1. Main Program.

Here is a quick skeleton that I put together to show how the inverse adding-doubling code works. I have only cursorily tested this. If you find obvious bugs, they are probably real but should not extend beyond this code snippet.

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 11 ⟩
  ⟨ print usage function 12 ⟩
  ⟨ stringdup together function 15 ⟩
  ⟨ seconds elapsed function 16 ⟩
  ⟨ print error legend 14 ⟩
  ⟨ print dot function 18 ⟩
  ⟨ calculate coefficients function 13 ⟩
  ⟨ parse string into array function 17 ⟩
  int main(int argc, char **argv)
  {
    ⟨ Declare variables for main 4 ⟩
    ⟨ Handle options 5 ⟩
    Initialize_Measure(&m);
    if (process_command_line) {
      r.method.quad_pts = 8;
      ⟨ Handle command-line measurements 10 ⟩
      if (m.num_spheres ≡ 0) MC_iterations = 0;
      ⟨ Calculate and write optical properties 7 ⟩
      return 0;
    }
    ⟨ prepare file for reading 6 ⟩
    if (Read_Header(stdin, &m, &params) ≡ 0) {
      start_time = clock( );
      while (Read_Data_Line(stdin, &m, params) ≡ 0) {
        ⟨ Calculate and write optical properties 7 ⟩
        first_line = 0;
      }
    }
    if (¬quiet) fprintf(stderr, "\n");
    if (any_error) 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
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <math.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"
  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;
char *g_out_name = Λ;
char c;
int machine_readable_output = 0;
int first_line = 1;
int quiet = 0;
int show_all = 0;
long n_photons = 100000;
int cl_quadrature_points = 8;
int MC_iterations = 20;
int any_error = 0;
int process_command_line = 0;
int params = 0;
double cl_default_a = UNINITIALIZED;
double cl_default_g = UNINITIALIZED;
double cl_default_b = UNINITIALIZED;
double cl_tolerance = UNINITIALIZED;
double cl_sample_d = UNINITIALIZED;
double cl_sample_n = UNINITIALIZED;
double cl_slide_d = UNINITIALIZED;
double cl_slide_n = UNINITIALIZED;
double cl_UR1 = UNINITIALIZED;
double cl_UT1 = UNINITIALIZED;
double cl_Tc = 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:b:B:d:D:e:g:hn:N:mM:o:p:q:Qr:sS:t:u:v")) ≠ EOF) {
    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 'b': cl_default_b = strtod(optarg, Λ);
      break;
    case 'B': Set_Grid_Debugging((unsigned int) 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 'g': cl_default_g = strtod(optarg, Λ);
      break;
    case 'o': g_out_name = strdup(optarg);
      break;
    case 'n': cl_sample_n = strtod(optarg, Λ);
      break;
    case 'N': cl_slide_n = strtod(optarg, Λ);
      break;
    case 'p': n_photons = (int) strtod(optarg, Λ);
      break;
    case 'm': machine_readable_output = 1;
      quiet = 1;
      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);
      }
      break;
    case 'M': MC_iterations = (int) strtod(optarg, Λ);
      break;
    case 'Q': quiet = 1;
      break;
    case 'r': cl_UR1 = strtod(optarg, Λ);
      process_command_line = 1;
      break;
    case 's': show_all = 1;
      break;
    case 'S': cl_num_spheres = (int) strtod(optarg, Λ);
      break;
    case 't': cl_UT1 = strtod(optarg, Λ);
      process_command_line = 1;
```

```
      break;
    case 'u': cl_Tc = strtod(optarg, Λ);
      process_command_line = 1;
      break;
    case 'v': print_version( );
      break;
    default: case 'h': case '?': print_usage( );
      break;
    }
  }
  argc -= optind;
  argv += optind;
```

This code is used in section 2.

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

```
⟨ prepare file for reading 6 ⟩ ≡
  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.

7. 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 7 ⟩ ≡
  { static int rt_data_count = 0;
  static int mc_iter_count = 0;
  struct measure_type m_mc, m_none;
  struct invert_type r_mc, r_none;
  int mc_iter = 0;
  double ur1 = 0;
  double ut1 = 0;
  double uru = 0;
  double utu = 0;
  double ur1_lost = 0;
  double ut1_lost = 0;
  double uru_lost = 0;
  double utu_lost = 0;
  double mu_a_none = 0;
  double mu_a_sphere = 0;
  double mu_a_both = 0;
  double mu_a_mc = 0;
  double mu_sp_none = 0;
  double mu_sp_sphere = 0;
  double mu_sp_both = 0;
  double mu_sp_mc = 0;
  double LR = 0;
  double LT = 0;
  rt_data_count++;
  Initialize_Result(m, &r);
  r.method.quad_pts = cl_quadrature_points;
  if (cl_default_a ≠ UNINITIALIZED) {
    r.default_a = cl_default_a;
  }
  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;
  }
  r.search = determine_search(m, r);
  if (rt_data_count ≡ 1 ∧ ¬machine_readable_output) {
    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, "# No photons were killed trying to figure out much light was lost\n");
    fprintf(stdout, "#\n");
    if (show_all) {
      fprintf(stdout, "#      \tMeas\tCalc \t");
      fprintf(stdout, "Meas \tCalc \tBest \tBest \tBest \t");
      fprintf(stdout, "Lost \tLost \tLost \tLost \tNone \tSpOnly\tMConly\t");
      fprintf(stdout, "None \tSpOnly\tMConly\tMC\tError\n");
      fprintf(stdout, "##wave\tM_R \tM_R   \t");
      fprintf(stdout, "M_T \tM_T   \tmu_a\tmu_s'\t   g\tR_drct\tR_diff\tT_drct\tT_diff\t");
      fprintf(stdout, "mu_a\tmu_a\tmu_a\tmu_s'\tmu_s'\tmu_s'\titer\tNumber\n");
      fprintf(stdout, "# [nm] \t[---]\t[---]\t[---]\t[---]\t");
      fprintf(stdout, "[1/mm]\t[1/mm]\t[---]\t[---]\t[---]\t[---]\t");
      fprintf(stdout, "[1/mm]\t[1/mm]\t");
      fprintf(stdout, "[1/mm]\t[1/mm]\t[1/mm]\t[1/mm]\t[---]\t[---]\n");
    }
    else {
      fprintf(stdout, "#      \tMeas\tCalc\t");
      fprintf(stdout, "Meas\tCalc\t");
      fprintf(stdout, "Calc\tCalc\tCalc\tError\n");
      fprintf(stdout, "##wave\tM_R \tM_R \tM_T \tM_T \t");
      fprintf(stdout, "mu_a\tmu_s'\tg\tNumber\n");
      fprintf(stdout, "# [nm] \t[---]\t[---]\t[---]\t[---]\t");
      fprintf(stdout, "[1/mm]\t[1/mm]\t[---]\t[---]\n");
    }
  }
  m.ur1_lost = 0;
  m.ut1_lost = 0;
  m.uru_lost = 0;
  m.utu_lost = 0; r . error = IAD_NO_ERROR;
  m.f_r = 0.0;
  memcpy(&m_none, &m, sizeof(struct measure_type));
  memcpy(&r_none, &r, sizeof(struct invert_type));
  memcpy(&m_mc, &m, sizeof(struct measure_type));
  memcpy(&r_mc, &r, sizeof(struct invert_type));
  Inverse_RT(m, &r);
  calculate_coefficients(m, r, &LR, &LT, &mu_sp_sphere, &mu_a_sphere);
    /* use the sphere corrected values as starting values */
  mu_sp_both = mu_sp_sphere;
  mu_sp_none = mu_sp_both;
  mu_sp_mc = mu_sp_both;
  mu_a_both = mu_a_sphere;
  mu_a_mc = mu_a_both;
  mu_a_none = mu_a_both;
    /* It makes no sense to do MC light loss when there is no sample port size! */
  if ( m.num_spheres > 0 ∧ r . error ≡ IAD_NO_ERROR ) {
  if (show_all) {      /* Worst possible estimate */
    m_none.num_spheres = 0;
    Inverse_RT(m_none, &r_none);
    calculate_coefficients(m_none, r_none, &LR, &LT, &mu_sp_none, &mu_a_none);
  }
  while ( mc_iter < MC_iterations ∧ r . error ≡ IAD_NO_ERROR ) { double ur1_max_loss, ut1_max_loss;
```

```
double mu_sp_last = mu_sp_both;
double mu_a_last = mu_a_both;
MC_Lost(m, r, n_photons, &ur1, &ut1, &uru, &utu, &ur1_lost, &ut1_lost, &uru_lost, &utu_lost);
mc_iter_count++;
mc_iter++;      /* now include both sphere and light loss corrections */
m.ur1_lost = ur1_lost;
m.ut1_lost = ut1_lost;
m.uru_lost = uru_lost;
m.utu_lost = utu_lost;
Max_Light_Loss(m, r, &ur1_max_loss, &ut1_max_loss);
m.ur1_lost = ur1_max_loss;
m.ut1_lost = ut1_max_loss; if (0 ∧ ur1_lost + m.m_r + ut1_lost + m.m_t > 1) { struct AD_slab_type s;
double ad_ur1, ad_ut1, ad_uru, ad_utu;
s.a = r.a;
s.b = r.b;
s.g = r.g;
s.phase_function = HENYEY_GREENSTEIN;
s.n_slab = m.slab_index;
s.n_top_slide = m.slab_top_slide_index;
s.n_bottom_slide = m.slab_top_slide_index;
s.b_top_slide = 0;
s.b_bottom_slide = 0;
RT(32, &s, &ad_ur1, &ad_ut1, &ad_uru, &ad_utu);
fprintf(stderr, "UR1_AD=%7.5f␣UR1_MC=%7.5f␣", ad_ur1, ur1);
fprintf(stderr, "UR1_LOST=%7.5f␣UR1_V1=%7.5f\n", ur1_lost, m.m_r);
fprintf(stderr, "UT1_AD=%7.5f␣UT1_MC=%7.5f␣", ad_ut1, ut1);
fprintf(stderr, "UT1_LOST=%7.5f␣UT1_V2=%7.5f\n", ut1_lost, m.m_t); r . error =
    IAD_EXCESSIVE_LIGHT_LOSS;
break; } Inverse_RT(m, &r);
calculate_coefficients(m, r, &LR, &LT, &mu_sp_both, &mu_a_both);
  /* no sphere corrections, but MC light loss corrections */
if (show_all) {
  m_mc.num_spheres = 0;
  m_mc.ur1_lost = ur1_lost;
  m_mc.ut1_lost = ut1_lost;
  m_mc.uru_lost = uru_lost;
  m_mc.utu_lost = utu_lost;
  Inverse_RT(m_mc, &r_mc);
  calculate_coefficients(m_mc, r_mc, &LR, &LT, &mu_sp_mc, &mu_a_mc);
}
if (fabs(mu_a_last − mu_a_both)/(mu_a_both + 0.0001) < r.MC_tolerance ∧ fabs(mu_sp_last −
      mu_sp_both)/(mu_sp_both + 0.0001) < r.MC_tolerance) break;
if (¬quiet) print_dot (start_time, r . error , mc_iter_count, rt_data_count, mc_iter, &any_error ) ; } }
if (m.lambda ≠ 0) fprintf(stdout, "%6.1f", m.lambda);
else fprintf(stdout, "␣␣␣␣␣␣");
fprintf(stdout, "\t");
fprintf(stdout, "%6.4f\t%6.4f\t", m.m_r, LR);
fprintf(stdout, "%6.4f\t%6.4f\t", m.m_t, LT);
fprintf(stdout, "%6.4f\t", mu_a_both);
fprintf(stdout, "%6.4f\t", mu_sp_both);
fprintf(stdout, "%6.4f\t", r.g);
if (show_all) {
```

```
    fprintf(stdout, "%6.4f\t%6.4f\t", m.ur1_lost, m.uru_lost);
    fprintf(stdout, "%6.4f\t%6.4f\t", m.ut1_lost, m.utu_lost);
    fprintf(stdout, "%6.4f\t", mu_a_none);
    fprintf(stdout, "%6.4f\t", mu_a_mc);
    fprintf(stdout, "%6.4f\t", mu_a_sphere);
    fprintf(stdout, "%6.4f\t", mu_sp_none);
    fprintf(stdout, "%6.4f\t", mu_sp_mc);
    fprintf(stdout, "%6.4f\t", mu_sp_sphere);
    fprintf(stdout, "%2d\t", mc_iter);
  }
  fprintf (stdout, "%2d\n", r . error ) ;
  fflush(stdout); if (¬quiet) print_dot (start_time, r . error , mc_iter_count, rt_data_count, 99, &any_error
      ) ; r . error = IAD_NO_ERROR; }
```

This code is used in section 2.

8. ⟨ Testing MC code 8 ⟩ ≡

```
{
  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);
}
```

9. ⟨ old formatting 9 ⟩ ≡

```
if (¬quiet ∧ count % 100 ≡ 0) fprintf(stderr, "\n");
if (¬machine_readable_output) 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);
```

10. put the values for command line arguments into the measurement record and set up the result record to handle the arguments properly so that the optical properties can be determined.

```
⟨ Handle command-line measurements 10 ⟩ ≡
  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_sample_d ≠ UNINITIALIZED) m.slab_thickness = cl_sample_d;
  if (cl_slide_d ≠ UNINITIALIZED) {
    m.slab_bottom_slide_thickness = cl_slide_d;
    m.slab_top_slide_thickness = cl_slide_d;
  }
  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_num_spheres ≠ UNINITIALIZED) m.num_spheres = (int) cl_num_spheres;
  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) * (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.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_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) * (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.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 = 0;
  s.b_bottom_slide = 0;
  cl_default_b = What_Is_B(s, m.m_u);
}
if (m.num_measures ≡ 1) {
  cl_default_b = HUGE_VAL;
}
```

This code is used in section 2.

11. ⟨ print version function 11 ⟩ ≡

```
static void print_version(void)
{
  fprintf(stderr, "iad␣%s\n", Version);
  fprintf(stderr, "Copyright␣2007␣Scott␣Prahl,␣prahl@bme.ogi.edu\n");
  fprintf(stderr, "␣␣␣␣␣␣␣␣␣␣(see␣Applied␣Optics,␣32:559-568,␣1993)\n");
  exit(0);
}
```

This code is used in section 2.

12. ⟨ print usage function 12 ⟩ ≡

```
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, "␣␣-a␣#␣␣␣␣␣␣␣␣␣␣␣␣use␣this␣albedo␣\n");
  fprintf(stderr, "␣␣-b␣#␣␣␣␣␣␣␣␣␣␣␣␣use␣this␣optical␣thickness␣\n");
  fprintf(stderr, "␣␣-e␣#␣␣␣␣␣␣␣␣␣␣␣␣error␣tolerance␣(default␣0.0001)␣\n");
  fprintf(stderr, "␣␣-g␣#␣␣␣␣␣␣␣␣␣␣␣␣default␣anisotropy␣(default␣0)␣\n");
  fprintf(stderr, "␣␣-h␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣display␣help\n");
  fprintf(stderr, "␣␣-m␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣machine␣readable␣output\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, "␣␣-Q␣␣␣␣␣␣␣␣␣␣␣␣␣␣quiet␣--␣silence␣output␣to␣stderr\n\n");
  fprintf(stderr, "␣␣-r␣#␣␣␣␣␣␣␣␣␣␣␣␣total␣reflection␣measurement\n");
  fprintf(stderr, "␣␣-s␣␣␣␣␣␣␣␣␣␣␣␣␣␣show␣sphere␣and␣MC␣light␣loss␣effects\n");
  fprintf(stderr, "␣␣-S␣#␣␣␣␣␣␣␣␣␣␣␣␣number␣of␣spheres␣used\n");
  fprintf(stderr, "␣␣-t␣#␣␣␣␣␣␣␣␣␣␣␣␣total␣transmission␣measurement\n");
  fprintf(stderr, "␣␣-u␣#␣␣␣␣␣␣␣␣␣␣␣␣unscattered␣transmission␣measurement\n");
  fprintf(stderr, "␣␣-v␣␣␣␣␣␣␣␣␣␣␣␣␣␣version␣information\n");
  fprintf(stderr, "Examples:\n");
  fprintf(stderr, "␣␣iad␣data␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣optical␣values␣put␣in␣data.txt\n");
  fprintf(stderr, "␣␣iad␣-e␣0.0001␣data␣␣␣␣␣␣␣␣␣better␣convergence␣to␣R␣&␣T␣values\n");
  fprintf(stderr, "␣␣iad␣-m␣data␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣data.abg␣in␣machine␣readable␣format\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␣-s␣data␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣show␣sphere␣and␣MC␣light␣loss␣effects\n");
  fprintf(stderr, "␣␣apply␣iad␣data1␣data2␣␣␣␣␣␣␣process␣multiple␣files\n\n");
  fprintf(stderr, "Report␣bugs␣to␣<prahl@bme.ogi.edu>\n\n");
  exit(0);
}
```

This code is used in section 2.

13. This can only be called immediately after *Invert_RT* You have been warned!

```
⟨ calculate coefficients function 13 ⟩ ≡
  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;
    *musp = 0;
    *mua = 0; if ( r . error ≠ IAD_NO_ERROR ) return;
    Calculate_Distance(LR, LT, &delta);
    if (r.default_b ≡ HUGE_VAL ∨ r.b ≡ HUGE_VAL) {
      double mus;
      mus = 1;
      *musp = mus * (1.0 − r.g);
      if (r.a ≠ 0) *mua = mus * (1.0 − r.a)/r.a;
      else *mua = 0.0;
    }
    else {
      *musp = r.a * r.b/m.slab_thickness * (1.0 − r.g);
      *mua = (1 − r.a) * r.b/m.slab_thickness;
    }
    }
```

This code is used in section 2.

14. ⟨ print error legend 14 ⟩ ≡

```
  static void print_error_legend(void)
  {
    fprintf(stderr, "-----------------␣Sorry,␣but␣...␣mistakes␣were␣made␣---------------\n");
    fprintf(stderr, "␣␣0-9␣==>␣Monte␣Carlo␣Iteration\n");
    fprintf(stderr, "␣␣␣*␣␣==>␣Successful␣Calculation\n");
    fprintf(stderr, "-------------------------------------------------------------------\n");
    fprintf(stderr, "␣␣␣R␣␣==>␣Bad␣Reflection\n");
    fprintf(stderr, "␣␣␣T␣␣==>␣Bad␣Transmission\n");
    fprintf(stderr, "␣␣␣U␣␣==>␣Bad␣Unscattered␣Transmission\n");
    fprintf(stderr, "␣␣␣L␣␣==>␣Reflection␣+␣Transmission␣+␣␣Monte␣Carlo␣Light␣Losses␣>␣1\n");
    fprintf(stderr, "␣␣␣!␣␣==>␣Reflection␣+␣Transmission␣+␣Unscattered␣>␣1\n");
    fprintf(stderr, "␣␣␣+␣␣==>␣Too␣many␣adding-doubling␣iterations\n\n");
  }
```

This code is used in section 2.

15. returns a new string consisting of s+t

```
⟨ stringdup together function 15 ⟩ ≡
  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.

16. assume that start time has already been set

```
⟨ seconds elapsed function 16 ⟩ ≡
  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.

17. 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 17 ⟩ ≡
  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.

18. ⟨ print dot function 18 ⟩ ≡

```
static void print_dot(clock_t start_time, int err, int count, int points, int final, int *any_error)
{
  static int counter = 1;
  if (err ≡ IAD_NO_ERROR) {
    if (final ≡ 99) fprintf(stderr, "*");
    else fprintf(stderr, "%d", (int) fmod(final, 10));
  }
  else {
    *any_error = 1;
    if (err ≡ IAD_TOO_MANY_ITERATIONS) fprintf(stderr, "+");
    else if (err ≡ IAD_R_LT_ZERO ∨ err ≡ IAD_R_GT_ONE ∨ err ≡ IAD_RT_GT_ONE ∨ err ≡
        IAD_RT_LT_MINIMUM ∨ err ≡ IAD_RD_LT_ZERO) fprintf(stderr, "R");
    else if (err ≡ IAD_T_LT_ZERO ∨ err ≡ IAD_T_GT_ONE ∨ err ≡ IAD_RD_IS_ZERO_BUT_NOT_TD)
      fprintf(stderr, "T");
    else if (err ≡ IAD_TU_LT_ZERO ∨ err ≡ IAD_TU_GT_ONE) fprintf(stderr, "U");
    else if (err ≡ IAD_R_PLUS_T_GT_ONE ∨ err ≡ IAD_RT_PLUS_TT_GT_ONE ∨ err ≡
        IAD_R_PLUS_T_PLUS_TU_GT_ONE) fprintf(stderr, "!");
    else if (err ≡ IAD_EXCESSIVE_LIGHT_LOSS) fprintf(stderr, "L");
    else {
      if (err ≥ IAD_AS_NOT_VALID ∧ err ≤ IAD_TOO_MANY_LAYERS)
        fprintf(stderr, "\nBad␣sphere␣parameter␣(error=%d)\n", err);
      else if (err ≡ IAD_MEMORY_ERROR) fprintf(stderr, "\nMemory␣error\n");
      else if (err ≡ IAD_FILE_ERROR) fprintf(stderr, "\nFile␣error\n");
      exit(1);
    }
  }
  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, "␣");
  counter++;
  fflush(stderr);      /* Needed for windows */
}
```

This code is used in section 2.

19. 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

20. 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` 20 ⟩ ≡
```
#undef FALSE
#undef TRUE
```
⟨ Preprocessor definitions ⟩
⟨ Structs to export from IAD Types 23 ⟩

21.
```
#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 RELATIVE 0
#define ABSOLUTE 1
#define COLLIMATED 0
#define DIFFUSE 1
#define FALSE 0
#define TRUE 1
#define IAD_MAX_ITERATIONS 5000
```

22. Need error codes for this silly program

```
#define IAD_NO_ERROR 0
#define IAD_TOO_MANY_ITERATIONS 1
#define IAD_R_LT_ZERO 2
#define IAD_R_GT_ONE 3
#define IAD_T_LT_ZERO 4
#define IAD_T_GT_ONE 5
#define IAD_TU_LT_ZERO 6
#define IAD_TU_GT_ONE 7
#define IAD_R_PLUS_T_GT_ONE 8
#define IAD_RD_IS_ZERO_BUT_NOT_TD 9
#define IAD_TD_LT_ZERO 10
#define IAD_RD_LT_ZERO 11
#define IAD_RT_GT_ONE 12
#define IAD_TT_GT_ONE 13
#define IAD_RT_PLUS_TT_GT_ONE 14
#define IAD_R_PLUS_T_PLUS_TU_GT_ONE 15
#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 UNINITIALIZED -99
```

23. 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 23 ⟩ ≡
  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;
    int num_spheres;
    int num_measures;
    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 24 and 25.

This code is used in section 20.

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

```
⟨ Structs to export from IAD Types 23 ⟩ +≡
  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; } IAD_invert_type;
```

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

```
⟨ Structs to export from IAD Types 23 ⟩ +≡
  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;
```

26. 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` 26 ⟩ ≡
```
#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_find.h"
#include "iad_pub.h"
#include "iad_io.h"
#include "mc_lost.h"
#define DEBUG_SEARCH 0
```
⟨ Definition for *Inverse_RT* 30 ⟩
⟨ Definition for *measure_OK* 34 ⟩
⟨ Definition for *determine_search* 45 ⟩
⟨ Definition for *Initialize_Result* 49 ⟩
⟨ Definition for *Initialize_Measure* 57 ⟩
⟨ Definition for *ez_Inverse_RT* 55 ⟩
⟨ Definition for *Spheres_Inverse_RT* 59 ⟩

27. 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` 27 ⟩ ≡
⟨ Prototype for *Inverse_RT* 29 ⟩;
⟨ Prototype for *measure_OK* 33 ⟩;
⟨ Prototype for *determine_search* 44 ⟩;
⟨ Prototype for *Initialize_Result* 48 ⟩;
⟨ Prototype for *ez_Inverse_RT* 54 ⟩;
⟨ Prototype for *Initialize_Measure* 56 ⟩;

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

⟨ `lib_iad.h` 28 ⟩ ≡
⟨ Prototype for *ez_Inverse_RT* 54 ⟩;
⟨ Prototype for *Spheres_Inverse_RT* 58 ⟩;

29. 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.

history: 6/8/94 changed the way the program writes error stuff. Use stderr uniformly throughout.

⟨ Prototype for *Inverse_RT* 29 ⟩ ≡

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

This code is used in sections 27 and 30.

30. ⟨ Definition for *Inverse_RT* 30 ⟩ ≡

```
⟨ Prototype for Inverse_RT 29 ⟩{ r→found = FALSE;
    ⟨ Exit with bad input data 31 ⟩
    r→search = determine_search(m, *r);
    ⟨ Find the optical properties 32 ⟩
    if ( r → error ≡ IAD_NO_ERROR ∧ r→final_distance ≤ r→tolerance ) r→found = TRUE; }
```

This code is used in section 26.

31. 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 31 ⟩ ≡

```
r → error = measure_OK(m, *r); if (r→method.quad_pts < 4) r → error = IAD_QUAD_PTS_NOT_VALID; if
    ( r → error ≠ IAD_NO_ERROR ) return;
```

This code is used in section 30.

32. 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 32 ⟩ ≡

```
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 30.

33. Validation. Now the question is — just what is bad data. Well, we first begin with a prototype for the function. It is important that *r.slab* be set correctly before calling this routine.

⟨ Prototype for *measure_OK* 33 ⟩ ≡
```
int measure_OK(struct measure_type m, struct invert_type r)
```
This code is used in sections 27 and 34.

34. ⟨ Definition for *measure_OK* 34 ⟩ ≡
```
⟨ Prototype for measure_OK 33 ⟩{ double rt, tt, rd, rc, td, tc; int error = IAD_NO_ERROR;
    ⟨ Test easy cases 35 ⟩
    ⟨ Calculate specular limits for reflection and transmission 38 ⟩
    ⟨ Check specular limits 39 ⟩
    ⟨ Check sphere parameters 42 ⟩
    return error ; }
```
This code is used in section 26.

35. The easy cases consist of making sure that the reflection and transmission and their sum is between zero and one — without worrying overmuch about the specular reflection. We begin by making sure that the reflection is valid.

Note that I just fake the case for corrections to R+T¿1 when spheres are present ... it is possible for R+T¿1 by a bit in this case!

⟨ Test easy cases 35 ⟩ ≡
```
if (m.m_r < 0) error = IAD_R_LT_ZERO; if (m.m_r > 1) error = IAD_R_GT_ONE;
```
See also sections 36 and 37.

This code is used in section 34.

36. We only should check the transmission if it is present. This means that the number of measurements must be 2 or 3. We can also make a quick check to ensure that energy conservation is fulfilled.

⟨ Test easy cases 35 ⟩ +≡
```
if (m.num_measures > 1) { if (m.m_t < 0) error = IAD_T_LT_ZERO; if (m.m_t > 1) error =
    IAD_T_GT_ONE; if (m.m_t + m.m_r > 1) error = IAD_R_PLUS_T_GT_ONE; }
```

37. Finally the unscattered transmission value should be checked if it is present. If the specular reflection is not included in the transmission value, then another check on energy conservation may be made.

⟨ Test easy cases 35 ⟩ +≡
```
if (m.num_measures ≡ 3) { if (m.m_u > 1) error = IAD_TU_GT_ONE; if (m.m_u < 0) error
    = IAD_TU_LT_ZERO; if (¬m.sphere_with_tc ∧ m.m_r + m.m_t + m.m_u > 1.0) error =
    IAD_R_PLUS_T_PLUS_TU_GT_ONE; }
```

38. Now we make estimates for the specular reflection and transmission so that the diffuse reflection and transmission for the sample can be guessed.

⟨ Calculate specular limits for reflection and transmission 38 ⟩ ≡
```
Estimate_RT(m, r.slab, &rt, &tt, &rd, &rc, &td, &tc);
```
This code is used in section 34.

39. The only cases that remain are those values that have been possibly altered by the addition or subtraction of the unscattered reflection or transmission. These are the diffuse reflection (it could have become negative), the total reflection (it could have become greater than one), the diffuse transmission (it could have become negative), the total transmission (it could have become greater than one) and the sum of the total transmission and total reflection. These are more stringent tests than those above, but are very useful for skipping bogus data points.

```
⟨ Check specular limits 39 ⟩ ≡
  if (rd < 0) error = IAD_RD_LT_ZERO; if (rt > 1.0) error = IAD_RT_GT_ONE; if (td < 0) error
      = IAD_TD_LT_ZERO; if (tt > 1.0) error = IAD_TT_GT_ONE; if (tt + rt > 1.0) error =
      IAD_RT_PLUS_TT_GT_ONE;
```

See also sections 40 and 41.

This code is used in section 34.

40. Another specular limit has arisen. Namely the non-scattering case. This only can be checked if there are two or three measurements.

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 $b = mu\_a * d$ ofthe slab. The minimum reflection is obtained by including all the specular reflectances from all the surfaces.

```
⟨ Check specular limits 39 ⟩ +≡
  if (m.num_measures > 1) { double rmin, tmin, b;
  if (m.m_u ≡ 0) b = What_Is_B(r.slab, m.m_t);
  else b = What_Is_B(r.slab, m.m_u);
  Sp_mu_RT(r.slab.n_top_slide, r.slab.n_slab, r.slab.n_bottom_slide,
      r.slab.b_top_slide, b, r.slab.b_bottom_slide, 1.0, &rmin, &tmin); if (m.m_r + 0.0001 < rmin) { error
      = IAD_RT_LT_MINIMUM;
  fprintf(stderr, "Not␣enough␣reflected␣light␣for␣this␣measurement\n");
  fprintf(stderr, "your␣␣␣␣reflectance␣=␣%7.5f␣␣transmittance␣=␣%7.5f␣\n", m.m_r, m.m_t);
  fprintf(stderr, "minimum␣reflectance␣=␣%7.5f\n", rmin); } }
```

41. There is one more point that needs to be checked. If the diffuse reflection is zero, then the diffuse transmission must also be zero (because the albedo must be zero). Note that the converse is not true since there may just be insufficient light to detect on the far side of the sample.

I needed to disable this check for when searching only for the scattering or the absorption coefficient alone.

```
⟨ Check specular limits 39 ⟩ +≡
  if (rd ≡ 0∧td ≠ 0) if (r.search ≠ FIND_Ba∧r.search ≠ FIND_Bs) error = IAD_RD_IS_ZERO_BUT_NOT_TD;
```

42. Make sure that reflection sphere parameters are reasonable

```
⟨ Check sphere parameters 42 ⟩ ≡
  if (m.num_spheres ≠ 0) { if (m.as_r < 0 ∨ m.as_r ≥ 0.2) error = IAD_AS_NOT_VALID; if
      (m.ad_r < 0 ∨ m.ad_r ≥ 0.2) error = IAD_AD_NOT_VALID; if (m.ae_r < 0 ∨ m.ae_r ≥ 0.2)
      error = IAD_AE_NOT_VALID; if (m.rw_r < 0 ∨ m.rw_r > 1.0) error = IAD_RW_NOT_VALID; if
      (m.rd_r < 0 ∨ m.rd_r > 1.0) error = IAD_RD_NOT_VALID; if (m.rstd_r < 0 ∨ m.rstd_r > 1.0) error
      = IAD_RSTD_NOT_VALID; if (m.f_r < 0 ∨ m.f_r > 1) error = IAD_F_NOT_VALID; }
```

See also section 43.

This code is used in section 34.

43. Make sure that transmission sphere parameters are reasonable

```
⟨ Check sphere parameters 42 ⟩ +≡
  if (m.num_spheres ≠ 0) { if (m.as_t < 0 ∨ m.as_t ≥ 0.2) error = IAD_AS_NOT_VALID; if
      (m.ad_t < 0 ∨ m.ad_t ≥ 0.2) error = IAD_AD_NOT_VALID; if (m.ae_t < 0 ∨ m.ae_t ≥ 0.2)
      error = IAD_AE_NOT_VALID; if (m.rw_t < 0 ∨ m.rw_r > 1.0) error = IAD_RW_NOT_VALID; if
      (m.rd_t < 0 ∨ m.rd_t > 1.0) error = IAD_RD_NOT_VALID; if (m.rstd_t < 0 ∨ m.rstd_t > 1.0) error
      = IAD_RSTD_NOT_VALID; if (m.f_t < 0 ∨ m.f_t > 1) error = IAD_F_NOT_VALID; }
```

44. 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* 44 ⟩ ≡

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

This code is used in sections 27 and 45.

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

```
⟨ Definition for determine_search 45 ⟩ ≡
  ⟨ Prototype for determine_search 44 ⟩
  {
    double rt, tt, rd, td, tc, rc;
    int search = 0;
    int independent = m.num_measures;
    Estimate_RT(m, r.slab, &rt, &tt, &rd, &rc, &td, &tc);
    if (tc ≡ 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 46 ⟩
    }
    else if (independent ≡ 2) {
      ⟨ Two parameter search 47 ⟩
    }     /* three real parameters with information! */
    else {
      search = FIND_AG;
    }
    if (DEBUG_SEARCH) {
      fprintf(stderr, "FIND_A␣␣␣␣=␣0\n");
      fprintf(stderr, "FIND_B␣␣␣␣=␣1\n");
      fprintf(stderr, "FIND_AB␣␣␣=␣2\n");
      fprintf(stderr, "FIND_AG␣␣␣=␣3\n");
      fprintf(stderr, "FIND_AUTO␣=␣4\n");
      fprintf(stderr, "FIND_BG␣␣␣=␣5\n");
      fprintf(stderr, "FIND_BaG␣␣=␣6\n");
      fprintf(stderr, "FIND_BsG␣␣=␣7\n");
      fprintf(stderr, "FIND_Ba␣␣␣=␣8\n");
      fprintf(stderr, "FIND_Bs␣␣␣=␣9\n");
      fprintf(stderr, "FIND_G␣␣␣␣=10\n");
      fprintf(stderr, "search␣␣␣␣=%2d\n", search);
    }
    return search;
  }
```

This code is used in section 26.

46. 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 diffuse reflectance. If this is zero, then we solve for the optical thickness, otherwise the sample is assumed infinitely thick and the albedo is found.

```
⟨ One parameter search 46 ⟩ ≡
  if (r.default_a ≠ UNINITIALIZED) {
    if (td ≡ 0) search = FIND_G;
    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 (rd ≠ 0 ∨ td ≡ 0) search = FIND_A;
  else search = FIND_B;
```

This code is used in section 45.

47. 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 47 ⟩ ≡
  if (r.default_a ≠ UNINITIALIZED) search = FIND_BG;
  else if (r.default_b ≠ UNINITIALIZED) search = FIND_AG;
  else if (r.default_ba ≠ UNINITIALIZED) search = FIND_BsG;
  else if (r.default_bs ≠ UNINITIALIZED) search = FIND_BaG;
  else search = FIND_AB;
```

This code is used in section 45.

48. 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 48 ⟩ ≡
  void Initialize_Result(struct measure_type m, struct invert_type *r)
```

This code is used in sections 27 and 49.

49.

```
⟨ Definition for Initialize_Result 49 ⟩ ≡
  ⟨ Prototype for Initialize_Result 48 ⟩
  {
    ⟨ Fill r with reasonable values 50 ⟩
  }
```

This code is used in section 26.

50. Start with the optical properties.

```
⟨ Fill r with reasonable values 50 ⟩ ≡
  r→a = 0.0;
  r→b = 0.0;
  r→g = 0.0;
```

See also sections 51, 52, and 53.

This code is used in section 49.

51. Continue with other useful stuff.

```
⟨ Fill r with reasonable values 50 ⟩ +≡
  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;
```

52. The defaults might be handy

```
⟨ Fill r with reasonable values 50 ⟩ +≡
  r→default_a = UNINITIALIZED;
  r→default_b = UNINITIALIZED;
  r→default_g = UNINITIALIZED;
  r→default_ba = UNINITIALIZED;
  r→default_bs = UNINITIALIZED;
```

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

```
⟨ Fill r with reasonable values 50 ⟩ +≡
  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→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;
```

54. 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 54 ⟩ ≡
  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 27, 28, and 55.

55.

```
⟨ Definition for ez_Inverse_RT 55 ⟩ ≡
  ⟨ Prototype for ez_Inverse_RT 54 ⟩{ 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.num_measures = 3;
      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 26.

56.

```
⟨ Prototype for Initialize_Measure 56 ⟩ ≡
  void Initialize_Measure(struct measure_type *m)
```

This code is used in sections 27 and 57.

57. ⟨ Definition for *Initialize_Measure* 57 ⟩ ≡

```
⟨ Prototype for Initialize_Measure 56 ⟩
{
  double default_sphere_d = 8.0 * 25.4;
  double default_sample_d = 0.5 * 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→num_spheres = 0;
  m→num_measures = 1;
  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 26.

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

```
⟨ Prototype for Spheres_Inverse_RT 58 ⟩ ≡
  void Spheres_Inverse_RT(double *setup, double *analysis, double *sphere_r, double *sphere_t, double
      *measurements, double *results)
```

This code is used in sections 28 and 59.

59.

```
⟨ Definition for Spheres_Inverse_RT 59 ⟩ ≡
  ⟨ Prototype for Spheres_Inverse_RT 58 ⟩{ 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 60 ⟩
      ⟨ handle reflection sphere 63 ⟩
      ⟨ handle transmission sphere 64 ⟩
      ⟨ handle measurement 62 ⟩
      Initialize_Result(m, &r);
      results[0] = 0;
      results[1] = 0;
      results[2] = 0;
      ⟨ handle analysis 61 ⟩
      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 26.

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

```
⟨ handle setup 60 ⟩ ≡
  {
    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;
  }
```

This code is used in section 59.

61. ⟨ handle analysis 61 ⟩ ≡

```
  r.method.quad_pts = (int) analysis[0];
  mc_runs = (int) analysis[1];
```

This code is used in section 59.

62.

```
⟨ handle measurement 62 ⟩ ≡
  m.m_r = measurements[0];
  m.m_t = measurements[1];
  m.m_u = measurements[2];
  m.num_measures = 3;
  if (m.m_t ≡ 0) m.num_measures--;
  if (m.m_u ≡ 0) m.num_measures--;
```

This code is used in section 59.

63.

⟨ handle reflection sphere 63 ⟩ ≡

```
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 59.

64.

⟨ handle transmission sphere 64 ⟩ ≡

```
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 59.

65. IAD Input Output.

The special define below is to get Visual C to suppress silly warnings.

```
⟨ iad_io.c 65 ⟩ ≡
#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 75 ⟩
  ⟨ Definition for read_number 77 ⟩
  ⟨ Definition for check_magic 79 ⟩
  ⟨ Definition for Read_Header 69 ⟩
  ⟨ Definition for Write_Header 81 ⟩
  ⟨ Definition for Read_Data_Line 73 ⟩
```

66. ⟨ iad_io.h 66 ⟩ ≡

```
  ⟨ Prototype for Read_Header 68 ⟩;
  ⟨ Prototype for Write_Header 80 ⟩;
  ⟨ Prototype for Read_Data_Line 72 ⟩;
```

67. Reading the file header.

68. ⟨ Prototype for *Read_Header* 68 ⟩ ≡

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

This code is used in sections 66 and 69.

69. 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 69 ⟩ ≡
  ⟨ Prototype for Read_Header 68 ⟩
  {
    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;
    ⟨ Read coefficients for reflection sphere 70 ⟩
    ⟨ Read coefficients for transmission sphere 71 ⟩
    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 65.

70. ⟨ Read coefficients for reflection sphere 70 ⟩ ≡

```
{
  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 69.

71. ⟨ Read coefficients for transmission sphere 71 ⟩ ≡

```
{
  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 69.

72. 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 72 ⟩ ≡
  int Read_Data_Line(FILE *fp, struct measure_type *m, int params)
```

This code is used in sections 66 and 73.

73.

```
⟨ Definition for Read_Data_Line 73 ⟩ ≡
  ⟨ Prototype for Read_Data_Line 72 ⟩
  {
    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→rstd_r)) return 1;
    return 0;
  }
```

This code is used in section 65.

74. 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 74 ⟩ ≡
  int skip_white(FILE *fp)
```

This code is used in section 75.

75.

```
⟨ Definition for skip_white 75 ⟩ ≡
  ⟨ Prototype for skip_white 74 ⟩
  {
    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 65.

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

```
⟨ Prototype for read_number 76 ⟩ ≡
  int read_number(FILE *fp, double *x)
```

This code is used in section 77.

77.

```
⟨ Definition for read_number 77 ⟩ ≡
  ⟨ Prototype for read_number 76 ⟩
  {
    if (skip_white(fp)) return 1;
    if (fscanf(fp, "%lf", x)) return 0;
    else return 1;
  }
```

This code is used in section 65.

78. 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 78 ⟩ ≡
  int check_magic(FILE *fp)
```

This code is used in section 79.

79.

```
⟨ Definition for check_magic 79 ⟩ ≡
  ⟨ Prototype for check_magic 78 ⟩
  {
    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 65.

80. Formatting the header information.

⟨ Prototype for *Write_Header* 80 ⟩ ≡
void *Write_Header*(**struct measure_type** *m*, **struct invert_type** *r*, **int** *params*)
This code is used in sections 66 and 81.

81. ⟨ Definition for *Write_Header* 81 ⟩ ≡
⟨ Prototype for *Write_Header* 80 ⟩
{
⟨ Write slab info 82 ⟩
⟨ Write irradiation info 83 ⟩
⟨ Write general sphere info 84 ⟩
⟨ Write first sphere info 85 ⟩
⟨ Write second sphere info 86 ⟩
⟨ Write measure and inversion info 87 ⟩
}
This code is used in section 65.

82. ⟨ Write slab info 82 ⟩ ≡
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 81.

83. ⟨ Write irradiation info 83 ⟩ ≡
printf("# \n");
This code is used in section 81.

84. ⟨ Write general sphere info 84 ⟩ ≡
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 81.

85. ⟨ Write first sphere info 85 ⟩ ≡
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 81.

86. ⟨ Write second sphere info 86 ⟩ ≡

```
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 81.

87. ⟨ Write measure and inversion info 87 ⟩ ≡

```
printf("#\n");
switch (params) {
case 1: printf("# Just M_R was measured.\n");
  break;
case 2: printf("# M_R and M_T were measured.\n");
  break;
case 3: printf("# M_R, M_T, and M_U were measured.\n");
  break;
case 4: printf("# M_R, M_T, M_U, and r_w were measured.\n");
  break;
case 5: printf("# M_R, M_T, M_U, r_w, and r_std were measured.\n");
  break;
default: printf("# Something went wrong ... measures should be 1 to 5!\n");
  break;
}
switch (m.num_spheres) {
case 0: printf("# No special corrections for integrating spheres 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");
  break;
case FIND_AG: printf("# The inverse routine varied the albedo and anisotropy\n");
  break;
case FIND_AUTO: printf("# The inverse routine adapted to the input data\n");
  break;
case FIND_A: printf("# The inverse routine varied only the albedo\n");
  break;
case FIND_B: printf("# The inverse routine varied only the optical depth\n");
  break;
case FIND_Ba: printf("# The inverse routine varied only the absorption\n");
  break;
case FIND_Bs: printf("# The inverse routine varied only the scattering\n");
  break;
}
printf("#\n");
switch (r.search) {
case FIND_AB: xx = 0;
  if (r.default_g ≠ UNINITIALIZED) xx = r.default_g;
  printf("# Default single scattering anisotropy = %7.3f\n", xx);
  break;
case FIND_AG: xx = 1;
  if (r.default_b ≠ UNINITIALIZED) xx = r.default_b;
  printf("#                    Default (mu_t*d) = %7.3g\n", xx);
  break;
case FIND_AUTO: printf("#\n");
  break;
```

```
case FIND_A: xx = 0;
  if (r.default_g ≠ UNINITIALIZED) xx = r.default_g;
  printf("# Default single scattering anisotropy = %7.3f\n", xx);
  xx = HUGE_VAL;
  if (r.default_b ≠ UNINITIALIZED) xx = r.default_b;
  printf("#                                   &(mu_t*d) = %7.3g\n", xx);
  break;
case FIND_B: printf("#\n");
  break;
case FIND_Ba: printf("#\n");
  break;
case FIND_Bs: 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 81.

88. IAD Calculation.

⟨ iad_calc.c 88 ⟩ ≡
```
#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 41
  static int CALCULATING_GRID = 1;
  static int DEBUG_ITERATION = 0;
  static int DEBUG_GRID = 0;
  static int DEBUG_LOST_LIGHT = 0;
  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;
  ⟨ Definition for Set_Grid_Debugging 92 ⟩
  ⟨ Definition for Set_Calc_State 106 ⟩
  ⟨ Definition for Get_Calc_State 108 ⟩
  ⟨ Prototype for Fill_AB_Grid 124 ⟩;
  ⟨ Prototype for Fill_AG_Grid 128 ⟩;
  ⟨ Definition for Allocate_Grid 110 ⟩
  ⟨ Definition for Valid_Grid 114 ⟩
  ⟨ Definition for fill_grid_entry 123 ⟩
  ⟨ Definition for Fill_Grid 138 ⟩
  ⟨ Definition for Near_Grid_Points 122 ⟩
  ⟨ Definition for Fill_AB_Grid 125 ⟩
```

⟨ Definition for *Fill_AG_Grid* 129 ⟩
⟨ Definition for *Fill_BG_Grid* 132 ⟩
⟨ Definition for *Fill_BaG_Grid* 134 ⟩
⟨ Definition for *Fill_BsG_Grid* 136 ⟩
⟨ Definition for *Grid_ABG* 112 ⟩
⟨ Definition for *Gain* 95 ⟩
⟨ Definition for *Gain_11* 97 ⟩
⟨ Definition for *Gain_22* 99 ⟩
⟨ Definition for *Two_Sphere_R* 101 ⟩
⟨ Definition for *Two_Sphere_T* 103 ⟩
⟨ Definition for *Calculate_Distance_With_Corrections* 144 ⟩
⟨ Definition for *Calculate_Grid_Distance* 142 ⟩
⟨ Definition for *Calculate_Distance* 140 ⟩
⟨ Definition for *abg_distance* 120 ⟩
⟨ Definition for *Find_AG_fn* 151 ⟩
⟨ Definition for *Find_AB_fn* 153 ⟩
⟨ Definition for *Find_Ba_fn* 155 ⟩
⟨ Definition for *Find_Bs_fn* 157 ⟩
⟨ Definition for *Find_A_fn* 159 ⟩
⟨ Definition for *Find_B_fn* 161 ⟩
⟨ Definition for *Find_G_fn* 163 ⟩
⟨ Definition for *Find_BG_fn* 165 ⟩
⟨ Definition for *Find_BaG_fn* 167 ⟩
⟨ Definition for *Find_BsG_fn* 169 ⟩
⟨ Definition for *maxloss* 171 ⟩
⟨ Definition for *Max_Light_Loss* 173 ⟩

89.

⟨ `iad_calc.h` 89 ⟩ ≡
⟨ Prototype for *Set_Grid_Debugging* 91 ⟩;
⟨ Prototype for *Gain* 94 ⟩;
⟨ Prototype for *Gain_11* 96 ⟩;
⟨ Prototype for *Gain_22* 98 ⟩;
⟨ Prototype for *Two_Sphere_R* 100 ⟩;
⟨ Prototype for *Two_Sphere_T* 102 ⟩;
⟨ Prototype for *Set_Calc_State* 105 ⟩;
⟨ Prototype for *Get_Calc_State* 107 ⟩;
⟨ Prototype for *Valid_Grid* 113 ⟩;
⟨ Prototype for *Allocate_Grid* 109 ⟩;
⟨ Prototype for *Fill_Grid* 137 ⟩;
⟨ Prototype for *Near_Grid_Points* 121 ⟩;
⟨ Prototype for *Grid_ABG* 111 ⟩;
⟨ Prototype for *Find_AG_fn* 150 ⟩;
⟨ Prototype for *Find_AB_fn* 152 ⟩;
⟨ Prototype for *Find_Ba_fn* 154 ⟩;
⟨ Prototype for *Find_Bs_fn* 156 ⟩;
⟨ Prototype for *Find_A_fn* 158 ⟩;
⟨ Prototype for *Find_B_fn* 160 ⟩;
⟨ Prototype for *Find_G_fn* 162 ⟩;
⟨ Prototype for *Find_BG_fn* 164 ⟩;
⟨ Prototype for *Find_BsG_fn* 168 ⟩;
⟨ Prototype for *Find_BaG_fn* 166 ⟩;
⟨ Prototype for *Fill_BG_Grid* 131 ⟩;
⟨ Prototype for *Fill_BsG_Grid* 135 ⟩;
⟨ Prototype for *Fill_BaG_Grid* 133 ⟩;
⟨ Prototype for *Calculate_Distance_With_Corrections* 143 ⟩;
⟨ Prototype for *Calculate_Distance* 139 ⟩;
⟨ Prototype for *Calculate_Grid_Distance* 141 ⟩;
⟨ Prototype for *abg_distance* 119 ⟩;
⟨ Prototype for *maxloss* 170 ⟩;
⟨ Prototype for *Max_Light_Loss* 172 ⟩;

90. 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*.

91. Some debugging stuff

```
⟨ Prototype for Set_Grid_Debugging 91 ⟩ ≡
  void Set_Grid_Debugging(unsigned int debug_level)
```

This code is used in sections 89 and 92.

92.

```
⟨ Definition for Set_Grid_Debugging 92 ⟩ ≡
  ⟨ Prototype for Set_Grid_Debugging 91 ⟩
  {
    unsigned int iteration_mask = 1;
    unsigned int grid_mask = 2;
    unsigned int lost_light_mask = 4;
    if (debug_level & iteration_mask) DEBUG_ITERATION = 1;
    if (debug_level & grid_mask) DEBUG_GRID = 1;
    if (debug_level & lost_light_mask) DEBUG_LOST_LIGHT = 1;
  }
```

This code is used in section 88.

93. 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 $r_d = 0$.

94. 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

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

⟨ Prototype for *Gain* 94 ⟩ ≡
double *Gain*(**int** *sphere*, **struct measure_type** *m*, **double** URU)
This code is used in sections 89 and 95.

95. ⟨ Definition for *Gain* 95 ⟩ ≡
```
⟨ Prototype for Gain 94 ⟩
{
  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/m.aw_r/(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/m.aw_t/(1.0 − tmp);
  }
  return G;
}
```
This code is used in section 88.

96. 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* 96 ⟩ ≡
double *Gain_11*(**struct measure_type** *m*, **double** URU, **double** *tdiffuse*)
This code is used in sections 89 and 97.

97. ⟨ Definition for *Gain_11* 97 ⟩ ≡
⟨ Prototype for *Gain_11* 96 ⟩
{
double G, GP, G11;
$G = Gain(\texttt{REFLECTION\_SPHERE}, m, \texttt{URU})$;
$\texttt{GP} = Gain(\texttt{TRANSMISSION\_SPHERE}, m, \texttt{URU})$;
$\texttt{G11} = G/(1 - m.as\_r * m.as\_t * m.aw\_r * m.aw\_t * (1 - m.ae\_r) * (1 - m.ae\_t) * G * \texttt{GP} * tdiffuse * tdiffuse)$;
return G11;
}
This code is used in section 88.

98. 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* 98 ⟩ ≡
double *Gain_22*(**struct measure_type** m, **double** URU, **double** *tdiffuse*)
This code is used in sections 89 and 99.

99. ⟨ Definition for *Gain_22* 99 ⟩ ≡
⟨ Prototype for *Gain_22* 98 ⟩
{
double G, GP, G22;
$G = Gain(\texttt{REFLECTION\_SPHERE}, m, \texttt{URU})$;
$\texttt{GP} = Gain(\texttt{TRANSMISSION\_SPHERE}, m, \texttt{URU})$;
$\texttt{G22} = \texttt{GP}/(1 - m.as\_r * m.as\_t * m.aw\_r * m.aw\_t * (1 - m.ae\_r) * (1 - m.ae\_t) * G * \texttt{GP} * tdiffuse * tdiffuse)$;
return G22;
}
This code is used in section 88.

100. The reflected power for two spheres is 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* 100 ⟩ ≡
double *Two_Sphere_R*(**struct measure_type** m, **double** UR1, **double** UT1, **double** URU, **double** UTU)
This code is used in sections 89 and 101.

101. ⟨ Definition for *Two_Sphere_R* 101 ⟩ ≡

```
⟨ Prototype for Two_Sphere_R 100 ⟩
{
  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 88.

102. 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* 102 ⟩ ≡

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

This code is used in sections 89 and 103.

103. ⟨ Definition for *Two_Sphere_T* 103 ⟩ ≡

```
⟨ Prototype for Two_Sphere_T 102 ⟩
{
  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 88.

104. 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.

105. 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* 105 ⟩ ≡
void *Set_Calc_State*(**struct measure_type** m, **struct invert_type** r)

This code is used in sections 89 and 106.

106. ⟨ Definition for *Set_Calc_State* 106 ⟩ ≡

```
⟨ Prototype for Set_Calc_State 105 ⟩
{
  memcpy(&MM, &m, sizeof(struct measure_type));
  memcpy(&RR, &r, sizeof(struct invert_type));
  if (DEBUG_ITERATION ∧ ¬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 88.

107. 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* 107 ⟩ ≡
void *Get_Calc_State*(**struct measure_type** $*m$, **struct invert_type** $*r$)

This code is used in sections 89 and 108.

108. ⟨ Definition for *Get_Calc_State* 108 ⟩ ≡

```
⟨ Prototype for Get_Calc_State 107 ⟩
{
  memcpy(m, &MM, sizeof(struct measure_type));
  memcpy(r, &RR, sizeof(struct invert_type));
}
```

This code is used in section 88.

109. ⟨ Prototype for *Allocate_Grid* 109 ⟩ ≡
void *Allocate_Grid*(**search_type** s)

This code is used in sections 89 and 110.

110. ⟨ Definition for *Allocate_Grid* 110 ⟩ ≡

```
⟨ Prototype for Allocate_Grid 109 ⟩
{
  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 88.

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

```
⟨ Prototype for Grid_ABG 111 ⟩ ≡
  void Grid_ABG(int i, int j, guess_type *guess)
```

This code is used in sections 89 and 112.

112. ⟨ Definition for *Grid_ABG* 112 ⟩ ≡

```
⟨ Prototype for Grid_ABG 111 ⟩
{
  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 88.

113. 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 113 ⟩ ≡
  boolean_type Valid_Grid(struct measure_type m, search_type s)
```

This code is used in sections 89 and 114.

114. ⟨ Definition for *Valid_Grid* 114 ⟩ ≡

```
⟨ Prototype for Valid_Grid 113 ⟩
{
  ⟨ Tests for invalid grid 115 ⟩
  return (TRUE);
}
```

This code is used in section 88.

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

```
⟨ Tests for invalid grid 115 ⟩ ≡
  if (The_Grid ≡ Λ) return (FALSE);
  if (¬The_Grid_Initialized) return (FALSE);
```

See also sections 116, 117, and 118.

This code is used in section 114.

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

```
⟨ Tests for invalid grid 115 ⟩ +≡
  if (The_Grid_Search ≠ s) return (FALSE);
```

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

```
⟨ Tests for invalid grid 115 ⟩ +≡
  if ((m.num_measures ≡ 3) ∧ (m.m_u ≠ MGRID.m_u)) return (FALSE);
```

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

```
⟨ Tests for invalid grid 115 ⟩ +≡
  if (m.slab_index ≠ MGRID.slab_index) 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);
```

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

```
⟨ Prototype for abg_distance 119 ⟩ ≡
  void abg_distance(double a, double b, double g, guess_type *guess)
```

This code is used in sections 89 and 120.

120.

```
⟨ Definition for abg_distance 120 ⟩ ≡
  ⟨ Prototype for abg_distance 119 ⟩
  {
    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 88.

121. 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* 121 ⟩ ≡

```
void Near_Grid_Points(double r, double t, search_type s, int *i_min, int *j_min)
```

This code is used in sections 89 and 122.

122. ⟨ Definition for *Near_Grid_Points* 122 ⟩ ≡

```
⟨ Prototype for Near_Grid_Points 121 ⟩
{
  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 88.

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

```
⟨ Definition for fill_grid_entry 123 ⟩ ≡
  static void fill_grid_entry(int i, int j)
  {
    double ur1, ut1, uru, utu;
    RT(RR.method.quad_pts, &RR.slab, &ur1, &ut1, &uru, &utu);
    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_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 88.

124. 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 124 ⟩ ≡
  void Fill_AB_Grid(struct measure_type m, struct invert_type r)
```

This code is used in sections 88 and 125.

125. ⟨ Definition for *Fill_AB_Grid* 125 ⟩ ≡

```
⟨ Prototype for Fill_AB_Grid 124 ⟩
{
  int i, j;
  double a;
  double min_b = −12;      /* exp(-10) is smallest thickness */
  double max_b = +8;       /* exp(+8) is greatest thickness */
  if (DEBUG_GRID) fprintf(stderr, "Filling␣AB␣grid\n");
  if (The_Grid ≡ Λ) Allocate_Grid(r.search);
  ⟨ Zero GG 130 ⟩
  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 127 ⟩
      fill_grid_entry(i, j);
    }
  }
  The_Grid_Initialized = TRUE;
  The_Grid_Search = FIND_AB;
}
```

This code is used in section 88.

126. 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 126 ⟩ ≡

```
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;
}
```

127. 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 127 ⟩ ≡
  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;
```

This code is used in sections 125 and 129.

128. 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 128 ⟩ ≡
  void Fill_AG_Grid(struct measure_type m, struct invert_type r)
```

This code is used in sections 88 and 129.

129.

```
⟨ Definition for Fill_AG_Grid 129 ⟩ ≡
  ⟨ Prototype for Fill_AG_Grid 128 ⟩
  {
    int i, j;
    double a;
    if (DEBUG_GRID) fprintf(stderr, "Filling␣AG␣grid\n");
    if (The_Grid ≡ Λ) Allocate_Grid(r.search);
    ⟨ Zero GG 130 ⟩
    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 127 ⟩
        fill_grid_entry(i, j);
      }
    }
    The_Grid_Initialized = TRUE;
    The_Grid_Search = FIND_AG;
  }
```

This code is used in section 88.

130.

```
⟨ Zero GG 130 ⟩ ≡
  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 125, 129, 132, 134, and 136.

131. 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* 131 ⟩ ≡
```
void Fill_BG_Grid(struct measure_type m, struct invert_type r)
```
This code is used in sections 89 and 132.

132. ⟨ Definition for *Fill_BG_Grid* 132 ⟩ ≡
```
⟨ Prototype for Fill_BG_Grid 131 ⟩
{
  int i, j;
  if (The_Grid ≡ Λ) Allocate_Grid(r.search);
  ⟨ Zero GG 130 ⟩
  if (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 88.

133. 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* 133 ⟩ ≡
```
void Fill_BaG_Grid(struct measure_type m, struct invert_type r)
```
This code is used in sections 89 and 134.

134. ⟨ Definition for *Fill_BaG_Grid* 134 ⟩ ≡

```
⟨ Prototype for Fill_BaG_Grid 133 ⟩
{
  int i, j;
  double bs, ba;
  if (The_Grid ≡ Λ) Allocate_Grid(r.search);
  ⟨ Zero GG 130 ⟩
  if (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 88.

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

```
⟨ Prototype for Fill_BsG_Grid 135 ⟩ ≡
  void Fill_BsG_Grid(struct measure_type m, struct invert_type r)
```

This code is used in sections 89 and 136.

136. ⟨ Definition for *Fill_BsG_Grid* 136 ⟩ ≡

```
⟨ Prototype for Fill_BsG_Grid 135 ⟩
{
  int i, j;
  double bs, ba;
  if (The_Grid ≡ Λ) Allocate_Grid(r.search);
  ⟨ Zero GG 130 ⟩
  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 88.

137. ⟨ Prototype for *Fill_Grid* 137 ⟩ ≡

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

This code is used in sections 89 and 138.

138. ⟨ Definition for *Fill_Grid* 138 ⟩ ≡

```
⟨ Prototype for Fill_Grid 137 ⟩
{
  switch (r.search) {
  case FIND_AB: Fill_AB_Grid(m, r);
    break;
  case FIND_AG: Fill_AG_Grid(m, r);
    break;
  case FIND_BG: Fill_BG_Grid(m, r);
    break;
  case FIND_BaG: Fill_BaG_Grid(m, r);
    break;
  case FIND_BsG: 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 88.

139. 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. has been Calc appropriately.

⟨ Prototype for *Calculate_Distance* 139 ⟩ ≡
void *Calculate_Distance*(**double** ∗`LR`, **double** ∗`LT`, **double** ∗*deviation*)

This code is used in sections 89 and 140.

140. ⟨ Definition for *Calculate_Distance* 140 ⟩ ≡
⟨ Prototype for *Calculate_Distance* 139 ⟩
{
double *Rc*, *Tc*, *ur1*, *ut1*, *uru*, *utu*;
`RT`(`RR`.*method*.*quad_pts*, &`RR`.*slab*, &*ur1*, &*ut1*, &*uru*, &*utu*);
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*, 1.0, &*Rc*, &*Tc*);
if ((`DEBUG_ITERATION` ∧ ¬`CALCULATING_GRID`) ∨ (`CALCULATING_GRID` ∧ `DEBUG_GRID`))
fprintf(*stderr*, "␣␣␣␣␣␣␣␣");
Calculate_Distance_With_Corrections(*ur1*, *ut1*, *Rc*, *Tc*, *uru*, *utu*, `LR`, `LT`, *deviation*);
}

This code is used in section 88.

141. ⟨ Prototype for *Calculate_Grid_Distance* 141 ⟩ ≡
double *Calculate_Grid_Distance*(**int** *i*, **int** *j*)

This code is used in sections 89 and 142.

142. ⟨ Definition for *Calculate_Grid_Distance* 142 ⟩ ≡
⟨ Prototype for *Calculate_Grid_Distance* 141 ⟩
{
double *ur1*, *ut1*, *uru*, *utu*, *Rc*, *Tc*, *b*, *dev*, `LR`, `LT`;
if (`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*, 1.0, &*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 88.

143. 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 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 143 ⟩ ≡
  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 89 and 144.

144.
```
⟨ Definition for Calculate_Distance_With_Corrections 144 ⟩ ≡
  ⟨ Prototype for Calculate_Distance_With_Corrections 143 ⟩
  {
    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;
    switch (MM.num_spheres) {
    case 0: ⟨ Calc M_R and M_T for no spheres 145 ⟩
      break;
    case 1: case −2: ⟨ Calc M_R and M_T for one sphere 146 ⟩
      break;
    case 2: ⟨ Calc M_R and M_T for two spheres 147 ⟩
      break;
    }
    ⟨ Print diagnostics 149 ⟩
    ⟨ Return the deviation 148 ⟩
  }
```

This code is used in section 88.

145. 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 145 ⟩ ≡
  *M_R = R_direct;
  *M_T = T_direct;
```

This code is used in section 144.

146. 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 one sphere 146 ⟩ ≡

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

This code is used in section 144.

147. When two integrating spheres are present then the double integrating sphere formulas are that much more complicated.

The normalized sphere measurements are then

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

and

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

⟨ Calc M_R and M_T for two spheres 147 ⟩ ≡

```
{
  double P_0 = Gain(REFLECTION_SPHERE, MM, 0.0) * MM.f_r * MM.rw_r;
  *M_R = MM.rstd_r * (Two_Sphere_R(MM, R_direct, T_direct, R_diffuse,
      T_diffuse) - P_0)/(Two_Sphere_R(MM, MM.rstd_r, 0, MM.rstd_r, 0) - P_0);
  *M_T = Two_Sphere_T(MM, R_direct, T_direct, R_diffuse, T_diffuse)/Two_Sphere_T(MM, 0, 1, 0, 1);
}
```

This code is used in section 144.

148. The only question at this point is whether we should use a relative or an absolute metric. At this point I do not really know. This should probably be a subroutine so that the logic does not get replicated in the next chunk of code also.

⟨ Return the deviation 148 ⟩ ≡

```
if (RR.metric ≡ RELATIVE)
  *dev = fabs(MM.m_r - *M_R)/(MM.m_r + ABIT) + fabs(MM.m_t - *M_T)/(MM.m_t + ABIT);
else *dev = (fabs(MM.m_r - *M_R) + fabs(MM.m_t - *M_T));
```

This code is used in section 144.

149. 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 149 ⟩ ≡

```
if ((DEBUG_ITERATION ∧ ¬CALCULATING_GRID) ∨ (CALCULATING_GRID ∧ DEBUG_GRID)) {
  static int once = 0;
  if (once ≡ 0) {
    fprintf(stderr, "%10s␣%10s␣%10s␣|%10s␣%10s␣|%10s␣%10s␣|%10s\n", "a", "b", "g", "m_r", "calc",
        "m_t", "calc", "delta");
    once = 1;
    fprintf(stderr, "␣␣␣␣␣␣␣␣");
  }
  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);
  if (RR.metric ≡ RELATIVE) fprintf(stderr,
      "%10.5f␣\n", 39fabs(MM.m_r - *M_R)/(MM.m_r + ABIT) + fabs(MM.m_t - *M_T)/(MM.m_t + ABIT));
  else fprintf(stderr, "%10.5f␣\n", fabs(MM.m_r - *M_R) + fabs(MM.m_t - *M_T));
}
```

This code is used in section 144.

150. ⟨ Prototype for *Find_AG_fn* 150 ⟩ ≡

```
double Find_AG_fn(double x[])
```

This code is used in sections 89 and 151.

151. ⟨ Definition for *Find_AG_fn* 151 ⟩ ≡
```
⟨ Prototype for Find_AG_fn 150 ⟩
{
  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 88.

152. ⟨ Prototype for *Find_AB_fn* 152 ⟩ ≡
```
double Find_AB_fn(double x[])
```
This code is used in sections 89 and 153.

153. ⟨ Definition for *Find_AB_fn* 153 ⟩ ≡
```
⟨ Prototype for Find_AB_fn 152 ⟩
{
  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 88.

154. ⟨ Prototype for *Find_Ba_fn* 154 ⟩ ≡
```
double Find_Ba_fn(double x)
```
This code is used in sections 89 and 155.

155. 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* 155 ⟩ ≡
```
⟨ Prototype for Find_Ba_fn 154 ⟩
{
  double m_r, m_t, deviation, ba, bs;
  bs = RR.slab.b;      /* swindle */
  ba = bcalc2b(x);
  RR.slab.b = ba + bs;
  RR.slab.a = bs/(ba + bs);
  Calculate_Distance(&m_r, &m_t, &deviation);
  deviation = fabs(MM.m_t - m_t);
  if (RR.metric ≡ RELATIVE) deviation /= (MM.m_t + ABIT);
  RR.slab.b = bs;      /* unswindle */
  return deviation;
}
```
This code is used in section 88.

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

```
⟨ Prototype for Find_Bs_fn 156 ⟩ ≡
  double Find_Bs_fn(double x)
```

This code is used in sections 89 and 157.

157.

```
⟨ Definition for Find_Bs_fn 157 ⟩ ≡
  ⟨ Prototype for Find_Bs_fn 156 ⟩
  {
    double m_r, m_t, deviation, ba, bs;
    ba = RR.slab.b;      /* swindle */
    bs = bcalc2b(x);
    RR.slab.b = ba + bs;
    RR.slab.a = bs/(ba + bs);
    Calculate_Distance(&m_r, &m_t, &deviation);
    deviation = fabs(MM.m_t - m_t);
    if (RR.metric ≡ RELATIVE) deviation /= (MM.m_t + ABIT);
    RR.slab.b = ba;      /* unswindle */
    return deviation;
  }
```

This code is used in section 88.

158.

```
⟨ Prototype for Find_A_fn 158 ⟩ ≡
  double Find_A_fn(double x)
```

This code is used in sections 89 and 159.

159.

```
⟨ Definition for Find_A_fn 159 ⟩ ≡
  ⟨ Prototype for Find_A_fn 158 ⟩
  {
    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 88.

160.

```
⟨ Prototype for Find_B_fn 160 ⟩ ≡
  double Find_B_fn(double x)
```

This code is used in sections 89 and 161.

161.

```
⟨ Definition for Find_B_fn 161 ⟩ ≡
  ⟨ Prototype for Find_B_fn 160 ⟩
  {
    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 88.

162. ⟨ Prototype for *Find_G_fn* 162 ⟩ ≡
double *Find_G_fn*(**double** *x*)
This code is used in sections 89 and 163.

163. ⟨ Definition for *Find_G_fn* 163 ⟩ ≡
⟨ Prototype for *Find_G_fn* 162 ⟩
{
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 88.

164. ⟨ Prototype for *Find_BG_fn* 164 ⟩ ≡
double *Find_BG_fn*(**double** *x*[])
This code is used in sections 89 and 165.

165. ⟨ Definition for *Find_BG_fn* 165 ⟩ ≡
⟨ Prototype for *Find_BG_fn* 164 ⟩
{
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 88.

166. 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* 166 ⟩ ≡
double *Find_BaG_fn*(**double** *x*[])
This code is used in sections 89 and 167.

167. ⟨ Definition for *Find_BaG_fn* 167 ⟩ ≡
⟨ Prototype for *Find_BaG_fn* 166 ⟩
{
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 88.

168. ⟨ Prototype for *Find_BsG_fn* 168 ⟩ ≡
double *Find_BsG_fn*(**double** *x*[ ])
This code is used in sections 89 and 169.

169. ⟨ Definition for *Find_BsG_fn* 169 ⟩ ≡
⟨ Prototype for *Find_BsG_fn* 168 ⟩
{
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 88.

170. 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* 170 ⟩ ≡
double *maxloss*(**double** *f*)
This code is used in sections 89 and 171.

171. ⟨ Definition for *maxloss* 171 ⟩ ≡
⟨ Prototype for *maxloss* 170 ⟩
{
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 88.

172. 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* 172 ⟩ ≡
void *Max_Light_Loss*(**struct measure_type** *m*, **struct invert_type** *r*, **double** ∗*ur1_loss*, **double** ∗*ut1_loss*)
This code is used in sections 89 and 173.

173. ⟨ Definition for *Max_Light_Loss* 173 ⟩ ≡

```
⟨ Prototype for Max_Light_Loss 172 ⟩
{
  struct measure_type m_old;
  struct invert_type r_old;
  *ur1_loss = m.ur1_lost;
  *ut1_loss = m.ut1_lost;
  if (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_LOST_LIGHT)
    fprintf(stderr, "lost␣after␣␣ur1=%7.5f,␣ut1=%7.5f\n", *ur1_loss, *ut1_loss);
}
```

This code is used in section 88.

174. this is currently unused

```
⟨ Unused diffusion fragment 174 ⟩ ≡
  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;
  }
```

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

```
⟨ iad_find.c 175 ⟩ ≡
#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 DEBUG_BEST_GUESS 0
#define NUMBER_OF_GUESSES 11
  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 189 ⟩
  ⟨ Definition for U_Find_Bs 187 ⟩
  ⟨ Definition for U_Find_A 191 ⟩
  ⟨ Definition for U_Find_B 195 ⟩
  ⟨ Definition for U_Find_G 193 ⟩
  ⟨ Definition for U_Find_AG 198 ⟩
  ⟨ Definition for U_Find_AB 178 ⟩
  ⟨ Definition for U_Find_BG 203 ⟩
  ⟨ Definition for U_Find_BaG 209 ⟩
  ⟨ Definition for U_Find_BsG 213 ⟩
```

176. 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 176 ⟩ ≡
  ⟨ Prototype for U_Find_Ba 188 ⟩;
  ⟨ Prototype for U_Find_Bs 186 ⟩;
  ⟨ Prototype for U_Find_A 190 ⟩;
  ⟨ Prototype for U_Find_B 194 ⟩;
  ⟨ Prototype for U_Find_G 192 ⟩;
  ⟨ Prototype for U_Find_AG 197 ⟩;
  ⟨ Prototype for U_Find_AB 177 ⟩;
  ⟨ Prototype for U_Find_BG 202 ⟩;
  ⟨ Prototype for U_Find_BaG 208 ⟩;
  ⟨ Prototype for U_Find_BsG 212 ⟩;
```

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

```
⟨ Prototype for U_Find_AB 177 ⟩ ≡
  void U_Find_AB(struct measure_type m, struct invert_type *r)
```

This code is used in sections 176 and 178.

178.
```
⟨ Definition for U_Find_AB 178 ⟩ ≡
  ⟨ Prototype for U_Find_AB 177 ⟩
  {
    ⟨ Allocate local simplex variables 179 ⟩
    r→slab.g = (r→default_g ≡ UNINITIALIZED) ? 0 : r→default_g;
    Set_Calc_State(m, *r);
    ⟨ Get the initial a, b, and g 180 ⟩
    ⟨ Initialize the nodes of the a and b simplex 181 ⟩
    ⟨ Evaluate the a and b simplex at the nodes 182 ⟩
    amoeba(p, y, 2, r→tolerance, Find_AB_fn, &r→iterations);
    ⟨ Choose the best node of the a and b simplex 183 ⟩
    ⟨ Free simplex data structures 185 ⟩
    ⟨ Put final values in result 184 ⟩
  }
```

This code is used in section 175.

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

```
⟨ Allocate local simplex variables 179 ⟩ ≡
  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 178, 198, 203, 209, and 213.

180. 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 180 ⟩ ≡
  {
    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]));      /* distance to heuristic guess */
    quick_guess(m, *r, &a3, &b3, &g3);
    abg_distance(a3, b3, g3, &(guess[10]));
    qsort((void *) guess, count, sizeof(guess_type), compare_guesses);
    if (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 178, 198, 203, 209, and 213.

181. ⟨ Initialize the nodes of the a and b simplex 181 ⟩ ≡

```
{
  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_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 178.

182. ⟨ Evaluate the a and b simplex at the nodes 182 ⟩ ≡

```
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 178.

183. ⟨ Choose the best node of the a and b simplex 183 ⟩ ≡

```
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 178.

184. ⟨ Put final values in result 184 ⟩ ≡

```
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 178, 187, 189, 191, 193, 195, 198, 203, 209, and 213.

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

⟨ Free simplex data structures 185 ⟩ ≡

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

This code is used in sections 178, 198, 203, 209, and 213.

186. 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* 186 ⟩ ≡
```
void U_Find_Bs(struct measure_type m, struct invert_type *r)
```
This code is used in sections 176 and 187.

187. ⟨ Definition for *U_Find_Bs* 187 ⟩ ≡
```
⟨ Prototype for U_Find_Bs 186 ⟩
{
  double ax, bx, cx, fa, fb, fc, bs;
  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);
  ax = b2bcalc(1.0);
  bx = b2bcalc(10.0);
  mnbrak(&ax, &bx, &cx, &fa, &fb, &fc, Find_Bs_fn);
  r→final_distance = brent(ax, bx, cx, Find_Bs_fn, r→tolerance, &bs);
  r→slab.b = bcalc2b(bs) + r→slab.b;      /* actual value of b */
  ⟨ Put final values in result 184 ⟩
}
```
This code is used in section 175.

188. 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 188 ⟩ ≡
  void U_Find_Ba(struct measure_type m, struct invert_type *r)
```

This code is used in sections 176 and 189.

189.

```
⟨ Definition for U_Find_Ba 189 ⟩ ≡
  ⟨ Prototype for U_Find_Ba 188 ⟩
  {
    double ax, bx, cx, fa, fb, fc, ba;
    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);
    ax = b2bcalc(0.1);
    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);
    r→slab.b = bcalc2b(ba) + r→slab.b;      /* actual value of b */
    ⟨ Put final values in result 184 ⟩
  }
```

This code is used in section 175.

190. 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 190 ⟩ ≡
  void U_Find_A(struct measure_type m, struct invert_type *r)
```

This code is used in sections 176 and 191.

191.

```
⟨ Definition for U_Find_A 191 ⟩ ≡
  ⟨ Prototype for U_Find_A 190 ⟩
  {
    double Rt, Tt, Rd, Rc, Td, Tc;
    Estimate_RT(m, r→slab, &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 (Rd > 0.0) {
      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 184 ⟩
  }
```

This code is used in section 175.

192. Fixed Optical Depth and Albedo.

```
⟨ Prototype for U_Find_G 192 ⟩ ≡
  void U_Find_G(struct measure_type m, struct invert_type *r)
```

This code is used in sections 176 and 193.

193.

```
⟨ Definition for U_Find_G 193 ⟩ ≡
  ⟨ Prototype for U_Find_G 192 ⟩
  {
    double Rt, Tt, Rd, Rc, Td, Tc;
    Estimate_RT(m, r→slab, &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);
    }
    ⟨ Put final values in result 184 ⟩
  }
```

This code is used in section 175.

194. 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 194 ⟩ ≡
  void U_Find_B(struct measure_type m, struct invert_type *r)
```

This code is used in sections 176 and 195.

195.

```
⟨ Definition for U_Find_B 195 ⟩ ≡
  ⟨ Prototype for U_Find_B 194 ⟩
  {
    double Rt, Tt, Rd, Rc, Td, Tc;
    Estimate_RT(m, r→slab, &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 = HUGE_VAL;
    r→final_distance = 0.0;      /* case when albedo is zero */
    if (Rd ≡ 0.0) {
      r→slab.b = What_Is_B(r→slab, Tc);
      Set_Calc_State(m, *r);
    }      /* case when albedo non-zero */
    else if (Tt ≠ 0.0) {
      Set_Calc_State(m, *r);
      ⟨ Iteratively solve for b 196 ⟩
    }
    ⟨ Put final values in result 184 ⟩
  }
```

This code is used in section 175.

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

```
⟨ Iteratively solve for b 196 ⟩ ≡
  {
    double x, ax, bx, cx, fa, fb, fc;
    ax = b2bcalc(1);
    bx = b2bcalc(2);
    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);
  }
```

This code is used in section 195.

197. 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 197 ⟩ ≡
  void U_Find_AG(struct measure_type m, struct invert_type *r)
```

This code is used in sections 176 and 198.

198.

```
⟨ Definition for U_Find_AG 198 ⟩ ≡
  ⟨ Prototype for U_Find_AG 197 ⟩
  {
    ⟨ Allocate local simplex variables 179 ⟩
    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 180 ⟩
    ⟨ Initialize the nodes of the a and g simplex 199 ⟩
    ⟨ Evaluate the a and g simplex at the nodes 200 ⟩
    amoeba(p, y, 2, r→tolerance, Find_AG_fn, &r→iterations);
    ⟨ Choose the best node of the a and g simplex 201 ⟩
    ⟨ Free simplex data structures 185 ⟩
    ⟨ Put final values in result 184 ⟩
  }
```

This code is used in section 175.

199. ⟨ Initialize the nodes of the a and g simplex 199 ⟩ ≡

```
{
  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_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 198.

200. ⟨ Evaluate the a and g simplex at the nodes 200 ⟩ ≡

```
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 198.

201. 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 201 ⟩ ≡
  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 198.

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

```
⟨ Prototype for U_Find_BG 202 ⟩ ≡
  void U_Find_BG(struct measure_type m, struct invert_type *r)
```

This code is used in sections 176 and 203.

203.

```
⟨ Definition for U_Find_BG 203 ⟩ ≡
  ⟨ Prototype for U_Find_BG 202 ⟩
  {
    ⟨ Allocate local simplex variables 179 ⟩
    r→slab.a = (r→default_a ≡ UNINITIALIZED) ? 0 : r→default_a;
    Set_Calc_State(m, *r);
    ⟨ Get the initial a, b, and g 180 ⟩
    ⟨ Initialize the nodes of the b and g simplex 205 ⟩
    ⟨ Evaluate the bg simplex at the nodes 206 ⟩
    amoeba(p, y, 2, r→tolerance, Find_BG_fn, &r→iterations);
    ⟨ Choose the best node of the b and g simplex 207 ⟩
    ⟨ Free simplex data structures 185 ⟩
    ⟨ Put final values in result 184 ⟩
  }
```

This code is used in section 175.

204. 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.

205. ⟨ Initialize the nodes of the b and g simplex 205 ⟩ ≡

```
{
  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_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 203.

206. ⟨ Evaluate the bg simplex at the nodes 206 ⟩ ≡

```
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 203.

207. 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 207 ⟩ ≡
  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 sections 203, 209, and 213.

208. 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 208 ⟩ ≡
  void U_Find_BaG(struct measure_type m, struct invert_type *r)
```

This code is used in sections 176 and 209.

209.
```
⟨ Definition for U_Find_BaG 209 ⟩ ≡
  ⟨ Prototype for U_Find_BaG 208 ⟩
  {
    ⟨ Allocate local simplex variables 179 ⟩
    Set_Calc_State(m, *r);
    ⟨ Get the initial a, b, and g 180 ⟩
    ⟨ Initialize the nodes of the ba and g simplex 210 ⟩
    ⟨ Evaluate the BaG simplex at the nodes 211 ⟩
    amoeba(p, y, 2, r→tolerance, Find_BaG_fn, &r→iterations);
    ⟨ Choose the best node of the b and g simplex 207 ⟩
    r→b += r→default_bs;
    r→a = (r→default_bs)/(r→b);
    ⟨ Free simplex data structures 185 ⟩
    ⟨ Put final values in result 184 ⟩
  }
```

This code is used in section 175.

210.
```
⟨ Initialize the nodes of the ba and g simplex 210 ⟩ ≡
  p[1][1] = b2bcalc(guess[0].b − r→default_bs);
  p[1][2] = g2gcalc(guess[0].g);
  p[2][1] = b2bcalc(2 * guess[0].b − 2 * r→default_bs);
  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 209.

211.
```
⟨ Evaluate the BaG simplex at the nodes 211 ⟩ ≡
  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 209.

212. 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 212 ⟩ ≡
  void U_Find_BsG(struct measure_type m, struct invert_type *r)
```

This code is used in sections 176 and 213.

213.
```
⟨ Definition for U_Find_BsG 213 ⟩ ≡
  ⟨ Prototype for U_Find_BsG 212 ⟩
  {
    ⟨ Allocate local simplex variables 179 ⟩
    Set_Calc_State(m, *r);
    ⟨ Get the initial a, b, and g 180 ⟩
    ⟨ Initialize the nodes of the bs and g simplex 214 ⟩
    ⟨ Evaluate the BsG simplex at the nodes 215 ⟩
    amoeba(p, y, 2, r→tolerance, Find_BsG_fn, &r→iterations);
    ⟨ Choose the best node of the b and g simplex 207 ⟩
    r→b += r→default_ba;
    r→a = 1 − (r→default_ba)/(r→b);
    ⟨ Free simplex data structures 185 ⟩
    ⟨ Put final values in result 184 ⟩
  }
```

This code is used in section 175.

214.
```
⟨ Initialize the nodes of the bs and g simplex 214 ⟩ ≡
  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 213.

215.
```
⟨ Evaluate the BsG simplex at the nodes 215 ⟩ ≡
  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 213.

216. IAD Utilities.

March 1995. Reincluded *quick_guess* code.

⟨ `iad_util.c` 216 ⟩ ≡
```
#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_util.h"
```
⟨ Preprocessor definitions ⟩
⟨ Definition for *What_Is_B* 219 ⟩
⟨ Definition for *Estimate_RT* 225 ⟩
⟨ Definition for *a2acalc* 231 ⟩
⟨ Definition for *acalc2a* 233 ⟩
⟨ Definition for *g2gcalc* 235 ⟩
⟨ Definition for *gcalc2g* 237 ⟩
⟨ Definition for *b2bcalc* 239 ⟩
⟨ Definition for *bcalc2b* 241 ⟩
⟨ Definition for *twoprime* 243 ⟩
⟨ Definition for *twounprime* 245 ⟩
⟨ Definition for *abgg2ab* 247 ⟩
⟨ Definition for *abgb2ag* 249 ⟩
⟨ Definition for *quick_guess* 256 ⟩

217. ⟨ `iad_util.h` 217 ⟩ ≡
⟨ Prototype for *What_Is_B* 218 ⟩;
⟨ Prototype for *Estimate_RT* 224 ⟩;
⟨ Prototype for *a2acalc* 230 ⟩;
⟨ Prototype for *acalc2a* 232 ⟩;
⟨ Prototype for *g2gcalc* 234 ⟩;
⟨ Prototype for *gcalc2g* 236 ⟩;
⟨ Prototype for *b2bcalc* 238 ⟩;
⟨ Prototype for *bcalc2b* 240 ⟩;
⟨ Prototype for *twoprime* 242 ⟩;
⟨ Prototype for *twounprime* 244 ⟩;
⟨ Prototype for *abgg2ab* 246 ⟩;
⟨ Prototype for *abgb2ag* 248 ⟩;
⟨ Prototype for *quick_guess* 255 ⟩;

218. 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* 218 ⟩ ≡

```
double What_Is_B(struct AD_slab_type slab, double Tc)
```

This code is used in sections 217 and 219.

219. ⟨ Definition for *What_Is_B* 219 ⟩ ≡

```
⟨ Prototype for What_Is_B 218 ⟩
{
  double r1, r2, t1, t2;
  ⟨ Calculate specular reflection and transmission 220 ⟩
  ⟨ Check for bad values of Tc 221 ⟩
  ⟨ Solve if multiple internal reflections are not present 222 ⟩
  ⟨ Find thickness when multiple internal reflections are present 223 ⟩
}
```

This code is used in section 216.

220. 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 220 ⟩ ≡

```
Absorbing_Glass_RT(1.0, slab.n_top_slide, slab.n_slab, 1.0, slab.b_top_slide, &r1, &t1);
Absorbing_Glass_RT(slab.n_slab, slab.n_bottom_slide, 1.0, 1.0, slab.b_bottom_slide, &r2, &t2);
```

This code is used in section 219.

221. 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* 221 ⟩ ≡

```
if (Tc ≤ 0) return (HUGE_VAL);
if (Tc ≥ t1 * t2/(1 - r1 * r2)) return (0.0);
```

This code is used in section 219.

222. 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

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

where b is the optical thickness. Clearly,

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

⟨ Solve if multiple internal reflections are not present 222 ⟩ ≡
  if (*r1* $\equiv 0 \vee$ *r2* $\equiv 0$) **return** (−*log*(*Tc*/*t1*/*t2*));
This code is used in section 219.

223. 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 223 ⟩ ≡

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

This code is used in section 219.

224. 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* 224 ⟩ ≡

```
void Estimate_RT(struct measure_type m, struct AD_slab_type s, double *rt, double *tt, double
    *rd, double *rc, double *td, double *tc)
```

This code is used in sections 217 and 225.

225. ⟨ Definition for *Estimate_RT* 225 ⟩ ≡

```
⟨ Prototype for Estimate_RT 224 ⟩
{
  double r, t;
  ⟨ Calculate the unscattered transmission and reflection 226 ⟩
  ⟨ Estimate the backscattered reflection 227 ⟩
  ⟨ Estimate the scattered transmission 228 ⟩
}
```

This code is used in section 216.

226. 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.

⟨ Calculate the unscattered transmission and reflection 226 ⟩ ≡

```
if (m.num_measures ≤ 2) {
  Absorbing_Glass_RT(1.0, s.n_top_slide, s.n_slab, 1.0, s.b_top_slide, &r, &t);
  *rc = r;
  *tc = 0.0;
}
else {
  double b;
  b = What_Is_B(s, m.m_u);
  Sp_mu_RT(s.n_top_slide, s.n_slab, s.n_bottom_slide, s.b_top_slide, b, s.b_bottom_slide, 1.0, rc, tc);
}
```

This code is used in section 225.

227. 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 227 ⟩ ≡
  if (m.sphere_with_rc) {
    *rt = m.m_r;
    *rd = *rt - *rc;
  }
  else {
    *rd = m.m_r;
    *rt = *rd + *rc;
  }
```

This code is used in section 225.

228. 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 228 ⟩ ≡
  if (m.num_measures ≡ 1) {
    *tt = 0.0;
    *td = 0.0;
  }
  else if (m.sphere_with_tc) {
    *tt = m.m_t;
    *td = *tt - *tc;
  }
  else {
    *td = m.m_t;
    *tt = *td + *tc;
  }
```

This code is used in section 225.

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

230. *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 230 ⟩ ≡
  double a2acalc(double a)
```

This code is used in sections 217 and 231.

231.
```
⟨ Definition for a2acalc 231 ⟩ ≡
  ⟨ Prototype for a2acalc 230 ⟩
  {
    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 216.

232. *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 232 ⟩ ≡
  double acalc2a(double acalc)
```

This code is used in sections 217 and 233.

233.
```
⟨ Definition for acalc2a 233 ⟩ ≡
  ⟨ Prototype for acalc2a 232 ⟩
  {
    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 216.

234. *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* 234 ⟩ ≡
```
double g2gcalc(double g)
```
This code is used in sections 217 and 235.

235. ⟨ Definition for *g2gcalc* 235 ⟩ ≡
```
⟨ Prototype for g2gcalc 234 ⟩
{
  if (g ≤ −1) return (−HUGE_VAL);
  if (g ≥ 1) return (HUGE_VAL);
  return (g/(1 − fabs(g)));
}
```
This code is used in section 216.

236. *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* 236 ⟩ ≡
```
double gcalc2g(double gcalc)
```
This code is used in sections 217 and 237.

237. ⟨ Definition for *gcalc2g* 237 ⟩ ≡
```
⟨ Prototype for gcalc2g 236 ⟩
{
  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 216.

238. *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* 238 ⟩ ≡
```
double b2bcalc(double b)
```
This code is used in sections 217 and 239.

239. ⟨ Definition for *b2bcalc* 239 ⟩ ≡

```
⟨ Prototype for b2bcalc 238 ⟩
{
  if (b ≡ HUGE_VAL) return HUGE_VAL;
  if (b ≤ 0) return 0.0;
  return (log(b));
}
```

This code is used in section 216.

240. *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* 240 ⟩ ≡

```
double bcalc2b(double bcalc)
```

This code is used in sections 217 and 241.

241. ⟨ Definition for *bcalc2b* 241 ⟩ ≡

```
⟨ Prototype for bcalc2b 240 ⟩
{
  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 216.

242. *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* 242 ⟩ ≡

```
void twoprime(double a, double b, double g, double *ap, double *bp)
```

This code is used in sections 217 and 243.

243. ⟨ Definition for *twoprime* 243 ⟩ ≡

```
⟨ Prototype for twoprime 242 ⟩
{
  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 216.

244. *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* 244 ⟩ ≡
void *twounprime*(**double** *ap*, **double** *bp*, **double** *g*, **double** ∗*a*, **double** ∗*b*)
This code is used in sections 217 and 245.

245. ⟨ Definition for *twounprime* 245 ⟩ ≡
⟨ Prototype for *twounprime* 244 ⟩
{
∗*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 216.

246. *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* 246 ⟩ ≡
void *abgg2ab*(**double** *a1*, **double** *b1*, **double** *g1*, **double** *g2*, **double** ∗*a2*, **double** ∗*b2*)
This code is used in sections 217 and 247.

247. ⟨ Definition for *abgg2ab* 247 ⟩ ≡
⟨ Prototype for *abgg2ab* 246 ⟩
{
double *a*, *b*;
twoprime(*a1*, *b1*, *g1*, &*a*, &*b*);
twounprime(*a*, *b*, *g2*, *a2*, *b2*);
}
This code is used in section 216.

248. *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* 248 ⟩ ≡
void *abgb2ag*(**double** *a1*, **double** *b1*, **double** *b2*, **double** ∗*a2*, **double** ∗*g2*)
This code is used in sections 217 and 249.

249. ⟨ Definition for *abgb2ag* 249 ⟩ ≡

```
⟨ Prototype for abgb2ag 248 ⟩
{
  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 216.

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

```
⟨ Prototype for slow_guess 250 ⟩ ≡
  void slow_guess(struct measure_type m, struct invert_type *r, double *a, double *b, double *g)
```

This code is used in section 251.

251.
```
⟨ Definition for slow_guess 251 ⟩ ≡
  ⟨ Prototype for slow_guess 250 ⟩
  {
    double fmin = 10.0;
    double fval;
    double *x;
    x = dvector(1, 2);
    switch (r→search) {
    case FIND_A: ⟨ Slow guess for a alone 252 ⟩
      break;
    case FIND_B: ⟨ Slow guess for b alone 253 ⟩
      break;
    case FIND_AB: case FIND_AG: ⟨ Slow guess for a and b or a and g 254 ⟩
      break;
    }
    *a = r→slab.a;
    *b = r→slab.b;
    *g = r→slab.g;
    free_dvector(x, 1, 2);
  }
```

252.
```
⟨ Slow guess for a alone 252 ⟩ ≡
  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 251.

253. 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 253 ⟩ ≡
  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 251.

254.

```
⟨ Slow guess for a and b or a and g 254 ⟩ ≡
  {
    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 251.

255.

```
⟨ Prototype for quick_guess 255 ⟩ ≡
  void quick_guess(struct measure_type m, struct invert_type r, double *a, double *b, double *g)
```

This code is used in sections 217 and 256.

256.

```
⟨ Definition for quick_guess 256 ⟩ ≡
  ⟨ Prototype for quick_guess 255 ⟩
  {
    double UR1, UT1, rd, td, tc, rc, bprime, aprime, alpha, beta, logr;
    Estimate_RT(m, r.slab, &UR1, &UT1, &rd, &rc, &td, &tc);
    ⟨ Estimate aprime 257 ⟩
    switch (m.num_measures) {
    case 1: ⟨ Guess when only reflection is known 259 ⟩
      break;
    case 2: ⟨ Guess when reflection and transmission are known 260 ⟩
      break;
    case 3: ⟨ Guess when all three measurements are known 261 ⟩
      break;
    }
    ⟨ Clean up guesses 266 ⟩
  }
```

This code is used in section 216.

257. ⟨ Estimate *aprime* 257 ⟩ ≡

```
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 256.

258. ⟨ Estimate *bprime* 258 ⟩ ≡

```
if (rd < 0.01) bprime = What_Is_B(r.slab, UT1);
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 260, 264, and 265.

259.

⟨ Guess when only reflection is known 259 ⟩ ≡

```
∗g = r.default_g;
∗a = aprime/(1 − ∗g + aprime ∗ (∗g));
∗b = HUGE_VAL;
```

This code is used in section 256.

260. ⟨ Guess when reflection and transmission are known 260 ⟩ ≡

```
⟨ Estimate bprime 258 ⟩
∗g = r.default_g;
∗a = aprime/(1 − ∗g + aprime ∗ ∗g);
∗b = bprime/(1 − ∗a ∗ ∗g);
```

This code is used in section 256.

261. ⟨ Guess when all three measurements are known 261 ⟩ ≡
```
switch (r.search) {
case FIND_A: ⟨ Guess when finding albedo 262 ⟩
  break;
case FIND_B: ⟨ Guess when finding optical depth 263 ⟩
  break;
case FIND_AB: ⟨ Guess when finding the albedo and optical depth 264 ⟩
  break;
case FIND_AG: ⟨ Guess when finding anisotropy and albedo 265 ⟩
  break;
}
```
This code is used in section 256.

262.

⟨ Guess when finding albedo 262 ⟩ ≡
```
*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 261.

263.

⟨ Guess when finding optical depth 263 ⟩ ≡
```
*g = r.default_g;
*a = 0.0;
*b = What_Is_B(r.slab, m.m_u);
```
This code is used in section 261.

264.

⟨ Guess when finding the albedo and optical depth 264 ⟩ ≡
```
*g = r.default_g;
if (*g ≡ 1) *a = 0.0;
else *a = aprime/(1 - *g + aprime * *g);
⟨ Estimate bprime 258 ⟩
if (bprime ≡ HUGE_VAL ∨ *a * *g ≡ 1) *b = HUGE_VAL;
else *b = bprime/(1 - *a * *g);
```
This code is used in section 261.

265.

⟨ Guess when finding anisotropy and albedo 265 ⟩ ≡
```
*b = What_Is_B(r.slab, m.m_u);
if (*b ≡ HUGE_VAL ∨ *b ≡ 0) {
  *a = aprime;
  *g = r.default_g;
}
else {
  ⟨ Estimate bprime 258 ⟩
  *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 261.

266.

⟨ Clean up guesses 266 ⟩ ≡

```
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 256.

267. 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 179 ⟩ Used in sections 178, 198, 203, 209, and 213.
⟨ Calc M_R and M_T for no spheres 145 ⟩ Used in section 144.
⟨ Calc M_R and M_T for one sphere 146 ⟩ Used in section 144.
⟨ Calc M_R and M_T for two spheres 147 ⟩ Used in section 144.
⟨ Calculate and write optical properties 7 ⟩ Used in section 2.
⟨ Calculate specular limits for reflection and transmission 38 ⟩ Used in section 34.
⟨ Calculate specular reflection and transmission 220 ⟩ Used in section 219.
⟨ Calculate the unscattered transmission and reflection 226 ⟩ Used in section 225.
⟨ Check for bad values of *Tc* 221 ⟩ Used in section 219.
⟨ Check specular limits 39, 40, 41 ⟩ Used in section 34.
⟨ Check sphere parameters 42, 43 ⟩ Used in section 34.
⟨ Choose the best node of the *a* and *b* simplex 183 ⟩ Used in section 178.
⟨ Choose the best node of the *a* and *g* simplex 201 ⟩ Used in section 198.
⟨ Choose the best node of the *b* and *g* simplex 207 ⟩ Used in sections 203, 209, and 213.
⟨ Clean up guesses 266 ⟩ Used in section 256.
⟨ Declare variables for *main* 4 ⟩ Used in section 2.
⟨ Definition for *Allocate_Grid* 110 ⟩ Used in section 88.
⟨ Definition for *Calculate_Distance_With_Corrections* 144 ⟩ Used in section 88.
⟨ Definition for *Calculate_Distance* 140 ⟩ Used in section 88.
⟨ Definition for *Calculate_Grid_Distance* 142 ⟩ Used in section 88.
⟨ Definition for *Estimate_RT* 225 ⟩ Used in section 216.
⟨ Definition for *Fill_AB_Grid* 125 ⟩ Used in section 88.
⟨ Definition for *Fill_AG_Grid* 129 ⟩ Used in section 88.
⟨ Definition for *Fill_BG_Grid* 132 ⟩ Used in section 88.
⟨ Definition for *Fill_BaG_Grid* 134 ⟩ Used in section 88.
⟨ Definition for *Fill_BsG_Grid* 136 ⟩ Used in section 88.
⟨ Definition for *Fill_Grid* 138 ⟩ Used in section 88.
⟨ Definition for *Find_AB_fn* 153 ⟩ Used in section 88.
⟨ Definition for *Find_AG_fn* 151 ⟩ Used in section 88.
⟨ Definition for *Find_A_fn* 159 ⟩ Used in section 88.
⟨ Definition for *Find_BG_fn* 165 ⟩ Used in section 88.
⟨ Definition for *Find_B_fn* 161 ⟩ Used in section 88.
⟨ Definition for *Find_BaG_fn* 167 ⟩ Used in section 88.
⟨ Definition for *Find_Ba_fn* 155 ⟩ Used in section 88.
⟨ Definition for *Find_BsG_fn* 169 ⟩ Used in section 88.
⟨ Definition for *Find_Bs_fn* 157 ⟩ Used in section 88.
⟨ Definition for *Find_G_fn* 163 ⟩ Used in section 88.
⟨ Definition for *Gain_11* 97 ⟩ Used in section 88.
⟨ Definition for *Gain_22* 99 ⟩ Used in section 88.
⟨ Definition for *Gain* 95 ⟩ Used in section 88.
⟨ Definition for *Get_Calc_State* 108 ⟩ Used in section 88.
⟨ Definition for *Grid_ABG* 112 ⟩ Used in section 88.
⟨ Definition for *Initialize_Measure* 57 ⟩ Used in section 26.
⟨ Definition for *Initialize_Result* 49 ⟩ Used in section 26.
⟨ Definition for *Inverse_RT* 30 ⟩ Used in section 26.
⟨ Definition for *Max_Light_Loss* 173 ⟩ Used in section 88.
⟨ Definition for *Near_Grid_Points* 122 ⟩ Used in section 88.
⟨ Definition for *Read_Data_Line* 73 ⟩ Used in section 65.
⟨ Definition for *Read_Header* 69 ⟩ Used in section 65.
⟨ Definition for *Set_Calc_State* 106 ⟩ Used in section 88.
⟨ Definition for *Set_Grid_Debugging* 92 ⟩ Used in section 88.
⟨ Definition for *Spheres_Inverse_RT* 59 ⟩ Used in section 26.

⟨ Definition for *Two_Sphere_R* 101 ⟩ Used in section 88.
⟨ Definition for *Two_Sphere_T* 103 ⟩ Used in section 88.
⟨ Definition for *U_Find_AB* 178 ⟩ Used in section 175.
⟨ Definition for *U_Find_AG* 198 ⟩ Used in section 175.
⟨ Definition for *U_Find_A* 191 ⟩ Used in section 175.
⟨ Definition for *U_Find_BG* 203 ⟩ Used in section 175.
⟨ Definition for *U_Find_BaG* 209 ⟩ Used in section 175.
⟨ Definition for *U_Find_Ba* 189 ⟩ Used in section 175.
⟨ Definition for *U_Find_BsG* 213 ⟩ Used in section 175.
⟨ Definition for *U_Find_Bs* 187 ⟩ Used in section 175.
⟨ Definition for *U_Find_B* 195 ⟩ Used in section 175.
⟨ Definition for *U_Find_G* 193 ⟩ Used in section 175.
⟨ Definition for *Valid_Grid* 114 ⟩ Used in section 88.
⟨ Definition for *What_Is_B* 219 ⟩ Used in section 216.
⟨ Definition for *Write_Header* 81 ⟩ Used in section 65.
⟨ Definition for *a2acalc* 231 ⟩ Used in section 216.
⟨ Definition for *abg_distance* 120 ⟩ Used in section 88.
⟨ Definition for *abgb2ag* 249 ⟩ Used in section 216.
⟨ Definition for *abgg2ab* 247 ⟩ Used in section 216.
⟨ Definition for *acalc2a* 233 ⟩ Used in section 216.
⟨ Definition for *b2bcalc* 239 ⟩ Used in section 216.
⟨ Definition for *bcalc2b* 241 ⟩ Used in section 216.
⟨ Definition for *check_magic* 79 ⟩ Used in section 65.
⟨ Definition for *determine_search* 45 ⟩ Used in section 26.
⟨ Definition for *ez_Inverse_RT* 55 ⟩ Used in section 26.
⟨ Definition for *fill_grid_entry* 123 ⟩ Used in section 88.
⟨ Definition for *g2gcalc* 235 ⟩ Used in section 216.
⟨ Definition for *gcalc2g* 237 ⟩ Used in section 216.
⟨ Definition for *maxloss* 171 ⟩ Used in section 88.
⟨ Definition for *measure_OK* 34 ⟩ Used in section 26.
⟨ Definition for *quick_guess* 256 ⟩ Used in section 216.
⟨ Definition for *read_number* 77 ⟩ Used in section 65.
⟨ Definition for *skip_white* 75 ⟩ Used in section 65.
⟨ Definition for *slow_guess* 251 ⟩
⟨ Definition for *twoprime* 243 ⟩ Used in section 216.
⟨ Definition for *twounprime* 245 ⟩ Used in section 216.
⟨ Estimate the backscattered reflection 227 ⟩ Used in section 225.
⟨ Estimate the scattered transmission 228 ⟩ Used in section 225.
⟨ Estimate *aprime* 257 ⟩ Used in section 256.
⟨ Estimate *bprime* 258 ⟩ Used in sections 260, 264, and 265.
⟨ Evaluate the *BaG* simplex at the nodes 211 ⟩ Used in section 209.
⟨ Evaluate the *BsG* simplex at the nodes 215 ⟩ Used in section 213.
⟨ Evaluate the a and b simplex at the nodes 182 ⟩ Used in section 178.
⟨ Evaluate the a and g simplex at the nodes 200 ⟩ Used in section 198.
⟨ Evaluate the bg simplex at the nodes 206 ⟩ Used in section 203.
⟨ Exit with bad input data 31 ⟩ Used in section 30.
⟨ Fill r with reasonable values 50, 51, 52, 53 ⟩ Used in section 49.
⟨ Find the optical properties 32 ⟩ Used in section 30.
⟨ Find thickness when multiple internal reflections are present 223 ⟩ Used in section 219.
⟨ Free simplex data structures 185 ⟩ Used in sections 178, 198, 203, 209, and 213.
⟨ Generate next albedo using j 127 ⟩ Used in sections 125 and 129.
⟨ Get the initial a, b, and g 180 ⟩ Used in sections 178, 198, 203, 209, and 213.

⟨ Prototype for *Set_Calc_State* 105 ⟩ Used in sections 89 and 106.
⟨ Prototype for *Set_Grid_Debugging* 91 ⟩ Used in sections 89 and 92.
⟨ Prototype for *Spheres_Inverse_RT* 58 ⟩ Used in sections 28 and 59.
⟨ Prototype for *Two_Sphere_R* 100 ⟩ Used in sections 89 and 101.
⟨ Prototype for *Two_Sphere_T* 102 ⟩ Used in sections 89 and 103.
⟨ Prototype for *U_Find_AB* 177 ⟩ Used in sections 176 and 178.
⟨ Prototype for *U_Find_AG* 197 ⟩ Used in sections 176 and 198.
⟨ Prototype for *U_Find_A* 190 ⟩ Used in sections 176 and 191.
⟨ Prototype for *U_Find_BG* 202 ⟩ Used in sections 176 and 203.
⟨ Prototype for *U_Find_BaG* 208 ⟩ Used in sections 176 and 209.
⟨ Prototype for *U_Find_Ba* 188 ⟩ Used in sections 176 and 189.
⟨ Prototype for *U_Find_BsG* 212 ⟩ Used in sections 176 and 213.
⟨ Prototype for *U_Find_Bs* 186 ⟩ Used in sections 176 and 187.
⟨ Prototype for *U_Find_B* 194 ⟩ Used in sections 176 and 195.
⟨ Prototype for *U_Find_G* 192 ⟩ Used in sections 176 and 193.
⟨ Prototype for *Valid_Grid* 113 ⟩ Used in sections 89 and 114.
⟨ Prototype for *What_Is_B* 218 ⟩ Used in sections 217 and 219.
⟨ Prototype for *Write_Header* 80 ⟩ Used in sections 66 and 81.
⟨ Prototype for *a2acalc* 230 ⟩ Used in sections 217 and 231.
⟨ Prototype for *abg_distance* 119 ⟩ Used in sections 89 and 120.
⟨ Prototype for *abgb2ag* 248 ⟩ Used in sections 217 and 249.
⟨ Prototype for *abgg2ab* 246 ⟩ Used in sections 217 and 247.
⟨ Prototype for *acalc2a* 232 ⟩ Used in sections 217 and 233.
⟨ Prototype for *b2bcalc* 238 ⟩ Used in sections 217 and 239.
⟨ Prototype for *bcalc2b* 240 ⟩ Used in sections 217 and 241.
⟨ Prototype for *check_magic* 78 ⟩ Used in section 79.
⟨ Prototype for *determine_search* 44 ⟩ Used in sections 27 and 45.
⟨ Prototype for *ez_Inverse_RT* 54 ⟩ Used in sections 27, 28, and 55.
⟨ Prototype for *g2gcalc* 234 ⟩ Used in sections 217 and 235.
⟨ Prototype for *gcalc2g* 236 ⟩ Used in sections 217 and 237.
⟨ Prototype for *maxloss* 170 ⟩ Used in sections 89 and 171.
⟨ Prototype for *measure_OK* 33 ⟩ Used in sections 27 and 34.
⟨ Prototype for *quick_guess* 255 ⟩ Used in sections 217 and 256.
⟨ Prototype for *read_number* 76 ⟩ Used in section 77.
⟨ Prototype for *skip_white* 74 ⟩ Used in section 75.
⟨ Prototype for *slow_guess* 250 ⟩ Used in section 251.
⟨ Prototype for *twoprime* 242 ⟩ Used in sections 217 and 243.
⟨ Prototype for *twounprime* 244 ⟩ Used in sections 217 and 245.
⟨ Put final values in result 184 ⟩ Used in sections 178, 187, 189, 191, 193, 195, 198, 203, 209, and 213.
⟨ Read coefficients for reflection sphere 70 ⟩ Used in section 69.
⟨ Read coefficients for transmission sphere 71 ⟩ Used in section 69.
⟨ Return the deviation 148 ⟩ Used in section 144.
⟨ Slow guess for a alone 252 ⟩ Used in section 251.
⟨ Slow guess for a and b or a and g 254 ⟩ Used in section 251.
⟨ Slow guess for b alone 253 ⟩ Used in section 251.
⟨ Solve if multiple internal reflections are not present 222 ⟩ Used in section 219.
⟨ Structs to export from IAD Types 23, 24, 25 ⟩ Used in section 20.
⟨ Test easy cases 35, 36, 37 ⟩ Used in section 34.
⟨ Testing MC code 8 ⟩
⟨ Tests for invalid grid 115, 116, 117, 118 ⟩ Used in section 114.
⟨ Two parameter search 47 ⟩ Used in section 45.
⟨ Unused diffusion fragment 174 ⟩

Inverse Adding-Doubling

(Version 3.3.0)