Inverse Adding-Doubling

(Version 3-15-1)

1. `iad` program.

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

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

⟨ `iad_main.h` 1 ⟩ ≡

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

⟨ `iad_main.c` 2 ⟩ ≡
  ⟨ Include files for *main* 3 ⟩
  ⟨ print version function 18 ⟩
  ⟨ print usage function 19 ⟩
  ⟨ stringdup together function 25 ⟩
  ⟨ mystrtod function 26 ⟩
  ⟨ seconds elapsed function 27 ⟩
  ⟨ print error legend function 24 ⟩
  ⟨ print dot function 29 ⟩
  ⟨ calculate coefficients function 20 ⟩
  ⟨ parse string into array function 28 ⟩
  ⟨ print results header function 22 ⟩
  ⟨ Print results function 23 ⟩

  int *main*(**int** *argc*, **char** ∗∗*argv*)
  {
    ⟨ Declare variables for *main* 4 ⟩
    ⟨ Handle options 5 ⟩
    Initialize_Measure(&*m*);
    ⟨ Command-line changes to *m* 16 ⟩
    Initialize_Result(*m*, &*r*);
    ⟨ Command-line changes to *r* 13 ⟩
    if (*cl_forward_calc* ≠ `UNINITIALIZED`) {
      ⟨ Calculate and Print the Forward Calculation 6 ⟩
      exit(`EXIT_SUCCESS`);
    }
    ⟨ prepare file for reading 10 ⟩
    if (*process_command_line*) {
      ⟨ Count command-line measurements 17 ⟩
      ⟨ Calculate and write optical properties 11 ⟩
      exit(`EXIT_SUCCESS`);
    }
    if (*Read_Header*(*stdin*, &*m*, &*params*) ≡ 0) {
      start_time = *clock*( );
      while (*Read_Data_Line*(*stdin*, &*m*, *params*) ≡ 0) {
        ⟨ Command-line changes to *m* 16 ⟩
        ⟨ Calculate and write optical properties 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"

See also section 31.

This code is used in sections 2 and 30.

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

    struct measure_type m;

    struct invert_type r;

    char $*g\_out\_name = \Lambda$;

    int c;

    long $n\_photons = 100000$;

    int $MC\_MAX\_iterations = 19$;

    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_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_rc_fraction = $ UNINITIALIZED;

    double $cl_tc_fraction = $ UNINITIALIZED;

    double $cl_gain_type = $ UNINITIALIZED;

    double $cl_lambda = $ 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:L:M:n:N:o:p:q:r:R:s:S:\
    t:T:u:vV:x:Xy:z";

See also section 32.

This code is used in sections 2 and 30.

5.   use the *getopt*( ) to process options.

⟨ Handle options 5 ⟩ ≡
  while $((c = getopt(argc, argv, command\_line\_options)) \neq$ 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 $(cl\_sphere\_one[4] \equiv$ 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_empty␣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 $(cl\_sphere\_two[4] \equiv$ 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_empty␣d_detector␣r_wall'\n");
        exit(EXIT_FAILURE);
      }
      break;
    case 'a': *cl_default_a* = *my_strtod*(*optarg*);
      if $(cl\_default\_a < 0 \lor cl\_default\_a > 1)$ {
        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_rc_fraction = my_strtod(optarg);
      if (cl_rc_fraction < 0.0 ∨ cl_rc_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_tc_fraction = my_strtod(optarg);
      if (cl_tc_fraction < 0.0 ∨ cl_tc_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 'L': cl_lambda = my_strtod(optarg);
    break;
case 'M': MC_MAX_iterations = (int) my_strtod(optarg);
    if (MC_MAX_iterations < 0 ∨ MC_MAX_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 'x': Set_Debugging((int) my_strtod(optarg));
      break;
   case 'X': cl_method = COMPARISON;
      break;
   case 'y': cl_gain_type = (int) my_strtod(optarg);
      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.

6.   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 6 ⟩ ≡
  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 7, 8, and 9.

This code is used in section 2.

7.   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 6 ⟩ +≡
  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*;

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

⟨ Calculate and Print the Forward Calculation 6 ⟩ +≡
  if (*cl_default_g* ≡ UNINITIALIZED) *r.g* = 0;
  else *r.g* = *cl_default_g*;

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

$r.slab.a = r.a;$

$r.slab.b = r.b;$

$r.slab.g = r.g;$

{

   double $mu\_sp, mu\_a, m\_r, m\_t;$

   $Calculate\_MR\_MT(m, r, \text{TRUE}, \text{TRUE}, \& m\_r, \& m\_t);$

   $Calculate\_Mua\_Musp(m, r, \& mu\_sp, \& mu\_a);$

   if $(cl\_verbosity > 0)$ {

     $Write\_Header(m, r, -1);$

     $print\_results\_header(stdout);$

   }

   $print\_optical\_property\_result(stdout, m, r, m\_r, m\_t, mu\_a, mu\_sp, 0);$

}

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

⟨ prepare file for reading 10 ⟩ ≡

  if $(argc > 1)$ {

    $fprintf(stderr, \texttt{"Only␣a␣single␣file␣can␣be␣processed␣at␣a␣time\n"});$

    $fprintf(stderr, \texttt{"try␣'apply␣iad␣file1␣file2␣...␣fileN'\n"});$

    $exit(\text{EXIT\_FAILURE});$

  }

  if $(argc \equiv 1 \wedge strcmp(argv[0], \texttt{"-"}) \neq 0)$ {   /* filename exists and != "−" */

    int $n;$

    char $*base\_name, *rt\_name;$

    $base\_name = strdup(argv[0]);$

    $n = (\textbf{int})(strlen(base\_name) - strlen(\texttt{".rxt"}));$

    if $(n > 0 \wedge strstr(base\_name + n, \texttt{".rxt"}) \neq \Lambda)$ $base\_name[n] = \texttt{'\0'};$

    $rt\_name = strdup\_together(base\_name, \texttt{".rxt"});$

    if $(freopen(argv[0], \texttt{"r"}, stdin) \equiv \Lambda \wedge freopen(rt\_name, \texttt{"r"}, stdin) \equiv \Lambda)$ {

      $fprintf(stderr, \texttt{"Could␣not␣open␣either␣'%s'␣or␣'%s'\n"}, argv[0], rt\_name);$

      $exit(\text{EXIT\_FAILURE});$

    }

    if $(g\_out\_name \equiv \Lambda)$ $g\_out\_name = strdup\_together(base\_name, \texttt{".txt"});$

    $free(rt\_name);$

    $free(base\_name);$

    $process\_command\_line = 0;$

  }

  if $(g\_out\_name \neq \Lambda)$ {

    if $(freopen(g\_out\_name, \texttt{"w"}, stdout) \equiv \Lambda)$ {

      $fprintf(stderr, \texttt{"Could␣not␣open␣file␣'%s'␣for␣output\n"}, g\_out\_name);$

      $exit(\text{EXIT\_FAILURE});$

    }

  }

This code is used in section 2.

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

⟨ Calculate and write optical properties 11 ⟩ ≡
  {
    ⟨ Local Variables for Calculation 12 ⟩
    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*);
    ⟨ Command-line changes to *r* 13 ⟩
    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);
    }
    ⟨ Write Header 14 ⟩
    m.ur1_lost = 0;
    m.uru_lost = 0;
    m.ut1_lost = 0;
    m.utu_lost = 0;
    Inverse_RT(*m*, &*r*);
    if (*r.error* ≡ IAD_NO_ERROR) {
      calculate_coefficients(*m*, *r*, &LR, &LT, &*mu_sp*, &*mu_a*);
      ⟨ Improve result using Monte Carlo 15 ⟩
    }
    print_optical_property_result(*stdout*, *m*, *r*, LR, LT, *mu_a*, *mu_sp*, *rt_total*);
    if (*Debug*(DEBUG_LOST_LIGHT)) {
      fprintf (*stderr*, "\n");
    }
    else {
      if (*r.error* ≠ IAD_NO_ERROR) *any_error* = 1;
    }
    print_dot(*start_time*, *r.error*, *mc_total*, TRUE, *cl_verbosity*);
  }

See also section 34.

This code is used in sections 2 and 30.

12.

⟨ Local Variables for Calculation 12 ⟩ ≡
  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 11.

13. ⟨ Command-line changes to r 13 ⟩ ≡
  if ($cl\_quadrature\_points \neq$ UNINITIALIZED) $r.method.quad\_pts = cl\_quadrature\_points$;
  else $r.method.quad\_pts = 8$;
  if ($cl\_default\_a \neq$ UNINITIALIZED) $r.default\_a = cl\_default\_a$;
  if ($cl\_default\_mua \neq$ UNINITIALIZED) {
    $r.default\_mua = cl\_default\_mua$;
    if ($cl\_sample\_d \neq$ UNINITIALIZED) $r.default\_ba = cl\_default\_mua * cl\_sample\_d$;
    else $r.default\_ba = cl\_default\_mua * m.slab\_thickness$;
  }
  if ($cl\_default\_b \neq$ UNINITIALIZED) $r.default\_b = cl\_default\_b$;
  if ($cl\_default\_g \neq$ UNINITIALIZED) $r.default\_g = cl\_default\_g$;
  if ($cl\_tolerance \neq$ UNINITIALIZED) {
    $r.tolerance = cl\_tolerance$;
    $r.MC\_tolerance = cl\_tolerance$;
  }
  if ($cl\_musp0 \neq$ UNINITIALIZED)
    $cl\_mus0 = (r.default\_g \neq$ UNINITIALIZED) ? $cl\_musp0 / (1 - r.default\_g) : cl\_musp0$;
  if ($cl\_mus0 \neq$ UNINITIALIZED $\wedge m.lambda \neq 0$)
    $cl\_default\_mus = cl\_mus0 * pow(m.lambda / cl\_mus0\_lambda, cl\_mus0\_pwr)$;
  if ($cl\_default\_mus \neq$ UNINITIALIZED) {
    $r.default\_mus = cl\_default\_mus$;
    if ($cl\_sample\_d \neq$ UNINITIALIZED) $r.default\_bs = cl\_default\_mus * cl\_sample\_d$;
    else $r.default\_bs = cl\_default\_mus * m.slab\_thickness$;
  }
  if ($cl\_search \neq$ UNINITIALIZED) $r.search = cl\_search$;

This code is used in sections 2 and 11.

14. ⟨ Write Header 14 ⟩ ≡
  if $(rt\_total \equiv 1 \wedge cl\_verbosity > 0)$ {
    $Write\_Header(m, r, params)$;
    if $(MC\_MAX\_iterations > 0)$ {
      if $(n\_photons \geq 0)$
        $fprintf(stdout, \texttt{"\#}_{\sqcup\sqcup}\texttt{Photons}_{\sqcup}\texttt{used}_{\sqcup}\texttt{to}_{\sqcup}\texttt{estimate}_{\sqcup}\texttt{lost}_{\sqcup}\texttt{light}_{\sqcup}\texttt{=}_{\sqcup\sqcup\sqcup}\texttt{\%ld\textbackslash n"}, n\_photons)$;
      else $fprintf(stdout, \texttt{"\#}_{\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup}\texttt{Time}_{\sqcup}\texttt{used}_{\sqcup}\texttt{to}_{\sqcup}\texttt{estimate}_{\sqcup}\texttt{lost}_{\sqcup}\texttt{light}_{\sqcup}\texttt{=}_{\sqcup\sqcup\sqcup}\texttt{\%ld}_{\sqcup}\texttt{ms\textbackslash n"}, -n\_photons)$;
    }
    else $fprintf(stdout, \texttt{"\#}_{\sqcup\sqcup}\texttt{Photons}_{\sqcup}\texttt{used}_{\sqcup}\texttt{to}_{\sqcup}\texttt{estimate}_{\sqcup}\texttt{lost}_{\sqcup}\texttt{light}_{\sqcup}\texttt{=}_{\sqcup\sqcup\sqcup}\texttt{0\textbackslash n"})$;
    $fprintf(stdout, \texttt{"\#\textbackslash n"})$;
    $print\_results\_header(stdout)$;
  }

This code is used in section 11.

15. 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 15 ⟩ ≡
  if $(m.as\_r \neq 0 \wedge r.default\_a \neq 0)$ {
    double $mu\_sp\_last = mu\_sp$;
    double $mu\_a\_last = mu\_a$;
    if $(Debug(\texttt{DEBUG\_LOST\_LIGHT}))$ {
      $print\_results\_header(stderr)$;
      $print\_optical\_property\_result(stderr, m, r, \texttt{LR}, \texttt{LT}, mu\_a, mu\_sp, rt\_total)$;
    }
    while $(r.MC\_iterations < MC\_MAX\_iterations)$ {
      if $(Debug(\texttt{DEBUG\_ITERATIONS}))$
        $fprintf(stderr, \texttt{"\textbackslash n−−−−−−−−−−−−}_{\sqcup}\texttt{Monte}_{\sqcup}\texttt{Carlo}_{\sqcup}\texttt{Iteration}_{\sqcup}\texttt{\%d}_{\sqcup}\texttt{−−−−−−−−−−−−−−−−−\textbackslash n"}$,
          $r.MC\_iterations + 1)$;
      $MC\_Lost(m, r, n\_photons, \&ur1, \&ut1, \&uru, \&utu, \&m.ur1\_lost, \&m.ut1\_lost, \&m.uru\_lost,$
        $\&m.utu\_lost)$;
      $mc\_total{+}{+}$;
      $r.MC\_iterations{+}{+}$;
      $Inverse\_RT(m, \&r)$;
      $calculate\_coefficients(m, r, \&\texttt{LR}, \&\texttt{LT}, \&mu\_sp, \&mu\_a)$;
      if $(fabs(mu\_a\_last - mu\_a)/(mu\_a + 0.0001) < r.MC\_tolerance \wedge fabs(mu\_sp\_last - mu\_sp)/(mu\_sp +$
        $0.0001) < r.MC\_tolerance)$ **break**;
      $mu\_a\_last = mu\_a$;
      $mu\_sp\_last = mu\_sp$;
      if $(Debug(\texttt{DEBUG\_LOST\_LIGHT}))$
        $print\_optical\_property\_result(stderr, m, r, \texttt{LR}, \texttt{LT}, mu\_a, mu\_sp, rt\_total)$;
      else $print\_dot(start\_time, r.error, mc\_total, \texttt{FALSE}, cl\_verbosity)$;
      if $(r.error \neq \texttt{IAD\_NO\_ERROR})$ **break**;
    }
  }

This code is used in section 11.

16.   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 16 ⟩ ≡
  if ($cl\_cos\_angle \neq$ UNINITIALIZED) {
    $m.slab\_cos\_angle = cl\_cos\_angle$;
    if ($cl\_quadrature\_points \equiv$ UNINITIALIZED) $cl\_quadrature\_points = 12$;
    if ($cl\_quadrature\_points \neq 12 * (cl\_quadrature\_points/12)$) {
      $fprintf$($stderr$,
        "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 ($cl\_sample\_n \neq$ UNINITIALIZED) $m.slab\_index = cl\_sample\_n$;
  if ($cl\_slide\_n \neq$ UNINITIALIZED) {
    $m.slab\_bottom\_slide\_index = cl\_slide\_n$;
    $m.slab\_top\_slide\_index = cl\_slide\_n$;
  }
  if ($cl\_slide\_OD \neq$ UNINITIALIZED) {
    $m.slab\_bottom\_slide\_b = cl\_slide\_OD$;
    $m.slab\_top\_slide\_b = cl\_slide\_OD$;
  }
  if ($cl\_sample\_d \neq$ UNINITIALIZED) $m.slab\_thickness = cl\_sample\_d$;
  if ($cl\_beam\_d \neq$ UNINITIALIZED) $m.d\_beam = cl\_beam\_d$;
  if ($cl\_slide\_d \neq$ UNINITIALIZED) {
    $m.slab\_bottom\_slide\_thickness = cl\_slide\_d$;
    $m.slab\_top\_slide\_thickness = cl\_slide\_d$;
  }
  if ($cl\_slides \equiv$ 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 ($cl\_slides \equiv$ ONE_SLIDE_ON_TOP $\vee cl\_slides \equiv$ ONE_SLIDE_NEAR_SPHERE) {
    $m.slab\_bottom\_slide\_index = 1.0$;
    $m.slab\_bottom\_slide\_thickness = 0.0$;
  }
  if ($cl\_slides \equiv$ ONE_SLIDE_ON_BOTTOM $\vee cl\_slides \equiv$ ONE_SLIDE_NOT_NEAR_SPHERE) {
    $m.slab\_top\_slide\_index = 1.0$;
    $m.slab\_top\_slide\_thickness = 0.0$;
  }
  if ($cl\_slides \equiv$ ONE_SLIDE_NEAR_SPHERE $\vee cl\_slides \equiv$ ONE_SLIDE_NOT_NEAR_SPHERE) $m.flip\_sample = 1$;
  else $m.flip\_sample = 0$;
  if ($cl\_method \neq$ UNINITIALIZED) $m.method = ($**int**$) cl\_method$;
  if ($cl\_rstd\_t \neq$ UNINITIALIZED) $m.rstd\_t = cl\_rstd\_t$;
  if ($cl\_rstd\_r \neq$ UNINITIALIZED) $m.rstd\_r = cl\_rstd\_r$;
  if ($cl\_sphere\_one[0] \neq$ UNINITIALIZED) {
    double $d\_sample\_r, d\_empty\_r, d\_detector\_r$;

    $m.d\_sphere\_r = cl\_sphere\_one[0]$;
    $d\_sample\_r = cl\_sphere\_one[1]$;

$d\_empty\_r = cl\_sphere\_one[2];$
$d\_detector\_r = cl\_sphere\_one[3];$
$m.rw\_r = cl\_sphere\_one[4];$
$m.as\_r = (d\_sample\_r/m.d\_sphere\_r/2) * (d\_sample\_r/m.d\_sphere\_r/2);$
$m.ae\_r = (d\_empty\_r/m.d\_sphere\_r/2) * (d\_empty\_r/m.d\_sphere\_r/2);$
$m.ad\_r = (d\_detector\_r/m.d\_sphere\_r/2) * (d\_detector\_r/m.d\_sphere\_r/2);$
$m.aw\_r = 1.0 - m.as\_r - m.ae\_r - m.ad\_r;$
$m.d\_sphere\_t = m.d\_sphere\_r;$
$m.as\_t = m.as\_r;$
$m.ae\_t = m.ae\_r;$
$m.ad\_t = m.ad\_r;$
$m.aw\_t = m.aw\_r;$
$m.rw\_t = m.rw\_r;$
if $(cl\_num\_spheres \equiv \texttt{UNINITIALIZED})$ $m.num\_spheres = 1;$
}
if $(cl\_sphere\_two[0] \neq \texttt{UNINITIALIZED})$ {
  double $d\_sample\_t, d\_empty\_t, d\_detector\_t;$

  $m.d\_sphere\_t = cl\_sphere\_two[0];$
  $d\_sample\_t = cl\_sphere\_two[1];$
  $d\_empty\_t = cl\_sphere\_two[2];$
  $d\_detector\_t = cl\_sphere\_two[3];$
  $m.rw\_t = cl\_sphere\_two[4];$
  $m.as\_t = (d\_sample\_t/m.d\_sphere\_t/2) * (d\_sample\_t/m.d\_sphere\_t/2);$
  $m.ae\_t = (d\_empty\_t/m.d\_sphere\_t/2) * (d\_empty\_t/m.d\_sphere\_t/2);$
  $m.ad\_t = (d\_detector\_t/m.d\_sphere\_t/2) * (d\_detector\_t/m.d\_sphere\_t/2);$
  $m.aw\_t = 1.0 - m.as\_t - m.ae\_t - m.ad\_t;$
  if $(cl\_num\_spheres \equiv \texttt{UNINITIALIZED})$ $m.num\_spheres = 2;$
}
if $(cl\_num\_spheres \neq \texttt{UNINITIALIZED})$ {
  $m.num\_spheres = (\textbf{int})\, cl\_num\_spheres;$
  if $(m.num\_spheres > 0 \wedge m.method \equiv \texttt{UNKNOWN})$ $m.method = \texttt{SUBSTITUTION};$
}
if $(cl\_rc\_fraction \neq \texttt{UNINITIALIZED})$ $m.fraction\_of\_rc\_in\_mr = cl\_rc\_fraction;$
if $(cl\_tc\_fraction \neq \texttt{UNINITIALIZED})$ $m.fraction\_of\_tc\_in\_mt = cl\_tc\_fraction;$
if $(cl\_gain\_type \neq \texttt{UNINITIALIZED})$ $m.gain\_type = cl\_gain\_type;$
if $(cl\_UR1 \neq \texttt{UNINITIALIZED})$ $m.m\_r = cl\_UR1;$
if $(cl\_UT1 \neq \texttt{UNINITIALIZED})$ $m.m\_t = cl\_UT1;$
if $(cl\_Tc \neq \texttt{UNINITIALIZED})$ $m.m\_u = cl\_Tc;$
if $(cl\_default\_fr \neq \texttt{UNINITIALIZED})$ $m.f\_r = cl\_default\_fr;$
if $(cl\_baffle\_r \neq \texttt{UNINITIALIZED})$ $m.baffle\_r = cl\_baffle\_r;$
if $(cl\_baffle\_t \neq \texttt{UNINITIALIZED})$ $m.baffle\_t = cl\_baffle\_t;$
if $(cl\_lambda \neq \texttt{UNINITIALIZED})$ $m.lambda = cl\_lambda;$

This code is used in section 2.

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

⟨ Count command-line measurements 17 ⟩ ≡
  $m.num\_measures = 3$;
  if $(m.m\_t \equiv 0)$ $m.num\_measures --$;
  if $(m.m\_u \equiv 0)$ $m.num\_measures --$;
  $params = m.num\_measures$;
  if $(m.num\_measures \equiv 3)$ {   /∗ need to fill slab entries to calculate the optical thickness ∗/
    struct AD_slab_type s;

    $s.n\_slab = m.slab\_index$;
    $s.n\_top\_slide = m.slab\_top\_slide\_index$;
    $s.n\_bottom\_slide = m.slab\_bottom\_slide\_index$;
    $s.b\_top\_slide = m.slab\_top\_slide\_b$;
    $s.b\_bottom\_slide = m.slab\_bottom\_slide\_b$;
    $s.cos\_angle = m.slab\_cos\_angle$;
    $cl\_default\_b = What\_Is\_B(s, m.m\_u)$;
  }

This code is used in section 2.

18.   ⟨ print version function 18 ⟩ ≡
  static void $print\_version($**int** $verbosity)$
  {
    if $(verbosity \equiv 0)$ {
      $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.

19.   ⟨ print usage function 19 ⟩ ≡
  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_empty_port␣d\
        _detector_port␣r_wall'\n");
    fprintf(*stdout*, "␣␣-2␣'#␣#␣#␣#␣#'␣␣␣transmission␣sphere␣parameters␣\n");
    fprintf(*stdout*, "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣'d_sphere␣d␣d_sample_port␣d_empty_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*, "␣␣-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*, "␣␣-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␣␣␣␣␣␣␣DEBUG_EVERY_CALC\n");
fprintf (*stdout*, "␣␣iad␣-x␣255␣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.

20.   Just figure out the damn scattering and absorption

⟨ calculate coefficients function 20 ⟩ ≡
```
  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 21.

This code is used in section 2.

21.   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 20 ⟩ +≡
```
  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);
  }
```

22.   ⟨ print results header function 22 ⟩ ≡
  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.

23.   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 23 ⟩ ≡
  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.

24. ⟨ print error legend function 24 ⟩ ≡
  static void *print_error_legend*(**void**)
  {
    fprintf(*stderr*, "−−−−−−−−−−−−−−−−−␣Sorry,␣but␣...␣errors␣encountered␣−−−−−−−−−−−−−−−\n");
    fprintf(*stderr*, "␣␣␣*␣␣==>␣Success␣␣␣␣␣␣␣␣␣␣␣␣");
    fprintf(*stderr*, "␣␣0−9␣==>␣Monte␣Carlo␣Iteration\n");
    fprintf(*stderr*, "␣␣␣R␣␣==>␣M_R␣is␣too␣big␣␣␣");
    fprintf(*stderr*, "␣␣␣r␣␣==>␣M_R␣is␣too␣small\n");
    fprintf(*stderr*, "␣␣␣T␣␣==>␣M_T␣is␣too␣big␣␣␣");
    fprintf(*stderr*, "␣␣␣t␣␣==>␣M_T␣is␣too␣small\n");
    fprintf(*stderr*, "␣␣␣U␣␣==>␣M_U␣is␣too␣big␣␣␣");
    fprintf(*stderr*, "␣␣␣u␣␣==>␣M_U␣is␣too␣small\n");
    fprintf(*stderr*, "␣␣␣!␣␣==>␣M_R␣+␣M_T␣>␣1␣␣␣␣␣");
    fprintf(*stderr*, "␣␣␣+␣␣==>␣Did␣not␣converge\n\n");
  }

This code is used in section 2.

25.   returns a new string consisting of s+t

⟨ stringdup together function 25 ⟩ ≡
  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.

26.   catch parsing errors in strtod

⟨ mystrtod function 26 ⟩ ≡
  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.

27.   assume that start time has already been set

⟨ seconds elapsed function 27 ⟩ ≡
  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.

28.   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 28 ⟩ ≡

```
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.

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

⟨ print dot function 29 ⟩ ≡

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

static void print_dot(clock_t start_time, int err, int 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.

30. Simple command-line shell program.
    Here is a quick skeleton that I put together to show how the inverse adding-doubling code works. I have only cursorily tested this. If you find obvious bugs, they are probably real but should not extend beyond this code snippet.
    All the output for this web file goes into `iad_main.c`

⟨`iad_main_mus.c` 30⟩ ≡
  ⟨Include files for *main* 3⟩
  int *main*(**int** *argc*, **char** ∗∗*argv*)
  {
    ⟨Declare variables for *main* 4⟩
    if (*Read_Header*(&*m*, &*r*) ≡ `TRUE`) {
      ⟨Process the header 33⟩
      m.num_measures = 2;
      m.m_r = 0.0;
      m.slab_thickness = 0.1;
      while (*fp* ≠ `EOF`) {
        fp = *scanf*("`%lf␣%lf␣%lf␣%lf`", &*lambda*, &*r.mu_a*, &*m.default_g*, &*m.m_t*);
        fp = *readln*(&*line*);
        ⟨Calculate and write optical properties 11⟩
      }
    }
    exit(`EXIT_SUCCESS`);
  }

31. ⟨Include files for *main* 3⟩ +≡
#**include** `<stdio.h>`
#**include** `<string.h>`
#**include** `<stdlib.h>`
#**include** `"ad_globl.h"`
#**include** `"iad_type.h"`
#**include** `"iad_pub.h"`
#**include** `"iad_io.h"`

32. ⟨Declare variables for *main* 4⟩ +≡
  struct measure_type *m*;
  struct invert_type *r*;
  int *lines*;
  char ∗*format1* = "`␣␣␣␣R␣␣␣␣\t␣␣␣␣T␣␣␣␣\t␣␣␣␣Tc␣␣␣␣\t␣␣␣␣␣a␣␣␣␣\t␣␣␣␣␣b␣␣␣␣\t␣\`
      `␣␣␣␣g␣␣␣␣\t␣␣␣␣␣!/?\n`";
  char ∗*format2* = "`␣%9.5f\t␣%9.5f\t␣%9.5f\t␣%9.5f\t␣%9.5f\t␣%9.5f\t␣%9c\n`";
  char *found* = '?';
  int *fp*;
  double *lambda*;
  int *line* = 1;

33. ⟨Process the header 33⟩ ≡
  m.slab_thickness = 1;
  Initialize_Result(*m*, &*r*);
  Write_Header(*m*, *r*);
  lines = 1;
  printf(*format1*, *m.m_r*, *m.m_t*, *m.m_u*, *r.a*, *r.b*, *r.g*);
  This code is used in section 30.

34. ⟨ Calculate and write optical properties 11 ⟩ +≡

```
{
    r.search = FIND_mus;
    Inverse_RT(m, &r);
    if (r.found ≡ TRUE) found = '!';
    else found = '?';
    printf(format2, m.m_r, m.m_t, m.m_u, r.a, r.b, r.g, found);
    fflush(stdout);
}
```

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_EVERY_CALC   128
#**define** DEBUG_ANY   #FFFFFFFF
#**define** UNKNOWN   0
#**define** COMPARISON   1
#**define** SUBSTITUTION   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 $gain\_type$;

      int $baffle\_r, baffle\_t$;

      double d_beam;

      double $fraction\_of\_rc\_in\_mr$;

      double $fraction\_of\_tc\_in\_mt$;

      double m_r, m_t, m_u;

      double $lambda$;

      double $as\_r, ad\_r, ae\_r, aw\_r, rd\_r, rw\_r, rstd\_r, f\_r$;

      double $as\_t, ad\_t, ae\_t, aw\_t, rd\_t, rw\_t, rstd\_t, f\_t$;

      double $ur1\_lost, uru\_lost, ut1\_lost, utu\_lost$;

      double d_sphere_r, d_sphere_t;

   } **IAD_measure_type**;

See also sections 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 ⟩
  {
     $r{\to}found = $ `FALSE`;
     if ($r{\to}search \equiv$ `FIND_AUTO`) $r{\to}search = determine\_search(m, *r)$;
     if ($r{\to}search \equiv$ `FIND_B_WITH_NO_ABSORPTION`) {
        $r{\to}default\_a = 1$;
        $r{\to}search = $ `FIND_B`;
     }
     if ($r{\to}search \equiv$ `FIND_B_WITH_NO_SCATTERING`) {
        $r{\to}default\_a = 0$;
        $r{\to}search = $ `FIND_B`;
     }
     ⟨ Exit with bad input data 46 ⟩
     ⟨ Find the optical properties 47 ⟩
     ⟨ Print basic sphere and MC effects 48 ⟩
     if ($r{\to}final\_distance \leq r{\to}tolerance$) $r{\to}found = $ `TRUE`;
  }

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 ⟩ ≡
    $r{\to}error = measure\_OK(m, *r)$;
    if ($r{\to}method.quad\_pts < 4$) $r{\to}error = $ `IAD_QUAD_PTS_NOT_VALID`;
    if ($r{\to}error \neq$ `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␣result␣after␣AD_iterations␣=␣%d\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*, `FALSE`, `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*, `FALSE`, `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*, `TRUE`, `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*, `TRUE`, `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\n");

  }

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);
      /∗ 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;
  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 $(m.m\_t > 0 \wedge m.m\_u > m.m\_t)$ **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 $(m.as\_r < 0 \vee m.as\_r \geq 0.2)$ **return** `IAD_AS_NOT_VALID`;
  if $(m.ad\_r < 0 \vee m.ad\_r \geq 0.2)$ **return** `IAD_AD_NOT_VALID`;
  if $(m.ae\_r < 0 \vee m.ae\_r \geq 0.2)$ **return** `IAD_AE_NOT_VALID`;
  if $(m.rw\_r < 0 \vee m.rw\_r > 1.0)$ **return** `IAD_RW_NOT_VALID`;
  if $(m.rd\_r < 0 \vee m.rd\_r > 1.0)$ **return** `IAD_RD_NOT_VALID`;
  if $(m.rstd\_r < 0 \vee m.rstd\_r > 1.0)$ **return** `IAD_RSTD_NOT_VALID`;
  if $(m.rstd\_t < 0 \vee m.rstd\_t > 1.0)$ **return** `IAD_TSTD_NOT_VALID`;
  if $(m.f\_r < 0 \vee m.f\_r > 1)$ **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 $(m.as\_t < 0 \vee m.as\_t \geq 0.2)$ **return** `IAD_AS_NOT_VALID`;
  if $(m.ad\_t < 0 \vee m.ad\_t \geq 0.2)$ **return** `IAD_AD_NOT_VALID`;
  if $(m.ae\_t < 0 \vee m.ae\_t \geq 0.2)$ **return** `IAD_AE_NOT_VALID`;
  if $(m.rw\_t < 0 \vee m.rw\_r > 1.0)$ **return** `IAD_RW_NOT_VALID`;
  if $(m.rd\_t < 0 \vee m.rd\_t > 1.0)$ **return** `IAD_RD_NOT_VALID`;
  if $(m.rstd\_t < 0 \vee m.rstd\_t > 1.0)$ **return** `IAD_TSTD_NOT_VALID`;
  if $(m.f\_t < 0 \vee m.f\_t > 1)$ **return** `IAD_F_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 $independent = m.num\_measures$;
    if $(Debug(\texttt{DEBUG\_SEARCH}))$ {
      $fprintf(stderr, \texttt{"SEARCH:}_⊔\texttt{starting}_⊔\texttt{with}_⊔\texttt{\%d}_⊔\texttt{measurement(s)\textbackslash n"}, m.num\_measures)$;
      $fprintf(stderr, \texttt{"SEARCH:}_⊔\texttt{m\_r}_⊔\texttt{=}_⊔\texttt{\%8.5f}_⊔\texttt{"}, m.m\_r)$;
      $fprintf(stderr, \texttt{"m\_t}_⊔\texttt{=}_⊔\texttt{\%8.5f}_⊔\texttt{"}, m.m\_t)$;
      $fprintf(stderr, \texttt{"m\_u}_⊔\texttt{=}_⊔\texttt{\%8.5f\textbackslash n"}, m.m\_u)$;
    }
    $Estimate\_RT(m, r, \&rt, \&tt, \&rd, \&rc, \&td, \&tc)$;
    if $(m.m\_u \equiv 0 \land independent \equiv 3)$ {
      if $(Debug(\texttt{DEBUG\_SEARCH}))$ $fprintf(stderr, \texttt{"SEARCH:}_⊔\texttt{no}_⊔\texttt{information}_⊔\texttt{in}_⊔\texttt{tc\textbackslash n"})$;
      $independent --$;
    }
    if $(rd \equiv 0 \land independent \geq 2)$ {
      if $(Debug(\texttt{DEBUG\_SEARCH}))$ $fprintf(stderr, \texttt{"SEARCH:}_⊔\texttt{no}_⊔\texttt{information}_⊔\texttt{in}_⊔\texttt{rd\textbackslash n"})$;
      $independent --$;
    }
    if $(td \equiv 0 \land independent \geq 2)$ {
      if $(Debug(\texttt{DEBUG\_SEARCH}))$ $fprintf(stderr, \texttt{"SEARCH:}_⊔\texttt{no}_⊔\texttt{information}_⊔\texttt{in}_⊔\texttt{td\textbackslash n"})$;
      $independent --$;
    }
    if $(independent \equiv 1 \lor independent \equiv -1)$ {
      ⟨ One parameter search 59 ⟩
    }
    else if $(independent \equiv 2)$ {
      ⟨ Two parameter search 60 ⟩
    }   /∗ three real parameters with information! ∗/
    else {
      $search = \texttt{FIND\_AG}$;
    }
    if $(search \equiv \texttt{FIND\_BG} \land m.m\_u > 0)$ $search = \texttt{FIND\_G}$;
    if $(Debug(\texttt{DEBUG\_SEARCH}))$ {
      $fprintf(stderr, \texttt{"SEARCH:}_⊔\texttt{ending}_⊔\texttt{with}_⊔\texttt{\%d}_⊔\texttt{measurement(s)\textbackslash n"}, m.num\_measures)$;
      $fprintf(stderr, \texttt{"SEARCH:}_⊔\texttt{final}_⊔\texttt{choice}_⊔\texttt{for}_⊔\texttt{search}_⊔\texttt{=}_⊔\texttt{"})$;
      if $(search \equiv \texttt{FIND\_A})$ $fprintf(stderr, \texttt{"FIND\_A\textbackslash n"})$;
      if $(search \equiv \texttt{FIND\_B})$ $fprintf(stderr, \texttt{"FIND\_B\textbackslash n"})$;
      if $(search \equiv \texttt{FIND\_AB})$ $fprintf(stderr, \texttt{"FIND\_AB\textbackslash n"})$;
      if $(search \equiv \texttt{FIND\_AG})$ $fprintf(stderr, \texttt{"FIND\_AG\textbackslash n"})$;
      if $(search \equiv \texttt{FIND\_AUTO})$ $fprintf(stderr, \texttt{"FIND\_AUTO\textbackslash n"})$;
      if $(search \equiv \texttt{FIND\_BG})$ $fprintf(stderr, \texttt{"FIND\_BG\textbackslash n"})$;
      if $(search \equiv FIND\_BaG)$ $fprintf(stderr, \texttt{"FIND\_BaG\textbackslash n"})$;
      if $(search \equiv FIND\_BsG)$ $fprintf(stderr, \texttt{"FIND\_BsG\textbackslash n"})$;
      if $(search \equiv FIND\_Ba)$ $fprintf(stderr, \texttt{"FIND\_Ba\textbackslash n"})$;
      if $(search \equiv FIND\_Bs)$ $fprintf(stderr, \texttt{"FIND\_Bs\textbackslash n"})$;
      if $(search \equiv \texttt{FIND\_G})$ $fprintf(stderr, \texttt{"FIND\_G\textbackslash n"})$;
      if $(search \equiv \texttt{FIND\_B\_WITH\_NO\_ABSORPTION})$ $fprintf(stderr, \texttt{"FIND\_B\_WITH\_NO\_ABSORPTION\textbackslash n"})$;
      if $(search \equiv \texttt{FIND\_B\_WITH\_NO\_SCATTERING})$ $fprintf(stderr, \texttt{"FIND\_B\_WITH\_NO\_SCATTERING\textbackslash n"})$;

```
    }
    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 (td ≡ 0)  search = FIND_A;
  else if (rd ≡ 0)  search = FIND_B_WITH_NO_SCATTERING;
  else  search = FIND_B_WITH_NO_ABSORPTION;
```
This code is used in section 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 $(r.default\_a \neq \texttt{UNINITIALIZED})$ {
    if $(r.default\_a \equiv 0)$ $search = \texttt{FIND\_B}$;
    else if $(r.default\_g \neq \texttt{UNINITIALIZED})$ $search = \texttt{FIND\_B}$;
    else $search = \texttt{FIND\_BG}$;
  }
  else if $(r.default\_b \neq \texttt{UNINITIALIZED})$ {
    if $(r.default\_g \neq \texttt{UNINITIALIZED})$ $search = \texttt{FIND\_A}$;
    else $search = \texttt{FIND\_AG}$;
  }
  else if $(r.default\_ba \neq \texttt{UNINITIALIZED})$ {
    if $(r.default\_g \neq \texttt{UNINITIALIZED})$ $search = FIND\_Bs$;
    else $search = FIND\_BsG$;
  }
  else if $(r.default\_bs \neq \texttt{UNINITIALIZED})$ {
    if $(r.default\_g \neq \texttt{UNINITIALIZED})$ $search = FIND\_Ba$;
    else $search = FIND\_BaG$;
  }
  else if $(rt + tt > 1 \wedge 0 \wedge m.num\_spheres \neq 2)$ $search = \texttt{FIND\_B\_WITH\_NO\_ABSORPTION}$;
  else $search = \texttt{FIND\_AB}$;
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 $Initialize\_Result(\textbf{struct measure\_type } m, \textbf{struct invert\_type } *r)$
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 ⟩ ≡
  $r{\rightarrow}a = 0.0$;
  $r{\rightarrow}b = 0.0$;
  $r{\rightarrow}g = 0.0$;
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 ⟩ +≡
  $r{\rightarrow}found$ = FALSE;
  $r{\rightarrow}tolerance$ = 0.0001;
  $r{\rightarrow}MC\_tolerance$ = 0.01;   /∗ percent ∗/
  $r{\rightarrow}search$ = FIND_AUTO;
  $r{\rightarrow}metric$ = RELATIVE;
  $r{\rightarrow}final\_distance$ = 10;
  $r{\rightarrow}AD\_iterations$ = 0;
  $r{\rightarrow}MC\_iterations$ = 0;
  $r{\rightarrow}error$ = IAD_NO_ERROR;

65.   The defaults might be handy

⟨ Fill r with reasonable values 63 ⟩ +≡
  $r{\rightarrow}default\_a$ = UNINITIALIZED;
  $r{\rightarrow}default\_b$ = UNINITIALIZED;
  $r{\rightarrow}default\_g$ = UNINITIALIZED;
  $r{\rightarrow}default\_ba$ = UNINITIALIZED;
  $r{\rightarrow}default\_bs$ = UNINITIALIZED;
  $r{\rightarrow}default\_mua$ = UNINITIALIZED;
  $r{\rightarrow}default\_mus$ = 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 ⟩ +≡
  $r{\rightarrow}slab.a$ = 0.5;
  $r{\rightarrow}slab.b$ = 1.0;
  $r{\rightarrow}slab.g$ = 0;
  $r{\rightarrow}slab.phase\_function$ = HENYEY_GREENSTEIN;
  $r{\rightarrow}slab.n\_slab$ = $m.slab\_index$;
  $r{\rightarrow}slab.n\_top\_slide$ = $m.slab\_top\_slide\_index$;
  $r{\rightarrow}slab.n\_bottom\_slide$ = $m.slab\_bottom\_slide\_index$;
  $r{\rightarrow}slab.b\_top\_slide$ = $m.slab\_top\_slide\_b$;
  $r{\rightarrow}slab.b\_bottom\_slide$ = $m.slab\_bottom\_slide\_b$;
  $r{\rightarrow}slab.cos\_angle$ = $m.slab\_cos\_angle$;
  $r{\rightarrow}method.a\_calc$ = 0.5;
  $r{\rightarrow}method.b\_calc$ = 1;
  $r{\rightarrow}method.g\_calc$ = 0.5;
  $r{\rightarrow}method.quad\_pts$ = 8;
  $r{\rightarrow}method.b\_thinnest$ = 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** Tc, **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 (Tc ≡ 0) $m.num\_measures$ −−;
    $m.m_r = $ UR1;
    $m.m_t = $ UT1;
    $m.m\_u = Tc$;
    Initialize_Result (m, &r);
    $r.method.quad\_pts = 8$;
    Inverse_RT (m, &r);
    $*error = r.error$;
    if ($r.error$ ≡ IAD_NO_ERROR) {
      $*a = r.a$;
      $*b = r.b$;
      $*g = r.g$;
    }
  }

This code is used in section 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*;

    m⃗slab_index = 1.0;
    m⃗slab_top_slide_index = 1.0;
    m⃗slab_top_slide_b = 0.0;
    m⃗slab_top_slide_thickness = 0.0;
    m⃗slab_bottom_slide_index = 1.0;
    m⃗slab_bottom_slide_b = 0.0;
    m⃗slab_bottom_slide_thickness = 0.0;
    m⃗slab_thickness = 1.0;
    m⃗slab_cos_angle = 1.0;
    m⃗num_spheres = 0;
    m⃗num_measures = 1;
    m⃗method = UNKNOWN;
    m⃗fraction_of_rc_in_mr = 1.0;
    m⃗fraction_of_tc_in_mt = 1.0;
    m⃗baffle_r = 1;
    m⃗baffle_t = 1;
    m⃗flip_sample = 0;
    m⃗gain_type = 0;
    m⃗m_r = 0.0;
    m⃗m_t = 0.0;
    m⃗m_u = 0.0;
    m⃗d_sphere_r = *default_sphere_d*;
    m⃗as_r = (M_PI ∗ *default_sample_d* ∗ *default_sample_d*/4.0)/*sphere_area*;
    m⃗ad_r = (M_PI ∗ *default_detector_d* ∗ *default_detector_d*/4.0)/*sphere_area*;
    m⃗ae_r = (M_PI ∗ *default_entrance_d* ∗ *default_entrance_d*/4.0)/*sphere_area*;
    m⃗aw_r = 1.0 − *m⃗as_r* − *m⃗ad_r* − *m⃗ae_r*;
    m⃗rd_r = 0.0;
    m⃗rw_r = 1.0;
    m⃗rstd_r = 1.0;
    m⃗f_r = 0.0;
    m⃗d_sphere_t = *default_sphere_d*;
    m⃗as_t = *m⃗as_r*;
    m⃗ad_t = *m⃗ad_r*;
    m⃗ae_t = 0;
    m⃗aw_t = 1.0 − *m⃗as_t* − *m⃗ad_t* − *m⃗ae_t*;
    m⃗rd_t = 0.0;
    m⃗rw_t = 1.0;
    m⃗rstd_t = 1.0;
    m⃗f_t = 0.0;
    m⃗lambda = 0.0;
    m⃗d_beam = 0.0;
    m⃗ur1_lost = 0;
    m⃗uru_lost = 0;
    m⃗ut1_lost = 0;

$m \rightarrow utu\_lost = 0;$

    }

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*);
        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, $d\_entrance\_t$, $d\_detector\_t$;

     $m.slab\_index = setup[0]$;
     $m.slab\_top\_slide\_index = setup[1]$;
     $m.slab\_thickness = setup[2]$;
     $m.slab\_top\_slide\_thickness = setup[3]$;
     $m.d\_beam = setup[4]$;
     $m.rstd\_r = setup[5]$;
     $m.num\_spheres = (\textbf{int})\,setup[6]$;
     $m.d\_sphere\_r = setup[7]$;
     $d\_sample\_r = setup[8]$;
     $d\_entrance\_r = setup[9]$;
     $d\_detector\_r = setup[10]$;
     $m.rw\_r = setup[11]$;
     $m.d\_sphere\_t = setup[12]$;
     $d\_sample\_t = setup[13]$;
     $d\_entrance\_t = setup[14]$;
     $d\_detector\_t = setup[15]$;
     $m.rw\_t = setup[16]$;
     $r.default\_g = setup[17]$;
     $num\_photons = (\textbf{long})\,setup[18]$;
     $m.as\_r = (d\_sample\_r/m.d\_sphere\_r/2.0) * (d\_sample\_r/m.d\_sphere\_r/2.0)$;
     $m.ae\_r = (d\_entrance\_r/m.d\_sphere\_r/2.0) * (d\_entrance\_r/m.d\_sphere\_r/2.0)$;
     $m.ad\_r = (d\_detector\_r/m.d\_sphere\_r/2.0) * (d\_detector\_r/m.d\_sphere\_r/2.0)$;
     $m.aw\_r = 1.0 - m.as\_r - m.ae\_r - m.ad\_r$;
     $m.as\_t = (d\_sample\_t/m.d\_sphere\_t/2.0) * (d\_sample\_t/m.d\_sphere\_t/2.0)$;
     $m.ae\_t = (d\_entrance\_t/m.d\_sphere\_t/2.0) * (d\_entrance\_t/m.d\_sphere\_t/2.0)$;
     $m.ad\_t = (d\_detector\_t/m.d\_sphere\_t/2.0) * (d\_detector\_t/m.d\_sphere\_t/2.0)$;
     $m.aw\_t = 1.0 - m.as\_t - m.ae\_t - m.ad\_t$;
     $m.slab\_bottom\_slide\_index = m.slab\_top\_slide\_index$;
     $m.slab\_bottom\_slide\_thickness = m.slab\_top\_slide\_thickness$;
     $fprintf(stderr, \texttt{"****}\_\texttt{executing}\_\texttt{FIXME}\_\texttt{****/n"})$;
     $m.slab\_cos\_angle = 1.0$;   /∗ FIXME ∗/
  }

This code is used in section 72.

74.   ⟨ handle analysis 74 ⟩ ≡
   $r.method.quad\_pts = (\textbf{int})\,analysis[0]$;
   $mc\_runs = (\textbf{int})\,analysis[1]$;

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 $(m.m\_t \equiv 0)$ $m.num\_measures$ −−;
  if $(m.m\_u \equiv 0)$ $m.num\_measures$ −−;
This code is used in section 72.

76.

⟨ handle reflection sphere 76 ⟩ ≡
  $m.as\_r = sphere\_r[0];$
  $m.ae\_r = sphere\_r[1];$
  $m.ad\_r = sphere\_r[2];$
  $m.rw\_r = sphere\_r[3];$
  $m.rd\_r = sphere\_r[4];$
  $m.rstd\_r = sphere\_r[5];$
  $m.f\_r = sphere\_r[7];$
This code is used in section 72.

77.

⟨ handle transmission sphere 77 ⟩ ≡
  $m.as\_t = sphere\_t[0];$
  $m.ae\_t = sphere\_t[1];$
  $m.ad\_t = sphere\_t[2];$
  $m.rw\_t = sphere\_t[3];$
  $m.rd\_t = sphere\_t[4];$
  $m.rstd\_t = sphere\_t[5];$
  $m.f\_t = sphere\_t[7];$
This code is used in section 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*) {
      m.ur1_lost = 0;
      m.ut1_lost = 0;
      m.uru_lost = 0;
      m.utu_lost = 0;
    }
    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*);
    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.

$\langle$ handle2 reflection sphere $88\rangle \equiv$
{
    double $d\_sample\_r, d\_entrance\_r, d\_detector\_r$;

    $m.d\_sphere\_r = sphere\_r[0]$;
    $d\_sample\_r = sphere\_r[1]$;
    $d\_entrance\_r = sphere\_r[2]$;
    $d\_detector\_r = sphere\_r[3]$;
    $m.rw\_r = sphere\_r[4]$;
    $m.rd\_r = sphere\_r[5]$;
    $m.as\_r = (d\_sample\_r/m.d\_sphere\_r/2.0) * (d\_sample\_r/m.d\_sphere\_r/2.0)$;
    $m.ae\_r = (d\_entrance\_r/m.d\_sphere\_r/2.0) * (d\_entrance\_r/m.d\_sphere\_r/2.0)$;
    $m.ad\_r = (d\_detector\_r/m.d\_sphere\_r/2.0) * (d\_detector\_r/m.d\_sphere\_r/2.0)$;
    $m.aw\_r = 1.0 - m.as\_r - m.ae\_r - m.ad\_r$;
}

This code is used in section 85.

89.

$\langle$ handle2 transmission sphere $89\rangle \equiv$
{
    double $d\_sample\_t, d\_entrance\_t, d\_detector\_t$;

    $m.d\_sphere\_t = sphere\_t[0]$;
    $d\_sample\_t = sphere\_t[1]$;
    $d\_entrance\_t = sphere\_t[2]$;
    $d\_detector\_t = sphere\_t[3]$;
    $m.rw\_t = sphere\_t[4]$;
    $m.rd\_t = sphere\_t[5]$;
    $m.as\_t = (d\_sample\_t/m.d\_sphere\_t/2.0) * (d\_sample\_t/m.d\_sphere\_t/2.0)$;
    $m.ae\_t = (d\_entrance\_t/m.d\_sphere\_t/2.0) * (d\_entrance\_t/m.d\_sphere\_t/2.0)$;
    $m.ad\_t = (d\_detector\_t/m.d\_sphere\_t/2.0) * (d\_detector\_t/m.d\_sphere\_t/2.0)$;
    $m.aw\_t = 1.0 - m.as\_t - m.ae\_t - m.ad\_t$;
}

This code is used in section 85.

90.

$\langle$ handle2 analysis $90\rangle \equiv$
  $r.method.quad\_pts = (\textbf{int}) \ analysis[0]$;
  $mc\_runs = (\textbf{int}) \ analysis[1]$;
  $num\_photons = (\textbf{long}) \ analysis[2]$;
This code is used in section 85.

91.

$\langle$ handle2 measurement $91\rangle \equiv$
  $m.rstd\_r = measurement[0]$;
  $m.m\_r = measurement[1]$;
  $m.m\_t = measurement[2]$;
  $m.m\_u = measurement[3]$;
  $m.num\_measures = 3$;
  if $(m.m\_t \equiv 0) \ m.num\_measures\ –$ –;
  if $(m.m\_u \equiv 0) \ m.num\_measures\ –$ –;
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
#**include** <string.h>
#**include** <stdio.h>
#**include** <ctype.h>
#**include** <math.h>
#**include** "ad_globl.h"
#**include** "iad_type.h"
#**include** "iad_io.h"
#**include** "iad_pub.h"
#**include** "version.h"
  ⟨ Definition for *skip_white* 102 ⟩
  ⟨ Definition for *read_number* 104 ⟩
  ⟨ Definition for *check_magic* 106 ⟩
  ⟨ Definition for *Read_Header* 96 ⟩
  ⟨ Definition for *Write_Header* 108 ⟩
  ⟨ Definition for *Read_Data_Line* 100 ⟩

93.   ⟨ iad_io.h   93 ⟩ ≡
  ⟨ Prototype for *Read_Header* 95 ⟩;
  ⟨ Prototype for *Write_Header* 107 ⟩;
  ⟨ Prototype for *Read_Data_Line* 99 ⟩;

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 ⟩
    if (*read_number* (*fp*, &*x*)) **return** 1;
    ∗*params* = (**int**) *x*;
    m→*num_measures* = (∗*params* ≥ 3) ? 3 : ∗*params*;
    return 0;
  }

This code is used in section 92.

97. ⟨ Read coefficients for reflection sphere 97 ⟩ ≡
  {
     double d_sample_r, d_empty_r, $d\_detector\_r$;

     if $(read\_number(\mathit{fp}, \&m \rightarrow d\_sphere\_r))$ **return** 1;
     if $(read\_number(\mathit{fp}, \&d\_sample\_r))$ **return** 1;
     if $(read\_number(\mathit{fp}, \&d\_empty\_r))$ **return** 1;
     if $(read\_number(\mathit{fp}, \&d\_detector\_r))$ **return** 1;
     if $(read\_number(\mathit{fp}, \&m \rightarrow rw\_r))$ **return** 1;
     $m \rightarrow as\_r = (d\_sample\_r / m \rightarrow d\_sphere\_r / 2.0) * (d\_sample\_r / m \rightarrow d\_sphere\_r / 2.0)$;
     $m \rightarrow ae\_r = (d\_empty\_r / m \rightarrow d\_sphere\_r / 2.0) * (d\_empty\_r / m \rightarrow d\_sphere\_r / 2.0)$;
     $m \rightarrow ad\_r = (d\_detector\_r / m \rightarrow d\_sphere\_r / 2.0) * (d\_detector\_r / m \rightarrow d\_sphere\_r / 2.0)$;
     $m \rightarrow aw\_r = 1.0 - m \rightarrow as\_r - m \rightarrow ae\_r - m \rightarrow ad\_r$;
  }

This code is used in section 96.

98. ⟨ Read coefficients for transmission sphere 98 ⟩ ≡
  {
     double d_sample_t, d_empty_t, $d\_detector\_t$;

     if $(read\_number(\mathit{fp}, \&m \rightarrow d\_sphere\_t))$ **return** 1;
     if $(read\_number(\mathit{fp}, \&d\_sample\_t))$ **return** 1;
     if $(read\_number(\mathit{fp}, \&d\_empty\_t))$ **return** 1;
     if $(read\_number(\mathit{fp}, \&d\_detector\_t))$ **return** 1;
     if $(read\_number(\mathit{fp}, \&m \rightarrow rw\_t))$ **return** 1;
     $m \rightarrow as\_t = (d\_sample\_t / m \rightarrow d\_sphere\_t / 2.0) * (d\_sample\_t / m \rightarrow d\_sphere\_t / 2.0)$;
     $m \rightarrow ae\_t = (d\_empty\_t / m \rightarrow d\_sphere\_t / 2.0) * (d\_empty\_t / m \rightarrow d\_sphere\_t / 2.0)$;
     $m \rightarrow ad\_t = (d\_detector\_t / m \rightarrow d\_sphere\_t / 2.0) * (d\_detector\_t / m \rightarrow d\_sphere\_t / 2.0)$;
     $m \rightarrow aw\_t = 1.0 - m \rightarrow as\_t - m \rightarrow ae\_t - m \rightarrow ad\_t$;
  }

This code is used in section 96.

99.   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.

$\langle$ Prototype for *Read_Data_Line* 99 $\rangle \equiv$

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

This code is used in sections 93 and 100.

100. $\langle$ Definition for *Read_Data_Line* 100 $\rangle \equiv$

  $\langle$ Prototype for *Read_Data_Line* 99 $\rangle$

  {

    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) {

      m→m_t = *m→m_r*;

      m→m_r = 0;

      return 0;

    }

    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.

101.   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.

$\langle$ Prototype for *skip_white* 101 $\rangle \equiv$

   int *skip_white*(**FILE** *∗fp*)

This code is used in section 102.

102.   ⟨ Definition for *skip_white* 102 ⟩ ≡
  ⟨ Prototype for *skip_white* 101 ⟩
  {
    int $c = fgetc(fp)$;
    while $(\neg feof(fp))$ {
      if $(isspace(c))$ $c = fgetc(fp)$;
      else if $(c \equiv \text{'#'})$ **do** $c = fgetc(fp)$; **while** $(\neg feof(fp) \wedge c \neq \text{'\textbackslash n'} \wedge c \neq \text{'\textbackslash r'})$;
      else break;
    }
    if $(feof(fp))$ **return** 1;
    $ungetc(c, fp)$;
    return 0;
  }
This code is used in section 92.

103.   Read a single number. Return 0 if there are no problems, otherwise return 1.
⟨ Prototype for *read_number* 103 ⟩ ≡
  int *read_number*(**FILE** $*fp$, **double** $*x$)
This code is used in section 104.

104.   ⟨ Definition for *read_number* 104 ⟩ ≡
  ⟨ Prototype for *read_number* 103 ⟩
  {
    if $(skip\_white(fp))$ **return** 1;
    if $(fscanf(fp, \text{"%lf"}, x))$ **return** 0;
    else return 1;
  }
This code is used in section 92.

105.   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* 105 ⟩ ≡
  int *check_magic*(**FILE** $*fp$)
This code is used in section 106.

106.   ⟨ Definition for *check_magic* 106 ⟩ ≡
  ⟨ Prototype for *check_magic* 105 ⟩
  {
    char $magic[\,] = \text{"IAD1"}$;
    int i, c;
    for $(i = 0; \ i < 4; \ i\mathrel{++})$ {
      $c = fgetc(fp)$;
      if $(feof(fp) \vee c \neq magic[i])$ {
        $fprintf(stderr, \text{"Sorry,}_\sqcup\text{but}_\sqcup\text{iad}_\sqcup\text{input}_\sqcup\text{files}_\sqcup\text{must}_\sqcup\text{begin}_\sqcup\text{with}_\sqcup\text{IAD1\textbackslash n"})$;
        $fprintf(stderr, \text{"}_{\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup}\text{as}_\sqcup\text{the}_\sqcup\text{first}_\sqcup\text{four}_\sqcup\text{characters}_\sqcup\text{of}_\sqcup\text{the}_\sqcup\text{file.\textbackslash n"})$;
        $fprintf(stderr, \text{"}_{\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup}\text{Perhaps}_\sqcup\text{you}_\sqcup\text{are}_\sqcup\text{using}_\sqcup\text{an}_\sqcup\text{old}_\sqcup\text{iad}_\sqcup\text{format?\textbackslash n"})$;
        return 1;
      }
    }
    return 0;
  }
This code is used in section 92.

107. Formatting the header information.

⟨ Prototype for *Write_Header* 107 ⟩ ≡
   void *Write_Header*(**struct measure_type** *m*, **struct invert_type** *r*, **int** *params*)

This code is used in sections 93 and 108.

108. ⟨ Definition for *Write_Header* 108 ⟩ ≡
   ⟨ Prototype for *Write_Header* 107 ⟩
   {
      ⟨ Write slab info 109 ⟩
      ⟨ Write irradiation info 110 ⟩
      ⟨ Write general sphere info 111 ⟩
      ⟨ Write first sphere info 112 ⟩
      ⟨ Write second sphere info 113 ⟩
      ⟨ Write measure and inversion info 114 ⟩
   }

This code is used in section 92.

109. ⟨ Write slab info 109 ⟩ ≡
   double *xx*;

   printf("#␣Inverse␣Adding−Doubling␣%s␣\n", *Version*);
   printf("#␣\n");
   printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Beam␣diameter␣=␣%7.1f␣mm\n", *m.d_beam*);
   printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Sample␣thickness␣=␣%7.3f␣mm\n", *m.slab_thickness*);
   printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Top␣slide␣thickness␣=␣%7.3f␣mm\n", *m.slab_top_slide_thickness*);
   printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Bottom␣slide␣thickness␣=␣%7.3f␣mm\n", *m.slab_bottom_slide_thickness*);
   printf("#␣␣␣␣␣␣␣␣␣␣␣␣␣Sample␣index␣of␣refraction␣=␣%7.4f\n", *m.slab_index*);
   printf("#␣␣␣␣␣␣␣␣␣Top␣slide␣index␣of␣refraction␣=␣%7.4f\n", *m.slab_top_slide_index*);
   printf("#␣␣␣␣␣Bottom␣slide␣index␣of␣refraction␣=␣%7.4f\n", *m.slab_bottom_slide_index*);

This code is used in section 108.

110. ⟨ Write irradiation info 110 ⟩ ≡
   printf("#␣\n");

This code is used in section 108.

111. ⟨ Write general sphere info 111 ⟩ ≡
   printf("#␣␣Percentage␣unscattered␣refl.␣in␣M_R␣=␣%7.1f␣%%\n", *m.fraction_of_rc_in_mr* * 100);
   printf("#␣Percentage␣unscattered␣trans.␣in␣M_T␣=␣%7.1f␣%%\n", *m.fraction_of_tc_in_mt* * 100);
   printf("#␣\n");

This code is used in section 108.

112.   ⟨ Write first sphere info 112 ⟩ ≡

  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* ("␣(all␣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 ("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣empty␣port␣diameter␣=␣%7.1f␣mm\n", 2 * *m.d_sphere_r* * *sqrt*(*m.ae_r*));

  printf ("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣detector␣port␣diameter␣=␣%7.1f␣mm\n", 2 * *m.d_sphere_r* * *sqrt*(*m.ad_r*));

  printf ("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣detector␣reflectance␣=␣%7.1f␣%%\n", *m.rd_r* * 100);

  printf ("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣wall␣reflectance␣=␣%7.1f␣%%\n", *m.rw_r* * 100);

  printf ("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣calibration␣standard␣=␣%7.1f␣%%\n", *m.rstd_r* * 100);

  printf ("#\n");

This code is used in section 108.

113.   ⟨ Write second sphere info 113 ⟩ ≡

  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* ("␣(all␣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 ("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣empty␣port␣diameter␣=␣%7.1f␣mm\n", 2 * *m.d_sphere_r* * *sqrt*(*m.ae_t*));

  printf ("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣detector␣port␣diameter␣=␣%7.1f␣mm\n", 2 * *m.d_sphere_r* * *sqrt*(*m.ad_t*));

  printf ("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣detector␣reflectance␣=␣%7.1f␣%%\n", *m.rd_t* * 100);

  printf ("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣wall␣reflectance␣=␣%7.1f␣%%", *m.rw_t* * 100);

  if (*m.ae_t* ≡ 0) *printf* ("␣(cal␣std␣since␣empty␣port␣is␣closed)");

  printf ("\n");

  printf ("#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣calibration␣standard␣=␣%7.1f␣%%", *m.rstd_t* * 100);

  if (*m.ae_t* ≡ 0) *printf* ("␣(ignored)");

  printf ("\n");

This code is used in section 108.

114. ⟨Write measure and inversion info 114⟩ ≡
  printf ("#\n");
  switch (*params*) {
  case −1: *printf* ("#␣No␣M_R␣or␣M_T␣--␣forward␣calculation.\n");
    break;
  case 1: *printf* ("#␣Just␣M_R␣was␣measured");
    break;
  case 2: *printf* ("#␣M_R␣and␣M_T␣were␣measured");
    break;
  case 3: *printf* ("#␣M_R,␣M_T,␣and␣M_U␣were␣measured");
    break;
  case 4: *printf* ("#␣M_R,␣M_T,␣M_U,␣and␣r_w␣were␣measured");
    break;
  case 5: *printf* ("#␣M_R,␣M_T,␣M_U,␣r_w,␣and␣t_w␣were␣measured");
    break;
  case 6: *printf* ("#␣M_R,␣M_T,␣M_U,␣r_w,␣t_w,␣and␣r_std␣were␣measured");
    break;
  case 7: *printf* ("#␣M_R,␣M_T,␣M_U,␣r_w,␣t_w,␣r_std␣and␣t_std␣were␣measured");
    break;
  default: *printf* ("#␣Something␣went␣wrong␣...␣measures␣should␣be␣1␣to␣5!\n");
    break;
  }
  if (1 ≤ *params* ∧ *params* ≤ 7) {
    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 108.

115. IAD Calculation.

$\langle$ `iad_calc.c`   115 $\rangle \equiv$

#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` $1 \cdot 10^{-6}$

#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` 101

#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 $** \, The\_Grid = \Lambda$;

  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 $The\_Grid\_Search = -1$;

  $\langle$ Definition for Set_Calc_State 131 $\rangle$

  $\langle$ Definition for Get_Calc_State 133 $\rangle$

  $\langle$ Definition for $Same\_Calc\_State$ 135 $\rangle$

  $\langle$ Prototype for $Fill\_AB\_Grid$ 153 $\rangle$;

  $\langle$ Prototype for $Fill\_AG\_Grid$ 157 $\rangle$;

⟨ Definition for *RT_Flip* 151 ⟩
⟨ Definition for *Allocate_Grid* 137 ⟩
⟨ Definition for *Valid_Grid* 141 ⟩
⟨ Definition for *fill_grid_entry* 152 ⟩
⟨ Definition for *Fill_Grid* 167 ⟩
⟨ Definition for *Near_Grid_Points* 149 ⟩
⟨ Definition for *Fill_AB_Grid* 154 ⟩
⟨ Definition for *Fill_AG_Grid* 158 ⟩
⟨ Definition for *Fill_BG_Grid* 161 ⟩
⟨ Definition for *Fill_BaG_Grid* 163 ⟩
⟨ Definition for *Fill_BsG_Grid* 165 ⟩
⟨ Definition for *Grid_ABG* 139 ⟩
⟨ Definition for *Gain* 120 ⟩
⟨ Definition for *Gain_11* 122 ⟩
⟨ Definition for *Gain_22* 124 ⟩
⟨ Definition for *Two_Sphere_R* 126 ⟩
⟨ Definition for *Two_Sphere_T* 128 ⟩
⟨ Definition for *Calculate_Distance_With_Corrections* 173 ⟩
⟨ Definition for *Calculate_Grid_Distance* 171 ⟩
⟨ Definition for *Calculate_Distance* 169 ⟩
⟨ Definition for *abg_distance* 147 ⟩
⟨ Definition for *Find_AG_fn* 185 ⟩
⟨ Definition for *Find_AB_fn* 187 ⟩
⟨ Definition for *Find_Ba_fn* 189 ⟩
⟨ Definition for *Find_Bs_fn* 191 ⟩
⟨ Definition for *Find_A_fn* 193 ⟩
⟨ Definition for *Find_B_fn* 195 ⟩
⟨ Definition for *Find_G_fn* 197 ⟩
⟨ Definition for *Find_BG_fn* 199 ⟩
⟨ Definition for *Find_BaG_fn* 201 ⟩
⟨ Definition for *Find_BsG_fn* 203 ⟩
⟨ Definition for *maxloss* 205 ⟩
⟨ Definition for *Max_Light_Loss* 207 ⟩

116.

⟨ `iad_calc.h` 116 ⟩ ≡
  ⟨ Prototype for *Gain* 119 ⟩;
  ⟨ Prototype for *Gain_11* 121 ⟩;
  ⟨ Prototype for *Gain_22* 123 ⟩;
  ⟨ Prototype for *Two_Sphere_R* 125 ⟩;
  ⟨ Prototype for *Two_Sphere_T* 127 ⟩;
  ⟨ Prototype for *Set_Calc_State* 130 ⟩;
  ⟨ Prototype for *Get_Calc_State* 132 ⟩;
  ⟨ Prototype for *Same_Calc_State* 134 ⟩;
  ⟨ Prototype for *Valid_Grid* 140 ⟩;
  ⟨ Prototype for *Allocate_Grid* 136 ⟩;
  ⟨ Prototype for *Fill_Grid* 166 ⟩;
  ⟨ Prototype for *Near_Grid_Points* 148 ⟩;
  ⟨ Prototype for *Grid_ABG* 138 ⟩;
  ⟨ Prototype for *Find_AG_fn* 184 ⟩;
  ⟨ Prototype for *Find_AB_fn* 186 ⟩;
  ⟨ Prototype for *Find_Ba_fn* 188 ⟩;
  ⟨ Prototype for *Find_Bs_fn* 190 ⟩;
  ⟨ Prototype for *Find_A_fn* 192 ⟩;
  ⟨ Prototype for *Find_B_fn* 194 ⟩;
  ⟨ Prototype for *Find_G_fn* 196 ⟩;
  ⟨ Prototype for *Find_BG_fn* 198 ⟩;
  ⟨ Prototype for *Find_BsG_fn* 202 ⟩;
  ⟨ Prototype for *Find_BaG_fn* 200 ⟩;
  ⟨ Prototype for *Fill_BG_Grid* 160 ⟩;
  ⟨ Prototype for *Fill_BsG_Grid* 164 ⟩;
  ⟨ Prototype for *Fill_BaG_Grid* 162 ⟩;
  ⟨ Prototype for *Calculate_Distance_With_Corrections* 172 ⟩;
  ⟨ Prototype for *Calculate_Distance* 168 ⟩;
  ⟨ Prototype for *Calculate_Grid_Distance* 170 ⟩;
  ⟨ Prototype for *abg_distance* 146 ⟩;
  ⟨ Prototype for *maxloss* 204 ⟩;
  ⟨ Prototype for *Max_Light_Loss* 206 ⟩;
  ⟨ Prototype for *RT_Flip* 150 ⟩;

117. 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.

118.   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

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

or with a baffle as

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

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 $r_w = 0$.

119.   ⟨ Prototype for $Gain$ 119 ⟩ ≡
    double $Gain$(**int** $sphere$, **struct measure_type** m, **double** URU)
This code is used in sections 116 and 120.

120.   ⟨ Definition for $Gain$ 120 ⟩ ≡
  ⟨ Prototype for $Gain$ 119 ⟩
  {
     double G, $denom$;
     if ($sphere \equiv$ REFLECTION_SPHERE) {
        if ($m.baffle\_r$) $denom = 1.0 - m.rw_r * (m.aw_r + (1 - m.ae_r) * (m.ad_r * m.rd_r + m.as_r * URU));$
        else $denom = 1.0 - m.aw_r * m.rw_r - m.ad_r * m.rd_r - m.as_r * URU;$
     }
     else {
        if ($m.baffle\_t$) $denom = 1.0 - m.rw_t * (m.aw_t + (1 - m.ae_t) * (m.ad_t * m.rd_t + m.as_t * URU));$
        else $denom = 1.0 - m.aw_t * m.rw_t - m.ad_t * m.rd_t - m.as_t * URU;$
     }
     $G = 1.0/denom;$
     return G;
  }
This code is used in section 115.

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

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

then the full expression for the gain is

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

⟨ Prototype for $Gain\_11$ 121 ⟩ ≡
    double $Gain\_11$(**struct measure_type** m, **double** URU, **double** $tdiffuse$)
This code is used in sections 116 and 122.

122.   ⟨Definition for *Gain_11* 122⟩ ≡
  ⟨Prototype for *Gain_11* 121⟩
  {
    double G, GP, G11;
    $G = Gain(\texttt{REFLECTION\_SPHERE}, m, \texttt{URU})$;
    $\texttt{GP} = Gain(\texttt{TRANSMISSION\_SPHERE}, m, \texttt{URU})$;
    $\texttt{G11} = G/(1 - m.as\_r * m.as\_t * m.aw\_r * m.aw\_t * (1 - m.ae\_r) * (1 - m.ae\_t) * G * \texttt{GP} * \textit{tdiffuse} * \textit{tdiffuse})$;
    return G11;
  }

This code is used in section 115.

123.   Similarly, when the light starts in the second sphere, the gain for light on the detector in the second sphere $G_{2\rightarrow2}$ is found by switching all primed variables to unprimed. Thus $G_{2\rightarrow1}(r_s, t_s)$ is

$$G_{2\rightarrow2}(r_s, t_s) = \frac{G'(r_s)}{1 - a_s a'_s r_w r'_w (1 - a_e)(1 - a'_e) G(r_s) G'(r_s) t_s^2}$$

⟨Prototype for *Gain_22* 123⟩ ≡
  double *Gain_22*(**struct measure_type** m, **double** URU, **double** *tdiffuse*)

This code is used in sections 116 and 124.

124.   ⟨Definition for *Gain_22* 124⟩ ≡
  ⟨Prototype for *Gain_22* 123⟩
  {
    double G, GP, G22;
    $G = Gain(\texttt{REFLECTION\_SPHERE}, m, \texttt{URU})$;
    $\texttt{GP} = Gain(\texttt{TRANSMISSION\_SPHERE}, m, \texttt{URU})$;
    $\texttt{G22} = \texttt{GP}/(1 - m.as\_r * m.as\_t * m.aw\_r * m.aw\_t * (1 - m.ae\_r) * (1 - m.ae\_t) * G * \texttt{GP} * \textit{tdiffuse} * \textit{tdiffuse})$;
    return G22;
  }

This code is used in section 115.

125.   The reflected power for two spheres makes use of the formulas for *Gain_11* above.
    The light on the detector in the reflection (first) sphere arises from three sources: the fraction of light directly reflected off the sphere wall $f r_w^2 (1 - a_e) P$, the fraction of light reflected by the sample $(1 - f) r_s^{\text{direct}} r_w^2 (1 - a_e) P$, and the light transmitted through the sample $(1 - f) t_s^{\text{direct}} r'_w (1 - a'_e) P$,

$$\begin{aligned} R(r_s^{\text{direct}}, r_s, t_s^{\text{direct}}, t_s) &= G_{1\rightarrow1}(r_s, t_s) \cdot a_d(1 - a_e) r_w^2 f P \\ &\quad + G_{1\rightarrow1}(r_s, t_s) \cdot a_d(1 - a_e) r_w (1 - f) r_s^{\text{direct}} P \\ &\quad + G_{2\rightarrow1}(r_s, t_s) \cdot a_d(1 - a'_e) r'_w (1 - f) t_s^{\text{direct}} P \end{aligned}$$

which simplifies slightly to

$$\begin{aligned} R(r_s^{\text{direct}}, r_s, t_s^{\text{direct}}, t_s) &= a_d(1 - a_e) r_w P \cdot G_{1\rightarrow1}(r_s, t_s) \\ &\quad \times \left[(1 - f) r_s^{\text{direct}} + f r_w + (1 - f) a'_s (1 - a'_e) r'_w t_s^{\text{direct}} t_s G'(r_s) \right] \end{aligned}$$

⟨Prototype for *Two_Sphere_R* 125⟩ ≡
  double *Two_Sphere_R*(**struct measure_type** m, **double** UR1, **double** URU, **double** UT1, **double** UTU)

This code is used in sections 116 and 126.

126.   ⟨ Definition for *Two_Sphere_R* 126 ⟩ ≡
  ⟨ Prototype for *Two_Sphere_R* 125 ⟩
  {
    double x, GP;

    GP = $Gain$(TRANSMISSION_SPHERE, m, URU);
    $x = m.ad\_r * (1 - m.ae\_r) * m.rw\_r * Gain\_11\,(m, \text{URU}, \text{UTU})$;
    $x *= (1 - m.f\_r) * \text{UR1} + m.rw\_r * m.f\_r + (1 - m.f\_r) * m.as\_t * (1 - m.ae\_t) * m.rw\_t * \text{UT1} * \text{UTU} * \text{GP}$;
    return x;
  }

This code is used in section 115.

127.   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

$$T(r_s^{\text{direct}}, r_s, t_s^{\text{direct}}, t_s) = G_{1\to2}(r_s, t_s) \cdot a'_d(1 - a_e)r_w^2 fP$$
$$+ G_{1\to2}(r_s, t_s) \cdot a'_d(1 - a_e)r_w(1 - f)r_s^{\text{direct}}P$$
$$+ G_{2\to2}(r_s, t_s) \cdot a'_d(1 - a'_e)r'_w(1 - f)t_s^{\text{direct}}P$$

or

$$T(r_s^{\text{direct}}, r_s, t_s^{\text{direct}}, t_s) = a'_d(1 - a'_e)r'_w P \cdot G_{2\to2}(r_s, t_s)$$
$$\times \left[(1 - f)t_s^{\text{direct}} + (1 - a_e)r_w a_s t_s(fr_w + (1 - f)r_s^{\text{direct}})G(r_s)\right]$$

⟨ Prototype for *Two_Sphere_T* 127 ⟩ ≡
  double *Two_Sphere_T*(**struct measure_type** m, **double** UR1, **double** URU, **double** UT1, **double** UTU)
This code is used in sections 116 and 128.

128.   ⟨ Definition for *Two_Sphere_T* 128 ⟩ ≡
  ⟨ Prototype for *Two_Sphere_T* 127 ⟩
  {
    double x, G;

    $G = Gain$(REFLECTION_SPHERE, m, URU);
    $x = m.ad\_t * (1 - m.ae\_t) * m.rw\_t * Gain\_22\,(m, \text{URU}, \text{UTU})$;
    $x *= (1 - m.f\_r) * \text{UT1} + (1 - m.ae\_r) * m.rw\_r * m.as\_r * \text{UTU} * (m.f\_r * m.rw\_r + (1 - m.f\_r) * \text{UR1}) * G$;
    return x;
  }

This code is used in section 115.

129.   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.

130.   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* 130 ⟩ ≡
  void *Set_Calc_State*(**struct measure_type** m, **struct invert_type** r)

This code is used in sections 116 and 131.

131.   ⟨ Definition for *Set_Calc_State* 131 ⟩ ≡
  ⟨ Prototype for *Set_Calc_State* 130 ⟩
  {
    memcpy(&MM, &m, **sizeof**(**struct measure_type**));
    memcpy(&RR, &r, **sizeof**(**struct invert_type**));
  }

This code is used in section 115.

132.   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* 132 ⟩ ≡
  void *Get_Calc_State*(**struct measure_type** $*m$, **struct invert_type** $*r$)

This code is used in sections 116 and 133.

133.   ⟨ Definition for *Get_Calc_State* 133 ⟩ ≡
  ⟨ Prototype for *Get_Calc_State* 132 ⟩
  {
    memcpy(m, &MM, **sizeof**(**struct measure_type**));
    memcpy(r, &RR, **sizeof**(**struct invert_type**));
  }

This code is used in section 115.

134.   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* 134 ⟩ ≡
  boolean_type *Same_Calc_State*(**struct measure_type** m, **struct invert_type** r)

This code is used in sections 116 and 135.

135. ⟨Definition for *Same_Calc_State* 135⟩ ≡
  ⟨Prototype for *Same_Calc_State* 134⟩
  {
    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 115.

136. ⟨Prototype for *Allocate_Grid* 136⟩ ≡
  void *Allocate_Grid*(**search_type** *s*)
This code is used in sections 116 and 137.

137. ⟨Definition for *Allocate_Grid* 137⟩ ≡
  ⟨Prototype for *Allocate_Grid* 136⟩
  {
    (**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 115.

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

⟨Prototype for *Grid_ABG* 138⟩ ≡
  void *Grid_ABG*(**int** *i*, **int** *j*, **guess_type** *∗guess*)
This code is used in sections 116 and 139.

139. ⟨ Definition for $Grid\_ABG$ 139 ⟩ ≡
  ⟨ Prototype for $Grid\_ABG$ 138 ⟩
  {
    if $(0 \leq i \wedge i < \texttt{GRID\_SIZE} \wedge 0 \leq j \wedge j < \texttt{GRID\_SIZE})$ {
      $guess\rightarrow a = The\_Grid[\texttt{GRID\_SIZE} * i + j][\texttt{A\_COLUMN}];$
      $guess\rightarrow b = The\_Grid[\texttt{GRID\_SIZE} * i + j][\texttt{B\_COLUMN}];$
      $guess\rightarrow g = The\_Grid[\texttt{GRID\_SIZE} * i + j][\texttt{G\_COLUMN}];$
      $guess\rightarrow distance = Calculate\_Grid\_Distance(i, j);$
    }
    else {
      $guess\rightarrow a = 0.5;$
      $guess\rightarrow b = 0.5;$
      $guess\rightarrow g = 0.5;$
      $guess\rightarrow distance = 999;$
    }
  }

This code is used in section 115.

140. 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 $Valid\_Grid$ 140 ⟩ ≡
  boolean_type $Valid\_Grid($**struct measure_type** $m,$ **search_type** $s)$

This code is used in sections 116 and 141.

141. ⟨ Definition for $Valid\_Grid$ 141 ⟩ ≡
  ⟨ Prototype for $Valid\_Grid$ 140 ⟩
  {
    ⟨ Tests for invalid grid 142 ⟩
    return (TRUE);
  }

This code is used in section 115.

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

⟨ Tests for invalid grid 142 ⟩ ≡
  if $(The\_Grid \equiv \Lambda)$ {
    if $(Debug(\texttt{DEBUG\_GRID}))$ $fprintf(stderr,$ "GRID:␣Fill␣because␣NULL\n"$);$
    return (FALSE);
  }
  if $(\neg The\_Grid\_Initialized)$ {
    if $(Debug(\texttt{DEBUG\_GRID}))$ $fprintf(stderr,$ "GRID:␣Fill␣because␣not␣initialized\n"$);$
    return (FALSE);
  }

See also sections 143, 144, and 145.

This code is used in section 141.

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

⟨ Tests for invalid grid 142 ⟩ +≡
  if (*The_Grid_Search* ≠ *s*) {
    if (*Debug*(DEBUG_GRID)) *fprintf*(*stderr*, "GRID:␣Fill␣because␣search␣type␣changed\n");
    return (FALSE);
  }

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

⟨ Tests for invalid grid 142 ⟩ +≡
  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);
  }

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

⟨ Tests for invalid grid 142 ⟩ +≡
  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);
  }

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

⟨ Prototype for *abg_distance* 146 ⟩ ≡
  void *abg_distance*(**double** *a*, **double** *b*, **double** *g*, **guess_type** *∗guess*)

This code is used in sections 116 and 147.

147.   ⟨ Definition for *abg_distance* 147 ⟩ ≡
  ⟨ Prototype for *abg_distance* 146 ⟩
  {
      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 115.

148.   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* 148 ⟩ ≡
    void *Near_Grid_Points*(**double** *r*, **double** *t*, **search_type** *s*, **int** *∗i_min*, **int** *∗j_min*)

This code is used in sections 116 and 149.

149. ⟨ Definition for *Near_Grid_Points* 149 ⟩ ≡
  ⟨ Prototype for *Near_Grid_Points* 148 ⟩
  {
    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 115.

150. 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* 150 ⟩ ≡
  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 116 and 151.

151.   ⟨ Definition for *RT_Flip* 151 ⟩ ≡
  ⟨ Prototype for *RT_Flip* 150 ⟩
  {
    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 115.

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

⟨ Definition for *fill_grid_entry* 152 ⟩ ≡
  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 115.

153.   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* 153 ⟩ ≡
  void *Fill_AB_Grid*(**struct measure_type** *m*, **struct invert_type** *r*)

This code is used in sections 115 and 154.

154.   ⟨ Definition for *Fill_AB_Grid* 154 ⟩ ≡
  ⟨ Prototype for *Fill_AB_Grid* 153 ⟩
  {
    int *i, j*;
    double *a*;
    double *min_b* = −8;   /∗ exp(-10) is smallest thickness ∗/
    double *max_b* = +8;   /∗ exp(+8) is greatest thickness ∗/
    if (*Debug*(DEBUG_GRID)) *fprintf* (*stderr*, "GRID:␣Filling␣AB␣grid\n");
    if (*The_Grid* ≡ Λ) *Allocate_Grid* (*r.search*);
    ⟨ Zero *GG* 159 ⟩
    Set_Calc_State (*m, r*);
    GG_g = RR.*slab.g*;
    for (*i* = 0; *i* < GRID_SIZE; *i*++) {
      double *x* = (**double**) *i*/(GRID_SIZE − 1.0);

      RR.*slab.b* = *exp* (*min_b* + (*max_b* − *min_b*) ∗ *x*);
      for (*j* = 0; *j* < GRID_SIZE; *j*++) {
        ⟨ Generate next albedo using j 156 ⟩
        fill_grid_entry (*i, j*);
      }
    }
    The_Grid_Initialized = TRUE;
    The_Grid_Search = FIND_AB;
  }
This code is used in section 115.

155.   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 155 ⟩ ≡
  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;
  }

156.   Here is heuristic that seems to work well

$\langle$ Generate next albedo using j 156 $\rangle \equiv$
   $a = (\textbf{double}) \, j/(\texttt{GRID\_SIZE} - 1.0);$
   $\texttt{RR}.slab.a = (1.0 - a * a) * (1.0 - a) + (1.0 - a) * (1.0 - a) * a;$
This code is used in sections 154 and 158.

157.   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.

$\langle$ Prototype for *Fill_AG_Grid* 157 $\rangle \equiv$
   $\textbf{void } Fill\_AG\_Grid(\textbf{struct measure\_type } m, \textbf{struct invert\_type } r)$
This code is used in sections 115 and 158.

158. $\langle$ Definition for *Fill_AG_Grid* 158 $\rangle \equiv$
   $\langle$ Prototype for *Fill_AG_Grid* 157 $\rangle$
   {
      $\textbf{int } i, j;$
      $\textbf{double } a;$
      $\textbf{if } (Debug(\texttt{DEBUG\_GRID})) \; fprintf(stderr, \texttt{"GRID:}_\sqcup\texttt{Filling}_\sqcup\texttt{AG}_\sqcup\texttt{grid}\backslash\texttt{n"});$
      $\textbf{if } (The\_Grid \equiv \Lambda) \; Allocate\_Grid(r.search);$
      $\langle$ Zero GG 159 $\rangle$
      $Set\_Calc\_State(m, r);$
      $GG\_b = r.slab.b;$
      $\textbf{for } (i = 0; \; i < \texttt{GRID\_SIZE}; \; i\texttt{++}) \; \{$
         $\texttt{RR}.slab.g = \texttt{MAX\_ABS\_G} * (2.0 * i/(\texttt{GRID\_SIZE} - 1.0) - 1.0);$
         $\textbf{for } (j = 0; \; j < \texttt{GRID\_SIZE}; \; j\texttt{++}) \; \{$
            $\langle$ Generate next albedo using j 156 $\rangle$
            $fill\_grid\_entry(i, j);$
         }
      }
      $The\_Grid\_Initialized = \texttt{TRUE};$
      $The\_Grid\_Search = \texttt{FIND\_AG};$
   }
This code is used in section 115.

159.

$\langle$ Zero GG 159 $\rangle \equiv$
   $GG\_a = 0.0;$
   $GG\_b = 0.0;$
   $GG\_g = 0.0;$
   $GG\_bs = 0.0;$
   $GG\_ba = 0.0;$
This code is used in sections 154, 158, 161, 163, and 165.

160.   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* 160 ⟩ ≡
  void *Fill_BG_Grid*(**struct measure_type** m, **struct invert_type** r)
This code is used in sections 116 and 161.

161.   ⟨ Definition for *Fill_BG_Grid* 161 ⟩ ≡
  ⟨ Prototype for *Fill_BG_Grid* 160 ⟩
  {
    int i, j;
    if (*The_Grid* ≡ Λ) *Allocate_Grid*(*r.search*);
    ⟨ Zero *GG* 159 ⟩
    if (*Debug*(DEBUG_GRID)) *fprintf*(*stderr*, "GRID:␣Filling␣BG␣grid\n");
    Set_Calc_State(*m, r*);
    RR.*slab*.b = 1.0/32.0;
    RR.*slab*.a = RR.*default_a*;
    GG_a = RR.*slab*.a;
    for ($i = 0$; $i <$ GRID_SIZE; i++) {
      RR.*slab*.b *= 2;
      for ($j = 0$; $j <$ GRID_SIZE; j++) {
        RR.*slab*.g = MAX_ABS_G * (2.0 * j/(GRID_SIZE − 1.0) − 1.0);
        fill_grid_entry(i, j);
      }
    }
    The_Grid_Initialized = TRUE;
    The_Grid_Search = FIND_BG;
  }
This code is used in section 115.

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

⟨ Prototype for *Fill_BaG_Grid* 162 ⟩ ≡
  void *Fill_BaG_Grid*(**struct measure_type** m, **struct invert_type** r)
This code is used in sections 116 and 163.

163.   ⟨ Definition for *Fill_BaG_Grid* 163 ⟩ ≡
  ⟨ Prototype for *Fill_BaG_Grid* 162 ⟩
  {
    int i, j;
    double bs, ba;
    if (*The_Grid* ≡ Λ) *Allocate_Grid*(*r.search*);
    ⟨ Zero *GG* 159 ⟩
    if (*Debug*(DEBUG_GRID)) *fprintf*(*stderr*, "GRID:␣Filling␣BaG␣grid\n");
    Set_Calc_State(*m, r*);
    $ba = 1.0/32.0$;
    $bs =$ RR.*default_bs*;
    $GG\_bs = bs$;
    for ($i = 0$; $i <$ GRID_SIZE; i++) {
      ba *= 2;
      $ba = exp((\textbf{double})\, i/(\text{GRID\_SIZE} - 1.0) * log(1024.0))/16.0$;
      RR.*slab.b* $= ba + bs$;
      if (RR.*slab.b* > 0) RR.*slab.a* $= bs/$RR.*slab.b*;
      else RR.*slab.a* $= 0$;
      for ($j = 0$; $j <$ GRID_SIZE; j++) {
        RR.*slab.g* $=$ MAX_ABS_G $* (2.0 * j/(\text{GRID\_SIZE} - 1.0) - 1.0)$;
        fill_grid_entry(i, j);
      }
    }
    The_Grid_Initialized = TRUE;
    The_Grid_Search = *FIND_BaG*;
  }
This code is used in section 115.

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

⟨ Prototype for *Fill_BsG_Grid* 164 ⟩ ≡
  void *Fill_BsG_Grid*(**struct measure_type** m, **struct invert_type** r)
This code is used in sections 116 and 165.

165.   ⟨ Definition for *Fill_BsG_Grid* 165 ⟩ ≡
  ⟨ Prototype for *Fill_BsG_Grid* 164 ⟩
  {
    int *i, j*;
    double *bs*, *ba*;
    if (*The_Grid* ≡ Λ) *Allocate_Grid*(*r.search*);
    ⟨ Zero *GG* 159 ⟩
    Set_Calc_State(*m, r*);
    bs = 1.0/32.0;
    ba = RR.*default_ba*;
    GG_ba = *ba*;
    for (*i* = 0; *i* < GRID_SIZE; *i*++) {
      bs ∗= 2;
      RR.*slab.b* = *ba* + *bs*;
      if (RR.*slab.b* > 0) RR.*slab.a* = *bs*/RR.*slab.b*;
      else RR.*slab.a* = 0;
      for (*j* = 0; *j* < GRID_SIZE; *j*++) {
        RR.*slab.g* = MAX_ABS_G ∗ (2.0 ∗ *j*/(GRID_SIZE − 1.0) − 1.0);
        fill_grid_entry(*i, j*);
      }
    }
    The_Grid_Initialized = TRUE;
    The_Grid_Search = *FIND_BsG*;
  }

This code is used in section 115.

166.   ⟨ Prototype for *Fill_Grid* 166 ⟩ ≡
  void *Fill_Grid*(**struct measure_type** *m*, **struct invert_type** *r*, **int** *force_new*)

This code is used in sections 116 and 167.

167.   ⟨ Definition for *Fill_Grid* 167 ⟩ ≡
  ⟨ Prototype for *Fill_Grid* 166 ⟩
  {
    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 115.

168.   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* 168 ⟩ ≡
   void *Calculate_Distance*(**double** ∗M_R, **double** ∗M_T, **double** ∗*deviation*)
This code is used in sections 116 and 169.

169.   ⟨ Definition for *Calculate_Distance* 169 ⟩ ≡
  ⟨ Prototype for *Calculate_Distance* 168 ⟩
  {
    double *Rc*, *Tc*, *ur1*, *ut1*, *uru*, *utu*;

    if (RR.*slab*.*b* ≤ $1 \cdot 10^{-6}$) RR.*slab*.*b* = $1 \cdot 10^{-6}$;
    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*, &*Rc*, &*Tc*);
    if ((¬CALCULATING_GRID ∧ *Debug*(DEBUG_ITERATIONS)) ∨ (CALCULATING_GRID ∧
        Debug(DEBUG_GRID_CALC))) *fprintf*(*stderr*, "␣␣␣␣␣␣␣␣␣");
    Calculate_Distance_With_Corrections(*ur1*, *ut1*, *Rc*, *Tc*, *uru*, *utu*, M_R, M_T, *deviation*);
  }
This code is used in section 115.

170.   ⟨ Prototype for *Calculate_Grid_Distance* 170 ⟩ ≡
  double *Calculate_Grid_Distance*(**int** *i*, **int** *j*)
This code is used in sections 116 and 171.

171. ⟨Definition for *Calculate_Grid_Distance* 171⟩ ≡
  ⟨Prototype for *Calculate_Grid_Distance* 170⟩
  {
    double *ur1*, *ut1*, *uru*, *utu*, *Rc*, *Tc*, *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*, &*Rc*, &*Tc*);
    CALCULATING_GRID = 1;
    Calculate_Distance_With_Corrections(*ur1*, *ut1*, *Rc*, *Tc*, *uru*, *utu*, &LR, &LT, &*dev*);
    CALCULATING_GRID = 0;
    return *dev*;
  }

This code is used in section 115.

172.   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.

  Rc and *Tc* 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* 172⟩ ≡
  void *Calculate_Distance_With_Corrections*(**double** UR1, **double** UT1, **double** *Rc*, **double** *Tc*, **double**
      URU, **double** UTU, **double** ∗M_R, **double** ∗M_T, **double** ∗*dev*)

This code is used in sections 116 and 173.

173.   ⟨ Definition for *Calculate_Distance_With_Corrections* 173 ⟩ ≡
   ⟨ Prototype for *Calculate_Distance_With_Corrections* 172 ⟩
   {
      double *R_direct*, *T_direct*, *R_diffuse*, *T_diffuse*;

      R_diffuse = URU − MM.*uru_lost*;
      if (*R_diffuse* < 0) *R_diffuse* = 0;
      T_diffuse = UTU − MM.*utu_lost*;
      if (*T_diffuse* < 0) *T_diffuse* = 0;
      R_direct = UR1 − MM.*ur1_lost* − (1.0 − MM.*fraction_of_rc_in_mr*) ∗ *Rc*;
      T_direct = UT1 − MM.*ut1_lost* − (1.0 − MM.*fraction_of_tc_in_mt*) ∗ *Tc*;
      switch (MM.*num_spheres*) {
      case 0: ⟨ Calc M_R and M_T for no spheres 174 ⟩
         break;
      case 1:
         if (MM.*method* ≡ COMPARISON) {
            ⟨ Calc M_R and M_T for dual beam sphere 178 ⟩
         }
         else {
            ⟨ Calc M_R and M_T for single beam sphere 175 ⟩
         }
         break;
      case 2: ⟨ Calc M_R and M_T for two spheres 179 ⟩
         break;
      default: *fprintf*(*stderr*, "Bad␣number␣of␣spheres␣=␣%d\n", MM.*num_spheres*);
      }
      ⟨ Calculate the deviation 180 ⟩
      ⟨ Print diagnostics 183 ⟩
   }

This code is used in section 115.

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

⟨ Calc M_R and M_T for no spheres 174 ⟩ ≡
   {
      ∗M_R = *R_direct*;
      ∗M_T = *T_direct*;
   }

This code is used in section 173.

175.   Define a bunch of temporary variable names

⟨ Calc M_R and M_T for single beam sphere 175 ⟩ ≡
   double *P_std*, *P*, P_0, *G*, G_0, *G_std*;
   int *tmp*;

See also sections 176 and 177.

This code is used in section 173.

176.   In a reflection experiment, some fraction f of the incident light P_i might hit the wall first. Thus the incident power on the sample is $(1 - f)P_i$ and the incident power on the sphere wall will be fP_i. The diffuse reflection entering the sphere depends on the presence of a baffle.

If a baffle is present then

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

and when there is no baffle

$$P_d = [a_d P_i](r_s^{\text{direct}} * (1 - f) + r_w f)G(r_s)$$

Since the quantities in square brackets are identical for $R(r_s^{\text{direct}}, r_s)$, $R(0,0)$, and $R(r_{\text{std}}, r_{\text{std}})$ and they all cancel out when calculating the normalized reflection measurement

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

This leads to the following code for `M_R`

$\langle$ Calc `M_R` and `M_T` for single beam sphere $175 \rangle +\equiv$
```
  G_0 = Gain(REFLECTION_SPHERE, MM, 0.0);
  G = Gain(REFLECTION_SPHERE, MM, R_diffuse);
  G_std = Gain(REFLECTION_SPHERE, MM, MM.rstd_r);
  P = G * (R_direct * (1 − MM.f_r) + MM.f_r * MM.rw_r);
  P_std = G_std * (MM.rstd_r * (1 − MM.f_r) + MM.f_r * MM.rw_r);
  P_0 = G_0 * (MM.f_r * MM.rw_r);
  *M_R = MM.rstd_r * (P − P_0)/(P_std − P_0);
```

177.   In a transmission experiment, the calculations are simpler and harder. First, the value of $T(0,0) = 0$ because computationally, there is no dark noise in the detector nor any possible light leakage from the outside into the sphere. This simplifies

$$M_T = r_0 \cdot \frac{T(t_s^{\,\text{direct}}, r_s) - T(0,0)}{T(t_{\text{std}}, r_{\text{std}}) - T(0,0)}$$

to

$$M_T = r_0 \cdot \frac{T(t_s^{\,\text{direct}}, r_s)}{T(t_{\text{std}}, r_{\text{std}})}$$

where r_0 might be r_{std} or r_w for the transmission sphere.

We do not need to worry about some fraction of the incident light P_i hitting the sphere wall before interacting with the sample.

Finally, if the transmission sphere has a baffle present for the sample measurement, then it is no longer in the right place and diffuse light entering the sphere is just $[a_d P_i] r_0$

When a baffle is present then the light falling on the detector in a transmission experiment is

$$P_d = T(t_s^{\,\text{direct}}, r_s) = [a_d P_i](1 - a_e) r_w t_s^{\,\text{direct}} G(r_s)$$

and with no baffle present

$$P_d = T(t_s^{\,\text{direct}}, r_s) = [a_d P_i] t_s^{\,\text{direct}} G(r_s)$$

The normalization $T(t_{\text{std}}, r_{\text{std}})$ can be measured in two ways.

When transmission measurements are made, typically the empty port (the one that let the light into the sphere for the reflection measurement) is filled with a white port cover whose reflectance matches the rest of the sphere. In this case, the natural way to make the standard transmission measurement is to shine the beam through the empty sample port onto the back side of the sphere. If the baffle was properly placed for the transmission experiment (between the sample port and the detector) then the calibration transmission measurement is now made in a sphere without a baffle. In addition, the beam is diffused only after bouncing off the sphere wall. Therefore the power falling on the detector is

$$P_{\text{std}} = T(1.0, r_w) = [a_d P_i] r_w G(0)$$

An alternate experiment is when there is an empty port in the sphere (perhaps to allow the unscattered light to leave). In any case, the calibration measurement is done by removing the sample and placing the calibration standare in what used to be the empty port. In this case, the roles of the sample and empty ports have switched. Consequently, the areas of the sample and empty ports must be swapped before the gain is calculated.

$$P_{\text{std}} = T(1.0, r_{\text{std}}) = [a_d P_i] r_{\text{std}} G(r_{\text{std}})$$

Note that r_w or r_{std} in P_{std} term cancel with r_0 when calculating M_T. Further, the quantities $a_d P_i$ also cancel.

⟨ Calc `M_R` and `M_T` for single beam sphere 175 ⟩ +≡
    $P = T\_direct * Gain(\texttt{TRANSMISSION\_SPHERE}, \text{MM}, R\_diffuse)$;
    if (MM.*baffle_t*) $P \mathrel{*}= (1 - \text{MM}.ae\_t) * \text{MM}.rw\_t$;
    $tmp = \text{MM}.baffle\_t$;
    MM.*baffle_t* = FALSE;
    if (MM.*ae_t* ≡ 0) {
        $P\_std = Gain(\texttt{TRANSMISSION\_SPHERE}, \text{MM}, 0)$;
    }
    else {
        SWAP(MM.*ae_t*, MM.*as_t*);
        $P\_std = Gain(\texttt{TRANSMISSION\_SPHERE}, \text{MM}, \text{MM}.rstd\_t)$;
        SWAP(MM.*ae_t*, MM.*as_t*);
    }
    MM.*baffle_t* = tmp;
    *M_T = P/P_std;

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

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

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

or

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

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

The normalized transmittance is simply t_s^{direct}.

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

⟨ Calc `M_R` and `M_T` for dual beam sphere 178 ⟩ ≡
```
  {
    *M_R = (1 − MM.f_r) ∗ R_direct /((1 − MM.f_r) + MM.f_r ∗ MM.rw_r /MM.rstd_r);
    *M_T = T_direct;
  }
```
This code is used in section 173.

179.   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 179 ⟩ ≡
```
  {
    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, R_direct, R_diffuse, T_direct,
        T_diffuse) − R_0)/(Two_Sphere_R(MM, MM.rstd_r, MM.rstd_r, 0, 0) − R_0);
    *M_T = (Two_Sphere_T(MM, R_direct, R_diffuse, T_direct, T_diffuse) − T_0)/(Two_Sphere_T(MM, 0, 0, 1,
        1) − T_0);
  }
```
This code is used in section 173.

180. 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 180 ⟩ ≡
  if (RR.*search* ≡ FIND_A ∨ RR.*search* ≡ FIND_G ∨ RR.*search* ≡ FIND_B ∨ RR.*search* ≡ *FIND_Bs* ∨ RR.*search* ≡
      FIND_Ba) {
    ⟨ One parameter deviation 181 ⟩
  }
  else {
    ⟨ Two parameter deviation 182 ⟩
  }

This code is used in section 173.

181. 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 181 ⟩ ≡
  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 180.

182. 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 182 ⟩ ≡
  if (RR.*metric* ≡ RELATIVE) {
    if (MM.$m_t > $ABIT) $*dev = T_TRUST_FACTOR * fabs($MM.$m_t - *M_T)/($*T_diffuse* + ABIT);
    if (RR.*default_a* ≠ 0) {
      $*dev\ += fabs($MM.$m\_r - *$M_R$)/($*R_diffuse* + 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 180.

183.   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 183 ⟩ ≡
  if ((*Debug*(DEBUG_ITERATIONS) ∧ ¬CALCULATING_GRID) ∨
        (*Debug*(DEBUG_GRID_CALC) ∧ CALCULATING_GRID)) {
    fprintf (*stderr*, "%10.5f␣%10.5f␣%10.5f␣|", RR.*slab*.*a*, RR.*slab*.*b*, RR.*slab*.*g*);
    fprintf (*stderr*, "␣%10.5f␣%10.5f␣|", MM.*m_r*, *M_R);
    fprintf (*stderr*, "␣%10.5f␣%10.5f␣|", MM.*m_t*, *M_T);
    fprintf (*stderr*, "%10.3f\n", *dev*);
  }

This code is used in section 173.

184.   ⟨ Prototype for *Find_AG_fn* 184 ⟩ ≡
  double *Find_AG_fn*(**double** *x*[ ])

This code is used in sections 116 and 185.

185.   ⟨ Definition for *Find_AG_fn* 185 ⟩ ≡
  ⟨ Prototype for *Find_AG_fn* 184 ⟩
  {
    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 115.

186.   ⟨ Prototype for *Find_AB_fn* 186 ⟩ ≡
  double *Find_AB_fn*(**double** *x*[ ])

This code is used in sections 116 and 187.

187.   ⟨ Definition for *Find_AB_fn* 187 ⟩ ≡
  ⟨ Prototype for *Find_AB_fn* 186 ⟩
  {
    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 115.

188.   ⟨ Prototype for *Find_Ba_fn* 188 ⟩ ≡
  double *Find_Ba_fn*(**double** *x*)

This code is used in sections 116 and 189.

189. 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* 189 ⟩ ≡
  ⟨ Prototype for *Find_Ba_fn* 188 ⟩
  {
    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 115.

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

⟨ Prototype for *Find_Bs_fn* 190 ⟩ ≡
  double *Find_Bs_fn*(**double** x)

This code is used in sections 116 and 191.

191. ⟨ Definition for *Find_Bs_fn* 191 ⟩ ≡
  ⟨ Prototype for *Find_Bs_fn* 190 ⟩
  {
    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 115.

192. ⟨ Prototype for *Find_A_fn* 192 ⟩ ≡
  double *Find_A_fn*(**double** x)

This code is used in sections 116 and 193.

193. ⟨ Definition for *Find_A_fn* 193 ⟩ ≡
  ⟨ Prototype for *Find_A_fn* 192 ⟩
  {
    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 115.

194. ⟨ Prototype for *Find_B_fn* 194 ⟩ ≡
  double *Find_B_fn*(**double** *x*)

This code is used in sections 116 and 195.

195. ⟨ Definition for *Find_B_fn* 195 ⟩ ≡
  ⟨ Prototype for *Find_B_fn* 194 ⟩
  {
    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 115.

196. ⟨ Prototype for *Find_G_fn* 196 ⟩ ≡
  double *Find_G_fn*(**double** *x*)

This code is used in sections 116 and 197.

197. ⟨ Definition for *Find_G_fn* 197 ⟩ ≡
  ⟨ Prototype for *Find_G_fn* 196 ⟩
  {
    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 115.

198. ⟨ Prototype for *Find_BG_fn* 198 ⟩ ≡
  double *Find_BG_fn*(**double** *x*[ ])

This code is used in sections 116 and 199.

199. ⟨ Definition for *Find_BG_fn* 199 ⟩ ≡
  ⟨ Prototype for *Find_BG_fn* 198 ⟩
  {
    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 115.

200. 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* 200 ⟩ ≡
  double *Find_BaG_fn*(**double** *x*[ ])

This code is used in sections 116 and 201.

201. ⟨ Definition for *Find_BaG_fn* 201 ⟩ ≡
  ⟨ Prototype for *Find_BaG_fn* 200 ⟩
  {
    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 115.

202. ⟨ Prototype for *Find_BsG_fn* 202 ⟩ ≡
  double *Find_BsG_fn*(**double** $x[\,]$)

This code is used in sections 116 and 203.

203. ⟨ Definition for *Find_BsG_fn* 203 ⟩ ≡
  ⟨ Prototype for *Find_BsG_fn* 202 ⟩
  {
    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 115.

204. 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* 204 ⟩ ≡
  double *maxloss*(**double** f)

This code is used in sections 116 and 205.

205.   ⟨Definition for *maxloss* 205⟩ ≡
  ⟨Prototype for *maxloss* 204⟩
  {
      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 115.

206.   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* 206⟩ ≡
  void *Max_Light_Loss*(**struct measure_type** *m*, **struct invert_type** *r*, **double** ∗*ur1_loss*, **double** ∗*ut1_loss*)

This code is used in sections 116 and 207.

207.   ⟨Definition for *Max_Light_Loss* 207⟩ ≡
  ⟨Prototype for *Max_Light_Loss* 206⟩
  {
      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 115.

208.   this is currently unused

⟨ Unused diffusion fragment 208 ⟩ ≡

```
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;
}
```

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

⟨ `iad_find.c`   209 ⟩ ≡
#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* 223 ⟩
   ⟨ Definition for *U_Find_Bs* 221 ⟩
   ⟨ Definition for *U_Find_A* 225 ⟩
   ⟨ Definition for *U_Find_B* 229 ⟩
   ⟨ Definition for *U_Find_G* 227 ⟩
   ⟨ Definition for *U_Find_AG* 232 ⟩
   ⟨ Definition for *U_Find_AB* 212 ⟩
   ⟨ Definition for *U_Find_BG* 237 ⟩
   ⟨ Definition for *U_Find_BaG* 243 ⟩
   ⟨ Definition for *U_Find_BsG* 248 ⟩

210.   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`   210 ⟩ ≡
   ⟨ Prototype for *U_Find_Ba* 222 ⟩;
   ⟨ Prototype for *U_Find_Bs* 220 ⟩;
   ⟨ Prototype for *U_Find_A* 224 ⟩;
   ⟨ Prototype for *U_Find_B* 228 ⟩;
   ⟨ Prototype for *U_Find_G* 226 ⟩;
   ⟨ Prototype for *U_Find_AG* 231 ⟩;
   ⟨ Prototype for *U_Find_AB* 211 ⟩;
   ⟨ Prototype for *U_Find_BG* 236 ⟩;
   ⟨ Prototype for *U_Find_BaG* 242 ⟩;
   ⟨ Prototype for *U_Find_BsG* 247 ⟩;

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

⟨ Prototype for *U_Find_AB* 211 ⟩ ≡
  void *U_Find_AB*(**struct measure_type** *m*, **struct invert_type** *∗r*)

This code is used in sections 210 and 212.

212.   ⟨ Definition for *U_Find_AB* 212 ⟩ ≡
  ⟨ Prototype for *U_Find_AB* 211 ⟩
  {
    ⟨ Allocate local simplex variables 213 ⟩
    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* 214 ⟩
    ⟨ Initialize the nodes of the *a* and *b* simplex 215 ⟩
    ⟨ Evaluate the *a* and *b* simplex at the nodes 216 ⟩
    amoeba(*p*, *y*, 2, *r→tolerance*, *Find_AB_fn*, &*r→AD_iterations*);
    ⟨ Choose the best node of the *a* and *b* simplex 217 ⟩
    ⟨ Free simplex data structures 219 ⟩
    ⟨ Put final values in result 218 ⟩
  }

This code is used in section 209.

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

⟨ Allocate local simplex variables 213 ⟩ ≡
  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 212, 232, 237, 243, and 248.

214. 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 214 ⟩ ≡

```
{     /* double a3,b3,g3; */
  size_t count = NUMBER_OF_GUESSES;      /* distance to last result */

  abg_distance(r→slab.a, r→slab.b, r→slab.g, &(guess[0]));
  if (¬Valid_Grid(m, r→search)) Fill_Grid(m, *r, 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 212, 232, 237, 243, and 248.

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

```
{
   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 212.

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

```
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 212.

217.   ⟨Choose the best node of the a and b simplex 217⟩ ≡
  r→$final\_distance = 10$;
  for $(i = 1;\ i \leq 3;\ i\mathrel{+}\mathrel{+})$ {
    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 212.

218.   ⟨Put final values in result 218⟩ ≡
  r→$a = r$→$slab.a$;
  r→$b = r$→$slab.b$;
  r→$g = r$→$slab.g$;
  r→$found = (r$→$tolerance \leq r$→$final\_distance)$;
  $Set\_Calc\_State(m, *r)$;

This code is used in sections 212, 221, 223, 225, 227, 229, 232, 237, 243, and 248.

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

⟨Free simplex data structures 219⟩ ≡
  $free\_dvector(x, 1, 2)$;
  $free\_dvector(y, 1, 3)$;
  $free\_dmatrix(p, 1, 3, 1, 2)$;

This code is used in sections 212, 232, 237, 243, and 248.

220. 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* 220 ⟩ ≡
   void *U_Find_Bs*(**struct measure_type** *m*, **struct invert_type** ∗*r*)

This code is used in sections 210 and 221.

221.   ⟨ Definition for *U_Find_Bs* 221 ⟩ ≡
  ⟨ Prototype for *U_Find_Bs* 220 ⟩
  {
    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→*final_distance* = *brent*(*ax*, *bx*, *cx*, *Find_Bs_fn*, *r*→*tolerance*, &*bs*);   /∗ recover true values ∗/
    r→*slab.a* = *bcalc2b*(*bs*)/(*bcalc2b*(*bs*) + *r*→*slab.b*);
    r→*slab.b* = *bcalc2b*(*bs*) + *r*→*slab.b*;
    ⟨ Put final values in result 218 ⟩
  }

This code is used in section 209.

222.   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* 222 ⟩ ≡
  void *U_Find_Ba*(**struct measure_type** *m*, **struct invert_type** *∗r*)

This code is used in sections 210 and 223.

223.   ⟨ Definition for *U_Find_Ba* 223 ⟩ ≡
  ⟨ Prototype for *U_Find_Ba* 222 ⟩
  {
    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→final_distance = *brent*(*ax*, *bx*, *cx*, *Find_Ba_fn*, *r→tolerance*, &*ba*);   /∗ recover true values ∗/
    r→slab.a = (*r→slab.b*)/(*bcalc2b*(*ba*) + *r→slab.b*);
    r→slab.b = *bcalc2b*(*ba*) + *r→slab.b*;   /∗ actual value of b ∗/
    ⟨ Put final values in result 218 ⟩
  }

This code is used in section 209.

224.   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 224 ⟩ ≡
  void U_Find_A(**struct measure_type** m, **struct invert_type** $*r$)

This code is used in sections 210 and 225.

225.   ⟨ Definition for U_Find_A 225 ⟩ ≡
  ⟨ Prototype for U_Find_A 224 ⟩
  {
    double Rt, Tt, Rd, Rc, Td, Tc;
    if ($Debug$(DEBUG_SEARCH)) {
      $fprintf$($stderr$, "SEARCH:␣Using␣U_Find_A()");
      $fprintf$($stderr$, "␣(mu=%6.4f)", $r{\rightarrow}slab.cos\_angle$);
      if ($r{\rightarrow}default\_b \neq$ UNINITIALIZED) $fprintf$($stderr$, "␣␣default_b␣=␣%8.5f", $r{\rightarrow}default\_b$);
      if ($r{\rightarrow}default\_g \neq$ UNINITIALIZED) $fprintf$($stderr$, "␣␣default_g␣=␣%8.5f", $r{\rightarrow}default\_g$);
      $fprintf$($stderr$, "\n");
    }
    $Estimate\_RT$($m, *r, \&Rt, \&Tt, \&Rd, \&Rc, \&Td, \&Tc$);
    $r{\rightarrow}slab.g = (r{\rightarrow}default\_g \equiv$ UNINITIALIZED$)\ ?\ 0 : r{\rightarrow}default\_g$;
    $r{\rightarrow}slab.b = (r{\rightarrow}default\_b \equiv$ UNINITIALIZED$)\ ?$ HUGE_VAL $: r{\rightarrow}default\_b$;
    $r{\rightarrow}slab.a = 0.0$;
    $r{\rightarrow}final\_distance = 0.0$;
    Set_Calc_State($m, *r$);
    if ($Rt > 0.99999$) {
      $r{\rightarrow}final\_distance = Find\_A\_fn$($a2acalc$(1.0));
      $r{\rightarrow}slab.a = 1.0$;
    }
    else {
      double $x, ax, bx, cx, fa, fb, fc$;
      $ax = a2acalc$(0.3);
      $bx = a2acalc$(0.5);
      $mnbrak$($\&ax, \&bx, \&cx, \&fa, \&fb, \&fc, Find\_A\_fn$);
      $r{\rightarrow}final\_distance = brent$($ax, bx, cx, Find\_A\_fn, r{\rightarrow}tolerance, \&x$);
      $r{\rightarrow}slab.a = acalc2a$($x$);
    }
    ⟨ Put final values in result 218 ⟩
  }

This code is used in section 209.

226. Fixed Optical Depth and Albedo.

⟨ Prototype for *U_Find_G* 226 ⟩ ≡
  void *U_Find_G*(**struct measure_type** *m*, **struct invert_type** *∗r*)

This code is used in sections 210 and 227.

227. ⟨ Definition for *U_Find_G* 227 ⟩ ≡
  ⟨ Prototype for *U_Find_G* 226 ⟩
  {
    double *Rt*, *Tt*, *Rd*, *Rc*, *Td*, *Tc*;
    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*, &*Rc*, &*Td*, &*Tc*);
    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→final_distance = *brent*(*ax*, *bx*, *cx*, *Find_G_fn*, *r→tolerance*, &*x*);
    r→slab.g = *gcalc2g*(*x*);
    Set_Calc_State(*m*, *∗r*);
    ⟨ Put final values in result 218 ⟩
  }

This code is used in section 209.

228.   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 228 ⟩ ≡
  void U_Find_B(**struct measure_type** m, **struct invert_type** $*r$)

This code is used in sections 210 and 229.

229.   ⟨ Definition for U_Find_B 229 ⟩ ≡
  ⟨ Prototype for U_Find_B 228 ⟩
  {
    double Rt, Tt, Rd, Rc, Td, Tc;
    if ($Debug$(DEBUG_SEARCH)) {
      $fprintf$($stderr$, "SEARCH:␣Using␣U_Find_B()");
      $fprintf$($stderr$, "␣(mu=%6.4f)", $r{\rightarrow}slab.cos\_angle$);
      if ($r{\rightarrow}default\_a \neq$ UNINITIALIZED) $fprintf$($stderr$, "␣␣default_a␣=␣%8.5f", $r{\rightarrow}default\_a$);
      if ($r{\rightarrow}default\_g \neq$ UNINITIALIZED) $fprintf$($stderr$, "␣␣default_g␣=␣%8.5f", $r{\rightarrow}default\_g$);
      $fprintf$($stderr$, "\n");
    }
    $Estimate\_RT$(m, $*r$, &Rt, &Tt, &Rd, &Rc, &Td, &Tc);
    $r{\rightarrow}slab.g = (r{\rightarrow}default\_g \equiv$ UNINITIALIZED$)\ ?\ 0 : r{\rightarrow}default\_g$;
    $r{\rightarrow}slab.a = (r{\rightarrow}default\_a \equiv$ UNINITIALIZED$)\ ?\ 0 : r{\rightarrow}default\_a$;
    $r{\rightarrow}slab.b = 0.5$;
    $r{\rightarrow}final\_distance = 0.0$;
    Set_Calc_State(m, $*r$);
    ⟨ Iteratively solve for b 230 ⟩
    ⟨ Put final values in result 218 ⟩
  }

This code is used in section 209.

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

⟨ Iteratively solve for b 230 ⟩ ≡
  {
    double $x, ax, bx, cx, fa, fb, fc$;
    $ax = b2bcalc(0.1)$;
    $bx = b2bcalc(10)$;
    $mnbrak$(&ax, &bx, &cx, &fa, &fb, &fc, $Find\_B\_fn$);
    $r{\rightarrow}final\_distance = brent(ax, bx, cx, Find\_B\_fn, r{\rightarrow}tolerance, \&x)$;
    $r{\rightarrow}slab.b = bcalc2b(x)$;
    Set_Calc_State(m, $*r$);
  }

This code is used in section 229.

231.   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 231⟩ ≡
    void U_Find_AG(**struct measure_type** m, **struct invert_type** $*r$)

This code is used in sections 210 and 232.

232.   ⟨Definition for U_Find_AG 232⟩ ≡
  ⟨Prototype for U_Find_AG 231⟩
  {
    ⟨Allocate local simplex variables 213⟩
    if ($Debug$(DEBUG_SEARCH)) {
      $fprintf$($stderr$, "SEARCH:␣Using␣U_Find_AG()");
      $fprintf$($stderr$, "␣(mu=%6.4f)", $r{\to}slab.cos\_angle$);
      if ($r{\to}default\_b \neq$ UNINITIALIZED) $fprintf$($stderr$, "␣␣default_b␣=␣%8.5f", $r{\to}default\_b$);
      $fprintf$($stderr$, "\n");
    }
    if ($m.num\_measures \equiv 3$) $r{\to}slab.b = What\_Is\_B$($r{\to}slab$, $m.m\_u$);
    else if ($r{\to}default\_b \equiv$ UNINITIALIZED) $r{\to}slab.b = 1$;
    else $r{\to}slab.b = r{\to}default\_b$;
    Set_Calc_State(m, $*r$);
    ⟨Get the initial a, b, and g 214⟩
    ⟨Initialize the nodes of the a and g simplex 233⟩
    ⟨Evaluate the a and g simplex at the nodes 234⟩
    $amoeba$(p, y, 2, $r{\to}tolerance$, $Find\_AG\_fn$, &$r{\to}AD\_iterations$);
    ⟨Choose the best node of the a and g simplex 235⟩
    ⟨Free simplex data structures 219⟩
    ⟨Put final values in result 218⟩
  }

This code is used in section 209.

233. ⟨Initialize the nodes of the a and g simplex 233⟩ ≡

```
{
  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 232.

234. ⟨Evaluate the a and g simplex at the nodes 234⟩ ≡

```
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 232.

235.   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 235 ⟩ ≡

```
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 232.

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

⟨ Prototype for *U_Find_BG* 236 ⟩ ≡
 void *U_Find_BG*(**struct measure_type** *m*, **struct invert_type** *∗r*)

This code is used in sections 210 and 237.

237.   ⟨ Definition for *U_Find_BG* 237 ⟩ ≡
 ⟨ Prototype for *U_Find_BG* 236 ⟩
 {
  ⟨ Allocate local simplex variables 213 ⟩
  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* 214 ⟩
  ⟨ Initialize the nodes of the *b* and *g* simplex 239 ⟩
  ⟨ Evaluate the *bg* simplex at the nodes 240 ⟩
  amoeba(*p*, *y*, 2, *r*→*tolerance*, *Find_BG_fn*, &*r*→*AD_iterations*);
  ⟨ Choose the best node of the *b* and *g* simplex 241 ⟩
  ⟨ Free simplex data structures 219 ⟩
  ⟨ Put final values in result 218 ⟩
 }

This code is used in section 209.

238.   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.

239. ⟨Initialize the nodes of the b and g simplex 239⟩ ≡

```
{
    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 237.

240. ⟨Evaluate the bg simplex at the nodes 240⟩ ≡

```
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 237.

241.   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 241 ⟩ ≡
```
  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 237.

242.   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 242 ⟩ ≡
   void U_Find_BaG(**struct measure_type** m, **struct invert_type** $*r$)

This code is used in sections 210 and 243.

243.   ⟨ Definition for U_Find_BaG 243 ⟩ ≡
   ⟨ Prototype for U_Find_BaG 242 ⟩
   {
      ⟨ Allocate local simplex variables 213 ⟩
      $Set\_Calc\_State(m, *r)$;
      ⟨ Get the initial a, b, and g 214 ⟩
      ⟨ Initialize the nodes of the ba and g simplex 244 ⟩
      ⟨ Evaluate the BaG simplex at the nodes 245 ⟩
      $amoeba(p, y, 2, r\rightarrow tolerance, Find\_BaG\_fn, \&r\rightarrow AD\_iterations)$;
      ⟨ Choose the best node of the ba and g simplex 246 ⟩
      ⟨ Free simplex data structures 219 ⟩
      ⟨ Put final values in result 218 ⟩
   }

This code is used in section 209.

244.   ⟨ Initialize the nodes of the ba and g simplex 244 ⟩ ≡
   if $(guess[0].b > r\rightarrow default\_bs)$ {
      $p[1][1] = b2bcalc(guess[0].b - r\rightarrow default\_bs)$;
      $p[2][1] = b2bcalc(2*(guess[0].b - r\rightarrow default\_bs))$;
      $p[3][1] = p[1][1]$;
   }
   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 243.

245.   ⟨ Evaluate the BaG simplex at the nodes 245 ⟩ ≡
   for $(i = 1;\ i \leq 3;\ i{+}{+})$ {
      $x[1] = p[i][1]$;
      $x[2] = p[i][2]$;
      $y[i] = Find\_BaG\_fn(x)$;
   }

This code is used in section 243.

246.   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 246 ⟩ ≡
  r→*final_distance* = 10;
  for (i = 1; $i \leq 3$; 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 243.

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

$$b_a = \mu_a d$$

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

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

This code is used in sections 210 and 248.

248. ⟨Definition for U_Find_BsG 248⟩ ≡
  ⟨Prototype for U_Find_BsG 247⟩
  {
    ⟨Allocate local simplex variables 213⟩
    if ($Debug$(DEBUG_SEARCH)) {
      $fprintf$($stderr$, "SEARCH:␣Using␣U_Find_BsG()");
      $fprintf$($stderr$, "␣(mu=%6.4f)", $r{\rightarrow}slab.cos\_angle$);
      if ($r{\rightarrow}default\_ba \neq$ UNINITIALIZED) $fprintf$($stderr$, "␣␣default_ba␣=␣%8.5f", $r{\rightarrow}default\_ba$);
      $fprintf$($stderr$, "\n");
    }
    Set_Calc_State(m, $*r$);
    ⟨Get the initial a, b, and g 214⟩
    ⟨Initialize the nodes of the bs and g simplex 249⟩
    ⟨Evaluate the BsG simplex at the nodes 250⟩
    $amoeba$(p, y, 2, $r{\rightarrow}tolerance$, $Find\_BsG\_fn$, &$r{\rightarrow}AD\_iterations$);
    ⟨Choose the best node of the bs and g simplex 251⟩
    ⟨Free simplex data structures 219⟩
    ⟨Put final values in result 218⟩
  }

This code is used in section 209.

249. ⟨Initialize the nodes of the bs and g simplex 249⟩ ≡
  $p[1][1] = b2bcalc$($guess[0].b - r{\rightarrow}default\_ba$);
  $p[1][2] = g2gcalc$($guess[0].g$);
  $p[2][1] = b2bcalc$($2 * guess[0].b - 2 * r{\rightarrow}default\_ba$);
  $p[2][2] = p[1][2]$;
  $p[3][1] = p[1][1]$;
  $p[3][2] = g2gcalc$($0.9 * guess[0].g + 0.05$);

This code is used in section 248.

250. ⟨Evaluate the BsG simplex at the nodes 250⟩ ≡
  for ($i = 1$; $i \leq 3$; i++) {
    $x[1] = p[i][1]$;
    $x[2] = p[i][2]$;
    $y[i] = Find\_BsG\_fn$(x);
  }

This code is used in section 248.

251.   ⟨ Choose the best node of the *bs* and *g* simplex 251 ⟩ ≡

   r→*final_distance* = 10;

   for $(i = 1;\ i \leq 3;\ i{+}{+})$ {

      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 248.

252. IAD Utilities.
   March 1995. Reincluded *quick_guess* code.

⟨ iad_util.c   252 ⟩ ≡
#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* 255 ⟩
   ⟨ Definition for *Estimate_RT* 261 ⟩
   ⟨ Definition for *a2acalc* 268 ⟩
   ⟨ Definition for *acalc2a* 270 ⟩
   ⟨ Definition for *g2gcalc* 272 ⟩
   ⟨ Definition for *gcalc2g* 274 ⟩
   ⟨ Definition for *b2bcalc* 276 ⟩
   ⟨ Definition for *bcalc2b* 278 ⟩
   ⟨ Definition for *twoprime* 280 ⟩
   ⟨ Definition for *twounprime* 282 ⟩
   ⟨ Definition for *abgg2ab* 284 ⟩
   ⟨ Definition for *abgb2ag* 286 ⟩
   ⟨ Definition for *quick_guess* 293 ⟩
   ⟨ Definition for *Set_Debugging* 306 ⟩
   ⟨ Definition for *Debug* 308 ⟩
   ⟨ Definition for *Print_Invert_Type* 310 ⟩
   ⟨ Definition for *Print_Measure_Type* 312 ⟩

253. ⟨ iad_util.h   253 ⟩ ≡
   ⟨ Prototype for *What_Is_B* 254 ⟩;
   ⟨ Prototype for *Estimate_RT* 260 ⟩;
   ⟨ Prototype for *a2acalc* 267 ⟩;
   ⟨ Prototype for *acalc2a* 269 ⟩;
   ⟨ Prototype for *g2gcalc* 271 ⟩;
   ⟨ Prototype for *gcalc2g* 273 ⟩;
   ⟨ Prototype for *b2bcalc* 275 ⟩;
   ⟨ Prototype for *bcalc2b* 277 ⟩;
   ⟨ Prototype for *twoprime* 279 ⟩;
   ⟨ Prototype for *twounprime* 281 ⟩;
   ⟨ Prototype for *abgg2ab* 283 ⟩;
   ⟨ Prototype for *abgb2ag* 285 ⟩;
   ⟨ Prototype for *quick_guess* 292 ⟩;
   ⟨ Prototype for *Set_Debugging* 305 ⟩;
   ⟨ Prototype for *Debug* 307 ⟩;
   ⟨ Prototype for *Print_Invert_Type* 309 ⟩;
   ⟨ Prototype for *Print_Measure_Type* 311 ⟩;

254.   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* 254 ⟩ ≡
    double *What_Is_B*(**struct AD_slab_type** *slab*, **double** *Tc*)

This code is used in sections 253 and 255.

255.   ⟨ Definition for *What_Is_B* 255 ⟩ ≡
    ⟨ Prototype for *What_Is_B* 254 ⟩
    {
        double *r1*, *r2*, *t1*, *t2*, *mu_in_slab*;

        ⟨ Calculate specular reflection and transmission 256 ⟩
        ⟨ Check for bad values of *Tc* 257 ⟩
        ⟨ Solve if multiple internal reflections are not present 258 ⟩
        ⟨ Find thickness when multiple internal reflections are present 259 ⟩
    }

This code is used in section 252.

256.   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 256 ⟩ ≡
    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 255.

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

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

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

⟨ Check for bad values of *Tc* 257 ⟩ ≡
    if (*Tc* ≤ 0) **return** (HUGE_VAL);
    if (*Tc* ≥ *t1* * *t2*/(1 − *r1* * *r2*)) **return** (0.001);

This code is used in section 255.

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

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

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

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

$\langle$ Solve if multiple internal reflections are not present 258 $\rangle \equiv$
   if $(r1 \equiv 0 \vee r2 \equiv 0)$ **return** $(-slab.cos\_angle * log(Tc/t1/t2))$;
This code is used in section 255.

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

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

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

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

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

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

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

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

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

with the two roots

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

Substituting our coefficients

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

With some algebra, this can be shown to be

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

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

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

(Not very pretty, but straightforward enough.)

$\langle$ Find thickness when multiple internal reflections are present 259 $\rangle \equiv$
   {
      double B;
     $B = t1 * t2$;
     return $(-slab.cos\_angle * log(2 * Tc/(B + sqrt(B * B + 4 * Tc * Tc * r1 * r2))))$;
   }
This code is used in section 255.

260.   Estimating R and T.

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

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

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

⟨ Prototype for *Estimate_RT* 260 ⟩ ≡
　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 253 and 261.

261.   ⟨ Definition for *Estimate_RT* 261 ⟩ ≡
　⟨ Prototype for *Estimate_RT* 260 ⟩
　{
　　⟨ Calculate the unscattered transmission and reflection 262 ⟩
　　⟨ Estimate the backscattered reflection 263 ⟩
　　⟨ Estimate the scattered transmission 264 ⟩
　　⟨ Debug info for estimate RT 265 ⟩
　}

This code is used in section 252.

262.   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 262 ⟩ ≡
　Calculate_Minimum_MR(m, r, rc, tc);

This code is used in section 261.

263.   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 263 ⟩ ≡
```
  if (m.fraction_of_rc_in_mr) {
    *rt = m.m_r;
    *rd = *rt − m.fraction_of_rc_in_mr * (*rc);
    if (*rd < 0) {
      *rd = 0;
      *rc = *rt;
    }
  }
  else {
    *rd = m.m_r;
    *rt = *rd + *rc;
  }
```
This code is used in section 261.

264.   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 264 ⟩ ≡
```
  if (m.num_measures ≡ 1) {
    *tt = 0.0;
    *td = 0.0;
  }
  else if (m.fraction_of_tc_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 261.

265.   Collect debugging info here

⟨ Debug info for estimate RT 265 ⟩ ≡
```
  if (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 261.

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

267.   *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 $[0, 1] \rightarrow (-\infty, +\infty)$.

⟨ Prototype for *a2acalc* 267 ⟩ ≡
  double *a2acalc*(**double** *a*)

This code is used in sections 253 and 268.

268.   ⟨ Definition for *a2acalc* 268 ⟩ ≡
  ⟨ Prototype for *a2acalc* 267 ⟩
  {
    if $(a \leq 0)$ **return** $-$`BIG_A_VALUE`;
    if $(a \geq 1)$ **return** `BIG_A_VALUE`;
    return $((2 * a - 1)/a/(1 - a))$;
  }

This code is used in section 252.

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

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

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* 269 ⟩ ≡
  double *acalc2a*(**double** *acalc*)

This code is used in sections 253 and 270.

270.   ⟨ Definition for *acalc2a* 270 ⟩ ≡
  ⟨ Prototype for *acalc2a* 269 ⟩
  {
    if $(acalc \equiv$ `BIG_A_VALUE`$)$ **return** 1.0;
    else if $(acalc \equiv -$`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 252.

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

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

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

⟨ Prototype for *g2gcalc* 271 ⟩ ≡
  double *g2gcalc*(**double** *g*)

This code is used in sections 253 and 272.

272.   ⟨ Definition for *g2gcalc* 272 ⟩ ≡
  ⟨ Prototype for *g2gcalc* 271 ⟩
  {
    if $(g \leq -0.99999)$ **return** $(-\text{HUGE\_VAL})$;
    if $(g \geq 0.99999)$ **return** (HUGE_VAL);
    return $(g/(1 - \mathit{fabs}(g)))$;
  }

This code is used in section 252.

273.   *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* 273 ⟩ ≡
  double *gcalc2g*(**double** *gcalc*)

This code is used in sections 253 and 274.

274.   ⟨ Definition for *gcalc2g* 274 ⟩ ≡
  ⟨ Prototype for *gcalc2g* 273 ⟩
  {
    if $(gcalc \equiv -\text{HUGE\_VAL})$ **return** -0.99999;
    if $(gcalc \equiv \text{HUGE\_VAL})$ **return** 0.99999;
    return $(gcalc/(1 + \mathit{fabs}(gcalc)))$;
  }

This code is used in section 252.

275.   *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* 275 ⟩ ≡
  double *b2bcalc*(**double** *b*)

This code is used in sections 253 and 276.

276.   ⟨Definition for *b2bcalc* 276⟩ ≡
  ⟨Prototype for *b2bcalc* 275⟩
  {
    if $(b \equiv$ HUGE_VAL) **return** HUGE_VAL;
    if $(b \leq 0)$ **return** 0.0;
    return $(log(b))$;
  }

This code is used in section 252.

277.   *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* 277⟩ ≡
  double *bcalc2b*(**double** *bcalc*)

This code is used in sections 253 and 278.

278.   ⟨Definition for *bcalc2b* 278⟩ ≡
  ⟨Prototype for *bcalc2b* 277⟩
  {
    if $(bcalc \equiv$ 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 252.

279.   *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* 279⟩ ≡
  void *twoprime*(**double** a, **double** b, **double** g, **double** $*ap$, **double** $*bp$)

This code is used in sections 253 and 280.

280.   ⟨Definition for *twoprime* 280⟩ ≡
  ⟨Prototype for *twoprime* 279⟩
  {
    if $(a \equiv 1 \wedge g \equiv 1)$ $*ap = 0.0$;
    else $*ap = (1 - g) * a/(1 - a * g)$;
    if $(b \equiv$ HUGE_VAL) $*bp =$ HUGE_VAL;
    else $*bp = (1 - a * g) * b$;
  }

This code is used in section 252.

281.   *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* 281 ⟩ ≡
  void *twounprime*(**double** *ap*, **double** *bp*, **double** *g*, **double** ∗*a*, **double** ∗*b*)

This code is used in sections 253 and 282.

282.   ⟨ Definition for *twounprime* 282 ⟩ ≡
  ⟨ Prototype for *twounprime* 281 ⟩
  {
    ∗*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 252.

283.   *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* 283 ⟩ ≡
  void *abgg2ab*(**double** *a1*, **double** *b1*, **double** *g1*, **double** *g2*, **double** ∗*a2*, **double** ∗*b2*)

This code is used in sections 253 and 284.

284.   ⟨ Definition for *abgg2ab* 284 ⟩ ≡
  ⟨ Prototype for *abgg2ab* 283 ⟩
  {
    double *a*, *b*;
    twoprime(*a1*, *b1*, *g1*, &*a*, &*b*);
    twounprime(*a*, *b*, *g2*, *a2*, *b2*);
  }

This code is used in section 252.

285.   *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* 285 ⟩ ≡
  void *abgb2ag*(**double** *a1*, **double** *b1*, **double** *b2*, **double** ∗*a2*, **double** ∗*g2*)

This code is used in sections 253 and 286.

286.   ⟨ Definition for $abgb2ag$ 286 ⟩ ≡
  ⟨ Prototype for $abgb2ag$ 285 ⟩
  {
    if $(b1 \equiv 0 \vee b2 \equiv 0)$ {
      $*a2 = a1$;
      $*g2 = 0$;
    }
    if $(b2 < b1)$ $b2 = b1$;
    if $(a1 \equiv 0)$ $*a2 = 0.0$;
    else {
      if $(a1 \equiv 1)$ $*a2 = 1.0$;
      else {
        if $(b1 \equiv 0 \vee b2 \equiv$ HUGE_VAL$)$ $*a2 = a1$;
        else $*a2 = 1 + b1/b2 * (a1 - 1)$;
      }
    }
    if $(*a2 \equiv 0 \vee b2 \equiv 0 \vee b2 \equiv$ HUGE_VAL$)$ $*g2 = 0.5$;
    else $*g2 = (1 - b1/b2)/(*a2)$;
  }

This code is used in section 252.

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

⟨ Prototype for *slow_guess* 287 ⟩ ≡
   void *slow_guess*(**struct measure_type** *m*, **struct invert_type** *∗r*, **double** *∗a*, **double** *∗b*, **double** *∗g*)
This code is used in section 288.

288.   ⟨ Definition for *slow_guess* 288 ⟩ ≡
   ⟨ Prototype for *slow_guess* 287 ⟩
   {
      double *fmin* = 10.0;
      double *fval*;
      double *∗x*;
      x = *dvector*(1, 2);
      switch (*r→search*) {
      case FIND_A: ⟨ Slow guess for *a* alone 289 ⟩
         break;
      case FIND_B: ⟨ Slow guess for *b* alone 290 ⟩
         break;
      case FIND_AB: **case** FIND_AG: ⟨ Slow guess for *a* and *b* or *a* and *g* 291 ⟩
         break;
      }
      ∗a = *r→slab.a*;
      ∗b = *r→slab.b*;
      ∗g = *r→slab.g*;
      free_dvector(*x*, 1, 2);
   }

289.   ⟨ Slow guess for *a* alone 289 ⟩ ≡
   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 288.

290.   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 290 ⟩ ≡
```
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 288.

291.   ⟨ Slow guess for a and b or a and g 291 ⟩ ≡
```
{
   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 288.

292.   ⟨ Prototype for *quick_guess* 292 ⟩ ≡
  void *quick_guess*(**struct measure_type** m, **struct invert_type** r, **double** $*a$, **double** $*b$, **double** $*g$)
This code is used in sections 253 and 293.

293.   ⟨ Definition for *quick_guess* 293 ⟩ ≡
  ⟨ Prototype for *quick_guess* 292 ⟩
```
{
   double UR1, UT1, rd, td, tc, rc, bprime, aprime, alpha, beta, logr;
   Estimate_RT(m, r, &UR1, &UT1, &rd, &rc, &td, &tc);
   ⟨ Estimate aprime 294 ⟩
   switch (m.num_measures) {
   case 1: ⟨ Guess when only reflection is known 296 ⟩
      break;
   case 2: ⟨ Guess when reflection and transmission are known 297 ⟩
      break;
   case 3: ⟨ Guess when all three measurements are known 298 ⟩
      break;
   }
   ⟨ Clean up guesses 303 ⟩
}
```
This code is used in section 252.

294.   ⟨ Estimate *aprime* 294 ⟩ ≡
  if (UT1 ≡ 1) *aprime* = 1.0;
  else if $(rd/(1 - \texttt{UT1}) \geq 0.1)$ {
      double $tmp = (1 - rd - \texttt{UT1})/(1 - \texttt{UT1})$;
      $aprime = 1 - 4.0/9.0 * tmp * tmp$;
  }
  else if $(rd < 0.05 \wedge \texttt{UT1} < 0.4)$ $aprime = 1 - (1 - 10 * rd) * (1 - 10 * rd)$;
  else if $(rd < 0.1 \wedge \texttt{UT1} < 0.4)$ $aprime = 0.5 + (rd - 0.05) * 4$;
  else {
      double $tmp = (1 - 4 * rd - \texttt{UT1})/(1 - \texttt{UT1})$;
      $aprime = 1 - tmp * tmp$;
  }
This code is used in section 293.

295.   ⟨ Estimate *bprime* 295 ⟩ ≡
  if $(rd < 0.01)$ {
    $bprime = What\_Is\_B(r.slab, \texttt{UT1})$;
    $fprintf(stderr, \texttt{"low␣rd<0.01!␣ut1=\%f␣aprime=\%f␣bprime=\%f\textbackslash n"}, \texttt{UT1}, aprime, bprime)$;
  }
  else if (UT1 ≤ 0) $bprime = \texttt{HUGE\_VAL}$;
  else if (UT1 > 0.1) $bprime = 2 * exp(5 * (rd - \texttt{UT1}) * log(2.0))$;
  else {
    $alpha = 1/log(0.05/1.0)$;
    $beta = log(1.0)/log(0.05/1.0)$;
    $logr = log(\texttt{UR1})$;
    $bprime = log(\texttt{UT1}) - beta * log(0.05) + beta * logr$;
    $bprime \mathrel{/}= alpha * log(0.05) - alpha * logr - 1$;
  }
This code is used in sections 297, 301, and 302.

296.

⟨ Guess when only reflection is known 296 ⟩ ≡
  $*g = r.default\_g$;
  $*a = aprime/(1 - *g + aprime * (*g))$;
  $*b = \texttt{HUGE\_VAL}$;
This code is used in section 293.

297.   ⟨ Guess when reflection and transmission are known 297 ⟩ ≡
  ⟨ Estimate *bprime* 295 ⟩
  $*g = r.default\_g$;
  $*a = aprime/(1 - *g + aprime * *g)$;
  $*b = bprime/(1 - *a * *g)$;
This code is used in section 293.

298.   ⟨Guess when all three measurements are known 298⟩ ≡
  switch (*r.search*) {
  case `FIND_A`: ⟨Guess when finding albedo 299⟩
    break;
  case `FIND_B`: ⟨Guess when finding optical depth 300⟩
    break;
  case `FIND_AB`: ⟨Guess when finding the albedo and optical depth 301⟩
    break;
  case `FIND_AG`: ⟨Guess when finding anisotropy and albedo 302⟩
    break;
  }

This code is used in section 293.

299.

⟨Guess when finding albedo 299⟩ ≡
  $*g = r.default\_g$;
  $*a = aprime/(1 - *g + aprime * *g)$;
  $*b = What\_Is\_B(r.slab, m.m\_u)$;

This code is used in section 298.

300.

⟨Guess when finding optical depth 300⟩ ≡
  $*g = r.default\_g$;
  $*a = 0.0$;
  $*b = What\_Is\_B(r.slab, m.m\_u)$;

This code is used in section 298.

301.

⟨Guess when finding the albedo and optical depth 301⟩ ≡
  $*g = r.default\_g$;
  if $(*g \equiv 1)$ $*a = 0.0$;
  else $*a = aprime/(1 - *g + aprime * *g)$;
  ⟨Estimate *bprime* 295⟩
  if $(bprime \equiv$ `HUGE_VAL` $\vee *a * *g \equiv 1)$ $*b =$ `HUGE_VAL`;
  else $*b = bprime/(1 - *a * *g)$;

This code is used in section 298.

302.

⟨Guess when finding anisotropy and albedo 302⟩ ≡
  $*b = What\_Is\_B(r.slab, m.m\_u)$;
  if $(*b \equiv$ `HUGE_VAL` $\vee *b \equiv 0)$ {
    $*a = aprime$;
    $*g = r.default\_g$;
  }
  else {
    ⟨Estimate *bprime* 295⟩
    $*a = 1 + bprime * (aprime - 1)/(*b)$;
    if $(*a < 0.1)$ $*g = 0.0$;
    else $*g = (1 - bprime/(*b))/(*a)$;
  }

This code is used in section 298.

303.

$\langle$ Clean up guesses $303 \rangle \equiv$

  if $(*a < 0)$ $*a = 0.0;$

  if $(*g < 0)$ $*g = 0.0;$

  else if $(*g \geq 1)$ $*g = 0.5;$

This code is used in section 293.

304.   Some debugging stuff.

305.   ⟨ Prototype for *Set_Debugging* 305 ⟩ ≡
  void *Set_Debugging*(**unsigned long** *debug_level*)

This code is used in sections 253 and 306.

306.

⟨ Definition for *Set_Debugging* 306 ⟩ ≡
  ⟨ Prototype for *Set_Debugging* 305 ⟩
  {
    g_util_debugging = *debug_level*;
  }

This code is used in section 252.

307.

⟨ Prototype for *Debug* 307 ⟩ ≡
  int *Debug*(**unsigned long** *mask*)

This code is used in sections 253 and 308.

308.

⟨ Definition for *Debug* 308 ⟩ ≡
  ⟨ Prototype for *Debug* 307 ⟩
  {
    if (*g_util_debugging* & *mask*) **return** 1;
    else return 0;
  }

This code is used in section 252.

309.

⟨ Prototype for *Print_Invert_Type* 309 ⟩ ≡
  void *Print_Invert_Type*(**struct invert_type** *r*)

This code is used in sections 253 and 310.

310.

⟨ Definition for *Print_Invert_Type* 310 ⟩ ≡
  ⟨ Prototype for *Print_Invert_Type* 309 ⟩
  {
    fprintf(*stderr*, "\n");
    fprintf(*stderr*, "default␣␣a=%10.5f␣␣␣b=%10.5f␣␣␣␣␣g=%10.5f\n", *r.default_a*, *r.default_b*, *r.default_g*);
    fprintf(*stderr*, "slab␣␣␣␣␣␣a=%10.5f␣␣␣b=%10.5f␣␣␣␣␣g=%10.5f\n", *r.slab.a*, *r.slab.b*, *r.slab.g*);
    fprintf(*stderr*, "n␣␣␣␣␣␣␣top=%10.5f␣mid=%10.5f␣␣bot=%10.5f\n", *r.slab.n_top_slide*, *r.slab.n_slab*,
        r.slab.n_bottom_slide);
    fprintf(*stderr*, "thick␣␣top=%10.5f␣cos=%10.5f␣␣bot=%10.5f\n", *r.slab.b_top_slide*, *r.slab.cos_angle*,
        r.slab.b_bottom_slide);
    fprintf(*stderr*, "search␣=␣%d␣quadrature␣points␣=␣%d\n", *r.search*, *r.method.quad_pts*);
  }

This code is used in section 252.

311.

⟨ Prototype for *Print_Measure_Type* 311 ⟩ ≡
  void *Print_Measure_Type*(**struct measure_type** *m*)

This code is used in sections 253 and 312.

312.

⟨ Definition for *Print_Measure_Type* 312 ⟩ ≡
  ⟨ Prototype for *Print_Measure_Type* 311 ⟩
  {
    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_rc_in_mr ∗ 100);
    fprintf(*stderr*, "#␣␣␣␣␣Fraction␣unscattered␣light␣in␣M_T␣=␣%7.1f␣%%\n",
        m.fraction_of_tc_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.ae_r*));
    fprintf(*stderr*, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣detector␣port␣diameter␣=␣%7.1f␣mm\n",
        2 ∗ *m.d_sphere_r* ∗ *sqrt*(*m.ad_r*));
    fprintf(*stderr*, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣wall␣reflectance␣=␣%7.1f␣%%\n", *m.rw_r* ∗ 100);
    fprintf(*stderr*, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣standard␣reflectance␣=␣%7.1f␣%%\n", *m.rstd_r* ∗ 100);
    fprintf(*stderr*, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣detector␣reflectance␣=␣%7.1f␣%%\n", *m.rd_r* ∗ 100);
    fprintf(*stderr*, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣spheres␣=␣%7d\n", *m.num_spheres*);
    fprintf(*stderr*, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣measures␣=␣%7d\n", *m.num_measures*);
    fprintf(*stderr*, "#␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣method␣=␣%7d\n", *m.method*);
    fprintf(*stderr*, "area_r␣as=%10.5f␣␣ad=%10.5f␣␣␣␣ae=%10.5f␣␣aw=%10.5f\n", *m.as_r*, *m.ad_r*,
        m.ae_r, *m.aw_r*);
    fprintf(*stderr*, "refls␣␣rd=%10.5f␣␣rw=%10.5f␣␣rstd=%10.5f␣␣␣f=%10.5f\n", *m.rd_r*, *m.rw_r*,
        m.rstd_r, *m.f_r*);
    fprintf(*stderr*, "area_t␣as=%10.5f␣␣ad=%10.5f␣␣␣␣␣ae=%10.5f␣␣aw=%10.5f\n", *m.as_t*, *m.ad_t*,
        m.ae_t, *m.aw_t*);
    fprintf(*stderr*, "refls␣␣rd=%10.5f␣␣rw=%10.5f␣␣rstd=%10.5f␣␣␣f=%10.5f\n", *m.rd_t*, *m.rw_t*,
        m.rstd_t, *m.f_t*);
    fprintf(*stderr*, "lost␣␣ur1=%10.5f␣ut1=%10.5f␣␣␣uru=%10.5f␣␣utu=%10.5f\n", *m.ur1_lost*,
        m.ut1_lost, *m.utu_lost*, *m.utu_lost*);
  }

This code is used in section 252.

313. 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.

⟨ Definition for *slow_guess* 288 ⟩
⟨ Definition for *twoprime* 280 ⟩   Used in section 252.
⟨ Definition for *twounprime* 282 ⟩   Used in section 252.
⟨ Estimate the backscattered reflection 263 ⟩   Used in section 261.
⟨ Estimate the scattered transmission 264 ⟩   Used in section 261.
⟨ Estimate *aprime* 294 ⟩   Used in section 293.
⟨ Estimate *bprime* 295 ⟩   Used in sections 297, 301, and 302.
⟨ Evaluate the BaG simplex at the nodes 245 ⟩   Used in section 243.
⟨ Evaluate the BsG simplex at the nodes 250 ⟩   Used in section 248.
⟨ Evaluate the bg simplex at the nodes 240 ⟩   Used in section 237.
⟨ Evaluate the a and b simplex at the nodes 216 ⟩   Used in section 212.
⟨ Evaluate the a and g simplex at the nodes 234 ⟩   Used in section 232.
⟨ Exit with bad input data 46 ⟩   Used in section 45.
⟨ Fill r with reasonable values 63, 64, 65, 66 ⟩   Used in section 62.
⟨ Find the optical properties 47 ⟩   Used in section 45.
⟨ Find thickness when multiple internal reflections are present 259 ⟩   Used in section 255.
⟨ Free simplex data structures 219 ⟩   Used in sections 212, 232, 237, 243, and 248.
⟨ Generate next albedo using j 156 ⟩   Used in sections 154 and 158.
⟨ Get the initial a, b, and g 214 ⟩   Used in sections 212, 232, 237, 243, and 248.
⟨ Guess when all three measurements are known 298 ⟩   Used in section 293.
⟨ Guess when finding albedo 299 ⟩   Used in section 298.
⟨ Guess when finding anisotropy and albedo 302 ⟩   Used in section 298.
⟨ Guess when finding optical depth 300 ⟩   Used in section 298.
⟨ Guess when finding the albedo and optical depth 301 ⟩   Used in section 298.
⟨ Guess when only reflection is known 296 ⟩   Used in section 293.
⟨ Guess when reflection and transmission are known 297 ⟩   Used in section 293.
⟨ Handle options 5 ⟩   Used in section 2.
⟨ Improve result using Monte Carlo 15 ⟩   Used in section 11.
⟨ Include files for *main* 3, 31 ⟩   Used in sections 2 and 30.
⟨ Initialize the nodes of the a and b simplex 215 ⟩   Used in section 212.
⟨ Initialize the nodes of the a and g simplex 233 ⟩   Used in section 232.
⟨ Initialize the nodes of the ba and g simplex 244 ⟩   Used in section 243.
⟨ Initialize the nodes of the bs and g simplex 249 ⟩   Used in section 248.
⟨ Initialize the nodes of the b and g simplex 239 ⟩   Used in section 237.
⟨ Iteratively solve for b 230 ⟩   Used in section 229.
⟨ Local Variables for Calculation 12 ⟩   Used in section 11.
⟨ Nonworking code 155 ⟩
⟨ One parameter deviation 181 ⟩   Used in section 180.
⟨ One parameter search 59 ⟩   Used in section 58.
⟨ Print basic sphere and MC effects 48 ⟩   Used in section 45.
⟨ Print diagnostics 183 ⟩   Used in section 173.
⟨ Print results function 23 ⟩   Used in section 2.
⟨ Process the header 33 ⟩   Used in section 30.
⟨ Prototype for *Allocate_Grid* 136 ⟩   Used in sections 116 and 137.
⟨ Prototype for *Calculate_Distance_With_Corrections* 172 ⟩   Used in sections 116 and 173.
⟨ Prototype for *Calculate_Distance* 168 ⟩   Used in sections 116 and 169.
⟨ Prototype for *Calculate_Grid_Distance* 170 ⟩   Used in sections 116 and 171.
⟨ Prototype for *Calculate_MR_MT* 78 ⟩   Used in sections 42 and 79.
⟨ Prototype for *Calculate_Minimum_MR* 80 ⟩   Used in sections 42 and 81.
⟨ Prototype for *Debug* 307 ⟩   Used in sections 253 and 308.
⟨ Prototype for *Estimate_RT* 260 ⟩   Used in sections 253 and 261.
⟨ Prototype for *Fill_AB_Grid* 153 ⟩   Used in sections 115 and 154.

⟨ Prototype for *Fill_AG_Grid* 157 ⟩   Used in sections 115 and 158.
⟨ Prototype for *Fill_BG_Grid* 160 ⟩   Used in sections 116 and 161.
⟨ Prototype for *Fill_BaG_Grid* 162 ⟩   Used in sections 116 and 163.
⟨ Prototype for *Fill_BsG_Grid* 164 ⟩   Used in sections 116 and 165.
⟨ Prototype for *Fill_Grid* 166 ⟩   Used in sections 116 and 167.
⟨ Prototype for *Find_AB_fn* 186 ⟩   Used in sections 116 and 187.
⟨ Prototype for *Find_AG_fn* 184 ⟩   Used in sections 116 and 185.
⟨ Prototype for *Find_A_fn* 192 ⟩   Used in sections 116 and 193.
⟨ Prototype for *Find_BG_fn* 198 ⟩   Used in sections 116 and 199.
⟨ Prototype for *Find_B_fn* 194 ⟩   Used in sections 116 and 195.
⟨ Prototype for *Find_BaG_fn* 200 ⟩   Used in sections 116 and 201.
⟨ Prototype for *Find_Ba_fn* 188 ⟩   Used in sections 116 and 189.
⟨ Prototype for *Find_BsG_fn* 202 ⟩   Used in sections 116 and 203.
⟨ Prototype for *Find_Bs_fn* 190 ⟩   Used in sections 116 and 191.
⟨ Prototype for *Find_G_fn* 196 ⟩   Used in sections 116 and 197.
⟨ Prototype for *Gain_11* 121 ⟩   Used in sections 116 and 122.
⟨ Prototype for *Gain_22* 123 ⟩   Used in sections 116 and 124.
⟨ Prototype for *Gain* 119 ⟩   Used in sections 116 and 120.
⟨ Prototype for *Get_Calc_State* 132 ⟩   Used in sections 116 and 133.
⟨ Prototype for *Grid_ABG* 138 ⟩   Used in sections 116 and 139.
⟨ Prototype for *Initialize_Measure* 69 ⟩   Used in sections 42 and 70.
⟨ Prototype for *Initialize_Result* 61 ⟩   Used in sections 42 and 62.
⟨ Prototype for *Inverse_RT* 44 ⟩   Used in sections 42 and 45.
⟨ Prototype for *Max_Light_Loss* 206 ⟩   Used in sections 116 and 207.
⟨ Prototype for *MinMax_MR_MT* 82 ⟩   Used in sections 42 and 83.
⟨ Prototype for *Near_Grid_Points* 148 ⟩   Used in sections 116 and 149.
⟨ Prototype for *Print_Invert_Type* 309 ⟩   Used in sections 253 and 310.
⟨ Prototype for *Print_Measure_Type* 311 ⟩   Used in sections 253 and 312.
⟨ Prototype for *RT_Flip* 150 ⟩   Used in sections 116 and 151.
⟨ Prototype for *Read_Data_Line* 99 ⟩   Used in sections 93 and 100.
⟨ Prototype for *Read_Header* 95 ⟩   Used in sections 93 and 96.
⟨ Prototype for *Same_Calc_State* 134 ⟩   Used in sections 116 and 135.
⟨ Prototype for *Set_Calc_State* 130 ⟩   Used in sections 116 and 131.
⟨ Prototype for *Set_Debugging* 305 ⟩   Used in sections 253 and 306.
⟨ Prototype for *Spheres_Inverse_RT2* 84 ⟩   Used in sections 42, 43, and 85.
⟨ Prototype for *Spheres_Inverse_RT* 71 ⟩   Used in sections 43 and 72.
⟨ Prototype for *Two_Sphere_R* 125 ⟩   Used in sections 116 and 126.
⟨ Prototype for *Two_Sphere_T* 127 ⟩   Used in sections 116 and 128.
⟨ Prototype for *U_Find_AB* 211 ⟩   Used in sections 210 and 212.
⟨ Prototype for *U_Find_AG* 231 ⟩   Used in sections 210 and 232.
⟨ Prototype for *U_Find_A* 224 ⟩   Used in sections 210 and 225.
⟨ Prototype for *U_Find_BG* 236 ⟩   Used in sections 210 and 237.
⟨ Prototype for *U_Find_BaG* 242 ⟩   Used in sections 210 and 243.
⟨ Prototype for *U_Find_Ba* 222 ⟩   Used in sections 210 and 223.
⟨ Prototype for *U_Find_BsG* 247 ⟩   Used in sections 210 and 248.
⟨ Prototype for *U_Find_Bs* 220 ⟩   Used in sections 210 and 221.
⟨ Prototype for *U_Find_B* 228 ⟩   Used in sections 210 and 229.
⟨ Prototype for *U_Find_G* 226 ⟩   Used in sections 210 and 227.
⟨ Prototype for *Valid_Grid* 140 ⟩   Used in sections 116 and 141.
⟨ Prototype for *What_Is_B* 254 ⟩   Used in sections 253 and 255.
⟨ Prototype for *Write_Header* 107 ⟩   Used in sections 93 and 108.
⟨ Prototype for *a2acalc* 267 ⟩   Used in sections 253 and 268.

⟨ Prototype for *abg_distance* 146 ⟩   Used in sections 116 and 147.
⟨ Prototype for *abgb2ag* 285 ⟩   Used in sections 253 and 286.
⟨ Prototype for *abgg2ab* 283 ⟩   Used in sections 253 and 284.
⟨ Prototype for *acalc2a* 269 ⟩   Used in sections 253 and 270.
⟨ Prototype for *b2bcalc* 275 ⟩   Used in sections 253 and 276.
⟨ Prototype for *bcalc2b* 277 ⟩   Used in sections 253 and 278.
⟨ Prototype for *check_magic* 105 ⟩   Used in section 106.
⟨ Prototype for *determine_search* 57 ⟩   Used in sections 42 and 58.
⟨ Prototype for *ez_Inverse_RT* 67 ⟩   Used in sections 42, 43, and 68.
⟨ Prototype for *g2gcalc* 271 ⟩   Used in sections 253 and 272.
⟨ Prototype for *gcalc2g* 273 ⟩   Used in sections 253 and 274.
⟨ Prototype for *maxloss* 204 ⟩   Used in sections 116 and 205.
⟨ Prototype for *measure_OK* 50 ⟩   Used in sections 42 and 51.
⟨ Prototype for *quick_guess* 292 ⟩   Used in sections 253 and 293.
⟨ Prototype for *read_number* 103 ⟩   Used in section 104.
⟨ Prototype for *skip_white* 101 ⟩   Used in section 102.
⟨ Prototype for *slow_guess* 287 ⟩   Used in section 288.
⟨ Prototype for *twoprime* 279 ⟩   Used in sections 253 and 280.
⟨ Prototype for *twounprime* 281 ⟩   Used in sections 253 and 282.
⟨ Put final values in result 218 ⟩   Used in sections 212, 221, 223, 225, 227, 229, 232, 237, 243, and 248.
⟨ Read coefficients for reflection sphere 97 ⟩   Used in section 96.
⟨ Read coefficients for transmission sphere 98 ⟩   Used in section 96.
⟨ Slow guess for *a* alone 289 ⟩   Used in section 288.
⟨ Slow guess for *a* and *b* or *a* and *g* 291 ⟩   Used in section 288.
⟨ Slow guess for *b* alone 290 ⟩   Used in section 288.
⟨ Solve if multiple internal reflections are not present 258 ⟩   Used in section 255.
⟨ Structs to export from IAD Types 38, 39, 40 ⟩   Used in section 35.
⟨ Tests for invalid grid 142, 143, 144, 145 ⟩   Used in section 141.
⟨ Two parameter deviation 182 ⟩   Used in section 180.
⟨ Two parameter search 60 ⟩   Used in section 58.
⟨ Unused diffusion fragment 208 ⟩
⟨ Write Header 14 ⟩   Used in section 11.
⟨ Write first sphere info 112 ⟩   Used in section 108.
⟨ Write general sphere info 111 ⟩   Used in section 108.
⟨ Write irradiation info 110 ⟩   Used in section 108.
⟨ Write measure and inversion info 114 ⟩   Used in section 108.
⟨ Write second sphere info 113 ⟩   Used in section 108.
⟨ Write slab info 109 ⟩   Used in section 108.
⟨ Zero *GG* 159 ⟩   Used in sections 154, 158, 161, 163, and 165.
⟨ calculate coefficients function 20, 21 ⟩   Used in section 2.
⟨ handle analysis 74 ⟩   Used in section 72.
⟨ handle measurement 75 ⟩   Used in section 72.
⟨ handle reflection sphere 76 ⟩   Used in section 72.
⟨ handle setup 73 ⟩   Used in section 72.
⟨ handle transmission sphere 77 ⟩   Used in section 72.
⟨ handle2 analysis 90 ⟩   Used in section 85.
⟨ handle2 illumination 87 ⟩   Used in section 85.
⟨ handle2 measurement 91 ⟩   Used in section 85.
⟨ handle2 reflection sphere 88 ⟩   Used in section 85.
⟨ handle2 sample 86 ⟩   Used in section 85.
⟨ handle2 transmission sphere 89 ⟩   Used in section 85.
⟨ `iad_calc.c` 115 ⟩