
```

This code is used in section 2.

23. ⟨ print error legend function 23 ⟩ ≡

```
static void print_error_legend(void)
{
  fprintf(stderr, "-----------------␣Sorry,␣but␣...␣errors␣encountered␣---------------\n");
  fprintf(stderr, "␣␣␣*␣␣==>␣Success␣␣␣␣␣␣␣␣␣␣");
  fprintf(stderr, "␣␣0-9␣==>␣Monte␣Carlo␣Iteration\n");
  fprintf(stderr, "␣␣␣R␣␣==>␣M_R␣is␣too␣big␣␣␣");
  fprintf(stderr, "␣␣␣r␣␣==>␣M_R␣is␣too␣small\n");
  fprintf(stderr, "␣␣␣T␣␣==>␣M_T␣is␣too␣big␣␣␣");
  fprintf(stderr, "␣␣␣t␣␣==>␣M_T␣is␣too␣small\n");
  fprintf(stderr, "␣␣␣U␣␣==>␣M_U␣is␣too␣big␣␣␣");
  fprintf(stderr, "␣␣␣u␣␣==>␣M_U␣is␣too␣small\n");
  fprintf(stderr, "␣␣␣!␣␣==>␣M_R␣+␣M_T␣>␣1␣␣␣␣");
  fprintf(stderr, "␣␣␣+␣␣==>␣Did␣not␣converge\n\n");
}
```

This code is used in section 2.

24. returns a new string consisting of s+t

⟨ stringdup together function 24 ⟩ ≡

```
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 2.

25. assume that start time has already been set

⟨ seconds elapsed function 25 ⟩ ≡

```
static double seconds_elapsed(clock_t start_time)
{
  clock_t finish_time = clock( );
  return (double)(finish_time − start_time)/CLOCKS_PER_SEC;
}
```

This code is used in section 2.

26. given a string and an array, this fills the array with numbers from the string. The numbers should be separated by spaces.

Returns 0 upon successfully filling n entries, returns 1 for any error.

```
⟨ parse string into array function 26 ⟩ ≡
  static int parse_string_into_array(char *s, double *a, int n)
  {
    char *t, *last, *r;
    int i = 0;
    t = s;
    last = s + strlen(s);
    while (t < last) {      /* a space should mark the end of number */
      r = t;
      while (*r ≠ ' ' ∧ *r ≠ '\0') r++;
      *r = '\0';      /* parse the number and save it */
      if (sscanf(t, "%lf", &(a[i])) ≡ 0) return 1;
      i++;      /* are we done ? */
      if (i ≡ n) return 0;      /* move pointer just after last number */
      t = r + 1;
    }
    return 1;
  }
```

This code is used in section 2.

27. ⟨ print dot function 27 ⟩ ≡

```
static char what_char(int err)
{
  if (err ≡ IAD_NO_ERROR) return '*';
  if (err ≡ IAD_TOO_MANY_ITERATIONS) return '+';
  if (err ≡ IAD_MR_TOO_BIG) return 'R';
  if (err ≡ IAD_MR_TOO_SMALL) return 'r';
  if (err ≡ IAD_MT_TOO_BIG) return 'T';
  if (err ≡ IAD_MT_TOO_SMALL) return 't';
  if (err ≡ IAD_MU_TOO_BIG) return 'U';
  if (err ≡ IAD_MU_TOO_SMALL) return 'u';
  if (err ≡ IAD_TOO_MUCH_LIGHT) return '!';
  return '?';
}
static void print_dot(clock_t start_time, int err, int count, int points, int final, int verbosity, int
    *any_error)
{
  static int counter = 0;
  counter++;
  if (err ≠ IAD_NO_ERROR) *any_error = err;
  if (verbosity ≡ 0) return;
  if (final ≡ 99) fprintf(stderr, "%c", what_char(err));
  else {
    counter--;
    fprintf(stderr, "%1d\b", final % 10);
  }
  if (final ≡ 99) {
    if (counter % 50 ≡ 0) {
      double rate = (seconds_elapsed(start_time)/points);
      fprintf(stderr, "␣␣%3d␣done␣(%5.2f␣s/pt)\n", points, rate);
    }
    else if (counter % 10 ≡ 0) fprintf(stderr, "␣");
  }
  fflush(stderr);
}
```

This code is used in section 2.

28. IAD Types. This file has no routines. It is responsible for creating the header file `iad_type.h` and nothing else. Altered 3/3/95 to change the version number below. Change June 95 to improve cross referencing using CTwill. Change August 97 to add root finding with known absorption

29. These are the various optical properties that can be found with this program. `FIND_AUTO` allows one to let the computer figure out what it should be looking for.

These determine what metric is used in the minimization process.

These give the two different types of illumination allowed.

Finally, for convenience I create a Boolean type.

⟨ `iad_type.h` 29 ⟩ ≡

```
#undef FALSE
#undef TRUE
  ⟨ Preprocessor definitions ⟩
  ⟨ Structs to export from IAD Types 32 ⟩
```

30.

```
#define FIND_A 0
#define FIND_B 1
#define FIND_AB 2
#define FIND_AG 3
#define FIND_AUTO 4
#define FIND_BG 5
#define FIND_BaG 6
#define FIND_BsG 7
#define FIND_Ba 8
#define FIND_Bs 9
#define FIND_G 10
#define FIND_B_WITH_NO_ABSORPTION 11
#define FIND_B_WITH_NO_SCATTERING 12
#define RELATIVE 0
#define ABSOLUTE 1
#define COLLIMATED 0
#define DIFFUSE 1
#define FALSE 0
#define TRUE 1
#define IAD_MAX_ITERATIONS 500
```

31. Need error codes for this silly program

```
#define IAD_NO_ERROR 0
#define IAD_TOO_MANY_ITERATIONS 1
#define IAD_AS_NOT_VALID 16
#define IAD_AE_NOT_VALID 17
#define IAD_AD_NOT_VALID 18
#define IAD_RW_NOT_VALID 19
#define IAD_RD_NOT_VALID 20
#define IAD_RSTD_NOT_VALID 21
#define IAD_GAMMA_NOT_VALID 22
#define IAD_F_NOT_VALID 23
#define IAD_BAD_PHASE_FUNCTION 24
#define IAD_QUAD_PTS_NOT_VALID 25
#define IAD_BAD_G_VALUE 26
#define IAD_TOO_MANY_LAYERS 27
#define IAD_MEMORY_ERROR 28
#define IAD_FILE_ERROR 29
#define IAD_EXCESSIVE_LIGHT_LOSS 30
#define IAD_RT_LT_MINIMUM 31
#define IAD_MR_TOO_SMALL 32
#define IAD_MR_TOO_BIG 33
#define IAD_MT_TOO_SMALL 34
#define IAD_MT_TOO_BIG 35
#define IAD_MU_TOO_SMALL 36
#define IAD_MU_TOO_BIG 37
#define IAD_TOO_MUCH_LIGHT 38
#define IAD_TSTD_NOT_VALID 39
#define UNINITIALIZED -99
#define DEBUG_A_LITTLE 1
#define DEBUG_GRID 2
#define DEBUG_ITERATIONS 4
#define DEBUG_LOST_LIGHT 8
#define DEBUG_SPHERE_EFFECTS 16
#define DEBUG_BEST_GUESS 32
#define DEBUG_EVERY_CALC 64
#define DEBUG_SEARCH 128
#define DEBUG_RD_ONLY 256
#define DEBUG_ANY #FFFFFFFF
#define UNKNOWN 0
#define COMPARISON 1
#define SUBSTITUTION 2
```

32. The idea of the structure *measure_type* is collect all the information regarding a single measurement together in one spot. No information regarding how the inversion procedure is supposed to be done is contained in this structure, unlike in previous incarnations of this program.

⟨ Structs to export from IAD Types 32 ⟩ ≡

```
typedef struct measure_type {
  double slab_index;
  double slab_thickness;
  double slab_top_slide_index;
  double slab_top_slide_b;
  double slab_top_slide_thickness;
  double slab_bottom_slide_index;
  double slab_bottom_slide_b;
  double slab_bottom_slide_thickness;
  double slab_cos_angle;
  int num_spheres;
  int num_measures;
  int method;
  double d_beam;
  double sphere_with_rc;
  double sphere_with_tc;
  double m_r, m_t, m_u;
  double lambda;
  double as_r, ad_r, ae_r, aw_r, rd_r, rw_r, rstd_r, f_r;
  double as_t, ad_t, ae_t, aw_t, rd_t, rw_t, rstd_t, f_t;
  double ur1_lost, uru_lost, ut1_lost, utu_lost;
  double d_sphere_r, d_sphere_t;
} IAD_measure_type;
```

See also sections 33 and 34.

This code is used in section 29.

33. This describes how the inversion process should proceed and also contains the results of that inversion process.

```
⟨ Structs to export from IAD Types 32 ⟩ +≡
  typedef struct invert_type { double a;      /* the calculated albedo */
  double b;      /* the calculated optical depth */
  double g;      /* the calculated anisotropy */
  int found;
  int search;
  int metric;
  double tolerance;
  double MC_tolerance;
  double final_distance;
  int iterations; int error ;
  struct AD_slab_type slab;
  struct AD_method_type method;
  double default_a;
  double default_b;
  double default_g;
  double default_ba;
  double default_bs;
  double default_mua;
  double default_mus; } IAD_invert_type;
```

34. A few types that used to be enum's are now int's.

```
⟨ Structs to export from IAD Types 32 ⟩ +≡
  typedef int search_type;
  typedef int boolean_type;
  typedef int illumination_type;
  typedef struct guess_t {
    double distance;
    double a;
    double b;
    double g;
  } guess_type;
  extern double FRACTION;
```

35. IAD Public.

This contains the routine *Inverse_RT* that should generally be the basic entry point into this whole mess. Call this routine with the proper values and true happiness is bound to be yours.

Altered accuracy of the standard method of root finding from 0.001 to 0.00001. Note, it really doesn't help to change the method from `ABSOLUTE` to `RELATIVE`, but I did anyway. (3/3/95)

⟨ `iad_pub.c` 35 ⟩ ≡

```
#include <stdio.h>
#include <math.h>
#include "nr_util.h"
#include "ad_globl.h"
#include "ad_frsnl.h"
#include "iad_type.h"
#include "iad_util.h"
#include "iad_calc.h"
#include "iad_find.h"
#include "iad_pub.h"
#include "iad_io.h"
#include "mc_lost.h"
```

⟨ Definition for *Inverse_RT* 39 ⟩
⟨ Definition for *measure_OK* 44 ⟩
⟨ Definition for *determine_search* 51 ⟩
⟨ Definition for *Initialize_Result* 55 ⟩
⟨ Definition for *Initialize_Measure* 63 ⟩
⟨ Definition for *ez_Inverse_RT* 61 ⟩
⟨ Definition for *Spheres_Inverse_RT* 65 ⟩
⟨ Definition for *Calculate_MR_MT* 72 ⟩
⟨ Definition for *MinMax_MR_MT* 76 ⟩
⟨ Definition for *Calculate_Minimum_MR* 74 ⟩

36. All the information that needs to be written to the header file `iad_pub.h`. This eliminates the need to maintain a set of header files as well.

⟨ `iad_pub.h` 36 ⟩ ≡

⟨ Prototype for *Inverse_RT* 38 ⟩;
⟨ Prototype for *measure_OK* 43 ⟩;
⟨ Prototype for *determine_search* 50 ⟩;
⟨ Prototype for *Initialize_Result* 54 ⟩;
⟨ Prototype for *ez_Inverse_RT* 60 ⟩;
⟨ Prototype for *Initialize_Measure* 62 ⟩;
⟨ Prototype for *Calculate_MR_MT* 71 ⟩;
⟨ Prototype for *MinMax_MR_MT* 75 ⟩;
⟨ Prototype for *Calculate_Minimum_MR* 73 ⟩;

37. Here is the header file needed to access one interesting routine in the `libiad.so` library.

⟨ `lib_iad.h` 37 ⟩ ≡

⟨ Prototype for *ez_Inverse_RT* 60 ⟩;
⟨ Prototype for *Spheres_Inverse_RT* 64 ⟩;

38. Inverse RT. *Inverse_RT* is the main function in this whole package. You pass the variable m containing your experimentally measured values to the function *Inverse_RT*. It hopefully returns the optical properties in r that are appropriate for your experiment.

```
⟨ Prototype for Inverse_RT 38 ⟩ ≡
  void Inverse_RT(struct measure_type m, struct invert_type *r)
```

This code is used in sections 36 and 39.

39.
```
⟨ Definition for Inverse_RT 39 ⟩ ≡
  ⟨ Prototype for Inverse_RT 38 ⟩
  {
    if (0 ∧ Debug(DEBUG_LOST_LIGHT)) {
      fprintf(stderr, "** Inverse_RT (%d spheres) **\n", m.num_spheres);
      fprintf(stderr, "    M_R       =%8.5f, MT        =%8.5f\n", m.m_r, m.m_t);
      fprintf(stderr, "    UR1 lost  =%8.5f, UT1 lost  =%8.5f\n", m.ur1_lost, m.ut1_lost);
    }
    r→found = FALSE;
    ⟨ Exit with bad input data 40 ⟩
    r→search = determine_search(m, *r);
    if (r→search ≡ FIND_B_WITH_NO_ABSORPTION) {
      r→default_a = 1;
      r→search = FIND_B;
    }
    if (r→search ≡ FIND_B_WITH_NO_SCATTERING) {
      r→default_a = 0;
      r→search = FIND_B;
    }
    ⟨ Find the optical properties 41 ⟩
    if (r→final_distance ≤ r→tolerance) r→found = TRUE;
  }
```

This code is used in section 35.

40. There is no sense going to all the trouble to try a multivariable minimization if the input data is bogus. So I wrote a single routine *measure_OK* to do just this.

```
⟨ Exit with bad input data 40 ⟩ ≡
  r → error = measure_OK(m, *r); if (r→method.quad_pts < 4) r → error = IAD_QUAD_PTS_NOT_VALID; if
      ( 0 ∧ ( r → error ≠ IAD_NO_ERROR ) ) return;
```

This code is used in section 39.

41. Now I fob the real work off to the unconstrained minimization routines. Ultimately, I would like to replace all these by constrained minimization routines. Actually the first five already are constrained. The real work will be improving the last five because these are 2-D minimization routines.

```
⟨ Find the optical properties 41 ⟩ ≡
  switch (r→search) {
  case FIND_A: U_Find_A(m, r);
    break;
  case FIND_B: U_Find_B(m, r);
    break;
  case FIND_G: U_Find_G(m, r);
    break;
  case FIND_Ba: U_Find_Ba(m, r);
    break;
  case FIND_Bs: U_Find_Bs(m, r);
    break;
  case FIND_AB: U_Find_AB(m, r);
    break;
  case FIND_AG: U_Find_AG(m, r);
    break;
  case FIND_BG: U_Find_BG(m, r);
    break;
  case FIND_BsG: U_Find_BsG(m, r);
    break;
  case FIND_BaG: U_Find_BaG(m, r);
    break;
  }
  if (r→iterations ≡ IAD_MAX_ITERATIONS) r → error = IAD_TOO_MANY_ITERATIONS;
```

This code is used in section 39.

42. Validation.

43. Now the question is — just what is bad data? Here's the prototype.

```
⟨ Prototype for measure_OK 43 ⟩ ≡
  int measure_OK(struct measure_type m, struct invert_type r)
```

This code is used in sections 36 and 44.

44. It would just be nice to stop computing with bad data. This does not work in practice becasue it turns out that there is often bogus data in a full wavelength scan. Often the reflectance is too low for short wavelengths and at long wavelengths the detector (photomultiplier tube) does not work worth a damn.

The two sphere checks are more complicated. For example, we can no longer categorically state that the transmittance is less than one or that the sum of the reflectance and transmittance is less than one. Instead we use the transmittance to bound the values for the reflectance — see the routine *MinMax_MR_MT* below.

```
⟨ Definition for measure_OK 44 ⟩ ≡
  ⟨ Prototype for measure_OK 43 ⟩{ double ru, tu;
      if (m.num_spheres ≠ 2) {
        ⟨ Check MR for zero or one spheres 45 ⟩
        ⟨ Check MT for zero or one spheres 46 ⟩
      }
      else { int error = MinMax_MR_MT(m, r); if ( error ≠ IAD_NO_ERROR ) return error ; } ⟨ Check
          MU 47 ⟩
      if (m.num_spheres ≠ 0) {
        ⟨ Check sphere parameters 48 ⟩
      }
      return IAD_NO_ERROR; }
```

This code is used in section 35.

45. The reflectance is constrained by the index of refraction of the material and the transmission. The upper bound for the reflectance is just one minus the transmittance. The specular (unscattered) reflectance from the boundaries imposes minimum for the reflectance. Obviously, the reflected light cannot be less than that from the first boundary. This might be calculated by assuming an infinite layer thickness. But we can do better.

There is a definite bound on the minimum reflectance from a sample. If you have a sample with a given transmittance m_t, the minimum reflectance possible is found by assuming that the sample does not scatter any light.

Knowledge of the indicies of refraction makes it a relatively simple matter to determine the optical thickness $b = mu\_a * d$ of the slab. The minimum reflection is obtained by including all the specular reflectances from all the surfaces.

If the default albedo has been specified as zero, then there is really no need to check `MR` because it is ignored.

```
⟨ Check MR for zero or one spheres 45 ⟩ ≡
  if (r.default_a ≡ UNINITIALIZED ∨ r.default_a > 0) {
    double mr, mt;
    Calculate_Minimum_MR(m, r, &mr, &mt);
    if (m.m_r < mr) return IAD_MR_TOO_SMALL;
  }
```

This code is used in section 44.

46. The transmittance is also constrained by the index of refraction of the material. The minimum transmittance is zero, but the maximum transmittance cannot exceed the total light passing through the sample when there is no scattering or absorption. This is calculated by assuming an infinitely thin (to eliminate any scattering or absorption effects).

```
⟨ Check MT for zero or one spheres 46 ⟩ ≡
  if (m.m_t < 0) return IAD_MT_TOO_SMALL;
  Sp_mu_RT(r.slab.n_top_slide, r.slab.n_slab, r.slab.n_bottom_slide, r.slab.b_top_slide, 0,
      r.slab.b_bottom_slide, r.slab.cos_angle, &ru, &tu);
  if (m.m_t > tu) {
    fprintf(stderr, "tu=%7.5f, m_t=%7.5f, t_std=%7.5f\n", tu, m.m_t, m.rstd_t);
    return IAD_MT_TOO_BIG;
  }
```

This code is used in section 44.

47. The unscattered transmission is now always included in the total transmittance. Therefore the unscattered transmittance must fall betwee zero and `M_T`

```
⟨ Check MU 47 ⟩ ≡
  if (m.m_u < 0) return IAD_MU_TOO_SMALL;
  if (m.m_u > m.m_t) return IAD_MU_TOO_BIG;
```

This code is used in section 44.

48. Make sure that reflection sphere parameters are reasonable

```
⟨ Check sphere parameters 48 ⟩ ≡
  if (m.as_r < 0 ∨ m.as_r ≥ 0.2) return IAD_AS_NOT_VALID;
  if (m.ad_r < 0 ∨ m.ad_r ≥ 0.2) return IAD_AD_NOT_VALID;
  if (m.ae_r < 0 ∨ m.ae_r ≥ 0.2) return IAD_AE_NOT_VALID;
  if (m.rw_r < 0 ∨ m.rw_r > 1.0) return IAD_RW_NOT_VALID;
  if (m.rd_r < 0 ∨ m.rd_r > 1.0) return IAD_RD_NOT_VALID;
  if (m.rstd_r < 0 ∨ m.rstd_r > 1.0) return IAD_RSTD_NOT_VALID;
  if (m.rstd_t < 0 ∨ m.rstd_t > 1.0) return IAD_TSTD_NOT_VALID;
  if (m.f_r < 0 ∨ m.f_r > 1) return IAD_F_NOT_VALID;
```

See also section 49.

This code is used in section 44.

49. Make sure that transmission sphere parameters are reasonable

```
⟨ Check sphere parameters 48 ⟩ +≡
  if (m.as_t < 0 ∨ m.as_t ≥ 0.2) return IAD_AS_NOT_VALID;
  if (m.ad_t < 0 ∨ m.ad_t ≥ 0.2) return IAD_AD_NOT_VALID;
  if (m.ae_t < 0 ∨ m.ae_t ≥ 0.2) return IAD_AE_NOT_VALID;
  if (m.rw_t < 0 ∨ m.rw_r > 1.0) return IAD_RW_NOT_VALID;
  if (m.rd_t < 0 ∨ m.rd_t > 1.0) return IAD_RD_NOT_VALID;
  if (m.rstd_t < 0 ∨ m.rstd_t > 1.0) return IAD_TSTD_NOT_VALID;
  if (m.f_t < 0 ∨ m.f_t > 1) return IAD_F_NOT_VALID;
```

50. Searching Method.

The original idea was that this routine would automatically determine what optical parameters could be figured out from the input data. This worked fine for a long while, but I discovered that often it was convenient to constrain the optical properties in various ways. Consequently, this routine got more and more complicated.

What should be done is to figure out whether the search will be 1D or 2D and split this routine into two parts.

It would be nice to enable the user to constrain two parameters, but the infrastructure is missing at this point.

⟨ Prototype for *determine_search* 50 ⟩ ≡

search_type *determine_search*(**struct measure_type** m, **struct invert_type** r)

This code is used in sections 36 and 51.

51. This routine is responsible for selecting the appropriate optical properties to determine.

```
⟨ Definition for determine_search 51 ⟩ ≡
  ⟨ Prototype for determine_search 50 ⟩
  {
    double rt, tt, rd, td, tc, rc;
    int search = 0;
    int independent = m.num_measures;
    Estimate_RT(m, r, &rt, &tt, &rd, &rc, &td, &tc);
    if (tc ≡ 0 ∧ m.m_u ≡ 0 ∧ independent ≡ 3)      /* no information in tc */
      independent −−;
    if (rd ≡ 0 ∧ independent ≡ 2)      /* no information in rd */
      independent −−;
    if (td ≡ 0 ∧ independent ≡ 2)      /* no information in td */
      independent −−;
    if (independent ≡ 1) {
      ⟨ One parameter search 52 ⟩
    }
    else if (independent ≡ 2) {
      ⟨ Two parameter search 53 ⟩
    }     /* three real parameters with information! */
    else {
      search = FIND_AG;
    }
    if (Debug(DEBUG_SEARCH)) {
      fprintf(stderr, "\n***␣Determine_Search()\n");
      fprintf(stderr, "␣␣␣␣independent␣measurements␣=␣%3d\n", independent);
      fprintf(stderr, "␣␣␣␣m_r=%8.5f␣m_t=%8.5f␣(rd␣=␣%8.5f␣td=%8.5f)\n", m.m_r, m.m_t, rd, td);
      if (search ≡ FIND_A) fprintf(stderr, "␣␣␣␣search␣=␣FIND_A\n");
      if (search ≡ FIND_B) fprintf(stderr, "␣␣␣␣search␣=␣FIND_B\n");
      if (search ≡ FIND_AB) fprintf(stderr, "␣␣␣␣search␣=␣FIND_AB\n");
      if (search ≡ FIND_AG) fprintf(stderr, "␣␣␣␣search␣=␣FIND_AG\n");
      if (search ≡ FIND_AUTO) fprintf(stderr, "␣␣␣␣search␣=␣FIND_AUTO\n");
      if (search ≡ FIND_BG) fprintf(stderr, "␣␣␣␣search␣=␣FIND_BG\n");
      if (search ≡ FIND_BaG) fprintf(stderr, "␣␣␣␣search␣=␣FIND_BaG\n");
      if (search ≡ FIND_BsG) fprintf(stderr, "␣␣␣␣search␣=␣FIND_BsG\n");
      if (search ≡ FIND_Ba) fprintf(stderr, "␣␣␣␣search␣=␣FIND_Ba\n");
      if (search ≡ FIND_Bs) fprintf(stderr, "␣␣␣␣search␣=␣FIND_Bs\n");
      if (search ≡ FIND_G) fprintf(stderr, "␣␣␣␣search␣=␣FIND_G\n");
      if (search ≡ FIND_B_WITH_NO_ABSORPTION)
        fprintf(stderr, "␣␣␣␣search␣=␣FIND_B_WITH_NO_ABSORPTION\n");
      if (search ≡ FIND_B_WITH_NO_SCATTERING)
        fprintf(stderr, "␣␣␣␣search␣=␣FIND_B_WITH_NO_SCATTERING\n");
    }
    return search;
  }
```

This code is used in section 35.

52. The fastest inverse problems are those in which just one measurement is known. This corresponds to a simple one-dimensional minimization problem. The only complexity is deciding exactly what should be allowed to vary. The basic assumption is that the anisotropy has been specified or will be assumed to be zero.

If the anistropy is assumed known, then one other assumption will allow us to figure out the last parameter to solve for.

Ultimately, if no default values are given, then we look at the value of the total transmittance. If this is zero, then we assume that the optical thickness is infinite and solve for the albedo. Otherwise we will just make a stab at solving for the optical thickness assuming the albedo is one.

```
⟨ One parameter search 52 ⟩ ≡
  if (r.default_a ≠ UNINITIALIZED) {
    if (r.default_a ≡ 0) search = FIND_B_WITH_NO_SCATTERING;
    else if (r.default_a ≡ 1) search = FIND_B_WITH_NO_ABSORPTION;
    else search = FIND_B;
  }
  else if (r.default_b ≠ UNINITIALIZED) search = FIND_A;
  else if (r.default_bs ≠ UNINITIALIZED) search = FIND_Ba;
  else if (r.default_ba ≠ UNINITIALIZED) search = FIND_Bs;
  else if (td ≡ 0) search = FIND_A;
  else if (rd ≡ 0) search = FIND_B_WITH_NO_SCATTERING;
  else search = FIND_B_WITH_NO_ABSORPTION;
```

This code is used in section 51.

53. If the absorption depth $\mu_a d$ is constrained return *FIND_BsG*. Recall that I use the bizarre mnemonic $bs = \mu_s d$ here and so this means that the program will search over various values of $\mu_s d$ and g.

If there are just two measurements then I assume that the anisotropy is not of interest and the only thing to calculate is the reduced albedo and optical thickness based on an assumed anisotropy.

```
⟨ Two parameter search 53 ⟩ ≡
  if (r.default_a ≠ UNINITIALIZED) {
    if ((r.default_a ≡ 0) ∨ (r.default_g ≠ UNINITIALIZED)) search = FIND_B;
    else search = FIND_BG;
  }
  else if (r.default_b ≠ UNINITIALIZED) {
    if (r.default_g ≠ UNINITIALIZED) search = FIND_A;
    else search = FIND_AG;
  }
  else if (r.default_ba ≠ UNINITIALIZED) {
    if (r.default_g ≠ UNINITIALIZED) search = FIND_Bs;
    else search = FIND_BsG;
  }
  else if (r.default_bs ≠ UNINITIALIZED) {
    if (r.default_g ≠ UNINITIALIZED) search = FIND_Ba;
    else search = FIND_BaG;
  }
  else if (rt + tt > 1 ∧ 0 ∧ m.num_spheres ≠ 2) search = FIND_B_WITH_NO_ABSORPTION;
  else search = FIND_AB;
```

This code is used in section 51.

54. This little routine just stuffs reasonable values into the structure we use to return the solution. This does not replace the values for *r.default_g* nor for *r.method.quad_pts*. Presumably these have been set correctly elsewhere.

```
⟨ Prototype for Initialize_Result 54 ⟩ ≡
  void Initialize_Result(struct measure_type m, struct invert_type *r)
```

This code is used in sections 36 and 55.

55.
```
⟨ Definition for Initialize_Result 55 ⟩ ≡
  ⟨ Prototype for Initialize_Result 54 ⟩
  {
    ⟨ Fill r with reasonable values 56 ⟩
  }
```

This code is used in section 35.

56. Start with the optical properties.

```
⟨ Fill r with reasonable values 56 ⟩ ≡
  r→a = 0.0;
  r→b = 0.0;
  r→g = 0.0;
```

See also sections 57, 58, and 59.

This code is used in section 55.

57. Continue with other useful stuff.

```
⟨ Fill r with reasonable values 56 ⟩ +≡
  r→found = FALSE;
  r→tolerance = 0.0001;
  r→MC_tolerance = 0.01;      /* percent */
  r→search = FIND_AUTO;
  r→metric = RELATIVE;
  r→final_distance = 10;
  r→iterations = 0; r → error = IAD_NO_ERROR;
```

58. The defaults might be handy

```
⟨ Fill r with reasonable values 56 ⟩ +≡
  r→default_a = UNINITIALIZED;
  r→default_b = UNINITIALIZED;
  r→default_g = UNINITIALIZED;
  r→default_ba = UNINITIALIZED;
  r→default_bs = UNINITIALIZED;
  r→default_mua = UNINITIALIZED;
  r→default_mus = UNINITIALIZED;
```

59. It is necessary to set up the slab correctly so, I stuff reasonable values into this record as well.

```
⟨ Fill r with reasonable values 56 ⟩ +≡
  r→slab.a = 0.5;
  r→slab.b = 1.0;
  r→slab.g = 0;
  r→slab.phase_function = HENYEY_GREENSTEIN;
  r→slab.n_slab = m.slab_index;
  r→slab.n_top_slide = m.slab_top_slide_index;
  r→slab.n_bottom_slide = m.slab_bottom_slide_index;
  r→slab.b_top_slide = m.slab_top_slide_b;
  r→slab.b_bottom_slide = m.slab_bottom_slide_b;
  r→slab.cos_angle = m.slab_cos_angle;
  r→method.a_calc = 0.5;
  r→method.b_calc = 1;
  r→method.g_calc = 0.5;
  r→method.quad_pts = 8;
  r→method.b_thinnest = 1.0/32.0;
```

60. EZ Inverse RT. *ez_Inverse_RT* is a simple interface to the main function *Inverse_RT* in this package. It eliminates the need for complicated data structures so that the command line interface (as well as those to Perl and Mathematica) will be simpler. This function assumes that the reflection and transmission include specular reflection and that the transmission also include unscattered transmission.

Other assumptions are that the top and bottom slides have the same index of refraction, that the illumination is collimated. Of course no sphere parameters are included.

```
⟨ Prototype for ez_Inverse_RT 60 ⟩ ≡
  void ez_Inverse_RT (double n, double nslide, double UR1, double UT1, double Tc, double
      *a, double *b, double *g, int * error )
```

This code is used in sections 36, 37, and 61.

61. ⟨ Definition for *ez_Inverse_RT* 61 ⟩ ≡

```
⟨ Prototype for ez_Inverse_RT 60 ⟩{ struct measure_type m;
    struct invert_type r;
    *a = 0;
    *b = 0;
    *g = 0;
    Initialize_Measure(&m);
    m.slab_index = n;
    m.slab_top_slide_index = nslide;
    m.slab_bottom_slide_index = nslide;
    m.slab_cos_angle = 1.0;
    m.num_measures = 3;
    fprintf(stderr, "ut1=%f\n", UT1);
    fprintf(stderr, "Tc=%f\n", Tc);
    if (UT1 ≡ 0) m.num_measures--;
    if (Tc ≡ 0) m.num_measures--;
    m.m_r = UR1;
    m.m_t = UT1;
    m.m_u = Tc;
    Initialize_Result(m, &r);
    r.method.quad_pts = 8;
    Inverse_RT(m, &r); * error = r . error ; if ( r . error ≡ IAD_NO_ERROR )
    {
      *a = r.a;
      *b = r.b;
      *g = r.g;
    }
    }
```

This code is used in section 35.

62. ⟨ Prototype for *Initialize_Measure* 62 ⟩ ≡

```
void Initialize_Measure(struct measure_type *m)
```

This code is used in sections 36 and 63.

63. ⟨ Definition for *Initialize_Measure* 63 ⟩ ≡

```
⟨ Prototype for Initialize_Measure 62 ⟩
{
  double default_sphere_d = 8.0 * 25.4;
  double default_sample_d = 0.0 * 25.4;
  double default_detector_d = 0.1 * 25.4;
  double default_entrance_d = 0.5 * 25.4;
  double sphere = default_sphere_d * default_sphere_d;
  m→slab_index = 1.0;
  m→slab_top_slide_index = 1.0;
  m→slab_top_slide_b = 0.0;
  m→slab_top_slide_thickness = 0.0;
  m→slab_bottom_slide_index = 1.0;
  m→slab_bottom_slide_b = 0.0;
  m→slab_bottom_slide_thickness = 0.0;
  m→slab_thickness = 1.0;
  m→slab_cos_angle = 1.0;
  m→num_spheres = 0;
  m→num_measures = 1;
  m→method = UNKNOWN;
  m→sphere_with_rc = 1.0;
  m→sphere_with_tc = 1.0;
  m→m_r = 0.0;
  m→m_t = 0.0;
  m→m_u = 0.0;
  m→d_sphere_r = default_sphere_d;
  m→as_r = default_sample_d * default_sample_d / sphere;
  m→ad_r = default_detector_d * default_detector_d / sphere;
  m→ae_r = default_entrance_d * default_entrance_d / sphere;
  m→aw_r = 1.0 - m→as_r - m→ad_r - m→ae_r;
  m→rd_r = 0.0;
  m→rw_r = 1.0;
  m→rstd_r = 1.0;
  m→f_r = 0.0;
  m→d_sphere_t = default_sphere_d;
  m→as_t = m→as_r;
  m→ad_t = m→ad_r;
  m→ae_t = m→ae_r;
  m→aw_t = m→aw_r;
  m→rd_t = 0.0;
  m→rw_t = 1.0;
  m→rstd_t = 1.0;
  m→f_t = 0.0;
  m→lambda = 0.0;
  m→d_beam = 0.0;
  m→ur1_lost = 0;
  m→uru_lost = 0;
  m→ut1_lost = 0;
  m→utu_lost = 0;
}
```

This code is used in section 35.

64. To avoid interfacing with C-structures it is necessary to pass the information as arrays. Here I have divided the experiment into (1) setup, (2) reflection sphere coefficients, (3) transmission sphere coefficients, (4) measurements, and (5) results.

```
⟨ Prototype for Spheres_Inverse_RT 64 ⟩ ≡
  void Spheres_Inverse_RT(double *setup, double *analysis, double *sphere_r, double *sphere_t, double
      *measurements, double *results)
```

This code is used in sections 37 and 65.

65.

```
⟨ Definition for Spheres_Inverse_RT 65 ⟩ ≡
  ⟨ Prototype for Spheres_Inverse_RT 64 ⟩{ struct measure_type m;
      struct invert_type r;
      long num_photons;
      double ur1, ut1, uru, utu;
      int i, mc_runs = 1;
      Initialize_Measure(&m);
      ⟨ handle setup 66 ⟩
      ⟨ handle reflection sphere 69 ⟩
      ⟨ handle transmission sphere 70 ⟩
      ⟨ handle measurement 68 ⟩
      Initialize_Result(m, &r);
      results[0] = 0;
      results[1] = 0;
      results[2] = 0;
      ⟨ handle analysis 67 ⟩
      Inverse_RT(m, &r);
      for (i = 0; i < mc_runs; i++) {
        MC_Lost(m, r, num_photons, &ur1, &ut1, &uru, &utu, &m.ur1_lost, &m.ut1_lost, &m.uru_lost,
            &m.utu_lost);
        Inverse_RT(m, &r);
      }
      if ( r . error ≡ IAD_NO_ERROR )
      {
        results[0] = (1 − r.a) * r.b/m.slab_thickness;
        results[1] = (r.a) * r.b/m.slab_thickness;
        results[2] = r.g;
      }
      results[3] = r . error ; }
```

This code is used in section 35.

66. These are in exactly the same order as the parameters in the .rxt header

```
⟨ handle setup 66 ⟩ ≡
  {
    double d_sample_r, d_entrance_r, d_detector_r;
    double d_sample_t, d_entrance_t, d_detector_t;
    m.slab_index = setup[0];
    m.slab_top_slide_index = setup[1];
    m.slab_thickness = setup[2];
    m.slab_top_slide_thickness = setup[3];
    m.d_beam = setup[4];
    m.rstd_r = setup[5];
    m.num_spheres = (int) setup[6];
    m.d_sphere_r = setup[7];
    d_sample_r = setup[8];
    d_entrance_r = setup[9];
    d_detector_r = setup[10];
    m.rw_r = setup[11];
    m.d_sphere_t = setup[12];
    d_sample_t = setup[13];
    d_entrance_t = setup[14];
    d_detector_t = setup[15];
    m.rw_t = setup[16];
    r.default_g = setup[17];
    num_photons = (long) setup[18];
    m.as_r = (d_sample_r/m.d_sphere_r) * (d_sample_r/m.d_sphere_r);
    m.ae_r = (d_entrance_r/m.d_sphere_r) * (d_entrance_r/m.d_sphere_r);
    m.ad_r = (d_detector_r/m.d_sphere_r) * (d_detector_r/m.d_sphere_r);
    m.aw_r = 1.0 - m.as_r - m.ae_r - m.ad_r;
    m.as_t = (d_sample_t/m.d_sphere_t) * (d_sample_t/m.d_sphere_t);
    m.ae_t = (d_entrance_t/m.d_sphere_t) * (d_entrance_t/m.d_sphere_t);
    m.ad_t = (d_detector_t/m.d_sphere_t) * (d_detector_t/m.d_sphere_t);
    m.aw_t = 1.0 - m.as_t - m.ae_t - m.ad_t;
    m.slab_bottom_slide_index = m.slab_top_slide_index;
    m.slab_bottom_slide_thickness = m.slab_top_slide_thickness;
    fprintf(stderr, "**** executing FIXME ****\n");
    m.slab_cos_angle = 1.0;      /* FIXME */
  }
```

This code is used in section 65.

67.

```
⟨ handle analysis 67 ⟩ ≡
  r.method.quad_pts = (int) analysis[0];
  mc_runs = (int) analysis[1];
```

This code is used in section 65.

68.

⟨ handle measurement 68 ⟩ ≡
```
m.m_r = measurements[0];
m.m_t = measurements[1];
m.m_u = measurements[2];
m.num_measures = 3;
fprintf(stderr, "m.m_t=%f\n", m.m_t);
fprintf(stderr, "m.m_u=%f\n", m.m_u);
if (m.m_t ≡ 0) m.num_measures −−;
if (m.m_u ≡ 0) m.num_measures −−;
```
This code is used in section 65.

69.

⟨ handle reflection sphere 69 ⟩ ≡
```
m.as_r = sphere_r[0];
m.ae_r = sphere_r[1];
m.ad_r = sphere_r[2];
m.rw_r = sphere_r[3];
m.rd_r = sphere_r[4];
m.rstd_r = sphere_r[5];
m.f_r = sphere_r[7];
```
This code is used in section 65.

70.

⟨ handle transmission sphere 70 ⟩ ≡
```
m.as_t = sphere_t[0];
m.ae_t = sphere_t[1];
m.ad_t = sphere_t[2];
m.rw_t = sphere_t[3];
m.rd_t = sphere_t[4];
m.rstd_t = sphere_t[5];
m.f_t = sphere_t[7];
```
This code is used in section 65.

71. I needed a routine that would calculate the values of `M_R` and `M_T` without doing the whole inversion process. It seems odd that this does not exist yet.

The values for the lost light *m.uru_lost* etc., should be calculated before calling this routine.

⟨ Prototype for *Calculate_MR_MT* 71 ⟩ ≡
```
void Calculate_MR_MT(struct measure_type m, struct invert_type r, int include_MC, double
    *M_R, double *M_T)
```
This code is used in sections 36 and 72.

72. ⟨ Definition for *Calculate_MR_MT* 72 ⟩ ≡

```
⟨ Prototype for Calculate_MR_MT 71 ⟩
{
  double distance, ur1, ut1, uru, utu;
  struct measure_type old_mm;
  struct invert_type old_rr;
  if (include_MC ∧ m.num_spheres > 0) MC_Lost(m, r, −2000, &ur1, &ut1, &uru, &utu, &(m.ur1_lost),
      &(m.ut1_lost), &(m.uru_lost), &(m.utu_lost));
  Get_Calc_State(&old_mm, &old_rr);
  Set_Calc_State(m, r);
  Calculate_Distance(M_R, M_T, &distance);
  Set_Calc_State(old_mm, old_rr);
}
```

This code is used in section 35.

73. So, it turns out that the minimum measured `M_R` can be less than four percent for black glass! This is because the sphere efficiency is much worse for the glass than for the white standard.

```
⟨ Prototype for Calculate_Minimum_MR 73 ⟩ ≡
  void Calculate_Minimum_MR(struct measure_type m, struct invert_type r, double *mr, double
      *mt)
```

This code is used in sections 36 and 74.

74. ⟨ Definition for *Calculate_Minimum_MR* 74 ⟩ ≡

```
⟨ Prototype for Calculate_Minimum_MR 73 ⟩
{
  if (r.default_b ≡ UNINITIALIZED) r.slab.b = 999;
  else r.slab.b = r.default_b;
  if (r.default_a ≡ UNINITIALIZED) r.slab.a = 0;
  else r.slab.a = r.default_a;
  if (r.default_g ≡ UNINITIALIZED) r.slab.g = 0;
  else r.slab.g = r.default_g;
  r.a = r.slab.a;
  r.b = r.slab.b;
  r.g = r.slab.g;
  Calculate_MR_MT(m, r, 0, mr, mt);
}
```

This code is used in section 35.

75. The minimum possible value of `MR` for a given `MT` will be when the albedo is zero and the maximal value will be when the albedo is unity. In the first case there will be light loss and in the second we will assume that light loss is neglible (to maximize `MR`).

The problem is that to calculate these values one must know the optical thickness. Fortunately with the recent addition of constrained minimization, we can do exactly this.

The only thing that remains is to sort out the light lost effect.

```
⟨ Prototype for MinMax_MR_MT 75 ⟩ ≡
  int MinMax_MR_MT(struct measure_type m, struct invert_type r)
```

This code is used in sections 36 and 76.

76. ⟨ Definition for *MinMax_MR_MT* 76 ⟩ ≡

```
⟨ Prototype for MinMax_MR_MT 75 ⟩
{
  double distance, m_r, x, min, max;
  if (m.m_r < 0) return IAD_MR_TOO_SMALL;
  if (m.m_r * m.rstd_r > 1) return IAD_MR_TOO_BIG;
  if (m.m_t < 0) return IAD_MT_TOO_SMALL;
  if (m.m_t ≡ 0) return IAD_NO_ERROR;
  m_r = m.m_r;
  m.m_r = 0;
  r.search = FIND_B;
  r.default_a = 0;
  U_Find_B(m, &r);
  Calculate_Distance(&min, &x, &distance);
  if (m_r < min) return IAD_MR_TOO_SMALL;
  r.default_a = 1.0;
  U_Find_B(m, &r);
  Calculate_Distance(&max, &x, &distance);
  if (m_r > max) return IAD_MR_TOO_BIG;
  return IAD_NO_ERROR;
}
```

This code is used in section 35.

77. IAD Input Output.
The special define below is to get Visual C to suppress silly warnings.

⟨ iad_io.c 77 ⟩ ≡
```
#define _CRT_SECURE_NO_WARNINGS
#include <string.h>
#include <stdio.h>
#include <ctype.h>
#include <math.h>
#include "ad_globl.h"
#include "iad_type.h"
#include "iad_io.h"
#include "iad_pub.h"
#include "version.h"
```
⟨ Definition for *skip_white* 87 ⟩
⟨ Definition for *read_number* 89 ⟩
⟨ Definition for *check_magic* 91 ⟩
⟨ Definition for *Read_Header* 81 ⟩
⟨ Definition for *Write_Header* 93 ⟩
⟨ Definition for *Read_Data_Line* 85 ⟩

78. ⟨ iad_io.h 78 ⟩ ≡
⟨ Prototype for *Read_Header* 80 ⟩;
⟨ Prototype for *Write_Header* 92 ⟩;
⟨ Prototype for *Read_Data_Line* 84 ⟩;

79. Reading the file header.

80. ⟨ Prototype for *Read_Header* 80 ⟩ ≡
int *Read_Header*(**FILE** ∗*fp*, **struct measure_type** ∗*m*, **int** ∗*params*)
This code is used in sections 78 and 81.

81. Pretty straightforward stuff. The only thing that needs to be commented on is that only one slide thickness/index is specified in the file. This must be applied to both the top and bottom slides. Finally, to specify no slide, then either setting the slide index to 1.0 or the thickness to 0.0 should do the trick.

```
⟨ Definition for Read_Header 81 ⟩ ≡
  ⟨ Prototype for Read_Header 80 ⟩
  {
    double x;
    Initialize_Measure(m);
    if (check_magic(fp)) return 1;
    if (read_number(fp, &m→slab_index)) return 1;
    if (read_number(fp, &m→slab_top_slide_index)) return 1;
    if (read_number(fp, &m→slab_thickness)) return 1;
    if (read_number(fp, &m→slab_top_slide_thickness)) return 1;
    if (read_number(fp, &m→d_beam)) return 1;
    if (m→slab_top_slide_thickness ≡ 0.0) m→slab_top_slide_index = 1.0;
    if (m→slab_top_slide_index ≡ 1.0) m→slab_top_slide_thickness = 0.0;
    if (m→slab_top_slide_index ≡ 0.0) {
      m→slab_top_slide_thickness = 0.0;
      m→slab_top_slide_index = 1.0;
    }
    m→slab_bottom_slide_index = m→slab_top_slide_index;
    m→slab_bottom_slide_thickness = m→slab_top_slide_thickness;
    if (read_number(fp, &m→rstd_r)) return 1;
    if (read_number(fp, &x)) return 1;
    m→num_spheres = (int) x;
    m→method = SUBSTITUTION;
    ⟨ Read coefficients for reflection sphere 82 ⟩
    ⟨ Read coefficients for transmission sphere 83 ⟩
    if (read_number(fp, &x)) return 1;
    *params = (int) x;
    m→num_measures = (*params ≥ 3) ? 3 : *params;
    return 0;
  }
```

This code is used in section 77.

82. ⟨ Read coefficients for reflection sphere 82 ⟩ ≡

```
  {
    double d_sample_r, d_entrance_r, d_detector_r;
    if (read_number(fp, &m→d_sphere_r)) return 1;
    if (read_number(fp, &d_sample_r)) return 1;
    if (read_number(fp, &d_entrance_r)) return 1;
    if (read_number(fp, &d_detector_r)) return 1;
    if (read_number(fp, &m→rw_r)) return 1;
    m→as_r = (d_sample_r/m→d_sphere_r) * (d_sample_r/m→d_sphere_r)/4.0;
    m→ae_r = (d_entrance_r/m→d_sphere_r) * (d_entrance_r/m→d_sphere_r)/4.0;
    m→ad_r = (d_detector_r/m→d_sphere_r) * (d_detector_r/m→d_sphere_r)/4.0;
    m→aw_r = 1.0 − m→as_r − m→ae_r − m→ad_r;
  }
```

This code is used in section 81.

83. ⟨ Read coefficients for transmission sphere 83 ⟩ ≡

```
{
  double d_sample_t, d_entrance_t, d_detector_t;
  if (read_number(fp, &m→d_sphere_t)) return 1;
  if (read_number(fp, &d_sample_t)) return 1;
  if (read_number(fp, &d_entrance_t)) return 1;
  if (read_number(fp, &d_detector_t)) return 1;
  if (read_number(fp, &m→rw_t)) return 1;
  m→as_t = (d_sample_t/m→d_sphere_t) * (d_sample_t/m→d_sphere_t)/4.0;
  m→ae_t = (d_entrance_t/m→d_sphere_t) * (d_entrance_t/m→d_sphere_t)/4.0;
  m→ad_t = (d_detector_t/m→d_sphere_t) * (d_detector_t/m→d_sphere_t)/4.0;
  m→aw_t = 1.0 − m→as_t − m→ae_t − m→ad_t;
}
```

This code is used in section 81.

84. Reading just one line of a data file.

This reads a line of data based on the value of *params*.

If the first number is greater than one then it is assumed to be the wavelength and is ignored. test on the first value of the line.

A non-zero value is returned upon a failure.

⟨ Prototype for *Read_Data_Line* 84 ⟩ ≡

```
int Read_Data_Line(FILE *fp, struct measure_type *m, int params)
```

This code is used in sections 78 and 85.

85. ⟨ Definition for *Read_Data_Line* 85 ⟩ ≡

```
⟨ Prototype for Read_Data_Line 84 ⟩
{
  if (read_number(fp, &m→m_r)) return 1;
  if (m→m_r > 1) {
    m→lambda = m→m_r;
    if (read_number(fp, &m→m_r)) return 1;
  }
  if (params ≡ 1) return 0;
  if (read_number(fp, &m→m_t)) return 1;
  if (params ≡ 2) return 0;
  if (read_number(fp, &m→m_u)) return 1;
  if (params ≡ 3) return 0;
  if (read_number(fp, &m→rw_r)) return 1;
  m→rw_t = m→rw_r;
  if (params ≡ 4) return 0;
  if (read_number(fp, &m→rw_t)) return 1;
  if (params ≡ 5) return 0;
  if (read_number(fp, &m→rstd_r)) return 1;
  if (params ≡ 6) return 0;
  if (read_number(fp, &m→rstd_t)) return 1;
  return 0;
}
```

This code is used in section 77.

86. Skip over white space and comments. It is assumed that # starts all comments and continues to the end of a line. This routine should work on files with nearly any line ending CR, LF, CRLF.

Failure is indicated by a non-zero return value.

```
⟨ Prototype for skip_white 86 ⟩ ≡
  int skip_white(FILE *fp)
```

This code is used in section 87.

87.

```
⟨ Definition for skip_white 87 ⟩ ≡
  ⟨ Prototype for skip_white 86 ⟩
  {
    int c = fgetc(fp);
    while (¬feof(fp)) {
      if (isspace(c)) c = fgetc(fp);
      else if (c ≡ '#') do c = fgetc(fp); while (¬feof(fp) ∧ c ≠ '\n' ∧ c ≠ '\r');
      else break;
    }
    if (feof(fp)) return 1;
    ungetc(c, fp);
    return 0;
  }
```

This code is used in section 77.

88. Read a single number. Return 0 if there are no problems, otherwise return 1.

```
⟨ Prototype for read_number 88 ⟩ ≡
  int read_number(FILE *fp, double *x)
```

This code is used in section 89.

89.

```
⟨ Definition for read_number 89 ⟩ ≡
  ⟨ Prototype for read_number 88 ⟩
  {
    if (skip_white(fp)) return 1;
    if (fscanf(fp, "%lf", x)) return 0;
    else return 1;
  }
```

This code is used in section 77.

90. Ensure that the data file is actually in the right form. Return 0 if the file has the right starting characters. Return 1 if on a failure.

```
⟨ Prototype for check_magic 90 ⟩ ≡
  int check_magic(FILE *fp)
```

This code is used in section 91.

91. ⟨ Definition for *check_magic* 91 ⟩ ≡

```
⟨ Prototype for check_magic 90 ⟩
{
  char magic[] = "IAD1";
  int i, c;
  for (i = 0; i < 4; i++) {
    c = fgetc(fp);
    if (feof(fp) ∨ c ≠ magic[i]) {
      fprintf(stderr, "Sorry, but iad input files must begin with IAD1\n");
      fprintf(stderr, "        as the first four characters of the file.\n");
      fprintf(stderr, "        Perhaps you are using an old iad format?\n");
      return 1;
    }
  }
  return 0;
}
```

This code is used in section 77.

92. Formatting the header information.

⟨ Prototype for *Write_Header* 92 ⟩ ≡

```
void Write_Header(struct measure_type m, struct invert_type r, int params)
```

This code is used in sections 78 and 93.

93. ⟨ Definition for *Write_Header* 93 ⟩ ≡

```
⟨ Prototype for Write_Header 92 ⟩
{
  ⟨ Write slab info 94 ⟩
  ⟨ Write irradiation info 95 ⟩
  ⟨ Write general sphere info 96 ⟩
  ⟨ Write first sphere info 97 ⟩
  ⟨ Write second sphere info 98 ⟩
  ⟨ Write measure and inversion info 99 ⟩
}
```

This code is used in section 77.

94. ⟨ Write slab info 94 ⟩ ≡

```
double xx;
printf("# Inverse Adding-Doubling %s \n", Version);
printf("# \n");
printf("#                          Beam diameter = %7.1f mm\n", m.d_beam);
printf("#                        Sample thickness = %7.1f mm\n", m.slab_thickness);
printf("#                   Top slide thickness = %7.1f mm\n", m.slab_top_slide_thickness);
printf("#                Bottom slide thickness = %7.1f mm\n", m.slab_bottom_slide_thickness);
printf("#           Sample index of refraction = %7.3f\n", m.slab_index);
printf("#        Top slide index of refraction = %7.3f\n", m.slab_top_slide_index);
printf("#     Bottom slide index of refraction = %7.3f\n", m.slab_bottom_slide_index);
```

This code is used in section 93.

95. ⟨ Write irradiation info 95 ⟩ ≡

```
printf("# \n");
```

This code is used in section 93.

96. ⟨ Write general sphere info 96 ⟩ ≡

```
printf("#␣␣␣Unscattered␣light␣collected␣in␣M_R␣=␣%7.1f␣%%\n", m.sphere_with_rc * 100);
printf("#␣␣␣Unscattered␣light␣collected␣in␣M_T␣=␣%7.1f␣%%\n", m.sphere_with_tc * 100);
printf("#␣\n");
```

This code is used in section 93.

97. ⟨ Write first sphere info 97 ⟩ ≡

```
printf("#␣Reflection␣sphere\n");
printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣sphere␣diameter␣=␣%7.1f␣mm\n", m.d_sphere_r);
printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣sample␣port␣diameter␣=␣%7.1f␣mm\n", 2 * m.d_sphere_r * sqrt(m.as_r));
printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣entrance␣port␣diameter␣=␣%7.1f␣mm\n", 2 * m.d_sphere_r * sqrt(m.ae_r));
printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣detector␣port␣diameter␣=␣%7.1f␣mm\n", 2 * m.d_sphere_r * sqrt(m.ad_r));
printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣wall␣reflectance␣=␣%7.1f␣%%\n", m.rw_r * 100);
printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣standard␣reflectance␣=␣%7.1f␣%%\n", m.rstd_r * 100);
printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣detector␣reflectance␣=␣%7.1f␣%%\n", m.rd_r * 100);
printf("#\n");
```

This code is used in section 93.

98. ⟨ Write second sphere info 98 ⟩ ≡

```
printf("#␣Transmission␣sphere\n");
printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣sphere␣diameter␣=␣%7.1f␣mm\n", m.d_sphere_t);
printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣sample␣port␣diameter␣=␣%7.1f␣mm\n", 2 * m.d_sphere_r * sqrt(m.as_t));
printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣entrance␣port␣diameter␣=␣%7.1f␣mm\n", 2 * m.d_sphere_r * sqrt(m.ae_t));
printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣detector␣port␣diameter␣=␣%7.1f␣mm\n", 2 * m.d_sphere_r * sqrt(m.ad_t));
printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣wall␣reflectance␣=␣%7.1f␣%%\n", m.rw_t * 100);
printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣standard␣transmittance␣=␣%7.1f␣%%\n", m.rstd_t * 100);
printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣detector␣reflectance␣=␣%7.1f␣%%\n", m.rd_t * 100);
```

This code is used in section 93.

99. ⟨ Write measure and inversion info 99 ⟩ ≡

```
printf("#\n");
switch (params) {
case −1: printf("# No M_R or M_T -- forward calculation.\n");
  break;
case 1: printf("# Just M_R was measured");
  break;
case 2: printf("# M_R and M_T were measured");
  break;
case 3: printf("# M_R, M_T, and M_U were measured");
  break;
case 4: printf("# M_R, M_T, M_U, and r_w were measured");
  break;
case 5: printf("# M_R, M_T, M_U, r_w, and t_w were measured");
  break;
case 6: printf("# M_R, M_T, M_U, r_w, t_w, and r_std were measured");
  break;
case 7: printf("# M_R, M_T, M_U, r_w, t_w, r_std and t_std were measured");
  break;
default: printf("# Something went wrong ... measures should be 1 to 5!\n");
  break;
}
if (1 ≤ params ∧ params ≤ 7) {
  switch (m.method) {
  case UNKNOWN: printf(" using an unknown method.");
    break;
  case SUBSTITUTION: printf(" using the substitution (single-beam) method.\n");
    break;
  case COMPARISON: printf(" using the comparison (dual-beam) method.\n");
  }
}
switch (m.num_spheres) {
case 0: printf("# No sphere corrections were used.\n");
  break;
case 1: printf("# Single sphere corrections were used.\n");
  break;
case 2: printf("# Double sphere corrections were used.\n");
  break;
}
switch (r.search) {
case FIND_AB: printf("# The inverse routine varied the albedo and optical depth.\n");
  printf("# \n");
  xx = (r.default_g ≠ UNINITIALIZED) ? r.default_g : 0;
  printf("# Default single scattering anisotropy = %7.3f \n", xx);
  break;
case FIND_AG: printf("# The inverse routine varied the albedo and anisotropy.\n");
  printf("# \n");
  if (r.default_b ≠ UNINITIALIZED)
    printf("#                    Default (mu_t*d) = %7.3g\n", r.default_b);
  else printf("# \n");
  break;
case FIND_AUTO: printf("# The inverse routine adapted to the input data.\n");
```

```
    printf("#␣\n");
    printf("#␣\n");
    break;
  case FIND_A: printf("#␣The␣inverse␣routine␣varied␣only␣the␣albedo.\n");
    printf("#␣\n");
    xx = (r.default_g ≠ UNINITIALIZED) ? r.default_g : 0;
    printf("#␣Default␣single␣scattering␣anisotropy␣is␣%7.3f␣", xx);
    xx = (r.default_b ≠ UNINITIALIZED) ? r.default_b : HUGE_VAL;
    printf("␣and␣(mu_t*d)␣=␣%7.3g\n", xx);
    break;
  case FIND_B: printf("#␣The␣inverse␣routine␣varied␣only␣the␣optical␣depth.\n");
    printf("#␣\n");
    xx = (r.default_g ≠ UNINITIALIZED) ? r.default_g : 0;
    printf("#␣Default␣single␣scattering␣anisotropy␣is␣%7.3f␣", xx);
    if (r.default_a ≠ UNINITIALIZED) printf("and␣default␣albedo␣=␣%7.3g\n", r.default_a);
    else printf("\n");
    break;
  case FIND_Ba: printf("#␣The␣inverse␣routine␣varied␣only␣the␣absorption.\n");
    printf("#␣\n");
    xx = (r.default_bs ≠ UNINITIALIZED) ? r.default_bs : 0;
    printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Default␣(mu_s*d)␣=␣%7.3g\n", xx);
    break;
  case FIND_Bs: printf("#␣The␣inverse␣routine␣varied␣only␣the␣scattering.\n");
    printf("#␣\n");
    xx = (r.default_ba ≠ UNINITIALIZED) ? r.default_ba : 0;
    printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Default␣(mu_a*d)␣=␣%7.3g\n", xx);
    break;
  default: printf("#␣\n");
    printf("#␣\n");
    printf("#␣\n");
    break;
  }
  printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣AD␣quadrature␣points␣=␣%3d\n", r.method.quad_pts);
  printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣AD␣tolerance␣for␣success␣=␣%9.5f\n", r.tolerance);
  printf("#␣␣␣␣␣␣MC␣tolerance␣for␣mu_a␣and␣mu_s'␣=␣%7.3f␣%%\n", r.MC_tolerance);
```

This code is used in section 93.

100. IAD Calculation.

⟨ iad_calc.c 100 ⟩ ≡
```
#include <math.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "nr_util.h"
#include "nr_zbrent.h"
#include "ad_globl.h"
#include "ad_frsnl.h"
#include "ad_prime.h"
#include "iad_type.h"
#include "iad_util.h"
#include "iad_calc.h"
#define ABIT 1.0e-6
#define A_COLUMN 1
#define B_COLUMN 2
#define G_COLUMN 3
#define URU_COLUMN 4
#define UTU_COLUMN 5
#define UR1_COLUMN 6
#define UT1_COLUMN 7
#define REFLECTION_SPHERE 1
#define TRANSMISSION_SPHERE 0
#define GRID_SIZE 101
#define T_TRUST_FACTOR 2
  static int CALCULATING_GRID = 1;
  static struct measure_type MM;
  static struct invert_type RR;
  static struct measure_type MGRID;
  static struct invert_type RGRID;
  static double **The_Grid = Λ;
  static double GG_a;
  static double GG_b;
  static double GG_g;
  static double GG_bs;
  static double GG_ba;
  static boolean_type The_Grid_Initialized = FALSE;
  static boolean_type The_Grid_Search = -1;
  double FRACTION = 0.0;
  ⟨ Definition for Set_Calc_State 116 ⟩
  ⟨ Definition for Get_Calc_State 118 ⟩
  ⟨ Definition for Same_Calc_State 120 ⟩
  ⟨ Prototype for Fill_AB_Grid 136 ⟩;
  ⟨ Prototype for Fill_AG_Grid 141 ⟩;
  ⟨ Definition for Allocate_Grid 122 ⟩
  ⟨ Definition for Valid_Grid 126 ⟩
  ⟨ Definition for fill_grid_entry 135 ⟩
  ⟨ Definition for Fill_Grid 151 ⟩
  ⟨ Definition for Near_Grid_Points 134 ⟩
  ⟨ Definition for Fill_AB_Grid 137 ⟩
  ⟨ Definition for Fill_AG_Grid 142 ⟩
```

⟨ Definition for *Fill_BG_Grid* 145 ⟩
⟨ Definition for *Fill_BaG_Grid* 147 ⟩
⟨ Definition for *Fill_BsG_Grid* 149 ⟩
⟨ Definition for *Grid_ABG* 124 ⟩
⟨ Definition for *Gain* 105 ⟩
⟨ Definition for *Gain_11* 107 ⟩
⟨ Definition for *Gain_22* 109 ⟩
⟨ Definition for *Two_Sphere_R* 111 ⟩
⟨ Definition for *Two_Sphere_T* 113 ⟩
⟨ Definition for *Calculate_Distance_With_Corrections* 157 ⟩
⟨ Definition for *Calculate_Grid_Distance* 155 ⟩
⟨ Definition for *Calculate_Distance* 153 ⟩
⟨ Definition for *abg_distance* 132 ⟩
⟨ Definition for *Find_AG_fn* 167 ⟩
⟨ Definition for *Find_AB_fn* 169 ⟩
⟨ Definition for *Find_Ba_fn* 171 ⟩
⟨ Definition for *Find_Bs_fn* 173 ⟩
⟨ Definition for *Find_A_fn* 175 ⟩
⟨ Definition for *Find_B_fn* 177 ⟩
⟨ Definition for *Find_G_fn* 179 ⟩
⟨ Definition for *Find_BG_fn* 181 ⟩
⟨ Definition for *Find_BaG_fn* 183 ⟩
⟨ Definition for *Find_BsG_fn* 185 ⟩
⟨ Definition for *maxloss* 187 ⟩
⟨ Definition for *Max_Light_Loss* 189 ⟩

101.

⟨ `iad_calc.h` 101 ⟩ ≡
⟨ Prototype for *Gain* 104 ⟩;
⟨ Prototype for *Gain_11* 106 ⟩;
⟨ Prototype for *Gain_22* 108 ⟩;
⟨ Prototype for *Two_Sphere_R* 110 ⟩;
⟨ Prototype for *Two_Sphere_T* 112 ⟩;
⟨ Prototype for *Set_Calc_State* 115 ⟩;
⟨ Prototype for *Get_Calc_State* 117 ⟩;
⟨ Prototype for *Same_Calc_State* 119 ⟩;
⟨ Prototype for *Valid_Grid* 125 ⟩;
⟨ Prototype for *Allocate_Grid* 121 ⟩;
⟨ Prototype for *Fill_Grid* 150 ⟩;
⟨ Prototype for *Near_Grid_Points* 133 ⟩;
⟨ Prototype for *Grid_ABG* 123 ⟩;
⟨ Prototype for *Find_AG_fn* 166 ⟩;
⟨ Prototype for *Find_AB_fn* 168 ⟩;
⟨ Prototype for *Find_Ba_fn* 170 ⟩;
⟨ Prototype for *Find_Bs_fn* 172 ⟩;
⟨ Prototype for *Find_A_fn* 174 ⟩;
⟨ Prototype for *Find_B_fn* 176 ⟩;
⟨ Prototype for *Find_G_fn* 178 ⟩;
⟨ Prototype for *Find_BG_fn* 180 ⟩;
⟨ Prototype for *Find_BsG_fn* 184 ⟩;
⟨ Prototype for *Find_BaG_fn* 182 ⟩;
⟨ Prototype for *Fill_BG_Grid* 144 ⟩;
⟨ Prototype for *Fill_BsG_Grid* 148 ⟩;
⟨ Prototype for *Fill_BaG_Grid* 146 ⟩;
⟨ Prototype for *Calculate_Distance_With_Corrections* 156 ⟩;
⟨ Prototype for *Calculate_Distance* 152 ⟩;
⟨ Prototype for *Calculate_Grid_Distance* 154 ⟩;
⟨ Prototype for *abg_distance* 131 ⟩;
⟨ Prototype for *maxloss* 186 ⟩;
⟨ Prototype for *Max_Light_Loss* 188 ⟩;

102. Initialization.

The functions in this file assume that the local variables `MM` and `RR` have been initialized appropriately. The variable `MM` contains all the information about how a particular experiment was done. The structure `RR` contains the data structure that is passed to the adding-doubling routines as well as the number of quadrature points.

history 6/8/94 changed error output to *stderr*.

103. Gain.

Assume that a sphere is illuminated with diffuse light having a power P. This light can reach all parts of sphere — specifically, light from this source is not blocked by a baffle. Multiple reflections in the sphere will increase the power falling on non-white areas in the sphere (e.g., the sample, detector, and entrance) To find the total we need to sum all the total of all incident light at a point. The first incidence is

$$P_w^{(1)} = a_w P, \qquad P_s^{(1)} = a_s P, \qquad P_d^{(1)} = a_d P$$

The light from the detector and sample is multiplied by $(1 - a_e)$ and not by a_w because the light from the detector (and sample) is not allowed to hit either the detector or sample. The second incidence on the wall is

$$P_w^{(2)} = a_w r_w P_w^{(1)} + (1 - a_e) r_d P_d^{(1)} + (1 - a_e) r_s P_s^{(1)}$$

The light that hits the walls after k bounces has the same form as above

$$P_w^{(k)} = a_w r_w P_w^{(k-1)} + (1 - a_e) r_d P_d^{(k-1)} + (1 - a_e) r_s P_s^{(k-1)}$$

Since the light falling on the sample and detector must come from the wall

$$P_s^{(k)} = a_s r_w P_w^{(k-1)} \qquad \text{and} \qquad P_d^{(k)} = a_d r_w P_w^{(k-1)},$$

Therefore,

$$P_w^{(k)} = a_w r_w P_w^{(k-1)} + (1 - a_e) r_w (a_d r_d + a_s r_s) P_w^{(k-2)}$$

The total power falling on the walls is just

$$P_w = \sum_{k=1}^{\infty} P_w^{(k)} = \frac{a_w + (1 - a_e)(a_d r_d + a_s r_s)}{1 - a_w r_w - (1 - a_e) r_w (a_d r_d + a_s r_s)} P$$

The total power falling the detector is

$$P_d = a_d P + \sum_{k=2}^{\infty} a_d r_w P_w^{(k-1)} = a_d P + a_d r_w P_w$$

The gain $G(r_s)$ on the irradiance on the detector (relative to a black sphere),

$$G(r_s) \equiv \frac{P_d / A_d}{P / A}$$

in terms of the sphere parameters

$$G(r_s) = 1 + \frac{1}{a_w} \cdot \frac{a_w r_w + (1 - a_e) r_w (a_d r_d + a_s r_s)}{1 - a_w r_w - (1 - a_e) r_w (a_d r_d + a_s r_s)}$$

The gain for a detector in a transmission sphere is similar, but with primed parameters to designate a second potential sphere that is used. For a black sphere the gain $G(0) = 1$, which is easily verified by setting $r_w = 0$, $r_s = 0$, and $r_d = 0$. Conversely, when the sphere walls and sample are perfectly white, the irradiance at the entrance port, the sample port, and the detector port must increase so that the total power leaving via these ports is equal to the incident diffuse power P. Thus the gain should be the ratio of the sphere wall area over the area of the ports through which light leaves or $G(1) = A/(A_e + A_d)$ which follows immediately from the gain formula with $r_w = 1$, $r_s = 1$, and $r_d = 0$.

104. The gain $G(r_s)$ on the irradiance on the detector (relative to a black sphere),

$$G(r_s) \equiv \frac{P_d/A_d}{P/A}$$

in terms of the sphere parameters

$$G(r_s) = 1 + \frac{a_w r_w + (1-a_e) r_w (a_d r_d + a_s r_s)}{1 - a_w r_w - (1-a_e) r_w (a_d r_d + a_s r_s)}$$

⟨ Prototype for *Gain* 104 ⟩ ≡
```
double Gain(int sphere, struct measure_type m, double URU)
```
This code is used in sections 101 and 105.

105. ⟨ Definition for *Gain* 105 ⟩ ≡
```
⟨ Prototype for Gain 104 ⟩
{
  double G, tmp;
  if (sphere ≡ REFLECTION_SPHERE) {
    tmp = m.rw_r * (m.aw_r + (1 - m.ae_r) * (m.ad_r * m.rd_r + m.as_r * URU));
    if (tmp ≡ 1.0) G = 1;
    else G = 1.0 + tmp/(1.0 - tmp);
  }
  else {
    tmp = m.rw_t * (m.aw_t + (1 - m.ae_t) * (m.ad_t * m.rd_t + m.as_t * URU));
    if (tmp ≡ 1.0) G = 1;
    else G = 1.0 + tmp/(1.0 - tmp);
  }
  return G;
}
```
This code is used in section 100.

106. The gain for light on the detector in the first sphere for diffuse light starting in that same sphere is defined as

$$G_{1\to1}(r_s, t_s) \equiv \frac{P_{1\to1}(r_s, t_s)/A_d}{P/A}$$

then the full expression for the gain is

$$G_{1\to1}(r_s, t_s) = \frac{G(r_s)}{1 - a_s a'_s r_w r'_w (1-a_e)(1-a'_e) G(r_s) G'(r_s) t_s^2}$$

⟨ Prototype for *Gain_11* 106 ⟩ ≡
```
double Gain_11(struct measure_type m, double URU, double tdiffuse)
```
This code is used in sections 101 and 107.

107. ⟨ Definition for *Gain_11* 107 ⟩ ≡

```
⟨ Prototype for Gain_11 106 ⟩
{
  double G, GP, G11;
  G = Gain(REFLECTION_SPHERE, m, URU);
  GP = Gain(TRANSMISSION_SPHERE, m, URU);
  G11 = G/(1-m.as_r*m.as_t*m.aw_r*m.aw_t*(1-m.ae_r)*(1-m.ae_t)*G*GP*tdiffuse*tdiffuse);
  return G11;
}
```

This code is used in section 100.

108. Similarly, when the light starts in the second sphere, the gain for light on the detector in the second sphere $G_{2\to2}$ is found by switching all primed variables to unprimed. Thus $G_{2\to1}(r_s, t_s)$ is

$$G_{2\to2}(r_s, t_s) = \frac{G'(r_s)}{1 - a_s a'_s r_w r'_w (1 - a_e)(1 - a'_e) G(r_s) G'(r_s) t_s^2}$$

⟨ Prototype for *Gain_22* 108 ⟩ ≡

```
double Gain_22(struct measure_type m, double URU, double tdiffuse)
```

This code is used in sections 101 and 109.

109. ⟨ Definition for *Gain_22* 109 ⟩ ≡

```
⟨ Prototype for Gain_22 108 ⟩
{
  double G, GP, G22;
  G = Gain(REFLECTION_SPHERE, m, URU);
  GP = Gain(TRANSMISSION_SPHERE, m, URU);
  G22 = GP/(1-m.as_r*m.as_t*m.aw_r*m.aw_t*(1-m.ae_r)*(1-m.ae_t)*G*GP*tdiffuse*tdiffuse);
  return G22;
}
```

This code is used in section 100.

110. The reflected power for two spheres makes use of the formulas for *Gain_11* above.

The light on the detector in the reflection (first) sphere arises from three sources: the fraction of light directly reflected off the sphere wall $f r_w^2 (1 - a_e) P$, the fraction of light reflected by the sample $(1 - f) r_s^{\text{direct}} r_w^2 (1 - a_e) P$, and the light transmitted through the sample $(1 - f) t_s^{\text{direct}} r'_w (1 - a'_e) P$,

$$\begin{aligned} R(r_s^{\text{direct}}, r_s, t_s^{\text{direct}}, t_s) = {} & G_{1\to1}(r_s, t_s) \cdot a_d (1 - a_e) r_w^2 f P \\ & + G_{1\to1}(r_s, t_s) \cdot a_d (1 - a_e) r_w (1 - f) r_s^{\text{direct}} P \\ & + G_{2\to1}(r_s, t_s) \cdot a_d (1 - a'_e) r'_w (1 - f) t_s^{\text{direct}} P \end{aligned}$$

which simplifies slightly to

$$\begin{aligned} R(r_s^{\text{direct}}, r_s, t_s^{\text{direct}}, t_s) = {} & a_d (1 - a_e) r_w P \cdot G_{1\to1}(r_s, t_s) \\ & \times \left[(1 - f) r_s^{\text{direct}} + f r_w + (1 - f) a'_s (1 - a'_e) r'_w t_s^{\text{direct}} t_s G'(r_s) \right] \end{aligned}$$

⟨ Prototype for *Two_Sphere_R* 110 ⟩ ≡

```
double Two_Sphere_R(struct measure_type m, double UR1, double URU, double UT1, double UTU)
```

This code is used in sections 101 and 111.

111. ⟨ Definition for *Two_Sphere_R* 111 ⟩ ≡

```
⟨ Prototype for Two_Sphere_R 110 ⟩
{
  double x, GP;
  GP = Gain(TRANSMISSION_SPHERE, m, URU);
  x = m.ad_r * (1 - m.ae_r) * m.rw_r * Gain_11(m, URU, UTU);
  x *= (1 - m.f_r) * UR1 + m.rw_r * m.f_r + (1 - m.f_r) * m.as_t * (1 - m.ae_t) * m.rw_t * UT1 * UTU * GP;
  return x;
}
```

This code is used in section 100.

112. For the power on the detector in the transmission (second) sphere we have the same three sources. The only difference is that the subscripts on the gain terms now indicate that the light ends up in the second sphere

$$\begin{aligned} T(r_s^{\text{direct}}, r_s, t_s^{\text{direct}}, t_s) &= G_{1\to 2}(r_s, t_s)\cdot a_d'(1-a_e)r_w^2 fP \\ &\quad + G_{1\to 2}(r_s, t_s)\cdot a_d'(1-a_e)r_w(1-f)r_s^{\text{direct}}P \\ &\quad + G_{2\to 2}(r_s, t_s)\cdot a_d'(1-a_e')r_w'(1-f)t_s^{\text{direct}}P \end{aligned}$$

or

$$\begin{aligned} T(r_s^{\text{direct}}, r_s, t_s^{\text{direct}}, t_s) &= a_d'(1-a_e')r_w'P\cdot G_{2\to 2}(r_s, t_s) \\ &\quad \times \left[(1-f)t_s^{\text{direct}} + (1-a_e)r_w a_s t_s(fr_w + (1-f)r_s^{\text{direct}})G(r_s)\right] \end{aligned}$$

⟨ Prototype for *Two_Sphere_T* 112 ⟩ ≡

```
double Two_Sphere_T(struct measure_type m, double UR1, double URU, double UT1, double UTU)
```

This code is used in sections 101 and 113.

113. ⟨ Definition for *Two_Sphere_T* 113 ⟩ ≡

```
⟨ Prototype for Two_Sphere_T 112 ⟩
{
  double x, G;
  G = Gain(REFLECTION_SPHERE, m, URU);
  x = m.ad_t * (1 - m.ae_t) * m.rw_t * Gain_22(m, URU, UTU);
  x *= (1 - m.f_r) * UT1 + (1 - m.ae_r) * m.rw_r * m.as_r * UTU * (m.f_r * m.rw_r + (1 - m.f_r) * UR1) * G;
  return x;
}
```

This code is used in section 100.

114. Grid Routines. There is a long story associated with these routines. I spent a lot of time trying to find an empirical function to allow a guess at a starting value for the inversion routine. Basically nothing worked very well. There were too many special cases and what not. So I decided to calculate a whole bunch of reflection and transmission values and keep their associated optical properties linked nearby.

I did the very simplest thing. I just allocate a matrix that is five columns wide. Then I fill every row with a calculated set of optical properties and observables. The distribution of values that I use could certainly use some work, but they currently work.

SO... how does this thing work anyway? There are two possible grids one for calculations requiring the program to find the albedo and the optical depth (a and b) and one to find the albedo and anisotropy (a and g). These grids must be allocated and initialized before use.

115. This is a pretty important routine that should have some explanation. The reason that it exists, is that we need some 'out-of-band' information during the minimization process. Since the light transport calculation depends on all sorts of stuff (e.g., the sphere parameters) and the minimization routines just vary one or two parameters this information needs to be put somewhere.

I chose the global variables `MM` and `RR` to save things in.

The bottom line is that you cannot do a light transport calculation without calling this routine first.

```
⟨ Prototype for Set_Calc_State 115 ⟩ ≡
  void Set_Calc_State(struct measure_type m, struct invert_type r)
```

This code is used in sections 101 and 116.

116.
```
⟨ Definition for Set_Calc_State 116 ⟩ ≡
  ⟨ Prototype for Set_Calc_State 115 ⟩
  {
    memcpy(&MM, &m, sizeof(struct measure_type));
    memcpy(&RR, &r, sizeof(struct invert_type));
    if (Debug(DEBUG_ITERATIONS) ∧ ¬CALCULATING_GRID) {
      fprintf(stderr, "UR1␣loss=%g,␣UT1␣loss=%g␣␣␣", m.ur1_lost, m.ut1_lost);
      fprintf(stderr, "URU␣loss=%g,␣UTU␣loss=%g\n", m.uru_lost, m.utu_lost);
    }
  }
```

This code is used in section 100.

117. The inverse of the previous routine. Note that you must have space for the parameters m and r already allocated.

```
⟨ Prototype for Get_Calc_State 117 ⟩ ≡
  void Get_Calc_State(struct measure_type *m, struct invert_type *r)
```

This code is used in sections 101 and 118.

118.
```
⟨ Definition for Get_Calc_State 118 ⟩ ≡
  ⟨ Prototype for Get_Calc_State 117 ⟩
  {
    memcpy(m, &MM, sizeof(struct measure_type));
    memcpy(r, &RR, sizeof(struct invert_type));
  }
```

This code is used in section 100.

119. The inverse of the previous routine. Note that you must have space for the parameters m and r already allocated.

```
⟨ Prototype for Same_Calc_State 119 ⟩ ≡
  boolean_type Same_Calc_State(struct measure_type m, struct invert_type r)
```

This code is used in sections 101 and 120.

120. ⟨ Definition for *Same_Calc_State* 120 ⟩ ≡

```
⟨ Prototype for Same_Calc_State 119 ⟩
{
  if (The_Grid ≡ Λ) return FALSE;
  if (¬The_Grid_Initialized) return FALSE;
  if (r.search ≠ RR.search) return FALSE;
  if (r.method.quad_pts ≠ RR.method.quad_pts) return FALSE;
  if (r.slab.a ≠ RR.slab.a) return FALSE;
  if (r.slab.b ≠ RR.slab.b) return FALSE;
  if (r.slab.g ≠ RR.slab.g) return FALSE;
  if (r.slab.phase_function ≠ RR.slab.phase_function) return FALSE;
  if (r.slab.n_slab ≠ RR.slab.n_slab) return FALSE;
  if (r.slab.n_top_slide ≠ RR.slab.n_top_slide) return FALSE;
  if (r.slab.n_bottom_slide ≠ RR.slab.n_bottom_slide) return FALSE;
  if (r.slab.b_top_slide ≠ RR.slab.b_top_slide) return FALSE;
  if (r.slab.b_bottom_slide ≠ RR.slab.b_bottom_slide) return FALSE;
  if (r.slab.cos_angle ≠ RR.slab.cos_angle) return FALSE;
  if ((m.num_measures ≡ 3) ∧ (m.m_u ≠ MGRID.m_u)) return (FALSE);
  return TRUE;
}
```

This code is used in section 100.

121. ⟨ Prototype for *Allocate_Grid* 121 ⟩ ≡

```
void Allocate_Grid(search_type s)
```

This code is used in sections 101 and 122.

122. ⟨ Definition for *Allocate_Grid* 122 ⟩ ≡

```
⟨ Prototype for Allocate_Grid 121 ⟩
{
  The_Grid = dmatrix(0, GRID_SIZE * GRID_SIZE, 1, 7);
  if (The_Grid ≡ Λ) AD_error("unable␣to␣allocate␣the␣grid␣matrix");
  The_Grid_Initialized = FALSE;
}
```

This code is used in section 100.

123. This routine will return the a, b, and g values for a particular row in the grid.

⟨ Prototype for *Grid_ABG* 123 ⟩ ≡

```
void Grid_ABG(int i, int j, guess_type *guess)
```

This code is used in sections 101 and 124.

124. ⟨ Definition for *Grid_ABG* 124 ⟩ ≡

```
⟨ Prototype for Grid_ABG 123 ⟩
{
  if (0 ≤ i ∧ i < GRID_SIZE ∧ 0 ≤ j ∧ j < GRID_SIZE) {
    guess¬a = The_Grid[GRID_SIZE * i + j][A_COLUMN];
    guess¬b = The_Grid[GRID_SIZE * i + j][B_COLUMN];
    guess¬g = The_Grid[GRID_SIZE * i + j][G_COLUMN];
    guess¬distance = Calculate_Grid_Distance(i, j);
  }
  else {
    guess¬a = 0.5;
    guess¬b = 0.5;
    guess¬g = 0.5;
    guess¬distance = 999;
  }
}
```

This code is used in section 100.

125. This routine is used to figure out if the current grid is valid. This can fail for several reasons. First the grid may not have been allocated. Or it may not have been initialized. The boundary conditions may have changed. The number or values of the sphere parameters may have changed. It is tedious, but straightforward to check these cases out.

If this routine returns true, then it is a pretty good bet that the values in the current grid can be used to guess the next starting set of values.

⟨ Prototype for *Valid_Grid* 125 ⟩ ≡

```
boolean_type Valid_Grid(struct measure_type m, search_type s)
```

This code is used in sections 101 and 126.

126. ⟨ Definition for *Valid_Grid* 126 ⟩ ≡

```
⟨ Prototype for Valid_Grid 125 ⟩
{
  ⟨ Tests for invalid grid 127 ⟩
  return (TRUE);
}
```

This code is used in section 100.

127. First check are to test if the grid has ever been filled

⟨ Tests for invalid grid 127 ⟩ ≡

```
if (The_Grid ≡ Λ) return (FALSE);
if (¬The_Grid_Initialized) return (FALSE);
```

See also sections 128, 129, and 130.

This code is used in section 126.

128. If the type of search has changed then report the grid as invalid

⟨ Tests for invalid grid 127 ⟩ +≡

```
if (The_Grid_Search ≠ s) return (FALSE);
```

129. Compare the $m.m\_u$ value only if there are three measurements

⟨ Tests for invalid grid 127 ⟩ +≡

```
if ((m.num_measures ≡ 3) ∧ (m.m_u ≠ MGRID.m_u)) return (FALSE);
```

130. Make sure that the boundary conditions have not changed.

```
⟨ Tests for invalid grid 127 ⟩ +≡
  if (m.slab_index ≠ MGRID.slab_index) return (FALSE);
  if (m.slab_cos_angle ≠ MGRID.slab_cos_angle) return (FALSE);
  if (m.slab_top_slide_index ≠ MGRID.slab_top_slide_index) return (FALSE);
  if (m.slab_bottom_slide_index ≠ MGRID.slab_bottom_slide_index) return (FALSE);
```

131. Routine to just figure out the distance to a particular a, b, g point

```
⟨ Prototype for abg_distance 131 ⟩ ≡
  void abg_distance(double a, double b, double g, guess_type *guess)
```

This code is used in sections 101 and 132.

132.

```
⟨ Definition for abg_distance 132 ⟩ ≡
  ⟨ Prototype for abg_distance 131 ⟩
  {
    double m_r, m_t, distance;
    struct measure_type old_mm;
    struct invert_type old_rr;
    Get_Calc_State(&old_mm, &old_rr);
    RR.slab.a = a;
    RR.slab.b = b;
    RR.slab.g = g;
    Calculate_Distance(&m_r, &m_t, &distance);
    Set_Calc_State(old_mm, old_rr);
    guess¬a = a;
    guess¬b = b;
    guess¬g = g;
    guess¬distance = distance;
  }
```

This code is used in section 100.

133. This just searches through the grid to find the minimum entry and returns the optical properties of that entry. The smallest, the next smallest, and the third smallest values are returned.

This has been rewritten to use *Calculate_Distance_With_Corrections* so that changes in sphere parameters won't necessitate recalculating the grid.

```
⟨ Prototype for Near_Grid_Points 133 ⟩ ≡
  void Near_Grid_Points(double r, double t, search_type s, int *i_min, int *j_min)
```

This code is used in sections 101 and 134.

134. ⟨ Definition for *Near_Grid_Points* 134 ⟩ ≡

```
⟨ Prototype for Near_Grid_Points 133 ⟩
{
  int i, j;
  double fval;
  double smallest = 10.0;
  struct measure_type old_mm;
  struct invert_type old_rr;
  Get_Calc_State(&old_mm, &old_rr);
  *i_min = 0;
  *j_min = 0;
  for (i = 0; i < GRID_SIZE; i++) {
    for (j = 0; j < GRID_SIZE; j++) {
      CALCULATING_GRID = 1;
      fval = Calculate_Grid_Distance(i, j);
      CALCULATING_GRID = 0;
      if (fval < smallest) {
        *i_min = i;
        *j_min = j;
        smallest = fval;
      }
    }
  }
  Set_Calc_State(old_mm, old_rr);
}
```

This code is used in section 100.

135. Simple routine to put values into the grid
Presumes that RR.*slab* is properly set up.

```
⟨ Definition for fill_grid_entry 135 ⟩ ≡
  static void fill_grid_entry(int i, int j)
  {
    double ur1, ut1, uru, utu;
    if (RR.slab.b ≤ 1 · 10^−6) RR.slab.b = 1 · 10^−6;
    if (Debug(DEBUG_EVERY_CALC))
      fprintf(stderr, "a=%8.5f␣b=%10.5f␣g=%8.5f␣", RR.slab.a, RR.slab.b, RR.slab.g);
    RT(RR.method.quad_pts, &RR.slab, &ur1, &ut1, &uru, &utu);
    if (Debug(DEBUG_EVERY_CALC)) fprintf(stderr, "ur1=%8.5f␣ut1=%8.5f\n", ur1, ut1);
    The_Grid[GRID_SIZE * i + j][A_COLUMN] = RR.slab.a;
    The_Grid[GRID_SIZE * i + j][B_COLUMN] = RR.slab.b;
    The_Grid[GRID_SIZE * i + j][G_COLUMN] = RR.slab.g;
    The_Grid[GRID_SIZE * i + j][UR1_COLUMN] = ur1;
    The_Grid[GRID_SIZE * i + j][UT1_COLUMN] = ut1;
    The_Grid[GRID_SIZE * i + j][URU_COLUMN] = uru;
    The_Grid[GRID_SIZE * i + j][UTU_COLUMN] = utu;
    if (Debug(DEBUG_GRID)) {
      fprintf(stderr, "+␣%2d␣%2d␣", i, j);
      fprintf(stderr, "%10.5f␣%10.5f␣%10.5f␣|", RR.slab.a, RR.slab.b, RR.slab.g);
      fprintf(stderr, "%10.5f␣%10.5f␣|", MM.m_r, uru);
      fprintf(stderr, "%10.5f␣%10.5f␣\n", MM.m_t, utu);
    }
  }
```

This code is used in section 100.

136. This routine fills the grid with a proper set of values. With a little work, this routine could be made much faster by (1) only generating the phase function matrix once, (2) Making only one pass through the array for each albedo value, i.e., using the matrix left over from $b = 1$ to generate the solution for $b = 2$. Unfortunately this would require a complete revision of the *Calculate_Distance* routine. Fortunately, this routine should only need to be calculated once at the beginning of each run.

```
⟨ Prototype for Fill_AB_Grid 136 ⟩ ≡
  void Fill_AB_Grid(struct measure_type m, struct invert_type r)
```

This code is used in sections 100 and 137.

137. ⟨ Definition for *Fill_AB_Grid* 137 ⟩ ≡

```
⟨ Prototype for Fill_AB_Grid 136 ⟩
{
  int i, j;
  double a;
  double min_b = −8;      /* exp(-10) is smallest thickness */
  double max_b = +8;      /* exp(+8) is greatest thickness */
  if (Debug(DEBUG_GRID)) fprintf(stderr, "Filling␣AB␣grid\n");
  if (The_Grid ≡ Λ) Allocate_Grid(r.search);
  ⟨ Zero GG 143 ⟩
  Set_Calc_State(m, r);
  GG_g = RR.slab.g;
  for (i = 0; i < GRID_SIZE; i++) {
    double x = (double) i/(GRID_SIZE − 1.0);
    RR.slab.b = exp(min_b + (max_b − min_b) * x);
    for (j = 0; j < GRID_SIZE; j++) {
      ⟨ Generate next albedo using j 139 ⟩
      fill_grid_entry(i, j);
    }
  }
  The_Grid_Initialized = TRUE;
  The_Grid_Search = FIND_AB;
}
```

This code is used in section 100.

138. Now it seems that I must be a bit more subtle in choosing the range of albedos to use in the grid. Originally I just spaced them according to

$$a = 1 - \left[\frac{j-1}{n-1}\right]^3$$

where $1 \le j \le n$. Long ago it seems that I based things only on the square of the bracketed term, but I seem to remember that I was forced to change it from a square to a cube to get more global convergence.

So why am I rewriting this? Well, because it works very poorly for samples with small albedos. For example, when $n = 11$ then the values chosen for a are (1, .999, .992, .973, .936, .875, .784, .657, .488, .271, 0). Clearly very skewed towards high albedos.

I am considering a two part division. I'm not too sure how it should go. Let the first half be uniformly divided and the last half follow the cubic scheme given above. The list of values should then be (1, .996, .968, .892, 0.744, .5, .4, .3, .2, .1, 0).

Maybe it would be best if I just went back to a quadratic term. Who knows?

In the **if** statement below, note that it could read $j \ge k$ and still generate the same results.

⟨ Nonworking code 138 ⟩ ≡

```
k = floor((GRID_SIZE − 1)/2);
if (j > k) {
  a = 0.5 * (1 − (j − k − 1)/(GRID_SIZE − k − 1));
  RR.slab.a = a;
}
else {
  a = (j − 1.0)/(GRID_SIZE − k − 1);
  RR.slab.a = 1.0 − a * a * a/2;
}
```

139. Well, the above code did not work well. So I futzed around and sort of empirically ended up using the very simple method below. The only real difference from the previous method what that the method is now quadratic and not cubic.

```
⟨ Generate next albedo using j 139 ⟩ ≡
  a = (double) j/(GRID_SIZE − 1.0);
  if (a < 0.25) RR.slab.a = 1.0 − a ∗ a;
  else if (a > 0.75) RR.slab.a = (1.0 − a) ∗ (1.0 − a);
  else RR.slab.a = 1 − a;
```

See also section 140.

This code is used in sections 137 and 142.

140. Well, the above code has gaps. Here is an attempt to eliminate the gaps

```
⟨ Generate next albedo using j 139 ⟩ +≡
  a = (double) j/(GRID_SIZE − 1.0);
  RR.slab.a = (1.0 − a ∗ a) ∗ (1.0 − a) + (1.0 − a) ∗ (1.0 − a) ∗ a;
```

141. This is quite similar to *Fill_AB_Grid*, with the exception of the little shuffle I do at the beginning to figure out the optical thickness to use. The problem is that the optical thickness may not be known. If it is known then the only way that we could have gotten here is if the user dictated `FIND_AG` and specified b and only provided two measurements. Otherwise, the user must have made three measurements and the optical depth can be figured out from $m.m\_u$.

This routine could also be improved by not recalculating the anisotropy matrix for every point. But this would only end up being a minor performance enhancement if it were fixed.

```
⟨ Prototype for Fill_AG_Grid 141 ⟩ ≡
  void Fill_AG_Grid(struct measure_type m, struct invert_type r)
```

This code is used in sections 100 and 142.

142.

```
⟨ Definition for Fill_AG_Grid 142 ⟩ ≡
  ⟨ Prototype for Fill_AG_Grid 141 ⟩
  {
    int i, j;
    double a;
    if (Debug(DEBUG_GRID)) fprintf(stderr, "Filling␣AG␣grid\n");
    if (The_Grid ≡ Λ) Allocate_Grid(r.search);
    ⟨ Zero GG 143 ⟩
    Set_Calc_State(m, r);
    GG_b = r.slab.b;
    for (i = 0; i < GRID_SIZE; i++) {
      RR.slab.g = 0.9999 ∗ (2.0 ∗ i/(GRID_SIZE − 1.0) − 1.0);
      for (j = 0; j < GRID_SIZE; j++) {
        ⟨ Generate next albedo using j 139 ⟩
        fill_grid_entry(i, j);
      }
    }
    The_Grid_Initialized = TRUE;
    The_Grid_Search = FIND_AG;
  }
```

This code is used in section 100.

143.

⟨ Zero GG 143 ⟩ ≡
```
GG_a = 0.0;
GG_b = 0.0;
GG_g = 0.0;
GG_bs = 0.0;
GG_ba = 0.0;
```
This code is used in sections 137, 142, 145, 147, and 149.

144. This is quite similar to *Fill_AB_Grid*, with the exception of the that the albedo is held fixed while b and g are varied.

This routine could also be improved by not recalculating the anisotropy matrix for every point. But this would only end up being a minor performance enhancement if it were fixed.

⟨ Prototype for *Fill_BG_Grid* 144 ⟩ ≡
```
void Fill_BG_Grid(struct measure_type m, struct invert_type r)
```
This code is used in sections 101 and 145.

145. ⟨ Definition for *Fill_BG_Grid* 145 ⟩ ≡
```
⟨ Prototype for Fill_BG_Grid 144 ⟩
{
  int i, j;
  if (The_Grid ≡ Λ) Allocate_Grid(r.search);
  ⟨ Zero GG 143 ⟩
  if (Debug(DEBUG_GRID)) fprintf(stderr, "Filling␣BG␣grid\n");
  Set_Calc_State(m, r);
  RR.slab.b = 1.0/32.0;
  RR.slab.a = RR.default_a;
  GG_a = RR.slab.a;
  for (i = 0; i < GRID_SIZE; i++) {
    RR.slab.b *= 2;
    for (j = 0; j < GRID_SIZE; j++) {
      RR.slab.g = 0.9999 * (2.0 * j/(GRID_SIZE − 1.0) − 1.0);
      fill_grid_entry(i, j);
    }
  }
  The_Grid_Initialized = TRUE;
  The_Grid_Search = FIND_BG;
}
```
This code is used in section 100.

146. This is quite similar to *Fill_BG_Grid*, with the exception of the that the $b_s = \mu_s d$ is held fixed. Here b and g are varied on the usual grid, but the albedo is forced to take whatever value is needed to ensure that the scattering constant remains fixed.

⟨ Prototype for *Fill_BaG_Grid* 146 ⟩ ≡
```
void Fill_BaG_Grid(struct measure_type m, struct invert_type r)
```
This code is used in sections 101 and 147.

147. ⟨ Definition for *Fill_BaG_Grid* 147 ⟩ ≡

```
⟨ Prototype for Fill_BaG_Grid 146 ⟩
{
  int i, j;
  double bs, ba;
  if (The_Grid ≡ Λ) Allocate_Grid(r.search);
  ⟨ Zero GG 143 ⟩
  if (Debug(DEBUG_GRID)) fprintf(stderr, "Filling␣BaG␣grid\n");
  Set_Calc_State(m, r);
  ba = 1.0/32.0;
  bs = RR.default_bs;
  GG_bs = bs;
  for (i = 0; i < GRID_SIZE; i++) {
    ba *= 2;
    ba = exp((double) i/(GRID_SIZE − 1.0) ∗ log(1024.0))/16.0;
    RR.slab.b = ba + bs;
    if (RR.slab.b > 0) RR.slab.a = bs/RR.slab.b;
    else RR.slab.a = 0;
    for (j = 0; j < GRID_SIZE; j++) {
      RR.slab.g = 0.9999 ∗ (2.0 ∗ j/(GRID_SIZE − 1.0) − 1.0);
      fill_grid_entry(i, j);
    }
  }
  The_Grid_Initialized = TRUE;
  The_Grid_Search = FIND_BaG;
}
```

This code is used in section 100.

148. Very similiar to the above routine. The value of $b_a = \mu_a d$ is held constant.

⟨ Prototype for *Fill_BsG_Grid* 148 ⟩ ≡

```
void Fill_BsG_Grid(struct measure_type m, struct invert_type r)
```

This code is used in sections 101 and 149.

149. ⟨ Definition for *Fill_BsG_Grid* 149 ⟩ ≡

```
⟨ Prototype for Fill_BsG_Grid 148 ⟩
{
  int i, j;
  double bs, ba;
  if (The_Grid ≡ Λ) Allocate_Grid(r.search);
  ⟨ Zero GG 143 ⟩
  Set_Calc_State(m, r);
  bs = 1.0/32.0;
  ba = RR.default_ba;
  GG_ba = ba;
  for (i = 0; i < GRID_SIZE; i++) {
    bs *= 2;
    RR.slab.b = ba + bs;
    if (RR.slab.b > 0) RR.slab.a = bs/RR.slab.b;
    else RR.slab.a = 0;
    for (j = 0; j < GRID_SIZE; j++) {
      RR.slab.g = 0.9999 * (2.0 * j/(GRID_SIZE − 1.0) − 1.0);
      fill_grid_entry(i, j);
    }
  }
  The_Grid_Initialized = TRUE;
  The_Grid_Search = FIND_BsG;
}
```

This code is used in section 100.

150. ⟨ Prototype for *Fill_Grid* 150 ⟩ ≡

```
void Fill_Grid(struct measure_type m, struct invert_type r)
```

This code is used in sections 101 and 151.

151. ⟨ Definition for *Fill_Grid* 151 ⟩ ≡

```
⟨ Prototype for Fill_Grid 150 ⟩
{
  if (¬Same_Calc_State(m, r)) {
    switch (r.search) {
    case FIND_AB:
      if (Debug(DEBUG_SEARCH)) fprintf(stderr, "filling␣AB␣Grid\n");
      Fill_AB_Grid(m, r);
      break;
    case FIND_AG:
      if (Debug(DEBUG_SEARCH)) fprintf(stderr, "filling␣AG␣Grid\n");
      Fill_AG_Grid(m, r);
      break;
    case FIND_BG:
      if (Debug(DEBUG_SEARCH)) fprintf(stderr, "filling␣BG␣Grid\n");
      Fill_BG_Grid(m, r);
      break;
    case FIND_BaG:
      if (Debug(DEBUG_SEARCH)) fprintf(stderr, "filling␣BaG␣Grid\n");
      Fill_BaG_Grid(m, r);
      break;
    case FIND_BsG:
      if (Debug(DEBUG_SEARCH)) fprintf(stderr, "filling␣BsG␣Grid\n");
      Fill_BsG_Grid(m, r);
      break;
    default: AD_error("Attempt␣to␣fill␣grid␣for␣unusual␣search␣case.");
    }
  }
  Get_Calc_State(&MGRID, &RGRID);
}
```

This code is used in section 100.

152. Calculating R and T.

Calculate_Distance returns the distance between the measured values in `MM` and the calculated values for the current guess at the optical properties. It assumes that the everything in the local variables `MM` and `RR` have been set appropriately.

⟨ Prototype for *Calculate_Distance* 152 ⟩ ≡

```
void Calculate_Distance(double *M_R, double *M_T, double *deviation)
```

This code is used in sections 101 and 153.

153. ⟨ Definition for *Calculate_Distance* 153 ⟩ ≡

```
⟨ Prototype for Calculate_Distance 152 ⟩
{
  double Rc, Tc, ur1, ut1, uru, utu;
  if (RR.slab.b ≤ 1 · 10^−6) RR.slab.b = 1 · 10^−6;
  if (Debug(DEBUG_EVERY_CALC))
    fprintf(stderr, "a=%8.5f␣b=%10.5f␣g=%8.5f␣", RR.slab.a, RR.slab.b, RR.slab.g);
  RT(RR.method.quad_pts, &RR.slab, &ur1, &ut1, &uru, &utu);
  if (Debug(DEBUG_EVERY_CALC))
    fprintf(stderr, "ur1=%8.5f␣ut1=%8.5f␣(not␣M_R␣and␣M_T!)\n", ur1, ut1);
  Sp_mu_RT(RR.slab.n_top_slide, RR.slab.n_slab, RR.slab.n_bottom_slide, RR.slab.b_top_slide, RR.slab.b,
      RR.slab.b_bottom_slide, RR.slab.cos_angle, &Rc, &Tc);
  if ((¬CALCULATING_GRID ∧ Debug(DEBUG_ITERATIONS)) ∨ (CALCULATING_GRID ∧ Debug(DEBUG_GRID)))
    fprintf(stderr, "␣␣␣␣␣␣␣␣");
  Calculate_Distance_With_Corrections(ur1, ut1, Rc, Tc, uru, utu, M_R, M_T, deviation);
}
```

This code is used in section 100.

154. ⟨ Prototype for *Calculate_Grid_Distance* 154 ⟩ ≡

```
double Calculate_Grid_Distance(int i, int j)
```

This code is used in sections 101 and 155.

155. ⟨ Definition for *Calculate_Grid_Distance* 155 ⟩ ≡

```
⟨ Prototype for Calculate_Grid_Distance 154 ⟩
{
  double ur1, ut1, uru, utu, Rc, Tc, b, dev, LR, LT;
  if (Debug(DEBUG_GRID)) fprintf(stderr, "g␣%2d␣%2d␣", i, j);
  b = The_Grid[GRID_SIZE * i + j][B_COLUMN];
  ur1 = The_Grid[GRID_SIZE * i + j][UR1_COLUMN];
  ut1 = The_Grid[GRID_SIZE * i + j][UT1_COLUMN];
  uru = The_Grid[GRID_SIZE * i + j][URU_COLUMN];
  utu = The_Grid[GRID_SIZE * i + j][UTU_COLUMN];
  RR.slab.a = The_Grid[GRID_SIZE * i + j][A_COLUMN];
  RR.slab.b = The_Grid[GRID_SIZE * i + j][B_COLUMN];
  RR.slab.g = The_Grid[GRID_SIZE * i + j][G_COLUMN];
  Sp_mu_RT(RR.slab.n_top_slide, RR.slab.n_slab, RR.slab.n_bottom_slide, RR.slab.b_top_slide, b,
      RR.slab.b_bottom_slide, RR.slab.cos_angle, &Rc, &Tc);
  CALCULATING_GRID = 1;
  Calculate_Distance_With_Corrections(ur1, ut1, Rc, Tc, uru, utu, &LR, &LT, &dev);
  CALCULATING_GRID = 0;
  return dev;
}
```

This code is used in section 100.

156. This is the routine that actually finds the distance. I have factored this part out so that it can be used in the *Near_Grid_Point* routine.

Rc and *Tc* refer to the ballistic reflection and transmission.

The only tricky part is to remember that the we are trying to match the measured values. The measured values are affected by sphere parameters and light loss. Since the values `UR1` and `UT1` are for an infinite slab sample with no light loss, the light loss out the edges must be subtracted. It is these values that are used with the sphere formulas to convert the modified `UR1` and `UT1` to values for `*M_R` and `*M_T`.

```
⟨ Prototype for Calculate_Distance_With_Corrections 156 ⟩ ≡
  void Calculate_Distance_With_Corrections(double UR1, double UT1, double Rc, double Tc, double
      URU, double UTU, double *M_R, double *M_T, double *dev)
```

This code is used in sections 101 and 157.

157.
```
⟨ Definition for Calculate_Distance_With_Corrections 157 ⟩ ≡
  ⟨ Prototype for Calculate_Distance_With_Corrections 156 ⟩
  {
    double R_direct, T_direct, R_diffuse, T_diffuse;
    R_diffuse = URU − MM.uru_lost;
    T_diffuse = UTU − MM.utu_lost;
    R_direct = UR1 − MM.ur1_lost − (1 − MM.sphere_with_rc) * Rc;
    T_direct = UT1 − MM.ut1_lost − (1 − MM.sphere_with_tc) * Tc;
    if (FRACTION) {
      if (UR1 − Rc > 0.01) R_direct = UR1 − MM.ur1_lost * (UR1 − Rc) − (1 − MM.sphere_with_rc) * Rc;
      if (UT1 − Tc > 0.01) T_direct = UT1 − MM.ut1_lost * (UT1 − Tc) − (1 − MM.sphere_with_tc) * Tc;
    }
    switch (MM.num_spheres) {
    case 0: ⟨ Calc M_R and M_T for no spheres 158 ⟩
      break;
    case 1: case −2:
      if (MM.method ≡ COMPARISON) ⟨ Calc M_R and M_T for dual beam sphere 160 ⟩
      else ⟨ Calc M_R and M_T for single beam sphere 159 ⟩
      break;
    case 2: ⟨ Calc M_R and M_T for two spheres 161 ⟩
      break;
    }
    ⟨ Calculate the deviation 162 ⟩
    ⟨ Print diagnostics 165 ⟩
  }
```

This code is used in section 100.

158. If no spheres were used in the measurement, then presumably the measured values are the reflection and transmission. Consequently, we just acertain what the irradiance was and whether the specular reflection ports were blocked and proceed accordingly. Note that blocking the ports does not have much meaning unless the light is collimated, and therefore the reflection and transmission is only modified for collimated irradiance.

```
⟨ Calc M_R and M_T for no spheres 158 ⟩ ≡
  *M_R = R_direct;
  *M_T = T_direct;
```

This code is used in section 157.

159. The direct incident power is $(1-f)P$. The reflected power will be $(1-f)r_s^{\text{direct}}P$. Since baffles ensure that the light cannot reach the detector, we must bounce the light off the sphere walls to use to above gain formulas. The contribution will then be $(1-f)r_s^{\text{direct}}(1-a_e)r_wP$. The measured power will be

$$P_d = a_d(1-a_e)r_w[(1-f)r_s^{\text{direct}} + fr_w]P \cdot G(r_s)$$

Similarly the power falling on the detector measuring transmitted light is

$$P'_d = a'_d t_s^{\text{direct}} r'_w (1-a'_e) P \cdot G'(r_s)$$

when the 'entrance' port in the transmission sphere is closed, $a'_e = 0$.

The normalized sphere measurements are

$$M_R = r_{\text{std}} \cdot \frac{R(r_s^{\text{direct}}, r_s) - R(0,0)}{R(r_{\text{std}}, r_{\text{std}}) - R(0,0)}$$

and

$$M_T = t_{\text{std}} \cdot \frac{T(t_s^{\text{direct}}, r_s) - T(0,0)}{T(t_{\text{std}}, r_{\text{std}}) - T(0,0)}$$

⟨ Calc `M_R` and `M_T` for single beam sphere 159 ⟩ ≡

```
{
  double P_std, P_d, P_0;
  double G, G_0, G_std, GP_std, GP;
  G_0 = Gain(REFLECTION_SPHERE, MM, 0.0);
  G = Gain(REFLECTION_SPHERE, MM, R_diffuse);
  G_std = Gain(REFLECTION_SPHERE, MM, MM.rstd_r);
  P_d = G * (R_direct * (1 - MM.f_r) + MM.f_r * MM.rw_r);
  P_std = G_std * (MM.rstd_r * (1 - MM.f_r) + MM.f_r * MM.rw_r);
  P_0 = G_0 * (MM.f_r * MM.rw_r);
  *M_R = MM.rstd_r * (P_d - P_0)/(P_std - P_0);
  GP = Gain(TRANSMISSION_SPHERE, MM, R_diffuse);
  GP_std = Gain(TRANSMISSION_SPHERE, MM, 0.0);
  *M_T = T_direct * GP/GP_std;
}
```

This code is used in section 157.

160. The dual beam case is different because the sphere efficiency is equivalent for measurement of light hitting the sample first or hitting the reference standard first. The dual beam measurement should report the ratio of these two reflectance measurements, thereby eliminating the need to calculate the gain completely. The same holds when no sample is present.

The normalized reflectance measurement (the difference between dual beam measurement for a port with the sample and with nothing) is

$$M_R = r_{\rm std} \cdot \frac{(1-f) r_s^{\rm direct} + f r_w}{(1-f') r_{\rm std} - f' r_w} - r_{\rm std} \cdot \frac{(1-f)(0) + f r_w}{(1-f') r_{\rm std} - f' r_w}$$

or

$$M_R = \frac{(1-f) r_s^{\rm direct}}{(1-f') - f' r_w / r_{\rm std}}$$

When $f = f' = 1$, then $M_R = 1$ no matter what the reflectance is. (Leave it in this form to avoid division by zero when $f = 1$.)

The normalized transmittance is simply $t_s^{\rm direct}$.

When $f = 0$ then this result is essentially the same as the no spheres result (because no sphere corrections are needed). However if the number of spheres is zero, then no lost light calculations are made and therefore that is a potential error.

⟨ Calc `M_R` and `M_T` for dual beam sphere 160 ⟩ ≡

```
{
  *M_R = (1 − MM.f_r) * R_direct/((1 − MM.f_r) + MM.f_r * MM.rw_r/MM.rstd_r);
  *M_T = T_direct;
}
```

This code is used in section 157.

161. When two integrating spheres are present then the double integrating sphere formulas are slightly more complicated.

I am not sure what it means when *rstd_t* is not unity.

The normalized sphere measurements for two spheres are

$$M_R = \frac{R(r_s^{\rm direct}, r_s, t_s^{\rm direct}, t_s) - R(0,0,0,0)}{R(r_{\rm std}, r_{\rm std}, 0, 0) - R(0,0,0,0)}$$

and

$$M_T = \frac{T(r_s^{\rm direct}, r_s, t_s^{\rm direct}, t_s) - T(0,0,0,0)}{T(0,0,1,1) - T(0,0,0,0)}$$

Note that `R_0` and `T_0` will be zero unless one has explicitly set the fraction $m.f\_r$ ore $m.f\_t$ to be non-zero.

⟨ Calc `M_R` and `M_T` for two spheres 161 ⟩ ≡

```
{
  double R_0, T_0;
  R_0 = Two_Sphere_R(MM, 0, 0, 0, 0);
  T_0 = Two_Sphere_T(MM, 0, 0, 0, 0);
  *M_R = (Two_Sphere_R(MM, R_direct, R_diffuse, T_direct, T_diffuse) − R_0)/(Two_Sphere_R(MM,
      MM.rstd_r, MM.rstd_r, 0, 0) − R_0);
  *M_T = (Two_Sphere_T(MM, R_direct, R_diffuse, T_direct, T_diffuse) − T_0)/(Two_Sphere_T(MM, 0, 0, 1,
      1) − T_0);
}
```

This code is used in section 157.

162. There are at least three things that need to be considered here. First, the number of measurements. Second, is the metric is relative or absolute. And third, is the albedo fixed at zero which means that the transmission measurement should be used instead of the reflection measurement.

```
⟨ Calculate the deviation 162 ⟩ ≡
  if (RR.search ≡ FIND_A ∨ RR.search ≡ FIND_G ∨ RR.search ≡ FIND_B ∨ RR.search ≡ FIND_Bs ∨ RR.search ≡
        FIND_Ba) {
    ⟨ One parameter deviation 163 ⟩
  }
  else {
    ⟨ Two parameter deviation 164 ⟩
  }
```

This code is used in section 157.

163. This part was slightly tricky. The crux of the problem was to decide if the transmission or the reflection was trustworthy. After looking a bunches of measurements, I decided that the transmission measurement was almost always more reliable. So when there is just a single measurement known, then use the total transmission if it exists.

```
⟨ One parameter deviation 163 ⟩ ≡
  if (MM.m_t > 0) {
    if (RR.metric ≡ RELATIVE) *dev = fabs(MM.m_t − *M_T)/(MM.m_t + ABIT);
    else *dev = fabs(MM.m_t − *M_T);
  }
  else {
    if (RR.metric ≡ RELATIVE) *dev = fabs(MM.m_r − *M_R)/(MM.m_r + ABIT);
    else *dev = fabs(MM.m_r − *M_R);
  }
```

This code is used in section 162.

164. This stuff happens when we are doing two parameter searches. In these cases there should be information in both R and T. The distance should be calculated using the deviation from both. The albedo stuff might be able to be take out. We'll see.

```
⟨ Two parameter deviation 164 ⟩ ≡
  if (RR.metric ≡ RELATIVE) {
    *dev = 0;
    if (MM.m_t > ABIT) *dev = T_TRUST_FACTOR * fabs(MM.m_t − *M_T)/(MM.m_t + ABIT);
    if (RR.default_a ≠ 0) *dev += fabs(MM.m_r − *M_R)/(MM.m_r + ABIT);
  }
  else {
    *dev = T_TRUST_FACTOR * fabs(MM.m_t − *M_T);
    if (RR.default_a ≠ 0) *dev += fabs(MM.m_r − *M_R);
  }
```

This code is used in section 162.

165. This is here so that I can figure out why the program is not converging. This is a little convoluted so that the global constants at the top of this file interact properly.

```
⟨ Print diagnostics 165 ⟩ ≡
  if ((Debug(DEBUG_ITERATIONS) ∧ ¬CALCULATING_GRID) ∨ (Debug(DEBUG_GRID) ∧ CALCULATING_GRID)) {
    static int once = 0;
    if (once ≡ 0) {
      fprintf(stderr, "%10s␣%10s␣%10s␣|%10s␣%10s␣|%10s␣%10s␣|%10s\n", "a", "b", "g", "m_r", "fit",
          "m_t", "fit", "delta");
      once = 1;
    }
    fprintf(stderr, "%10.5f␣%10.5f␣%10.5f␣|", RR.slab.a, RR.slab.b, RR.slab.g);
    fprintf(stderr, "%10.5f␣%10.5f␣|", MM.m_r, *M_R);
    fprintf(stderr, "%10.5f␣%10.5f␣|", MM.m_t, *M_T);
    fprintf(stderr, "%10.5f␣\n", *dev);
  }
```

This code is used in section 157.

166.
```
⟨ Prototype for Find_AG_fn 166 ⟩ ≡
  double Find_AG_fn(double x[])
```

This code is used in sections 101 and 167.

167.
```
⟨ Definition for Find_AG_fn 167 ⟩ ≡
  ⟨ Prototype for Find_AG_fn 166 ⟩
  {
    double m_r, m_t, deviation;
    RR.slab.a = acalc2a(x[1]);
    RR.slab.g = gcalc2g(x[2]);
    Calculate_Distance(&m_r, &m_t, &deviation);
    return deviation;
  }
```

This code is used in section 100.

168.
```
⟨ Prototype for Find_AB_fn 168 ⟩ ≡
  double Find_AB_fn(double x[])
```

This code is used in sections 101 and 169.

169.
```
⟨ Definition for Find_AB_fn 169 ⟩ ≡
  ⟨ Prototype for Find_AB_fn 168 ⟩
  {
    double m_r, m_t, deviation;
    RR.slab.a = acalc2a(x[1]);
    RR.slab.b = bcalc2b(x[2]);
    Calculate_Distance(&m_r, &m_t, &deviation);
    return deviation;
  }
```

This code is used in section 100.

170.
```
⟨ Prototype for Find_Ba_fn 170 ⟩ ≡
  double Find_Ba_fn(double x)
```

This code is used in sections 101 and 171.

171. This is tricky only because the value in RR.*slab.b* is used to hold the value of *bs* or $d \cdot \mu_s$. It must be switched to the correct value for the optical thickness and then switched back at the end of the routine.

```
⟨ Definition for Find_Ba_fn 171 ⟩ ≡
  ⟨ Prototype for Find_Ba_fn 170 ⟩
  {
    double m_r, m_t, deviation, ba, bs;
    bs = RR.slab.b;
    ba = bcalc2b(x);
    RR.slab.b = ba + bs;      /* unswindle */
    RR.slab.a = bs/(ba + bs);
    Calculate_Distance(&m_r, &m_t, &deviation);
    RR.slab.b = bs;      /* swindle */
    return deviation;
  }
```

This code is used in section 100.

172. See the comments for the *Find_Ba_fn* routine above. Play the same trick but use *ba*.

```
⟨ Prototype for Find_Bs_fn 172 ⟩ ≡
  double Find_Bs_fn(double x)
```

This code is used in sections 101 and 173.

173.

```
⟨ Definition for Find_Bs_fn 173 ⟩ ≡
  ⟨ Prototype for Find_Bs_fn 172 ⟩
  {
    double m_r, m_t, deviation, ba, bs;
    ba = RR.slab.b;      /* unswindle */
    bs = bcalc2b(x);
    RR.slab.b = ba + bs;
    RR.slab.a = bs/(ba + bs);
    Calculate_Distance(&m_r, &m_t, &deviation);
    RR.slab.b = ba;      /* swindle */
    return deviation;
  }
```

This code is used in section 100.

174.

```
⟨ Prototype for Find_A_fn 174 ⟩ ≡
  double Find_A_fn(double x)
```

This code is used in sections 101 and 175.

175.

```
⟨ Definition for Find_A_fn 175 ⟩ ≡
  ⟨ Prototype for Find_A_fn 174 ⟩
  {
    double m_r, m_t, deviation;
    RR.slab.a = acalc2a(x);
    Calculate_Distance(&m_r, &m_t, &deviation);
    return deviation;
  }
```

This code is used in section 100.

176. ⟨ Prototype for *Find_B_fn* 176 ⟩ ≡
double *Find_B_fn*(**double** *x*)
This code is used in sections 101 and 177.

177. ⟨ Definition for *Find_B_fn* 177 ⟩ ≡
⟨ Prototype for *Find_B_fn* 176 ⟩
{
double *m_r*, *m_t*, *deviation*;
RR.*slab*.*b* = *bcalc2b*(*x*);
Calculate_Distance(&*m_r*, &*m_t*, &*deviation*);
return *deviation*;
}
This code is used in section 100.

178. ⟨ Prototype for *Find_G_fn* 178 ⟩ ≡
double *Find_G_fn*(**double** *x*)
This code is used in sections 101 and 179.

179. ⟨ Definition for *Find_G_fn* 179 ⟩ ≡
⟨ Prototype for *Find_G_fn* 178 ⟩
{
double *m_r*, *m_t*, *deviation*;
RR.*slab*.*g* = *gcalc2g*(*x*);
Calculate_Distance(&*m_r*, &*m_t*, &*deviation*);
return *deviation*;
}
This code is used in section 100.

180. ⟨ Prototype for *Find_BG_fn* 180 ⟩ ≡
double *Find_BG_fn*(**double** *x*[])
This code is used in sections 101 and 181.

181. ⟨ Definition for *Find_BG_fn* 181 ⟩ ≡
⟨ Prototype for *Find_BG_fn* 180 ⟩
{
double *m_r*, *m_t*, *deviation*;
RR.*slab*.*b* = *bcalc2b*(*x*[1]);
RR.*slab*.*g* = *gcalc2g*(*x*[2]);
RR.*slab*.*a* = RR.*default_a*;
Calculate_Distance(&*m_r*, &*m_t*, &*deviation*);
return *deviation*;
}
This code is used in section 100.

182. For this function the first term $x[1]$ will contain the value of $\mu_s d$, the second term will contain the anisotropy. Of course the first term is in the bizarre calculation space and needs to be translated back into normal terms before use. We just at the scattering back on and voilá we have a useable value for the optical depth.

⟨ Prototype for *Find_BaG_fn* 182 ⟩ ≡
double *Find_BaG_fn*(**double** *x*[])
This code is used in sections 101 and 183.

183. ⟨ Definition for *Find_BaG_fn* 183 ⟩ ≡

```
⟨ Prototype for Find_BaG_fn 182 ⟩
{
  double m_r, m_t, deviation;
  RR.slab.b = bcalc2b(x[1]) + RR.default_bs;
  if (RR.slab.b ≤ 0) RR.slab.a = 0;
  else RR.slab.a = RR.default_bs/RR.slab.b;
  RR.slab.g = gcalc2g(x[2]);
  Calculate_Distance(&m_r, &m_t, &deviation);
  return deviation;
}
```

This code is used in section 100.

184. ⟨ Prototype for *Find_BsG_fn* 184 ⟩ ≡

```
double Find_BsG_fn(double x[])
```

This code is used in sections 101 and 185.

185. ⟨ Definition for *Find_BsG_fn* 185 ⟩ ≡

```
⟨ Prototype for Find_BsG_fn 184 ⟩
{
  double m_r, m_t, deviation;
  RR.slab.b = bcalc2b(x[1]) + RR.default_ba;
  if (RR.slab.b ≤ 0) RR.slab.a = 0;
  else RR.slab.a = 1.0 − RR.default_ba/RR.slab.b;
  RR.slab.g = gcalc2g(x[2]);
  Calculate_Distance(&m_r, &m_t, &deviation);
  return deviation;
}
```

This code is used in section 100.

186. Routine to figure out if the light loss exceeds what is physically possible. Returns the descrepancy betweent the current values and the maximum possible values for the the measurements m_r and m_t.

⟨ Prototype for *maxloss* 186 ⟩ ≡

```
double maxloss(double f)
```

This code is used in sections 101 and 187.

187. ⟨ Definition for *maxloss* 187 ⟩ ≡

```
⟨ Prototype for maxloss 186 ⟩
{
  struct measure_type m_old;
  struct invert_type r_old;
  double m_r, m_t, deviation;
  Get_Calc_State(&m_old, &r_old);
  RR.slab.a = 1.0;
  MM.ur1_lost *= f;
  MM.ut1_lost *= f;
  Calculate_Distance(&m_r, &m_t, &deviation);
  Set_Calc_State(m_old, r_old);
  deviation = ((MM.m_r + MM.m_t) − (m_r + m_t));
  return deviation;
}
```

This code is used in section 100.

188. This checks the two light loss values *ur1_loss* and *ut1_loss* to see if they exceed what is physically possible. If they do, then these values are replaced by a couple that are the maximum possible for the current values in m and r.

```
⟨ Prototype for Max_Light_Loss 188 ⟩ ≡
  void Max_Light_Loss(struct measure_type m, struct invert_type r, double *ur1_loss, double
      *ut1_loss)
```

This code is used in sections 101 and 189.

189. ⟨ Definition for *Max_Light_Loss* 189 ⟩ ≡

```
⟨ Prototype for Max_Light_Loss 188 ⟩
{
  struct measure_type m_old;
  struct invert_type r_old;
  *ur1_loss = m.ur1_lost;
  *ut1_loss = m.ut1_lost;
  if (Debug(DEBUG_LOST_LIGHT))
    fprintf(stderr, "\nlost␣before␣ur1=%7.5f,␣ut1=%7.5f\n", *ur1_loss, *ut1_loss);
  Get_Calc_State(&m_old, &r_old);
  Set_Calc_State(m, r);
  if (maxloss(1.0) * maxloss(0.0) < 0) {
    double frac;
    frac = zbrent(maxloss, 0.00, 1.0, 0.001);
    *ur1_loss = m.ur1_lost * frac;
    *ut1_loss = m.ut1_lost * frac;
  }
  Set_Calc_State(m_old, r_old);
  if (Debug(DEBUG_LOST_LIGHT))
    fprintf(stderr, "lost␣after␣␣ur1=%7.5f,␣ut1=%7.5f\n", *ur1_loss, *ut1_loss);
}
```

This code is used in section 100.

190. this is currently unused

```
⟨ Unused diffusion fragment 190 ⟩ ≡
  static void DE_RT(int nfluxes, AD_slab_type slab, double *UR1, double *UT1, double *URU, double
      *UTU)
  {
    slabtype s;
    double rp, tp, rs, ts;
    s.f = slab.g * slab.g;
    s.gprime = slab.g/(1 + slab.g);
    s.aprime = (1 − s.f) * slab.a/(1 − slab.a * s.f);
    s.bprime = (1 − slab.a * s.f) * slab.b;
    s.boundary_method = Egan;
    s.n_top = slab.n_slab;
    s.n_bottom = slab.n_slab;
    s.slide_top = slab.n_top_slide;
    s.slide_bottom = slab.n_bottom_slide;
    s.FO = 1/pi;
    s.depth = 0.0;
    s.Exact_coll_flag = false;
    if (MM.illumination ≡ collimated) {
      compute_R_and_T(&s, 1.0, &rp, &rs, &tp, &ts);
      *UR1 = rp + rs;
      *UT1 = tp + ts;
      *URU = 0.0;
      *UTU = 0.0;
      return;
    }
    quad_Dif_Calc_R_and_T(&s, &rp, &rs, &tp, &ts);
    *URU = rp + rs;
    *UTU = tp + ts;
    *UR1 = 0.0;
    *UT1 = 0.0;
  }
```

191. IAD Find. March 1995. Incorporated the *quick_guess* algorithm for low albedos.

```
⟨ iad_find.c 191 ⟩ ≡
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "ad_globl.h"
#include "nr_util.h"
#include "nr_mnbrk.h"
#include "nr_brent.h"
#include "nr_amoeb.h"
#include "iad_type.h"
#include "iad_util.h"
#include "iad_calc.h"
#include "iad_find.h"
#define NUMBER_OF_GUESSES 10
  guess_type guess[NUMBER_OF_GUESSES];
  int compare_guesses(const void *p1, const void *p2)
  {
    guess_type *g1 = (guess_type *) p1;
    guess_type *g2 = (guess_type *) p2;
    if (g1→distance < g2→distance) return -1;
    else if (g1→distance ≡ g2→distance) return 0;
    else return 1;
  }
  ⟨ Definition for U_Find_Ba 205 ⟩
  ⟨ Definition for U_Find_Bs 203 ⟩
  ⟨ Definition for U_Find_A 207 ⟩
  ⟨ Definition for U_Find_B 211 ⟩
  ⟨ Definition for U_Find_G 209 ⟩
  ⟨ Definition for U_Find_AG 214 ⟩
  ⟨ Definition for U_Find_AB 194 ⟩
  ⟨ Definition for U_Find_BG 219 ⟩
  ⟨ Definition for U_Find_BaG 225 ⟩
  ⟨ Definition for U_Find_BsG 230 ⟩
```

192. All the information that needs to be written to the header file `iad_find.h`. This eliminates the need to maintain a set of header files as well.

```
⟨ iad_find.h 192 ⟩ ≡
  ⟨ Prototype for U_Find_Ba 204 ⟩;
  ⟨ Prototype for U_Find_Bs 202 ⟩;
  ⟨ Prototype for U_Find_A 206 ⟩;
  ⟨ Prototype for U_Find_B 210 ⟩;
  ⟨ Prototype for U_Find_G 208 ⟩;
  ⟨ Prototype for U_Find_AG 213 ⟩;
  ⟨ Prototype for U_Find_AB 193 ⟩;
  ⟨ Prototype for U_Find_BG 218 ⟩;
  ⟨ Prototype for U_Find_BaG 224 ⟩;
  ⟨ Prototype for U_Find_BsG 229 ⟩;
```

193. Fixed Anisotropy.
This is the most common case.

```
⟨ Prototype for U_Find_AB 193 ⟩ ≡
  void U_Find_AB(struct measure_type m, struct invert_type *r)
```

This code is used in sections 192 and 194.

194.
```
⟨ Definition for U_Find_AB 194 ⟩ ≡
  ⟨ Prototype for U_Find_AB 193 ⟩
  {
    if (Debug(DEBUG_SEARCH)) {
      fprintf(stderr, "In U_Find_AB");
      fprintf(stderr, " (mu=%6.4f)", r->slab.cos_angle);
      if (r->default_g ≠ UNINITIALIZED) fprintf(stderr, "  default_g = %8.5f", r->default_g);
      fprintf(stderr, "\n");
    }
    ⟨ Allocate local simplex variables 195 ⟩
    r->slab.g = (r->default_g ≡ UNINITIALIZED) ? 0 : r->default_g;
    Set_Calc_State(m, *r);
    ⟨ Get the initial a, b, and g 196 ⟩
    ⟨ Initialize the nodes of the a and b simplex 197 ⟩
    ⟨ Evaluate the a and b simplex at the nodes 198 ⟩
    amoeba(p, y, 2, r->tolerance, Find_AB_fn, &r->iterations);
    ⟨ Choose the best node of the a and b simplex 199 ⟩
    ⟨ Free simplex data structures 201 ⟩
    ⟨ Put final values in result 200 ⟩
  }
```

This code is used in section 191.

195. To use the simplex algorithm, we need to vectors and a matrix.

```
⟨ Allocate local simplex variables 195 ⟩ ≡
  int i, i_best, j_best;
  double *x, *y, **p;
  x = dvector(1, 2);
  y = dvector(1, 3);
  p = dmatrix(1, 3, 1, 2);
```

This code is used in sections 194, 214, 219, 225, and 230.

196. Just get the optimal optical properties to start the search process.

I had to add the line that tests to make sure the albedo is greater than 0.2 because the grid just does not work so well in this case. The problem is that for low albedos there is really very little information about the anisotropy available. This change was also made in the analagous code for a and b.

```
⟨ Get the initial a, b, and g 196 ⟩ ≡
  {     /* double a3,b3,g3; */
    size_t count = NUMBER_OF_GUESSES;     /* distance to last result */
    abg_distance(r→slab.a, r→slab.b, r→slab.g, &(guess[0]));
    if (¬Valid_Grid(m, r→search)) Fill_Grid(m, *r);     /* distance to nearest grid point */
    Near_Grid_Points(m.m_r, m.m_t, r→search, &i_best, &j_best);
    Grid_ABG(i_best, j_best, &(guess[1]));
    Grid_ABG(i_best + 1, j_best, &(guess[2]));
    Grid_ABG(i_best − 1, j_best, &(guess[3]));
    Grid_ABG(i_best, j_best + 1, &(guess[4]));
    Grid_ABG(i_best, j_best − 1, &(guess[5]));
    Grid_ABG(i_best + 1, j_best + 1, &(guess[6]));
    Grid_ABG(i_best − 1, j_best − 1, &(guess[7]));
    Grid_ABG(i_best + 1, j_best − 1, &(guess[8]));
    Grid_ABG(i_best − 1, j_best + 1, &(guess[9]));
    qsort((void *) guess, count, sizeof(guess_type), compare_guesses);
    if (Debug(DEBUG_BEST_GUESS)) {
      int k;
      fprintf(stderr, "after\n");
      for (k = 0; k ≤ 6; k++) {
        fprintf(stderr, "%3d␣␣", k);
        fprintf(stderr, "%10.5f␣", guess[k].a);
        fprintf(stderr, "%10.5f␣", guess[k].b);
        fprintf(stderr, "%10.5f␣", guess[k].g);
        fprintf(stderr, "%10.5f\n", guess[k].distance);
      }
    }
  }
```

This code is used in sections 194, 214, 219, 225, and 230.

197. ⟨ Initialize the nodes of the a and b simplex 197 ⟩ ≡

```
{
  int k, kk;
  p[1][1] = a2acalc(guess[0].a);
  p[1][2] = b2bcalc(guess[0].b);
  for (k = 1; k < 7; k++) {
    if (guess[0].a ≠ guess[k].a) break;
  }
  p[2][1] = a2acalc(guess[k].a);
  p[2][2] = b2bcalc(guess[k].b);
  for (kk = 1; kk < 7; kk++) {
    if (guess[0].b ≠ guess[kk].b ∧ guess[k].b ≠ guess[kk].b) break;
  }
  p[3][1] = a2acalc(guess[kk].a);
  p[3][2] = b2bcalc(guess[kk].b);
  if (Debug(DEBUG_BEST_GUESS)) {
    fprintf(stderr, "guess␣1");
    fprintf(stderr, "%10.5f␣", guess[0].a);
    fprintf(stderr, "%10.5f␣", guess[0].b);
    fprintf(stderr, "%10.5f␣", guess[0].g);
    fprintf(stderr, "%10.5f\n", guess[0].distance);
    fprintf(stderr, "guess␣2");
    fprintf(stderr, "%10.5f␣", guess[k].a);
    fprintf(stderr, "%10.5f␣", guess[k].b);
    fprintf(stderr, "%10.5f␣", guess[k].g);
    fprintf(stderr, "%10.5f\n", guess[k].distance);
    fprintf(stderr, "guess␣3");
    fprintf(stderr, "%10.5f␣", guess[kk].a);
    fprintf(stderr, "%10.5f␣", guess[kk].b);
    fprintf(stderr, "%10.5f␣", guess[kk].g);
    fprintf(stderr, "%10.5f\n", guess[kk].distance);
  }
}
```

This code is used in section 194.

198. ⟨ Evaluate the a and b simplex at the nodes 198 ⟩ ≡

```
for (i = 1; i ≤ 3; i++) {
  x[1] = p[i][1];
  x[2] = p[i][2];
  y[i] = Find_AB_fn(x);
}
```

This code is used in section 194.

199. ⟨ Choose the best node of the a and b simplex 199 ⟩ ≡

```
r→final_distance = 10;
for (i = 1; i ≤ 3; i++) {
  if (y[i] < r→final_distance) {
    r→slab.a = acalc2a(p[i][1]);
    r→slab.b = bcalc2b(p[i][2]);
    r→final_distance = y[i];
  }
}
```

This code is used in section 194.

200. ⟨ Put final values in result 200 ⟩ ≡

```
r→a = r→slab.a;
r→b = r→slab.b;
r→g = r→slab.g;
r→found = (r→tolerance ≤ r→final_distance);
```

This code is used in sections 194, 203, 205, 207, 209, 211, 214, 219, 225, and 230.

201. Since we allocated these puppies, we got to get rid of them.

⟨ Free simplex data structures 201 ⟩ ≡

```
free_dvector(x, 1, 2);
free_dvector(y, 1, 3);
free_dmatrix(p, 1, 3, 1, 2);
```

This code is used in sections 194, 214, 219, 225, and 230.

202. Fixed Absorption and Anisotropy. Typically, this routine is called when the absorption coefficient is known, the anisotropy is known, and the physical thickness of the sample is known. This routine calculates the varies the scattering coefficient until the measurements are matched.

This was written for Ted Moffitt to analyze some intralipid data. We wanted to know what the scattering coefficient of the Intralipid was and made total transmission measurements through a sample with a fixed physical thickness. We did not make reflection measurements because the light source diverged too much, and we could not make reflection measurements easily.

In retrospect, we could have made URU measurements by illuminating the wall of the integrating sphere. However, these diffuse type of measurements are very difficult to make accurately.

This is tricky only because the value in *slab.b* is used to hold the value of ba or $d \cdot \mu_a$ when the *Find_Bs_fn* is used.

⟨ Prototype for *U_Find_Bs* 202 ⟩ ≡

```
void U_Find_Bs(struct measure_type m, struct invert_type *r)
```

This code is used in sections 192 and 203.

203. ⟨ Definition for *U_Find_Bs* 203 ⟩ ≡

```
⟨ Prototype for U_Find_Bs 202 ⟩
{
  double ax, bx, cx, fa, fb, fc, bs;
  if (Debug(DEBUG_SEARCH)) {
    fprintf(stderr, "In␣U_Find_Bs");
    fprintf(stderr, "␣(mu=%6.4f)", r→slab.cos_angle);
    if (r→default_ba ≠ UNINITIALIZED) fprintf(stderr, "␣␣default_ba␣=␣%8.5f", r→default_ba);
    if (r→default_g ≠ UNINITIALIZED) fprintf(stderr, "␣␣default_g␣=␣%8.5f", r→default_g);
    fprintf(stderr, "\n");
  }
  r→slab.a = 0;
  r→slab.g = (r→default_g ≡ UNINITIALIZED) ? 0 : r→default_g;
  r→slab.b = (r→default_ba ≡ UNINITIALIZED) ? HUGE_VAL : r→default_ba;
  Set_Calc_State(m, *r);      /* store ba in RR.slab.b */
  ax = b2bcalc(0.1);      /* first try for bs */
  bx = b2bcalc(1.0);
  mnbrak(&ax, &bx, &cx, &fa, &fb, &fc, Find_Bs_fn);
  r→final_distance = brent(ax, bx, cx, Find_Bs_fn, r→tolerance, &bs);      /* recover true values */
  r→slab.a = bcalc2b(bs)/(bcalc2b(bs) + r→slab.b);
  r→slab.b = bcalc2b(bs) + r→slab.b;
  Set_Calc_State(m, *r);
  ⟨ Put final values in result 200 ⟩
}
```

This code is used in section 191.

204. Fixed Absorption and Scattering. Typically, this routine is called when the scattering coefficient is known, the anisotropy is known, and the physical thickness of the sample is known. This routine calculates the varies the absorption coefficient until the measurements are matched.

This is tricky only because the value in *slab.b* is used to hold the value of *bs* or $d \cdot \mu_s$ when the *Find_Ba_fn* is used.

⟨ Prototype for *U_Find_Ba* 204 ⟩ ≡

```
void U_Find_Ba(struct measure_type m, struct invert_type *r)
```

This code is used in sections 192 and 205.

205. ⟨ Definition for *U_Find_Ba* 205 ⟩ ≡

```
⟨ Prototype for U_Find_Ba 204 ⟩
{
  double ax, bx, cx, fa, fb, fc, ba;
  if (Debug(DEBUG_SEARCH)) {
    fprintf(stderr, "In U_Find_Bs");
    fprintf(stderr, " (mu=%6.4f)", r->slab.cos_angle);
    if (r->default_bs ≠ UNINITIALIZED) fprintf(stderr, "  default_bs = %8.5f", r->default_bs);
    if (r->default_g ≠ UNINITIALIZED) fprintf(stderr, "  default_g = %8.5f", r->default_g);
    fprintf(stderr, "\n");
  }
  r->slab.a = 0;
  r->slab.g = (r->default_g ≡ UNINITIALIZED) ? 0 : r->default_g;
  r->slab.b = (r->default_bs ≡ UNINITIALIZED) ? HUGE_VAL : r->default_bs;
  Set_Calc_State(m, *r);      /* store bs in RR.slab.b */
  ax = b2bcalc(0.1);      /* first try for ba */
  bx = b2bcalc(1.0);
  mnbrak(&ax, &bx, &cx, &fa, &fb, &fc, Find_Ba_fn);
  r->final_distance = brent(ax, bx, cx, Find_Ba_fn, r->tolerance, &ba);      /* recover true values */
  r->slab.a = (r->slab.b)/(bcalc2b(ba) + r->slab.b);
  r->slab.b = bcalc2b(ba) + r->slab.b;      /*actual value of b */
  Set_Calc_State(m, *r);
  ⟨ Put final values in result 200 ⟩
}
```

This code is used in section 191.

206. Fixed Optical Depth and Anisotropy. Typically, this routine is called when the optical thickness is assumed infinite. However, it may also be called when the optical thickness is assumed to be fixed at a particular value. Typically the only reasonable situation for this to occur is when the diffuse transmission is non-zero but the collimated transmission is zero. If this is the case then there is no information in the collimated transmission measurement and there is no sense even using it because the slab is not infinitely thick.

⟨ Prototype for *U_Find_A* 206 ⟩ ≡

```
void U_Find_A(struct measure_type m, struct invert_type *r)
```

This code is used in sections 192 and 207.

207. ⟨ Definition for *U_Find_A* 207 ⟩ ≡

```
⟨ Prototype for U_Find_A 206 ⟩
{
  double Rt, Tt, Rd, Rc, Td, Tc;
  if (Debug(DEBUG_SEARCH)) {
    fprintf(stderr, "In␣U_Find_A");
    fprintf(stderr, "␣(mu=%6.4f)", r→slab.cos_angle);
    if (r→default_b ≠ UNINITIALIZED) fprintf(stderr, "␣␣default_b␣=␣%8.5f", r→default_b);
    if (r→default_g ≠ UNINITIALIZED) fprintf(stderr, "␣␣default_g␣=␣%8.5f", r→default_g);
    fprintf(stderr, "\n");
  }
  Estimate_RT(m, ∗r, &Rt, &Tt, &Rd, &Rc, &Td, &Tc);
  r→slab.g = (r→default_g ≡ UNINITIALIZED) ? 0 : r→default_g;
  r→slab.b = (r→default_b ≡ UNINITIALIZED) ? HUGE_VAL : r→default_b;
  r→slab.a = 0.0;
  r→final_distance = 0.0;
  Set_Calc_State(m, ∗r);
  if (Rt > 0.99999) r→final_distance = Find_A_fn(a2acalc(1.0));
  else {
    double x, ax, bx, cx, fa, fb, fc;
    ax = a2acalc(0.3);
    bx = a2acalc(0.5);
    mnbrak(&ax, &bx, &cx, &fa, &fb, &fc, Find_A_fn);
    r→final_distance = brent(ax, bx, cx, Find_A_fn, r→tolerance, &x);
    r→slab.a = acalc2a(x);
  }
  ⟨ Put final values in result 200 ⟩
}
```

This code is used in section 191.

208. Fixed Optical Depth and Albedo.

⟨ Prototype for *U_Find_G* 208 ⟩ ≡

```
void U_Find_G(struct measure_type m, struct invert_type ∗r)
```

This code is used in sections 192 and 209.

209. ⟨ Definition for *U_Find_G* 209 ⟩ ≡

```
⟨ Prototype for U_Find_G 208 ⟩
{
  double Rt, Tt, Rd, Rc, Td, Tc;
  if (Debug(DEBUG_SEARCH)) {
    fprintf(stderr, "In␣U_Find_A");
    fprintf(stderr, "␣(mu=%6.4f)", r→slab.cos_angle);
    if (r→default_a ≠ UNINITIALIZED) fprintf(stderr, "␣␣default_a␣=␣%8.5f", r→default_a);
    if (r→default_b ≠ UNINITIALIZED) fprintf(stderr, "␣␣default_b␣=␣%8.5f", r→default_b);
    fprintf(stderr, "\n");
  }
  Estimate_RT(m, ∗r, &Rt, &Tt, &Rd, &Rc, &Td, &Tc);
  r→slab.a = (r→default_a ≡ UNINITIALIZED) ? 0.5 : r→default_a;
  r→slab.b = (r→default_b ≡ UNINITIALIZED) ? HUGE_VAL : r→default_b;
  r→slab.g = 0.0;
  r→final_distance = 0.0;
  Set_Calc_State(m, ∗r);
  if (Rd > 0.0) {
    double x, ax, bx, cx, fa, fb, fc;
    ax = g2gcalc(−0.99);
    bx = g2gcalc(0.99);
    mnbrak(&ax, &bx, &cx, &fa, &fb, &fc, Find_G_fn);
    r→final_distance = brent(ax, bx, cx, Find_G_fn, r→tolerance, &x);
    r→slab.g = gcalc2g(x);
    Set_Calc_State(m, ∗r);
  }
  ⟨ Put final values in result 200 ⟩
}
```

This code is used in section 191.

210. Fixed Anisotropy and Albedo. This routine can be called in three different situations: (1) the albedo is zero, (2) the albedo is one, or (3) the albedo is fixed at a default value. I calculate the individual reflections and transmissions to establish which of these cases we happen to have.

⟨ Prototype for *U_Find_B* 210 ⟩ ≡

```
void U_Find_B(struct measure_type m, struct invert_type ∗r)
```

This code is used in sections 192 and 211.

211. ⟨ Definition for *U_Find_B* 211 ⟩ ≡

```
⟨ Prototype for U_Find_B 210 ⟩
{
  double Rt, Tt, Rd, Rc, Td, Tc;
  if (Debug(DEBUG_SEARCH)) {
    fprintf(stderr, "In U_Find_B");
    fprintf(stderr, " (mu=%6.4f)", r->slab.cos_angle);
    if (r->default_a ≠ UNINITIALIZED) fprintf(stderr, "  default_a = %8.5f", r->default_a);
    if (r->default_g ≠ UNINITIALIZED) fprintf(stderr, "  default_g = %8.5f", r->default_g);
    fprintf(stderr, "\n");
  }
  Estimate_RT(m, *r, &Rt, &Tt, &Rd, &Rc, &Td, &Tc);
  r->slab.g = (r->default_g ≡ UNINITIALIZED) ? 0 : r->default_g;
  r->slab.a = (r->default_a ≡ UNINITIALIZED) ? 0 : r->default_a;
  r->slab.b = 0.5;
  r->final_distance = 0.0;
  Set_Calc_State(m, *r);
  ⟨ Iteratively solve for b 212 ⟩
  ⟨ Put final values in result 200 ⟩
  if (Debug(DEBUG_SEARCH)) {
    fprintf(stderr, "In U_Find_B final (a,b,g) = ");
    fprintf(stderr, "(%8.5f,%8.5f,%8.5f)\n", r->a, r->b, r->g);
  }
}
```

This code is used in section 191.

212. This could be improved tremendously. I just don't want to mess with it at the moment.

⟨ Iteratively solve for *b* 212 ⟩ ≡

```
{
  double x, ax, bx, cx, fa, fb, fc;
  ax = b2bcalc(0.1);
  bx = b2bcalc(10);
  mnbrak(&ax, &bx, &cx, &fa, &fb, &fc, Find_B_fn);
  r->final_distance = brent(ax, bx, cx, Find_B_fn, r->tolerance, &x);
  r->slab.b = bcalc2b(x);
  Set_Calc_State(m, *r);
}
```

This code is used in section 211.

213. Fixed Optical Depth.

We can get here a couple of different ways.

First there can be three real measurements, i.e., t_c is not zero, in this case we want to fix b based on the t_c measurement.

Second, we can get here if a default value for b has been set.

Otherwise, we really should not be here. Just set $b = 1$ and calculate away.

⟨ Prototype for *U_Find_AG* 213 ⟩ ≡

```
void U_Find_AG(struct measure_type m, struct invert_type *r)
```

This code is used in sections 192 and 214.

214. ⟨ Definition for *U_Find_AG* 214 ⟩ ≡

```
⟨ Prototype for U_Find_AG 213 ⟩
{
  if (Debug(DEBUG_SEARCH)) {
    fprintf(stderr, "In␣U_Find_AG");
    fprintf(stderr, "␣(mu=%6.4f)", r→slab.cos_angle);
    if (r→default_b ≠ UNINITIALIZED) fprintf(stderr, "␣␣default_b␣=␣%8.5f", r→default_b);
    fprintf(stderr, "\n");
  }
  ⟨ Allocate local simplex variables 195 ⟩
  if (m.num_measures ≡ 3) r→slab.b = What_Is_B(r→slab, m.m_u);
  else if (r→default_b ≡ UNINITIALIZED) r→slab.b = 1;
  else r→slab.b = r→default_b;
  Set_Calc_State(m, *r);
  ⟨ Get the initial a, b, and g 196 ⟩
  ⟨ Initialize the nodes of the a and g simplex 215 ⟩
  ⟨ Evaluate the a and g simplex at the nodes 216 ⟩
  amoeba(p, y, 2, r→tolerance, Find_AG_fn, &r→iterations);
  ⟨ Choose the best node of the a and g simplex 217 ⟩
  ⟨ Free simplex data structures 201 ⟩
  ⟨ Put final values in result 200 ⟩
}
```

This code is used in section 191.

215. ⟨ Initialize the nodes of the a and g simplex 215 ⟩ ≡

```
{
  int k, kk;
  p[1][1] = a2acalc(guess[0].a);
  p[1][2] = g2gcalc(guess[0].g);
  for (k = 1; k < 7; k++) {
    if (guess[0].a ≠ guess[k].a) break;
  }
  p[2][1] = a2acalc(guess[k].a);
  p[2][2] = g2gcalc(guess[k].g);
  for (kk = 1; kk < 7; kk++) {
    if (guess[0].g ≠ guess[kk].g ∧ guess[k].g ≠ guess[kk].g) break;
  }
  p[3][1] = a2acalc(guess[kk].a);
  p[3][2] = g2gcalc(guess[kk].g);
  if (Debug(DEBUG_BEST_GUESS)) {
    fprintf(stderr, "guess␣1");
    fprintf(stderr, "%10.5f␣", guess[0].a);
    fprintf(stderr, "%10.5f␣", guess[0].b);
    fprintf(stderr, "%10.5f␣", guess[0].g);
    fprintf(stderr, "%10.5f\n", guess[0].distance);
    fprintf(stderr, "guess␣2");
    fprintf(stderr, "%10.5f␣", guess[k].a);
    fprintf(stderr, "%10.5f␣", guess[k].b);
    fprintf(stderr, "%10.5f␣", guess[k].g);
    fprintf(stderr, "%10.5f\n", guess[k].distance);
    fprintf(stderr, "guess␣3");
    fprintf(stderr, "%10.5f␣", guess[kk].a);
    fprintf(stderr, "%10.5f␣", guess[kk].b);
    fprintf(stderr, "%10.5f␣", guess[kk].g);
    fprintf(stderr, "%10.5f\n", guess[kk].distance);
  }
}
```

This code is used in section 214.

216. ⟨ Evaluate the a and g simplex at the nodes 216 ⟩ ≡

```
for (i = 1; i ≤ 3; i++) {
  x[1] = p[i][1];
  x[2] = p[i][2];
  y[i] = Find_AG_fn(x);
}
```

This code is used in section 214.

217. Here we find the node of the simplex that gave the best result and save that one. At the same time we save the whole simplex for later use if needed.

```
⟨ Choose the best node of the a and g simplex 217 ⟩ ≡
  r→final_distance = 10;
  for (i = 1; i ≤ 3; i++) {
    if (y[i] < r→final_distance) {
      r→slab.a = acalc2a(p[i][1]);
      r→slab.g = gcalc2g(p[i][2]);
      r→final_distance = y[i];
    }
  }
```

This code is used in section 214.

218. Fixed Albedo. Here the optical depth and the anisotropy are varied (for a fixed albedo).

```
⟨ Prototype for U_Find_BG 218 ⟩ ≡
  void U_Find_BG(struct measure_type m, struct invert_type *r)
```

This code is used in sections 192 and 219.

219.

```
⟨ Definition for U_Find_BG 219 ⟩ ≡
  ⟨ Prototype for U_Find_BG 218 ⟩
  {
    if (Debug(DEBUG_SEARCH)) {
      fprintf(stderr, "In␣U_Find_BG");
      fprintf(stderr, "␣(mu=%6.4f)", r→slab.cos_angle);
      if (r→default_a ≠ UNINITIALIZED) fprintf(stderr, "␣␣default_a␣=␣%8.5f", r→default_a);
      fprintf(stderr, "\n");
    }
    ⟨ Allocate local simplex variables 195 ⟩
    r→slab.a = (r→default_a ≡ UNINITIALIZED) ? 0 : r→default_a;
    Set_Calc_State(m, *r);
    ⟨ Get the initial a, b, and g 196 ⟩
    ⟨ Initialize the nodes of the b and g simplex 221 ⟩
    ⟨ Evaluate the bg simplex at the nodes 222 ⟩
    amoeba(p, y, 2, r→tolerance, Find_BG_fn, &r→iterations);
    ⟨ Choose the best node of the b and g simplex 223 ⟩
    ⟨ Free simplex data structures 201 ⟩
    ⟨ Put final values in result 200 ⟩
  }
```

This code is used in section 191.

220. A very simple start for variation of b and g. This should work fine for the cases in which the absorption or scattering are fixed.

221. ⟨ Initialize the nodes of the b and g simplex 221 ⟩ ≡

```
{
  int k, kk;
  p[1][1] = b2bcalc(guess[0].b);
  p[1][2] = g2gcalc(guess[0].g);
  for (k = 1; k < 7; k++) {
    if (guess[0].b ≠ guess[k].b) break;
  }
  p[2][1] = b2bcalc(guess[k].b);
  p[2][2] = g2gcalc(guess[k].g);
  for (kk = 1; kk < 7; kk++) {
    if (guess[0].g ≠ guess[kk].g ∧ guess[k].g ≠ guess[kk].g) break;
  }
  p[3][1] = b2bcalc(guess[kk].b);
  p[3][2] = g2gcalc(guess[kk].g);
  if (Debug(DEBUG_BEST_GUESS)) {
    fprintf(stderr, "guess␣1");
    fprintf(stderr, "%10.5f␣", guess[0].a);
    fprintf(stderr, "%10.5f␣", guess[0].b);
    fprintf(stderr, "%10.5f␣", guess[0].g);
    fprintf(stderr, "%10.5f\n", guess[0].distance);
    fprintf(stderr, "guess␣2");
    fprintf(stderr, "%10.5f␣", guess[k].a);
    fprintf(stderr, "%10.5f␣", guess[k].b);
    fprintf(stderr, "%10.5f␣", guess[k].g);
    fprintf(stderr, "%10.5f\n", guess[k].distance);
    fprintf(stderr, "guess␣3");
    fprintf(stderr, "%10.5f␣", guess[kk].a);
    fprintf(stderr, "%10.5f␣", guess[kk].b);
    fprintf(stderr, "%10.5f␣", guess[kk].g);
    fprintf(stderr, "%10.5f\n", guess[kk].distance);
  }
}
```

This code is used in section 219.

222. ⟨ Evaluate the bg simplex at the nodes 222 ⟩ ≡

```
for (i = 1; i ≤ 3; i++) {
  x[1] = p[i][1];
  x[2] = p[i][2];
  y[i] = Find_BG_fn(x);
}
```

This code is used in section 219.

223. Here we find the node of the simplex that gave the best result and save that one. At the same time we save the whole simplex for later use if needed.

```
⟨ Choose the best node of the b and g simplex 223 ⟩ ≡
  r→final_distance = 10;
  for (i = 1; i ≤ 3; i++) {
    if (y[i] < r→final_distance) {
      r→slab.b = bcalc2b(p[i][1]);
      r→slab.g = gcalc2g(p[i][2]);
      r→final_distance = y[i];
    }
  }
```

This code is used in section 219.

224. Fixed Scattering. Here I assume that a constant b_s,

$$b_s = \mu_s d$$

where d is the physical thickness of the sample and μ_s is of course the absorption coefficient. This is just like *U_Find_BG* except that $b_a = \mu_a d$ is varied instead of b.

```
⟨ Prototype for U_Find_BaG 224 ⟩ ≡
  void U_Find_BaG(struct measure_type m, struct invert_type *r)
```

This code is used in sections 192 and 225.

225.
```
⟨ Definition for U_Find_BaG 225 ⟩ ≡
  ⟨ Prototype for U_Find_BaG 224 ⟩
  {
    ⟨ Allocate local simplex variables 195 ⟩
    Set_Calc_State(m, *r);
    ⟨ Get the initial a, b, and g 196 ⟩
    ⟨ Initialize the nodes of the ba and g simplex 226 ⟩
    ⟨ Evaluate the BaG simplex at the nodes 227 ⟩
    amoeba(p, y, 2, r→tolerance, Find_BaG_fn, &r→iterations);
    ⟨ Choose the best node of the ba and g simplex 228 ⟩
    ⟨ Free simplex data structures 201 ⟩
    ⟨ Put final values in result 200 ⟩
  }
```

This code is used in section 191.

226. ⟨ Initialize the nodes of the *ba* and *g* simplex 226 ⟩ ≡

```
if (guess[0].b > r→default_bs) {
  p[1][1] = b2bcalc(guess[0].b − r→default_bs);
  p[2][1] = b2bcalc(2 ∗ (guess[0].b − r→default_bs));
  p[3][1] = p[1][1];
}
else {
  p[1][1] = b2bcalc(0.0001);
  p[2][1] = b2bcalc(0.001);
  p[3][1] = p[1][1];
}
p[1][2] = g2gcalc(guess[0].g);
p[2][2] = p[1][2];
p[3][2] = g2gcalc(0.9 ∗ guess[0].g + 0.05);
```

This code is used in section 225.

227. ⟨ Evaluate the *BaG* simplex at the nodes 227 ⟩ ≡

```
for (i = 1; i ≤ 3; i++) {
  x[1] = p[i][1];
  x[2] = p[i][2];
  y[i] = Find_BaG_fn(x);
}
```

This code is used in section 225.

228. Here we find the node of the simplex that gave the best result and save that one. At the same time we save the whole simplex for later use if needed.

⟨ Choose the best node of the *ba* and *g* simplex 228 ⟩ ≡

```
r→final_distance = 10;
for (i = 1; i ≤ 3; i++) {
  if (y[i] < r→final_distance) {
    r→slab.b = bcalc2b(p[i][1]) + r→default_bs;
    r→slab.a = r→default_bs/r→slab.b;
    r→slab.g = gcalc2g(p[i][2]);
    r→final_distance = y[i];
  }
}
```

This code is used in section 225.

229. Fixed Absorption. Here I assume that a constant b_a,

$$b_a = \mu_a d$$

where d is the physical thickness of the sample and μ_a is of course the absorption coefficient. This is just like *U_Find_BG* except that $b_s = \mu_s d$ is varied instead of b.

⟨ Prototype for *U_Find_BsG* 229 ⟩ ≡

```
void U_Find_BsG(struct measure_type m, struct invert_type ∗r)
```

This code is used in sections 192 and 230.

230. ⟨ Definition for *U_Find_BsG* 230 ⟩ ≡

```
⟨ Prototype for U_Find_BsG 229 ⟩
{
  if (Debug(DEBUG_SEARCH)) {
    fprintf(stderr, "In␣U_Find_BsG");
    fprintf(stderr, "␣(mu=%6.4f)", r→slab.cos_angle);
    if (r→default_ba ≠ UNINITIALIZED) fprintf(stderr, "␣␣default_ba␣=␣%8.5f", r→default_ba);
    fprintf(stderr, "\n");
  }
  ⟨ Allocate local simplex variables 195 ⟩
  Set_Calc_State(m, *r);
  ⟨ Get the initial a, b, and g 196 ⟩
  ⟨ Initialize the nodes of the bs and g simplex 231 ⟩
  ⟨ Evaluate the BsG simplex at the nodes 232 ⟩
  amoeba(p, y, 2, r→tolerance, Find_BsG_fn, &r→iterations);
  ⟨ Choose the best node of the bs and g simplex 233 ⟩
  ⟨ Free simplex data structures 201 ⟩
  ⟨ Put final values in result 200 ⟩
}
```

This code is used in section 191.

231. ⟨ Initialize the nodes of the *bs* and *g* simplex 231 ⟩ ≡

```
p[1][1] = b2bcalc(guess[0].b − r→default_ba);
p[1][2] = g2gcalc(guess[0].g);
p[2][1] = b2bcalc(2 * guess[0].b − 2 * r→default_ba);
p[2][2] = p[1][2];
p[3][1] = p[1][1];
p[3][2] = g2gcalc(0.9 * guess[0].g + 0.05);
```

This code is used in section 230.

232. ⟨ Evaluate the *BsG* simplex at the nodes 232 ⟩ ≡

```
for (i = 1; i ≤ 3; i++) {
  x[1] = p[i][1];
  x[2] = p[i][2];
  y[i] = Find_BsG_fn(x);
}
```

This code is used in section 230.

233. ⟨ Choose the best node of the *bs* and *g* simplex 233 ⟩ ≡

```
r→final_distance = 10;
for (i = 1; i ≤ 3; i++) {
  if (y[i] < r→final_distance) {
    r→slab.b = bcalc2b(p[i][1]) + r→default_ba;
    r→slab.a = 1 − r→default_ba/r→slab.b;
    r→slab.g = gcalc2g(p[i][2]);
    r→final_distance = y[i];
  }
}
```

This code is used in section 230.

234. IAD Utilities.

March 1995. Reincluded *quick_guess* code.

⟨ `iad_util.c` 234 ⟩ ≡

```
#include <math.h>
#include <float.h>
#include <stdio.h>
#include "nr_util.h"
#include "ad_globl.h"
#include "ad_frsnl.h"
#include "ad_bound.h"
#include "iad_type.h"
#include "iad_calc.h"
#include "iad_pub.h"
#include "iad_util.h"
  unsigned long g_util_debugging = 0;
```

⟨ Preprocessor definitions ⟩

⟨ Definition for *What_Is_B* 237 ⟩
⟨ Definition for *Estimate_RT* 243 ⟩
⟨ Definition for *a2acalc* 249 ⟩
⟨ Definition for *acalc2a* 251 ⟩
⟨ Definition for *g2gcalc* 253 ⟩
⟨ Definition for *gcalc2g* 255 ⟩
⟨ Definition for *b2bcalc* 257 ⟩
⟨ Definition for *bcalc2b* 259 ⟩
⟨ Definition for *twoprime* 261 ⟩
⟨ Definition for *twounprime* 263 ⟩
⟨ Definition for *abgg2ab* 265 ⟩
⟨ Definition for *abgb2ag* 267 ⟩
⟨ Definition for *quick_guess* 274 ⟩
⟨ Definition for *Set_Debugging* 287 ⟩
⟨ Definition for *Debug* 289 ⟩
⟨ Definition for *Print_Invert_Type* 291 ⟩
⟨ Definition for *Print_Measure_Type* 293 ⟩

235. ⟨ iad_util.h 235 ⟩ ≡
⟨ Prototype for *What_Is_B* 236 ⟩;
⟨ Prototype for *Estimate_RT* 242 ⟩;
⟨ Prototype for *a2acalc* 248 ⟩;
⟨ Prototype for *acalc2a* 250 ⟩;
⟨ Prototype for *g2gcalc* 252 ⟩;
⟨ Prototype for *gcalc2g* 254 ⟩;
⟨ Prototype for *b2bcalc* 256 ⟩;
⟨ Prototype for *bcalc2b* 258 ⟩;
⟨ Prototype for *twoprime* 260 ⟩;
⟨ Prototype for *twounprime* 262 ⟩;
⟨ Prototype for *abgg2ab* 264 ⟩;
⟨ Prototype for *abgb2ag* 266 ⟩;
⟨ Prototype for *quick_guess* 273 ⟩;
⟨ Prototype for *Set_Debugging* 286 ⟩;
⟨ Prototype for *Debug* 288 ⟩;
⟨ Prototype for *Print_Invert_Type* 290 ⟩;
⟨ Prototype for *Print_Measure_Type* 292 ⟩;

236. Finding optical thickness.
This routine figures out what the optical thickness of a slab based on the index of refraction of the slab and the amount of collimated light that gets through it.

It should be pointed out right here in the front that this routine does not work for diffuse irradiance, but then the whole concept of estimating the optical depth for diffuse irradiance is bogus anyway.

In version 1.3 changed all error output to *stderr*. Version 1.4 included cases involving absorption in the boundaries.

#define BIG_A_VALUE 999999.0
#define SMALL_A_VALUE 0.000001

⟨ Prototype for *What_Is_B* 236 ⟩ ≡
double *What_Is_B*(**struct** *AD_slab_type* *slab*, **double** *Tc*)
This code is used in sections 235 and 237.

237. ⟨ Definition for *What_Is_B* 237 ⟩ ≡
⟨ Prototype for *What_Is_B* 236 ⟩
{
double *r1*, *r2*, *t1*, *t2*;
⟨ Calculate specular reflection and transmission 238 ⟩
⟨ Check for bad values of *Tc* 239 ⟩
⟨ Solve if multiple internal reflections are not present 240 ⟩
⟨ Find thickness when multiple internal reflections are present 241 ⟩
}
This code is used in section 234.

238. The first thing to do is to find the specular reflection for light interacting with the top and bottom air-glass-sample interfaces. I make a simple check to ensure that the the indices are different before calculating the bottom reflection. Most of the time the $r1 \equiv r2$, but there are always those annoying special cases.

⟨ Calculate specular reflection and transmission 238 ⟩ ≡
Absorbing_Glass_RT(1.0, *slab*.*n_top_slide*, *slab*.*n_slab*, *slab*.*cos_angle*, *slab*.*b_top_slide*, &*r1*, &*t1*);
Absorbing_Glass_RT(*slab*.*n_slab*, *slab*.*n_bottom_slide*, 1.0, *slab*.*cos_angle*, *slab*.*b_bottom_slide*, &*r2*, &*t2*);
This code is used in section 237.

239. Bad values for the unscattered transmission are those that are non-positive, those greater than one, and those greater than are possible in a non-absorbing medium, i.e.,

$$T_c > \frac{t_1 t_2}{1 - r_1 r_2}$$

Since this routine has no way to report errors, I just set the optical thickness to the natural values in these cases.

⟨ Check for bad values of Tc 239 ⟩ ≡
```
if (Tc ≤ 0) return (HUGE_VAL);
if (Tc ≥ t1 * t2/(1 − r1 * r2)) return (0.001);
```
This code is used in section 237.

240. If either $r1$ or $r2 \equiv 0$ then things are very simple because the sample does not sustain multiple internal reflections and the unscattered transmission is

$$T_c = t_1 t_2 \exp(-b/\nu)$$

where b is the optical thickness and ν is $slab.cos\_angle$. Clearly,

$$b = -\nu \ln\left(\frac{T_c}{t_1 t_2}\right)$$

⟨ Solve if multiple internal reflections are not present 240 ⟩ ≡
```
if (r1 ≡ 0 ∨ r2 ≡ 0) return (−slab.cos_angle * log(Tc/t1/t2));
```
This code is used in section 237.

241. Well I kept putting it off, but now comes the time to solve the following equation for b

$$T_c = \frac{t_1 t_2 \exp(-b)}{1 - r_1 r_2 \exp(-2b)}$$

We note immediately that this is a quadratic equation in $x = \exp(-b)$.

$$r_1 r_2 T_c x^2 + t_1 t_2 x - T_c = 0$$

Sufficient tests have been made above to ensure that none of the coefficients are exactly zero. However, it is clear that the leading quadratic term has a much smaller coefficient than the other two. Since r_1 and r_2 are typically about four percent the product is roughly 10^{-3}. The collimated transmission can be very small and this makes things even worse. A further complication is that we need to choose the only positive root.

Now the roots of $ax^2 + bx + c = 0$ can be found using the standard quadratic formula,

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

This is very bad for small values of a. Instead I use

$$q = -\frac{1}{2}\left[b + \mathrm{sgn}(b)\sqrt{b^2 - 4ac}\right]$$

with the two roots

$$x = \frac{q}{a} \qquad \text{and} \qquad x = \frac{c}{q}$$

Substituting our coefficients

$$q = -\frac{1}{2}\left[t_1 t_2 + \sqrt{t_1^2 t_2^2 + 4 r_1 r_2 T_c^2}\right]$$

With some algebra, this can be shown to be

$$q = -t_1 t_2 \left[1 + \frac{r_1 r_2 T_c^2}{t_1^2 t_2^2} + \cdots\right]$$

The only positive root is $x = -T_c/q$. Therefore

$$x = \frac{2T_c}{t_1 t_2 + \sqrt{t_1^2 t_2^2 + 4 r_1 r_2 T_c^2}}$$

(Not very pretty, but straightforward enough.)

⟨ Find thickness when multiple internal reflections are present 241 ⟩ ≡

```
{
  double B;
  B = t1 * t2;
  return (-slab.cos_angle * log(2 * Tc/(B + sqrt(B * B + 4 * Tc * Tc * r1 * r2))));
}
```

This code is used in section 237.

242. Estimating R and T.

In several places, it is useful to know an *estimate* for the values of the reflection and transmission of the sample based on the measurements. This routine provides such an estimate, but it currently ignores anything corrections that might be made for the integrating spheres.

Good values are only really obtainable when $num\_measures \equiv 3$, otherwise we need to make pretty strong assumptions about the reflection and transmission values. If $num\_measures < 3$, then we will assume that no collimated light makes it all the way through the sample. The specular reflection is then just that for a semi-infinite sample and $Tc = 0$. If $num\_measures \equiv 1$, then Td is also set to zero.

rt	total reflection
rc	primary or specular reflection
rd	diffuse or scattered reflection
tt	total transmission
tp	primary or unscattered transmission
td	diffuse or scattered transmission

⟨ Prototype for *Estimate_RT* 242 ⟩ ≡

void *Estimate_RT*(**struct measure_type** *m*, **struct invert_type** *r*, **double** ∗*rt*, **double** ∗*tt*, **double** ∗*rd*, **double** ∗*rc*, **double** ∗*td*, **double** ∗*tc*)

This code is used in sections 235 and 243.

243. ⟨ Definition for *Estimate_RT* 243 ⟩ ≡

```
⟨ Prototype for Estimate_RT 242 ⟩
{
  ⟨ Calculate the unscattered transmission and reflection 244 ⟩
  ⟨ Estimate the backscattered reflection 245 ⟩
  ⟨ Estimate the scattered transmission 246 ⟩
}
```

This code is used in section 234.

244. If there are three measurements then the specular reflection can be calculated pretty well. If there are fewer then the unscattered transmission is assumed to be zero. This is not necessarily the case, but after all, this routine only makes estimates of the various reflection and transmission quantities.

If there are three measurements, the optical thickness of the sample is required. Of course if there are three measurements then the illumination must be collimated and we can call *What_Is_B* to find out the optical thickness. We pass this value to a routine in the `fresnel.h` unit and sit back and wait.

All the above is true if sphere corrections are not needed. Now, we just fob this off on another function.

⟨ Calculate the unscattered transmission and reflection 244 ⟩ ≡

```
Calculate_Minimum_MR(m, r, rc, tc);
```

This code is used in section 243.

245. Finding the diffuse reflection is now just a matter of checking whether V1% contains the specular reflection from the sample or not and then just adding or subtracting the specular reflection as appropriate.

```
⟨ Estimate the backscattered reflection 245 ⟩ ≡
  if (m.sphere_with_rc) {
    *rt = m.m_r;
    *rd = *rt - *rc;
    if (*rd < 0) {
      *rd = 0;
      *rc = *rt;
    }
  }
  else {
    *rd = m.m_r;
    *rt = *rd + *rc;
  }
```

This code is used in section 243.

246. The transmission values follow in much the same way as the diffuse reflection values — just subtract the specular transmission from the total transmission.

```
⟨ Estimate the scattered transmission 246 ⟩ ≡
  if (m.num_measures ≡ 1) {
    *tt = 0.0;
    *td = 0.0;
  }
  else if (m.sphere_with_tc) {
    *tt = m.m_t;
    *td = *tt - *tc;
    if (*td < 0) {
      *tc = *tt;
      *td = 0;
    }
  }
  else {
    *td = m.m_t;
    *tt = *td + *tc;
  }
```

This code is used in section 243.

247. Transforming properties. Routines to convert optical properties to calculation space and back.

248. *a2acalc* is used for the albedo transformations according to

$$a_{calc} = \frac{2a - 1}{a(1 - a)}$$

Care is taken to avoid division by zero. Why was this function chosen? Well mostly because it maps the region between $[0, 1] \to (-\infty, +\infty)$.

```
⟨ Prototype for a2acalc 248 ⟩ ≡
  double a2acalc(double a)
```

This code is used in sections 235 and 249.

249. ⟨ Definition for *a2acalc* 249 ⟩ ≡

```
⟨ Prototype for a2acalc 248 ⟩
{
  if (a ≤ 0) return −BIG_A_VALUE;
  if (a ≥ 1) return BIG_A_VALUE;
  return ((2 ∗ a − 1)/a/(1 − a));
}
```

This code is used in section 234.

250. *acalc2a* is used for the albedo transformations Now when we solve

$$a_calc = \frac{2a-1}{a(1-a)}$$

we obtain the quadratic equation

$$a_{calc}a^2 + (2 - a_{calc})a - 1 = 0$$

The only root of this equation between zero and one is

$$a = \frac{-2 + a_{calc} + \sqrt{a_{calc}^2 + 4}}{2a_{calc}}$$

I suppose that I should spend the time to recast this using the more appropriate numerical solutions of the quadratic equation, but this worked and I will leave it as it is for now.

⟨ Prototype for *acalc2a* 250 ⟩ ≡

```
double acalc2a(double acalc)
```

This code is used in sections 235 and 251.

251. ⟨ Definition for *acalc2a* 251 ⟩ ≡

```
⟨ Prototype for acalc2a 250 ⟩
{
  if (acalc ≡ BIG_A_VALUE) return 1.0;
  else if (acalc ≡ −BIG_A_VALUE) return 0.0;
  else if (fabs(acalc) < SMALL_A_VALUE) return 0.5;
  else return ((−2 + acalc + sqrt(acalc ∗ acalc + 4))/(2 ∗ acalc));
}
```

This code is used in section 234.

252. *g2gcalc* is used for the anisotropy transformations according to

$$g_{calc} = \frac{g}{1 + |g|}$$

which maps $(-1, 1) \to (-\infty, +\infty)$.

⟨ Prototype for *g2gcalc* 252 ⟩ ≡

```
double g2gcalc(double g)
```

This code is used in sections 235 and 253.

253. ⟨ Definition for *g2gcalc* 253 ⟩ ≡
```
⟨ Prototype for g2gcalc 252 ⟩
{
  if (g ≤ −1) return (−HUGE_VAL);
  if (g ≥ 1) return (HUGE_VAL);
  return (g/(1 − fabs(g)));
}
```
This code is used in section 234.

254. *gcalc2g* is used for the anisotropy transformations it is the inverse of *g2gcalc*. The relation is

$$g = \frac{g_{calc}}{1 + |g_{calc}|}$$

⟨ Prototype for *gcalc2g* 254 ⟩ ≡
```
double gcalc2g(double gcalc)
```
This code is used in sections 235 and 255.

255. ⟨ Definition for *gcalc2g* 255 ⟩ ≡
```
⟨ Prototype for gcalc2g 254 ⟩
{
  if (gcalc ≡ −HUGE_VAL) return −1.0;
  if (gcalc ≡ HUGE_VAL) return 1.0;
  return (gcalc/(1 + fabs(gcalc)));
}
```
This code is used in section 234.

256. *b2bcalc* is used for the optical depth transformations it is the inverse of *bcalc2b*. The relation is

$$b_{calc} = \ln(b)$$

The only caveats are to ensure that I don't take the logarithm of something big or non-positive.

⟨ Prototype for *b2bcalc* 256 ⟩ ≡
```
double b2bcalc(double b)
```
This code is used in sections 235 and 257.

257. ⟨ Definition for *b2bcalc* 257 ⟩ ≡
```
⟨ Prototype for b2bcalc 256 ⟩
{
  if (b ≡ HUGE_VAL) return HUGE_VAL;
  if (b ≤ 0) return 0.0;
  return (log(b));
}
```
This code is used in section 234.

258. *bcalc2b* is used for the anisotropy transformations it is the inverse of *b2bcalc*. The relation is

$$b = \exp(b_{calc})$$

The only tricky part is to ensure that I don't exponentiate something big and get an overflow error. In ANSI C the maximum value for x such that 10^x is in the range of representable finite floating point numbers (for doubles) is given by `DBL_MAX_10_EXP`. Thus if we want to know if

$$e^{b_{calc}} > 10^x$$

or

$$b_{calc} > x \ln(10) \approx 2.3x$$

and this is the criterion that I use.

⟨ Prototype for *bcalc2b* 258 ⟩ ≡
double *bcalc2b*(**double** *bcalc*)
This code is used in sections 235 and 259.

259. ⟨ Definition for *bcalc2b* 259 ⟩ ≡
⟨ Prototype for *bcalc2b* 258 ⟩
{
if (*bcalc* ≡ `HUGE_VAL`) **return** `HUGE_VAL`;
if (*bcalc* > 2.3 ∗ `DBL_MAX_10_EXP`) **return** `HUGE_VAL`;
return (*exp*(*bcalc*));
}
This code is used in section 234.

260. *twoprime* converts the true albedo a, optical depth b to the reduced albedo ap and reduced optical depth bp that correspond to $g = 0$.

⟨ Prototype for *twoprime* 260 ⟩ ≡
void *twoprime*(**double** *a*, **double** *b*, **double** *g*, **double** ∗*ap*, **double** ∗*bp*)
This code is used in sections 235 and 261.

261. ⟨ Definition for *twoprime* 261 ⟩ ≡
⟨ Prototype for *twoprime* 260 ⟩
{
if (*a* ≡ 1 ∧ *g* ≡ 1) ∗*ap* = 0.0;
else ∗*ap* = (1 − *g*) ∗ *a*/(1 − *a* ∗ *g*);
if (*b* ≡ `HUGE_VAL`) ∗*bp* = `HUGE_VAL`;
else ∗*bp* = (1 − *a* ∗ *g*) ∗ *b*;
}
This code is used in section 234.

262. *twounprime* converts the reduced albedo ap and reduced optical depth bp (for $g = 0$) to the true albedo a and optical depth b for an anisotropy g.

⟨ Prototype for *twounprime* 262 ⟩ ≡
void *twounprime*(**double** *ap*, **double** *bp*, **double** *g*, **double** ∗*a*, **double** ∗*b*)
This code is used in sections 235 and 263.

263. ⟨ Definition for *twounprime* 263 ⟩ ≡

```
⟨ Prototype for twounprime 262 ⟩
{
  *a = ap/(1 - g + ap * g);
  if (bp ≡ HUGE_VAL) *b = HUGE_VAL;
  else *b = (1 + ap * g/(1 - g)) * bp;
}
```

This code is used in section 234.

264. *abgg2ab* assume a, b, g, and *g1* are given this does the similarity translation that you would expect it should by converting it to the reduced optical properties and then transforming back using the new value of g

⟨ Prototype for *abgg2ab* 264 ⟩ ≡

```
void abgg2ab(double a1, double b1, double g1, double g2, double *a2, double *b2)
```

This code is used in sections 235 and 265.

265. ⟨ Definition for *abgg2ab* 265 ⟩ ≡

```
⟨ Prototype for abgg2ab 264 ⟩
{
  double a, b;
  twoprime(a1, b1, g1, &a, &b);
  twounprime(a, b, g2, a2, b2);
}
```

This code is used in section 234.

266. *abgb2ag* translates reduced optical properties to unreduced values assuming that the new optical thickness is given i.e., *a1* and *b1* are a' and b' for $g = 0$. This routine then finds the appropriate anisotropy and albedo which correspond to an optical thickness *b2*.

If both *b1* and *b2* are zero then just assume $g = 0$ for the unreduced values.

⟨ Prototype for *abgb2ag* 266 ⟩ ≡

```
void abgb2ag(double a1, double b1, double b2, double *a2, double *g2)
```

This code is used in sections 235 and 267.

267. ⟨ Definition for *abgb2ag* 267 ⟩ ≡

```
⟨ Prototype for abgb2ag 266 ⟩
{
  if (b1 ≡ 0 ∨ b2 ≡ 0) {
    *a2 = a1;
    *g2 = 0;
  }
  if (b2 < b1) b2 = b1;
  if (a1 ≡ 0) *a2 = 0.0;
  else {
    if (a1 ≡ 1) *a2 = 1.0;
    else {
      if (b1 ≡ 0 ∨ b2 ≡ HUGE_VAL) *a2 = a1;
      else *a2 = 1 + b1/b2 * (a1 − 1);
    }
  }
  if (*a2 ≡ 0 ∨ b2 ≡ 0 ∨ b2 ≡ HUGE_VAL) *g2 = 0.5;
  else *g2 = (1 − b1/b2)/(*a2);
}
```

This code is used in section 234.

268. Guessing an inverse.
This routine is not used anymore.

⟨ Prototype for *slow_guess* 268 ⟩ ≡

```
void slow_guess(struct measure_type m, struct invert_type *r, double *a, double *b, double *g)
```

This code is used in section 269.

269. ⟨ Definition for *slow_guess* 269 ⟩ ≡

```
⟨ Prototype for slow_guess 268 ⟩
{
  double fmin = 10.0;
  double fval;
  double *x;
  x = dvector(1, 2);
  switch (r→search) {
  case FIND_A: ⟨ Slow guess for a alone 270 ⟩
    break;
  case FIND_B: ⟨ Slow guess for b alone 271 ⟩
    break;
  case FIND_AB: case FIND_AG: ⟨ Slow guess for a and b or a and g 272 ⟩
    break;
  }
  *a = r→slab.a;
  *b = r→slab.b;
  *g = r→slab.g;
  free_dvector(x, 1, 2);
}
```

270. ⟨ Slow guess for a alone 270 ⟩ ≡

```
r→slab.b = HUGE_VAL;
r→slab.g = r→default_g;
Set_Calc_State(m, *r);
for (r→slab.a = 0.0; r→slab.a ≤ 1.0; r→slab.a += 0.1) {
  fval = Find_A_fn(a2acalc(r→slab.a));
  if (fval < fmin) {
    r→a = r→slab.a;
    fmin = fval;
  }
}
r→slab.a = r→a;
```

This code is used in section 269.

271. Presumably the only time that this will need to be called is when the albedo is fixed or is one. For now, I'll just assume that it is one.

⟨ Slow guess for b alone 271 ⟩ ≡

```
r→slab.a = 1;
r→slab.g = r→default_g;
Set_Calc_State(m, *r);
for (r→slab.b = 1/32.0; r→slab.b ≤ 32; r→slab.b *= 2) {
  fval = Find_B_fn(b2bcalc(r→slab.b));
  if (fval < fmin) {
    r→b = r→slab.b;
    fmin = fval;
  }
}
r→slab.b = r→b;
```

This code is used in section 269.

272. ⟨ Slow guess for a and b or a and g 272 ⟩ ≡

```
{
  double min_a, min_b, min_g;
  if (¬Valid_Grid(m, r→search)) Fill_Grid(m, *r);
  Near_Grid_Points(m.m_r, m.m_t, r→search, &min_a, &min_b, &min_g);
  r→slab.a = min_a;
  r→slab.b = min_b;
  r→slab.g = min_g;
}
```

This code is used in section 269.

273. ⟨ Prototype for *quick_guess* 273 ⟩ ≡

```
void quick_guess(struct measure_type m, struct invert_type r, double *a, double *b, double *g)
```

This code is used in sections 235 and 274.

274. ⟨ Definition for *quick_guess* 274 ⟩ ≡

```
⟨ Prototype for quick_guess 273 ⟩
{
  double UR1, UT1, rd, td, tc, rc, bprime, aprime, alpha, beta, logr;
  Estimate_RT(m, r, &UR1, &UT1, &rd, &rc, &td, &tc);
  ⟨ Estimate aprime 275 ⟩
  switch (m.num_measures) {
  case 1: ⟨ Guess when only reflection is known 277 ⟩
    break;
  case 2: ⟨ Guess when reflection and transmission are known 278 ⟩
    break;
  case 3: ⟨ Guess when all three measurements are known 279 ⟩
    break;
  }
  ⟨ Clean up guesses 284 ⟩
}
```

This code is used in section 234.

275. ⟨ Estimate *aprime* 275 ⟩ ≡

```
if (UT1 ≡ 1) aprime = 1.0;
else if (rd/(1 − UT1) ≥ 0.1) {
  double tmp = (1 − rd − UT1)/(1 − UT1);
  aprime = 1 − 4.0/9.0 ∗ tmp ∗ tmp;
}
else if (rd < 0.05 ∧ UT1 < 0.4) aprime = 1 − (1 − 10 ∗ rd) ∗ (1 − 10 ∗ rd);
else if (rd < 0.1 ∧ UT1 < 0.4) aprime = 0.5 + (rd − 0.05) ∗ 4;
else {
  double tmp = (1 − 4 ∗ rd − UT1)/(1 − UT1);
  aprime = 1 − tmp ∗ tmp;
}
```

This code is used in section 274.

276. ⟨ Estimate *bprime* 276 ⟩ ≡

```
if (rd < 0.01) {
  bprime = What_Is_B(r.slab, UT1);
  fprintf(stderr, "low␣rd<0.01!␣ut1=%f␣aprime=%f␣bprime=%f\n", UT1, aprime, bprime);
}
else if (UT1 ≤ 0) bprime = HUGE_VAL;
else if (UT1 > 0.1) bprime = 2 ∗ exp(5 ∗ (rd − UT1) ∗ log(2.0));
else {
  alpha = 1/log(0.05/1.0);
  beta = log(1.0)/log(0.05/1.0);
  logr = log(UR1);
  bprime = log(UT1) − beta ∗ log(0.05) + beta ∗ logr;
  bprime /= alpha ∗ log(0.05) − alpha ∗ logr − 1;
}
```

This code is used in sections 278, 282, and 283.

277.

⟨ Guess when only reflection is known 277 ⟩ ≡
```
*g = r.default_g;
*a = aprime/(1 - *g + aprime * (*g));
*b = HUGE_VAL;
```
This code is used in section 274.

278. ⟨ Guess when reflection and transmission are known 278 ⟩ ≡
```
⟨ Estimate bprime 276 ⟩
*g = r.default_g;
*a = aprime/(1 - *g + aprime * *g);
*b = bprime/(1 - *a * *g);
```
This code is used in section 274.

279. ⟨ Guess when all three measurements are known 279 ⟩ ≡
```
switch (r.search) {
case FIND_A: ⟨ Guess when finding albedo 280 ⟩
  break;
case FIND_B: ⟨ Guess when finding optical depth 281 ⟩
  break;
case FIND_AB: ⟨ Guess when finding the albedo and optical depth 282 ⟩
  break;
case FIND_AG: ⟨ Guess when finding anisotropy and albedo 283 ⟩
  break;
}
```
This code is used in section 274.

280.

⟨ Guess when finding albedo 280 ⟩ ≡
```
*g = r.default_g;
*a = aprime/(1 - *g + aprime * *g);
*b = What_Is_B(r.slab, m.m_u);
```
This code is used in section 279.

281.

⟨ Guess when finding optical depth 281 ⟩ ≡
```
*g = r.default_g;
*a = 0.0;
*b = What_Is_B(r.slab, m.m_u);
```
This code is used in section 279.

282.

⟨ Guess when finding the albedo and optical depth 282 ⟩ ≡
```
*g = r.default_g;
if (*g ≡ 1) *a = 0.0;
else *a = aprime/(1 - *g + aprime * *g);
⟨ Estimate bprime 276 ⟩
if (bprime ≡ HUGE_VAL ∨ *a * *g ≡ 1) *b = HUGE_VAL;
else *b = bprime/(1 - *a * *g);
```
This code is used in section 279.

283.

```
⟨ Guess when finding anisotropy and albedo 283 ⟩ ≡
  *b = What_Is_B(r.slab, m.m_u);
  if (*b ≡ HUGE_VAL ∨ *b ≡ 0) {
    *a = aprime;
    *g = r.default_g;
  }
  else {
    ⟨ Estimate bprime 276 ⟩
    *a = 1 + bprime * (aprime − 1)/(*b);
    if (*a < 0.1) *g = 0.0;
    else *g = (1 − bprime/(*b))/(*a);
  }
```

This code is used in section 279.

284.

```
⟨ Clean up guesses 284 ⟩ ≡
  if (*a < 0) *a = 0.0;
  if (*g < 0) *g = 0.0;
  else if (*g ≥ 1) *g = 0.5;
```

This code is used in section 274.

285. Some debugging stuff.

286.

```
⟨ Prototype for Set_Debugging 286 ⟩ ≡
  void Set_Debugging(unsigned long debug_level)
```

This code is used in sections 235 and 287.

287.

```
⟨ Definition for Set_Debugging 287 ⟩ ≡
  ⟨ Prototype for Set_Debugging 286 ⟩
  {
    g_util_debugging = debug_level;
  }
```

This code is used in section 234.

288.

```
⟨ Prototype for Debug 288 ⟩ ≡
  int Debug(unsigned long mask)
```

This code is used in sections 235 and 289.

289.

```
⟨ Definition for Debug 289 ⟩ ≡
  ⟨ Prototype for Debug 288 ⟩
  {
    if (g_util_debugging & mask) return 1;
    else return 0;
  }
```

This code is used in section 234.

290.

⟨ Prototype for *Print_Invert_Type* 290 ⟩ ≡
void *Print_Invert_Type*(**struct invert_type** *r*)

This code is used in sections 235 and 291.

291.

⟨ Definition for *Print_Invert_Type* 291 ⟩ ≡
⟨ Prototype for *Print_Invert_Type* 290 ⟩
{
fprintf(*stderr*, "\n");
fprintf(*stderr*, "default␣␣a=%10.5f␣␣␣b=%10.5f␣␣␣␣g=%10.5f\n", *r.default_a*, *r.default_b*, *r.default_g*);
fprintf(*stderr*, "slab␣␣␣␣␣a=%10.5f␣␣␣b=%10.5f␣␣␣␣g=%10.5f\n", *r.slab.a*, *r.slab.b*, *r.slab.g*);
fprintf(*stderr*, "n␣␣␣␣␣␣␣top=%10.5f␣mid=%10.5f␣␣bot=%10.5f\n", *r.slab.n_top_slide*, *r.slab.n_slab*,
r.slab.n_bottom_slide);
fprintf(*stderr*, "thick␣␣top=%10.5f␣cos=%10.5f␣␣bot=%10.5f\n", *r.slab.b_top_slide*, *r.slab.cos_angle*,
r.slab.b_bottom_slide);
fprintf(*stderr*, "search␣=␣%d␣quadrature␣points␣=␣%d\n", *r.search*, *r.method.quad_pts*);
}

This code is used in section 234.

292.

⟨ Prototype for *Print_Measure_Type* 292 ⟩ ≡
void *Print_Measure_Type*(**struct measure_type** *m*)

This code is used in sections 235 and 293.

293.

```
⟨ Definition for Print_Measure_Type 293 ⟩ ≡
  ⟨ Prototype for Print_Measure_Type 292 ⟩
  {
    fprintf(stderr, "\n");
    fprintf(stderr, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Beam␣diameter␣=␣%7.1f␣mm\n", m.d_beam);
    fprintf(stderr, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Sample␣thickness␣=␣%7.1f␣mm\n", m.slab_thickness);
    fprintf(stderr, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Top␣slide␣thickness␣=␣%7.1f␣mm\n",
        m.slab_top_slide_thickness);
    fprintf(stderr, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Bottom␣slide␣thickness␣=␣%7.1f␣mm\n",
        m.slab_bottom_slide_thickness);
    fprintf(stderr, "#␣␣␣␣␣␣␣␣␣␣␣Sample␣index␣of␣refraction␣=␣%7.3f\n", m.slab_index);
    fprintf(stderr, "#␣␣␣␣␣␣␣␣␣Top␣slide␣index␣of␣refraction␣=␣%7.3f\n", m.slab_top_slide_index);
    fprintf(stderr, "#␣␣␣␣␣␣Bottom␣slide␣index␣of␣refraction␣=␣%7.3f\n", m.slab_bottom_slide_index);
    fprintf(stderr, "#␣␣␣␣Unscattered␣light␣collected␣in␣M_R␣=␣%7.1f␣%%\n", m.sphere_with_rc * 100);
    fprintf(stderr, "#␣␣␣␣Unscattered␣light␣collected␣in␣M_T␣=␣%7.1f␣%%\n", m.sphere_with_tc * 100);
    fprintf(stderr, "#␣\n");
    fprintf(stderr, "#␣Reflection␣sphere\n");
    fprintf(stderr, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣sphere␣diameter␣=␣%7.1f␣mm\n", m.d_sphere_r);
    fprintf(stderr, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣sample␣port␣diameter␣=␣%7.1f␣mm\n",
        2 * m.d_sphere_r * sqrt(m.as_r));
    fprintf(stderr, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣entrance␣port␣diameter␣=␣%7.1f␣mm\n",
        2 * m.d_sphere_r * sqrt(m.ae_r));
    fprintf(stderr, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣detector␣port␣diameter␣=␣%7.1f␣mm\n",
        2 * m.d_sphere_r * sqrt(m.ad_r));
    fprintf(stderr, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣wall␣reflectance␣=␣%7.1f␣%%\n", m.rw_r * 100);
    fprintf(stderr, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣standard␣reflectance␣=␣%7.1f␣%%\n", m.rstd_r * 100);
    fprintf(stderr, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣detector␣reflectance␣=␣%7.1f␣%%\n", m.rd_r * 100);
    fprintf(stderr, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣spheres␣=␣%7d\n", m.num_spheres);
    fprintf(stderr, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣measures␣=␣%7d\n", m.num_measures);
    fprintf(stderr, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣method␣=␣%7d\n", m.method);
    fprintf(stderr, "area_r␣as=%10.5f␣␣ad=%10.5f␣␣␣␣ae=%10.5f␣␣aw=%10.5f\n", m.as_r, m.ad_r,
        m.ae_r, m.aw_r);
    fprintf(stderr, "refls␣␣rd=%10.5f␣␣rw=%10.5f␣␣rstd=%10.5f␣␣␣f=%10.5f\n", m.rd_r, m.rw_r,
        m.rstd_r, m.f_r);
    fprintf(stderr, "area_t␣as=%10.5f␣␣ad=%10.5f␣␣␣␣ae=%10.5f␣␣aw=%10.5f\n", m.as_t, m.ad_t,
        m.ae_t, m.aw_t);
    fprintf(stderr, "refls␣␣rd=%10.5f␣␣rw=%10.5f␣␣rstd=%10.5f␣␣␣f=%10.5f\n", m.rd_t, m.rw_t,
        m.rstd_t, m.f_t);
    fprintf(stderr, "lost␣␣ur1=%10.5f␣ut1=%10.5f␣␣␣uru=%10.5f␣␣utu=%10.5f\n", m.ur1_lost,
        m.ut1_lost, m.utu_lost, m.utu_lost);
  }
```

This code is used in section 234.

294. Index. Here is a cross-reference table for the inverse 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.

⟨ Allocate local simplex variables 195 ⟩ Used in sections 194, 214, 219, 225, and 230.
⟨ Calc M_R and M_T for dual beam sphere 160 ⟩ Used in section 157.
⟨ Calc M_R and M_T for no spheres 158 ⟩ Used in section 157.
⟨ Calc M_R and M_T for single beam sphere 159 ⟩ Used in section 157.
⟨ Calc M_R and M_T for two spheres 161 ⟩ Used in section 157.
⟨ Calculate and Print the Forward Calculation 6 ⟩ Used in section 2.
⟨ Calculate and write optical properties 8 ⟩ Used in section 2.
⟨ Calculate specular reflection and transmission 238 ⟩ Used in section 237.
⟨ Calculate the deviation 162 ⟩ Used in section 157.
⟨ Calculate the unscattered transmission and reflection 244 ⟩ Used in section 243.
⟨ Check MR for zero or one spheres 45 ⟩ Used in section 44.
⟨ Check MT for zero or one spheres 46 ⟩ Used in section 44.
⟨ Check MU 47 ⟩ Used in section 44.
⟨ Check for bad values of Tc 239 ⟩ Used in section 237.
⟨ Check sphere parameters 48, 49 ⟩ Used in section 44.
⟨ Choose the best node of the a and b simplex 199 ⟩ Used in section 194.
⟨ Choose the best node of the a and g simplex 217 ⟩ Used in section 214.
⟨ Choose the best node of the ba and g simplex 228 ⟩ Used in section 225.
⟨ Choose the best node of the bs and g simplex 233 ⟩ Used in section 230.
⟨ Choose the best node of the b and g simplex 223 ⟩ Used in section 219.
⟨ Clean up guesses 284 ⟩ Used in section 274.
⟨ Command-line changes to m 15 ⟩ Used in section 2.
⟨ Command-line changes to r 10 ⟩ Used in sections 2 and 8.
⟨ Count command-line measurements 16 ⟩ Used in section 2.
⟨ Declare variables for *main* 4 ⟩ Used in section 2.
⟨ Definition for *Allocate_Grid* 122 ⟩ Used in section 100.
⟨ Definition for *Calculate_Distance_With_Corrections* 157 ⟩ Used in section 100.
⟨ Definition for *Calculate_Distance* 153 ⟩ Used in section 100.
⟨ Definition for *Calculate_Grid_Distance* 155 ⟩ Used in section 100.
⟨ Definition for *Calculate_MR_MT* 72 ⟩ Used in section 35.
⟨ Definition for *Calculate_Minimum_MR* 74 ⟩ Used in section 35.
⟨ Definition for *Debug* 289 ⟩ Used in section 234.
⟨ Definition for *Estimate_RT* 243 ⟩ Used in section 234.
⟨ Definition for *Fill_AB_Grid* 137 ⟩ Used in section 100.
⟨ Definition for *Fill_AG_Grid* 142 ⟩ Used in section 100.
⟨ Definition for *Fill_BG_Grid* 145 ⟩ Used in section 100.
⟨ Definition for *Fill_BaG_Grid* 147 ⟩ Used in section 100.
⟨ Definition for *Fill_BsG_Grid* 149 ⟩ Used in section 100.
⟨ Definition for *Fill_Grid* 151 ⟩ Used in section 100.
⟨ Definition for *Find_AB_fn* 169 ⟩ Used in section 100.
⟨ Definition for *Find_AG_fn* 167 ⟩ Used in section 100.
⟨ Definition for *Find_A_fn* 175 ⟩ Used in section 100.
⟨ Definition for *Find_BG_fn* 181 ⟩ Used in section 100.
⟨ Definition for *Find_B_fn* 177 ⟩ Used in section 100.
⟨ Definition for *Find_BaG_fn* 183 ⟩ Used in section 100.
⟨ Definition for *Find_Ba_fn* 171 ⟩ Used in section 100.
⟨ Definition for *Find_BsG_fn* 185 ⟩ Used in section 100.
⟨ Definition for *Find_Bs_fn* 173 ⟩ Used in section 100.
⟨ Definition for *Find_G_fn* 179 ⟩ Used in section 100.
⟨ Definition for *Gain_11* 107 ⟩ Used in section 100.
⟨ Definition for *Gain_22* 109 ⟩ Used in section 100.
⟨ Definition for *Gain* 105 ⟩ Used in section 100.

⟨ Definition for *Get_Calc_State* 118 ⟩ Used in section 100.
⟨ Definition for *Grid_ABG* 124 ⟩ Used in section 100.
⟨ Definition for *Initialize_Measure* 63 ⟩ Used in section 35.
⟨ Definition for *Initialize_Result* 55 ⟩ Used in section 35.
⟨ Definition for *Inverse_RT* 39 ⟩ Used in section 35.
⟨ Definition for *Max_Light_Loss* 189 ⟩ Used in section 100.
⟨ Definition for *MinMax_MR_MT* 76 ⟩ Used in section 35.
⟨ Definition for *Near_Grid_Points* 134 ⟩ Used in section 100.
⟨ Definition for *Print_Invert_Type* 291 ⟩ Used in section 234.
⟨ Definition for *Print_Measure_Type* 293 ⟩ Used in section 234.
⟨ Definition for *Read_Data_Line* 85 ⟩ Used in section 77.
⟨ Definition for *Read_Header* 81 ⟩ Used in section 77.
⟨ Definition for *Same_Calc_State* 120 ⟩ Used in section 100.
⟨ Definition for *Set_Calc_State* 116 ⟩ Used in section 100.
⟨ Definition for *Set_Debugging* 287 ⟩ Used in section 234.
⟨ Definition for *Spheres_Inverse_RT* 65 ⟩ Used in section 35.
⟨ Definition for *Two_Sphere_R* 111 ⟩ Used in section 100.
⟨ Definition for *Two_Sphere_T* 113 ⟩ Used in section 100.
⟨ Definition for *U_Find_AB* 194 ⟩ Used in section 191.
⟨ Definition for *U_Find_AG* 214 ⟩ Used in section 191.
⟨ Definition for *U_Find_A* 207 ⟩ Used in section 191.
⟨ Definition for *U_Find_BG* 219 ⟩ Used in section 191.
⟨ Definition for *U_Find_BaG* 225 ⟩ Used in section 191.
⟨ Definition for *U_Find_Ba* 205 ⟩ Used in section 191.
⟨ Definition for *U_Find_BsG* 230 ⟩ Used in section 191.
⟨ Definition for *U_Find_Bs* 203 ⟩ Used in section 191.
⟨ Definition for *U_Find_B* 211 ⟩ Used in section 191.
⟨ Definition for *U_Find_G* 209 ⟩ Used in section 191.
⟨ Definition for *Valid_Grid* 126 ⟩ Used in section 100.
⟨ Definition for *What_Is_B* 237 ⟩ Used in section 234.
⟨ Definition for *Write_Header* 93 ⟩ Used in section 77.
⟨ Definition for *a2acalc* 249 ⟩ Used in section 234.
⟨ Definition for *abg_distance* 132 ⟩ Used in section 100.
⟨ Definition for *abgb2ag* 267 ⟩ Used in section 234.
⟨ Definition for *abgg2ab* 265 ⟩ Used in section 234.
⟨ Definition for *acalc2a* 251 ⟩ Used in section 234.
⟨ Definition for *b2bcalc* 257 ⟩ Used in section 234.
⟨ Definition for *bcalc2b* 259 ⟩ Used in section 234.
⟨ Definition for *check_magic* 91 ⟩ Used in section 77.
⟨ Definition for *determine_search* 51 ⟩ Used in section 35.
⟨ Definition for *ez_Inverse_RT* 61 ⟩ Used in section 35.
⟨ Definition for *fill_grid_entry* 135 ⟩ Used in section 100.
⟨ Definition for *g2gcalc* 253 ⟩ Used in section 234.
⟨ Definition for *gcalc2g* 255 ⟩ Used in section 234.
⟨ Definition for *maxloss* 187 ⟩ Used in section 100.
⟨ Definition for *measure_OK* 44 ⟩ Used in section 35.
⟨ Definition for *quick_guess* 274 ⟩ Used in section 234.
⟨ Definition for *read_number* 89 ⟩ Used in section 77.
⟨ Definition for *skip_white* 87 ⟩ Used in section 77.
⟨ Definition for *slow_guess* 269 ⟩
⟨ Definition for *twoprime* 261 ⟩ Used in section 234.
⟨ Definition for *twounprime* 263 ⟩ Used in section 234.

⟨ Estimate the backscattered reflection 245 ⟩ Used in section 243.
⟨ Estimate the scattered transmission 246 ⟩ Used in section 243.
⟨ Estimate *aprime* 275 ⟩ Used in section 274.
⟨ Estimate *bprime* 276 ⟩ Used in sections 278, 282, and 283.
⟨ Evaluate the *BaG* simplex at the nodes 227 ⟩ Used in section 225.
⟨ Evaluate the *BsG* simplex at the nodes 232 ⟩ Used in section 230.
⟨ Evaluate the *a* and *b* simplex at the nodes 198 ⟩ Used in section 194.
⟨ Evaluate the *a* and *g* simplex at the nodes 216 ⟩ Used in section 214.
⟨ Evaluate the *bg* simplex at the nodes 222 ⟩ Used in section 219.
⟨ Exit with bad input data 40 ⟩ Used in section 39.
⟨ Fill *r* with reasonable values 56, 57, 58, 59 ⟩ Used in section 55.
⟨ Find the optical properties 41 ⟩ Used in section 39.
⟨ Find thickness when multiple internal reflections are present 241 ⟩ Used in section 237.
⟨ Free simplex data structures 201 ⟩ Used in sections 194, 214, 219, 225, and 230.
⟨ Generate next albedo using j 139, 140 ⟩ Used in sections 137 and 142.
⟨ Get the initial *a*, *b*, and *g* 196 ⟩ Used in sections 194, 214, 219, 225, and 230.
⟨ Guess when all three measurements are known 279 ⟩ Used in section 274.
⟨ Guess when finding albedo 280 ⟩ Used in section 279.
⟨ Guess when finding anisotropy and albedo 283 ⟩ Used in section 279.
⟨ Guess when finding optical depth 281 ⟩ Used in section 279.
⟨ Guess when finding the albedo and optical depth 282 ⟩ Used in section 279.
⟨ Guess when only reflection is known 277 ⟩ Used in section 274.
⟨ Guess when reflection and transmission are known 278 ⟩ Used in section 274.
⟨ Handle options 5 ⟩ Used in section 2.
⟨ Improve result using Monte Carlo 12 ⟩ Used in section 8.
⟨ Include files for *main* 3 ⟩ Used in section 2.
⟨ Initialize the nodes of the *a* and *b* simplex 197 ⟩ Used in section 194.
⟨ Initialize the nodes of the *a* and *g* simplex 215 ⟩ Used in section 214.
⟨ Initialize the nodes of the *ba* and *g* simplex 226 ⟩ Used in section 225.
⟨ Initialize the nodes of the *bs* and *g* simplex 231 ⟩ Used in section 230.
⟨ Initialize the nodes of the *b* and *g* simplex 221 ⟩ Used in section 219.
⟨ Iteratively solve for *b* 212 ⟩ Used in section 211.
⟨ Local Variables for Calculation 9 ⟩ Used in section 8.
⟨ Nonworking code 138 ⟩
⟨ One parameter deviation 163 ⟩ Used in section 162.
⟨ One parameter search 52 ⟩ Used in section 51.
⟨ Print diagnostics 165 ⟩ Used in section 157.
⟨ Print results function 22 ⟩ Used in section 2.
⟨ Prototype for *Allocate_Grid* 121 ⟩ Used in sections 101 and 122.
⟨ Prototype for *Calculate_Distance_With_Corrections* 156 ⟩ Used in sections 101 and 157.
⟨ Prototype for *Calculate_Distance* 152 ⟩ Used in sections 101 and 153.
⟨ Prototype for *Calculate_Grid_Distance* 154 ⟩ Used in sections 101 and 155.
⟨ Prototype for *Calculate_MR_MT* 71 ⟩ Used in sections 36 and 72.
⟨ Prototype for *Calculate_Minimum_MR* 73 ⟩ Used in sections 36 and 74.
⟨ Prototype for *Debug* 288 ⟩ Used in sections 235 and 289.
⟨ Prototype for *Estimate_RT* 242 ⟩ Used in sections 235 and 243.
⟨ Prototype for *Fill_AB_Grid* 136 ⟩ Used in sections 100 and 137.
⟨ Prototype for *Fill_AG_Grid* 141 ⟩ Used in sections 100 and 142.
⟨ Prototype for *Fill_BG_Grid* 144 ⟩ Used in sections 101 and 145.
⟨ Prototype for *Fill_BaG_Grid* 146 ⟩ Used in sections 101 and 147.
⟨ Prototype for *Fill_BsG_Grid* 148 ⟩ Used in sections 101 and 149.
⟨ Prototype for *Fill_Grid* 150 ⟩ Used in sections 101 and 151.

⟨ Prototype for *Find_AB_fn* 168 ⟩ Used in sections 101 and 169.
⟨ Prototype for *Find_AG_fn* 166 ⟩ Used in sections 101 and 167.
⟨ Prototype for *Find_A_fn* 174 ⟩ Used in sections 101 and 175.
⟨ Prototype for *Find_BG_fn* 180 ⟩ Used in sections 101 and 181.
⟨ Prototype for *Find_B_fn* 176 ⟩ Used in sections 101 and 177.
⟨ Prototype for *Find_BaG_fn* 182 ⟩ Used in sections 101 and 183.
⟨ Prototype for *Find_Ba_fn* 170 ⟩ Used in sections 101 and 171.
⟨ Prototype for *Find_BsG_fn* 184 ⟩ Used in sections 101 and 185.
⟨ Prototype for *Find_Bs_fn* 172 ⟩ Used in sections 101 and 173.
⟨ Prototype for *Find_G_fn* 178 ⟩ Used in sections 101 and 179.
⟨ Prototype for *Gain_11* 106 ⟩ Used in sections 101 and 107.
⟨ Prototype for *Gain_22* 108 ⟩ Used in sections 101 and 109.
⟨ Prototype for *Gain* 104 ⟩ Used in sections 101 and 105.
⟨ Prototype for *Get_Calc_State* 117 ⟩ Used in sections 101 and 118.
⟨ Prototype for *Grid_ABG* 123 ⟩ Used in sections 101 and 124.
⟨ Prototype for *Initialize_Measure* 62 ⟩ Used in sections 36 and 63.
⟨ Prototype for *Initialize_Result* 54 ⟩ Used in sections 36 and 55.
⟨ Prototype for *Inverse_RT* 38 ⟩ Used in sections 36 and 39.
⟨ Prototype for *Max_Light_Loss* 188 ⟩ Used in sections 101 and 189.
⟨ Prototype for *MinMax_MR_MT* 75 ⟩ Used in sections 36 and 76.
⟨ Prototype for *Near_Grid_Points* 133 ⟩ Used in sections 101 and 134.
⟨ Prototype for *Print_Invert_Type* 290 ⟩ Used in sections 235 and 291.
⟨ Prototype for *Print_Measure_Type* 292 ⟩ Used in sections 235 and 293.
⟨ Prototype for *Read_Data_Line* 84 ⟩ Used in sections 78 and 85.
⟨ Prototype for *Read_Header* 80 ⟩ Used in sections 78 and 81.
⟨ Prototype for *Same_Calc_State* 119 ⟩ Used in sections 101 and 120.
⟨ Prototype for *Set_Calc_State* 115 ⟩ Used in sections 101 and 116.
⟨ Prototype for *Set_Debugging* 286 ⟩ Used in sections 235 and 287.
⟨ Prototype for *Spheres_Inverse_RT* 64 ⟩ Used in sections 37 and 65.
⟨ Prototype for *Two_Sphere_R* 110 ⟩ Used in sections 101 and 111.
⟨ Prototype for *Two_Sphere_T* 112 ⟩ Used in sections 101 and 113.
⟨ Prototype for *U_Find_AB* 193 ⟩ Used in sections 192 and 194.
⟨ Prototype for *U_Find_AG* 213 ⟩ Used in sections 192 and 214.
⟨ Prototype for *U_Find_A* 206 ⟩ Used in sections 192 and 207.
⟨ Prototype for *U_Find_BG* 218 ⟩ Used in sections 192 and 219.
⟨ Prototype for *U_Find_BaG* 224 ⟩ Used in sections 192 and 225.
⟨ Prototype for *U_Find_Ba* 204 ⟩ Used in sections 192 and 205.
⟨ Prototype for *U_Find_BsG* 229 ⟩ Used in sections 192 and 230.
⟨ Prototype for *U_Find_Bs* 202 ⟩ Used in sections 192 and 203.
⟨ Prototype for *U_Find_B* 210 ⟩ Used in sections 192 and 211.
⟨ Prototype for *U_Find_G* 208 ⟩ Used in sections 192 and 209.
⟨ Prototype for *Valid_Grid* 125 ⟩ Used in sections 101 and 126.
⟨ Prototype for *What_Is_B* 236 ⟩ Used in sections 235 and 237.
⟨ Prototype for *Write_Header* 92 ⟩ Used in sections 78 and 93.
⟨ Prototype for *a2acalc* 248 ⟩ Used in sections 235 and 249.
⟨ Prototype for *abg_distance* 131 ⟩ Used in sections 101 and 132.
⟨ Prototype for *abgb2ag* 266 ⟩ Used in sections 235 and 267.
⟨ Prototype for *abgg2ab* 264 ⟩ Used in sections 235 and 265.
⟨ Prototype for *acalc2a* 250 ⟩ Used in sections 235 and 251.
⟨ Prototype for *b2bcalc* 256 ⟩ Used in sections 235 and 257.
⟨ Prototype for *bcalc2b* 258 ⟩ Used in sections 235 and 259.
⟨ Prototype for *check_magic* 90 ⟩ Used in section 91.

Inverse Adding-Doubling

(Version 3.6.3)