Inverse Adding-Doubling

(Version 3-16-1)

1. iad command-line program.

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

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 21 ⟩
  ⟨ print usage function 22 ⟩
  ⟨ stringdup together function 28 ⟩
  ⟨ mystrtod function 29 ⟩
  ⟨ seconds elapsed function 30 ⟩
  ⟨ print error legend function 27 ⟩
  ⟨ what_char function 32 ⟩
  ⟨ print long error function 33 ⟩
  ⟨ print dot function 34 ⟩
  ⟨ calculate coefficients function 23 ⟩
  ⟨ parse string into array function 31 ⟩
  ⟨ print results header function 25 ⟩
  ⟨ Print results function 26 ⟩
  int main(int argc, char **argv)
  {
    ⟨ Declare variables for main 4 ⟩
    ⟨ Save command-line for use later 5 ⟩
    ⟨ Handle options 6 ⟩
    Initialize_Measure(&m);
    ⟨ Command-line changes to m 18 ⟩
    Initialize_Result(m, &r, TRUE);
    ⟨ Command-line changes to r 15 ⟩
    if (cl_forward_calc ≠ UNINITIALIZED) {
      ⟨ Calculate and Print the Forward Calculation 7 ⟩
      exit(EXIT_SUCCESS);
    }
    ⟨ prepare file for reading 12 ⟩
    if (process_command_line) {
      ⟨ Count command-line measurements 20 ⟩
      ⟨ Calculate and write optical properties 13 ⟩
      exit(EXIT_SUCCESS);
    }
    if (Read_Header(stdin, &m, &params) ≡ 0) {
      start_time = clock();
      while (Read_Data_Line(stdin, &m, &r, params) ≡ 0) {
        ⟨ Command-line changes to m 18 ⟩
        ⟨ Calculate and write optical properties 13 ⟩
      }
      ⟨ Generate and write grid 11 ⟩
    }
    if (cl_verbosity > 0) fprintf(stderr, "\n\n");
    if (any_error ∧ cl_verbosity > 1) print_error_legend();
    exit(EXIT_SUCCESS);
  }
```

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

⟨ Include files for *main* 3 ⟩ ≡

```
#define _CRT_SECURE_NO_WARNINGS
#define _CRT_NONSTDC_NO_WARNINGS
#define NO_SLIDES 0
#define ONE_SLIDE_ON_TOP 1
#define TWO_IDENTICAL_SLIDES 2
#define ONE_SLIDE_ON_BOTTOM 3
#define ONE_SLIDE_NEAR_SPHERE 4
#define ONE_SLIDE_NOT_NEAR_SPHERE 5
#define MR_IS_ONLY_RD 1
#define MT_IS_ONLY_TD 2
#define NO_UNSCATTERED_LIGHT 3
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <math.h>
#include <ctype.h>
#include <errno.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 "version.h"
#include "mc_lost.h"
#include "ad_frsnl.h"
```

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 *g_grid_name = Λ;
int c;
long n_photons = 100000;
int MAX_MC_iterations = 19;
int any_error = 0;
int process_command_line = 0;
int params = 0;
int cl_quadrature_points = UNINITIALIZED;
int cl_verbosity = 2;
double cl_forward_calc = UNINITIALIZED;
double cl_grid_calc = UNINITIALIZED;
double cl_default_a = UNINITIALIZED;
double cl_default_g = UNINITIALIZED;
double cl_default_b = UNINITIALIZED;
double cl_default_mua = UNINITIALIZED;
double cl_default_mus = UNINITIALIZED;
double cl_tolerance = UNINITIALIZED;
double cl_slide_OD = UNINITIALIZED;
double cl_cos_angle = UNINITIALIZED;
double cl_beam_d = UNINITIALIZED;
double cl_sample_d = UNINITIALIZED;
double cl_sample_n = UNINITIALIZED;
double cl_slide_d = UNINITIALIZED;
double cl_slide_n = UNINITIALIZED;
double cl_slides = UNINITIALIZED;
double cl_default_fr = UNINITIALIZED;
double cl_rstd_t = UNINITIALIZED;
double cl_rstd_r = UNINITIALIZED;
double cl_baffle_r = UNINITIALIZED;
double cl_baffle_t = UNINITIALIZED;
double cl_ru_fraction = UNINITIALIZED;
double cl_tu_fraction = UNINITIALIZED;
double cl_lambda = UNINITIALIZED;
double cl_rwall_r = UNINITIALIZED;
double cl_rwall_t = UNINITIALIZED;
double cl_search = UNINITIALIZED;
double cl_mus0 = UNINITIALIZED;
double cl_musp0 = UNINITIALIZED;
double cl_mus0_pwr = UNINITIALIZED;
double cl_mus0_lambda = UNINITIALIZED;
double cl_UR1 = UNINITIALIZED;
double cl_UT1 = UNINITIALIZED;
double cl_Tc = UNINITIALIZED;
double cl_method = UNINITIALIZED;
int 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( );
  char command_line_options[ ] = "1:2:a:A:b:B:c:C:d:D:e:E:f:F:g:G:hH:i:JL:M:n:N:o:p:q:r:R:s:S\
      :t:T:u:vV:w:W:x:Xz";
  char *command_line = Λ;
```

This code is used in section 2.

5. I want to add the command line to the output file. To do this, we need to save the entire thing before the options get processed. The extra +1 in the total length calculation is for the space character between options. Finally, we need to reset *optind* to 1 to start *getopt*( ) processing from the beginning. It should be noted that this strips any quotes from the command-line.

⟨ Save command-line for use later 5 ⟩ ≡

```
  {
    size_t command_line_length = 0;
    for (int i = 0; i < argc; ++i) {
      command_line_length += strlen(argv[i]) + 3;
    }
    command_line = (char *) malloc(command_line_length);
    if (command_line ≡ Λ) {
      fprintf(stderr, "Memory␣allocation␣failed\n");
      return 1;
    }
    strcpy(command_line, "");
    for (int i = 0; i < argc; ++i) {
      if (strchr(argv[i], '␣') ≠ Λ) {
        strcat(command_line, "'");
        strcat(command_line, argv[i]);
        strcat(command_line, "'␣");
      }
      else {
        strcat(command_line, argv[i]);
        strcat(command_line, "␣");
      }
    }
    optind = 1;
  }
```

This code is used in section 2.

6. Handling command-line options.

```
⟨ Handle options 6 ⟩ ≡
  while ((c = getopt(argc, argv, command_line_options)) ≠ EOF) {
    int n;
    char cc;
    char *tmp_str = Λ;
    switch (c) {
    case '1': tmp_str = strdup(optarg);
      parse_string_into_array(optarg, cl_sphere_one, 5);
      if (cl_sphere_one[4] ≡ UNINITIALIZED) {
        fprintf(stderr, "Error␣in␣command-line␣argument␣for␣-1\n");
        fprintf(stderr, "␣␣␣␣the␣current␣argument␣is␣'%s'␣but␣it␣must␣have␣5␣terms:␣", tmp_str);
        fprintf(stderr, "'d_sphere␣d_sample␣d_entrance␣d_detector␣r_wall'\n");
        exit(EXIT_FAILURE);
      }
      break;
    case '2': tmp_str = strdup(optarg);
      parse_string_into_array(optarg, cl_sphere_two, 5);
      if (cl_sphere_two[4] ≡ UNINITIALIZED) {
        fprintf(stderr, "Error␣in␣command-line␣argument␣for␣-2\n");
        fprintf(stderr, "␣␣␣␣the␣current␣argument␣is␣'%s'␣but␣it␣must␣have␣5␣terms:␣", tmp_str);
        fprintf(stderr, "'d_sphere␣d_sample␣d_third␣d_detector␣r_wall'\n");
        exit(EXIT_FAILURE);
      }
      break;
    case 'a': cl_default_a = my_strtod(optarg);
      if (cl_default_a < 0 ∨ cl_default_a > 1) {
        fprintf(stderr, "Error␣in␣command-line\n");
        fprintf(stderr, "␣␣␣␣albedo␣'-a␣%s'\n", optarg);
        exit(EXIT_FAILURE);
      }
      break;
    case 'A': cl_default_mua = my_strtod(optarg);
      if (cl_default_mua < 0) {
        fprintf(stderr, "Error␣in␣command-line\n");
        fprintf(stderr, "␣␣␣␣absorption␣'-A␣%s'\n", optarg);
        exit(EXIT_FAILURE);
      }
      break;
    case 'b': cl_default_b = my_strtod(optarg);
      if (cl_default_b < 0) {
        fprintf(stderr, "Error␣in␣command-line\n");
        fprintf(stderr, "␣␣␣␣optical␣thickness␣'-b␣%s'\n", optarg);
        exit(EXIT_FAILURE);
      }
      break;
    case 'B': cl_beam_d = my_strtod(optarg);
      if (cl_beam_d < 0) {
        fprintf(stderr, "Error␣in␣command-line\n");
        fprintf(stderr, "␣␣␣␣beam␣diameter␣'-B␣%s'\n", optarg);
        exit(EXIT_FAILURE);
      }
      break;
    case 'c': cl_ru_fraction = my_strtod(optarg);
      if (cl_ru_fraction < 0.0 ∨ cl_ru_fraction > 1.0) {
        fprintf(stderr, "Error␣in␣command-line\n");
        fprintf(stderr, "␣␣␣␣unscattered␣refl␣fraction␣'-c␣%s'\n", optarg);
        fprintf(stderr, "␣␣␣␣must␣be␣between␣0␣and␣1\n");
        exit(EXIT_SUCCESS);
      }
      break;
    case 'C': cl_tu_fraction = my_strtod(optarg);
      if (cl_tu_fraction < 0.0 ∨ cl_tu_fraction > 1.0) {
        fprintf(stderr, "Error␣in␣command-line\n");
        fprintf(stderr, "␣␣␣␣unscattered␣trans␣fraction␣'-C␣%s'\n", optarg);
        fprintf(stderr, "␣␣␣␣must␣be␣between␣0␣and␣1\n");
        exit(EXIT_SUCCESS);
      }
      break;
    case 'd': cl_sample_d = my_strtod(optarg);
      if (cl_sample_d < 0) {
        fprintf(stderr, "Error␣in␣command-line\n");
        fprintf(stderr, "␣␣␣␣sample␣thickness␣'-d␣%s'\n", optarg);
        exit(EXIT_FAILURE);
      }
      break;
    case 'D': cl_slide_d = my_strtod(optarg);
      if (cl_slide_d < 0) {
        fprintf(stderr, "Error␣in␣command-line\n");
        fprintf(stderr, "␣␣␣␣slide␣thickness␣'-D␣%s'\n", optarg);
        exit(EXIT_FAILURE);
      }
      break;
    case 'e': cl_tolerance = my_strtod(optarg);
      if (cl_tolerance < 0) {
        fprintf(stderr, "Error␣in␣command-line\n");
        fprintf(stderr, "␣␣␣␣error␣tolerance␣'-e␣%s'\n", optarg);
        exit(EXIT_FAILURE);
      }
      break;
    case 'E': cl_slide_OD = my_strtod(optarg);
      if (cl_slide_OD < 0) {
        fprintf(stderr, "Error␣in␣command-line\n");
        fprintf(stderr, "␣␣␣␣slide␣optical␣depth␣'-E␣%s'\n", optarg);
        exit(EXIT_FAILURE);
      }
      break;
    case 'f': cl_default_fr = my_strtod(optarg);
      if (cl_default_fr < 0.0 ∨ cl_default_fr > 1.0) {
        fprintf(stderr, "Error␣in␣command-line␣argument:␣");
        fprintf(stderr, "'-f␣%s'␣The␣argument␣must␣be␣between␣0␣and␣1.\n", optarg);
        exit(EXIT_SUCCESS);
      }
      break;
```

```
case 'F':     /* initial digit means this is mus is constant */
  if (isdigit(optarg[0])) {
    cl_default_mus = my_strtod(optarg);
    if (cl_default_mus < 0) {
      fprintf(stderr, "Error␣in␣command-line\n");
      fprintf(stderr, "␣␣␣␣mus␣'-F␣%s'\n", optarg);
      exit(EXIT_FAILURE);
    }
    break;
  }     /* should be a string like 'R 1000 1.2 -1.8' */
  n = sscanf(optarg, "%c␣%lf␣%lf␣%lf", &cc, &cl_mus0_lambda, &cl_mus0, &cl_mus0_pwr);
  if (n ≠ 4 ∨ (cc ≠ 'P' ∧ cc ≠ 'R')) {
    fprintf(stderr, "Error␣in␣command-line\n");
    fprintf(stderr, "␣␣␣␣bad␣-F␣option.␣'-F␣%s'\n", optarg);
    fprintf(stderr, "␣␣␣␣-F␣1.0␣␣␣␣␣␣␣␣␣␣␣␣␣␣for␣mus␣=1.0\n");
    fprintf(stderr, "␣␣␣␣-F␣'P␣500␣1.0␣-1.3'␣for␣mus␣=1.0*(lambda/500)^(-1.3)\n");
    fprintf(stderr, "␣␣␣␣-F␣'R␣500␣1.0␣-1.3'␣for␣mus'=1.0*(lambda/500)^(-1.3)\n");
    exit(EXIT_FAILURE);
  }
  if (cc ≡ 'R' ∨ cc ≡ 'r') {
    cl_musp0 = cl_mus0;
    cl_mus0 = UNINITIALIZED;
  }
  break;
case 'g': cl_default_g = my_strtod(optarg);
  if (cl_default_g < −1 ∨ cl_default_g > 1) {
    fprintf(stderr, "Error␣in␣command-line\n");
    fprintf(stderr, "␣␣␣␣anisotropy␣'-g␣%s'\n", optarg);
    exit(EXIT_FAILURE);
  }
  break;
case 'G':
  if (optarg[0] ≡ '0') cl_slides = NO_SLIDES;
  else if (optarg[0] ≡ '2') cl_slides = TWO_IDENTICAL_SLIDES;
  else if (optarg[0] ≡ 't' ∨ optarg[0] ≡ 'T') cl_slides = ONE_SLIDE_ON_TOP;
  else if (optarg[0] ≡ 'b' ∨ optarg[0] ≡ 'B') cl_slides = ONE_SLIDE_ON_BOTTOM;
  else if (optarg[0] ≡ 'n' ∨ optarg[0] ≡ 'N') cl_slides = ONE_SLIDE_NEAR_SPHERE;
  else if (optarg[0] ≡ 'f' ∨ optarg[0] ≡ 'F') cl_slides = ONE_SLIDE_NOT_NEAR_SPHERE;
  else {
    fprintf(stderr, "Error␣in␣command-line\n");
    fprintf(stderr, "␣␣␣␣Argument␣for␣'-G␣%s'␣must␣be␣\n", optarg);
    fprintf(stderr, "␣␣␣␣'t'␣---␣light␣always␣hits␣top␣slide␣first\n");
    fprintf(stderr, "␣␣␣␣'b'␣---␣light␣always␣hits␣bottom␣slide␣first\n");
    fprintf(stderr, "␣␣␣␣'n'␣---␣slide␣always␣closest␣to␣sphere\n");
    fprintf(stderr, "␣␣␣␣'f'␣---␣slide␣always␣farthest␣from␣sphere\n");
    exit(EXIT_FAILURE);
  }
  break;
case 'H':
  if (optarg[0] ≡ '0') {
    cl_baffle_r = 0;
    cl_baffle_t = 0;
  }
  else if (optarg[0] ≡ '1') {
    cl_baffle_r = 1;
    cl_baffle_t = 0;
  }
  else if (optarg[0] ≡ '2') {
    cl_baffle_r = 0;
    cl_baffle_t = 1;
  }
  else if (optarg[0] ≡ '3') {
    cl_baffle_r = 1;
    cl_baffle_t = 1;
  }
  else {
    fprintf(stderr, "Error␣in␣command-line␣-H␣argument\n");
    fprintf(stderr, "␣␣␣␣argument␣is␣'%s',␣but␣", optarg);
    fprintf(stderr, "must␣be␣0,␣1,␣2,␣or␣3\n");
    exit(EXIT_FAILURE);
  }
case 'i': cl_cos_angle = my_strtod(optarg);
  if (cl_cos_angle < 0 ∨ cl_cos_angle > 90) {
    fprintf(stderr, "Error␣in␣command-line\n");
    fprintf(stderr, "␣␣␣␣incident␣angle␣'-i␣%s'\n", optarg);
    fprintf(stderr, "␣␣␣␣must␣be␣between␣0␣and␣90␣degrees\n");
    exit(EXIT_FAILURE);
  }
  cl_cos_angle = cos(cl_cos_angle * M_PI/180.0);
  break;
case 'J': cl_grid_calc = 1;
  break;
case 'L': cl_lambda = my_strtod(optarg);
  break;
case 'M': MAX_MC_iterations = (int) my_strtod(optarg);
  if (MAX_MC_iterations < 0 ∨ MAX_MC_iterations > 50) {
    fprintf(stderr, "Error␣in␣command-line\n");
    fprintf(stderr, "␣␣␣␣MC␣iterations␣'-M␣%s'\n", optarg);
    exit(EXIT_FAILURE);
  }
  break;
case 'n': cl_sample_n = my_strtod(optarg);
  if (cl_sample_n < 0.1 ∨ cl_sample_n > 10) {
    fprintf(stderr, "Error␣in␣command-line\n");
    fprintf(stderr, "␣␣␣␣slab␣index␣'-n␣%s'\n", optarg);
    exit(EXIT_FAILURE);
  }
  break;
case 'N': cl_slide_n = my_strtod(optarg);
  if (cl_slide_n < 0.1 ∨ cl_slide_n > 10) {
    fprintf(stderr, "Error␣in␣command-line\n");
    fprintf(stderr, "␣␣␣␣slide␣index␣'-N␣%s'\n", optarg);
    exit(EXIT_FAILURE);
  }
  break;
case 'o': g_out_name = strdup(optarg);
  break;
case 'p': n_photons = (long) my_strtod(optarg);
  break;
case 'q': cl_quadrature_points = (int) my_strtod(optarg);
  if (cl_quadrature_points % 4 ≠ 0) {
    fprintf(stderr, "Error␣in␣command-line\n");
    fprintf(stderr, "␣␣␣␣'-q␣%s'\n", optarg);
    fprintf(stderr, "␣␣␣␣Quadrature␣points␣must␣be␣a␣multiple␣of␣4\n");
    exit(EXIT_FAILURE);
  }
  if ((cl_cos_angle ≠ UNINITIALIZED) ∧ (cl_quadrature_points % 12 ≠ 0)) {
    fprintf(stderr, "Error␣in␣command-line\n");
    fprintf(stderr, "␣␣␣␣'-q␣%s'\n", optarg);
    fprintf(stderr,
        "␣␣␣␣Quadrature␣points␣must␣be␣multiple␣of␣12␣for␣oblique␣incidence\n");
    exit(EXIT_FAILURE);
  }
  break;
case 'r': cl_UR1 = my_strtod(optarg);
  process_command_line = 1;
  if (cl_UR1 < 0 ∨ cl_UR1 > 1) {
    fprintf(stderr, "Error␣in␣command-line\n");
    fprintf(stderr, "␣␣␣␣UR1␣value␣'-r␣%s'\n", optarg);
    fprintf(stderr, "␣␣␣␣must␣be␣between␣0␣and␣1\n");
    exit(EXIT_FAILURE);
  }
  break;
case 'R': cl_rstd_r = my_strtod(optarg);
  if (cl_rstd_r < 0 ∨ cl_rstd_r > 1) {
    fprintf(stderr, "Error␣in␣command-line\n");
    fprintf(stderr, "␣␣␣␣Rstd␣value␣'-R␣%s'\n", optarg);
    fprintf(stderr, "␣␣␣␣must␣be␣between␣0␣and␣1\n");
    exit(EXIT_FAILURE);
  }
  break;
case 's': cl_search = (int) my_strtod(optarg);
  break;
case 'S': cl_num_spheres = (int) my_strtod(optarg);
  if (cl_num_spheres ≠ 0 ∧ cl_num_spheres ≠ 1 ∧ cl_num_spheres ≠ 2) {
    fprintf(stderr, "Error␣in␣command-line\n");
    fprintf(stderr, "␣␣␣␣sphere␣number␣'-S␣%s'\n", optarg);
    fprintf(stderr, "␣␣␣␣must␣be␣0,␣1,␣or␣2\n");
    exit(EXIT_FAILURE);
  }
  break;
case 't': cl_UT1 = my_strtod(optarg);
  if (cl_UT1 < 0 ∨ cl_UT1 > 1) {
    fprintf(stderr, "Error␣in␣command-line\n");
    fprintf(stderr, "␣␣␣␣UT1␣value␣'-t␣%s'\n", optarg);
    fprintf(stderr, "␣␣␣␣must␣be␣between␣0␣and␣1\n");
    exit(EXIT_FAILURE);
  }
  process_command_line = 1;
  break;
case 'T': cl_rstd_t = my_strtod(optarg);
  if (cl_rstd_t < 0 ∨ cl_rstd_t > 1) {
    fprintf(stderr, "Error␣in␣command-line\n");
    fprintf(stderr, "␣␣␣␣transmission␣standard␣'-T␣%s'\n", optarg);
    fprintf(stderr, "␣␣␣␣must␣be␣between␣0␣and␣1\n");
    exit(EXIT_FAILURE);
  }
  break;
case 'u': cl_Tc = my_strtod(optarg);
  if (cl_Tc < 0 ∨ cl_Tc > 1) {
    fprintf(stderr, "Error␣in␣command-line\n");
    fprintf(stderr, "␣␣␣␣unscattered␣transmission␣'-u␣%s'\n", optarg);
    fprintf(stderr, "␣␣␣␣must␣be␣between␣0␣and␣1\n");
    exit(EXIT_FAILURE);
  }
  process_command_line = 1;
  break;
case 'v': print_version(cl_verbosity);
  exit(EXIT_SUCCESS);
  break;
case 'V': cl_verbosity = my_strtod(optarg);
  break;
case 'w': cl_rwall_r = my_strtod(optarg);
  if (cl_rwall_r < 0 ∨ cl_rwall_r > 1) {
    fprintf(stderr, "Error␣in␣command-line\n");
    fprintf(stderr, "␣␣␣␣refl␣sphere␣wall␣'-w␣%s'\n", optarg);
    fprintf(stderr, "␣␣␣␣must␣be␣between␣0␣and␣1\n");
    exit(EXIT_FAILURE);
  }
  break;
case 'W': cl_rwall_t = my_strtod(optarg);
  if (cl_rwall_t < 0 ∨ cl_rwall_r > 1) {
    fprintf(stderr, "Error␣in␣command-line\n");
    fprintf(stderr, "␣␣␣␣trans␣sphere␣wall␣'-w␣%s'\n", optarg);
    fprintf(stderr, "␣␣␣␣must␣be␣between␣0␣and␣1\n");
    exit(EXIT_FAILURE);
  }
  break;
case 'x': Set_Debugging((int) my_strtod(optarg));
  break;
case 'X': cl_method = COMPARISON;
  break;
case 'z': cl_forward_calc = 1;
  process_command_line = 1;
  break;
default: fprintf(stderr, "unknown␣option␣'%c'\n", c);      /* fall through */
case 'h': print_usage();
  exit(EXIT_SUCCESS);
  }
}
argc −= optind;
argv += optind;
```

This code is used in section 2.

7. The forward calculation.

We are doing a forward calculation. We still need to set the albedo and optical depth appropriately. Obviously when the -a switch is used then the albedo should be fixed as a constant equal to *cl_default_a*. The other cases are less clear. If scattering and absorption are both specified, then calculate the albedo using these values. If the scattering is not specified, then we assume that the sample is an unscattering sample and therefore the albedo is zero. On the other hand, if the scattering is specified and the absorption is not, then the albedo is set to one.

```
⟨ Calculate and Print the Forward Calculation 7 ⟩ ≡
  if (cl_default_a ≡ UNINITIALIZED) {
    if (cl_default_mus ≡ UNINITIALIZED) r.a = 0;
    else if (cl_default_mua ≡ UNINITIALIZED) r.a = 1;
    else r.a = cl_default_mus/(cl_default_mua + cl_default_mus);
  }
  else r.a = cl_default_a;
```

See also sections 8, 9, and 10.

This code is used in section 2.

8. This is slightly more tricky because there are four things that can affect the optical thickness — *cl_default_b*, the default mua, default mus and the thickness. If the sample thickness is unspecified, then the only reasonable thing to do is to assume that the sample is very thick. Otherwise, we use the sample thickness to calculate the optical thickness.

```
⟨ Calculate and Print the Forward Calculation 7 ⟩ +≡
  if (cl_default_b ≡ UNINITIALIZED) {
    if (cl_sample_d ≡ UNINITIALIZED) r.b = HUGE_VAL;
    else if (r.a ≡ 0) {
      if (cl_default_mua ≡ UNINITIALIZED) r.b = HUGE_VAL;
      else r.b = cl_default_mua ∗ cl_sample_d;
    }
    else {
      if (cl_default_mus ≡ UNINITIALIZED) r.b = HUGE_VAL;
      else r.b = cl_default_mus/r.a ∗ cl_sample_d;
    }
  }
  else r.b = cl_default_b;
```

9. The easiest case, use the default value or set it to zero

```
⟨ Calculate and Print the Forward Calculation 7 ⟩ +≡
  if (cl_default_g ≡ UNINITIALIZED) r.g = 0;
  else r.g = cl_default_g;
```

10. ⟨ Calculate and Print the Forward Calculation 7 ⟩ +≡

```
r.slab.a = r.a;
r.slab.b = r.b;
r.slab.g = r.g;
{
   double mu_sp, mu_a, m_r, m_t;
   if (MAX_MC_iterations ≡ 0) {
      Calculate_MR_MT(m, r, MC_NONE, TRUE, &m_r, &m_t);
   }
   else {
      Calculate_MR_MT(m, r, MC_REDO, TRUE, &m_r, &m_t);
   }
   Calculate_Mua_Musp(m, r, &mu_sp, &mu_a);
   if (cl_verbosity > 0) {
      Write_Header(m, r, −1, command_line);
      print_results_header(stdout);
   }
   print_optical_property_result(stdout, m, r, m_r, m_t, mu_a, mu_sp, 0);
}
```

11. Calculating a grid for graphing.

We will start simple. Just vary a' and b'.

```
⟨ Generate and write grid 11 ⟩ ≡
  if (cl_grid_calc ≠ UNINITIALIZED) {
    double m_r, m_t, aprime, bprime, g;
    double aa[ ] = {0, 0.8, 0.9, 0.95, 0.98, 0.99, 1.0};
    double bb[ ] = {0, 0.2, 0.5, 1.0, 3.0, 10.0, 100};
    int i, j;
    FILE *grid;

    grid = fopen(g_grid_name, "w");
    if (grid ≡ Λ) {
      fprintf(stderr, "Could␣not␣open␣grid␣file␣'%s'␣for␣output\n", g_out_name);
      exit(EXIT_FAILURE);
    }
    m.ur1_lost = 0;
    m.uru_lost = 0;
    m.ut1_lost = 0;
    m.utu_lost = 0;
    if (r.default_g ≠ UNINITIALIZED) {
      g = r.default_g;
    }
    else if (r.found) {
      g = r.slab.g;
    }
    else {
      g = 0;
    }
    fprintf(grid, "#␣%s␣(g=%6.4f)\n", command_line, g);
    fprintf(grid, "#␣␣␣␣a'␣␣␣␣␣␣␣␣␣␣b'␣␣␣␣␣␣␣␣␣␣g␣␣␣␣␣␣␣␣␣␣␣M_R␣␣␣␣␣␣␣␣␣M_T\n");
    fprintf(stderr, "\ndoing␣grid␣calculation\n");
    for (i = 0; i < 7; i++) {
      aprime = aa[i];
      for (j = 0; j < 7; j++) {
        bprime = bb[j];
        r.slab.a = aprime/(1 − g + aprime * g);
        r.slab.b = bprime/(1 − r.slab.a * g);
        if (MAX_MC_iterations ≡ 0) {
          Calculate_MR_MT(m, r, MC_NONE, TRUE, &m_r, &m_t);
        }
        else {
          Calculate_MR_MT(m, r, MC_REDO, TRUE, &m_r, &m_t);
        }
        fprintf(stderr, "*");
        fprintf(grid, "%10.5f,␣%10.5f,␣%10.5f,␣%10.5f,␣%10.5f\n", 39aprime, bprime, g, m_r, m_t);
      }
    }
    fclose(grid);
    fprintf(stderr, "\n");
  }
```

This code is used in section 2.

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

```
⟨ prepare file for reading 12 ⟩ ≡
  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(EXIT_FAILURE);
  }
  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(EXIT_FAILURE);
    }
    if (g_out_name ≡ Λ) g_out_name = strdup_together(base_name, ".txt");
    if (g_grid_name ≡ Λ) g_grid_name = strdup_together(base_name, ".grid");
    free(rt_name);
    free(base_name);
    process_command_line = 0;
  }
  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(EXIT_FAILURE);
    }
  }
```

This code is used in section 2.

13. Need to explicitly reset *r.search* each time through the loop, because it will get altered by the calculation process. This also allows the command line to overwrite the reflection or transmission value from the command-line with something like `-r 0` or `-t 0`.

We also 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 13 ⟩ ≡
  {
    ⟨ Local Variables for Calculation 14 ⟩
    if (Debug(DEBUG_ANY)) {
      fprintf(stderr, "\n-------------------NEXT␣DATA␣POINT---------------------\n");
      if (m.lambda ≠ 0) fprintf(stderr, "lambda=%6.1f␣", m.lambda);
      fprintf(stderr, "MR=%8.5f␣MT=%8.5f\n\n", m.m_r, m.m_t);
    }
    Initialize_Result(m, &r, FALSE);
    r.default_a = UNINITIALIZED;
    r.default_b = UNINITIALIZED;
    r.default_g = UNINITIALIZED;
    r.default_mua = UNINITIALIZED;
    r.default_mus = UNINITIALIZED;
    ⟨ Command-line changes to r 15 ⟩
    ⟨ Warn and quit for bad options 19 ⟩
    ⟨ Write Header 16 ⟩
    m.ur1_lost = 0;
    m.uru_lost = 0;
    m.ut1_lost = 0;
    m.utu_lost = 0;
    Inverse_RT(m, &r);
    ⟨ Improve result using Monte Carlo 17 ⟩
    calculate_coefficients(m, r, &LR, &LT, &mu_sp, &mu_a);
    print_optical_property_result(stdout, m, r, LR, LT, mu_a, mu_sp, rt_total);
    if (r.error ≠ IAD_NO_ERROR) any_error = 1;
    if (Debug(DEBUG_ANY)) print_long_error(r.error);
    else print_dot(start_time, r.error, mc_total, TRUE, cl_verbosity);
  }
```

This code is used in section 2.

14.

```
⟨ Local Variables for Calculation 14 ⟩ ≡
  static int rt_total = 0;
  static int mc_total = 0;
  double ur1 = 0;
  double ut1 = 0;
  double uru = 0;
  double utu = 0;
  double mu_a = 0;
  double mu_sp = 0;
  double LR = 0;
  double LT = 0;

  rt_total++;
```

This code is used in section 13.

15. ⟨ Command-line changes to *r* 15 ⟩ ≡

```
if (cl_quadrature_points ≠ UNINITIALIZED) r.method.quad_pts = cl_quadrature_points;
else r.method.quad_pts = 8;
if (cl_default_a ≠ UNINITIALIZED) r.default_a = cl_default_a;
if (cl_default_mua ≠ UNINITIALIZED) {
  r.default_mua = cl_default_mua;
  if (cl_sample_d ≠ UNINITIALIZED) r.default_ba = cl_default_mua ∗ cl_sample_d;
  else r.default_ba = cl_default_mua ∗ m.slab_thickness;
}
if (cl_default_b ≠ UNINITIALIZED) r.default_b = cl_default_b;
if (cl_default_g ≠ UNINITIALIZED) r.default_g = cl_default_g;
if (cl_tolerance ≠ UNINITIALIZED) {
  r.tolerance = cl_tolerance;
  r.MC_tolerance = cl_tolerance;
}
if (cl_musp0 ≠ UNINITIALIZED)
  cl_mus0 = (r.default_g ≠ UNINITIALIZED) ? cl_musp0/(1 − r.default_g) : cl_musp0;
if (cl_mus0 ≠ UNINITIALIZED ∧ m.lambda ≠ 0)
  cl_default_mus = cl_mus0 ∗ pow(m.lambda/cl_mus0_lambda, cl_mus0_pwr);
if (cl_default_mus ≠ UNINITIALIZED) {
  r.default_mus = cl_default_mus;
  if (cl_sample_d ≠ UNINITIALIZED) r.default_bs = cl_default_mus ∗ cl_sample_d;
  else r.default_bs = cl_default_mus ∗ m.slab_thickness;
}
if (cl_search ≠ UNINITIALIZED) r.search = cl_search;
```

This code is used in sections 2 and 13.

16. ⟨ Write Header 16 ⟩ ≡

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

This code is used in section 13.

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

```
⟨ Improve result using Monte Carlo 17 ⟩ ≡
  if (r.found ∧ m.num_spheres > 0) {
    double mu_sp_last = mu_sp;
    double mu_a_last = mu_a;
    if (Debug(DEBUG_LOST_LIGHT)) {
      print_results_header(stderr);
      print_optical_property_result(stderr, m, r, LR, LT, mu_a, mu_sp, rt_total);
    }
    while (r.MC_iterations < MAX_MC_iterations) {
      if (Debug(DEBUG_ITERATIONS))
        fprintf(stderr, "\n-------------␣Monte␣Carlo␣Iteration␣%d␣----------------\n",
            r.MC_iterations + 1);
      MC_Lost(m, r, n_photons, &ur1, &ut1, &uru, &utu, &m.ur1_lost, &m.ut1_lost, &m.uru_lost,
          &m.utu_lost);
      mc_total++;
      r.MC_iterations++;
      Inverse_RT(m, &r);
      calculate_coefficients(m, r, &LR, &LT, &mu_sp, &mu_a);
      if (fabs(mu_a_last − mu_a)/(mu_a + 0.0001) < r.MC_tolerance ∧ fabs(mu_sp_last − mu_sp)/(mu_sp +
            0.0001) < r.MC_tolerance) break;
      mu_a_last = mu_a;
      mu_sp_last = mu_sp;
      if (Debug(DEBUG_LOST_LIGHT))
        print_optical_property_result(stderr, m, r, LR, LT, mu_a, mu_sp, rt_total);
      else print_dot(start_time, r.error, mc_total, FALSE, cl_verbosity);
      if (r.found ≡ FALSE) break;
    }
  }
```

This code is used in section 13.

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

⟨ Command-line changes to *m* 18 ⟩ ≡

```
if (cl_cos_angle ≠ UNINITIALIZED) {
  m.slab_cos_angle = cl_cos_angle;
  if (cl_quadrature_points ≡ UNINITIALIZED) cl_quadrature_points = 12;
  if (cl_quadrature_points ≠ 12 ∗ (cl_quadrature_points/12)) {
    fprintf(stderr,
        "If␣you␣use␣the␣-i␣option␣to␣specify␣an␣oblique␣incidence␣angle,␣then\n");
    fprintf(stderr, "the␣number␣of␣quadrature␣points␣must␣be␣a␣multiple␣of␣12\n");
    exit(EXIT_SUCCESS);
  }
}
if (cl_sample_n ≠ UNINITIALIZED) m.slab_index = cl_sample_n;
if (cl_slide_n ≠ UNINITIALIZED) {
  m.slab_bottom_slide_index = cl_slide_n;
  m.slab_top_slide_index = cl_slide_n;
}
if (cl_slide_OD ≠ UNINITIALIZED) {
  m.slab_bottom_slide_b = cl_slide_OD;
  m.slab_top_slide_b = cl_slide_OD;
}
if (cl_sample_d ≠ UNINITIALIZED) m.slab_thickness = cl_sample_d;
if (cl_beam_d ≠ UNINITIALIZED) m.d_beam = cl_beam_d;
if (cl_slide_d ≠ UNINITIALIZED) {
  m.slab_bottom_slide_thickness = cl_slide_d;
  m.slab_top_slide_thickness = cl_slide_d;
}
if (cl_slides ≡ NO_SLIDES) {
  m.slab_bottom_slide_index = 1.0;
  m.slab_bottom_slide_thickness = 0.0;
  m.slab_top_slide_index = 1.0;
  m.slab_top_slide_thickness = 0.0;
}
if (cl_slides ≡ ONE_SLIDE_ON_TOP ∨ cl_slides ≡ ONE_SLIDE_NEAR_SPHERE) {
  m.slab_bottom_slide_index = 1.0;
  m.slab_bottom_slide_thickness = 0.0;
}
if (cl_slides ≡ ONE_SLIDE_ON_BOTTOM ∨ cl_slides ≡ ONE_SLIDE_NOT_NEAR_SPHERE) {
  m.slab_top_slide_index = 1.0;
  m.slab_top_slide_thickness = 0.0;
}
if (cl_slides ≡ ONE_SLIDE_NEAR_SPHERE ∨ cl_slides ≡ ONE_SLIDE_NOT_NEAR_SPHERE) m.flip_sample = 1;
else m.flip_sample = 0;
if (cl_method ≠ UNINITIALIZED) m.method = (int) cl_method;
if (cl_rstd_t ≠ UNINITIALIZED) m.rstd_t = cl_rstd_t;
if (cl_rstd_r ≠ UNINITIALIZED) m.rstd_r = cl_rstd_r;
if (cl_rwall_r ≠ UNINITIALIZED) m.rw_r = cl_rwall_r;
if (cl_rwall_t ≠ UNINITIALIZED) m.rw_t = cl_rwall_t;
if (cl_sphere_one[0] ≠ UNINITIALIZED) {
  double d_sample_r, d_third_r, d_detector_r;
```

```
    m.d_sphere_r = cl_sphere_one[0];
    d_sample_r = cl_sphere_one[1];
    d_third_r = cl_sphere_one[2];
    d_detector_r = cl_sphere_one[3];
    m.rw_r = cl_sphere_one[4];
    m.as_r = (d_sample_r/m.d_sphere_r/2) ∗ (d_sample_r/m.d_sphere_r/2);
    m.at_r = (d_third_r/m.d_sphere_r/2) ∗ (d_third_r/m.d_sphere_r/2);
    m.ad_r = (d_detector_r/m.d_sphere_r/2) ∗ (d_detector_r/m.d_sphere_r/2);
    m.aw_r = 1.0 − m.as_r − m.at_r − m.ad_r;
    m.d_sphere_t = m.d_sphere_r;
    m.as_t = m.as_r;
    m.at_t = m.at_r;
    m.ad_t = m.ad_r;
    m.aw_t = m.aw_r;
    m.rw_t = m.rw_r;
    if (cl_num_spheres ≡ UNINITIALIZED) m.num_spheres = 1;
  }
  if (cl_sphere_two[0] ≠ UNINITIALIZED) {
    double d_sample_t, d_third_t, d_detector_t;

    m.d_sphere_t = cl_sphere_two[0];
    d_sample_t = cl_sphere_two[1];
    d_third_t = cl_sphere_two[2];
    d_detector_t = cl_sphere_two[3];
    m.rw_t = cl_sphere_two[4];
    m.as_t = (d_sample_t/m.d_sphere_t/2) ∗ (d_sample_t/m.d_sphere_t/2);
    m.at_t = (d_third_t/m.d_sphere_t/2) ∗ (d_third_t/m.d_sphere_t/2);
    m.ad_t = (d_detector_t/m.d_sphere_t/2) ∗ (d_detector_t/m.d_sphere_t/2);
    m.aw_t = 1.0 − m.as_t − m.at_t − m.ad_t;
    if (cl_num_spheres ≡ UNINITIALIZED) m.num_spheres = 2;
  }
  if (cl_num_spheres ≠ UNINITIALIZED) {
    m.num_spheres = (int) cl_num_spheres;
    if (m.num_spheres > 0 ∧ m.method ≡ UNKNOWN) m.method = SUBSTITUTION;
  }
  if (cl_ru_fraction ≠ UNINITIALIZED) m.fraction_of_ru_in_mr = cl_ru_fraction;
  if (cl_tu_fraction ≠ UNINITIALIZED) m.fraction_of_tu_in_mt = cl_tu_fraction;
  if (cl_UR1 ≠ UNINITIALIZED) m.m_r = cl_UR1;
  if (cl_UT1 ≠ UNINITIALIZED) m.m_t = cl_UT1;
  if (cl_Tc ≠ UNINITIALIZED) m.m_u = cl_Tc;
  if (cl_default_fr ≠ UNINITIALIZED) m.f_r = cl_default_fr;
  if (cl_baffle_r ≠ UNINITIALIZED) m.baffle_r = cl_baffle_r;
  if (cl_baffle_t ≠ UNINITIALIZED) m.baffle_t = cl_baffle_t;
  if (cl_lambda ≠ UNINITIALIZED) m.lambda = cl_lambda;
```

This code is used in section 2.

19. ⟨ Warn and quit for bad options 19 ⟩ ≡

```
if (cl_method ≡ COMPARISON ∧ m.d_sphere_r ≠ 0 ∧ m.as_r ≡ 0) {
  fprintf(stderr, "A␣dual-beam␣measurement␣is␣specified,␣but␣no␣port␣sizes.\n");
  fprintf(stderr, "You␣might␣forsake␣the␣-X␣option␣and␣use␣zero␣spheres␣(which␣gives\n");
  fprintf(stderr, "the␣same␣result␣except␣lost␣light␣is␣not␣taken␣into␣account).\n");
  fprintf(stderr, "Alternatively,␣bite␣the␣bullet␣and␣enter␣your␣sphere␣parameters,\n");
  fprintf(stderr, "with␣the␣knowledge␣that␣only␣the␣beam␣diameter␣and␣sample␣port\n");
  fprintf(stderr, "diameter␣will␣be␣used␣to␣estimate␣lost␣light␣from␣the␣edges.\n");
  exit(EXIT_SUCCESS);
}
if (cl_method ≡ COMPARISON ∧ m.num_spheres ≡ 2) {
  fprintf(stderr, "A␣dual-beam␣measurement␣is␣specified,␣but␣a␣two␣sphere␣experiment\n");
  fprintf(stderr, "is␣specified.␣Since␣this␣seems␣impossible,␣I␣will␣make␣it\n");
  fprintf(stderr, "impossible␣for␣you␣unless␣you␣specify␣0␣or␣1␣sphere.\n");
  exit(EXIT_SUCCESS);
}
if (cl_method ≡ COMPARISON ∧ m.f_r ≠ 0) {
  fprintf(stderr, "A␣dual-beam␣measurement␣is␣specified,␣but␣a␣fraction␣of␣light\n");
  fprintf(stderr, "is␣specified␣to␣hit␣the␣sphere␣wall␣first.␣␣This␣situation\n");
  fprintf(stderr, "is␣not␣supported␣by␣iad.␣␣Sorry.\n");
  exit(EXIT_SUCCESS);
}
```

This code is used in section 13.

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

⟨ Count command-line measurements 20 ⟩ ≡

```
m.num_measures = 3;
if (m.m_r ≡ 0) m.num_measures−−;
if (m.m_t ≡ 0) m.num_measures−−;
if (m.m_u ≡ 0) m.num_measures−−;
params = m.num_measures;
if (m.num_measures ≡ 3) {     /* need to fill slab entries to calculate the optical thickness */
  struct AD_slab_type s;

  s.n_slab = m.slab_index;
  s.n_top_slide = m.slab_top_slide_index;
  s.n_bottom_slide = m.slab_bottom_slide_index;
  s.b_top_slide = m.slab_top_slide_b;
  s.b_bottom_slide = m.slab_bottom_slide_b;
  s.cos_angle = m.slab_cos_angle;
}
```

This code is used in section 2.

21. ⟨ print version function 21 ⟩ ≡

```
static void print_version(int verbosity)
{
  if (verbosity ≡ 0) {
    fprintf(stdout, "%s", VersionShort);
  }
  else {
    fprintf(stdout, "iad␣%s\n", Version);
    fprintf(stdout, "Copyright␣1993-2024␣Scott␣Prahl,␣scott.prahl@oit.edu\n");
    fprintf(stdout, "␣␣␣␣␣␣␣␣␣␣␣(see␣Applied␣Optics,␣32:559-568,␣1993)\n\n");
    fprintf(stdout, "This␣is␣free␣software;␣see␣the␣source␣for␣copying␣conditions.\n");
    fprintf(stdout, "There␣is␣no␣warranty;␣not␣even␣for␣MERCHANTABILITY␣or␣FITNESS.\n");
    fprintf(stdout, "FOR␣A␣PARTICULAR␣PURPOSE.\n");
  }
}
```

This code is used in section 2.

22. ⟨ print usage function 22 ⟩ ≡

```
static void print_usage(void)
{
  fprintf(stdout, "iad␣%s\n\n", Version);
  fprintf(stdout, "iad␣finds␣optical␣properties␣from␣measurements\n\n");
  fprintf(stdout, "Usage:␣␣iad␣[options]␣input\n\n");
  fprintf(stdout, "Options:\n");
  fprintf(stdout, "␣␣-1␣'#␣#␣#␣#␣#'␣␣␣reflection␣sphere␣parameters␣\n");
  fprintf(stdout, "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣'d_sphere␣d␣d_sample_port␣d_entrance_por\
      t␣d_detector_port␣r_wall'\n");
  fprintf(stdout, "␣␣-2␣'#␣#␣#␣#␣#'␣␣␣transmission␣sphere␣parameters␣\n");
  fprintf(stdout, "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣'d_sphere␣d␣d_sample_port␣d_third_port␣d\
      _detector_port␣r_wall'\n");
  fprintf(stdout, "␣␣-a␣#␣␣␣␣␣␣␣␣␣␣␣␣␣use␣this␣albedo␣\n");
  fprintf(stdout, "␣␣-A␣#␣␣␣␣␣␣␣␣␣␣␣␣␣use␣this␣absorption␣coefficient␣\n");
  fprintf(stdout, "␣␣-b␣#␣␣␣␣␣␣␣␣␣␣␣␣␣use␣this␣optical␣thickness␣\n");
  fprintf(stdout, "␣␣-B␣#␣␣␣␣␣␣␣␣␣␣␣␣␣beam␣diameter␣\n");
  fprintf(stdout, "␣␣-c␣#␣␣␣␣␣␣␣␣␣␣␣␣␣fraction␣of␣unscattered␣refl␣in␣MR\n");
  fprintf(stdout, "␣␣-C␣#␣␣␣␣␣␣␣␣␣␣␣␣␣fraction␣of␣unscattered␣trans␣in␣MT\n");
  fprintf(stdout, "␣␣-d␣#␣␣␣␣␣␣␣␣␣␣␣␣␣thickness␣of␣sample␣\n");
  fprintf(stdout, "␣␣-D␣#␣␣␣␣␣␣␣␣␣␣␣␣␣thickness␣of␣slide␣\n");
  fprintf(stdout, "␣␣-e␣#␣␣␣␣␣␣␣␣␣␣␣␣␣error␣tolerance␣(default␣0.0001)␣\n");
  fprintf(stdout, "␣␣-E␣#␣␣␣␣␣␣␣␣␣␣␣␣␣optical␣depth␣(=mua*D)␣for␣slides\n");
  fprintf(stdout,
      "␣␣-f␣#␣␣␣␣␣␣␣␣␣␣␣␣␣allow␣a␣fraction␣0.0-1.0␣of␣light␣to␣hit␣sphere␣wall␣first\n");
  fprintf(stdout, "␣␣-F␣#␣␣␣␣␣␣␣␣␣␣␣␣␣constrain␣scattering␣coefficient␣\n");
  fprintf(stdout, "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣#␣=␣constant:␣use␣constant␣scattering␣coefficient␣\n");
  fprintf(stdout, "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣#␣=␣'P␣lambda0␣mus0␣gamma'␣then␣mus=mus0\
      *(lambda/lambda0)^gamma\n");
  fprintf(stdout, "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣#␣=␣'R␣lambda0␣musp0␣gamma'␣␣musp=musp0*\
      (lambda/lambda0)^gamma\n");
  fprintf(stdout, "␣␣-g␣#␣␣␣␣␣␣␣␣␣␣␣␣␣scattering␣anisotropy␣(default␣0)␣\n");
  fprintf(stdout, "␣␣-G␣#␣␣␣␣␣␣␣␣␣␣␣␣␣type␣of␣boundary␣'0',␣'2',␣'t',␣'b',␣'n',␣'f'␣\n");
  fprintf(stdout, "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣'0'␣or␣'2'␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣---␣number␣of␣slides\n");
  fprintf(stdout, "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣'t'␣(top)␣or␣'b'␣(bottom)␣---␣one␣slide␣\
      that␣is␣hit␣by␣light␣first\n");
  fprintf(stdout, "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣'n'␣(near)␣or␣'f'␣(far)␣␣␣---␣one␣slide␣\
      position␣relative␣to␣sphere\n");
  fprintf(stdout, "␣␣-h␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣display␣help\n");
  fprintf(stdout, "␣␣-H␣#␣␣␣␣␣␣␣␣␣␣␣␣␣#␣=␣0,␣no␣baffles␣for␣R␣or␣T␣spheres\n");
  fprintf(stdout, "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣#␣=␣1,␣baffle␣for␣R␣but␣not␣for␣T␣sphere\n");
  fprintf(stdout, "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣#␣=␣2,␣baffle␣for␣T␣but␣not␣for␣R␣sphere\n");
  fprintf(stdout, "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣#␣=␣3,␣baffle␣for␣both␣R␣and␣T␣spheres␣(default)\n");
  fprintf(stdout, "␣␣-L␣#␣␣␣␣␣␣␣␣␣␣␣␣␣specify␣the␣wavelength␣lambda\n");
  fprintf(stdout, "␣␣-M␣#␣␣␣␣␣␣␣␣␣␣␣␣␣number␣of␣Monte␣Carlo␣iterations\n");
  fprintf(stdout, "␣␣-n␣#␣␣␣␣␣␣␣␣␣␣␣␣␣specify␣index␣of␣refraction␣of␣slab\n");
  fprintf(stdout, "␣␣-N␣#␣␣␣␣␣␣␣␣␣␣␣␣␣specify␣index␣of␣refraction␣of␣slides\n");
  fprintf(stdout, "␣␣-o␣filename␣␣␣␣␣␣explicitly␣specify␣filename␣for␣output\n");
  fprintf(stdout, "␣␣-p␣#␣␣␣␣␣␣␣␣␣␣␣␣␣#␣of␣Monte␣Carlo␣photons␣(default␣100000)\n");
  fprintf(stdout, "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣a␣negative␣number␣is␣max␣MC␣time␣in␣milliseconds\n");
  fprintf(stdout, "␣␣-q␣#␣␣␣␣␣␣␣␣␣␣␣␣␣number␣of␣quadrature␣points␣(default=8)\n");
  fprintf(stdout, "␣␣-r␣#␣␣␣␣␣␣␣␣␣␣␣␣␣total␣reflection␣measurement\n");
  fprintf(stdout, "␣␣-R␣#␣␣␣␣␣␣␣␣␣␣␣␣␣␣actual␣reflectance␣for␣100%%␣measurement␣\n");
  fprintf(stdout, "␣␣-S␣#␣␣␣␣␣␣␣␣␣␣␣␣␣␣number␣of␣spheres␣used\n");
  fprintf(stdout, "␣␣-t␣#␣␣␣␣␣␣␣␣␣␣␣␣␣␣total␣transmission␣measurement\n");
  fprintf(stdout, "␣␣-T␣#␣␣␣␣␣␣␣␣␣␣␣␣␣␣actual␣transmission␣for␣100%%␣measurement␣\n");
  fprintf(stdout, "␣␣-u␣#␣␣␣␣␣␣␣␣␣␣␣␣␣␣unscattered␣transmission␣measurement\n");
  fprintf(stdout, "␣␣-v␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣version␣information\n");
  fprintf(stdout, "␣␣-V␣0␣␣␣␣␣␣␣␣␣␣␣␣␣verbosity␣low␣---␣no␣output␣to␣stdout\n");
  fprintf(stdout, "␣␣-V␣1␣␣␣␣␣␣␣␣␣␣␣␣␣verbosity␣moderate␣\n");
  fprintf(stdout, "␣␣-V␣2␣␣␣␣␣␣␣␣␣␣␣␣␣verbosity␣high\n");
  fprintf(stdout, "␣␣-w␣#␣␣␣␣␣␣␣␣␣␣␣␣␣␣wall␣reflectivity␣for␣reflection␣sphere\n");
  fprintf(stdout, "␣␣-W␣#␣␣␣␣␣␣␣␣␣␣␣␣␣␣wall␣reflectivity␣for␣transmission␣sphere\n");
  fprintf(stdout, "␣␣-x␣#␣␣␣␣␣␣␣␣␣␣␣␣␣␣set␣debugging␣level\n");
  fprintf(stdout, "␣␣-X␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣dual␣beam␣configuration\n");
  fprintf(stdout, "␣␣-z␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣do␣forward␣calculation\n");
  fprintf(stdout, "Examples:\n");
  fprintf(stdout, "␣␣iad␣file.rxt␣␣␣␣␣␣␣␣␣␣␣␣␣␣Results␣will␣be␣put␣in␣file.txt\n");
  fprintf(stdout, "␣␣iad␣file␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Same␣as␣above\n");
  fprintf(stdout, "␣␣iad␣-c␣0.9␣file.rxt␣␣␣␣␣␣␣Assume␣M_R␣includes␣90%%␣of␣uns\
      cattered␣reflectance\n");
  fprintf(stdout, "␣␣iad␣-C␣0.8␣file.rxt␣␣␣␣␣␣␣Assume␣M_T␣includes␣80%%␣of␣uns\
      cattered␣transmittance\n");
  fprintf(stdout, "␣␣iad␣-e␣0.0001␣file.rxt␣␣␣␣Better␣convergence␣to␣R␣&␣T␣values\n");
  fprintf(stdout,
      "␣␣iad␣-f␣1.0␣file.rxt␣␣␣␣␣␣␣All␣light␣hits␣reflectance␣sphere␣wall␣first\n");
  fprintf(stdout, "␣␣iad␣-o␣out␣file.rxt␣␣␣␣␣␣␣Calculated␣values␣in␣out\n");
  fprintf(stdout, "␣␣iad␣-r␣0.3␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣R_total=0.3,␣b=inf,␣find␣albedo\n");
  fprintf(stdout, "␣␣iad␣-r␣0.3␣-t␣0.4␣␣␣␣␣␣␣␣␣R_total=0.3,␣T_total=0.4,␣find␣a,b,g\n");
  fprintf(stdout, "␣␣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(stdout, "␣␣iad␣-r␣0.3␣-t␣0.4␣␣␣␣␣␣␣␣␣R_total=0.3,␣T_total=0.4,␣find␣a,b\n");
  fprintf(stdout, "␣␣iad␣-p␣1000␣file.rxt␣␣␣␣␣␣Only␣1000␣photons\n");
  fprintf(stdout, "␣␣iad␣-p␣-100␣file.rxt␣␣␣␣␣␣Allow␣only␣100ms␣per␣iteration\n");
  fprintf(stdout, "␣␣iad␣-q␣4␣file.rxt␣␣␣␣␣␣␣␣␣Four␣quadrature␣points\n");
  fprintf(stdout, "␣␣iad␣-M␣0␣file.rxt␣␣␣␣␣␣␣␣␣No␣MC␣␣␣␣(iad)\n");
  fprintf(stdout, "␣␣iad␣-M␣1␣file.rxt␣␣␣␣␣␣␣␣␣MC␣once␣␣(iad␣->␣MC␣->␣iad)\n");
  fprintf(stdout, "␣␣iad␣-M␣2␣file.rxt␣␣␣␣␣␣␣␣␣MC␣twice␣(iad␣->␣MC␣->␣iad␣->␣MC␣->␣iad)\n");
  fprintf(stdout, "␣␣iad␣-M␣0␣-q␣4␣file.rxt␣␣␣␣Fast␣and␣crude␣conversion\n");
  fprintf(stdout,
      "␣␣iad␣-G␣t␣file.rxt␣␣␣␣␣␣␣␣␣One␣top␣slide␣with␣properties␣from␣file.rxt\n");
  fprintf(stdout,
      "␣␣iad␣-G␣b␣-N␣1.5␣-D␣1␣file␣Use␣1␣bottom␣slide␣with␣n=1.5␣and␣thickness=1\n");
  fprintf(stdout, "␣␣iad␣-x␣␣␣1␣file.rxt␣␣␣␣␣␣␣Show␣sphere␣and␣MC␣effects\n");
  fprintf(stdout, "␣␣iad␣-x␣␣␣2␣file.rxt␣␣␣␣␣␣␣Show␣grid␣decisions\n");
  fprintf(stdout, "␣␣iad␣-x␣␣␣4␣file.rxt␣␣␣␣␣␣␣Show␣interations\n");
  fprintf(stdout, "␣␣iad␣-x␣␣␣8␣file.rxt␣␣␣␣␣␣␣Show␣lost␣light␣effects\n");
  fprintf(stdout, "␣␣iad␣-x␣␣16␣file.rxt␣␣␣␣␣␣␣Show␣best␣grid␣points\n");
  fprintf(stdout, "␣␣iad␣-x␣␣32␣file.rxt␣␣␣␣␣␣␣Show␣decisions␣for␣type␣of␣search\n");
  fprintf(stdout, "␣␣iad␣-x␣␣64␣file.rxt␣␣␣␣␣␣␣Show␣all␣grid␣calculations\n");
  fprintf(stdout, "␣␣iad␣-x␣128␣file.rxt␣␣␣␣␣␣␣Show␣sphere␣calculations\n");
  fprintf(stdout, "␣␣iad␣-x␣256␣file.rxt␣␣␣␣␣␣␣DEBUG_EVERY_CALC\n");
  fprintf(stdout, "␣␣iad␣-x␣511␣file.rxt␣␣␣␣␣␣␣Show␣all␣debugging␣output\n");
  fprintf(stdout,
      "␣␣iad␣-X␣-i␣8␣file.rxt␣␣␣␣␣␣Dual␣beam␣spectrometer␣with␣8␣degree␣incidence\n\n");
```

```
  fprintf(stdout,
      "␣␣iad␣-z␣-a␣0.9␣-b␣1␣-i␣45␣␣Forward␣calc␣assuming␣45␣degree␣incidence\n\n");
  fprintf(stdout, "␣␣apply␣iad␣x.rxt␣y.rxt␣␣␣␣␣Process␣multiple␣files\n\n");
  fprintf(stdout, "Report␣bugs␣to␣<scott.prahl@oit.edu>\n\n");
}
```

This code is used in section 2.

23. Just figure out the damn scattering and absorption

```
⟨ calculate coefficients function 23 ⟩ ≡
  static void Calculate_Mua_Musp(struct measure_type m, struct invert_type r, double
        *musp, double *mua)
  {
    if (r.b ≡ HUGE_VAL) {
      if (r.a ≤ 1 · 10^{−5}) {
        *musp = 0.0;
        *mua = 1.0;
        return;
      }
      if (r.default_mus ≠ UNINITIALIZED) {
        *musp = r.default_mus * (1 − r.g);
        *mua = r.default_mus/r.a − r.default_mus;
        return;
      }
      if (r.default_mua ≠ UNINITIALIZED) {
        *musp = (r.default_mua/(1 − r.a) − r.default_mua) * (1 − r.g);
        *mua = r.default_mua;
        return;
      }
      *musp = 1.0 − r.g;
      *mua = (1.0 − r.a)/r.a;
      return;
    }
    *musp = r.a * r.b/m.slab_thickness * (1.0 − r.g);
    *mua = (1 − r.a) * r.b/m.slab_thickness;
  }
```

See also section 24.

This code is used in section 2.

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

```
⟨ calculate coefficients function 23 ⟩ +≡
  static void calculate_coefficients(struct measure_type m, struct invert_type r, double *LR, double
        *LT, double *musp, double *mua)
  {
    double delta;
    *LR = 0;
    *LT = 0;
    Calculate_Distance(LR, LT, &delta);
    Calculate_Mua_Musp(m, r, musp, mua);
  }
```

25. ⟨ print results header function 25 ⟩ ≡

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

This code is used in section 2.

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

```
⟨ Print results function 26 ⟩ ≡
  void print_optical_property_result(FILE *fp, struct measure_type m, struct invert_type r, double
        LR, double LT, double mu_a, double mu_sp, int line)
  {
    if (Debug(DEBUG_LOST_LIGHT)) {
      if (m.lambda ≠ 0) fprintf(fp, "%6.1f␣␣␣", m.lambda);
      else fprintf(fp, "%6d␣␣␣", line);
      if (mu_a ≥ 200) mu_a = 199.9999;
      if (mu_sp ≥ 1000) mu_sp = 999.9999;
      fprintf(fp, "%6.4f␣%␣6.4f␣|␣", m.m_r, LR);
      fprintf(fp, "%6.4f␣%␣6.4f␣|␣", m.m_t, LT);
      fprintf(fp, "%6.3f␣", mu_a);
      fprintf(fp, "%6.3f␣", mu_sp);
      fprintf(fp, "%6.3f␣|", r.g);
      fprintf(fp, "␣%6.4f␣%6.4f␣", m.ur1_lost, m.uru_lost);
      fprintf(fp, "%6.4f␣%6.4f␣|␣", m.ut1_lost, m.utu_lost);
      fprintf(fp, "%2d␣␣", r.MC_iterations);
      fprintf(fp, "%3d", r.AD_iterations);
      fprintf(fp, "␣␣␣␣%c␣\n", what_char(r.error));
    }
    else {
      if (m.lambda ≠ 0) fprintf(fp, "%6.1f\t", m.lambda);
      else fprintf(fp, "%6d\t", line);
      if (mu_a ≥ 200) mu_a = 199.9999;
      if (mu_sp ≥ 1000) mu_sp = 999.9999;
      fprintf(fp, "%␣9.4f\t%␣9.4f\t", m.m_r, LR);
      fprintf(fp, "%␣9.4f\t%␣9.4f\t", m.m_t, LT);
      fprintf(fp, "%␣9.4f\t", mu_a);
      fprintf(fp, "%␣9.4f\t", mu_sp);
      fprintf(fp, "%␣9.4f\t", r.g);
      fprintf(fp, "␣%c␣\n", what_char(r.error));
    }
    fflush(fp);
  }
```

This code is used in section 2.

27. ⟨ print error legend function 27 ⟩ ≡

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

This code is used in section 2.

28. returns a new string consisting of s+t

⟨ stringdup together function 28 ⟩ ≡

```
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.

29. catch parsing errors in strtod

```
⟨ mystrtod function 29 ⟩ ≡
  static double my_strtod(const char *str)
  {
    char *endptr;
    errno = 0;
    double val = strtod(str, &endptr);
    if (endptr ≡ str) {   /* No digits were found */
      fprintf(stderr, "Error␣in␣command-line\n");
      fprintf(stderr, "␣␣␣␣No␣conversion␣could␣be␣performed␣for␣`%s`.\n", str);
      exit(EXIT_FAILURE);
    }
    if (*endptr ≠ '\0') {   /* String contains extra characters after the number */
      fprintf(stderr, "Error␣in␣command-line\n");
      fprintf(stderr, "␣␣␣␣Partial␣conversion␣of␣string␣=␣'%s'\n", str);
      exit(EXIT_FAILURE);
    }
    if (errno ≡ ERANGE) {   /* The converted value is out of range of representable values by a double */
      fprintf(stderr, "Error␣in␣command-line\n");
      printf("␣␣␣␣The␣value␣'%s'␣is␣out␣of␣range␣of␣double.\n", str);
      exit(EXIT_FAILURE);
    }
    return val;
  }
```

This code is used in section 2.

30. assume that start time has already been set

```
⟨ seconds elapsed function 30 ⟩ ≡
  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.

31. 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 31 ⟩ ≡
  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) {
        if (a[i − 1] ≤ 0 ∨ a[i − 1] > 1) {
          fprintf(stderr,
              "Sphere␣wall␣reflectivity␣(r_w=%g)␣must␣be␣a␣fraction␣less␣than␣one.\n",
              a[i − 1]);
          exit(EXIT_FAILURE);
        }
        return 0;
      }      /* move pointer just after last number */
      t = r + 1;
    }
    return 1;
  }
```

This code is used in section 2.

32.

```
⟨ what_char function 32 ⟩ ≡
  static char what_char(int err)
  {
    if (err ≡ IAD_NO_ERROR) return '*';
    if (err ≡ IAD_TOO_MANY_ITERATIONS) return '+';
    if (err ≡ IAD_MR_TOO_BIG) return 'R';
    if (err ≡ IAD_MR_TOO_SMALL) return 'r';
    if (err ≡ IAD_MT_TOO_BIG) return 'T';
    if (err ≡ IAD_MT_TOO_SMALL) return 't';
    if (err ≡ IAD_MU_TOO_BIG) return 'U';
    if (err ≡ IAD_MU_TOO_SMALL) return 'u';
    if (err ≡ IAD_TOO_MUCH_LIGHT) return '!';
    return '?';
  }
```

This code is used in section 2.

33. ⟨ print long error function 33 ⟩ ≡

```
static void print_long_error(int err)
{
  if (err ≡ IAD_TOO_MANY_ITERATIONS) fprintf(stderr, "Failed␣Search,␣too␣many␣iterations\n");
  if (err ≡ IAD_MR_TOO_BIG) fprintf(stderr, "Failed␣Search,␣M_R␣is␣too␣big\n");
  if (err ≡ IAD_MR_TOO_SMALL) fprintf(stderr, "Failed␣Search,␣M_R␣is␣too␣small\n");
  if (err ≡ IAD_MT_TOO_BIG) fprintf(stderr, "Failed␣Search,␣M_T␣is␣too␣big\n");
  if (err ≡ IAD_MT_TOO_SMALL) fprintf(stderr, "Failed␣Search,␣M_T␣is␣too␣small\n");
  if (err ≡ IAD_MU_TOO_BIG) fprintf(stderr, "Failed␣Search,␣M_U␣is␣too␣big\n");
  if (err ≡ IAD_MU_TOO_SMALL) fprintf(stderr, "Failed␣Search,␣M_U␣is␣too␣snall\n");
  if (err ≡ IAD_TOO_MUCH_LIGHT) fprintf(stderr, "Failed␣Search,␣Total␣light␣bigger␣than␣1\n");
  if (err ≡ IAD_NO_ERROR) fprintf(stderr, "Successful␣Search\n");
  fprintf(stderr, "\n");
}
```

This code is used in section 2.

34. The idea here is to show some intermediate output while a file is being processed.

⟨ print dot function 34 ⟩ ≡

```
static void print_dot(clock_t start_time, int err, int points, int final, int verbosity)
{
  static int counter = 0;

  counter++;
  if (verbosity ≡ 0 ∨ Debug(DEBUG_ANY)) return;
  if (final) fprintf(stderr, "%c", what_char(err));
  else {
    counter--;
    fprintf(stderr, "%1d\b", points % 10);
  }
  if (final) {
    if (counter % 50 ≡ 0) {
      double rate = (seconds_elapsed(start_time)/counter);

      fprintf(stderr, "␣␣%3d␣done␣(%5.2f␣s/pt)\n", counter, rate);
    }
    else if (counter % 10 ≡ 0) fprintf(stderr, "␣");
  }
  fflush(stderr);
}
```

This code is used in section 2.

35. IAD Types. This file has no routines. It is responsible for creating the header file `iad_type.h` and nothing else.

```
⟨ iad_type.h 35 ⟩ ≡
#undef FALSE
#undef TRUE
  ⟨ Preprocessor definitions ⟩
  ⟨ Structs to export from IAD Types 38 ⟩
```

36.

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

37. Need error codes for this silly program

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

38. 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 38⟩ ≡

```
  typedef struct measure_type {
    double slab_index;
    double slab_thickness;
    double slab_top_slide_index;
    double slab_top_slide_b;
    double slab_top_slide_thickness;
    double slab_bottom_slide_index;
    double slab_bottom_slide_b;
    double slab_bottom_slide_thickness;
    double slab_cos_angle;
    int num_spheres;
    int num_measures;
    int method;
    int flip_sample;
    int baffle_r, baffle_t;
    double d_beam;
    double fraction_of_ru_in_mr;
    double fraction_of_tu_in_mt;
    double m_r, m_t, m_u;
    double lambda;
    double as_r, ad_r, at_r, aw_r, rd_r, rw_r, rstd_r, f_r;
    double as_t, ad_t, at_t, aw_t, rd_t, rw_t, rstd_t;
    double ur1_lost, uru_lost, ut1_lost, utu_lost;
    double d_sphere_r, d_sphere_t;
  } IAD_measure_type;
```

See also sections 39 and 40.

This code is used in section 35.

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

```
⟨ Structs to export from IAD Types 38 ⟩ +≡
  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 error;
    struct AD_slab_type slab;
    struct AD_method_type method;
    int AD_iterations;
    int MC_iterations;
    double default_a;
    double default_b;
    double default_g;
    double default_ba;
    double default_bs;
    double default_mua;
    double default_mus;
  } IAD_invert_type;
```

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

```
⟨ Structs to export from IAD Types 38 ⟩ +≡
  typedef int search_type;
  typedef int boolean_type;
  typedef int illumination_type;
  typedef struct guess_t {
    double distance;
    double a;
    double b;
    double g;
  } guess_type;
  extern double FRACTION;
```

41. 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 41 ⟩ ≡
#include <stdio.h>
#include <math.h>
#include "nr_util.h"
#include "ad_globl.h"
#include "ad_frsnl.h"
#include "iad_type.h"
#include "iad_util.h"
#include "iad_calc.h"
#include "iad_find.h"
#include "iad_pub.h"
#include "iad_io.h"
#include "stdlib.h"
#include "mc_lost.h"
  ⟨ Definition for Inverse_RT 45 ⟩
  ⟨ Definition for measure_OK 51 ⟩
  ⟨ Definition for determine_search 58 ⟩
  ⟨ Definition for Initialize_Result 62 ⟩
  ⟨ Definition for Initialize_Measure 70 ⟩
  ⟨ Definition for ez_Inverse_RT 68 ⟩
  ⟨ Definition for Spheres_Inverse_RT 72 ⟩
  ⟨ Definition for Spheres_Inverse_RT2 85 ⟩
  ⟨ Definition for Calculate_MR_MT 79 ⟩
  ⟨ Definition for MinMax_MR_MT 83 ⟩
  ⟨ Definition for Calculate_Minimum_MR 81 ⟩
```

42. 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 42 ⟩ ≡
  ⟨ Prototype for Inverse_RT 44 ⟩;
  ⟨ Prototype for measure_OK 50 ⟩;
  ⟨ Prototype for determine_search 57 ⟩;
  ⟨ Prototype for Initialize_Result 61 ⟩;
  ⟨ Prototype for ez_Inverse_RT 67 ⟩;
  ⟨ Prototype for Initialize_Measure 69 ⟩;
  ⟨ Prototype for Calculate_MR_MT 78 ⟩;
  ⟨ Prototype for MinMax_MR_MT 82 ⟩;
  ⟨ Prototype for Calculate_Minimum_MR 80 ⟩;
  ⟨ Prototype for Spheres_Inverse_RT2 84 ⟩;
```

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

```
⟨ lib_iad.h 43 ⟩ ≡
  ⟨ Prototype for ez_Inverse_RT 67 ⟩;
  ⟨ Prototype for Spheres_Inverse_RT 71 ⟩;
  ⟨ Prototype for Spheres_Inverse_RT2 84 ⟩;
```

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

```
⟨ Prototype for Inverse_RT 44 ⟩ ≡
  void Inverse_RT(struct measure_type m, struct invert_type *r)
```

This code is used in sections 42 and 45.

45. ⟨ Definition for *Inverse_RT* 45 ⟩ ≡

```
  ⟨ Prototype for Inverse_RT 44 ⟩
  {
    if (m.m_u > 0 ∧ r→default_b ≡ UNINITIALIZED) {
      r→default_b = What_Is_B(r→slab, m.m_u);
    }
    if (r→search ≡ FIND_AUTO) {
      r→search = determine_search(m, *r);
    }
    if (r→search ≡ FIND_B_WITH_NO_ABSORPTION) {
      r→default_a = 1;
      r→search = FIND_B;
    }
    if (r→search ≡ FIND_B_WITH_NO_SCATTERING) {
      r→default_a = 0;
      r→search = FIND_B;
    }
    ⟨ Exit with bad input data 46 ⟩
    ⟨ Find the optical properties 47 ⟩
    ⟨ Print basic sphere and MC effects 48 ⟩
  }
```

This code is used in section 41.

46. 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 46 ⟩ ≡
  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 45.

47. 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 47 ⟩ ≡
  if (Debug(DEBUG_ITERATIONS)) {
    fprintf(stderr, "----------------␣Beginning␣New␣Search␣-----------------\n");
    fprintf(stderr, "␣␣␣␣␣␣␣␣␣␣");
    fprintf(stderr, "␣␣␣␣␣␣a␣␣␣␣␣␣␣␣␣b␣␣␣␣␣␣␣␣␣␣g␣␣␣|");
    fprintf(stderr, "␣␣␣␣␣M_R␣␣␣␣␣␣␣␣calc␣␣␣|");
    fprintf(stderr, "␣␣␣␣␣M_T␣␣␣␣␣␣␣␣calc␣␣␣|");
    if (r→metric ≡ RELATIVE) fprintf(stderr, "␣relative␣distance\n");
    else fprintf(stderr, "␣absolute␣distance\n");
  }
  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 (Debug(DEBUG_ITERATIONS))
    fprintf(stderr, "Final␣amoeba/brent␣result␣after␣%d␣iterations\n", r→AD_iterations);
  if (r→AD_iterations ≥ IAD_MAX_ITERATIONS) r→error = IAD_TOO_MANY_ITERATIONS;
```

This code is used in section 45.

48. This is to support -x 1

```
⟨ Print basic sphere and MC effects 48 ⟩ ≡
  if (Debug(DEBUG_A_LITTLE)) {
    double M_R, M_T;
    fprintf(stderr, "AD␣iterations=␣%3d␣␣␣MC␣iterations=%3d", r→AD_iterations, r→MC_iterations);
    fprintf(stderr, "␣␣␣␣␣␣␣␣␣␣␣␣a=%6.4f␣b=%8.4f␣g=%6.4f\n", r→slab.a, r→slab.b, r→slab.g);
    fprintf(stderr, "␣␣␣␣M_R␣loss␣␣␣␣␣%8.5f␣␣M_T␣loss␣␣␣␣␣%8.5f", m.ur1_lost, m.ut1_lost);
    if (r→MC_iterations ≡ 0) fprintf(stderr, "␣(␣no␣MC␣calculation␣yet)\n");
    else fprintf(stderr, "␣(␣MC␣loss␣calculation)\n");
    Calculate_MR_MT(m, *r, MC_NONE, FALSE, &M_R, &M_T);
    fprintf(stderr, "␣␣␣␣M_R␣bare␣␣␣␣␣%8.5f␣␣M_T␣bare␣␣␣␣␣%8.5f", M_R, M_T);
    fprintf(stderr, "␣(␣---␣MC␣loss,␣---␣sphere␣effects)\n");
    Calculate_MR_MT(m, *r, MC_NONE, TRUE, &M_R, &M_T);
    fprintf(stderr, "␣␣␣␣M_R␣sphere␣␣␣%8.5f␣␣M_T␣sphere␣␣␣%8.5f", M_R, M_T);
    fprintf(stderr, "␣(␣---␣MC␣loss,␣+++␣sphere␣effects)\n");
    Calculate_MR_MT(m, *r, MC_USE_EXISTING, FALSE, &M_R, &M_T);
    fprintf(stderr, "␣␣␣␣M_R␣mc␣␣␣␣␣␣␣%8.5f␣␣M_T␣mc␣␣␣␣␣␣␣%8.5f", M_R, M_T);
    fprintf(stderr, "␣(␣+++␣MC␣loss,␣---␣sphere␣effects)\n");
    Calculate_MR_MT(m, *r, MC_USE_EXISTING, TRUE, &M_R, &M_T);
    fprintf(stderr, "␣␣␣␣M_R␣both␣␣␣␣␣%8.5f␣␣M_T␣both␣␣␣␣␣%8.5f", M_R, M_T);
    fprintf(stderr, "␣(␣+++␣MC␣loss,␣+++␣sphere␣effects)\n");
    fprintf(stderr, "␣␣␣␣M_R␣measured␣%8.5f␣␣M_T␣measured␣%8.5f", m.m_r, m.m_t);
    fprintf(stderr, "␣(␣␣target␣values)\n");
    fprintf(stderr, "Final␣distance␣%8.5f\n\n", r→final_distance);
  }
```

This code is used in section 45.

49. Validation.

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

```
⟨ Prototype for measure_OK 50 ⟩ ≡
  int measure_OK(struct measure_type m, struct invert_type r)
```

This code is used in sections 42 and 51.

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

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

```
⟨ Definition for measure_OK 51 ⟩ ≡
  ⟨ Prototype for measure_OK 50 ⟩
  {
    double ru, tu;
    if (m.num_spheres ≠ 2) {
      ⟨ Check MT for zero or one spheres 53 ⟩
      ⟨ Check MR for zero or one spheres 52 ⟩
    }
    else {
      int error = MinMax_MR_MT(m, r);
      if (error ≠ IAD_NO_ERROR) return error;
    }
    ⟨ Check MU 54 ⟩
    if (m.num_spheres ≠ 0) {
      ⟨ Check sphere parameters 55 ⟩
    }
    return IAD_NO_ERROR;
  }
```

This code is used in section 41.

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

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

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

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

⟨ Check `MR` for zero or one spheres 52 ⟩ ≡

```
{
  double mr, mt;
  Calculate_Minimum_MR(m, r, &mr, &mt);
  if (m.m_r < 0) return IAD_MR_TOO_SMALL;
  if (m.m_r > 1) return IAD_MR_TOO_BIG;
      /* one parameter search only needs one good measurement */
  if (r.search ≡ FIND_A ∨ r.search ≡ FIND_G ∨ r.search ≡ FIND_B ∨ r.search ≡ FIND_Bs ∨ r.search ≡
        FIND_Ba) {
    if (m.m_r < mr ∧ m.m_t ≤ 0) return IAD_MR_TOO_SMALL;
  }
  else {
    if (r.default_a ≡ UNINITIALIZED ∨ r.default_a > 0) {
      if (m.m_r < mr) return IAD_MR_TOO_SMALL;
    }
  }
}
```

This code is used in section 51.

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

There is a problem when spheres are present. The estimated values for the transmittance using *Sp_mu_RT* are not actually limiting cases. This will require a bit of fixing, but for now that test is omitted if the number of spheres is more than zero.

⟨ Check `MT` for zero or one spheres 53 ⟩ ≡

```
if (m.m_t < 0) return IAD_MT_TOO_SMALL;
if (m.m_t > 1) return IAD_MR_TOO_BIG;
Sp_mu_RT_Flip(m.flip_sample, r.slab.n_top_slide, r.slab.n_slab, r.slab.n_bottom_slide, r.slab.b_top_slide, 0,
      r.slab.b_bottom_slide, r.slab.cos_angle, &ru, &tu);
if (m.num_spheres ≡ 0 ∧ m.m_t > tu) {
  fprintf(stderr, "ntop=%7.5f,␣nslab=%7.5f,␣nbottom=%7.5f\n", r.slab.n_top_slide, r.slab.n_slab,
      r.slab.n_bottom_slide);
  fprintf(stderr, "tu_max=%7.5f,␣m_t=%7.5f,␣t_std=%7.5f\n", tu, m.m_t, m.rstd_t);
  return IAD_MT_TOO_BIG;
}
```

This code is used in section 51.

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

⟨ Check MU 54 ⟩ ≡

```
if ($m.m\_u < 0$) return IAD_MU_TOO_SMALL;
if ($m.m\_t > 0 \land m.m\_u > m.m\_t$) return IAD_MU_TOO_BIG;
```

This code is used in section 51.

55. Make sure that reflection sphere parameters are reasonable

⟨ Check sphere parameters 55 ⟩ ≡

```
if ($m.as\_r < 0 \lor m.as\_r \geq 0.2$) return IAD_AS_NOT_VALID;
if ($m.ad\_r < 0 \lor m.ad\_r \geq 0.2$) return IAD_AD_NOT_VALID;
if ($m.at\_r < 0 \lor m.at\_r \geq 0.2$) return IAD_AE_NOT_VALID;
if ($m.rw\_r < 0 \lor m.rw\_r > 1.0$) return IAD_RW_NOT_VALID;
if ($m.rd\_r < 0 \lor m.rd\_r > 1.0$) return IAD_RD_NOT_VALID;
if ($m.rstd\_r < 0 \lor m.rstd\_r > 1.0$) return IAD_RSTD_NOT_VALID;
if ($m.rstd\_t < 0 \lor m.rstd\_t > 1.0$) return IAD_TSTD_NOT_VALID;
if ($m.f\_r < 0 \lor m.f\_r > 1$) return IAD_F_NOT_VALID;
```

See also section 56.

This code is used in section 51.

56. Make sure that transmission sphere parameters are reasonable

⟨ Check sphere parameters 55 ⟩ +≡

```
if ($m.as\_t < 0 \lor m.as\_t \geq 0.2$) return IAD_AS_NOT_VALID;
if ($m.ad\_t < 0 \lor m.ad\_t \geq 0.2$) return IAD_AD_NOT_VALID;
if ($m.at\_t < 0 \lor m.at\_t \geq 0.2$) return IAD_AE_NOT_VALID;
if ($m.rw\_t < 0 \lor m.rw\_r > 1.0$) return IAD_RW_NOT_VALID;
if ($m.rd\_t < 0 \lor m.rd\_t > 1.0$) return IAD_RD_NOT_VALID;
if ($m.rstd\_t < 0 \lor m.rstd\_t > 1.0$) return IAD_TSTD_NOT_VALID;
```

57. 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* 57 ⟩ ≡

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

This code is used in sections 42 and 58.

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

```
⟨ Definition for determine_search 58 ⟩ ≡
  ⟨ Prototype for determine_search 57 ⟩
  {
    double rt, tt, rd, td, tc, rc;
    int search = 0;
    int independent = 0;
    int constraints = 0;

    if (m.m_r > 0) independent++;
    if (m.m_t > 0) independent++;
    if (r.default_a ≠ UNINITIALIZED) constraints++;
    if (r.default_b ≠ UNINITIALIZED) constraints++;
    if (r.default_g ≠ UNINITIALIZED) constraints++;
    if (r.default_mua ≠ UNINITIALIZED) constraints++;
    if (r.default_mus ≠ UNINITIALIZED) constraints++;
    Estimate_RT(m, r, &rt, &tt, &rd, &rc, &td, &tc);
    if (Debug(DEBUG_SEARCH)) {
      fprintf(stderr, "SEARCH:␣starting␣with␣%d␣measurement(s)\n", independent);
      fprintf(stderr, "SEARCH:␣␣␣␣␣␣and␣with␣%d␣constraint(s)\n", constraints);
      fprintf(stderr, "SEARCH:␣m_r␣=␣%8.5f␣", m.m_r);
      fprintf(stderr, "m_t␣=␣%8.5f␣", m.m_t);
      fprintf(stderr, "m_u␣=␣%8.5f\n", m.m_u);
      fprintf(stderr, "SEARCH:␣␣rt␣=␣%8.5f␣", rt);
      fprintf(stderr, "␣rd␣=␣%8.5f␣", rd);
      fprintf(stderr, "␣ru␣=␣%8.5f\n", rc);
      fprintf(stderr, "SEARCH:␣␣tt␣=␣%8.5f␣", tt);
      fprintf(stderr, "␣td␣=␣%8.5f␣", td);
      fprintf(stderr, "␣tu␣=␣%8.5f\n", tc);
    }
    if (rd ≡ 0 ∧ independent ≥ 2) {
      if (Debug(DEBUG_SEARCH)) fprintf(stderr, "SEARCH:␣no␣information␣in␣rd\n");
      independent−−;
    }
    if (td ≡ 0 ∧ independent ≥ 2) {
      if (Debug(DEBUG_SEARCH)) fprintf(stderr, "SEARCH:␣no␣information␣in␣td\n");
      independent−−;
    }
    if (constraints + independent > 3) {
      fprintf(stderr, "Too␣many␣constraints!\n");
    }
    if (independent ≡ 0) {
      search = FIND_A;
    }
    else if (independent ≡ 1) {
      ⟨ One parameter search 59 ⟩
    }
    else if (independent ≡ 2) {
      ⟨ Two parameter search 60 ⟩
    }   /* three real parameters with information! */
    else {
      search = FIND_AG;
    }
    if (search ≡ FIND_BG ∧ m.m_u > 0) search = FIND_G;
    if (Debug(DEBUG_SEARCH)) {
      fprintf(stderr, "SEARCH:␣ending␣with␣%d␣measurement(s)\n", independent);
      fprintf(stderr, "SEARCH:␣␣␣␣and␣with␣%d␣constraint(s)\n", constraints);
      fprintf(stderr, "SEARCH:␣final␣choice␣for␣search␣=␣");
      if (search ≡ FIND_A) fprintf(stderr, "FIND_A\n");
      if (search ≡ FIND_B) fprintf(stderr, "FIND_B\n");
      if (search ≡ FIND_AB) fprintf(stderr, "FIND_AB\n");
      if (search ≡ FIND_AG) fprintf(stderr, "FIND_AG\n");
      if (search ≡ FIND_AUTO) fprintf(stderr, "FIND_AUTO\n");
      if (search ≡ FIND_BG) fprintf(stderr, "FIND_BG\n");
      if (search ≡ FIND_BaG) fprintf(stderr, "FIND_BaG\n");
      if (search ≡ FIND_BsG) fprintf(stderr, "FIND_BsG\n");
      if (search ≡ FIND_Ba) fprintf(stderr, "FIND_Ba\n");
      if (search ≡ FIND_Bs) fprintf(stderr, "FIND_Bs\n");
      if (search ≡ FIND_G) fprintf(stderr, "FIND_G\n");
      if (search ≡ FIND_B_WITH_NO_ABSORPTION) fprintf(stderr, "FIND_B_WITH_NO_ABSORPTION\n");
      if (search ≡ FIND_B_WITH_NO_SCATTERING) fprintf(stderr, "FIND_B_WITH_NO_SCATTERING\n");
    }
    return search;
  }
```

This code is used in section 41.

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

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

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

```
⟨ One parameter search 59 ⟩ ≡
  if (r.default_a ≠ UNINITIALIZED) {
    if (r.default_a ≡ 0) search = FIND_B_WITH_NO_SCATTERING;
    else if (r.default_a ≡ 1) search = FIND_B_WITH_NO_ABSORPTION;
    else if (tt ≡ 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 (m.m_t ≡ 0) search = FIND_A;
  else if (rd ≡ 0) search = FIND_B_WITH_NO_SCATTERING;
  else search = FIND_B_WITH_NO_ABSORPTION;
```

This code is used in section 58.

60. 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 60 ⟩ ≡
  if (r.default_a ≠ UNINITIALIZED) {
    if (r.default_a ≡ 0) search = FIND_B;
    else if (r.default_g ≠ UNINITIALIZED) search = FIND_B;
    else if (r.default_b ≠ UNINITIALIZED) search = FIND_G;
    else search = FIND_BG;
  }
  else if (r.default_b ≠ UNINITIALIZED) {
    if (r.default_g ≠ UNINITIALIZED) search = FIND_A;
    else search = FIND_AG;
  }
  else if (r.default_g ≠ UNINITIALIZED) {
    search = FIND_AB;
  }
  else if (r.default_ba ≠ UNINITIALIZED) {
    if (r.default_g ≠ UNINITIALIZED) search = FIND_Bs;
    else search = FIND_BsG;
  }
  else if (r.default_bs ≠ UNINITIALIZED) {
    if (r.default_g ≠ UNINITIALIZED) search = FIND_Ba;
    else search = FIND_BaG;
  }
  else if (rt + tt > 1 ∧ 0 ∧ m.num_spheres ≠ 2) search = FIND_B_WITH_NO_ABSORPTION;
  else search = FIND_AB;
```

This code is used in section 58.

61. 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 61 ⟩ ≡
  void Initialize_Result(struct measure_type m, struct invert_type *r, int overwrite_defaults)
```

This code is used in sections 42 and 62.

62.
```
⟨ Definition for Initialize_Result 62 ⟩ ≡
  ⟨ Prototype for Initialize_Result 61 ⟩
  {
    ⟨ Fill r with reasonable values 63 ⟩
  }
```

This code is used in section 41.

63. Start with the optical properties.

```
⟨ Fill r with reasonable values 63 ⟩ ≡
  r→a = 0.0;
  r→b = 0.0;
  r→g = 0.0;
```

See also sections 64, 65, and 66.

This code is used in section 62.

64. Continue with other useful stuff.

```
⟨ Fill r with reasonable values 63 ⟩ +≡
  r->found = FALSE;
  r->tolerance = 0.0001;
  r->MC_tolerance = 0.01;    /* percent */
  r->search = FIND_AUTO;
  r->metric = ABSOLUTE;
  r->final_distance = 10;
  r->AD_iterations = 0;
  r->MC_iterations = 0;
  r->error = IAD_NO_ERROR;
```

65. The defaults might be handy

```
⟨ Fill r with reasonable values 63 ⟩ +≡
  if (overwrite_defaults) {
    r->default_a = UNINITIALIZED;
    r->default_b = UNINITIALIZED;
    r->default_g = UNINITIALIZED;
    r->default_ba = UNINITIALIZED;
    r->default_bs = UNINITIALIZED;
    r->default_mua = UNINITIALIZED;
    r->default_mus = UNINITIALIZED;
  }
```

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

```
⟨ Fill r with reasonable values 63 ⟩ +≡
  r->slab.a = 0.5;
  r->slab.b = 1.0;
  r->slab.g = 0;
  r->slab.phase_function = HENYEY_GREENSTEIN;
  r->slab.n_slab = m.slab_index;
  r->slab.n_top_slide = m.slab_top_slide_index;
  r->slab.n_bottom_slide = m.slab_bottom_slide_index;
  r->slab.b_top_slide = m.slab_top_slide_b;
  r->slab.b_bottom_slide = m.slab_bottom_slide_b;
  r->slab.cos_angle = m.slab_cos_angle;
  r->method.a_calc = 0.5;
  r->method.b_calc = 1;
  r->method.g_calc = 0.5;
  r->method.quad_pts = 8;
  r->method.b_thinnest = 1.0/32.0;
```

67. 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 67 ⟩ ≡
  void ez_Inverse_RT(double n, double nslide, double UR1, double UT1, double Tu, double ∗a, double
      ∗b, double ∗g, int ∗error)
```

This code is used in sections 42, 43, and 68.

68. ⟨ Definition for *ez_Inverse_RT* 68 ⟩ ≡

```
  ⟨ Prototype for ez_Inverse_RT 67 ⟩
  {
    struct measure_type m;
    struct invert_type r;

    ∗a = 0;
    ∗b = HUGE_VAL;
    ∗g = 0;
    Initialize_Measure(&m);
    m.slab_index = n;
    m.slab_top_slide_index = nslide;
    m.slab_bottom_slide_index = nslide;
    m.slab_cos_angle = 1.0;
    m.num_measures = 3;
    if (UT1 ≡ 0) m.num_measures −−;
    if (Tu ≡ 0) m.num_measures −−;
    m.m_r = UR1;
    m.m_t = UT1;
    m.m_u = Tu;
    Initialize_Result(m, &r, TRUE);
    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 41.

69. ⟨ Prototype for *Initialize_Measure* 69 ⟩ ≡

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

This code is used in sections 42 and 70.

70. ⟨ Definition for *Initialize_Measure* 70 ⟩ ≡

```
⟨ Prototype for Initialize_Measure 69 ⟩
{
  double default_sphere_d = 8.0 ∗ 25.4;
  double default_sample_d = 0.0 ∗ 25.4;
  double default_detector_d = 0.1 ∗ 25.4;
  double default_entrance_d = 0.5 ∗ 25.4;
  double sphere_area = M_PI ∗ default_sphere_d ∗ default_sphere_d;

  m→slab_index = 1.0;
  m→slab_top_slide_index = 1.0;
  m→slab_top_slide_b = 0.0;
  m→slab_top_slide_thickness = 0.0;
  m→slab_bottom_slide_index = 1.0;
  m→slab_bottom_slide_b = 0.0;
  m→slab_bottom_slide_thickness = 0.0;
  m→slab_thickness = 1.0;
  m→slab_cos_angle = 1.0;
  m→num_spheres = 0;
  m→num_measures = 1;
  m→method = UNKNOWN;
  m→fraction_of_ru_in_mr = 1.0;
  m→fraction_of_tu_in_mt = 1.0;
  m→baffle_r = 1;
  m→baffle_t = 1;
  m→flip_sample = 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 = (M_PI ∗ default_sample_d ∗ default_sample_d/4.0)/sphere_area;
  m→ad_r = (M_PI ∗ default_detector_d ∗ default_detector_d/4.0)/sphere_area;
  m→at_r = (M_PI ∗ default_entrance_d ∗ default_entrance_d/4.0)/sphere_area;
  m→aw_r = 1.0 − m→as_r − m→ad_r − m→at_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→at_t = 0;
  m→aw_t = 1.0 − m→as_t − m→ad_t − m→at_t;
  m→rd_t = 0.0;
  m→rw_t = 1.0;
  m→rstd_t = 1.0;
  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 41.

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

```
⟨ Prototype for Spheres_Inverse_RT 71 ⟩ ≡
  void Spheres_Inverse_RT(double ∗setup, double ∗analysis, double ∗sphere_r, double ∗sphere_t, double
        ∗measurements, double ∗results)
```

This code is used in sections 43 and 72.

72. ⟨ Definition for *Spheres_Inverse_RT* 72 ⟩ ≡

```
  ⟨ Prototype for Spheres_Inverse_RT 71 ⟩
  {
    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 73 ⟩
    ⟨ handle reflection sphere 76 ⟩
    ⟨ handle transmission sphere 77 ⟩
    ⟨ handle measurement 75 ⟩
    Initialize_Result(m, &r, TRUE);
    results[0] = 0;
    results[1] = 0;
    results[2] = 0;
    ⟨ handle analysis 74 ⟩
    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 41.

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

```
⟨ handle setup 73 ⟩ ≡
  {
    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/2.0) ∗ (d_sample_r/m.d_sphere_r/2.0);
    m.at_r = (d_entrance_r/m.d_sphere_r/2.0) ∗ (d_entrance_r/m.d_sphere_r/2.0);
    m.ad_r = (d_detector_r/m.d_sphere_r/2.0) ∗ (d_detector_r/m.d_sphere_r/2.0);
    m.aw_r = 1.0 − m.as_r − m.at_r − m.ad_r;
    m.as_t = (d_sample_t/m.d_sphere_t/2.0) ∗ (d_sample_t/m.d_sphere_t/2.0);
    m.at_t = (d_entrance_t/m.d_sphere_t/2.0) ∗ (d_entrance_t/m.d_sphere_t/2.0);
    m.ad_t = (d_detector_t/m.d_sphere_t/2.0) ∗ (d_detector_t/m.d_sphere_t/2.0);
    m.aw_t = 1.0 − m.as_t − m.at_t − m.ad_t;
    m.slab_bottom_slide_index = m.slab_top_slide_index;
    m.slab_bottom_slide_thickness = m.slab_top_slide_thickness;
    fprintf(stderr, "****␣executing␣FIXME␣****/n");
    m.slab_cos_angle = 1.0;    /∗ FIXME ∗/
  }
```

This code is used in section 72.

74.

```
⟨ handle analysis 74 ⟩ ≡
  r.method.quad_pts = (int) analysis[0];
  mc_runs = (int) analysis[1];
```

This code is used in section 72.

75.

```
⟨ handle measurement 75 ⟩ ≡
  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 72.

76.

```
⟨ handle reflection sphere 76 ⟩ ≡
  m.as_r = sphere_r[0];
  m.at_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 72.

77.

```
⟨ handle transmission sphere 77 ⟩ ≡
  m.as_t = sphere_t[0];
  m.at_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];
```

This code is used in section 72.

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

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

```
⟨ Prototype for Calculate_MR_MT 78 ⟩ ≡
  void Calculate_MR_MT(struct measure_type m, struct invert_type r, int include_MC, int
      include_spheres, double *M_R, double *M_T)
```

This code is used in sections 42 and 79.

79. ⟨ Definition for *Calculate_MR_MT* 79 ⟩ ≡

```
⟨ Prototype for Calculate_MR_MT 78 ⟩
{
  double distance;
  struct measure_type old_mm;
  struct invert_type old_rr;
  if (include_MC ≡ MC_NONE) {
    m.ur1_lost = 0;
    m.ut1_lost = 0;
    m.uru_lost = 0;
    m.utu_lost = 0;
  }
  if (include_MC ≡ MC_REDO) {
    double ur1, ut1, uru, utu;
    long n_photons = 100000;
    MC_Lost(m, r, n_photons, &ur1, &ut1, &uru, &utu, &m.ur1_lost, &m.ut1_lost, &m.uru_lost,
        &m.utu_lost);
  }
  if (¬include_spheres) {
    m.num_spheres = 0;
  }
  Get_Calc_State(&old_mm, &old_rr);
  Set_Calc_State(m, r);
  Calculate_Distance(M_R, M_T, &distance);
  Set_Calc_State(old_mm, old_rr);
}
```

This code is used in section 41.

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

⟨ Prototype for *Calculate_Minimum_MR* 80 ⟩ ≡

```
void Calculate_Minimum_MR(struct measure_type m, struct invert_type r, double *mr, double
    *mt)
```

This code is used in sections 42 and 81.

81. ⟨ Definition for *Calculate_Minimum_MR* 81 ⟩ ≡

```
⟨ Prototype for Calculate_Minimum_MR 80 ⟩
{
  if (m.m_u > 0) r.slab.b = What_Is_B(r.slab, m.m_u);
  else if (r.default_b ≠ UNINITIALIZED) r.slab.b = r.default_b;
  else r.slab.b = HUGE_VAL;
  r.slab.a = 0;
  if (r.default_g ≡ UNINITIALIZED) r.slab.g = 0.0;
  else r.slab.g = r.default_g;
  r.a = r.slab.a;
  r.b = r.slab.b;
  r.g = r.slab.g;
  Calculate_MR_MT(m, r, FALSE, TRUE, mr, mt);
}
```

This code is used in section 41.

82. The minimum possible value of `MR` for a given `MT` will be when the albedo is zero and the maximum value will be when the albedo is one. In the first case there will be no light loss and in the second we will assume that any light loss is neglible (to maximize `MR`).

The second case is perhaps over-simplified. Obviously for a fixed thickness as the albedo increases, the reflectance will increase. So how does *U_Find_B*( ) work when the albedo is set to 1?

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

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

```
⟨ Prototype for MinMax_MR_MT 82 ⟩ ≡
  int MinMax_MR_MT(struct measure_type m, struct invert_type r)
```

This code is used in sections 42 and 83.

83.

```
⟨ Definition for MinMax_MR_MT 83 ⟩ ≡
  ⟨ Prototype for MinMax_MR_MT 82 ⟩
  {
    double distance, measured_m_r, min_possible_m_r, max_possible_m_r, temp_m_t;

    if (m.m_r < 0) return IAD_MR_TOO_SMALL;
    if (m.m_r ∗ m.rstd_r > 1) return IAD_MR_TOO_BIG;
    if (m.m_t < 0) return IAD_MT_TOO_SMALL;
    if (m.m_t ≡ 0) return IAD_NO_ERROR;
    measured_m_r = m.m_r;
    m.m_r = 0;
    r.search = FIND_B;
    if (Debug(DEBUG_ITERATIONS))
      fprintf(stderr, "Determining␣minimum␣possible␣M_R␣for␣given␣M_T\n");
    r.default_a = 0;
    U_Find_B(m, &r);
    Calculate_Distance(&min_possible_m_r, &temp_m_t, &distance);
    if (measured_m_r < min_possible_m_r) return IAD_MR_TOO_SMALL;
    if (Debug(DEBUG_ITERATIONS))
      fprintf(stderr, "Determining␣maximum␣possible␣M_R␣for␣given␣M_T\n");
    r.default_a = 1.0;
    U_Find_B(m, &r);
    Calculate_Distance(&max_possible_m_r, &temp_m_t, &distance);
    if (measured_m_r > max_possible_m_r) return IAD_MR_TOO_BIG;
    return IAD_NO_ERROR;
  }
```

This code is used in section 41.

84.

```
⟨ Prototype for Spheres_Inverse_RT2 84 ⟩ ≡
  void Spheres_Inverse_RT2(double ∗sample, double ∗illumination, double ∗sphere_r, double
      ∗sphere_t, double ∗analysis, double ∗measurement, double ∗a, double ∗b, double ∗g)
```

This code is used in sections 42, 43, and 85.

85. ⟨ Definition for *Spheres_Inverse_RT2* 85 ⟩ ≡

```
⟨ Prototype for Spheres_Inverse_RT2 84 ⟩
{
  struct measure_type m;
  struct invert_type r;
  long num_photons;
  double ur1, ut1, uru, utu;
  int i, mc_runs = 1;
  Initialize_Measure(&m);
  ⟨ handle2 sample 86 ⟩
  ⟨ handle2 illumination 87 ⟩
  ⟨ handle2 reflection sphere 88 ⟩
  ⟨ handle2 transmission sphere 89 ⟩
  ⟨ handle2 analysis 90 ⟩
  ⟨ handle2 measurement 91 ⟩
  Initialize_Result(m, &r, TRUE);
  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) {
    *a = r.a;
    *b = r.b;
    *g = r.g;
  }
}
```

This code is used in section 41.

86. Just move the values from the sample array into the right places

⟨ handle2 sample 86 ⟩ ≡

```
m.slab_index = sample[0];
m.slab_top_slide_index = sample[1];
m.slab_bottom_slide_index = sample[2];
m.slab_thickness = sample[3];
m.slab_top_slide_thickness = sample[4];
m.slab_bottom_slide_thickness = sample[5];
m.slab_top_slide_thickness = 0;
m.slab_bottom_slide_thickness = 0;
```

This code is used in section 85.

87. Just move the values from the illumination array into the right places. Need to spend time to figure out how to integrate items 2, 3, and 4

⟨ handle2 illumination 87 ⟩ ≡

```
m.d_beam = illumination[0];    /* m.lambda = illumination[1]; */    /* m.specular-reflection-excluded
      = illumination[2]; */    /* m.direct-transmission-excluded = illumination[3]; */
    /* m.diffuse-illumination = illumination[4]; */
m.num_spheres = illumination[5];
```

This code is used in section 85.

88.

⟨ handle2 reflection sphere 88 ⟩ ≡

```
{
  double d_sample_r, d_entrance_r, d_detector_r;
  m.d_sphere_r = sphere_r[0];
  d_sample_r = sphere_r[1];
  d_entrance_r = sphere_r[2];
  d_detector_r = sphere_r[3];
  m.rw_r = sphere_r[4];
  m.rd_r = sphere_r[5];
  m.as_r = (d_sample_r/m.d_sphere_r/2.0) ∗ (d_sample_r/m.d_sphere_r/2.0);
  m.at_r = (d_entrance_r/m.d_sphere_r/2.0) ∗ (d_entrance_r/m.d_sphere_r/2.0);
  m.ad_r = (d_detector_r/m.d_sphere_r/2.0) ∗ (d_detector_r/m.d_sphere_r/2.0);
  m.aw_r = 1.0 − m.as_r − m.at_r − m.ad_r;
}
```

This code is used in section 85.

89.

⟨ handle2 transmission sphere 89 ⟩ ≡

```
{
  double d_sample_t, d_entrance_t, d_detector_t;
  m.d_sphere_t = sphere_t[0];
  d_sample_t = sphere_t[1];
  d_entrance_t = sphere_t[2];
  d_detector_t = sphere_t[3];
  m.rw_t = sphere_t[4];
  m.rd_t = sphere_t[5];
  m.as_t = (d_sample_t/m.d_sphere_t/2.0) ∗ (d_sample_t/m.d_sphere_t/2.0);
  m.at_t = (d_entrance_t/m.d_sphere_t/2.0) ∗ (d_entrance_t/m.d_sphere_t/2.0);
  m.ad_t = (d_detector_t/m.d_sphere_t/2.0) ∗ (d_detector_t/m.d_sphere_t/2.0);
  m.aw_t = 1.0 − m.as_t − m.at_t − m.ad_t;
}
```

This code is used in section 85.

90.

⟨ handle2 analysis 90 ⟩ ≡

```
r.method.quad_pts = (int) analysis[0];
mc_runs = (int) analysis[1];
num_photons = (long) analysis[2];
```

This code is used in section 85.

91.

⟨ handle2 measurement 91 ⟩ ≡

```
m.rstd_r = measurement[0];
m.m_r = measurement[1];
m.m_t = measurement[2];
m.m_u = measurement[3];
m.num_measures = 3;
if (m.m_t ≡ 0) m.num_measures −−;
if (m.m_u ≡ 0) m.num_measures −−;
thirdthird
```

This code is used in section 85.

92. IAD Input Output.

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

```
⟨ iad_io.c 92 ⟩ ≡
#define _CRT_SECURE_NO_WARNINGS
#define MAX_COLUMNS 256
  char COLUMN_LABELS[MAX_COLUMNS] = "";
#include <string.h>
#include <stdio.h>
#include <stdlib.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 104 ⟩
  ⟨ Definition for read_number 106 ⟩
  ⟨ Definition for check_magic 108 ⟩
  ⟨ Definition for remove_whitespace 117 ⟩
  ⟨ Definition for remove_comment 118 ⟩
  ⟨ Definition for remove_first_char 119 ⟩
  ⟨ Definition for print_maybe 120 ⟩
  ⟨ Definition for Read_Data_Legend 122 ⟩
  ⟨ Definition for Read_Data_Line_Per_Labels 102 ⟩
  ⟨ Definition for Read_Header 96 ⟩
  ⟨ Definition for Write_Header 110 ⟩
  ⟨ Definition for Read_Data_Line 101 ⟩
```

93. ⟨ iad_io.h 93 ⟩ ≡

```
  ⟨ Prototype for Read_Header 95 ⟩;
  ⟨ Prototype for Write_Header 109 ⟩;
  ⟨ Prototype for Read_Data_Line 100 ⟩;
```

94. Reading the file header.

95. ⟨ Prototype for *Read_Header* 95 ⟩ ≡

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

This code is used in sections 93 and 96.

96. 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* 96 ⟩ ≡

```
⟨ Prototype for Read_Header 95 ⟩
{
  double x;

  Initialize_Measure(m);
  if (check_magic(fp)) return 1;
  if (read_number(fp, &m→slab_index)) return 1;
  if (read_number(fp, &m→slab_top_slide_index)) return 1;
  if (read_number(fp, &m→slab_thickness)) return 1;
  if (read_number(fp, &m→slab_top_slide_thickness)) return 1;
  if (read_number(fp, &m→d_beam)) return 1;
  if (m→slab_top_slide_thickness ≡ 0.0) m→slab_top_slide_index = 1.0;
  if (m→slab_top_slide_index ≡ 1.0) m→slab_top_slide_thickness = 0.0;
  if (m→slab_top_slide_index ≡ 0.0) {
    m→slab_top_slide_thickness = 0.0;
    m→slab_top_slide_index = 1.0;
  }
  m→slab_bottom_slide_index = m→slab_top_slide_index;
  m→slab_bottom_slide_thickness = m→slab_top_slide_thickness;
  if (read_number(fp, &m→rstd_r)) return 1;
  if (read_number(fp, &x)) return 1;
  m→num_spheres = (int) x;
  m→method = SUBSTITUTION;
  ⟨ Read coefficients for reflection sphere 97 ⟩
  ⟨ Read coefficients for transmission sphere 98 ⟩
  ⟨ Read info about measurements 99 ⟩
  return 0;
}
```

This code is used in section 92.

97. ⟨ Read coefficients for reflection sphere 97 ⟩ ≡

```
{
  double d_sample_r, d_third_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_third_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/2.0) ∗ (d_sample_r/m→d_sphere_r/2.0);
  m→at_r = (d_third_r/m→d_sphere_r/2.0) ∗ (d_third_r/m→d_sphere_r/2.0);
  m→ad_r = (d_detector_r/m→d_sphere_r/2.0) ∗ (d_detector_r/m→d_sphere_r/2.0);
  m→aw_r = 1.0 − m→as_r − m→at_r − m→ad_r;
}
```

This code is used in section 96.

98. ⟨ Read coefficients for transmission sphere 98 ⟩ ≡

```
{
  double d_sample_t, d_third_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_third_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/2.0) ∗ (d_sample_t/m→d_sphere_t/2.0);
  m→at_t = (d_third_t/m→d_sphere_t/2.0) ∗ (d_third_t/m→d_sphere_t/2.0);
  m→ad_t = (d_detector_t/m→d_sphere_t/2.0) ∗ (d_detector_t/m→d_sphere_t/2.0);
  m→aw_t = 1.0 − m→as_t − m→at_t − m→ad_t;
}
```

This code is used in section 96.

99. ⟨ Read info about measurements 99 ⟩ ≡

```
∗params = Read_Data_Legend(fp);
if (COLUMN_LABELS[0] ≠ '\0') {
  m→num_measures = 0;
  if (strchr(COLUMN_LABELS, 'r')) m→num_measures++;
  if (strchr(COLUMN_LABELS, 't')) m→num_measures++;
  if (strchr(COLUMN_LABELS, 'u')) m→num_measures++;
  if (m→num_measures ≡ 0) {
    fprintf(stderr, "Column␣labels␣must␣have␣at␣least␣one␣'r',␣'t',␣or␣'u'\n");
    fprintf(stderr, "Column␣labels␣=␣'%s'\n", COLUMN_LABELS);
    exit(EXIT_FAILURE);
  }
}
else m→num_measures = (∗params ≥ 3) ? 3 : ∗params;
```

This code is used in section 96.

100. 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 100 ⟩ ≡
  int Read_Data_Line(FILE *fp, struct measure_type *m, struct invert_type *r, int params)
```

This code is used in sections 93 and 101.

101.

```
⟨ Definition for Read_Data_Line 101 ⟩ ≡
  ⟨ Prototype for Read_Data_Line 100 ⟩
  {
    if (strlen(COLUMN_LABELS) > 0) return Read_Data_Line_Per_Labels(fp, m, r, params);
    if (read_number(fp, &m→m_r)) return 1;
    if (m→m_r > 1) {
      m→lambda = m→m_r;
      if (read_number(fp, &m→m_r)) return 1;
    }
    if (params ≡ 1) return 0;
    if (read_number(fp, &m→m_t)) return 1;
    if (params ≡ 2) return 0;
    if (read_number(fp, &m→m_u)) return 1;
    if (params ≡ 3) return 0;
    if (read_number(fp, &m→rw_r)) return 1;
    m→rw_t = m→rw_r;
    if (params ≡ 4) return 0;
    if (read_number(fp, &m→rw_t)) return 1;
    if (params ≡ 5) return 0;
    if (read_number(fp, &m→rstd_r)) return 1;
    if (params ≡ 6) return 0;
    if (read_number(fp, &m→rstd_t)) return 1;
    return 0;
  }
```

This code is used in section 92.

102. ⟨ Definition for *Read_Data_Line_Per_Labels* 102 ⟩ ≡

```
int Read_Data_Line_Per_Labels(FILE *fp, struct measure_type *m, struct invert_type *r, int
        params)
{
  int count = 0;
  double x;

  while (count < params) {
    if (read_number(fp, &x)) return 1;
    char c = COLUMN_LABELS[count];
    if (FALSE) fprintf(stderr, "count␣=␣%2d,␣option␣=␣%c,␣value␣=␣%10.5f\n", count, c, x);
    switch (c) {
    case 'a': r→default_a = x;
      break;
    case 'A': r→default_mua = x;
      r→default_ba = x * m→slab_thickness;
      break;
    case 'b': r→default_b = x;
      break;
    case 'B': m→d_beam = x;
      break;
    case 'c': m→fraction_of_ru_in_mr = x;
      break;
    case 'C': m→fraction_of_tu_in_mt = x;
      break;
    case 'd': m→slab_thickness = x;
      break;
    case 'D': m→slab_top_slide_thickness = x;
      m→slab_bottom_slide_thickness = x;
      break;
    case 'e': r→tolerance = x;
      r→MC_tolerance = x;
      break;
    case 'E': m→slab_bottom_slide_b = x;
      m→slab_top_slide_b = x;
      break;
    case 'F': r→default_mus = x;
      r→default_bs = x * m→slab_thickness;
      break;
    case 'g': r→default_g = x;
      break;
    case 'L': m→lambda = x;
      break;
    case 'M': m→num_spheres = (int) x;
      break;
    case 'n': m→slab_index = x;
      break;
    case 'N': m→slab_top_slide_index = x;
      m→slab_bottom_slide_index = x;
      break;
    case 'q': r→method.quad_pts = (int) x;
      break;
    case 'r': m→m_r = x;
      break;
    case 'R': m→rstd_r = x;
      break;
    case 't': m→m_t = x;
      break;
    case 'S': m→num_spheres = (int) x;
      break;
    case 'T': m→rstd_t = x;
      break;
    case 'u': m→m_u = x;
      break;
    case 'w': m→rw_r = x;
      break;
    case 'W': m→rw_t = x;
      break;
    default: fprintf(stderr, "legend␣variable␣'%c'␣unimplemented", c);
      return 1;
    }
    count++;
  }
  return 0;
}
```

This code is used in section 92.

103. 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 103⟩ ≡
  int skip_white(FILE *fp)
```

This code is used in section 104.

104.

```
⟨Definition for skip_white 104⟩ ≡
  ⟨Prototype for skip_white 103⟩
  {
    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 92.

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

```
⟨Prototype for read_number 105⟩ ≡
  int read_number(FILE *fp, double *x)
```

This code is used in section 106.

106. ⟨ Definition for *read_number* 106 ⟩ ≡

```
⟨ Prototype for read_number 105 ⟩
{
  if (skip_white(fp)) return 1;
  if (fscanf(fp, "%lf", x)) return 0;
  else return 1;
}
```

This code is used in section 92.

107. 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* 107 ⟩ ≡

```
  int check_magic(FILE *fp)
```

This code is used in section 108.

108. ⟨ Definition for *check_magic* 108 ⟩ ≡

```
⟨ Prototype for check_magic 107 ⟩
{
  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 92.

109. Formatting the header information.

```
⟨ Prototype for Write_Header 109 ⟩ ≡
  void Write_Header(struct measure_type m, struct invert_type r, int params, char *cmd)
```

This code is used in sections 93 and 110.

110. ⟨ Definition for *Write_Header* 110 ⟩ ≡

```
  ⟨ Prototype for Write_Header 109 ⟩
  {
    ⟨ Write slab info 111 ⟩
    ⟨ Write irradiation info 112 ⟩
    ⟨ Write general sphere info 113 ⟩
    ⟨ Write first sphere info 114 ⟩
    ⟨ Write second sphere info 115 ⟩
    ⟨ Write measure and inversion info 116 ⟩
  }
```

This code is used in section 92.

111. ⟨ Write slab info 111 ⟩ ≡

```
  double xx;
  printf("#␣Inverse␣Adding-Doubling␣%s␣\n", Version);
  printf("#␣%s\n", cmd);
  printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Beam␣diameter␣=␣");
  print_maybe('B', "%7.1f␣mm\n", m.d_beam);
  printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Sample␣thickness␣=␣");
  print_maybe('d', "%7.3f␣mm\n", m.slab_thickness);
  printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Top␣slide␣thickness␣=␣");
  print_maybe('D', "%7.3f␣mm\n", m.slab_top_slide_thickness);
  printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Bottom␣slide␣thickness␣=␣");
  print_maybe('D', "%7.3f␣mm\n", m.slab_bottom_slide_thickness);
  printf("#␣␣␣␣␣␣␣␣␣␣␣Sample␣index␣of␣refraction␣=␣");
  print_maybe('n', "%7.4f␣mm\n", m.slab_index);
  printf("#␣␣␣␣␣␣␣␣Top␣slide␣index␣of␣refraction␣=␣");
  print_maybe('N', "%7.4f␣mm\n", m.slab_top_slide_index);
  printf("#␣␣␣␣␣Bottom␣slide␣index␣of␣refraction␣=␣");
  print_maybe('N', "%7.4f␣mm\n", m.slab_bottom_slide_index);
```

This code is used in section 110.

112. ⟨ Write irradiation info 112 ⟩ ≡

```
  printf("#␣\n");
```

This code is used in section 110.

113. ⟨ Write general sphere info 113 ⟩ ≡

```
  printf("#␣␣Percentage␣unscattered␣refl.␣in␣M_R␣=␣");
  print_maybe('c', "%7.1f␣%%\n", m.fraction_of_ru_in_mr * 100);
  printf("#␣Percentage␣unscattered␣trans.␣in␣M_T␣=␣");
  print_maybe('C', "%7.1f␣%%\n", m.fraction_of_tu_in_mt * 100);
  printf("#␣\n");
```

This code is used in section 110.

114. ⟨ Write first sphere info 114 ⟩ ≡

```
printf("#␣Reflection␣sphere");
if (m.baffle_r) printf("␣has␣a␣baffle␣between␣sample␣and␣detector");
else printf("␣has␣no␣baffle␣between␣sample␣and␣detector");
if (m.num_spheres > 0) printf("\n");
else printf("␣(ignored␣since␣no␣spheres␣used)\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.at_r));
printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣detector␣port␣diameter␣=␣%7.1f␣mm\n", 2 * m.d_sphere_r * sqrt(m.ad_r));
printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣detector␣reflectance␣=␣%7.1f␣%%\n", m.rd_r * 100);
printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣wall␣reflectance␣=␣");
print_maybe('w', "%7.1f␣%%\n", m.rw_r * 100);
printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣calibration␣standard␣=␣");
print_maybe('R', "%7.1f␣%%\n", m.rstd_r * 100);
printf("#\n");
```

This code is used in section 110.

115. ⟨ Write second sphere info 115 ⟩ ≡

```
printf("#␣Transmission␣sphere");
if (m.baffle_t) printf("␣has␣a␣baffle␣between␣sample␣and␣detector");
else printf("␣has␣no␣baffle␣between␣sample␣and␣detector");
if (m.num_spheres > 0) printf("\n");
else printf("␣(ignored␣since␣no␣spheres␣used)\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("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣third␣port␣diameter␣=␣%7.1f␣mm\n", 2 * m.d_sphere_r * sqrt(m.at_t));
printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣detector␣port␣diameter␣=␣%7.1f␣mm\n", 2 * m.d_sphere_r * sqrt(m.ad_t));
printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣detector␣reflectance␣=␣%7.1f␣%%\n", m.rd_t * 100);
if (m.at_t ≡ 0) printf("#␣␣␣␣wall␣reflectance␣and␣cal␣standard␣=␣");
else printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣wall␣reflectance␣=␣");
print_maybe('w', "%7.1f␣%%\n", m.rw_t * 100);
printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣calibration␣standard␣=␣%7.1f␣%%", m.rstd_t * 100);
if (m.at_t ≡ 0) printf("␣(ignored)");
printf("\n");
```

This code is used in section 110.

116. ⟨ Write measure and inversion info 116 ⟩ ≡

```
printf("#\n");
if (COLUMN_LABELS[0] ≡ '\0') {
  switch (params) {
  case −1: printf("#␣No␣M_R␣or␣M_T␣--␣forward␣calculation.\n");
    break;
  case 1: printf("#␣Just␣M_R␣was␣measured");
    break;
  case 2: printf("#␣M_R␣and␣M_T␣were␣measured");
    break;
  case 3: printf("#␣M_R,␣M_T,␣and␣M_U␣were␣measured");
    break;
  case 4: printf("#␣M_R,␣M_T,␣M_U,␣and␣r_w␣were␣measured");
    break;
  case 5: printf("#␣M_R,␣M_T,␣M_U,␣r_w,␣and␣t_w␣were␣measured");
    break;
  case 6: printf("#␣M_R,␣M_T,␣M_U,␣r_w,␣t_w,␣and␣r_std␣were␣measured");
    break;
  case 7: printf("#␣M_R,␣M_T,␣M_U,␣r_w,␣t_w,␣r_std␣and␣t_std␣were␣measured");
    break;
  default: printf("#␣Something␣went␣wrong␣...␣measures␣should␣be␣1␣to␣7!\n");
    break;
  }
}
else {
  int i;

  printf("#␣%d␣input␣columns␣with␣LABELS:", params);
  for (i = 0; i < params; i++) {
    printf("␣%c␣", COLUMN_LABELS[i]);
  }
}
if (m.flip_sample) printf("␣(sample␣flipped)␣");
switch (m.method) {
case UNKNOWN: printf("␣using␣an␣unknown␣method.\n");
  break;
case SUBSTITUTION: printf("␣using␣the␣substitution␣(single-beam)␣method.\n");
  break;
case COMPARISON: printf("␣using␣the␣comparison␣(dual-beam)␣method.\n");
}
switch (m.num_spheres) {
case 0: printf("#␣No␣sphere␣corrections␣were␣used");
  break;
case 1:
  if (m.method ≡ COMPARISON) printf("#␣No␣sphere␣corrections␣were␣needed");
  else printf("#␣Single␣sphere␣corrections␣were␣used");
  break;
case 2: printf("#␣Double␣sphere␣corrections␣were␣used");
  break;
}
printf("␣and␣light␣was␣incident␣at␣%d␣degrees␣from␣the␣normal",
    (int)(acos(m.slab_cos_angle) ∗ 57.2958));
printf(".\n");
```

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

This code is used in section 110.

117. Discard white space and dashes in the legend string

```
⟨ Definition for remove_whitespace 117 ⟩ ≡
  void remove_whitespace(char *str)
  {
    int i, j = 0;
    for (i = 0; str[i] ≠ '\0'; i++) {
      if (¬isspace(str[i]) ∧ str[i] ≠ '-') {
        str[j++] = str[i];
      }
    }
    str[j] = '\0';
  }
```

This code is used in section 92.

118. ⟨ Definition for *remove_comment* 118 ⟩ ≡

```
  void remove_comment(char *str)
  {
    int i;
    for (i = 0; str[i] ≠ '\0'; i++) {
      if (str[i] ≡ '#') {
        str[i] = '\0';
        break;
      }
    }
  }
```

This code is used in section 92.

119. ⟨ Definition for *remove_first_char* 119 ⟩ ≡

```
  void remove_first_char(char *str)
  {
    int len = strlen(str);
    if (len > 0) {
      for (int i = 0; i < len; i++) {
        str[i] = str[i + 1];
      }
    }
  }
```

This code is used in section 92.

120. ⟨ Definition for *print_maybe* 120 ⟩ ≡

```
  void print_maybe(char c, char *format, double x)
  {
    char *result = strchr(COLUMN_LABELS, c);
    if (result ≡ Λ) printf(format, x);
    else printf("␣(varies␣with␣input␣row)\n");
  }
```

This code is used in section 92.

121. ⟨ Prototype for *Read_Data_Legend* 121 ⟩ ≡

```
  int Read_Data_Legend(FILE *fp)
```

This code is used in section 122.

122. ⟨ Definition for *Read_Data_Legend* 122 ⟩ ≡

```
⟨ Prototype for Read_Data_Legend 121 ⟩
{
  int n = 0;
  char c;

  skip_white(fp);
  if (fgets(COLUMN_LABELS, MAX_COLUMNS, fp) ≡ Λ) {
    fprintf(stderr, "could␣not␣read␣Data␣Legend␣String␣in␣file\n");
    exit(EXIT_FAILURE);
  }
  remove_whitespace(COLUMN_LABELS);
  remove_comment(COLUMN_LABELS);
  c = COLUMN_LABELS[0];
  if (c ≡ '1' ∨ c ≡ '2' ∨ c ≡ '3' ∨ c ≡ '4' ∨ c ≡ '5' ∨ c ≡ '6' ∨ c ≡ '7') {
    n = COLUMN_LABELS[0] − '0';
    COLUMN_LABELS[0] = '\0';
  }
  else {
    n = strlen(COLUMN_LABELS);
  }
  return n;
}
```

This code is used in section 92.

123. IAD Calculation.

```
⟨ iad_calc.c 123 ⟩ ≡
#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 · 10^-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 201
#define T_TRUST_FACTOR 1
#define MAX_ABS_G 0.999999
#define SWAP(a, b)
  {
    double swap = (a);
    (a) = (b);
    (b) = swap;
  }
  static int CALCULATING_GRID = 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_Calc_State 139 ⟩
  ⟨ Definition for Get_Calc_State 141 ⟩
  ⟨ Definition for Same_Calc_State 143 ⟩
  ⟨ Prototype for Fill_AB_Grid 161 ⟩;
  ⟨ Prototype for Fill_AG_Grid 165 ⟩;
```

⟨ Definition for *RT_Flip* 159 ⟩
⟨ Definition for *Allocate_Grid* 145 ⟩
⟨ Definition for *Valid_Grid* 149 ⟩
⟨ Definition for *fill_grid_entry* 160 ⟩
⟨ Definition for *Fill_Grid* 175 ⟩
⟨ Definition for *Near_Grid_Points* 157 ⟩
⟨ Definition for *Fill_AB_Grid* 162 ⟩
⟨ Definition for *Fill_AG_Grid* 166 ⟩
⟨ Definition for *Fill_BG_Grid* 169 ⟩
⟨ Definition for *Fill_BaG_Grid* 171 ⟩
⟨ Definition for *Fill_BsG_Grid* 173 ⟩
⟨ Definition for *Grid_ABG* 147 ⟩
⟨ Definition for *Gain* 128 ⟩
⟨ Definition for *Gain_11* 130 ⟩
⟨ Definition for *Gain_22* 132 ⟩
⟨ Definition for *Two_Sphere_R* 134 ⟩
⟨ Definition for *Two_Sphere_T* 136 ⟩
⟨ Definition for *Calculate_Distance_With_Corrections* 181 ⟩
⟨ Definition for *Calculate_Grid_Distance* 179 ⟩
⟨ Definition for *Calculate_Distance* 177 ⟩
⟨ Definition for *abg_distance* 155 ⟩
⟨ Definition for *Find_AG_fn* 206 ⟩
⟨ Definition for *Find_AB_fn* 208 ⟩
⟨ Definition for *Find_Ba_fn* 210 ⟩
⟨ Definition for *Find_Bs_fn* 212 ⟩
⟨ Definition for *Find_A_fn* 214 ⟩
⟨ Definition for *Find_B_fn* 216 ⟩
⟨ Definition for *Find_G_fn* 218 ⟩
⟨ Definition for *Find_BG_fn* 220 ⟩
⟨ Definition for *Find_BaG_fn* 222 ⟩
⟨ Definition for *Find_BsG_fn* 224 ⟩
⟨ Definition for *maxloss* 226 ⟩
⟨ Definition for *Max_Light_Loss* 228 ⟩

124.

```
⟨ iad_calc.h 124 ⟩ ≡
  ⟨ Prototype for Gain 127 ⟩;
  ⟨ Prototype for Gain_11 129 ⟩;
  ⟨ Prototype for Gain_22 131 ⟩;
  ⟨ Prototype for Two_Sphere_R 133 ⟩;
  ⟨ Prototype for Two_Sphere_T 135 ⟩;
  ⟨ Prototype for Set_Calc_State 138 ⟩;
  ⟨ Prototype for Get_Calc_State 140 ⟩;
  ⟨ Prototype for Same_Calc_State 142 ⟩;
  ⟨ Prototype for Valid_Grid 148 ⟩;
  ⟨ Prototype for Allocate_Grid 144 ⟩;
  ⟨ Prototype for Fill_Grid 174 ⟩;
  ⟨ Prototype for Near_Grid_Points 156 ⟩;
  ⟨ Prototype for Grid_ABG 146 ⟩;
  ⟨ Prototype for Find_AG_fn 205 ⟩;
  ⟨ Prototype for Find_AB_fn 207 ⟩;
  ⟨ Prototype for Find_Ba_fn 209 ⟩;
  ⟨ Prototype for Find_Bs_fn 211 ⟩;
  ⟨ Prototype for Find_A_fn 213 ⟩;
  ⟨ Prototype for Find_B_fn 215 ⟩;
  ⟨ Prototype for Find_G_fn 217 ⟩;
  ⟨ Prototype for Find_BG_fn 219 ⟩;
  ⟨ Prototype for Find_BsG_fn 223 ⟩;
  ⟨ Prototype for Find_BaG_fn 221 ⟩;
  ⟨ Prototype for Fill_BG_Grid 168 ⟩;
  ⟨ Prototype for Fill_BsG_Grid 172 ⟩;
  ⟨ Prototype for Fill_BaG_Grid 170 ⟩;
  ⟨ Prototype for Calculate_Distance_With_Corrections 180 ⟩;
  ⟨ Prototype for Calculate_Distance 176 ⟩;
  ⟨ Prototype for Calculate_Grid_Distance 178 ⟩;
  ⟨ Prototype for abg_distance 154 ⟩;
  ⟨ Prototype for maxloss 225 ⟩;
  ⟨ Prototype for Max_Light_Loss 227 ⟩;
  ⟨ Prototype for RT_Flip 158 ⟩;
```

125. 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.

126. Gain.

Assume that a sphere is illuminated with diffuse light having a power P. This light will undergo multiple reflections in the sphere walls that will increase the power falling on the detector. The gain on the detector due to integrating sphere effects varies with the presence of a baffle between the sample and the detector. If a baffle is present then

$$G_{\text{no baffle}}(r_s, r_t) = \frac{1}{1 - a_w r_w - a_d r_d - a_s r_s - a_s r_t}$$

or with a baffle as

$$G_{\text{baffle}}(r_s, r_t) = \frac{1}{1 - a_w r'_w - r'_w(1 - a_t)(a_d r_d + a_s r_s)}$$

where

$$r'_w = r_w + (a_t/a_w) r_t$$

For a black sphere the gain does not depend on the diffuse reflectivity of the sample and is unity. $G(r_s) = 1$, which is easily verified by setting $r_w = 0$.

The value *uru_sample* is the total reflectance from the sample for diffuse incident light and *uru_third* is the total reflectance from the third port for diffuse incident light. For a reflection sphere, the third port is the entrance port and *uru_third* $= 0$.

127. ⟨ Prototype for *Gain* 127 ⟩ ≡

```
double Gain(int sphere, struct measure_type m, double uru_sample, double uru_third)
```

This code is used in sections 124 and 128.

128. ⟨ Definition for *Gain* 128 ⟩ ≡

```
⟨ Prototype for Gain 127 ⟩
{
  double inv_gain;
  if (sphere ≡ REFLECTION_SPHERE) {
    if (m.baffle_r) {
      inv_gain = m.rw_r + (m.at_r/m.aw_r) ∗ uru_third;
      inv_gain ∗= m.aw_r + (1 − m.at_r) ∗ (m.ad_r ∗ m.rd_r + m.as_r ∗ uru_sample);
      inv_gain = 1.0 − inv_gain;
    }
    else {
      inv_gain = 1.0 − m.aw_r ∗ m.rw_r − m.ad_r ∗ m.rd_r − m.as_r ∗ uru_sample − m.at_r ∗ uru_third;
    }
  }
  else if (m.baffle_t) {
    inv_gain = m.rw_t + (m.at_t/m.aw_t) ∗ uru_third;
    inv_gain ∗= m.aw_t + (1 − m.at_t) ∗ (m.ad_t ∗ m.rd_t + m.as_t ∗ uru_sample);
    inv_gain = 1.0 − inv_gain;
  }
  else {
    inv_gain = 1.0 − m.aw_t ∗ m.rw_t − m.ad_t ∗ m.rd_t − m.as_t ∗ uru_sample − m.at_t ∗ uru_third;
  }
  return 1.0/inv_gain;
}
```

This code is used in section 123.

129. 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

$$G_{1\to1}(r_s,t_s) = \frac{G(r_s)}{1 - a_s a_s' r_w r_w' (1-a_t)(1-a_t')G(r_s)G'(r_s)t_s^2}$$

⟨ Prototype for *Gain_11* 129 ⟩ ≡

```
double Gain_11(struct measure_type m, double URU, double tdiffuse)
```

This code is used in sections 124 and 130.

130. ⟨ Definition for *Gain_11* 130 ⟩ ≡

```
⟨ Prototype for Gain_11 129 ⟩
{
  double G, GP, G11;
  G = Gain(REFLECTION_SPHERE, m, URU, 0);
  GP = Gain(TRANSMISSION_SPHERE, m, URU, 0);
  G11 = G/(1 − m.as_r ∗ m.as_t ∗ m.aw_r ∗ m.aw_t ∗ (1 − m.at_r) ∗ (1 − m.at_t) ∗ G ∗ GP ∗ tdiffuse ∗ tdiffuse);
  return G11;
}
```

This code is used in section 123.

131. 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

$$G_{2\to2}(r_s,t_s) = \frac{G'(r_s)}{1 - a_s a_s' r_w r_w' (1-a_t)(1-a_t')G(r_s)G'(r_s)t_s^2}$$

⟨ Prototype for *Gain_22* 131 ⟩ ≡

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

This code is used in sections 124 and 132.

132. ⟨ Definition for *Gain_22* 132 ⟩ ≡

```
⟨ Prototype for Gain_22 131 ⟩
{
  double G, GP, G22;
  G = Gain(REFLECTION_SPHERE, m, URU, 0);
  GP = Gain(TRANSMISSION_SPHERE, m, URU, 0);
  G22 = GP/(1 − m.as_r ∗ m.as_t ∗ m.aw_r ∗ m.aw_t ∗ (1 − m.at_r) ∗ (1 − m.at_t) ∗ G ∗ GP ∗ tdiffuse ∗ tdiffuse);
  return G22;
}
```

This code is used in section 123.

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

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

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

which simplifies slightly to

$$\begin{aligned}R(r_s^{\text{direct}},r_s,t_s^{\text{direct}},t_s) &= a_d(1-a_t)r_w P\cdot G_{1\to 1}(r_s,t_s)\\ &\quad\times\Big[(1-f)r_s^{\text{direct}} + fr_w + (1-f)a'_s(1-a'_t)r'_w t_s^{\text{direct}} t_s G'(r_s)\Big]\end{aligned}$$

⟨ Prototype for *Two_Sphere_R* 133 ⟩ ≡
```
double Two_Sphere_R(struct measure_type m, double UR1, double URU, double UT1, double UTU)
```
This code is used in sections 124 and 134.

134. ⟨ Definition for *Two_Sphere_R* 134 ⟩ ≡
```
⟨ Prototype for Two_Sphere_R 133 ⟩
{
  double x, GP;
  GP = Gain(TRANSMISSION_SPHERE, m, URU, 0);
  x = m.ad_r ∗ (1 − m.at_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.at_t) ∗ m.rw_t ∗ UT1 ∗ UTU ∗ GP;
  return x;
}
```
This code is used in section 123.

135. 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

$$\begin{aligned}T(r_s^{\text{direct}},r_s,t_s^{\text{direct}},t_s) &= G_{1\to 2}(r_s,t_s)\cdot a'_d(1-a_t)r_w^2 fP\\ &\quad + G_{1\to 2}(r_s,t_s)\cdot a'_d(1-a_t)r_w(1-f)r_s^{\text{direct}}P\\ &\quad + G_{2\to 2}(r_s,t_s)\cdot a'_d(1-a'_t)r'_w(1-f)t_s^{\text{direct}}P\end{aligned}$$

or

$$\begin{aligned}T(r_s^{\text{direct}},r_s,t_s^{\text{direct}},t_s) &= a'_d(1-a'_t)r'_w P\cdot G_{2\to 2}(r_s,t_s)\\ &\quad\times\Big[(1-f)t_s^{\text{direct}} + (1-a_t)r_w a_s t_s(fr_w + (1-f)r_s^{\text{direct}})G(r_s)\Big]\end{aligned}$$

⟨ Prototype for *Two_Sphere_T* 135 ⟩ ≡
```
double Two_Sphere_T(struct measure_type m, double UR1, double URU, double UT1, double UTU)
```
This code is used in sections 124 and 136.

136.

```
⟨Definition for Two_Sphere_T 136⟩ ≡
  ⟨Prototype for Two_Sphere_T 135⟩
  {
    double x, G;
    G = Gain(REFLECTION_SPHERE, m, URU, 0);
    x = m.ad_t ∗ (1 − m.at_t) ∗ m.rw_t ∗ Gain_22(m, URU, UTU);
    x ∗= (1 − m.f_r) ∗ UT1 + (1 − m.at_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 123.

137. 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.

138. 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 138 ⟩ ≡
  void Set_Calc_State(struct measure_type m, struct invert_type r)
```

This code is used in sections 124 and 139.

139.
```
⟨ Definition for Set_Calc_State 139 ⟩ ≡
  ⟨ Prototype for Set_Calc_State 138 ⟩
  {
    memcpy(&MM, &m, sizeof(struct measure_type));
    memcpy(&RR, &r, sizeof(struct invert_type));
  }
```

This code is used in section 123.

140. 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 140 ⟩ ≡
  void Get_Calc_State(struct measure_type *m, struct invert_type *r)
```

This code is used in sections 124 and 141.

141.
```
⟨ Definition for Get_Calc_State 141 ⟩ ≡
  ⟨ Prototype for Get_Calc_State 140 ⟩
  {
    memcpy(m, &MM, sizeof(struct measure_type));
    memcpy(r, &RR, sizeof(struct invert_type));
  }
```

This code is used in section 123.

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

```
⟨ Prototype for Same_Calc_State 142 ⟩ ≡
  boolean_type Same_Calc_State(struct measure_type m, struct invert_type r)
```

This code is used in sections 124 and 143.

143. ⟨ Definition for *Same_Calc_State* 143 ⟩ ≡

```
⟨ Prototype for Same_Calc_State 142 ⟩
{
  if (The_Grid ≡ Λ) return FALSE;
  if (¬The_Grid_Initialized) return FALSE;
  if (r.search ≠ RR.search) return FALSE;
  if (r.method.quad_pts ≠ RR.method.quad_pts) return FALSE;
  if (r.slab.a ≠ RR.slab.a) return FALSE;
  if (r.slab.b ≠ RR.slab.b) return FALSE;
  if (r.slab.g ≠ RR.slab.g) return FALSE;
  if (r.slab.phase_function ≠ RR.slab.phase_function) return FALSE;
  if (r.slab.n_slab ≠ RR.slab.n_slab) return FALSE;
  if (r.slab.n_top_slide ≠ RR.slab.n_top_slide) return FALSE;
  if (r.slab.n_bottom_slide ≠ RR.slab.n_bottom_slide) return FALSE;
  if (r.slab.b_top_slide ≠ RR.slab.b_top_slide) return FALSE;
  if (r.slab.b_bottom_slide ≠ RR.slab.b_bottom_slide) return FALSE;
  if (r.slab.cos_angle ≠ RR.slab.cos_angle) return FALSE;
  if ((m.num_measures ≡ 3) ∧ (m.m_u ≠ MGRID.m_u)) return (FALSE);
  return TRUE;
}
```

This code is used in section 123.

144. ⟨ Prototype for *Allocate_Grid* 144 ⟩ ≡

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

This code is used in sections 124 and 145.

145. ⟨ Definition for *Allocate_Grid* 145 ⟩ ≡

```
⟨ Prototype for Allocate_Grid 144 ⟩
{
  (void) s;
  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 123.

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

⟨ Prototype for *Grid_ABG* 146 ⟩ ≡

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

This code is used in sections 124 and 147.

147. ⟨ Definition for *Grid_ABG* 147 ⟩ ≡

```
⟨ Prototype for Grid_ABG 146 ⟩
{
  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 123.

148. 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* 148 ⟩ ≡

```
  boolean_type Valid_Grid(struct measure_type m, struct invert_type r)
```

This code is used in sections 124 and 149.

149. ⟨ Definition for *Valid_Grid* 149 ⟩ ≡

```
⟨ Prototype for Valid_Grid 148 ⟩
{
  int s = r.search;

  ⟨ Tests for invalid grid 150 ⟩
  return (TRUE);
}
```

This code is used in section 123.

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

⟨ Tests for invalid grid 150 ⟩ ≡

```
  if (The_Grid ≡ Λ) {
    if (Debug(DEBUG_GRID)) fprintf(stderr, "GRID:␣Fill␣because␣NULL\n");
    return (FALSE);
  }
  if (¬The_Grid_Initialized) {
    if (Debug(DEBUG_GRID)) fprintf(stderr, "GRID:␣Fill␣because␣not␣initialized\n");
    return (FALSE);
  }
```

See also sections 151, 152, and 153.

This code is used in section 149.

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

```
⟨ Tests for invalid grid 150 ⟩ +≡
  if (The_Grid_Search ≠ s) {
    if (Debug(DEBUG_GRID)) fprintf(stderr, "GRID:␣Fill␣because␣search␣type␣changed\n");
    return (FALSE);
  }
```

152. Compare the *m.m_u* value only if there are three measurements

```
⟨ Tests for invalid grid 150 ⟩ +≡
  if ((m.num_measures ≡ 3) ∧ (m.m_u ≠ MGRID.m_u)) {
    if (Debug(DEBUG_GRID)) fprintf(stderr, "GRID:␣Fill␣because␣unscattered␣light␣changed\n");
    return (FALSE);
  }
```

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

```
⟨ Tests for invalid grid 150 ⟩ +≡
  if (m.slab_index ≠ MGRID.slab_index) {
    if (Debug(DEBUG_GRID))
      fprintf(stderr, "GRID:␣Fill␣because␣slab␣refractive␣index␣changed\n");
    return (FALSE);
  }
  if (m.slab_cos_angle ≠ MGRID.slab_cos_angle) {
    if (Debug(DEBUG_GRID)) fprintf(stderr, "GRID:␣Fill␣because␣light␣angle␣changed\n");
    return (FALSE);
  }
  if (m.slab_top_slide_index ≠ MGRID.slab_top_slide_index) {
    if (Debug(DEBUG_GRID)) fprintf(stderr, "GRID:␣Fill␣because␣top␣slide␣index␣changed\n");
    return (FALSE);
  }
  if (m.slab_bottom_slide_index ≠ MGRID.slab_bottom_slide_index) {
    if (Debug(DEBUG_GRID)) fprintf(stderr, "GRID:␣Fill␣because␣bottom␣slide␣index␣changed\n");
    return (FALSE);
  }
  if (s ≡ FIND_AB ∧ r.slab.g ≠ RGRID.slab.g) {
    if (Debug(DEBUG_GRID)) fprintf(stderr, "GRID:␣Fill␣because␣anisotropy␣changed\n");
    return (FALSE);
  }
  if (s ≡ FIND_AG ∧ r.slab.b ≠ RGRID.slab.b) {
    if (Debug(DEBUG_GRID)) fprintf(stderr, "GRID:␣Fill␣because␣optical␣depth␣changed\n");
    return (FALSE);
  }
  if (s ≡ FIND_BsG ∧ r.default_ba ≠ RGRID.default_ba) {
    if (Debug(DEBUG_GRID)) fprintf(stderr, "GRID:␣Fill␣because␣mu_a␣changed\n");
    return (FALSE);
  }
  if (s ≡ FIND_BaG ∧ r.default_bs ≠ RGRID.default_bs) {
    if (Debug(DEBUG_GRID)) fprintf(stderr, "GRID:␣Fill␣because␣mu_s␣changed\n");
    return (FALSE);
  }
```

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

```
⟨ Prototype for abg_distance 154 ⟩ ≡
  void abg_distance(double a, double b, double g, guess_type *guess)
```

This code is used in sections 124 and 155.

155.
```
⟨ Definition for abg_distance 155 ⟩ ≡
  ⟨ Prototype for abg_distance 154 ⟩
  {
    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 123.

156. 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 156 ⟩ ≡
  void Near_Grid_Points(double r, double t, search_type s, int *i_min, int *j_min)
```

This code is used in sections 124 and 157.

157. ⟨ Definition for *Near_Grid_Points* 157 ⟩ ≡

```
⟨ Prototype for Near_Grid_Points 156 ⟩
{
  int i, j;
  double fval;
  double smallest = 10.0;
  struct measure_type old_mm;
  struct invert_type old_rr;

  (void) r;
  (void) t;
  (void) s;
  if (Debug(DEBUG_GRID)) fprintf(stderr, "GRID:␣Finding␣best␣grid␣points\n");
  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 123.

158. Routine to incorporate flipping of sample if needed. This is pretty simple. The assumption is that flipping is handled relative to the reflection side of the sphere. Thus even when flipping is needed, the usual call to `RT( )` will result in the correct values for the reflectances. The transmission values can then be calculated by swapping the top and bottom slides.

Technically, the value of slab should be **const** but it is not so that we don't pay a copying overhead whenever *flip* is false (the usual case).

⟨ Prototype for *RT_Flip* 158 ⟩ ≡

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

This code is used in sections 124 and 159.

159.

```
⟨ Definition for RT_Flip 159 ⟩ ≡
  ⟨ Prototype for RT_Flip 158 ⟩
  {
    double correct_UR1, correct_URU;
    RT(n, slab, UR1, UT1, URU, UTU);
    if (flip) {
      correct_UR1 = *UR1;
      correct_URU = *URU;
      SWAP(slab→n_top_slide, slab→n_bottom_slide)SWAP(slab→b_top_slide, slab→b_bottom_slide)RT(n, slab, UR1,
          UT1, URU, UTU);
      SWAP(slab→n_top_slide, slab→n_bottom_slide)SWAP(slab→b_top_slide,
          slab→b_bottom_slide) * UR1 = correct_UR1;
      *URU = correct_URU;
    }
  }
```

This code is used in section 123.

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

```
⟨ Definition for fill_grid_entry 160 ⟩ ≡
  static void fill_grid_entry(int i, int j)
  {
    double ur1, ut1, uru, utu;
    if (RR.slab.b ≤ 1 · 10^-6) RR.slab.b = 1 · 10^-6;
    if (Debug(DEBUG_GRID_CALC) ∧ i ≡ 0 ∧ j ≡ 0) {
      fprintf(stderr, "+␣␣␣i␣␣␣j␣");
      fprintf(stderr, "␣␣␣␣␣␣a␣␣␣␣␣␣␣␣␣b␣␣␣␣␣␣␣␣␣␣␣g␣␣␣␣␣|");
      fprintf(stderr, "␣␣␣␣␣M_R␣␣␣␣␣␣␣␣grid␣␣|");
      fprintf(stderr, "␣␣␣␣␣M_T␣␣␣␣␣␣␣␣grid\n");
    }
    if (Debug(DEBUG_EVERY_CALC)) {
      if (¬CALCULATING_GRID)
        fprintf(stderr, "a=%8.5f␣b=%10.5f␣g=%8.5f␣", RR.slab.a, RR.slab.b, RR.slab.g);
      else {
        if (j ≡ 0) fprintf(stderr, ".");
        if (i + 1 ≡ GRID_SIZE ∧ j ≡ 0) fprintf(stderr, "\n");
      }
    }
    RT_Flip(MM.flip_sample, RR.method.quad_pts, &RR.slab, &ur1, &ut1, &uru, &utu);
    if (Debug(DEBUG_EVERY_CALC) ∧ ¬CALCULATING_GRID)
      fprintf(stderr, "ur1=%8.5f␣ut1=%8.5f\n", ur1, ut1);
    The_Grid[GRID_SIZE * i + j][A_COLUMN] = RR.slab.a;
    The_Grid[GRID_SIZE * i + j][B_COLUMN] = RR.slab.b;
    The_Grid[GRID_SIZE * i + j][G_COLUMN] = RR.slab.g;
    The_Grid[GRID_SIZE * i + j][UR1_COLUMN] = ur1;
    The_Grid[GRID_SIZE * i + j][UT1_COLUMN] = ut1;
    The_Grid[GRID_SIZE * i + j][URU_COLUMN] = uru;
    The_Grid[GRID_SIZE * i + j][UTU_COLUMN] = utu;
    if (Debug(DEBUG_GRID_CALC)) {
      fprintf(stderr, "+␣%3d␣%3d␣", 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 123.

161. 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 161 ⟩ ≡
  void Fill_AB_Grid(struct measure_type m, struct invert_type r)
```

This code is used in sections 123 and 162.

162.

```
⟨ Definition for Fill_AB_Grid 162 ⟩ ≡
  ⟨ Prototype for Fill_AB_Grid 161 ⟩
  {
    int i, j;
    double a;
    double min_log_b = −8;     /∗ exp(-10) is smallest thickness ∗/
    double max_log_b = +8;     /∗ exp(+8) is greatest thickness ∗/
    if (Debug(DEBUG_GRID)) fprintf(stderr, "GRID:␣Filling␣AB␣grid\n");
    if (The_Grid ≡ Λ) Allocate_Grid(r.search);
    ⟨ Zero GG 167 ⟩
    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_log_b + (max_log_b − min_log_b) ∗ x);
      for (j = 0; j < GRID_SIZE; j++) {
        ⟨ Generate next albedo using j 164 ⟩
        fill_grid_entry(i, j);
      }
    }
    The_Grid_Initialized = TRUE;
    The_Grid_Search = FIND_AB;
  }
```

This code is used in section 123.

163. 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 163 ⟩ ≡
  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;
  }
```

164. Here is heuristic that seems to work well

```
⟨ Generate next albedo using j 164 ⟩ ≡
  a = (double) j/(GRID_SIZE − 1.0);
  RR.slab.a = (1.0 − a ∗ a) ∗ (1.0 − a) + (1.0 − a) ∗ (1.0 − a) ∗ a;
```

This code is used in sections 162 and 166.

165. 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 165 ⟩ ≡
  void Fill_AG_Grid(struct measure_type m, struct invert_type r)
```

This code is used in sections 123 and 166.

166.

```
⟨ Definition for Fill_AG_Grid 166 ⟩ ≡
  ⟨ Prototype for Fill_AG_Grid 165 ⟩
  {
    int i, j;
    double max_a = −10;
    double min_a = 10;

    if (Debug(DEBUG_GRID)) fprintf(stderr, "GRID:␣Filling␣AG␣grid\n");
    if (The_Grid ≡ Λ) Allocate_Grid(r.search);
    ⟨ Zero GG 167 ⟩
    Set_Calc_State(m, r);
    GG_b = r.slab.b;
    for (i = 0; i < GRID_SIZE; i++) {
      RR.slab.g = MAX_ABS_G ∗ (2.0 ∗ i/(GRID_SIZE − 1.0) − 1.0);
      for (j = 0; j < GRID_SIZE; j++) {
        double a;

        ⟨ Generate next albedo using j 164 ⟩
        fill_grid_entry(i, j);
        if (a < 0) RR.slab.a = 0;
        if (a < min_a) min_a = a;
        if (a > max_a) max_a = a;
      }
    }
    if (Debug(DEBUG_GRID)) {
      fprintf(stderr, "GRID:␣a␣␣␣␣␣␣␣␣␣=␣%9.7f␣to␣%9.7f␣\n", min_a, max_a);
      fprintf(stderr, "GRID:␣b␣␣␣␣␣␣␣␣␣=␣%9.5f␣\n", r.slab.b);
      fprintf(stderr, "GRID:␣g␣␣range␣=␣%9.6f␣to␣%9.6f␣\n", −MAX_ABS_G, MAX_ABS_G);
    }
    The_Grid_Initialized = TRUE;
    The_Grid_Search = FIND_AG;
  }
```

This code is used in section 123.

167.

```
⟨ Zero GG 167 ⟩ ≡
  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 162, 166, 169, 171, and 173.

168. 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 168 ⟩ ≡
  void Fill_BG_Grid(struct measure_type m, struct invert_type r)
```

This code is used in sections 124 and 169.

169. ⟨ Definition for *Fill_BG_Grid* 169 ⟩ ≡

```
  ⟨ Prototype for Fill_BG_Grid 168 ⟩
  {
    int i, j;
    double min_log_b = −8;      /∗ exp(-10) is smallest thickness ∗/
    double max_log_b = +10;     /∗ exp(+8) is greatest thickness ∗/
    if (The_Grid ≡ Λ) Allocate_Grid(r.search);
    ⟨ Zero GG 167 ⟩
    if (Debug(DEBUG_GRID)) fprintf(stderr, "GRID:␣Filling␣BG␣grid\n");
    Set_Calc_State(m, r);
    RR.slab.a = RR.default_a;
    GG_a = RR.slab.a;
    for (i = 0; i < GRID_SIZE; i++) {
      double x = (double) i/(GRID_SIZE − 1.0);
      RR.slab.b = exp(min_log_b + (max_log_b − min_log_b) ∗ x);
      for (j = 0; j < GRID_SIZE; j++) {
        RR.slab.g = MAX_ABS_G ∗ (2.0 ∗ j/(GRID_SIZE − 1.0) − 1.0);
        fill_grid_entry(i, j);
      }
    }
    if (Debug(DEBUG_GRID)) {
      fprintf(stderr, "GRID:␣a␣␣␣␣␣␣␣␣=␣%9.7f␣\n", RR.default_a);
      fprintf(stderr, "GRID:␣b␣␣range␣=␣%9.5f␣to␣%9.3f␣\n", exp(min_log_b), exp(max_log_b));
      fprintf(stderr, "GRID:␣g␣␣range␣=␣%9.6f␣to␣%9.6f␣\n", −MAX_ABS_G, MAX_ABS_G);
    }
    The_Grid_Initialized = TRUE;
    The_Grid_Search = FIND_BG;
  }
```

This code is used in section 123.

170. 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 remains fixed.

```
⟨ Prototype for Fill_BaG_Grid 170 ⟩ ≡
  void Fill_BaG_Grid(struct measure_type m, struct invert_type r)
```

This code is used in sections 124 and 171.

171. ⟨ Definition for *Fill_BaG_Grid* 171 ⟩ ≡

```
⟨ Prototype for Fill_BaG_Grid 170 ⟩
{
  int i, j;
  double max_a = −10;
  double min_a = 10;
  double bs = r.default_bs;
  double min_log_ba = −8;    /∗ exp(-10) is smallest thickness ∗/
  double max_log_ba = +10;    /∗ exp(+8) is greatest thickness ∗/
  if (The_Grid ≡ Λ) Allocate_Grid(r.search);
  ⟨ Zero GG 167 ⟩
  if (Debug(DEBUG_GRID)) {
    fprintf(stderr, "GRID:␣Filling␣BaG␣grid\n");
    fprintf(stderr, "GRID:␣␣␣␣␣␣␣bs␣=␣%9.5f\n", bs);
    fprintf(stderr, "GRID:␣ba␣range␣=␣%9.6f␣to␣%9.3f␣\n", exp(min_log_ba), exp(max_log_ba));
  }
  Set_Calc_State(m, r);
  GG_bs = bs;
  for (i = 0; i < GRID_SIZE; i++) {
    double x = (double) i/(GRID_SIZE − 1.0);
    double ba = exp(min_log_ba + (max_log_ba − min_log_ba) ∗ x);
    RR.slab.b = ba + bs;
    if (RR.slab.b > 0) RR.slab.a = bs/RR.slab.b;
    else RR.slab.a = 0;
    if (RR.slab.a < 0) RR.slab.a = 0;
    if (RR.slab.a < min_a) min_a = RR.slab.a;
    if (RR.slab.a > max_a) max_a = RR.slab.a;
    for (j = 0; j < GRID_SIZE; j++) {
      RR.slab.g = MAX_ABS_G ∗ (2.0 ∗ j/(GRID_SIZE − 1.0) − 1.0);
      fill_grid_entry(i, j);
    }
  }
  if (Debug(DEBUG_GRID)) {
    fprintf(stderr, "GRID:␣a␣␣␣␣␣␣␣␣=␣%9.7f␣to␣%9.7f␣\n", min_a, max_a);
    fprintf(stderr, "GRID:␣b␣␣range␣=␣%9.5f␣to␣%9.3f␣\n", exp(min_log_ba)+bs, exp(max_log_ba)+bs);
    fprintf(stderr, "GRID:␣g␣␣range␣=␣%9.6f␣to␣%9.6f␣\n", −MAX_ABS_G, MAX_ABS_G);
  }
  The_Grid_Initialized = TRUE;
  The_Grid_Search = FIND_BaG;
}
```

This code is used in section 123.

172. Very similiar to the above routine, but holding $b_a = \mu_a d$ 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 absorption remains fixed.

```
⟨ Prototype for Fill_BsG_Grid 172 ⟩ ≡
  void Fill_BsG_Grid(struct measure_type m, struct invert_type r)
```

This code is used in sections 124 and 173.

173.

```
⟨ Definition for Fill_BsG_Grid 173 ⟩ ≡
  ⟨ Prototype for Fill_BsG_Grid 172 ⟩
  {
    int i, j;
    double max_a = −10;
    double min_a = 10;
    double ba = r.default_ba;
    double min_log_bs = −8;     /∗ exp(-10) is smallest thickness ∗/
    double max_log_bs = +10;     /∗ exp(+8) is greatest thickness ∗/

    if (The_Grid ≡ Λ) Allocate_Grid(r.search);
    ⟨ Zero GG 167 ⟩
    if (Debug(DEBUG_GRID)) {
      fprintf(stderr, "GRID:␣Filling␣BsG␣grid\n");
      fprintf(stderr, "GRID:␣␣␣␣␣␣␣␣ba␣=␣%9.5f\n", ba);
      fprintf(stderr, "GRID:␣bs␣range␣=␣%9.6f␣to␣%9.3f␣\n", exp(min_log_bs), exp(max_log_bs));
    }
    Set_Calc_State(m, r);
    GG_ba = RR.default_ba;
    for (i = 0; i < GRID_SIZE; i++) {
      double x = (double) i/(GRID_SIZE − 1.0);
      double bs = exp(min_log_bs + (max_log_bs − min_log_bs) ∗ x);

      RR.slab.b = ba + bs;
      if (RR.slab.b > 0) RR.slab.a = 1 − RR.default_ba/RR.slab.b;
      else RR.slab.a = 0;
      if (RR.slab.a < 0) RR.slab.a = 0;
      if (RR.slab.a < min_a) min_a = RR.slab.a;
      if (RR.slab.a > max_a) max_a = RR.slab.a;
      for (j = 0; j < GRID_SIZE; j++) {
        RR.slab.g = MAX_ABS_G ∗ (2.0 ∗ j/(GRID_SIZE − 1.0) − 1.0);
        fill_grid_entry(i, j);
      }
    }
    if (Debug(DEBUG_GRID)) {
      fprintf(stderr, "GRID:␣a␣␣range␣=␣%9.7f␣to␣%9.7f␣\n", min_a, max_a);
      fprintf(stderr, "GRID:␣b␣␣range␣=␣%9.5f␣to␣%9.3f␣\n", exp(min_log_bs)+ba, exp(max_log_bs)+ba);
      fprintf(stderr, "GRID:␣g␣␣range␣=␣%9.6f␣to␣%9.6f␣\n", −MAX_ABS_G, MAX_ABS_G);
    }
    The_Grid_Initialized = TRUE;
    The_Grid_Search = FIND_BsG;
  }
```

This code is used in section 123.

174.

```
⟨ Prototype for Fill_Grid 174 ⟩ ≡
  void Fill_Grid(struct measure_type m, struct invert_type r, int force_new)
```

This code is used in sections 124 and 175.

175. ⟨ Definition for *Fill_Grid* 175 ⟩ ≡

```
⟨ Prototype for Fill_Grid 174 ⟩
{
  if (force_new ∨ ¬Same_Calc_State(m, r)) {
    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␣unknown␣search␣case.");
    }
  }
  Get_Calc_State(&MGRID, &RGRID);
}
```

This code is used in section 123.

176. Calculating R and T.

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

```
⟨ Prototype for Calculate_Distance 176 ⟩ ≡
  void Calculate_Distance(double *M_R, double *M_T, double *deviation)
```

This code is used in sections 124 and 177.

177.

```
⟨ Definition for Calculate_Distance 177 ⟩ ≡
  ⟨ Prototype for Calculate_Distance 176 ⟩
  {
    double Ru, Tu, ur1, ut1, uru, utu;
    if (RR.slab.b ≤ 1 · 10^−6) RR.slab.b = 1 · 10^−6;
    RT_Flip(MM.flip_sample, RR.method.quad_pts, &RR.slab, &ur1, &ut1, &uru, &utu);
    Sp_mu_RT_Flip(MM.flip_sample, RR.slab.n_top_slide, RR.slab.n_slab, RR.slab.n_bottom_slide,
        RR.slab.b_top_slide, RR.slab.b, RR.slab.b_bottom_slide, RR.slab.cos_angle, &Ru, &Tu);
    if ((¬CALCULATING_GRID ∧ Debug(DEBUG_ITERATIONS)) ∨ (CALCULATING_GRID ∧
          Debug(DEBUG_GRID_CALC))) fprintf(stderr, "␣␣␣␣␣␣␣␣");
    Calculate_Distance_With_Corrections(ur1, ut1, Ru, Tu, uru, utu, M_R, M_T, deviation);
  }
```

This code is used in section 123.

178.

```
⟨ Prototype for Calculate_Grid_Distance 178 ⟩ ≡
  double Calculate_Grid_Distance(int i, int j)
```

This code is used in sections 124 and 179.

179. ⟨ Definition for *Calculate_Grid_Distance* 179 ⟩ ≡

```
⟨ Prototype for Calculate_Grid_Distance 178 ⟩
{
  double ur1, ut1, uru, utu, Ru, Tu, b, dev, LR, LT;
  if (Debug(DEBUG_GRID_CALC) ∧ i ≡ 0 ∧ j ≡ 0) {
    fprintf(stderr, "+␣␣␣i␣␣␣j␣");
    fprintf(stderr, "␣␣␣␣␣␣a␣␣␣␣␣␣␣␣␣b␣␣␣␣␣␣␣␣␣␣g␣␣␣␣␣|");
    fprintf(stderr, "␣␣␣␣␣M_R␣␣␣␣␣␣␣␣grid␣␣␣|");
    fprintf(stderr, "␣␣␣␣␣M_T␣␣␣␣␣␣␣␣grid␣␣␣|␣␣distance\n");
  }
  if (Debug(DEBUG_GRID_CALC)) fprintf(stderr, "g␣%3d␣%3d␣", 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_Flip(MM.flip_sample, RR.slab.n_top_slide, RR.slab.n_slab, RR.slab.n_bottom_slide,
      RR.slab.b_top_slide, b, RR.slab.b_bottom_slide, RR.slab.cos_angle, &Ru, &Tu);
  CALCULATING_GRID = 1;
  Calculate_Distance_With_Corrections(ur1, ut1, Ru, Tu, uru, utu, &LR, &LT, &dev);
  CALCULATING_GRID = 0;
  return dev;
}
```

This code is used in section 123.

180. 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_Points* routine.

Ru and *Tu* refer to the unscattered (collimated) reflection and transmission.

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

⟨ Prototype for *Calculate_Distance_With_Corrections* 180 ⟩ ≡

```
void Calculate_Distance_With_Corrections(double UR1, double UT1, double Ru, double Tu, double
    URU, double UTU, double ∗M_R, double ∗M_T, double ∗dev)
```

This code is used in sections 124 and 181.

181. ⟨ Definition for *Calculate_Distance_With_Corrections* 181 ⟩ ≡

```
⟨ Prototype for Calculate_Distance_With_Corrections 180 ⟩
{
  ⟨ Determine calculated light to be used 182 ⟩
  switch (MM.num_spheres) {
  case 0: ⟨ Calc M_R and M_T for no spheres 184 ⟩
    break;
  case 1:
    if (MM.method ≡ COMPARISON) {
      ⟨ Calc M_R and M_T for dual beam sphere 198 ⟩
    }
    else {
      ⟨ Calc M_R and M_T for single beam sphere 191 ⟩
    }
    break;
  case 2: ⟨ Calc M_R and M_T for two spheres 200 ⟩
    break;
  default: fprintf(stderr, "Bad␣number␣of␣spheres␣=␣%d\n", MM.num_spheres);
    exit(EXIT_FAILURE);
  }
  ⟨ Calculate the deviation 201 ⟩
  ⟨ Print diagnostics 204 ⟩
}
```

This code is used in section 123.

182. The calculated values for `M_R` and `M_T` must be adapted to match the measurements. The diffuse light `URU` and `UTU` are used to determine the gain from the sphere. They're only modified by the lost light calculation. All values can become slightly negative because the Monte Carlo is noisy. Negative values are set to zero.

⟨ Determine calculated light to be used 182 ⟩ ≡

```
double UR1_calc, UT1_calc, URU_calc, UTU_calc;

URU_calc = URU − MM.uru_lost;
if (URU_calc < 0) URU_calc = 0;
UTU_calc = UTU − MM.utu_lost;
if (UTU_calc < 0) UTU_calc = 0;
```

See also section 183.

This code is used in section 181.

183. The measurements for `UR1` and `UT1` need to be modified to accommodate light that misses the detector either because it is intentionally not collected (unscattered light) or it leaks out (lost light). Since none of the light that leaks out could be unscattered light, these two are independent of one another.

⟨ Determine calculated light to be used 182 ⟩ +≡

```
UR1_calc = UR1 − (1.0 − MM.fraction_of_ru_in_mr) ∗ Ru − MM.ur1_lost;
if (UR1_calc < 0) UR1_calc = 0;
UT1_calc = UT1 − (1.0 − MM.fraction_of_tu_in_mt) ∗ Tu − MM.ut1_lost;
if (UT1_calc < 0) UT1_calc = 0;
```

184. When no spheres are used, then no corrections can or need to be made. The lost light estimates in `MM`.*ur1_lost* and `MM`.*ut1_lost* should be zero and so the values for *UR1_calc* and *UT1_calc* properly account for the presence or absence of unscattered light.

⟨ Calc `M_R` and `M_T` for no spheres 184 ⟩ ≡

```
{
  *M_R = UR1_calc;
  *M_T = UT1_calc;
}
```

This code is used in section 181.

185. Reflectance measurement for one sphere.

The total reflection from a slab may be broken down into light that has and has not been scattered. The total reflectance for normal incidence on an infinite slab is denoted by van de Hulst by `UR1`. To track integrating sphere corrections, this is separated into scattered and unscattered parts.

$$\texttt{UR1} = R_{\text{unscat}} + R_{\text{scat}}$$

`UR1` is calculated from the current guess of optical properties, but we can also calculate R_{unscat} by setting $\mu_s = 0$ which means that the albedo is zero. Assuming the incident power is P_i then the scattered light in the sphere P_{ss} and the unscattered light in the sphere P_{su} start as

$$P_{ss} = (\texttt{UR1} - R_{\text{unscat}})P_i \qquad \textit{and} \qquad P_{su} = R_{\text{unscat}}P_i$$

186. Light that is lost.

In an experiment, the scattered light in the sphere will be reduced by any light that leaks out. This is determined by doing a Monte Carlo simulation to determine $\texttt{UR1}_{\text{lost}}$ or how much light is scattered within the sample and escapes outside the sphere. Since the only way that light may be lost is through scattering, this does not affect the unscattered fraction.

The fraction of unscattered light collected by the sphere f_{unscat} is determined by the experimentalist. The unscattered reflected light can aligned so that it bounces off the sample and exits completely through the entrance port so that $f_{\text{unscat}} = 0$. Alternatively, the beam may be incident on the sample at an angle so that the unscattered light will bounce and remain completely within the sphere so that $f_{\text{unscat}} = 1$.

$$P_{ss} = (\texttt{UR1} - \texttt{UR1}_{\text{unscat}} - R_{\text{lost}})P_i \qquad \textit{and} \qquad P_{su} = f_{\text{unscat}}R_{\text{unscat}}P_i$$

187. Incident light that misses the sample.

In an experiment, a fraction f_{miss} of the incident beam might miss the sample and hit the sphere wall instead. In this case, both R_{ss} and R_{su} are affected. Typically the beam is much smaller than the sample and $f_{\text{miss}} = 0$, however sometimes the beam diverges too much and some of the beam hits the wall. After hitting the sphere wall then $f_{\text{miss}}r_w$ will be added to the scattered light in the sphere. However, now only $(1 - f_{\text{miss}})$ hits the sample directly, both the scattered and unscattered light in the sphere must be adjusted accordingly. So in the unusual case of non-zero f_{miss}, we have

$$R_{ss} = (1 - f_{\text{miss}})(\texttt{UR1} - R_{\text{unscat}} - \texttt{UR1}_{\text{lost}})P_i + f_{\text{miss}}r_wP_i \qquad \textit{and} \qquad R_{su} = (1 - f_{\text{miss}})f_{\text{unscat}}R_{\text{unscat}}P_i$$

188. Reflectance with baffle.

When a baffle blocks light from passing directly between the sample to the detector then the light reflected by the sample must bounce once. Some of the light will be reflected by the sphere walls, but some may be reflected by the third port. The weighted reflectance of the first bounce is

$$r_{\text{first}} = (1 - a_t) r_w + a_t r_t$$

We can safely assume that the fraction of light generated by f_{miss} will originate close enough to the sample that a baffle, if present, will prevent light from directly reaching the detector. The scattered light in the sphere after the first bounce will be

$$P_{ss} = r_{\text{first}} \left[(1 - f_{\text{miss}})(\texttt{UR1} - R_{\text{unscat}} - R_{\text{lost}}) + f_{\text{miss}} r_w \right] P_i$$

All unscattered reflectance that is collected must hit the sphere wall (otherwise it would exit through the entrance port and not be collected!). This means that the correction in r_{first} for the entrance port is not needed and the unreflected light can just be multiplied by r_w

$$P_{su} = r_w (1 - f_{\text{miss}}) f_{\text{unscat}} R_{\text{unscat}} P_i$$

The last step is to account for the sphere gain. The sample is held in the sample port and the entrance port is empty. The total reflection for diffuse illumination of the sample is `URU`. This quantity must also be corrected for light that is not collected by the sphere $\texttt{UR1}_{\text{lost}}$. The gain for such a sphere is $G(\texttt{URU} - \texttt{URU}_{\text{lost}}, 0)$. Finally,

$$P_s = [a_d (1 - r_d)] \cdot G(\texttt{URU} - \texttt{URU}_{\text{lost}}, 0) \left[P_{ss} + P_{su} \right]$$

189. The reflection standard.

We let $\texttt{UR1} = \texttt{URU} = r_{\text{std}}$, $R_{\text{unscat}} = 0$, $R_{\text{lost}} = 0$ to get

$$P_{\text{std}} = [a_d (1 - r_d)] \cdot r_{\text{first}} \left[(1 - f_{\text{miss}}) r_{\text{std}} + f_{\text{miss}} r_w \right] G(r_{\text{std}}, 0) P_i$$

190. The open port.

We let $\texttt{UR1} = \texttt{URU} = 0$, $R_{\text{unscat}} = 0$, $R_{\text{lost}} = 0$ to get

$$P_0 = [a_d (1 - r_d)] \cdot r_{\text{first}} f_{\text{miss}} r_w G(0, 0) P_i$$

191. The unbaffled reflectance sphere.

In this case, light can reach the detector from the sample. The first bounce is not needed which can be accommodated by letting $r_{\text{first}} = 1$. We, of course, assume that the gain is calculated assuming that no baffle is present. The incident power P_i and the quantities in square brackets are identical for P_s, P_{std}, and P_0 and cancel in the normalized reflection fraction

$$M_R = r_{\text{std}} \cdot \frac{P_s - P_0}{P_{\text{std}} - P_0}$$

In addition, the entrance port is empty and therefore $r_t = 0$ and can be omitted from the r_{first} calculation. This leads to the following code for `M_R`

⟨ Calc `M_R` and `M_T` for single beam sphere 191 ⟩ ≡

```
double P_std, P, P_0, G, G_0, G_std, r_first, P_ss, P_su;
r_first = 1;
if (MM.baffle_r) r_first = MM.rw_r ∗ (1 − MM.at_r);
UR1_calc = UR1 − Ru − MM.ur1_lost;
if (UR1_calc < 0) UR1_calc = 0;
G_0 = Gain(REFLECTION_SPHERE, MM, 0.0, 0.0);
G = Gain(REFLECTION_SPHERE, MM, URU_calc, 0.0);
G_std = Gain(REFLECTION_SPHERE, MM, MM.rstd_r, 0.0);
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);
P_ss = r_first ∗ (UR1_calc ∗ (1 − MM.f_r) + MM.f_r ∗ MM.rw_r);
P_su = MM.rw_r ∗ (1 − MM.f_r) ∗ MM.fraction_of_ru_in_mr ∗ Ru;
P = G ∗ (P_ss + P_su);
∗M_R = MM.rstd_r ∗ (P − P_0)/(P_std − P_0);
if (Debug(DEBUG_SPHERE_GAIN) ∧ ¬CALCULATING_GRID) {
  fprintf(stderr, "SPHERE:␣REFLECTION\n");
  fprintf(stderr, "SPHERE:␣␣␣␣␣␣G0␣=␣%7.3f␣␣␣␣␣␣G␣␣=␣%7.3f␣G_std␣=␣%7.3f\n", G_0, G, G_std);
  fprintf(stderr, "SPHERE:␣␣␣␣␣␣P0␣=␣%7.3f␣␣␣␣␣␣P␣␣=␣%7.3f␣P_std␣=␣%7.3f\n", P_0, P, P_std);
  fprintf(stderr, "SPHERE:␣␣␣␣␣UR1␣=␣%7.3f␣UR1calc␣=␣%7.3f␣␣␣M_R␣=␣%7.3f\n", UR1, UR1_calc,
      ∗M_R);
}
```

See also section 196.

This code is used in section 181.

192. Transmittance measurement for one sphere.

Like in the reflection case, the total transmission can be split into unscattered transmission and scattered transmission,

$$UT1 = T_u + T_s$$

We define P_{ss} as the scattered light in the sphere and P_{su} as the unscattered light in the sphere for an incident power P_i

$$P_{ss} = (\mathtt{UT1} - T_u)P_i \qquad \textit{and} \qquad P_{su} = f_{\text{unscat}} T_u P_i$$

193. Transmitted light not collected.

The scattered light will be reduced by $\texttt{UT1}_{\rm lost}$. The unscattered light will be affected by the fraction of unscattered light collected by the sphere. When transmission measurements are made, the third port (opposite the sample port in the sphere) is typically filled with a known standard. However, the third port might also be left open so that all the scattered light might exit and only scattered light is collected. In the former case $f_{\rm unscat} = 1$ and in the latter case $f_{\rm unscat} = 0$. So including these effects gives

$$P_{ss} = (\texttt{UT1} - T_u - T_{\rm lost})P_i \qquad \textit{and} \qquad P_{su} = f_{\rm unscat} T_u$$

194. The baffling case of transmission.

With a baffle, then scattered light from the sample cannot reach the detector without a bounce. This weighted reflection is given by

$$r_{\rm first} = (1 - a_t)r_w + a_t r_t$$

The unscattered light will be reflected by the standard r_t in the third port and so

$$P_{ss} = r_{\rm first}(\texttt{UT1} - T_u - T_{\rm lost})P_i \qquad \textit{and} \qquad P_{su} = r_t f_{\rm unscat} T_u$$

The last step is to account for the sphere gain. The sample is held in the sample port and the third port reflects r_t. The total reflection for diffuse illumination of the sample is $\texttt{URU}$. This quantity must also be corrected for light that is not collected by the sphere $\texttt{UR1}_{\rm lost}$. The gain for such a sphere is $G(\texttt{URU} - \texttt{URU}_{\rm lost}, r_t)$. Finally,

$$P_s = G(\texttt{URU} - \texttt{URU}_{\rm lost}, r_t)(P_{ss} + P_{su})$$

195. No baffle.

Here $r_{\rm first} = 1$ and the gain should be calculated assuming no baffle.

196. The standard measurement.

When transmission measurements are made, typically the third port (the one that let the light into the sphere for the reflection measurement) is filled with a known standard. In this case, the natural way to make the 100% transmission measurement is to shine the beam through the empty sample port onto the known standard.

We let `UT1` $= T_u = 1$, $T_{\text{lost}} = 0$, `URU` $= 0$, $r_t = r_{\text{std}}$, and $f_{\text{unscat}} = 1$ so

$$P_{\text{std}} = G(0, r_{\text{std}}) r_{\text{std}} P_i$$

The estimate for the measured transmittance is

$$M_T = r_{\text{std}} \frac{P_s}{P_{\text{std}}} = \frac{P_{ss} + P_{su}}{P_i} \cdot \frac{G(\texttt{URU} - \texttt{URU}_{\text{lost}}, r_{\text{third}})}{G(0, r_{\text{std}})}$$

```
⟨ Calc M_R and M_T for single beam sphere 191 ⟩ +≡
  {
    double r_first = 1;
    double r_third = MM.rstd_t;
    if (MM.fraction_of_tu_in_mt ≡ 0) r_third = 0;
    if (MM.baffle_t) r_first = MM.rw_t * (1 − MM.at_t) + MM.rstd_t * MM.at_t;
    UT1_calc = UT1 − Tu − MM.ut1_lost;
    if (UT1_calc < 0) UT1_calc = 0;
    G = Gain(TRANSMISSION_SPHERE, MM, URU_calc, r_third);
    G_std = Gain(TRANSMISSION_SPHERE, MM, 0, MM.rstd_t);
    *M_T = (r_third * Tu * MM.fraction_of_tu_in_mt + r_first * UT1_calc) * G/G_std;
    if (Debug(DEBUG_SPHERE_GAIN) ∧ ¬CALCULATING_GRID) {
      fprintf(stderr, "SPHERE:␣TRANSMISSION\n");
      fprintf(stderr, "SPHERE:␣␣␣␣␣␣G␣␣=␣%7.3f␣␣␣G_std␣=␣%7.3f\n", G, G_std);
      fprintf(stderr, "SPHERE:␣␣␣␣␣UT1␣=␣%7.3f␣UT1calc␣=␣%7.3f␣T_c␣=␣%7.3f\n", UT1, UT1_calc, Tu);
      fprintf(stderr, "SPHERE:␣␣␣␣␣M_T␣=␣%7.3f\n", *M_T);
      fprintf(stderr, "\n");
    }
  }
```

197. Dual beam case for one sphere.

198. The dual beam case is different because the sphere efficiency is equivalent for measurement of light hitting the sample first or hitting the reference standard first. The dual beam measurement should report the ratio of these two reflectance measurements, thereby eliminating the need to calculate the sphere gain.

The only correction that needs to be made have already been made, namely subtracting the `UR1` or `UT1` lost light and also accounting for whether or not unscattered light is collected.

Originally, I had a bunch of calculations trying to account for light that hits the sphere wall first. Since the exact details of how a dual beam spectrometer reports its measurements is unknown, it makes no sense to try and account for it.

```
⟨ Calc M_R and M_T for dual beam sphere 198 ⟩ ≡
  {
    *M_R = UR1_calc;
    *M_T = UT1_calc;
  }
```

This code is used in section 181.

199. Double integrating spheres.

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

The normalized sphere measurements for two spheres are

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

and

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

Note that `R_0` and `T_0` will be zero unless one has explicitly set the fraction *m.f_r* or *m.f_t* to be non-zero.

⟨ Calc `M_R` and `M_T` for two spheres 200 ⟩ ≡

```
{
  double R_0, T_0;
  R_0 = Two_Sphere_R(MM, 0, 0, 0, 0);
  T_0 = Two_Sphere_T(MM, 0, 0, 0, 0);
  *M_R = MM.rstd_r * (Two_Sphere_R(MM, UR1_calc, URU_calc, UT1_calc,
        UTU_calc) − R_0)/(Two_Sphere_R(MM, MM.rstd_r, MM.rstd_r, 0, 0) − R_0);
  *M_T = (Two_Sphere_T(MM, UR1_calc, URU_calc, UT1_calc, UTU_calc) − T_0)/(Two_Sphere_T(MM, 0, 0,
        1, 1) − T_0);
}
```

This code is used in section 181.

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

⟨ Calculate the deviation 201 ⟩ ≡

```
if (RR.search ≡ FIND_A ∨ RR.search ≡ FIND_G ∨ RR.search ≡ FIND_B ∨ RR.search ≡ FIND_Bs ∨ RR.search ≡
      FIND_Ba) {
  ⟨ One parameter deviation 202 ⟩
}
else {
  ⟨ Two parameter deviation 203 ⟩
}
```

This code is used in section 181.

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

⟨ One parameter deviation 202 ⟩ ≡

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

This code is used in section 201.

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

```
⟨ Two parameter deviation 203 ⟩ ≡
  if (RR.metric ≡ RELATIVE) {
    if (MM.m_t > ABIT) ∗dev = T_TRUST_FACTOR ∗ fabs(MM.m_t − ∗M_T)/(UTU_calc + ABIT);
    if (RR.default_a ≠ 0) {
      ∗dev += fabs(MM.m_r − ∗M_R)/(URU_calc + ABIT);
    }
  }
  else {
    ∗dev = T_TRUST_FACTOR ∗ fabs(MM.m_t − ∗M_T);
    if (RR.default_a ≠ 0) ∗dev += fabs(MM.m_r − ∗M_R);
  }
```

This code is used in section 201.

204. 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 204 ⟩ ≡
  if ((Debug(DEBUG_ITERATIONS) ∧ ¬CALCULATING_GRID) ∨
        (Debug(DEBUG_GRID_CALC) ∧ CALCULATING_GRID)) {
    fprintf(stderr, "%10.5f␣%10.4f␣%10.5f␣|", RR.slab.a, RR.slab.b, RR.slab.g);
    fprintf(stderr, "␣%10.5f␣%10.5f␣|", MM.m_r, ∗M_R);
    fprintf(stderr, "␣%10.5f␣%10.5f␣|", MM.m_t, ∗M_T);
    fprintf(stderr, "%10.3f\n", ∗dev);
  }
```

This code is used in section 181.

205.
```
⟨ Prototype for Find_AG_fn 205 ⟩ ≡
  double Find_AG_fn(double x[ ])
```

This code is used in sections 124 and 206.

206.
```
⟨ Definition for Find_AG_fn 206 ⟩ ≡
  ⟨ Prototype for Find_AG_fn 205 ⟩
  {
    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 123.

207.
```
⟨ Prototype for Find_AB_fn 207 ⟩ ≡
  double Find_AB_fn(double x[ ])
```

This code is used in sections 124 and 208.

208.

```
⟨ Definition for Find_AB_fn 208 ⟩ ≡
  ⟨ Prototype for Find_AB_fn 207 ⟩
  {
    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 123.

209.

```
⟨ Prototype for Find_Ba_fn 209 ⟩ ≡
  double Find_Ba_fn(double x)
```

This code is used in sections 124 and 210.

210. 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 210 ⟩ ≡
  ⟨ Prototype for Find_Ba_fn 209 ⟩
  {
    double m_r, m_t, deviation, ba, bs;

    bs = RR.slab.b;
    ba = bcalc2b(x);
    RR.slab.b = ba + bs;    /* unswindle */
    RR.slab.a = bs/(ba + bs);
    Calculate_Distance(&m_r, &m_t, &deviation);
    RR.slab.b = bs;    /* swindle */
    return deviation;
  }
```

This code is used in section 123.

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

```
⟨ Prototype for Find_Bs_fn 211 ⟩ ≡
  double Find_Bs_fn(double x)
```

This code is used in sections 124 and 212.

212.

```
⟨ Definition for Find_Bs_fn 212 ⟩ ≡
  ⟨ Prototype for Find_Bs_fn 211 ⟩
  {
    double m_r, m_t, deviation, ba, bs;

    ba = RR.slab.b;    /* unswindle */
    bs = bcalc2b(x);
    RR.slab.b = ba + bs;
    RR.slab.a = bs/(ba + bs);
    Calculate_Distance(&m_r, &m_t, &deviation);
    RR.slab.b = ba;    /* swindle */
    return deviation;
  }
```

This code is used in section 123.

213. ⟨ Prototype for *Find_A_fn* 213 ⟩ ≡

```
double Find_A_fn(double x)
```

This code is used in sections 124 and 214.

214. ⟨ Definition for *Find_A_fn* 214 ⟩ ≡

```
⟨ Prototype for Find_A_fn 213 ⟩
{
  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 123.

215. ⟨ Prototype for *Find_B_fn* 215 ⟩ ≡

```
double Find_B_fn(double x)
```

This code is used in sections 124 and 216.

216. ⟨ Definition for *Find_B_fn* 216 ⟩ ≡

```
⟨ Prototype for Find_B_fn 215 ⟩
{
  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 123.

217. ⟨ Prototype for *Find_G_fn* 217 ⟩ ≡

```
double Find_G_fn(double x)
```

This code is used in sections 124 and 218.

218. ⟨ Definition for *Find_G_fn* 218 ⟩ ≡

```
⟨ Prototype for Find_G_fn 217 ⟩
{
  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 123.

219. ⟨ Prototype for *Find_BG_fn* 219 ⟩ ≡

```
double Find_BG_fn(double x[ ])
```

This code is used in sections 124 and 220.

220. ⟨ Definition for *Find_BG_fn* 220 ⟩ ≡

```
⟨ Prototype for Find_BG_fn 219 ⟩
{
    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 123.

221. 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* 221 ⟩ ≡

```
  double Find_BaG_fn(double x[ ])
```

This code is used in sections 124 and 222.

222. ⟨ Definition for *Find_BaG_fn* 222 ⟩ ≡

```
⟨ Prototype for Find_BaG_fn 221 ⟩
{
    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 123.

223. ⟨ Prototype for *Find_BsG_fn* 223 ⟩ ≡

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

This code is used in sections 124 and 224.

224. ⟨ Definition for *Find_BsG_fn* 224 ⟩ ≡

```
⟨ Prototype for Find_BsG_fn 223 ⟩
{
    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 123.

225. 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 225 ⟩ ≡
  double maxloss(double f)
```

This code is used in sections 124 and 226.

226.

```
⟨ Definition for maxloss 226 ⟩ ≡
  ⟨ Prototype for maxloss 225 ⟩
  {
    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 123.

227. 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 227 ⟩ ≡
  void Max_Light_Loss(struct measure_type m, struct invert_type r, double *ur1_loss, double
      *ut1_loss)
```

This code is used in sections 124 and 228.

228. ⟨ Definition for *Max_Light_Loss* 228 ⟩ ≡

```
⟨ Prototype for Max_Light_Loss 227 ⟩
{
  struct measure_type m_old;
  struct invert_type r_old;

  *ur1_loss = m.ur1_lost;
  *ut1_loss = m.ut1_lost;
  if (Debug(DEBUG_LOST_LIGHT))
    fprintf(stderr, "\nlost␣before␣ur1=%7.5f,␣ut1=%7.5f\n", *ur1_loss, *ut1_loss);
  Get_Calc_State(&m_old, &r_old);
  Set_Calc_State(m, r);
  if (maxloss(1.0) * maxloss(0.0) < 0) {
    double frac;

    frac = zbrent(maxloss, 0.00, 1.0, 0.001);
    *ur1_loss = m.ur1_lost * frac;
    *ut1_loss = m.ut1_lost * frac;
  }
  Set_Calc_State(m_old, r_old);
  if (Debug(DEBUG_LOST_LIGHT))
    fprintf(stderr, "lost␣after␣␣ur1=%7.5f,␣ut1=%7.5f\n", *ur1_loss, *ut1_loss);
}
```

This code is used in section 123.

229. this is currently unused

```
⟨ Unused diffusion fragment 229 ⟩ ≡
  typedef struct {
    double f;
    double aprime;
    double bprime;
    double gprime;
    double boundary_method;
    double n_top;
    double n_bottom;
    double slide_top;
    double slide_bottom;
    double F0;
    double depth;
    double Exact_coll_flag;
  } slabtype;

  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.F0 = 1/M_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;
  }
```

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

```
⟨ iad_find.c 230 ⟩ ≡
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "ad_globl.h"
#include "nr_util.h"
#include "nr_mnbrk.h"
#include "nr_brent.h"
#include "nr_amoeb.h"
#include "iad_type.h"
#include "iad_util.h"
#include "iad_calc.h"
#include "iad_find.h"
#define NUMBER_OF_GUESSES 10
  guess_type guess[NUMBER_OF_GUESSES];

  int compare_guesses(const void *p1, const void *p2)
  {
    guess_type *g1 = (guess_type *) p1;
    guess_type *g2 = (guess_type *) p2;

    if (g1→distance < g2→distance) return −1;
    else if (g1→distance ≡ g2→distance) return 0;
    else return 1;
  }
  ⟨ Definition for U_Find_Ba 244 ⟩
  ⟨ Definition for U_Find_Bs 242 ⟩
  ⟨ Definition for U_Find_A 246 ⟩
  ⟨ Definition for U_Find_B 250 ⟩
  ⟨ Definition for U_Find_G 248 ⟩
  ⟨ Definition for U_Find_AG 253 ⟩
  ⟨ Definition for U_Find_AB 233 ⟩
  ⟨ Definition for U_Find_BG 258 ⟩
  ⟨ Definition for U_Find_BaG 264 ⟩
  ⟨ Definition for U_Find_BsG 269 ⟩
```

231. 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 231 ⟩ ≡
  ⟨ Prototype for U_Find_Ba 243 ⟩;
  ⟨ Prototype for U_Find_Bs 241 ⟩;
  ⟨ Prototype for U_Find_A 245 ⟩;
  ⟨ Prototype for U_Find_B 249 ⟩;
  ⟨ Prototype for U_Find_G 247 ⟩;
  ⟨ Prototype for U_Find_AG 252 ⟩;
  ⟨ Prototype for U_Find_AB 232 ⟩;
  ⟨ Prototype for U_Find_BG 257 ⟩;
  ⟨ Prototype for U_Find_BaG 263 ⟩;
  ⟨ Prototype for U_Find_BsG 268 ⟩;
```

232. Fixed Anisotropy.

This is the most common case.

```
⟨ Prototype for U_Find_AB 232 ⟩ ≡
  void U_Find_AB(struct measure_type m, struct invert_type *r)
```

This code is used in sections 231 and 233.

233.

```
⟨ Definition for U_Find_AB 233 ⟩ ≡
  ⟨ Prototype for U_Find_AB 232 ⟩
  {
    ⟨ Allocate local simplex variables 234 ⟩
    if (Debug(DEBUG_SEARCH)) {
      fprintf(stderr, "SEARCH:␣Using␣U_Find_AB()");
      fprintf(stderr, "␣(mu=%6.4f)", r→slab.cos_angle);
      if (r→default_g ≠ UNINITIALIZED) fprintf(stderr, "␣␣default_g␣=␣%8.5f", r→default_g);
      fprintf(stderr, "\n");
    }
    r→slab.g = (r→default_g ≡ UNINITIALIZED) ? 0 : r→default_g;
    Set_Calc_State(m, *r);
    ⟨ Get the initial a, b, and g 235 ⟩
    ⟨ Initialize the nodes of the a and b simplex 236 ⟩
    ⟨ Evaluate the a and b simplex at the nodes 237 ⟩
    amoeba(p, y, 2, r→tolerance, Find_AB_fn, &r→AD_iterations);
    ⟨ Choose the best node of the a and b simplex 238 ⟩
    ⟨ Free simplex data structures 240 ⟩
    ⟨ Put final values in result 239 ⟩
  }
```

This code is used in section 230.

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

```
⟨ Allocate local simplex variables 234 ⟩ ≡
  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 233, 253, 258, 264, and 269.

235. 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* 235 ⟩ ≡

```
  {     /∗ 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)) Fill_Grid(m, ∗r, 1);     /∗ distance to nearest grid point ∗/
    Near_Grid_Points(m.m_r, m.m_t, r→search, &i_best, &j_best);
    Grid_ABG(i_best, j_best, &(guess[1]));
    Grid_ABG(i_best + 1, j_best, &(guess[2]));
    Grid_ABG(i_best − 1, j_best, &(guess[3]));
    Grid_ABG(i_best, j_best + 1, &(guess[4]));
    Grid_ABG(i_best, j_best − 1, &(guess[5]));
    Grid_ABG(i_best + 1, j_best + 1, &(guess[6]));
    Grid_ABG(i_best − 1, j_best − 1, &(guess[7]));
    Grid_ABG(i_best + 1, j_best − 1, &(guess[8]));
    Grid_ABG(i_best − 1, j_best + 1, &(guess[9]));
    qsort((void ∗) guess, count, sizeof(guess_type), compare_guesses);
    if (Debug(DEBUG_BEST_GUESS)) {
      int k;
      fprintf(stderr, "BEST:␣GRID␣GUESSES\n");
      fprintf(stderr, "BEST:␣␣k␣␣␣␣␣␣albedo␣␣␣␣␣␣␣␣␣␣b␣␣␣␣␣␣␣␣␣␣g␣␣␣distance\n");
      for (k = 0; k ≤ 6; k++) {
        fprintf(stderr, "BEST:%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 233, 253, 258, 264, and 269.

236. ⟨ Initialize the nodes of the *a* and *b* simplex 236 ⟩ ≡

```
{
  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 (k ≡ kk) continue;
    if (guess[0].b ≠ guess[kk].b ∨ guess[k].b ≠ guess[kk].b) break;
  }
  p[3][1] = a2acalc(guess[kk].a);
  p[3][2] = b2bcalc(guess[kk].b);
  if (Debug(DEBUG_BEST_GUESS)) {
    fprintf(stderr, "-----------------------------------------------------\n");
    fprintf(stderr, "BEST:␣<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, "BEST:␣<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, "BEST:␣<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);
    fprintf(stderr, "\n");
  }
}
```

This code is used in section 233.

237. ⟨ Evaluate the *a* and *b* simplex at the nodes 237 ⟩ ≡

```
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 233.

238. ⟨ Choose the best node of the *a* and *b* simplex 238 ⟩ ≡

```
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 233.

239. ⟨ Put final values in result 239 ⟩ ≡

```
r→a = r→slab.a;
r→b = r→slab.b;
r→g = r→slab.g;
r→found = (r→tolerance > r→final_distance);
Set_Calc_State(m, *r);
```

This code is used in sections 233, 242, 244, 246, 248, 250, 253, 258, 264, and 269.

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

⟨ Free simplex data structures 240 ⟩ ≡

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

This code is used in sections 233, 253, 258, 264, and 269.

241. 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 241 ⟩ ≡
  void U_Find_Bs(struct measure_type m, struct invert_type *r)
```

This code is used in sections 231 and 242.

242. ⟨ Definition for *U_Find_Bs* 242 ⟩ ≡

```
  ⟨ Prototype for U_Find_Bs 241 ⟩
  {
    double ax, bx, cx, fa, fb, fc, bs;
    if (Debug(DEBUG_SEARCH)) {
      fprintf(stderr, "SEARCH:␣Using␣U_Find_Bs()");
      fprintf(stderr, "␣(mu=%6.4f)", r→slab.cos_angle);
      if (r→default_ba ≠ UNINITIALIZED) fprintf(stderr, "␣␣default_ba␣=␣%8.5f", r→default_ba);
      if (r→default_g ≠ UNINITIALIZED) fprintf(stderr, "␣␣default_g␣=␣%8.5f", r→default_g);
      fprintf(stderr, "\n");
    }
    if (m.m_t ≡ 0) {
      r→slab.b = HUGE_VAL;
      U_Find_A(m, r);
      return;
    }
    r→slab.a = 0;
    r→slab.g = (r→default_g ≡ UNINITIALIZED) ? 0 : r→default_g;
    r→slab.b = (r→default_ba ≡ UNINITIALIZED) ? HUGE_VAL : r→default_ba;
    Set_Calc_State(m, *r);    /* store ba in RR.slab.b */
    ax = b2bcalc(0.1);    /* first try for bs */
    bx = b2bcalc(1.0);
    mnbrak(&ax, &bx, &cx, &fa, &fb, &fc, Find_Bs_fn, &r→AD_iterations);
    r→final_distance = brent(ax, bx, cx, Find_Bs_fn, r→tolerance, &bs, &r→AD_iterations);
      /* recover true values */
    r→slab.a = bcalc2b(bs)/(bcalc2b(bs) + r→slab.b);
    r→slab.b = bcalc2b(bs) + r→slab.b;
    ⟨ Put final values in result 239 ⟩
  }
```

This code is used in section 230.

243. 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 243⟩ ≡
  void U_Find_Ba(struct measure_type m, struct invert_type *r)
```

This code is used in sections 231 and 244.

244.

```
⟨Definition for U_Find_Ba 244⟩ ≡
  ⟨Prototype for U_Find_Ba 243⟩
  {
    double ax, bx, cx, fa, fb, fc, ba;
    if (Debug(DEBUG_SEARCH)) {
      fprintf(stderr, "SEARCH:␣Using␣U_Find_Bs()");
      fprintf(stderr, "␣(mu=%6.4f)", r→slab.cos_angle);
      if (r→default_bs ≠ UNINITIALIZED) fprintf(stderr, "␣␣default_bs␣=␣%8.5f", r→default_bs);
      if (r→default_g ≠ UNINITIALIZED) fprintf(stderr, "␣␣default_g␣=␣%8.5f", r→default_g);
      fprintf(stderr, "\n");
    }
    r→slab.a = 0;
    r→slab.g = (r→default_g ≡ UNINITIALIZED) ? 0 : r→default_g;
    r→slab.b = (r→default_bs ≡ UNINITIALIZED) ? HUGE_VAL : r→default_bs;
    if (m.m_t ≡ 0) {
      r→slab.b = HUGE_VAL;
      U_Find_A(m, r);
      return;
    }
    Set_Calc_State(m, *r);    /* store bs in RR.slab.b */
    ax = b2bcalc(0.1);    /* first try for ba */
    bx = b2bcalc(1.0);
    mnbrak(&ax, &bx, &cx, &fa, &fb, &fc, Find_Ba_fn, &r→AD_iterations);
    r→final_distance = brent(ax, bx, cx, Find_Ba_fn, r→tolerance, &ba, &r→AD_iterations);
      /* recover true values */
    r→slab.a = (r→slab.b)/(bcalc2b(ba) + r→slab.b);
    r→slab.b = bcalc2b(ba) + r→slab.b;    /* actual value of b */
    ⟨Put final values in result 239⟩
  }
```

This code is used in section 230.

245. 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 245⟩ ≡
  void U_Find_A(struct measure_type m, struct invert_type *r)
```

This code is used in sections 231 and 246.

246. ⟨Definition for *U_Find_A* 246⟩ ≡

```
  ⟨Prototype for U_Find_A 245⟩
  {
    double Rt, Tt, Rd, Ru, Td, Tu;
    if (Debug(DEBUG_SEARCH)) {
      fprintf(stderr, "SEARCH:␣Using␣U_Find_A()");
      fprintf(stderr, "␣(mu=%6.4f)", r→slab.cos_angle);
      if (r→default_b ≠ UNINITIALIZED) fprintf(stderr, "␣␣default_b␣=␣%8.5f", r→default_b);
      if (r→default_g ≠ UNINITIALIZED) fprintf(stderr, "␣␣default_g␣=␣%8.5f", r→default_g);
      fprintf(stderr, "\n");
    }
    Estimate_RT(m, *r, &Rt, &Tt, &Rd, &Ru, &Td, &Tu);
    r→slab.g = (r→default_g ≡ UNINITIALIZED) ? 0 : r→default_g;
    r→slab.b = (r→default_b ≡ UNINITIALIZED) ? HUGE_VAL : r→default_b;
    r→slab.a = 0.0;
    r→final_distance = 0.0;
    Set_Calc_State(m, *r);
    if (Rt > 0.99999) {
      r→final_distance = Find_A_fn(a2acalc(1.0));
      r→slab.a = 1.0;
    }
    else {
      double x, ax, bx, cx, fa, fb, fc;
      ax = a2acalc(0.3);
      bx = a2acalc(0.5);
      mnbrak(&ax, &bx, &cx, &fa, &fb, &fc, Find_A_fn, &r→AD_iterations);
      r→final_distance = brent(ax, bx, cx, Find_A_fn, r→tolerance, &x, &r→AD_iterations);
      r→slab.a = acalc2a(x);
    }
    ⟨Put final values in result 239⟩
  }
```

This code is used in section 230.

247. Fixed Optical Depth and Albedo.

```
⟨ Prototype for U_Find_G 247 ⟩ ≡
  void U_Find_G(struct measure_type m, struct invert_type *r)
```

This code is used in sections 231 and 248.

248.

```
⟨ Definition for U_Find_G 248 ⟩ ≡
  ⟨ Prototype for U_Find_G 247 ⟩
  {
    double Rt, Tt, Rd, Ru, Td, Tu;
    double x, ax, bx, cx, fa, fb, fc;
    if (Debug(DEBUG_SEARCH)) {
      fprintf(stderr, "SEARCH:␣Using␣U_Find_G()");
      fprintf(stderr, "␣(mu=%6.4f)", r→slab.cos_angle);
      if (r→default_a ≠ UNINITIALIZED) fprintf(stderr, "␣␣default_a␣=␣%8.5f", r→default_a);
      if (r→default_b ≠ UNINITIALIZED) fprintf(stderr, "␣␣default_b␣=␣%8.5f", r→default_b);
      fprintf(stderr, "\n");
    }
    Estimate_RT(m, *r, &Rt, &Tt, &Rd, &Ru, &Td, &Tu);
    r→slab.a = (r→default_a ≡ UNINITIALIZED) ? 0.5 : r→default_a;
    if (r→default_b ≠ UNINITIALIZED) r→slab.b = r→default_b;
    else if (m.m_u > 0) r→slab.b = What_Is_B(r→slab, m.m_u);
    else r→slab.b = HUGE_VAL;
    r→slab.g = 0.0;
    r→final_distance = 0.0;
    Set_Calc_State(m, *r);
    ax = g2gcalc(−0.99);
    bx = g2gcalc(0.99);
    mnbrak(&ax, &bx, &cx, &fa, &fb, &fc, Find_G_fn, &r→AD_iterations);
    r→final_distance = brent(ax, bx, cx, Find_G_fn, r→tolerance, &x, &r→AD_iterations);
    r→slab.g = gcalc2g(x);
    Set_Calc_State(m, *r);
    ⟨ Put final values in result 239 ⟩
  }
```

This code is used in section 230.

249. 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 249 ⟩ ≡
  void U_Find_B(struct measure_type m, struct invert_type *r)
```

This code is used in sections 231 and 250.

250.

```
⟨ Definition for U_Find_B 250 ⟩ ≡
  ⟨ Prototype for U_Find_B 249 ⟩
  {
    double Rt, Tt, Rd, Ru, Td, Tu;
    if (Debug(DEBUG_SEARCH)) {
      fprintf(stderr, "SEARCH:␣Using␣U_Find_B()");
      fprintf(stderr, "␣(mu=%6.4f)", r→slab.cos_angle);
      if (r→default_a ≠ UNINITIALIZED) fprintf(stderr, "␣␣default_a␣=␣%8.5f", r→default_a);
      if (r→default_g ≠ UNINITIALIZED) fprintf(stderr, "␣␣default_g␣=␣%8.5f", r→default_g);
      fprintf(stderr, "\n");
    }
    Estimate_RT(m, *r, &Rt, &Tt, &Rd, &Ru, &Td, &Tu);
    r→slab.g = (r→default_g ≡ UNINITIALIZED) ? 0 : r→default_g;
    r→slab.a = (r→default_a ≡ UNINITIALIZED) ? 0 : r→default_a;
    r→slab.b = 0.5;
    r→final_distance = 0.0;
    Set_Calc_State(m, *r);
    ⟨ Iteratively solve for b 251 ⟩
    ⟨ Put final values in result 239 ⟩
  }
```

This code is used in section 230.

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

```
⟨ Iteratively solve for b 251 ⟩ ≡
  {
    double x, ax, bx, cx, fa, fb, fc;
    ax = b2bcalc(0.1);
    bx = b2bcalc(10);
    mnbrak(&ax, &bx, &cx, &fa, &fb, &fc, Find_B_fn, &r→AD_iterations);
    r→final_distance = brent(ax, bx, cx, Find_B_fn, r→tolerance, &x, &r→AD_iterations);
    r→slab.b = bcalc2b(x);
    Set_Calc_State(m, *r);
  }
```

This code is used in section 250.

252. 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 252 ⟩ ≡
  void U_Find_AG(struct measure_type m, struct invert_type *r)
```

This code is used in sections 231 and 253.

253.

```
⟨ Definition for U_Find_AG 253 ⟩ ≡
  ⟨ Prototype for U_Find_AG 252 ⟩
  {
    ⟨ Allocate local simplex variables 234 ⟩
    if (Debug(DEBUG_SEARCH)) {
      fprintf(stderr, "SEARCH:␣Using␣U_Find_AG()");
      fprintf(stderr, "␣(mu=%6.4f)", r→slab.cos_angle);
      if (r→default_b ≠ UNINITIALIZED) fprintf(stderr, "␣␣default_b␣=␣%8.5f", r→default_b);
      fprintf(stderr, "\n");
    }
    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 235 ⟩
    ⟨ Initialize the nodes of the a and g simplex 254 ⟩
    ⟨ Evaluate the a and g simplex at the nodes 255 ⟩
    amoeba(p, y, 2, r→tolerance, Find_AG_fn, &r→AD_iterations);
    ⟨ Choose the best node of the a and g simplex 256 ⟩
    ⟨ Free simplex data structures 240 ⟩
    ⟨ Put final values in result 239 ⟩
  }
```

This code is used in section 230.

254. ⟨Initialize the nodes of the a and g simplex 254⟩ ≡

```
{
  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 (kk ≡ k) continue;
    if (guess[0].g ≠ guess[kk].g ∨ guess[k].g ≠ guess[kk].g) break;
  }
  p[3][1] = a2acalc(guess[kk].a);
  p[3][2] = g2gcalc(guess[kk].g);
  if (Debug(DEBUG_BEST_GUESS)) {
    fprintf(stderr, "-----------------------------------------------------\n");
    fprintf(stderr, "BEST:␣<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, "BEST:␣<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, "BEST:␣<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);
    fprintf(stderr, "\n");
  }
}
```

This code is used in section 253.

255. ⟨Evaluate the a and g simplex at the nodes 255⟩ ≡

```
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 253.

256. 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 256 ⟩ ≡
  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 253.

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

```
⟨ Prototype for U_Find_BG 257 ⟩ ≡
  void U_Find_BG(struct measure_type m, struct invert_type *r)
```

This code is used in sections 231 and 258.

258.

```
⟨ Definition for U_Find_BG 258 ⟩ ≡
  ⟨ Prototype for U_Find_BG 257 ⟩
  {
    ⟨ Allocate local simplex variables 234 ⟩
    if (Debug(DEBUG_SEARCH)) {
      fprintf(stderr, "SEARCH:␣Using␣U_Find_BG()");
      fprintf(stderr, "␣(mu=%6.4f)", r→slab.cos_angle);
      if (r→default_a ≠ UNINITIALIZED) fprintf(stderr, "␣␣default_a␣=␣%8.5f", r→default_a);
      fprintf(stderr, "\n");
    }
    r→slab.a = (r→default_a ≡ UNINITIALIZED) ? 0 : r→default_a;
    Set_Calc_State(m, *r);
    ⟨ Get the initial a, b, and g 235 ⟩
    ⟨ Initialize the nodes of the b and g simplex 260 ⟩
    ⟨ Evaluate the bg simplex at the nodes 261 ⟩
    amoeba(p, y, 2, r→tolerance, Find_BG_fn, &r→AD_iterations);
    ⟨ Choose the best node of the b and g simplex 262 ⟩
    ⟨ Free simplex data structures 240 ⟩
    ⟨ Put final values in result 239 ⟩
  }
```

This code is used in section 230.

259. 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.

260. ⟨ Initialize the nodes of the *b* and *g* simplex 260 ⟩ ≡

```
{
  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 (kk ≡ k) continue;
    if (guess[0].g ≠ guess[kk].g ∨ guess[k].g ≠ guess[kk].g) break;
  }
  p[3][1] = b2bcalc(guess[kk].b);
  p[3][2] = g2gcalc(guess[kk].g);
  if (Debug(DEBUG_BEST_GUESS)) {
    fprintf(stderr, "--------------------------------------------------\n");
    fprintf(stderr, "BEST:␣<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, "BEST:␣<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, "BEST:␣<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);
    fprintf(stderr, "\n");
  }
}
```

This code is used in section 258.

261. ⟨ Evaluate the *bg* simplex at the nodes 261 ⟩ ≡

```
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 258.

262. 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 262 ⟩ ≡
  r→final_distance = 10;
  for (i = 1; i ≤ 3; i++) {
    if (y[i] < r→final_distance) {
      r→slab.b = bcalc2b(p[i][1]);
      r→slab.g = gcalc2g(p[i][2]);
      r→final_distance = y[i];
    }
  }
```

This code is used in section 258.

263. 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 263 ⟩ ≡
  void U_Find_BaG(struct measure_type m, struct invert_type ∗r)
```

This code is used in sections 231 and 264.

264.

```
⟨ Definition for U_Find_BaG 264 ⟩ ≡
  ⟨ Prototype for U_Find_BaG 263 ⟩
  {
    ⟨ Allocate local simplex variables 234 ⟩
    Set_Calc_State(m, ∗r);
    ⟨ Get the initial a, b, and g 235 ⟩
    ⟨ Initialize the nodes of the ba and g simplex 265 ⟩
    ⟨ Evaluate the BaG simplex at the nodes 266 ⟩
    amoeba(p, y, 2, r→tolerance, Find_BaG_fn, &r→AD_iterations);
    ⟨ Choose the best node of the ba and g simplex 267 ⟩
    ⟨ Free simplex data structures 240 ⟩
    ⟨ Put final values in result 239 ⟩
  }
```

This code is used in section 230.

265.

```
⟨ Initialize the nodes of the ba and g simplex 265 ⟩ ≡
  if (guess[0].b > r→default_bs) {
    p[1][1] = b2bcalc(guess[0].b − r→default_bs);
    p[2][1] = b2bcalc(2 ∗ (guess[0].b − r→default_bs));
    p[3][1] = p[1][1];
  }
  else {
    p[1][1] = b2bcalc(0.0001);
    p[2][1] = b2bcalc(0.001);
    p[3][1] = p[1][1];
  }
  p[1][2] = g2gcalc(guess[0].g);
  p[2][2] = p[1][2];
  p[3][2] = g2gcalc(0.9 ∗ guess[0].g + 0.05);
```

This code is used in section 264.

266.

```
⟨ Evaluate the BaG simplex at the nodes 266 ⟩ ≡
  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 264.

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

```
⟨ Choose the best node of the ba and g simplex 267 ⟩ ≡
  r→final_distance = 10;
  for (i = 1; i ≤ 3; i++) {
    if (y[i] < r→final_distance) {
      r→slab.b = bcalc2b(p[i][1]) + r→default_bs;
      r→slab.a = r→default_bs/r→slab.b;
      r→slab.g = gcalc2g(p[i][2]);
      r→final_distance = y[i];
    }
  }
```

This code is used in section 264.

268. Fixed Absorption. Here I assume that we have 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 268 ⟩ ≡
  void U_Find_BsG(struct measure_type m, struct invert_type *r)
```

This code is used in sections 231 and 269.

269.

```
⟨ Definition for U_Find_BsG 269 ⟩ ≡
  ⟨ Prototype for U_Find_BsG 268 ⟩
  {
    ⟨ Allocate local simplex variables 234 ⟩
    if (Debug(DEBUG_SEARCH)) {
      fprintf(stderr, "SEARCH:␣Using␣U_Find_BsG()");
      fprintf(stderr, "␣(mu=%6.4f)", r→slab.cos_angle);
      if (r→default_ba ≠ UNINITIALIZED) fprintf(stderr, "␣␣default_ba␣=␣%8.5f", r→default_ba);
      fprintf(stderr, "\n");
    }
    Set_Calc_State(m, *r);
    ⟨ Get the initial a, b, and g 235 ⟩
    ⟨ Initialize the nodes of the bs and g simplex 270 ⟩
    ⟨ Evaluate the BsG simplex at the nodes 271 ⟩
    amoeba(p, y, 2, r→tolerance, Find_BsG_fn, &r→AD_iterations);
    ⟨ Choose the best node of the bs and g simplex 272 ⟩
    ⟨ Free simplex data structures 240 ⟩
    ⟨ Put final values in result 239 ⟩
  }
```

This code is used in section 230.

270.

```
⟨ Initialize the nodes of the bs and g simplex 270 ⟩ ≡
  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 269.

271.

```
⟨ Evaluate the BsG simplex at the nodes 271 ⟩ ≡
  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 269.

272. ⟨Choose the best node of the *bs* and *g* simplex 272⟩ ≡

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

This code is used in section 269.

273. IAD Utilities.

March 1995. Reincluded *quick_guess* code.

```
⟨ iad_util.c 273 ⟩ ≡
#include <math.h>
#include <float.h>
#include <stdio.h>
#include "nr_util.h"
#include "ad_globl.h"
#include "ad_frsnl.h"
#include "ad_bound.h"
#include "iad_type.h"
#include "iad_calc.h"
#include "iad_pub.h"
#include "iad_util.h"
  unsigned long g_util_debugging = 0;
  ⟨ Preprocessor definitions ⟩
  ⟨ Definition for What_Is_B 276 ⟩
  ⟨ Definition for Estimate_RT 282 ⟩
  ⟨ Definition for a2acalc 289 ⟩
  ⟨ Definition for acalc2a 291 ⟩
  ⟨ Definition for g2gcalc 293 ⟩
  ⟨ Definition for gcalc2g 295 ⟩
  ⟨ Definition for b2bcalc 297 ⟩
  ⟨ Definition for bcalc2b 299 ⟩
  ⟨ Definition for twoprime 301 ⟩
  ⟨ Definition for twounprime 303 ⟩
  ⟨ Definition for abgg2ab 305 ⟩
  ⟨ Definition for abgb2ag 307 ⟩
  ⟨ Definition for quick_guess 314 ⟩
  ⟨ Definition for Set_Debugging 327 ⟩
  ⟨ Definition for Debug 329 ⟩
  ⟨ Definition for Print_Invert_Type 331 ⟩
  ⟨ Definition for Print_Measure_Type 333 ⟩
```

274. ⟨ `iad_util.h` 274 ⟩ ≡

```
  ⟨ Prototype for What_Is_B 275 ⟩;
  ⟨ Prototype for Estimate_RT 281 ⟩;
  ⟨ Prototype for a2acalc 288 ⟩;
  ⟨ Prototype for acalc2a 290 ⟩;
  ⟨ Prototype for g2gcalc 292 ⟩;
  ⟨ Prototype for gcalc2g 294 ⟩;
  ⟨ Prototype for b2bcalc 296 ⟩;
  ⟨ Prototype for bcalc2b 298 ⟩;
  ⟨ Prototype for twoprime 300 ⟩;
  ⟨ Prototype for twounprime 302 ⟩;
  ⟨ Prototype for abgg2ab 304 ⟩;
  ⟨ Prototype for abgb2ag 306 ⟩;
  ⟨ Prototype for quick_guess 313 ⟩;
  ⟨ Prototype for Set_Debugging 326 ⟩;
  ⟨ Prototype for Debug 328 ⟩;
  ⟨ Prototype for Print_Invert_Type 330 ⟩;
  ⟨ Prototype for Print_Measure_Type 332 ⟩;
```

275. 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 275⟩ ≡
  double What_Is_B(struct AD_slab_type slab, double Tu)
```

This code is used in sections 274 and 276.

276.

```
⟨Definition for What_Is_B 276⟩ ≡
  ⟨Prototype for What_Is_B 275⟩
  {
    double r1, r2, t1, t2, mu_in_slab;
    ⟨Calculate specular reflection and transmission 277⟩
    ⟨Check for bad values of Tu 278⟩
    ⟨Solve if multiple internal reflections are not present 279⟩
    ⟨Find thickness when multiple internal reflections are present 280⟩
  }
```

This code is used in section 273.

277. 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* ≡ *r2*, but there are always those annoying special cases.

```
⟨Calculate specular reflection and transmission 277⟩ ≡
  Absorbing_Glass_RT(1.0, slab.n_top_slide, slab.n_slab, slab.cos_angle, slab.b_top_slide, &r1, &t1);
  mu_in_slab = Cos_Snell(1.0, slab.cos_angle, slab.n_slab);
  Absorbing_Glass_RT(slab.n_slab, slab.n_bottom_slide, 1.0, mu_in_slab, slab.b_bottom_slide, &r2, &t2);
```

This code is used in section 276.

278. 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 Tu 278⟩ ≡
  if (Tu ≤ 0) return (HUGE_VAL);
  if (Tu ≥ t1 * t2/(1 − r1 * r2)) return (0.001);
```

This code is used in section 276.

279. If either *r1* or *r2* ≡ 0 then things are very simple because the sample does not sustain multiple internal reflections and the unscattered transmission is

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

where b is the optical thickness and ν is *slab.cos_angle*. Clearly,

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

⟨ Solve if multiple internal reflections are not present 279 ⟩ ≡

```
  if (r1 ≡ 0 ∨ r2 ≡ 0) return (−slab.cos_angle ∗ log(Tu/t1/t2));
```

This code is used in section 276.

280. 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

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

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 280 ⟩ ≡

```
  {
    double B;
    B = t1 ∗ t2;
    return (−slab.cos_angle ∗ log(2 ∗ Tu/(B + sqrt(B ∗ B + 4 ∗ Tu ∗ Tu ∗ r1 ∗ r2))));
  }
```

This code is used in section 276.

281. 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 $Tu = 0$. 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* 281 ⟩ ≡

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

This code is used in sections 274 and 282.

282. ⟨ Definition for *Estimate_RT* 282 ⟩ ≡

```
⟨ Prototype for Estimate_RT 281 ⟩
{
  ⟨ Calculate the unscattered transmission and reflection 283 ⟩
  ⟨ Estimate the backscattered reflection 284 ⟩
  ⟨ Estimate the scattered transmission 285 ⟩
  ⟨ Debug info for estimate RT 286 ⟩
}
```

This code is used in section 273.

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

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

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

⟨ Calculate the unscattered transmission and reflection 283 ⟩ ≡

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

This code is used in section 282.

284. 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 284⟩ ≡

```
if (m.fraction_of_ru_in_mr) {
  ∗rt = m.m_r;
  ∗rd = ∗rt − m.fraction_of_ru_in_mr ∗ (∗rc);
  if (∗rd < 0) {
    ∗rd = 0;
    ∗rc = ∗rt;
  }
}
else {
  ∗rd = m.m_r;
  ∗rt = ∗rd + ∗rc;
}
```

This code is used in section 282.

285. 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 285⟩ ≡

```
if (m.fraction_of_tu_in_mt) {
  ∗tt = m.m_t;
  ∗td = ∗tt − ∗tc;
  if (∗td < 0) {
    ∗tc = ∗tt;
    ∗td = 0;
  }
}
else {
  ∗td = m.m_t;
  ∗tt = ∗td + ∗tc;
}
```

This code is used in section 282.

286. Collect debugging info here

⟨Debug info for estimate RT 286⟩ ≡

```
if (0 ∧ Debug(DEBUG_SEARCH)) {
  fprintf(stderr, "SEARCH:␣r_t␣=␣%8.5f␣", ∗rt);
  fprintf(stderr, "r_d␣=␣%8.5f␣", ∗rd);
  fprintf(stderr, "r_u␣=␣%8.5f\n", ∗rc);
  fprintf(stderr, "SEARCH:␣t_t␣=␣%8.5f␣", ∗tt);
  fprintf(stderr, "t_d␣=␣%8.5f␣", ∗td);
  fprintf(stderr, "t_u␣=␣%8.5f\n", ∗tc);
}
```

This code is used in section 282.

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

288. *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 288 ⟩ ≡
  double a2acalc(double a)
```

This code is used in sections 274 and 289.

289.

```
⟨ Definition for a2acalc 289 ⟩ ≡
  ⟨ Prototype for a2acalc 288 ⟩
  {
    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 273.

290. *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 290 ⟩ ≡
  double acalc2a(double acalc)
```

This code is used in sections 274 and 291.

291.

```
⟨ Definition for acalc2a 291 ⟩ ≡
  ⟨ Prototype for acalc2a 290 ⟩
  {
    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 273.

292. *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 292 ⟩ ≡
  double g2gcalc(double g)
```

This code is used in sections 274 and 293.

293.

```
⟨ Definition for g2gcalc 293 ⟩ ≡
  ⟨ Prototype for g2gcalc 292 ⟩
  {
    if (g ≤ −0.99999) return (−HUGE_VAL);
    if (g ≥ 0.99999) return (HUGE_VAL);
    return (g/(1 − fabs(g)));
  }
```

This code is used in section 273.

294. *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 294 ⟩ ≡
  double gcalc2g(double gcalc)
```

This code is used in sections 274 and 295.

295.

```
⟨ Definition for gcalc2g 295 ⟩ ≡
  ⟨ Prototype for gcalc2g 294 ⟩
  {
    if (gcalc ≡ −HUGE_VAL) return −0.99999;
    if (gcalc ≡ HUGE_VAL) return 0.99999;
    return (gcalc/(1 + fabs(gcalc)));
  }
```

This code is used in section 273.

296. *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 296 ⟩ ≡
  double b2bcalc(double b)
```

This code is used in sections 274 and 297.

297. ⟨ Definition for *b2bcalc* 297 ⟩ ≡

```
⟨ Prototype for b2bcalc 296 ⟩
{
  if (b ≡ HUGE_VAL) return HUGE_VAL;
  if (b ≤ 0) return 0.0;
  return (log(b));
}
```

This code is used in section 273.

298. *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* 298 ⟩ ≡

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

This code is used in sections 274 and 299.

299. ⟨ Definition for *bcalc2b* 299 ⟩ ≡

```
⟨ Prototype for bcalc2b 298 ⟩
{
  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 273.

300. *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* 300 ⟩ ≡

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

This code is used in sections 274 and 301.

301. ⟨ Definition for *twoprime* 301 ⟩ ≡

```
⟨ Prototype for twoprime 300 ⟩
{
  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 273.

302. *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 302 ⟩ ≡
  void twounprime(double ap, double bp, double g, double *a, double *b)
```

This code is used in sections 274 and 303.

303.
```
⟨ Definition for twounprime 303 ⟩ ≡
  ⟨ Prototype for twounprime 302 ⟩
  {
    *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 273.

304. *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 304 ⟩ ≡
  void abgg2ab(double a1, double b1, double g1, double g2, double *a2, double *b2)
```

This code is used in sections 274 and 305.

305.
```
⟨ Definition for abgg2ab 305 ⟩ ≡
  ⟨ Prototype for abgg2ab 304 ⟩
  {
    double a, b;

    twoprime(a1, b1, g1, &a, &b);
    twounprime(a, b, g2, a2, b2);
  }
```

This code is used in section 273.

306. *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 306 ⟩ ≡
  void abgb2ag(double a1, double b1, double b2, double *a2, double *g2)
```

This code is used in sections 274 and 307.

307. ⟨Definition for *abgb2ag* 307⟩ ≡

```
⟨Prototype for abgb2ag 306⟩
{
  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 273.

308. Guessing an inverse.

This routine is not used anymore.

```
⟨ Prototype for slow_guess 308 ⟩ ≡
  void slow_guess(struct measure_type m, struct invert_type ∗r, double ∗a, double ∗b, double ∗g)
```

This code is used in section 309.

309.

```
⟨ Definition for slow_guess 309 ⟩ ≡
  ⟨ Prototype for slow_guess 308 ⟩
  {
    double fmin = 10.0;
    double fval;
    double ∗x;

    x = dvector(1, 2);
    switch (r→search) {
    case FIND_A: ⟨ Slow guess for a alone 310 ⟩
      break;
    case FIND_B: ⟨ Slow guess for b alone 311 ⟩
      break;
    case FIND_AB: case FIND_AG: ⟨ Slow guess for a and b or a and g 312 ⟩
      break;
    }
    ∗a = r→slab.a;
    ∗b = r→slab.b;
    ∗g = r→slab.g;
    free_dvector(x, 1, 2);
  }
```

310.

```
⟨ Slow guess for a alone 310 ⟩ ≡
  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 309.

311. 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 311 ⟩ ≡
  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 309.

312. ⟨ Slow guess for a and b or a and g 312 ⟩ ≡

```
  {
    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 309.

313. ⟨ Prototype for *quick_guess* 313 ⟩ ≡

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

This code is used in sections 274 and 314.

314. ⟨ Definition for *quick_guess* 314 ⟩ ≡

```
  ⟨ Prototype for quick_guess 313 ⟩
  {
    double UR1, UT1, rd, td, tc, rc, bprime, aprime, alpha, beta, logr;
    Estimate_RT(m, r, &UR1, &UT1, &rd, &rc, &td, &tc);
    ⟨ Estimate aprime 315 ⟩
    switch (m.num_measures) {
    case 1: ⟨ Guess when only reflection is known 317 ⟩
      break;
    case 2: ⟨ Guess when reflection and transmission are known 318 ⟩
      break;
    case 3: ⟨ Guess when all three measurements are known 319 ⟩
      break;
    }
    ⟨ Clean up guesses 324 ⟩
  }
```

This code is used in section 273.

315. ⟨Estimate *aprime* 315⟩ ≡

```
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 314.

316. ⟨Estimate *bprime* 316⟩ ≡

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

This code is used in sections 318, 322, and 323.

317.

⟨Guess when only reflection is known 317⟩ ≡

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

This code is used in section 314.

318. ⟨Guess when reflection and transmission are known 318⟩ ≡

```
⟨Estimate bprime 316⟩
∗g = r.default_g;
∗a = aprime/(1 − ∗g + aprime ∗ ∗g);
∗b = bprime/(1 − ∗a ∗ ∗g);
```

This code is used in section 314.

319.
```
⟨ Guess when all three measurements are known 319 ⟩ ≡
  switch (r.search) {
  case FIND_A: ⟨ Guess when finding albedo 320 ⟩
    break;
  case FIND_B: ⟨ Guess when finding optical depth 321 ⟩
    break;
  case FIND_AB: ⟨ Guess when finding the albedo and optical depth 322 ⟩
    break;
  case FIND_AG: ⟨ Guess when finding anisotropy and albedo 323 ⟩
    break;
  }
```
This code is used in section 314.

320.
```
⟨ Guess when finding albedo 320 ⟩ ≡
  *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 319.

321.
```
⟨ Guess when finding optical depth 321 ⟩ ≡
  *g = r.default_g;
  *a = 0.0;
  *b = What_Is_B(r.slab, m.m_u);
```
This code is used in section 319.

322.
```
⟨ Guess when finding the albedo and optical depth 322 ⟩ ≡
  *g = r.default_g;
  if (*g ≡ 1) *a = 0.0;
  else *a = aprime/(1 − *g + aprime * *g);
  ⟨ Estimate bprime 316 ⟩
  if (bprime ≡ HUGE_VAL ∨ *a * *g ≡ 1) *b = HUGE_VAL;
  else *b = bprime/(1 − *a * *g);
```
This code is used in section 319.

323.
```
⟨ Guess when finding anisotropy and albedo 323 ⟩ ≡
  *b = What_Is_B(r.slab, m.m_u);
  if (*b ≡ HUGE_VAL ∨ *b ≡ 0) {
    *a = aprime;
    *g = r.default_g;
  }
  else {
    ⟨ Estimate bprime 316 ⟩
    *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 319.

324.

```
⟨ Clean up guesses 324 ⟩ ≡
  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 314.

325. Some debugging stuff.

326.

```
⟨ Prototype for Set_Debugging 326 ⟩ ≡
  void Set_Debugging(unsigned long debug_level)
```

This code is used in sections 274 and 327.

327.

```
⟨ Definition for Set_Debugging 327 ⟩ ≡
  ⟨ Prototype for Set_Debugging 326 ⟩
  {
    g_util_debugging = debug_level;
  }
```

This code is used in section 273.

328.

```
⟨ Prototype for Debug 328 ⟩ ≡
  int Debug(unsigned long mask)
```

This code is used in sections 274 and 329.

329.

```
⟨ Definition for Debug 329 ⟩ ≡
  ⟨ Prototype for Debug 328 ⟩
  {
    if (g_util_debugging & mask) return 1;
    else return 0;
  }
```

This code is used in section 273.

330.

```
⟨ Prototype for Print_Invert_Type 330 ⟩ ≡
  void Print_Invert_Type(struct invert_type r)
```

This code is used in sections 274 and 331.

331.

```
⟨ Definition for Print_Invert_Type 331 ⟩ ≡
  ⟨ Prototype for Print_Invert_Type 330 ⟩
  {
    fprintf (stderr, "\n");
    fprintf (stderr, "default␣␣a=%10.5f␣␣␣b=%10.5f␣␣␣␣g=%10.5f\n", r.default_a, r.default_b, r.default_g);
    fprintf (stderr, "slab␣␣␣␣␣a=%10.5f␣␣␣b=%10.5f␣␣␣␣g=%10.5f\n", r.slab.a, r.slab.b, r.slab.g);
    fprintf (stderr, "n␣␣␣␣␣␣top=%10.5f␣mid=%10.5f␣␣bot=%10.5f\n", r.slab.n_top_slide, r.slab.n_slab,
        r.slab.n_bottom_slide);
    fprintf (stderr, "thick␣␣top=%10.5f␣cos=%10.5f␣␣bot=%10.5f\n", r.slab.b_top_slide, r.slab.cos_angle,
        r.slab.b_bottom_slide);
    fprintf (stderr, "search␣=␣%d␣quadrature␣points␣=␣%d\n", r.search, r.method.quad_pts);
    fprintf (stderr, "default_a␣=␣%10.5f\n", r.default_a);
    fprintf (stderr, "default_b␣=␣%10.5f\n", r.default_b);
    fprintf (stderr, "default_g␣=␣%10.5f\n", r.default_g);
    fprintf (stderr, "default_mua␣=␣%10.5f\n", r.default_mua);
    fprintf (stderr, "default_mus␣=␣%10.5f\n", r.default_mus);
  }
```

This code is used in section 273.

332.

```
⟨ Prototype for Print_Measure_Type 332 ⟩ ≡
  void Print_Measure_Type(struct measure_type m)
```

This code is used in sections 274 and 333.

333.

```
⟨ Definition for Print_Measure_Type 333 ⟩ ≡
  ⟨ Prototype for Print_Measure_Type 332 ⟩
  {
    fprintf(stderr, "\n");
    fprintf(stderr, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Beam␣diameter␣=␣%7.1f␣mm\n", m.d_beam);
    fprintf(stderr, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Sample␣thickness␣=␣%7.1f␣mm\n", m.slab_thickness);
    fprintf(stderr, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Top␣slide␣thickness␣=␣%7.1f␣mm\n",
        m.slab_top_slide_thickness);
    fprintf(stderr, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Bottom␣slide␣thickness␣=␣%7.1f␣mm\n",
        m.slab_bottom_slide_thickness);
    fprintf(stderr, "#␣␣␣␣␣␣␣␣␣␣␣Sample␣index␣of␣refraction␣=␣%7.3f\n", m.slab_index);
    fprintf(stderr, "#␣␣␣␣␣␣␣␣Top␣slide␣index␣of␣refraction␣=␣%7.3f\n", m.slab_top_slide_index);
    fprintf(stderr, "#␣␣␣␣␣Bottom␣slide␣index␣of␣refraction␣=␣%7.3f\n", m.slab_bottom_slide_index);
    fprintf(stderr, "#␣␣␣␣Fraction␣unscattered␣light␣in␣M_R␣=␣%7.1f␣%%\n",
        m.fraction_of_ru_in_mr * 100);
    fprintf(stderr, "#␣␣␣␣Fraction␣unscattered␣light␣in␣M_T␣=␣%7.1f␣%%\n",
        m.fraction_of_tu_in_mt * 100);
    fprintf(stderr, "#␣\n");
    fprintf(stderr, "#␣Reflection␣sphere\n");
    fprintf(stderr, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣sphere␣diameter␣=␣%7.1f␣mm\n", m.d_sphere_r);
    fprintf(stderr, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣sample␣port␣diameter␣=␣%7.1f␣mm\n",
        2 * m.d_sphere_r * sqrt(m.as_r));
    fprintf(stderr, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣entrance␣port␣diameter␣=␣%7.1f␣mm\n",
        2 * m.d_sphere_r * sqrt(m.at_r));
    fprintf(stderr, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣detector␣port␣diameter␣=␣%7.1f␣mm\n",
        2 * m.d_sphere_r * sqrt(m.ad_r));
    fprintf(stderr, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣wall␣reflectance␣=␣%7.1f␣%%\n", m.rw_r * 100);
    fprintf(stderr, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣standard␣reflectance␣=␣%7.1f␣%%\n", m.rstd_r * 100);
    fprintf(stderr, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣detector␣reflectance␣=␣%7.1f␣%%\n", m.rd_r * 100);
    fprintf(stderr, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣spheres␣=␣%7d\n", m.num_spheres);
    fprintf(stderr, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣measures␣=␣%7d\n", m.num_measures);
    fprintf(stderr, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣method␣=␣%7d\n", m.method);
    fprintf(stderr, "area_r␣as=%10.5f␣␣ad=%10.5f␣␣␣␣ae=%10.5f␣␣aw=%10.5f\n", m.as_r, m.ad_r,
        m.at_r, m.aw_r);
    fprintf(stderr, "refls␣␣rd=%10.5f␣␣rw=%10.5f␣␣rstd=%10.5f␣␣␣f=%10.5f\n", m.rd_r, m.rw_r,
        m.rstd_r, m.f_r);
    fprintf(stderr, "area_t␣as=%10.5f␣␣ad=%10.5f␣␣␣␣ae=%10.5f␣␣aw=%10.5f\n", m.as_t, m.ad_t,
        m.at_t, m.aw_t);
    fprintf(stderr, "refls␣␣rd=%10.5f␣␣rw=%10.5f␣␣rstd=%10.5f\n", m.rd_t, m.rw_t, m.rstd_t);
    fprintf(stderr, "lost␣␣ur1=%10.5f␣ut1=%10.5f␣␣␣uru=%10.5f␣␣utu=%10.5f\n", m.ur1_lost,
        m.ut1_lost, m.utu_lost, m.utu_lost);
  }
```

This code is used in section 273.

334. 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.