Light Scattering by
Individual and Groups
of Spheroidal Particles

Diploma paper
by
Peter Alsholm

Lund Reports on Atomic Physics, LRAP-200
Lund, August 1996
Abstract

In order to simulate the light propagation in illuminated samples containing suspensions of spheroidal particles, taking into consideration size, shape, concentration and different orientations, a Monte Carlo computer program was developed. The program, based on an old Monte Carlo program, was able to treat samples of finite extension laterally, as well as in depth. The computations of the angular scattering of each particle, were made by separate programs based on the T-matrix theory. Modifications of existing programs for this purpose were performed, enabling scattering calculations for particles with larger size parameters. Simulations were made by using a sample with properties similar to human blood, i.e. a suspension of erythrocytes, approximated by oblate spheroids, as scattering particles. These results were then finally compared with measurements and results obtained by the original Monte Carlo program.
1. Introduction

The knowledge of light interacting with biological tissue is important when developing new diagnostic methods and medical treatments utilizing lasers. The more these methods are refined, the more detailed studies of the propagation of light in tissue are required. In diagnosing diseases such as cancer and arteriosclerosis, using lasers, the fluorescence, inelastic or elastic scattered light, transmitted through or reflected by the tissue is analyzed. Distinctions in the obtained spectra originate from deviations in optical properties, which makes it possible to differentiate between healthy and malignant tissue. When laser light is used to treat medical disorders, e.g. in obliteration of port wine stains or cancer cell destruction using photodynamic therapy (PDT), the energy of the laser light is transformed into heat energy or induces chemical reactions. In order to optimize the results of these treatments, the understanding of light propagation properties in tissue is essential.

When light propagates, two processes - absorption and scattering - occur to various extents due to the medium. A common way of describing the optical properties of the medium is by the absorption coefficient, $\mu_a$, the scattering coefficient, $\mu_s$, and the anisotropy coefficient, $g$. The absorption and scattering coefficients describe the probability per unit path length of a photon being absorbed or scattered, respectively. The $g$-factor is defined by the mean value of the cosine of the scattering angle. These optical parameters can be used as input parameters for Monte Carlo simulations, to obtain the distribution of light transmitted through or reflected by the medium. In the Monte Carlo method, the paths of a large number of photons are simulated in a randomized manner using the known geometry and optical coefficients.

In a previous work, the optical properties of human blood were measured in conjunction with slow heating. Sudden changes in $g$-factor and scattering coefficient were observed simultaneously as the red blood cells were transformed from spheroidal into spherical shape. Other authors have studied the light scattering by one erythrocyte and indicated that the cell shape influences the propagation of light. The scattering process in the Monte Carlo model has also been criticized, when simulating the light distribution in human blood.

The purpose of this work is to study the angular scattering by one spheroidal particle, the influence of size, shape, concentration and orientation of spheroidal particles distributed in a homogenous non-absorbing medium and to develop a computer program taking these characteristics into consideration, when evaluating the propagation of light. The study includes in particular particles with the size of erythrocytes. For calculations of the angular scattering by one spheroidal particle, the T-matrix method is an effective tool, described and used in this work. Nevertheless, the idea is that the developed program will be used not only for investigations of light propagating in human blood, but also in any tissue or media in general, using other shapes and sizes of the scattering particles. Interesting results of investigations of the optical and printing properties of paper, correlated to the structures of fibres and pores, have also given inspiration to this work and might be another application of this model.
2. Theory

2.1. Scattering from one particle

2.1.1. Electric time-harmonic fields

The time-dependent electric field of a uniform plane wave in space can be expressed as

\[ E(t,z) = \text{Re}(E(kz) e^{i\omega t}) \tag{2.1} \]

\( \text{Re}(\cdot) \) extracts the real part of a complex variable.

The complex field \( E(kz) \) is written as

\[ E(kz) = E_0 e^{i(kz - \phi)} \tag{2.2} \]

where \( z \) is the position in the direction of propagation \( e_z \), \( \phi \) is the phase at \( z = 0 \), \( n \) represents the complex index of refraction and \( \lambda_0 \) the free space wavelength. The notation \( e_z \) symbolizes the unity vector in the direction of the \( E \)-field \( (e_z \perp e_z) \).

The intensity (time-averaged power per unit area) is determined by

\[ I = \frac{1}{2} \left| \text{Re}(E \times H^*) \right| = \frac{1}{2} \left| E \right|^2 \text{Re}(\frac{1}{\eta^*}) \quad (^* \text{ stands for the complex conjugate}) \tag{2.3} \]

\( H \) represents the magnetic field and is proportional to \( E \). The two fields are perpendicular to each other and to the propagating direction. Thus, the intensity is proportional to \( \left| E \right|^2 \). The complex variable \( \eta \) symbolizes the impedance of the medium.

When a plane wave is scattered by a particle, a scattered wave is generated. Far from the particle, the time-independent scattered electric field can be expressed as

\[ E(kr) = E_0 \frac{e^{ikr}}{r} \tag{2.4} \]

and is hence termed the far-field. The amplitude of the near-field has a stronger \( r \)-dependence and decays faster.

2.1.2. The T-matrix method

Assuming a particle is illuminated by the incident electric field \( E_i \). This field then induces an internal field \( E_{in} \) and a scattered field denoted by \( E_s \). The incident field is a plane-wave in the \( k \)-direction:

\[ E_i(kr) = E_0 e^{ikr} e_z \quad ; \quad k = k e_k \tag{2.5} \]

This expression can be expanded as
$$E_i(kr) = E_0 \sum_\nu D_\nu [a_\nu M_\nu(kr) + b_\nu N_\nu(kr)]$$

(2.6)

where \( \nu = (\sigma, m, n) \) is the triple index for spherical harmonic functions (\( \sigma \) indicates whether the harmonic function is odd (o) or even (e), \( m \) represents the azimuthal mode and \( n \) the expansion-coefficient). \( M_\nu \) and \( N_\nu \) are Bessel-radial dependent vector spherical functions in the two polarization directions and \( D_\nu \) is a normalization constant. The incident field expansion coefficients \( a_\nu \) and \( b_\nu \) are obtained by evaluating different combinations of trigonometric functions and associated Legendre-functions \( P_n^m(\cos \theta) \). For example, \( a_{oemn} \) is determined by

$$a_{oemn} = 4 i^n e \cdot [-e_\theta \sin (m\phi) m \frac{P_n^m(\cos \theta)}{\sin \theta} - e_\phi \cos (m\phi) \frac{d}{d\theta} (P_n^m(\cos \theta))]$$

(2.7)

where \( \theta \) and \( \phi \) indicate the direction of \( k \) and \( e_\theta \) and \( e_\phi \) are unity vectors in these directions. The coefficients \( a_{oemn}, b_{oemn} \) and \( b_{eemn} \) are expressed in a similar way. Jackson\(^8\) has made a more detailed description about spherical wave-expansion of plane waves.

In the same way as the incident field, the internal and scattered field can be expanded:

$$E_{int}(mkr) = E_0 \sum_\mu D_\mu [c_\mu M_\mu(mkr) + d_\mu N_\mu(mkr)]$$

(2.8)

$$E_s(kr) = E_0 \sum_\nu D_\nu [f_\nu M_\nu(kr) + g_\nu N_\nu(kr)]$$

(2.9)

The vector spherical functions \( M \) and \( N \) have a Bessel-radial dependence in equation (2.8) and a Hankel-radial dependence in equation (2.9). The coefficients \( c_\mu, d_\mu, f_\nu \) and \( g_\nu \) are unknown and the refractive index of the particle relative the surrounding medium is denoted by \( m \).

Because of the linearity of Maxwell’s equations, the relation between the different expansion coefficients is linear. Thus, when the coefficients \( a_\nu \) and \( b_\nu \) are known as well as the shape and size of the particle, it is possible to determine \( c_\mu \) and \( d_\mu \):

$$A - matrix \begin{bmatrix} c_\mu \\ d_\mu \end{bmatrix} = -i \begin{bmatrix} a_\nu \\ b_\nu \end{bmatrix}$$

(2.10)

\( f_\nu \) and \( g_\nu \) are calculated in the same way:

$$\begin{bmatrix} f_\nu \\ g_\nu \end{bmatrix} = -i \begin{bmatrix} c_\mu \\ d_\mu \end{bmatrix}$$

(2.11)

Thus, this relation can be expressed as
\[
\begin{bmatrix}
    f_v \\
    g_v
\end{bmatrix} = - \begin{bmatrix}
    B - \text{matrix} \\
    A - \text{matrix}
\end{bmatrix}^{-1} \begin{bmatrix}
    a_v \\
    b_v
\end{bmatrix} = - \begin{bmatrix}
    \text{T - matrix}
\end{bmatrix} \begin{bmatrix}
    a_v \\
    b_v
\end{bmatrix}
\] \quad (2.12)

where T-matrix is a shorter name for the so called transition-matrix. Obviously, the T-matrix can be used to calculate \( f_v \) and \( g_v \) and thus the scattered field.

The elements of the A- and B-matrices consist of surface-integrals containing products of spherical functions. The integrals have to be numerically determined over the surface of the particle for every \( v = (\sigma, m, n) \) combination. If the particle is axi-symmetric simplifications can be made so that the surface integrals can be reduced to one-dimensional integrals containing combinations of Bessel-, Hankel-, associated Legendre- and trigonometric functions. As the elements are independent of the incident and scattered fields, it's enough to compute the T-matrix once for a specific scattering particle at a fixed wavelength. In the case of a sphere, the T-matrix will be diagonal and explicit expressions can be obtained for the diagonal elements. These expressions will be identical with the ones obtained in Mie-scattering.

When \( E_z \) is evaluated, the far-field \( F \) is defined by

\[
E_z(kr) = F(\theta, \phi) \frac{e^{ikr}}{r}
\] \quad (2.13)

in the limit \( kr \to \infty \).

\( F_\theta \) and \( F_\phi \) are the two components of \( F \). By replacing the Hankel-functions with asymptotic expressions (in the \( kr \to \infty \) limit), an explicit form of \( F \) can be obtained. For example, for \( e_\theta \) incident polarization, \( F_\theta \) is evaluated by

\[
F_\theta = \frac{i^{-(n+1)}}{k} \cos(m\phi) \left( \frac{P_n^m(\cos \theta)}{\sin \theta} \right) D_{nm} f_{\text{em}} + \\
+ \frac{i^m}{k} \cos(m\phi) \left[ n \cos \theta \left( \frac{P_n^m(\cos \theta)}{\sin \theta} \right) - (n+m) \left( \frac{P_{n+1}^m(\cos \theta)}{\sin \theta} \right) \right] D_{nm} g_{\text{em}}
\] \quad (2.14)

The differential scattering cross section, expressing the scattered intensity in a specific direction relative the intensity of the incident field, can then be determined:

\[
\sigma_d(\theta, \phi) = |F(\theta, \phi)/E_0|^2
\] \quad (2.15)

The total scattering cross section includes all scattering directions and is given by

\[
\sigma_s = \pi k^2 \sum_v D_v \left( |f_v|^2 + |g_v|^2 \right)
\] \quad (2.16)

By using the optical theorem, the total extinction cross section \( \sigma_e \), which describes how much the incident field is influenced by the particle, can be expressed as
\[ \sigma_e = 4\pi k^2 \text{Im}(e_e \cdot k F / E_0) \]  

(2.17)

\( F \) is evaluated in the direction straight forward and \( e_e \) is the polarization unity vector of the incident field. \( \text{Im}(\cdot) \) extracts the imaginary part.

Finally, the absorption cross section is defined by

\[ \sigma_a = \sigma_e - \sigma_s \]  

(2.18)

By dividing the cross sections by \( \pi a^2 \), where \( a \) is a characteristic dimension of the particle, normalized cross sections can be calculated.

Other authors\cite{8, 11-12} have described the T-matrix method more extensively and specific properties of the T-matrix are analyzed by Waterman\cite{13}.

2.2. Multiple scattering

2.2.1. The original Monte Carlo method

Light propagation in turbid media can be simulated using Monte Carlo methods\cite{2}. In those, photons or photon packages are sent on a random walk through the tissue. The phase and polarization of the photons are not taken into consideration in this method, due to their random behaviour after multiple scattering in the tissue. The wave phenomena of the photon are thus ignored and the photon is treated as a particle. The path of every photon package is simulated until it is emerged or absorbed. When a large amount of photon packages are used, good statistic results of the light distribution in the tissue can be obtained.

The computer code MCML, written by Wang and Jacques\cite{2}, is a program based on the Monte Carlo method, where the distribution of absorbed light in a medium, containing layers with different optical parameters and refractive indices, can be computed as well as the distributions of reflected and transmitted light.

When the photon package enters the media, it is assigned an initial weight \( W \). This weight will be reduced each time the photon package interacts with the media. The fraction \( \Delta W \) which is subtracted from the initial weight is given by

\[ \Delta W = \frac{\mu_a}{\mu_a + \mu_s} W \]  

(2.19)

The path length, \( S \), between the points of interaction is

\[ S = \frac{-\ln(1 - R)}{\mu_a + \mu_s} \]  

(2.20)

where \( R \) represents a random number between 0 and 1.
In every interaction point a new direction of the photon package is obtained. The azimuthal angle $\phi$ is calculated by

$$\phi = 2\pi R$$

(2.21)

where $R$ is assigned a new random value between 0 and 1. In order to determine the deflection angle $\theta$, the Henyey-Greenstein phase function is utilized. This function, often used in tissue optics, is in fact a probability distribution expressed as

$$p(\cos \theta) = \frac{1 - g^2}{4\pi(1 + g^2 - 2g \cos \theta)^{3/2}}$$

(2.22)

where the anisotropy coefficient $g$ is in the range between -1 and 1. Complete back and forward scattering are described by $g = -1$ and $g = 1$, respectively and $g = 0$ represents isotropic scattering. The deflection angle $\theta$, assumed to be distributed according to the Henyey-Greenstein distribution can then be determined by using a new random value $R$, between 0 and 1:

$$\cos \theta = \frac{1 + g^2 - \left(\frac{1 - g^2}{1 - g + 2gR}\right)^2}{2g}$$

(2.23)

A special routine is used in the program to determine when the photon package is totally absorbed. To ensure the conservation of energy, a roulette routine is used if the photon weight falls below a particular threshold value. The photon package survives at a certain probability, $m$, and is then assigned the value of $m$ times the old weight. It is otherwise terminated. The MCML-program finally also takes care of the processes occurring when a photon package hits a boundary between two layers with different optical properties, using Snell’s law and Fresnel’s formulas. The Monte Carlo method is discussed more rigorously by Keijzer.
3. Material and methods

3.1. The T-matrix computer programs

A series of computer programs, written in FORTRAN, which can be utilized for determination of the scattering and absorption characteristics of the finite-thickness slab, infinite circular cylinder, general axi-symmetric particle and sphere, is developed by Barber and Hill. In this work two of their programs, T1 and T3, were modified to be able to compute the angular scattering of oblate spheroids with larger size parameters as for example the erythrocyte. A short description of the originally programs as well as the necessary modifications are described below.

3.1.1. The original computer program T1

Input parameters to T1 are:

\begin{align*}
2a & : \quad \text{The diameter of the spheroidal particle along the symmetry axis.} \\
2b & : \quad \text{The diameter of the spheroidal particle perpendicular to the symmetry axis.} \\
x = ka = 2\pi na / \lambda_0 & : \quad \text{The size parameter.} \\
aovrb = a/b & : \quad \text{Axial ratio.} \\
Mr, Mi & : \quad \text{The real and imaginary part of the relative index of refraction.} \\
nrank & : \quad \text{Number of expansion coefficients (number of terms in eq. 2.6).} \\
ntheta & : \quad \text{Number of integration points in the one-dimensional integrals in the calculation of the elements of the A- and B-matrices.} \\
\text{case 0} & : \quad \text{The program T1 checks the convergence over ntheta.} \\
\text{case 1} & : \quad \text{The program T1 checks the convergence over nrank.} \\
\text{case 2} & : \quad \text{The program T1 checks the convergence over m. (m is the number of azimuthal modes).}
\end{align*}

In order to evaluate a correct T-matrix, a sufficient amount of expansion coefficients, integration points and azimuthal modes are needed depending on the size and shape of the scattering particle. Large particles and oblate- and prolate-shaped particles with a considerable axial ratio require larger values of nrank and ntheta, compared to small and spherical-shaped particles. The possibility of obtaining a converged solution depends on the computer capacity and the number of precision variables.

The program T1 first checks the number of expansion coefficients, integration points and azimuthal modes needed to obtain a converged solution for a specific scattering particle in a specific surrounding medium and then it determines the T-matrix. Figure 3.1 shows a cross sectional picture of one possible scattering particle. The particle must be axi-symmetric and fulfil the equation

\begin{align}
  r(\theta) &= \frac{a}{\sqrt{\cos^2\theta + \left(\frac{a}{b}\right)^2 \sin^2\theta}} \\
  &= \frac{a}{\sqrt{\cos^2\theta + \left(\frac{a}{b}\right)^2 \sin^2\theta}} \\
  &\text{ (3.1)}
\end{align}
Figure 3.1. A cross sectional picture of a scattering oblate-shaped spheroid. The particle is rotational symmetric around its semiminor axis. The diameter of the particle along this axis equals 2a.

When determining convergence over \( n_{\text{rank}} \) and \( n_{\text{theta}} \) for a specific particle, the incident field is assumed along the symmetry axis and the azimuthal mode \( m \) can be set to one, as all the other azimuthal modes then are zero. Convergence is verified when the angular scattering outputs for \( n_{\text{rank}} \) and \( n_{\text{rank}}-1 \) or \( n_{\text{theta}} \) and \( n_{\text{theta}}+4 \), respectively, correspond with a certain accuracy.

The obtained values of \( n_{\text{rank}} \) and \( n_{\text{theta}} \) are then used when evaluating convergence over \( m \). In this case (2), the incident field is assumed along a direction 45 degrees from the symmetry axis (the most complicated case). The program calculates and compares the angular scattering for increasing \( m \)-values until convergence is obtained or \( m > n_{\text{rank}} \) (which means that no converged solution exists). By the use of case 2, the T-matrix is determined, but convergence over \( n_{\text{rank}} \) and \( n_{\text{theta}} \) must first have been obtained to be sure that the determination is correct. The number of rows (or columns) in this matrix equals twice \( n_{\text{rank}} \). When the T-matrix finally is stored it can be used to compute various cross sections and resulting scattered fields in different directions.

3.1.2. Modifications of program T1

Using a larger T-matrix, more integration-points and extended precision variables, converged solutions are possible for particles with larger size parameters and axial ratios. Extended precision uses a significance of 32 digits instead of normally 8 digits. The extended precision variables are necessary, as the round-off-errors in the ill-conditioned process of calculating the integrals in the A- and B-matrices (2.12) become increasingly significant for these kind of particles. The DEC OSF/1 compiler, used in this work, applies extended precision for real variables, but only double precision (16 digits) for each part of a complex variable. With these facts in mind, some modifications of the T1-program have been done:

- The maximum size of the matrices used to evaluate the T-matrix has been increased.
- The real variables have been declared real*16 (extended precision).
- The complex variables have been split up into two real variables, through critical parts of the program.
- The accuracy in the integration routine has been increased, as there are values with more digits to pay attention to.
- The accuracy when a solution is accepted as being converged has been increased. The original program requires an accuracy of 1.0% in 8 of 10 angles, whereas the modified version requires 0.1% in all angles.
However, one consequence of these modifications is that the index of refraction must be real, which results in that no absorption cross section can be evaluated. This can be adjusted by splitting all complex variables in the program into two real ones.

To prove the correctness of the modifications, comparing tests were performed for particles which have a converged solution in both the original and the modified program. Additional tests were made to check that the obtained T-matrices still fulfill the condition of energy conservation: $T^*T + \text{Re}(T) = 0$ (* symbolizes the complex conjugate). The modified computer program is called EXTENDT1 and is able to calculate accurate T-matrices, even for particles with the dimension of a human blood cell.

### 3.1.3. The original computer program T3

Program T3 uses the T-matrix, obtained from program T1, to calculate the differential scattering cross sections in all scattering directions. The input parameters to this program are:

- **angint**: The angle between the symmetry axis of the particle and the incident wave.
- **anginp**: The orientation of the particle relative a reference frame in the plane perpendicular to the symmetry axis.
- **anglab**: The orientation of the polarization direction.
- **npnts**: The number of points in which the angular scattering is evaluated is npnts^2.

The normalized differential scattering cross sections as a function of the scattering angles $\theta$ and $\phi$ are put in a vector and stored in the file T3.DAT. The graphical representation is a planar grid as shown in figure 3.2. The deflection angle $\theta$ is mapped along the radius $r$ in the graph ($\theta = 0^\circ$ when $r = 0$ and $\theta = 180^\circ$ when $r = 1$) and $\phi$ is mapped in the normal way using cylindrical coordinates. The logarithm of the normalized differential scattering cross section at each grid-point is calculated and stored in the vector starting in the corner $(X,Y) = (-1,-1)$, continuing by increasing $X$ until $X = 1$ (constant $Y$), followed by the same procedure for the next $Y$-value etc. The value of the grid points outside the unit circle is set to the logarithm of the back scattering value.

![Figure 3.2. The representation of the scattering angles ($\theta, \phi$) on a planar grid.](image-url)
3.1.4. Modifications of program T3

The modified version of T3, called EXTENDT3, is able to use the modified T-matrix. No further modifications have been made in this program.

In program MCDATA, an additional modified version of T3, probability distributions of the scattering angles are computed instead of the normalized differential scattering cross sections. One probability distribution is obtained for each of a certain number of incident angles. The number of incident angles, uniformly distributed between 0° and 90°, is an input parameter (nbrangin) and the variable anginp from program T3 is initially always set to zero. The input parameter npnts is replaced by the two parameters nbrthsc and nbrphsc, which describe the number of spherical coordinate angles θ and φ, respectively, uniformly distributed in space, in which the scattering probability shall be evaluated for every incident angle. Thus, these variables indicate the resolution of the probability distribution. When evaluating the distribution for one incident angle, the normalized differential scattering cross sections are first determined for all combinations of θ and φ, and the probability of scattering in a certain direction is obtained by normalization with the sum of the scattering in all other directions, and with the relative area this scattering direction represents. The obtained probability values are then stored in the file MC.DAT, making it easy to implement a program utilizing these values. A further description of the technical details of the implementation is found in Appendix C.

3.2. The Monte Carlo computer programs

3.2.1. The original Monte Carlo program

As mentioned before, the standard C computer code MCML 2 can be used to simulate the distributions of reflected, absorbed and transmitted light of a specified illuminated medium. The medium consists of a number of infinitely wide parallel layers, each with particular optical properties. The optical parameters (μs, μt, and g), index of refraction and thickness of each layer are input parameters to the program, as well as the number of photon packages and variables defining the resolution of the light distribution. The variables dr and da represent the radial and angular resolution, respectively, and dz is the resolution along the axis parallel to the incident beam.

The output file, obtained from the program, contains the angularly and spatially resolved fractions as well as the total fractions of reflected, absorbed and transmitted light. Other terms, used in the output file are the reflectance per unit area and transmittance per unit area. These terms actually represent the probabilities of detection of a photon per unit area 2. The reflectance per unit area at a specific distance r from the centre of the sample is thus calculated by dividing the total photon weight detected at this distance by the total number of photon package and the area of an annular ring Δa = 2πr dr. The transmittance per unit area is calculated in a corresponding way. Specular reflectance is the amount of light, reflected from the outer boundary of the sample, without interacting the with medium. The remaining reflectance is called diffuse reflectance.
3.2.2. Modifications due to boundary conditions

To be able to model the light distribution of samples, which have a finite extension, a modified version of the computer program is needed. The modified version MCMLBOUN assumes that the sample is laterally surrounded by glass layers, as shown in figure 3.3. The processes occurring when the photon package hits the surrounding glass are the same as when it hits a boundary between two layers. As glass is non-scattering, photons propagating in this glass, away from the sample, will never turn back. The lateral dimension of the sample and the refractive index of the surrounding glass are new input parameters.

![Figure 3.3. The cross sectional geometry for a sample and surrounding glass layers, used in program MCMLBOUN.](image)

3.2.3. Implementation of specific scattering objects

An additional modified version, TMMCML, has been developed within this work. This version is based on the assumption of specific scattering objects, spheroidal particles, are distributed in the medium. The optical parameters in the original version are replaced, in the input file as well as in the program, by the orientation of the scattering objects and the mean distance between them. Standard deviations of these new parameters, assuming Gaussian distributions, are also defined in the input file.

The two main differences between this version and the original one, are the process of the photon package making a new step and the calculation of the new photon direction from an interaction point. The interaction points are in this case equal to the scattering particles. The step of the photon package before a new interaction occurs is obtained from a Gaussian distribution\(^{17}\), where the mean distance and standard deviation are defined in the input file as mentioned before. A new scattering object is hit and the angle between the symmetry axis of the scattering spheroid and the incident direction of the photon package is evaluated. A new direction of the photon package is then obtained using a random number and the data base in MC.DAT. This data base contains the probability of scattering in different directions for several incident angles, as mentioned in chapter 3.1.4. The name of the data base, as well as the resolution of incident and scattering directions are defined by the user as input parameters. If the scattering objects are spheres, only one incident angle is needed. These modifications also require a modification of the absorbing process. A new variable, the normalized absorption cross section, is defined in the input parameters.
file and specifies the relative weight of the incident photon package being absorbed by the scattering object. The input parameters of one layer in program TMMCML is summarized below.

- **meanstep**: The mean distance between the particles.
- **meanstepdev**: The standard deviation of this distance.
- **abscross**: The normalized absorption cross section. (The part of the light absorbed by one particle.)
- **bcux, bcuy, bcuz**: The direction of the particle symmetry axis (the bcuz-direction is parallel with the incident field).
- **st. dev**: The standard deviation of these directions.
- **d**: The thickness

### 3.3. Testing program MCMLBOUN, due to boundary conditions

Monte Carlo simulations, using the program MCMLBOUN accounting for lateral boundary effects, were performed for four samples with identical optical properties but different lateral extensions. The results were compared with each other and with results from an original Monte Carlo simulation in order to investigate the influence of boundary conditions, as the original Monte Carlo method assumes an infinite medium, but the extension of experimental samples are finite. The optical properties were chosen to be similar to those of human tissue and the thickness of the sample was chosen close to experimentally used values. The sizes of the samples were 10000 cm × 10000 cm, 5 cm × 3 cm, 1 cm × 1 cm and 0.1 cm × 0.1 cm. Table 3.1 shows further input parameters.

<table>
<thead>
<tr>
<th># No. of photons</th>
<th>500000</th>
</tr>
</thead>
<tbody>
<tr>
<td># dz, dr</td>
<td>0.25</td>
</tr>
<tr>
<td># No. of dz, dr, da</td>
<td>1 75 30</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># Number of layers</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td># layer</td>
<td>n</td>
</tr>
<tr>
<td># n for medium above</td>
<td>1</td>
</tr>
<tr>
<td># layer 1</td>
<td>1.5</td>
</tr>
<tr>
<td># layer 2</td>
<td>1.4</td>
</tr>
<tr>
<td># layer 3</td>
<td>1.5</td>
</tr>
<tr>
<td># n for medium below</td>
<td>1</td>
</tr>
<tr>
<td># n for surrounding glass</td>
<td>1.5</td>
</tr>
</tbody>
</table>

Table 3.1. Input parameters of the tests in section 3.3, investigating the influence of boundary conditions. The variables mua and mus represent the absorption and scattering coefficients $\mu_a$ and $\mu_s$, $n$ is the refractive index and $g$ is the anisotropy factor. The radial and angular resolutions are described by dz, dr and da. The length unit is always cm.

### 3.4. Testing limitations and accuracy

Several different comparisons have been made, using the modified programs EXTENDT1, EXTENDT3 and MCDATA. The calculations were mainly performed for particles with sizes and properties similar to erythrocytes and a surrounding medium
similar to blood plasma. The parameters (chapter 3.1.1 and 3.1.3), common in all the tests using these programs, were thus \( n = 1.345 \) (refractive index of blood plasma\(^{21}\)), \( m_r = 1.045 \) (refractive index of a blood cell relative the blood plasma\(^{22}\)), \( m_i = 0 \) (the imaginary refractive index of a blood cell, which is negligible when evaluating the scattered field\(^{23}\)) and \( \lambda_0 = 632.8 \) nm (wavelength of a He-Ne laser\(^{24}\)). The variable \( n_{\text{theta}} \) (number of integration points) was always assigned the value of twice \( n_{\text{rank}} \).

In this section, tests were made to study the limits in size and shape of the particle in order to obtain a converged solution, as well as the influence of different polarization directions of the incident light and the number of incident and scattering angles needed to obtain a sufficient accuracy. In the last-mentioned study, the program TMMCML was executed for four different values of the variable \( n_{\text{brthsc}} \), which represents the number of scattering deflection angles \( \theta \), in which the differential scattering cross section shall be determined. The values assigned to this variable were 46, 91, 181 and 721, respectively, and the remaining input parameters of these simulations are specified in table 3.2. The scattering particles were spheres with a volume equal to a blood cell. This volume equals \( 94 \ \mu\text{m}^3 \), which leads to \( k_a = 37.67 \), \( a/b = 1 \) and requires \( n_{\text{rank}} = 50 \), where \( n_{\text{rank}} \) represents the number of expansion coefficients in eq. 2.6, 2.8 and 2.9. The orientation of polarization (anglab) was set to zero as well as the incident angles (angint and anginp), which are arbitrary in the case of a sphere.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td># nbrangin and nbrophsc</td>
<td>1 37</td>
</tr>
<tr>
<td># No. of photons</td>
<td>100000</td>
</tr>
<tr>
<td># dz, dr</td>
<td>102 4</td>
</tr>
<tr>
<td># No. of dz, dr, da</td>
<td>1 75 30</td>
</tr>
<tr>
<td># x_size and y_size</td>
<td>5E+06 5E+06</td>
</tr>
<tr>
<td># Number of layers</td>
<td>3</td>
</tr>
<tr>
<td># layer</td>
<td>n m-step m-dev. abs. thickness</td>
</tr>
<tr>
<td># n for medium above</td>
<td>1</td>
</tr>
<tr>
<td># layer 1</td>
<td>1.5 0 0 0 1</td>
</tr>
<tr>
<td># layer 2</td>
<td>1.345 0.1 0.02 0.0001 100</td>
</tr>
<tr>
<td># layer 3</td>
<td>1.5 0 0 0 1</td>
</tr>
<tr>
<td># n for medium below</td>
<td>1</td>
</tr>
<tr>
<td># n for surrounding glass</td>
<td>1.5</td>
</tr>
</tbody>
</table>

Table 3.2. Input parameters of the TMMCML-tests in section 3.4, performed to study the necessary accuracy of the variable \( n_{\text{brthsc}} \) (number of scattering deflection angles \( \theta \)). The length unit is cm.

3.5. Evaluations and tests of the programs EXTENDT3 and TMMCML

**Test 1.** In order to investigate the influence of particle size on the light scattering by one particle as well as on the macroscopical light distribution, executions of the programs EXTENDT3 and TMMCML were made for spheres of three different sizes as scattering particles. One sphere with a volume equal to a blood cell (\( V = 94 \ \mu\text{m}^3, k_a = 37.67, a/b = 1 \) and \( n_{\text{rank}} = 50 \)), one larger sphere (\( V = 390 \ \mu\text{m}^3, k_a = 60 \) and \( n_{\text{rank}} = 100 \)) and one
smaller sphere \((V = 14 \, \mu m^3, \, ka = 20\) and \(nrank = 30\)). The three variables \(angint\), \(anginp\) (incident angles) and \(anglab\) (polarization direction) were set to zero. The variable \(npnts\) was assigned the value of 181 (the number of evaluated scattering angles is \(npnts^2\)). The normalized differential scattering cross sections in various directions for the spheres were calculated using program EXTENDT3. The logarithms of the obtained cross sections were then plotted in three-dimensional graphs utilizing a small MATLAB program. The input parameters of program TMMCML, used for the calculation of the macroscopical light distribution, are found in table 3.3. The g-factor (average value of the cosine of the scattering deflection angle \(\theta\)) was calculated for the three spheres using a small FORTRAN program.

<table>
<thead>
<tr>
<th># nbrangin, nbtsc and nbtnphsc</th>
<th>1</th>
<th>181</th>
<th>37</th>
</tr>
</thead>
<tbody>
<tr>
<td># No. of photons</td>
<td>100000</td>
<td></td>
<td></td>
</tr>
<tr>
<td># dz, dr</td>
<td>102</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td># No. of dz, dr, da</td>
<td>1</td>
<td>75</td>
<td>30</td>
</tr>
<tr>
<td># x size and y size</td>
<td>5E+06</td>
<td>5E+06</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.3. Input parameters of the TMMCML-simulations in test 1, made to study the influence of particle size on the macroscopic light distribution. The length unit is cm.

**Test 2.** The influence of shape and orientation of scattering particles were investigated in this test. The normalized differential scattering cross sections in various directions for an oblate spheroid with the dimension of an erythrocyte were calculated using program EXTENDT3. The values of \(V = 94 \, \mu m^3\) and \(b = 3.91 \, \mu m^2\) result in \(alb = 0.3754\) and \(ka = 19.60\). These dimensions require \(nrank = 66\) (number of expansion coefficients). Computations were made for three different incident angles, \(angint = 0^\circ, 45^\circ\) and \(90^\circ\). The variables \(anginp\) and \(anglab\) were set to zero and \(npnts = 181\). Three-dimensional graphs were plotted to show the angular scattering and the small FORTRAN program, mentioned before, was used to find the anisotropy factor \(g\), in the different cases. The influence of particle shape and orientation on the macroscopical light distribution was analyzed by comparing the output data from program TMMCML, using spheres and the recently described oblate spheroids in various orientations as scattering particles. The volume of the spheres were equal to human blood cells (94 \(\mu m^3\)) and further properties were described in test 1. One simulation was made with the spheres and six with different orientations of the spheroids, described in table 3.4. Remaining input parameters were unchanged from test 1, except the variable \(nbrangin\), which had a value of 91 in the case of spheroids. The absorption cross section was assumed to be independent of the shape of the particle.²⁵

²⁵
# bcux rand. 1 0 0 1 1 sphere
# bcuy rand. 0 1 0 1 1 sphere
# bcuz rand. 0 0 1 1 1 sphere
# st. dev. 0 0 0 0 0.5 sphere

Table 3.4. The orientation of the scattering spheroids in the seven samples in test 2. The parameters (bcux, bcuy, bcuz) represent the direction of the symmetry axis of the spheroids and this axis is parallel with the incident light beam in the case of (bcux, bcuy, bcuz) = (0, 0, 1). The standard deviation of this direction is denoted by st. dev. in the table and rand. indicates randomized orientation. If st. dev. equals zero, all spheroids have exactly the same direction. In case of spheres, no orientation is needed.

The outlines of a normal blood cell22 and an oblate spheroid with an equal volume are drawn in figure 3.4 and 3.5.

\[\begin{array}{ccccccc}
\text{spheroid} & 1 & \text{spheroid} & 2 & \text{spheroid} & 3 & \text{spheroid} & 4 & \text{spheroid} & 5 & \text{spheroid} & 6 & \text{sphere} \\
\# bcux & \text{rand.} & 1 & 0 & 0 & 1 & 1 & \text{sphere} \\
\# bcuy & \text{rand.} & 0 & 1 & 0 & 1 & 1 & \text{sphere} \\
\# bcuz & \text{rand.} & 0 & 0 & 1 & 1 & 1 & \text{sphere} \\
\# \text{st. dev.} & & 0 & 0 & 0 & 0 & 0.5 & \text{sphere} \\
\end{array}\]

Figure 3.4. The dimension of a blood cell.
Test 3 was performed to investigate the influence of the distance between the scattering particles on the transmitted and reflected light obtained from TMMCCML-simulations. Two different distances were used. In the first case, the average distance was assigned a value of 0.2 cm and in the second a value of 0.05 cm. The standard deviation was 20% of the average distance in both cases. The same particles and orientations as in the previous test were utilized. Table 3.5. displays the remaining input parameters.

| # dz, dr | 7 | 0.1 |
| # No. of dz, dr, da | 1 | 75 | 30 |
| # x size and y size | 5000 | 5000 |

| # Number of layers | 3 |
| # layer | n | abs. thickness |
| # n for medium above | 1 |
| # layer 1 | 1.5 | 0 | 1 |
| # layer 2 | 1.345 | 0.004 | 5 |
| # layer 3 | 1.5 | 0 | 1 |
| # n for medium below | 1 |
| # n for surrounding glass | 1.5 |

Table 3.5. Input parameters of TMMCCML-simulations in test 3, investigating the influence of the distance between the scattering particles on the light distribution. The length unit is cm.
3.6. **Comparison with the original Monte Carlo model**

A comparison between an original Monte Carlo simulation and a simulation with scattering spheres using TMMCMCL was made, where the extension and thickness of the sample were very large, in order to obtain multiple scattering for all photons. The properties of the spheres were still unchanged (volume equal to an erythrocyte). The mean distance between two interaction points and the absorption during an interaction were identical in the two simulations and the $g$-factor in the original Monte Carlo program was assigned the mean value of the cosine of the scattering deflection angle $\theta$ ($0.9940$), obtained by a small FORTRAN program from the scattering probability distribution of one sphere. In order to save time, the original Monte Carlo program was used instead of MCMLBOUN, as the lateral extension of the samples were assumed to be large. Table 3.6a and 3.6b present the input parameters.

<table>
<thead>
<tr>
<th># No. of photons</th>
<th>250000</th>
</tr>
</thead>
<tbody>
<tr>
<td># dz, dr</td>
<td>202 6</td>
</tr>
<tr>
<td># No. of dz, dr, da</td>
<td>1 75 30</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># Number of layers</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td># One line for each layer</td>
<td>n mua mus g thickness</td>
</tr>
<tr>
<td># n for medium above</td>
<td>1</td>
</tr>
<tr>
<td># layer 1</td>
<td>1.5 0 0 0 1</td>
</tr>
<tr>
<td># layer 2</td>
<td>1.345 0.001 9.9999 0.994 200</td>
</tr>
<tr>
<td># layer 3</td>
<td>1.5 0 0 0 1</td>
</tr>
<tr>
<td># n for medium below</td>
<td>1</td>
</tr>
</tbody>
</table>

**Table 3.6a. Input parameters of the original Monte Carlo simulations in section 3.6.**

<table>
<thead>
<tr>
<th># nbrangin, nbrthsc and nbrphsc</th>
<th>1 181 37</th>
</tr>
</thead>
<tbody>
<tr>
<td># No. of photons</td>
<td>250000</td>
</tr>
<tr>
<td># dz, dr</td>
<td>202 6</td>
</tr>
<tr>
<td># No. of dz, dr, da</td>
<td>1 75 30</td>
</tr>
<tr>
<td># x_size and y_size</td>
<td>5E+06 5E+06</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># Number of layers</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td># layer</td>
<td>n m-step m-dev. abs. thickness</td>
</tr>
<tr>
<td># n for medium above</td>
<td>1</td>
</tr>
<tr>
<td># layer 1</td>
<td>1.5 0 0 0 1</td>
</tr>
<tr>
<td># layer 2</td>
<td>1.345 0.1 0.02 1.0E-05 200</td>
</tr>
<tr>
<td># layer 3</td>
<td>1.5 0 0 0 1</td>
</tr>
<tr>
<td># n for medium below</td>
<td>1</td>
</tr>
<tr>
<td># n for surrounding glass</td>
<td>1.5</td>
</tr>
</tbody>
</table>

**Table 3.6b. Input parameters of the TMMCMCL-simulations in section 3.6.**

23
4. Results

The results and comparisons of the calculations and simulations, using the modified computer programs will be presented in this chapter. In all diagrams and tables, the length unit is cm. Reflectance per unit area has consequently the unit cm$^{-2}$.

4.1. Testing program MCMLBOUN, due to boundary conditions

The distributions of reflected and transmitted light obtained when executing the program MCMLBOUN for four samples with different lateral extensions are shown in figures 4.1 and 4.2, as well as the corresponding result from an original Monte Carlo simulation. Table 4.1 presents the total reflected, absorbed and transmitted parts and table 4.2 indicates the time of each simulation. The difference in reflected and transmitted light between the original Monte Carlo simulation and the two largest samples (10000 cm $\times$ 10000 cm and 5 cm $\times$ 3 cm) in the MCMLBOUN-simulations never exceeds 3%, except close to origo. Regarding the total amounts, this deviation is much less than 1%. From the results of the simulations using samples with sizes less than 1 cm $\times$ 1 cm a larger deviation can be observed, up to 25% at some radial distances.

![Figure 4.1](image)

*Figure 4.1. The reflectance per unit area versus the radial distance from the centre of the sample. The filled rhombs correspond to the result of the original Monte Carlo program. The other symbols correspond to results of program MCMLBOUN, where the sizes of the samples are described in the figure. The used parameter were $\mu_a = 2.6$ cm$^{-1}$, $\mu_s = 267$ cm$^{-1}$ and $g = 0.95$. The thickness was 0.05 cm.*
Figure 4.2. The transmittance per unit area versus the radial distance from the centre of the sample. The filled rhombs correspond to the result of the original Monte Carlo program. The other symbols correspond to results of program MCMLBOUN, where the sizes of the samples are described in the figure. The used parameter were $\mu_a = 2.6 \text{ cm}^{-1}$, $\mu_s = 267 \text{ cm}^{-1}$ and $g = 0.95$. The thickness was 0.05 cm.

Table 4.1. The parts of reflected, absorbed and transmitted light for the five different sizes of the samples used in the tests described in section 3.3, investigating the influence of boundary conditions.

<table>
<thead>
<tr>
<th></th>
<th>orig. MC (infinite)</th>
<th>10000 cm$^*$</th>
<th>5.0 cm$^*$</th>
<th>1.0 cm$^*$</th>
<th>0.1 cm$^*$</th>
</tr>
</thead>
<tbody>
<tr>
<td># Specular reflectance</td>
<td>0.04110</td>
<td>0.04110</td>
<td>0.04110</td>
<td>0.04110</td>
<td>0.04110</td>
</tr>
<tr>
<td># Diffuse reflectance</td>
<td>0.16110</td>
<td>0.16059</td>
<td>0.15965</td>
<td>0.13306</td>
<td>0.06595</td>
</tr>
<tr>
<td># Absorbed fraction</td>
<td>0.30816</td>
<td>0.30813</td>
<td>0.30671</td>
<td>0.26503</td>
<td>0.15045</td>
</tr>
<tr>
<td># Transmittance</td>
<td>0.48965</td>
<td>0.49018</td>
<td>0.48900</td>
<td>0.46251</td>
<td>0.39259</td>
</tr>
</tbody>
</table>

Table 4.2. The simulation time for the five different sizes of the samples used in the tests described in section 3.3, investigating the influence of boundary conditions.

<table>
<thead>
<tr>
<th></th>
<th>orig. MC (infinite)</th>
<th>10000 cm$^*$</th>
<th>5.0 cm$^*$</th>
<th>1.0 cm$^*$</th>
<th>0.1 cm$^*$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t$</td>
<td>2152s</td>
<td>3683s</td>
<td>3092s</td>
<td>2167s</td>
<td>1108s</td>
</tr>
</tbody>
</table>

4.2. Testing limitations and accuracy

When testing which sizes and shapes of particles, possible for obtaining a converged solution in program EXTENDT1, the minimal axial ratio obtained for $k_a = 19.6$ was $a/b = 0.34$, the largest size parameter for $a/b = 0.3754$ was $k_a = 22$ and the highest possible value of the relative refractive index for this kind of particles was $m_r = 1.06$. Converged
solutions were obtained for prolate particles with about 2% larger volumes than the corresponding oblate particles.

Tests were also performed indicating that the angular scattering for two incident angles separated by $1^\circ$ deviate less than 1%. Results obtained with different polarization directions of the incident light showed the same small deviation.

The obtained radial resolutions of the reflectance per unit area and transmittance per unit area from the TMMCML-simulations when testing the output resolution needed by changing the variable $\text{nbrthsc}$ (the number of scattering deflection angles $\theta$) are shown in figures 4.3 and 4.4. The total amounts are shown in table 4.3. The differences in total amounts of reflected and transmitted light are less than 2%, comparing $\text{nbrthsc} = 46$ and $\text{nbrthsc} = 721$. Investigating radial resolution of the light, the deviation is below 6%, except close to origo.

![Figure 4.3](image)

*Figure 4.3. The reflectance per unit area versus the radial distance from the centre of the sample in the tests described in section 3.4, investigating the accuracy needed. The symbols in the graph correspond to $\text{nbrthsc} = 46, 91, 181$ and 721, respectively, where $\text{nbrthsc}$ represents the number of scattering deflection angles $\theta$. The mean distance between the particles was 0.1 cm and the sample thickness was 100 cm.*
Figure 4.4. The transmittance per unit area versus the radial distance from the centre of the sample in the tests described in section 3.4, investigating the accuracy needed. The symbols in the graph correspond to \( \text{nbrthsc} = 46, 91, 181 \) and 721, respectively, where \( \text{nbrthsc} \) represents the number of scattering deflection angles \( \theta \). The mean distance between the particles was 0.1 cm and the sample thickness was 100 cm.

<table>
<thead>
<tr>
<th>( \text{nbrthsc} )</th>
<th>( \text{Specular reflectance} )</th>
<th>( \text{Diffuse reflectance} )</th>
<th>( \text{Absorbed fraction} )</th>
<th>( \text{Transmittance} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>46</td>
<td>0.04274</td>
<td>0.51734</td>
<td>0.28436</td>
<td>0.15556</td>
</tr>
<tr>
<td>91</td>
<td>0.04274</td>
<td>0.51296</td>
<td>0.28490</td>
<td>0.15941</td>
</tr>
<tr>
<td>181</td>
<td>0.04274</td>
<td>0.51136</td>
<td>0.28513</td>
<td>0.16077</td>
</tr>
<tr>
<td>721</td>
<td>0.04274</td>
<td>0.51557</td>
<td>0.28324</td>
<td>0.15846</td>
</tr>
</tbody>
</table>

*Table 4.3. The parts of reflected, absorbed and transmitted light for the four values of \( \text{nbrthsc} \) in the tests described in section 3.4, investigating the accuracy needed of the variable \( \text{nbrthsc} \) (the number of scattering deflection angles \( \theta \)).*

### 4.3. Evaluations and tests of the programs EXTENDT3 and TMMCML

**Test 1.** This test was made in order to investigate the influence of particle size on light scattering and distribution. The following pages contain three-dimensional graphs, with the logarithm of the normalized differential scattering cross section plotted versus the scattering angles according to the rules described in section 3.1.3 (\( \theta \) along the radius), for the spheres with size parameters \( \text{ka} = 20.0 \) (\( V = 14 \ \mu\text{m}^3 \)), \( \text{ka} = 37.7 \) (\( V = 94 \ \mu\text{m}^3 \)) and \( \text{ka} = 60.0 \) (\( V = 390 \ \mu\text{m}^3 \)), respectively. Figure 4.5 shows the scattering probability versus the scattering deflection angle \( \theta \) for these three spheres and the radially resolved reflectance and transmittance per unit area obtained from the TMMCML-simulations are presented in figures 4.6 and 4.7. Corresponding total amounts are found in table 4.4. The g-values obtained from the FORTRAN program were \( g = 0.9891 \) (\( \text{ka} = 20 \)), \( g = 0.9940 \) (\( \text{ka} = 37.7 \)) and \( g = 0.9920 \) (\( \text{ka} = 60 \)).
TEST 1
Logarithm of the normalized differential scattering cross section of a sphere.
The volume is 14 μm³.
TEST 1
Logarithm of the normalized differential scattering cross section of a sphere.
The volume is 94 $\mu$m$^3$. 
TEST 1

Logarithm of the normalized differential scattering cross section of a sphere.
The volume is 390 $\mu$m$^3$. 
Figure 4.5. The scattering probability versus the deflection angle $\theta$, for three spheres with size parameters $ka = 37.7$, 60.0 and 20.0, respectively (test 1). $\theta = 0^\circ$ represents the scattering probability between $0^\circ$ and $0.5^\circ$, whereas $\theta = 1^\circ$ represents the interval between $0.5^\circ$ and $1.5^\circ$ etc.

Figure 4.6. The reflectance per unit area versus the radial distance from the centre of the sample obtained from the TMMCML-simulations in test 1. The rhombs, squares and triangles correspond to scattering spheres of volumes $14 \mu m^3$ ($ka = 20.0$), $94 \mu m^3$ ($ka = 37.7$) and $390 \mu m^3$ ($ka = 60.0$), respectively. The mean distance between the particles was 0.1 cm and the sample thickness was 100 cm.
Figure 4.7. The transmittance per unit area versus the radial distance from the centre of the sample obtained from the TMMCML-simulations in test 1. The rhombs, squares and triangles correspond to scattering spheres of volumes $14\mu m^3$ ($ka = 20.0$), $94\mu m^3$ ($ka = 37.7$) and $390\mu m^3$ ($ka = 60.0$), respectively. The mean distance between the particles was $0.1\ cm$ and the sample thickness was $100\ cm$.

<table>
<thead>
<tr>
<th></th>
<th>$ka = 20.0$</th>
<th>$ka = 37.7$</th>
<th>$ka = 60.0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Specular reflectance</td>
<td>0.04274</td>
<td>0.04274</td>
<td>0.04274</td>
</tr>
<tr>
<td>#Diffuse reflectance</td>
<td>0.60380</td>
<td>0.51136</td>
<td>0.47667</td>
</tr>
<tr>
<td>#Absorbed fraction</td>
<td>0.26809</td>
<td>0.28513</td>
<td>0.28862</td>
</tr>
<tr>
<td>#Transmittance</td>
<td>0.08538</td>
<td>0.16077</td>
<td>0.19198</td>
</tr>
</tbody>
</table>

Table 4.4. The parts of reflected, absorbed and transmitted light obtained by TMMCML-simulations in test 1, investigating the influence of particle size on the light distribution.

Test 2. In this test the influence of particle shape and orientation on light scattering and distribution was studied. Three-dimensional graphs, with the logarithm of the normalized differential scattering cross section plotted versus the scattering angles according to the rules described in section 3.1.3, are presented on the three following pages for an oblate spheroid with the dimension of a human blood cell. The incident angles are $0^\circ$, $45^\circ$ and $90^\circ$, respectively. These pictures show much smaller variations for the scattering azimuthal angle $\phi$, than for the deflection angle $\theta$. Figure 4.8 shows the scattering probability versus the scattering deflection angle $\theta$, for these three cases and the obtained corresponding $g$-values were 0.9967, 0.9960 and 0.9887, respectively.
TEST 2
Logarithm of the normalized differential scattering cross section of an oblate spheroid.
The incident angle is 0°.
TEST 2
Logarithm of the normalized differential scattering cross section of an oblate spheroid.
The incident angle is 45°.
TEST 2
Logarithm of the normalized differential scattering cross section of an oblate spheroid.
The incident angle is $90^\circ$. 
Figure 4.8a. The scattering probability versus the deflection angle $\theta$ for three different orientations of an oblate spheroid with the dimension of a blood cell. The incident angles are $0^\circ$, $45^\circ$ and $90^\circ$, respectively. The cross symbols represent the scattering probability by a sphere with an equal volume (94 $\mu$m$^3$). $\theta = 0^\circ$ represents the scattering probability between $0^\circ$ and $0.5^\circ$, whereas $\theta = 1^\circ$ represents the interval between $0.5^\circ$ and $1.5^\circ$ etc.

Figure 4.8b. Continuation of figure 4.8a for larger radial distances.

The radially resolved transmittance per unit area obtained from the TMMCML-simulations for the different orientations of the spheroids are presented in figure 4.9. Corresponding total amounts are found in table 4.5. The different symbols in the graph in figure 4.9 and the different columns in table 4.5 represent the different orientations of the scattering spheroids.
Table 4.5. The parts of reflected, absorbed and transmitted light obtained by the TMMCML-simulation in test 2, using different orientations of the scattering spheroidal particles. Each column represents one sample containing spheroidal particles and the particles have different orientations in the different samples. The orientation of the particle symmetry axis is described by the three values on the first row, where the direction (0, 0, 1) is parallel with the incident light. If there is a deviation in the orientation, this is denoted by st.dev. The last column corresponds to scattering spheres with an equal volume.

<table>
<thead>
<tr>
<th>Orientation</th>
<th>rando-mized</th>
<th>(1,0,0)</th>
<th>(0,1,0)</th>
<th>(0,0,1)</th>
<th>(1,1,1)</th>
<th>st.dev.=0.5</th>
<th>spheres</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Specular reflectance</td>
<td>0.04274</td>
<td>0.04274</td>
<td>0.04274</td>
<td>0.04274</td>
<td>0.04274</td>
<td>0.04274</td>
<td>0.04274</td>
</tr>
<tr>
<td>#Diffuse reflectance</td>
<td>0.00777</td>
<td>0.00772</td>
<td>0.00854</td>
<td>0.02942</td>
<td>0.00870</td>
<td>0.00870</td>
<td>0.51136</td>
</tr>
<tr>
<td>#Absorbed fraction</td>
<td>0.12497</td>
<td>0.15070</td>
<td>0.15059</td>
<td>0.11829</td>
<td>0.13419</td>
<td>0.12746</td>
<td>0.28513</td>
</tr>
<tr>
<td>#Transmittance</td>
<td>0.82453</td>
<td>0.79885</td>
<td>0.79814</td>
<td>0.80956</td>
<td>0.81437</td>
<td>0.8211</td>
<td>0.16077</td>
</tr>
</tbody>
</table>

Figure 4.9a. The transmittance per unit area versus the radial distance from the centre of the sample in the TMMCML-simulations in test 2, using samples containing spheroids in different orientations in the different cases. The different symbols represent the orientations of the scattering spheroidal particles. The small strokes in the lower part of the figure represent the transmittance per unit area obtained by a simulation using spheres of equal volume as scattering particles.
Figure 4.9b. Continuation of figure 4.9a for larger radial distances.

Test 3. The amounts of reflected, absorbed and transmitted light obtained by two TMMCML simulations, using different average distances, 0.2 cm and 0.05 cm, between the scattering particles, are shown in tables 4.6 and 4.7.
Table 4.6. The parts of reflected, absorbed and transmitted light obtained by the TMMCML-simulations in test 3, investigating the influence of the distance between the scattering particles on the light distribution. The results correspond to a simulation where \( \text{meanstep} = 0.2 \) cm. Different orientations of the scattering spheroidal particles were used. The last column corresponds to scattering spheres with an equal volume. The sample thickness was 5 cm.

<table>
<thead>
<tr>
<th>Orientation</th>
<th>randomized</th>
<th>(1,0,0)</th>
<th>(0,1,0)</th>
<th>(0,0,1)</th>
<th>(1,1,1)</th>
<th>(1,1,1) st.dev. = 0.5</th>
<th>spheres</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Specular reflectance</td>
<td>0.04274</td>
<td>0.04274</td>
<td>0.04274</td>
<td>0.04274</td>
<td>0.04274</td>
<td>0.04274</td>
<td>0.04274</td>
</tr>
<tr>
<td>#Diffuse reflectance</td>
<td>0.03362</td>
<td>0.03247</td>
<td>0.03223</td>
<td>0.04042</td>
<td>0.03131</td>
<td>0.03433</td>
<td>0.05911</td>
</tr>
<tr>
<td>#Absorbed fraction</td>
<td>0.12335</td>
<td>0.14856</td>
<td>0.15162</td>
<td>0.10265</td>
<td>0.12277</td>
<td>0.12501</td>
<td>0.14840</td>
</tr>
<tr>
<td>#Transmittance</td>
<td>0.80030</td>
<td>0.77623</td>
<td>0.77341</td>
<td>0.81419</td>
<td>0.80319</td>
<td>0.79792</td>
<td>0.74975</td>
</tr>
</tbody>
</table>

Table 4.7. The parts of reflected, absorbed and transmitted light obtained by the TMMCML-simulations in test 3, investigating the influence of the distance between the scattering particles on the light distribution. The results correspond to a simulation where \( \text{meanstep} = 0.05 \) cm. Different orientations of the scattering spheroidal particles were used. The last column corresponds to scattering spheres with an equal volume. The sample thickness was 5 cm.

<table>
<thead>
<tr>
<th>Orientation</th>
<th>randomized</th>
<th>(1,0,0)</th>
<th>(0,1,0)</th>
<th>(0,0,1)</th>
<th>(1,1,1)</th>
<th>(1,1,1) st.dev. = 0.5</th>
<th>spheres</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Specular reflectance</td>
<td>0.04274</td>
<td>0.04274</td>
<td>0.04274</td>
<td>0.04274</td>
<td>0.04274</td>
<td>0.04274</td>
<td>0.04274</td>
</tr>
<tr>
<td>#Diffuse reflectance</td>
<td>0.00829</td>
<td>0.00753</td>
<td>0.00756</td>
<td>0.02516</td>
<td>0.00923</td>
<td>0.00876</td>
<td>0.06356</td>
</tr>
<tr>
<td>#Absorbed fraction</td>
<td>0.41788</td>
<td>0.47328</td>
<td>0.47316</td>
<td>0.37935</td>
<td>0.42021</td>
<td>0.42293</td>
<td>0.58636</td>
</tr>
<tr>
<td>#Transmittance</td>
<td>0.53110</td>
<td>0.47645</td>
<td>0.47654</td>
<td>0.55277</td>
<td>0.52783</td>
<td>0.52558</td>
<td>0.30735</td>
</tr>
</tbody>
</table>

Obviously, the reflected and transmitted parts of the light increase when the distance between the scattering particles increases. The absorbed part decreases.

4.4. **Comparison with original Monte Carlo model**

A three-dimensional picture representing the Henyey-Greenstein phase function is shown on next page. The representation of the scattering angles on a planar grid was described in section 3.1.3 (the scattering deflection angle \( \theta \) along the radius). The probability values in each grid point are normalized in order to obtain a normalized total scattering cross section equal to the sphere with a volume of an erythrocyte (\( \sigma_g / \pi a^2 = 3.1 \)).
Henyey-Greenstein distribution for $g = 0.994$. 
The radial resolutions of the reflected and transmitted light for the simulation of the original Monte Carlo program are shown in figures 4.10 and 4.11, as well as the results from the corresponding TMMCML-simulation, using spheres as scattering particles. The reflected, absorbed and transmitted parts of the light are displayed in table 4.8.

![Graph of reflectance per unit area vs. radial distance](attachment:graph.png)

Figure 4.10a. The reflectance per unit area versus the radial distance from the centre of the sample, comparing the original Monte Carlo model with the TMMCML program, using spheres as scattering particles. The rhombs correspond to the original MCML program and the squares to TMMCML. The sample thickness was 200 cm and the optical properties in the original program were $\mu_a = 0.0001 \text{ cm}^{-1}$, $\mu_s = 9.9999 \text{ cm}^{-1}$ and $g = 0.9940$. Corresponding parameters in the modified version were meanstep = 0.1 cm (meanstepdev = 0.02 cm) and abscross = 0.00001.

![Graph of reflectance per unit area vs. radial distance](attachment:graph2.png)

Figure 4.10b. Continuation of figure 4.10a for larger radial distances.
Figure 4.11. The transmittance per unit area versus the radial distance from the centre of the sample, comparing the original Monte Carlo model with the TMMCMCL program, using spheres as scattering particles. The rhombs correspond to the original MCML program and the squares to TMMCMCL.

<table>
<thead>
<tr>
<th></th>
<th>original MC</th>
<th>TMMCMCL</th>
</tr>
</thead>
<tbody>
<tr>
<td># Specular reflectance</td>
<td>0.04274</td>
<td>0.04274</td>
</tr>
<tr>
<td># Diffuse reflectance</td>
<td>0.74134</td>
<td>0.73988</td>
</tr>
<tr>
<td># Absorbed fraction</td>
<td>0.07153</td>
<td>0.07399</td>
</tr>
<tr>
<td># Transmittance</td>
<td>0.14439</td>
<td>0.14340</td>
</tr>
</tbody>
</table>

Table 4.8. The parts of reflected, absorbed and transmitted light obtained by simulations of the original Monte Carlo program and the TMMCMCL program, using spheres as scattering particles.

The difference in transmitted and reflected light is less than 5% at all radial distances except close to origo. The simulation time in the modified version is almost four times that used by the original program (34.0 hours for the modified program and 9.6 hours for the original program).
5. Discussion and conclusions

5.1. Comments to the tests and evaluations

5.1.1. Testing program MCMLBOUN, due to boundary conditions

The conclusion of this test, when investigating the influence of the lateral extension of the sample on the light distribution, is that for normal experimentally used tissue samples of common thickness, in the order of millimetres, the extension of the sample ought to be at least a few centimetres in each direction, if the original Monte Carlo program, assuming infinitely wide layers, should be a correct model. Higher values of $\mu_s$ and lower values of $g$ require larger samples. The refractive indices of the sample and the surrounding medium influence the necessary size of the sample as well, as a big difference between these two indices results in more light reflected back towards the surrounding medium into the sample instead of escape in the borders. The tests confirm reasonable results by the program MCMLBOUN and the fact that this modified version requires more simulation time for samples, possible to approximate as infinite than the original program. The increase in simulation time depends on more necessary computations in MCMLBOUN, as the distance from the position of the photon package to three different boundaries, instead of one in the original program, must be calculated for every new step. The small deviations in reflectance per unit area close to origo in figure 4.1, depend on the low number of photons emerged in this area.

5.1.2. Testing limitations and accuracy

These tests, investigating needed accuracy, assure that it is sufficient to assign the number of scattering deflection angles, nbrthsc, a value of 46, when the scattering particles are spheres. The results show also that the number of incident angles, nbrangin does not need to exceed 91 and that the polarization direction of the incident light is arbitrary. With these facts in mind together with the smaller dependence of the azimuthal scattering angle $\phi$ (test 2 in section 4.2) and information of the computer capacity, reasonable input parameters to program TMMCML are nbrangin = 91, nbrthsc = 181 and nbrphsc = 37. The last parameter represents the number of azimuthal scattering angles $\phi$. The small deviations close to origo in the distributions of transmitted and reflected light (figures 4.3 and 4.4) depend on the low number of emerged photons in these areas. Finally, the limits of size and shape of the scattering particles in order to obtain converged solutions by program EXTENDT1 agree very well with those obtained by Mishchenko et al.\textsuperscript{26}, and are much better than the values achieved by Hill et al.\textsuperscript{27}, probably using non-extended variables.

5.1.3 Evaluations and tests of the programs EXTENDT3 and TMMCML

Test 1 shows the well-known phenomenon\textsuperscript{28}, that large spheres tend to scatter more light in the forward direction than small ones, comparing three spheres with size parameters $ka = 20.0, 37.7$ and $60.0$ ($V = 14 \ \mu m^3$, $94 \ \mu m^3$ and $390 \ \mu m^3$). The three-dimensional figures on pages 28-30, as well as the scattering probability distribution in figure 4.5 indicate this property. The obtained $g$-values (0.9891, 0.9940 and 0.9920) for the three spheres, respectively, show that this value is lower for the largest sphere than for the sphere with a size parameter of 37.7. The $g$-factor, which only describes the mean value of the cosine of
the scattering deflection angle $\theta$, is thus no sufficient quantity of describing the distribution of light scattered by a particle. Neither is the forward scattering. The oscillating patterns of the scattering probability distributions depend on interference phenomena. The TMMCML-simulations made for the three mentioned spheres, as scattering particles, were mainly performed to check the program TMMCML rather than to obtain any exact results. The mean distance between two scattering particles and the absorption in each of these particles were assumed to be equal in these three simulations. The first assumption results in different volume proportions of particles in relation to surrounding medium and the second is wrong, while the absorption cross section is volume dependent. Nevertheless, the TMMCML-simulations manifest the fact, that more light is forward scattered when larger particles are used (figures 4.6 and 4.7). The high value of reflectance per unit area in origo (figure 4.6) for the largest sphere is probably due to the higher complete back scatter value for this particle. The program TMMCML yields thus reasonable results in well accordance with general scattering theory.

When comparing the results of scattering by an oblate spheroidal particle, for three different incident angles ($0^\circ$, $45^\circ$ and $90^\circ$) in test 2, it is evident that a smaller incident angle results in more light scattered straight forward than a larger incident angle. This is shown in the three-dimensional figures on pages 33-35, the scattering probability distribution in figure 4.8, as well as indicated by the obtained g-values (0.9967, 0.9960 and 0.9891 in the three cases, respectively) and is due to the larger area of the particle perpendicular to the direction of the incident wave for a smaller incident angle. The back scattering for an incident angle of $0^\circ$ is about 100 times higher than that for the case of $45^\circ$. When the incident angle is $90^\circ$, the back scattering is also quite high. These results may depend on the higher reflectance in straight opposite direction, which occurs when the light falls perpendicular towards a surface, than for the case of slant incident light, e.g. $45^\circ$. However, influence on the probability of back scattering of particle size and shape is very irregular and is hard to predict. Another discovery, from the three-dimensional figures is that the scattering dependence of the azimuthal angle $\phi$, seems to be much less than that of the deflection angle $\theta$. Not totally negligible, though.

Several interesting facts are shown in the results of the TMMCML-simulations in test 2, using oblate spheroids in different orientations as scattering particles. The most remarkable property is the much lower amounts of reflected light and higher amounts of transmitted light, obtained when the oblate spheroids were used as scattering objects compared to when spheres were used, irrespective of the orientations of the spheroids (see table 4.5). This observation has earlier been described and discussed in case of randomly oriented spheroids. The total amount of absorbed light, which indicates the interaction sites and events of the photons in the medium is also much less in the case of spheroids. The results presented in table 4.5, show that the absorbed fraction of light using spheroids was about half the fraction obtained, when using spheres as scattering particles.

When comparing two simulations with the same orientation of the particles with an uncertainty (normal distribution with a given standard deviation) in the direction in one of them (black stars and filled circles in figure 4.9), the total transmittance and the light transmitted close to origo are higher in the standard deviation case. Far from origo, the non-standard deviation case yields more transmitted light. This is probably a consequence of once the light has been scattered in a quite large angle, it tends to continue in this direction as it is more probable. In the standard deviation case, the photon package will
have a higher probability to hit a spheroid at an angle with lower g-factor \((i.e.\) at an incident angle close to \(90^\circ\)), than in the non-standard deviation case. The probability of scattering in wide angles will therefore be higher and more scattering events result in wide scattering angles. This can lead to that it is more rare that photons are transmitted far from the centre of the sample.

The results of two simulations using spheroids oriented with their symmetry axis in the two directions perpendicular to the incident beam, respectively (squares and triangles in figure 4.9) are almost identical, which is expected due to the axial symmetry of the particle. If the symmetry axis of the spheroids is parallel to the incident light (grey stars in figure 4.9), both the total reflectance and the total transmittance will be higher. This is in agreement with the results from three-dimensional pictures (pages 33-35) of the angular scattering by an oblate spheroid for different incident angles. They show higher values of forward scattering as well as back scattering for an incident angle of \(0^\circ\) than for other angles \((45^\circ\) and \(90^\circ\)).

The fact that a sample with randomly oriented spheroids (filled rhombs in figure 4.9) results in more light transmitted straight forward than for spheroids with their symmetry axis parallel to the incident light (grey stars in figure 4.9) seems rather confusing at first sight, as the highest g-factor is obtained for an incident angle of \(0^\circ\). Nevertheless, taking into consideration that the total amount of transmitted light is lower in the latter case, due to the high probability of back scattering and the facts mentioned before, that once the light has been scattered in quite a large angle, it tends to continue in this direction more probable in the latter case than in the first, this phenomenon can seem reasonable. Even more surprising seems a comparison between the results of the TMMCML-simulations using spheres (small grey strokes in figure 4.9) and spheroids oriented with their symmetry axis in the direction perpendicular to the incident beam, as scattering particles. Results from test 1 and test 2, obtained using a small FORTRAN program, shows \(g = 0.9940\) for the spheres and \(g = 0.9891\) for the spheroids, when the incident angle equals \(90^\circ\). Thus, less light would actually be scattered in the forward direction using spheroids oriented with the symmetry axis perpendicular to the incident beam than in the case with spheres. However, the anisotropy factor, \(g\), describes only the mean value of the cosine of the deflection scattering angle \(\theta\), but not the scattering probability distribution. In fact, the scattering probability of a scattering angle between \(5^\circ\) and \(35^\circ\) is much higher for this kind of spheroids than for spheres, which can be seen in figure 4.8. The incident angle is around \(90^\circ\) only in the first interaction points and causing other directions of the incident angle at later interactions. The explanation concerning samples with no deviations in the orientations of the spheroids is also most likely in this case, due to the high transmission far from origo, using the spheroids as scattering particles (grey stars in figure 4.9).

Comparing the results of test 3, where two different distances \((0.2\ \text{cm and 0.05 cm})\) between the particles in TMMCML-simulations, reasonable results are obtained \(i.e.\) more absorption and less transmission for the shorter particle distance due to more interaction occasions. The higher amounts of reflected light for samples containing spheroids, with larger distance in-between are probably due to higher amounts of light reflected in the lowest boundary, which is transmitted through the sample once more.
5.1.4. Comparison with original Monte Carlo model

When simulating multiple light scattering by executing TMMCML with scattering spheres and the original Monte Carlo program with identical values of the g-factor, the mean distance between two interaction points and the absorption during an interaction, the results show that light distribution is independent of the scattering phase function Henyey-Greenstein versus T-matrix computation for spheres if the photons are scattered many times. The deviation in the distribution of reflected light close to origo (figure 4.10), is mainly caused by photons, not multiple scattered. The small deviations in the transmittance diagram (figure 4.11) might be due to the low number of photons in these areas. The difference in simulation time indicates the more complicated computations of the scattering events, required by the TMMCML program.

5.2. Summary and concluding remarks

The modified programs have been used to simulate the distributions of transmitted and reflected light of samples with a finite extension containing spheroidal scattering particles and the results presented here correspond well with those by others within this field.\(^{25,26,28,30}\)

One purpose of this work was to get a more accurate model for predicting light propagation in human blood. The main disadvantage of the developed model is the large distance between the scattering particles needed due to the assumption of far-field (eq. 2.13). The original Monte Carlo model used, is a conventional way of describing light propagation in tissue\(^{20,31,32}\), in spite of its insufficient description on the microscopic level. Monte Carlo simulations, using the Henyey-Greenstein function, has been proved to be in good agreement to measured values of reflected and transmitted light in human tissue.\(^{33}\)

The most important assumption by the Monte Carlo model is that the light scatters according to the Henyey-Greenstein function.

In the tests made during this work, no agreement have been obtained between the results of the program TMMCML with oblate spheroids as scattering particles and the original Monte Carlo program with corresponding input parameters. One explanation for the case of short distances between the blood cells (normal human blood) might be that this distance is too small to assume that the incident field of one cell is equal to the scattered far-field from the previous cell. Nevertheless, this cannot be the whole truth, as there is no agreement when the distance between the cells is increased (Test 2). One possible reason to this fact is that the original Monte Carlo method and the Henyey-Greenstein phase function maybe are no longer valid in extremely diluted blood, in spite of a sample thickness resulting in multiple scattering. Thus, the modified model would in this case be a better description of the light propagation. The Monte Carlo model might even not be valid for blood at all, in spite of its good agreement for tissue in general. Another reason to the mentioned disagreements could be that the shape deviation between a blood cell and an oblate spheroid might influence the scattering process too much to make such an approximation. This can be due to different distribution of light scattering, different diffraction and occurrence of interference\(^{20}\) or some resonance phenomenon taking place. Finally, the assumptions made in the work such as angular resolution of incident and scattered field, disregarding of the light polarization and neglected dependence of particle light absorption on the angular scattering, might have influenced the results.
Measurements made on slightly diluted blood\(^1\), also indicate higher values of reflectance in proportion to transmittance than ever obtained by TMMCMCL simulations, with corresponding sample sizes and distances between the scattering spheroids. However, no measurements are made at the low concentrations required for accurate T-matrix computations. Nevertheless, some optical measurements on human blood qualitatively agree with the simulations in this work. The transmitted light for instance decrease when the blood cells were transformed from spheroidal into spherical shape\(^3\). This is in line with the results obtained from the simulations (figure 4.5) showing less transmitted light from the sample with spherical particles than from the one containing volume equivalent spheroids. Agreements is also obtained between measurements showing more reflected and less transmitted light with increasing blood flow\(^3\), when the cells are less randomly oriented and the results from simulations with totally randomized versus well aligned spheroids showing more transmitted light for the former (Table 4.5).

### 5.3. Future

As mentioned before, the computer programs, developed during this work, can be used to evaluate the scattering in various directions from large spheroidal particles, as well as the distributions of transmitted and reflected light of a sample containing a lot of such particles in different orientations. These programs can hopefully be of use when such results are needed to interpret measurements and to understand which particle parameters influence the distributions of scattered light. The number of possible shapes and sizes of the scattering particles has increased considerably due to this work, and the application field has thus expanded.

In order to continue the investigations of scattering by blood cells and light distribution in human blood, some possible further works are suggested:

- To establish an explicit expression, \(r(\theta)\) (see figure 3.1), of the shape of the blood cell (figure 3.4) and to obtain a converged solution for such a particle by the program T1. The first part would be possible, modifying an available implicit expression\(^22\), while the convergence probably requires more advanced computers and further programming efforts.

- Another modification of the programs T1 and T2\(^9\), in order to obtain the absorption cross sections for different incident angles and the angular scattering without neglecting the absorption. Measured values of the imaginary refractive index of a blood cell are available\(^23\) and using a compiler with extended precision for complex variables, the necessary modifications should be few.

- Measurements of transmitted and reflected light, from a sample containing blood, or another suspensions with spheroidal particles diluted to such an extension that the assumption of far-field, described earlier, is fulfilled. The idea would be to compare the results obtained by the program TMMCMCL with measured values.
6. Acknowledgements

First of all, I would like to thank my supervisor Annika Nilsson for her good advices, positive attitude to the emerged problems and for always having time for all my questions.

Special thanks to Anders Karlsson at the Department of Electromagnetic Theory, Lund Institute of Technology, who shared his knowledge about the T-matrix method and the belonging computer programs.

Many thanks to Stefan Andersson-Engels for taking part in stimulating discussions, as well as the outlining of this project and to Jörgen Carlsson who helped me, with the large-scaled calculations on his computer.

I am grateful to my family, who has supported me during my work.

Finally, I would like to thank Professor Sune Svanberg, who woke my interest for applications of atomic physics and all the other staff members at the division, for helping me solving different kind of problems and many pleasant conversations.
7. References


APPENDIX A: The T-matrix computer programs

A.1. Computer code listing including comments

The computer code listing of the three modified T-matrix computer programs, includes some comments of the specific modifications. Remarks are inserted in the programs EXTENDT1 and MCDATA where modifications of the T1- and EXTENDT3-programs have been made. When changing program T3 to EXTENDT3, only general modifications have been made and they are all described in the beginning of program EXTENDT3. Original comments in program T1 and T3, which are not valid in the modified versions, are removed.

A.1.1. EXTENDT1

```c
program EXTENDT1

.c Light Scattering by Particles: Computational Methods

.c by P.W. Barber and S.C. Hill

.c copyright (c) 1990 by World Scientific Publishing Co Pte Ltd

.c Modifications 1996 by P. Alsholm and A. Nilsson

.c

calculate the scattering by axisymmetric dielectric objects

.c convergence test over three parameters:

.c nrank : number of terms (matrix order)

c ntheta : number of integration steps

c nm : number of azimuthal modes m

.c inputs: nrank = matrix order

c ntheta = integration steps

c ic = convergence case

.c ntheta ( = 0 ), nrank ( = 1 ), or nm ( = 2 )

.c x = size parameter (ka)

c aovrb = a/b ratio

c cm = complex index of refraction, (real,imag)

.c (imag is positive for absorption)

.c EXTENDT1 : arrays are expanded to nrank = 200 and

c ntheta = 1000 maximum

.c

.c calculate [T] = [B] * [A] = ([A]' * [B]')

.c use transposed matrices to permit the efficient

.c -1

.c [ ] * [ ] operation rather than [ ] * [ ]

.c -1

.c (1) obtain the transposed matrices [A] and [B]

.c -1

.c (2) calculate [T]' = [A]' * [B]'

.c -1

.c (3) transpose [T]' to obtain [T]

c EXTENDT1 :
```

These modifications have been made:

- no imaginary part of refractive index (erased everywhere)
- all real variables are declared real*16
- all variables which have no imaginary part, if the
  refraction index is real, are also declared real*16
- all complex variables which are used before returning
  from procsm are split up in a real and an imaginary part.
- Both parts are declared real*16
- at the end of procsm, the critical region is passed and
  the complex variables can be declared double complex.
- The DEC OSF/1 compiler uses 32 digits for real
  variables(real*16) and (16+16) digits for each part of a
  complex variable(double complex). We have to split the
  complex variables into two real variables.

52
The size of \(a, b, a_2, b_2, \ldots\) are \((2n_{\text{rankmax}})^2\).

- The size of \(c_{\text{mxnrm,ls}}, a_{\text{bl}}, b_{\text{bl}}, f_{\text{gl}}, f_{\text{g2}}\) is \((n_{\text{rankmax}})\).
- The size of \(b_{\text{slcmp}}, h_{\text{ankelre}}, h_{\text{ankelim}}\) is \((n_{\text{rankmax}}+1)\).
- The size of \(\alpha_{\text{sc}}, \omega_{\text{t}}\) is \((n_{\text{theta}_{\text{max}}})\).

The common-blocks have been changed and split up, in order to prevent compiler warnings.

- \(\text{EXTENDT1} \). The refraction index is always real.
- \(\text{cmi}\) is erased.
- \(\text{em} = \text{cmr}\)
- \(\text{den} = \text{cmr}^2\)
- \(\text{em}, \text{den}\) etc. are declared \(\text{real*16}\).

- \(\text{utheta}\) and \(\text{uphi}\) are the amplitudes of the incident field in the theta (parallel) and phi (perpendicular) directions.
- \(\text{utheta} = 1.0q/q\sqrt{2.0q}\)
- \(\text{uphi} = 1.0q/q\sqrt{2.0q}\)

- convergence over \(m\) (\(ic = 2\)) requires \(n_{\text{rank}}\) azimuthal modes, a nonsymmetric orientation, and a file for the matrices.
- \(\text{EXTENDT1} : \text{status} = '\text{unknown}'\) is added in the open arguments, to prevent compiler warnings.

\(n_{\text{rankmax}} = 200\)
\(n_{\text{theta}_{\text{max}}} = 1000\)
\(\pi = 3.1415926535897932384626433832795\)
\(\text{dtr} = \pi/180.q\)
\(\text{rtd} = 180.q/\pi\)

- set the index of refraction
- for an \(\exp(-i\omega t)\) time variation
- \(\text{EXTENDT1} : \text{status} = '\text{unknown}'\) is added in the open arguments, to prevent compiler warnings.
convergence over ntheta (ic = 0) requires
solutions for two values of ntheta.

\begin{verbatim}
if(ic.eq.0) it = 2
do 160 icon = 1, it
.. set the integration points and weighting values..
  call gauss(wt, asc, ntheta, 0, q0, pi2)
.. enter a loop for each azimuthal mode m.
  do 150 im = 1, nm
    .. set m-dependent variables.
    kmv = im - 1
    if(nm.eq.1) kmv = 1
    cmv = real(kmv)
    cm2 = cmv**2
    prodm = 1.0
    if(kmv.eq.0) then
      em = 1.0
    else
      em = 2.0
      quanm = cmv
      prodm = quanm*prodm/2.0
    continue
  end if
  qem = 2.0/qm
  twm = 2.0*cmv
10  continue
.. set indices for matrix compression when n < m.
.. note: useful only when m > 1.
  iij = kmv - 1
  if(iij.lt.0) iij = 0
  ijt = 2*iij
  ns = nrank - iij
  ns2 = 2*ns
.. initialize all matrix elements to zero.
.. EXTENDT1 : Different initialization
.. do iii 1, 2*nrankmax
.. do jjj 1, 2*nrankmax
   a2(iii, jjj) = 0.0
   b2(iii, jjj) = 0.0
  enddo
  enddo
.. enter a loop to integrate over the surface.
.. (theta is the integration variable)
  do 50 ithta = 1, ntheta
    theta = asc(ithta)
    costh = cos(theta)
    sinth = sin(theta)
    .. calculate the associated Legendre functions.
    .. at each integration point.
    .. call genlpg(pmmlg, nrank)
    .. calculate kr and it's derivative.
    .. at each integration point.
    .. current shape descriptor is for a spheroid:
      a < b : oblate, a = 0 : sphere, a > b : prolate.
      dimension = 2a (along the symmetry axis) x 2b.
      All axisymmetric particles can be treated,
      (not only prolates and oblates) but it will be harder
to obtain converged solutions.
      The expression of r(theta) is needed.
      \( q_b = \frac{r(\theta)}{a} \)
      \( dckr = x * \frac{d(q_b)}{d\theta} \)
      \( x = k \times a \) (input parameter)
      2a is the dimension along the symmetry axis.
      2a \textit{is NOT} the dimension along semimajor axis!!
      \begin{align*}
      q_b &= \frac{1.0}{\sqrt{\cos^2 \theta + (aovrb*\sinh)^2}} \\
ckr &= x * q_b \\
dckr &= -x * \cos \theta * \sinh \times (aovrb^2 - 1.0) \times q_b^3 \\
      \end{align*}
    .. calculate the Hankel functions (real argument) and.
    .. Bessel functions (complex argument) at each.
    .. integration point.
    .. EXTENDT1 : two variables, hankelre and hankelim, are used
    .. instead of hankel and hbkre, hbkim instead of hbk.
\end{verbatim}
bslcmp is always real, while refraction index is real. In TL bslcmp was declared complex.

call besh(ckr, hankelre, hankelim, nranki)

call besj(cspr, bslcmp, nranki)
d = dckr*sinth

wtsin = 2*q0*wt(iitheta)*sinth

call besh(ckr, hankelre, hankelim, nranki)

call besj(cspr, bslcmp, nranki)
d = dckr*sinth

wtsin = 2*q0*wt(iitheta)*sinth

enter a loop for each row of the matrix.

h is the imaginary hankel-element-quotum, while it's only used in the a2-calculation (the imaginary part of AJ

dcmt, h, bk, bbk, sa, sb, ckpr are real.

calculate variable combinations.

c = cm*ckr

d = dckr*sinth

wtsin = 2*q0*wt(iitheta)*sinth

calculate the [A]' and [B]' submatrices, e.g.,

[A]' = |K + cm * J| 1|J + cm * K|
     |1 |1 |1 |1 |1 |

Thus, modifications have been made in the a2-expressions (hbkim instead of hbk).

simplification for mirror-symmetric particles:

if (irow+icol) is even then the I and L elements are zero.

if (irow+icol) is odd then the J and K elements are zero.

if(mod(irow+icol),2).ne.0) then
  increment the -(I + cm * L)' submatrix element.

if(kmv.ne.0) then
  b1 = bia+bib
sa = pmrlc1
1
  d = cm*ckr

end if
else
  increment the (J + cm * K)' submatrix element.
a12 = cmr0*prnl1-qem*(crow*ccol)*costh*prnrl0+
col*(crow)*prn0-cl-crown*ccol*prn0cl
a22 = al2*ckr**2
bla = ccol*ccol*bla
blb = crow*crowl*blb
s = ck*(bk-dcn*hm-dcn*crow-ccol)*a12*ckr*(bla-dcn*blb)+qem*d
b2(icol-ijt, irowl-ijt) = b2(iicol-ijt, irowl-ijt)
1
+(sa-(br-hi)*qem*a22*bbk/cm

if(ic.eq.1) then
   do 75 j = 1,nr2
      do 80 i = 1,nr2
      aa(i,j) = a2(i,i,j,j)
      bb(i,j) = b2(i,i,j,j)
   continue
   continue
   end if

continue

end if

if(ic.eq.1) then
   do 80 i = 1,nr2
      do 95 j = 1,nr2
      a(i,j) = cmplx(b2(j,i),a2(j,i))
   continue
   continue
   end if

write(111, aa

end if

if(ic.eq.2.and.mtxsav.eq.2) then
   do 90 ir = 1,ns2
      do 100 jr = 1,ns2
      a(ir,jr) = b(jr,ir)
   continue
   continue
   end if

write(111, aa
The \( T \) matrix is stored in the \( a(*,*) \) array.

if(ic.eq.2.and.mtxsav.eq.1) write(11) a,cmxnrm

1. Compute the angular scattering and efficiencies.
2. call addprc

if(ic.eq.1) then:
  1. Restore the original \([A]^{'} \) and \([B]^{'} \) matrices.
  2. \( nrank \) is set to \( nrank-1 \).

do 110 ii=1,nr2
  1. Do loop over the last row and column of each submatrix.
  2. Reduce \( nrank \) to \( nrank - 1 \).
  3. The \( a2- \) and \( b2- \) submatrices are used instead of \( A \) and \( B \).

110 continue

convergence over \( ntheta \) requires a second solution for a second value of \( ntheta \).

if(ic.eq.0.and.icon.eq.1) then:
  1. \( ntheta \) is set to \( ntheta+4 \).
  2. Write parameters to file. 

write(6,230) nm,nrank,ntheta,ic

end if

stop

format( ' .......................................... ', i, 1 'convergence test- save the matrices', /
  1 ' [T] is written to file 'T', /
  2 ' [A] is written to file 'a', /
  3 ' parameters are in file 'case'. ', /
  4 'nm' ,i8,3x, 'nrank' ,i6,3x, 'ntheta' ,i8,3x, 'ic' ,i12)

end subroutine gauss(wt,asc,n,aa)

calculate abscissas and weights for \( n \)-point Gaussian.
quadrature over integration limits of $aa$ to $bb$

$aa$ = lower integration limit

$bb$ = upper integration limit

$n$ = number of integration points

$asc$ = abscissas

$wt$ = weights

a well-documented program which performs the same function is given in Chapter 4.5 of W.H. Press, B.P. Flannery, S.A. Teukolsky and W.T. Vetterling, Numerical Recipes (Cambridge University Press, Cambridge, 1986)

the results (either program) compare with Table 25.4 in M. Abramowitz and I.A. Stegun, Handbook of Mathematical Functions (National Bureau of Standards, Washington, 1964)

---

implicit real*16 (a-h,o-z)
real*16 wt(n),asc(n)
data pi/3.1415926535897932384626433832795/
data const,tol/.148678816357,1.0d-31/

---

the tolerance ($tol$) may be decreased to $1.0d-10$ or less when double precision variables are used.

---

EXTENDT1: the tolerance ($tol$) is decreased to $1.0d-31$

---

data cl,c2,c3,c4/.125,-.08246286458,-1.824438767/
if(n.eq.1) then
  asc(1) = 1.0d0/sqrt(3.0d0)
  wt(1) = 1.0d0
  return
end if
cn = real(n)
div2 = n/2
pml = n+1

---

pnmlg = associated Legendre function/sin(theta)
sinth = sin(theta), costh = cos(theta)
nc = number of orders (0 to nc-1)

---

calculate associated Legendre functions (argument

---

..
calculate for the special case theta = 0 degrees or 180 degrees - note that for theta = 180 degrees, costh = -1, but theta is set to 0 degrees prior to subroutine entry.

if theta = 0 degrees and m.ne.1 all functions are zero.

if(theta.eq.0.and.kmv.ne.1) then
  do 10 ilg = 1,nc
  pnmllg(ilg) = 0.0
  continue
  return
end if

if(theta.eq.0.and.kmv.eq.1) then
  pnmllg(1) = 0.0
  pnmllg(2) = 1.0
  pla = 3.0*costh
  pnmllg(3) = pla
  pnmllg(4) = plb
  ibeg = 4
else if(kmv.eq.0) then
  pla = sin(x)/x
  byl2 = byl1/x-a
else if(kmv.eq.1) then
  pb = costh*pla
  pnmllg(1) = pla
  pnmllg(2) = pb
  ibeg = 3
else do 20 ilg = 1,kmv
  pnmllg(ilg) = 0.0
  continue
  pla = prod*m*sinth**(kmv-1)
end if

recur upward to obtain all remaining orders.

if(ibeg.gt.nc) return

ncmul = real(2*ibeg-3)
cmm = 2.0

do 30 ilgr = ibeg,nc
  plc = (cmmul*costh*plb-cnm*pla)/cmm
  pnmllg(ilgr) = plc
  pla = plb
  plb = plc
  cmm = cmm+1.0
  ibeg = kmv+3
  end if

implicit real*16 (a-h,o-z)
real*16 hankelre(nc),hankelim(nc)
real*16 bj(201),by(201),t(3)

by(*) calculation - obtain the zeroeth and first order functions.

bj = Bessel function of the first kind
by = Bessel function of the second kind
x = real argument
nc = number of orders (0 to nc-1)
the order of the functions is incremented by
one in the bj(*),by(*) and hankelre/im(*) arrays
arrays are set for nc = 201 (nrank+1) maximum
EXTENDT1 : hankelre and hankelim are used instead
of hankel in T1
obtain the higher order functions by upward recursion

```
do 10 n = 3, nc
    rn = real(n-2)
    by(n) = (2.0*rn+1.0)*by(n-1)/x-by(n-2)
10  continue
```

calculate the scale factor and the functions

```
alpha = a/by(1)
do 40 k = 1, nc
    hankelre(k) = by(k)*alpha
    hankelim(k) = by(k)
40  continue
return
```

```
implicit real*16 (a-h,o-z)
real*16 bslcmp(nc) ,t(3)
do 10 i = nc-1,nc-1,-1
    ri = real(i)
    t(1) = (2.0*ri+1.0)*t(2)/x-t(3)
    t(3) = t(2)
    t(2) = t(1)
10  continue
```

calculate Bessel functions of the first kind

```c
bslcmp = Bessel function of the first kind
```

```c
c = number of orders (0 to nc-1)
c = the order of the Bessel functions is incremented.
```

```c
set the starting order for downward recursion
```

```c
set the starting order for downward recursion
```
alpha = sin(z)/(z*bslcmp(k))
do 30 k = 1,nc
  bslcmp(k) = bslcmp(k)*alpha
30 continue
return
end

subroutine prcssm(n)
  implicit real*16(a~h,o~z)
  double complex a,b
  common /mtxcom/ nr,nri,a(400,400),b(400,400)
  common / mtxcom2/ cmxrnrm(200)
  common/mtxcom3/ a2(400,400),b2(400,400)
  real*16 ls(400) ,are(400,400) ,aim(400,400) ,bre(400,400)
  real*16 bim(400,400), areoldi400, 400),breold(400, 400)
  real*16 bs1cmp(k) = bslcmp(k)*alpha
  continue
  return
  end

calculate \(T' = A' \times B'\) using Gauss-Jordan reduction

---

start reduction of \(A\)
don 60 ii = 1,n
  do jj = 1,n
    are(ii,jj) = b2(ii,jj)
    aim(ii,jj) = a2(ii,jj)
    bre(ii,jj) = b2(ii,jj)
    bim(ii,jj) = 0.q0
  enddo
endo

---

search for the maximum element in the \(i\)th row of \([A]\).

EXTENDTL: \(\text{are},\text{aim},\text{bre},\text{bim}\) are used (\(\text{are} = \text{real}(\text{a})\) etc.)

---

normalize the \(i\)th row of \([A]\) and \([B]\) by \(\text{aijmax}\)

---

.............

---

apply the same row transformations to \([B]\)

do 50 k = 1,n
  if (k.ne.i) then
    aratre = -are(k,jmax)
    aratim = aim(k,jmax)
    do 30 j = 1,n
      if(abs(are(i,j)**2+aim(i,j)**2).gt.0.0) then
        are(k,j) = aratre*are(i,j)-aratim*aim(i,j)
        aim(k,j) = aratim*areold(i,j)+aratre*aim(i,j)
      end if
      continue
    end do
  end if
  continue
  are(k,jmax) = 0.q0
  aim(k,jmax) = 0.q0
endo

---

store row counter in array \(ls(*)\), such that \(ls(*)\) contains the location of the pivot (unity).
calculate the elements of the transform matrix for the polarization vectors

\[
\begin{align*}
\text{C} & \quad \text{calculate scattering angles in the particle frame corresponding to scattering angles in the laboratory frame} \\
\text{C} & \quad \begin{align*}
\text{do} & \quad i = 1, n \\
\text{tang}(i) & = \text{atan2}(r, z) \\
\text{if}(\text{tang}(i) \lt \pi) & \quad \text{tang}(i) = \pi + \text{tang}(i) \\
\end{align*}
\end{align*}
\]

\[
\begin{align*}
\text{C} & \quad \begin{align*}
\text{do} & \quad j = 1, n \\
\text{b}(i, j) & = \text{cmplx}(\text{bre}(i, j), \text{bim}(i, j)) \\
\end{align*}
\end{align*}
\]

\[
\begin{align*}
\text{C} & \quad \text{[T]} \text{ is stored in the b(*,*) array} \\
\text{C} & \quad \text{return}
\end{align*}
\]

subroutine genang(n, tang, pang, cosb, sinb)
c cosine (particle theta orientation angle)
  c sine (particle phi orientation angle)
  c cosine (phi scattering angle in laboratory frame)
  c cosine (theta scattering angle in laboratory frame)
  c pang(i) = phi scattering angle in particle frame

10 continue
return
end subroutine

transform the incident field polarization vector
from the laboratory frame to the particle frame
u1 = cosb(1)*utheta-sinb(1)*uphi
u2 = sinb(1)*utheta+cosb(1)*uphi
end if

clear the result accumulators when ic = 0 or ic = 1 and
the first time the subroutine is entered for ic = 2

if(ic.le.1.or.kmv.eq.0) then
    s = 0.0
    do 10 i = 1,40
        clrtot(i) = 0.0
        continue
    nr2 = 2*nrank
end if

set indices for matrix compression when n < m
note: useful only when m > 1

i1 = kmv-1
if(i1.lt.0) i1 = 0
i1 = 2*i1
ns = nrank-i1
ns2 = 2*ns

calculate the associated Legendre functions for the
incident angle = tang(1) in the particle frame
theta = tang(1)
costh = cos(theta)
sinth = sin(theta)
if(tang(1) .eq.pi) then
    theta = 0.0
    costh = -1.0
end if
calculate the incident field coefficients:

ab1(*) = theta polarization
ab2(*) = phi polarization

note: ab1(*) and ab2(*) used in scattered
field programs are i times ab1(*) and
ab2(*) used in internal field programs
do 20 n = 1,nrank
   if(n.le.ij) go to 20
   cn = real(n)
   n1 = n+1
   cl = 4.80*clnt*n
   c2 = 4.80*cia**n1
   pl = cn*cos**nmllg(n1)-(cn+cmv)*pmllg(n)
   p2 = cmv*pmllg(n1)
   ab1(n-ij) = cl*p2*uu1
   ab1(np-ij) = -cl*p1*uu2
   ab2(n-ij) = c2*p2*uu2
   ab2(np-ij) = c2*pl*uu2
20 continue

   .. the scattered field coefficients = -[T] .
   .. times the incident field coefficients .
   .. (g2(ns1) to g2(ns2)) is -g(omn) .
   .. continue ..............................................
   .. (g2(ns1) to g2(ns2)) is -g(omn) .
   do 40 i = 1,ns2
      ifng2(i) = 0.00
      ifng2(i) = 0.00
      do 30 j = 1,ns2
         ifng2(i) = ifng2(i) - tmat(i,j)*ab1(j)
         ifng2(i) = ifng2(i) - tmat(i,j)*ab2(j)
30 continue
40 continue

c. evaluate the scattered field at each scattering angle .
c. theta = tang(*) and phi = pang(*) in the particle frame .
c. results are accumulated in acans(*,*J for each 
c. azimuthal mode m .
c. acans = kF , where F is a component of the vector .
c. continue ..............................................

do 60 iu = 1,10

c. calculate the associated Legendre functions .
c. at each scattering angle .
c. theta = tang(iu)
   sinth = sin(theta)
   costh = cos(theta)
   if(tang(iu).eq.pi) then
      theta = 0.00
      costh = -1.00
   end if
   call genlgp(pmllg,nrank)

phi = cmv*pang(iu)
   cosp = cos(phil
   sinphi = sin(phil
   do 50 n = 1,nrank
      if(n.le.ij) go to 50
      cn = real(n)
      n1 = n+1
      pl = cn*cos*pmllg(n1)-(cn+cmv)*pmllg(n)
      p2 = cmv*pmllg(n1)
      aa = sinphi*p1
      bb = cosp*phil
      cc = sinphi*p2
      dd = cosp*phil
      cim = cl**(-n1)
      nij = n-ij
      npijt = np-ijt
      continue
50 continue

   .. solve for the theta polarized scattered .
   .. field in the particle frame .
   .. write the scattering results in the laboratory frame .
   .. v = vertical (perpendicular) or phi polarization .
   write(6,100i
      kmv
   do 70 jup = 1,10
      ang = 20.*real(jup-1J
      h = snorm*abs(cosb(jup)*acans(jup,1)+sinb(jup)*acans(jup,2))**2
      v = snorm*abs(cosb(jup)*acans(jup,2)-sinb(jup)*acans(jup,1))**2
      write(6,1101
50 continue
60 continue
70 continue
program EXTENDT3  

c calculates the differential scattering cross section in all  
c directions by projecting the surface of a spherical  
c on to a rectangular coordinate system.  
c thsc: the theta scattering angle, is projected onto the  
c radius r in the rectangular plane  
c r = thsc/180 degrees (r varies from 0 to 1)  
c when r > 1, set scattering to backscatter value  
c phsc: the phi scattering angle, is the azimuthal angle in  
c the spherical and rectangular coordinate systems  

c A.1.2. EXTENDT3

```fortran

if (.not. first) then
  if (abs(l(q0-oldh(jup))/h).le.1.e-03) nconv = nconv+1
  if (abs(l(q0-oldv(jup))/v).le.1.e-03) mconv = mconv+1
end if
oldh(jup) = h
oldv(jup) = v
continue

    calculate the normalized \( \pi a^2 \) extinction \((ex)\),  
    scattering \((sc)\) and absorption \((ab)\) cross sections  
    the \( ex \) calculation can be performed in either  
    the reference frame - use the particle frame  
    \( EXTENDT1\) : imag instead of aimag while double complex is used  
    \( ex = (\text{imag}(acans(1,1))*uu1+\text{imag}(acans(1,2))*uu2) \times 4.q0/x**2 \)

do 80 n = 1,ns
  \( n_r = n + n_r \)
  \( s = s+(abs(fg1(n))*\pi*2+abs(fg1(np))*\pi*2 \)
  \( 1*abs(fg2(n))*\pi*2+abs(fg2(np))*\pi*2 \)^cmxnr(n)
80 continue

sc = s/x**2

ab = ex-sc
if(ab.lt.1.e-08) ab = 0.q0
write(6,120) sc,ab,ex

100 continue

if (nconv.ge.10.and.mconv.ge.10) then
  write(6,*1) *** solution has converged ***
  if (ic.eq.2) then
    open(unit=12, file='case', status='unknown')
    rewind 12
    nt = kmv+1
    write(12,130) nt,nrank,x,aovrb,cmr
    close(unit=12)
    close(unit=11)
  end if
else
  first = .false.
  return
end if
stop

110 format(/,12x,'*** m = ',i3,'
  1 ',d9.4,2(5x,1pe13.4))
120 format(2i4,4f8.4)
130 format(2i4,4f8.4)

end
```

A.1.2. EXTENDT3

program EXTENDT3  

...
start at \((x,y) = (-1,-1)\) and continue in the + x direction.
then increment \(y\) by \(\text{dlt}\) and continue at \((-1,-1+\text{dlt})\).

**Inputs:**
- \(\text{angint}\) = theta orientation in the laboratory frame.
- \(\text{anginp}\) = phi orientation in the laboratory frame.
- \(\text{anglab}\) = angle of the E-field polarization vector
  - in the x-y plane
- \(\text{npnts}\) = number of grid points = \(\text{npnts} \times \text{npnts}\)

**Complex** ----> double complex
real ----> real*16

**Alog** ----> qlog
0 ----> q0 (cmplx(0.d0,0.d0))

**Common-blocks are changed**
open-arguments are changed

**implicit real*16 (a-h,o-z)**
double complex \(tmat, acans\)
common dtr, rtd, pi
common /mtcmax/ nrank, nranki, tmat(400,400)
common /mtcmax2/ cmaxnm(200,201)
common /cmvcom/ nm, kmv
common /cmvcom2/ cmv, twm, prodm
common /tvtcom/ theta, sintheta, costheta
common /uvvcom/ angint, anginp, utheta, uphi, ic
common /uuuvcom2/ thsc
dimension cmx(200), ac(1000)

**set program constants**
dtr = 0.17453292519943

**rtd = 57.2957795131**
pi = 3.14159265358979

 calculation for \(\text{angint} = 0\) or 180 degrees uses \(m = 1\) only.

**calculation for angint not 0 or 180 degrees** (all \(m\)).

**set the do 30 and do 50 loop indices for all \(m\)**.

**calculation for angint not 0 or 180 degrees (all \(m\)).**

**set the do 30 and do 50 loop indices for all \(m\).**

**End if**
\(\text{angint} = \text{angint} \times \text{dtr}\)
\(\text{anginp} = \text{anginp} \times \text{dtr}\)
\(\text{anglab} = \text{anglab} \times \text{dtr}\)
\(\text{tp} = \text{anginp}\)

**utheta and uphi are the amplitudes of the incident field in**
- the theta (parallel) and phi (perpendicular) directions.

**utheta = cos(\text{anglab})**
calculate the scattered field expansion coefficients and the backscatter.

```fortran
uphi = sin(anglab)
calculate the scattered field expansion coefficients and the backscatter.

```
calculate the differential scattering cross section at each grid point - process is an alternate entry to subroutine addprc for scattered field calculations using the previously calculated expansion coefficients.
call prcess
continue
calculate the logarithm of the total differential scattering cross section at (xg,yg) and store in array ac(*)
ac(jg) = qlog10(total)
else
set the differential scattering cross section to the backscatter value for all grid points on or outside the unit circle
ac(jg) = bkscat
end if
increment x value
xg = xg+dlt
continue
write out data for all xg values
(yg yg-dlt) continue
write(9,120) (ac(j),j=1,npnts)
increment y value
yg = yg+dlt
continue
close(unit=9)
stop

format(214,f8.4)
100 format(214,f8.4)
110 format(214,f8.4)
7 'vector in the x-y plane, degrees',/
8 'npnts: number of grid points (npnts x npnts)',/}
end subroutine genlgp(pnmllg,nc)
calculate associated Legendre functions (argument cos(theta)) divided by sin(theta) for each azimuthal mode m generate first two orders by formula and remaining orders by recursion

pnmllg = associated Legendre function/sin(theta) for each order of the associated Legendre functions is incremented by one in the pnmllg(*) array

implicit real*16 (a-h,o-z)
common /cmvcom/ nm,kmv
common /cmvcom2/ cmv,twm,prodm
common /thtcom/ theta,sinth,costh,x
dimension pnmllg(nc)
dtwm = twm+1.q0

calculate for the special case theta = 0 degrees or 180 degrees - note that for theta = 180 degrees, cos(theta) = -1,
but theta is set to 0 degrees prior to subroutine entry
if theta = 0 degrees and m.ne.1 all functions are zero
if(theta.eq.O.and.kmv.ne.l) then
  do ilg = 1,nc
    pnmllg(ilg) = 0.q0
  continue
  return
end if
if theta = 0 degrees and m.eq.1 calculate orders 0,1,2
if(theta.eq.O.and.kmv.eq.1) then
  pnmllg(1) = 0.q0
  pnmllg(2) = 1.q0
  pla = 1.q0
  plb = 3.q0*costh
  pnmllg(3) = plb
ibeg = 4
C do if(theta.ne.0 degrees and m.eq.0) calculate orders 0 and 1.
C else if(kmv.eq.0) then
   pla = 1.0d0/sinth
   plb = costh*pla
   pmnllg(kmv) = pla
   pmnllg2 = plb
   ibeg = 3
C non-zero orders - the associated Legendre function.
C is zero for orders less than the azimuthal mode m.
C else do
   ilg = 1,kmv
   pmnllg(ilg) = 0.0d0
   do 20 ilg = 1,kmv
      pla = prodm*sinth**(kmv-1)
      plb = dtwm*costh*pla
      pmnllg(kmv+2) = plb
      ibeg = kmv+3
   end if
C recur upward to obtain all remaining orders.
C if(ibeg.gt.nc) return
C cmnul = real(2*ibeg-3)
C cnm = 2.0d0
C cnmm = dtwm
C do 30 ilgr = ibeg,nc
   plc = (cmnul*costh*plb-cnmm*pla)/cnm
   pmnllg(ilgr) = plc
   plb = plc
   ibeg = ibeg-2
   cmnul = cmnul-2.0d0
   cnm = cmn+1.0d0
   cnmm = cnmm+1.0d0
30 continue
C return
end subroutine genang
C calculate scattering angles in the particle frame.
C calculate the elements of the transform matrix for the polarization vectors.
calculate the scattered field expansion coefficients and the differential scattering cross section at angles.
tang, pang in the particle frame corresponding to angles thsc, phsc in the laboratory frame

entry at address calculates the incident field expansion coefficients and scattered field expansion coefficients which only need to be calculated once for each m

entry at process uses the previously calculated scattered field expansion coefficients to calculate the differential scattering cross section calculations are performed in the particle frame

implicit real*16 (a-h,o-z)
double complex tmat,abl(400),ab2(400),ci,c1,c2,cm,acans
double complex fg1(400,2011),fg2(400,2011)
common dtr,rtd,pi
c
common /mtxcom/,nrank,nranki,tmat(400,400)
common /mtxcom2/,cmxrm(200,201)
common /cmvcom/,nm,kmv
common /cmvcom2/,cmv,twm,prodm
common /thtcom/,theta,sinth,costh,x
common /uvcom/,angint,anginp,utheta,uphi,ic
common /uvcom2/,thsc
common /fjutc/,total
dimension pnmllg(201),acans(2)
logical first.
data first/.true./

set indices for matrix compression when n < m

ij = kmv-1
if(ij.lt.0) ij = 0
ijt = 2*ij
ns = nrank-ij
ns2 = 2*ns

calculate the associated Legendre functions for the incident angle = thtinc in the particle frame

theta = thtinc
sinth = sinltheta)
costh = cosltheta)
call genang(/lpnmllg,nranki)

calculate the incident field coefficients:

c. ab1(*) = theta polarization

c. ab2(*) = phi polarization

c. note: ab1(*) and ab2(*) used in scattered field programs are j times ab1(*) and ab2(*) used in internal field programs

do 10 n = 1,nrank

ij = kmv-1
if(ij.lt.0) ij = 0
ijt = 2*ij
ns = nrank-ij
ns2 = 2*ns

continue
continue

alternate entry to calculate the differential scattering cross section using the previously calculated scattered field expansion coefficients

clear the result accumulators and calculate the coordinate transformation variables for each new grid point

if (i1.eq.1 or kmv.eq.0) then
    acans(1) = 0.q0
    acans(2) = 0.q0
endif

solve for the theta polarized scattered field in the particle frame

solve for the phi polarized scattered field in the particle frame

normalize the converged results and calculate the total differential scattering cross section (both polarizations) in the particle frame (same as in the laboratory frame)

if (i1.eq.1 or kmv.eq.(nm-1)) then
    total = snorm*abs(acans(1))**2+abs(acans(2))**2
endif

return

end
A.1.3. MCDATA

program MCDATA

.. Light Scattering by Particles: Computational Methods
by P.W. Barber and S.C. Hill
copyright (c) 1990 by World Scientific Publishing Co Pte Ltd

.. Modifications 1996 by P. Alsholm and A. Nilsson

.. The original program T3 calculates the differential
scattering cross section in all directions by projecting
the surface of a spherical surface onto a rectangular
coordinate system. (See comments in the T3-code)

.. In the modified version MCDATA the probability of scattering
in all direction is evaluated for an amount of incident
theta angles. The incident phi angle (anginp) is always zero
Output is written to mc.dat
Comments are included, where MCDATA differs from EXTENDT3

.. Inputs to MCDATA:

  .. nb rang int = number of different theta orientations in the
  .. laboratory frame (number of incident angles)
  .. ([0,dth,90]; dth = 90/(nb rang int-1))
  .. ang lab = angle of the E-field polarization vector
  .. in the x-y plane
  .. nb thsc = number of thsc (theta scattering angle)
  .. ([0,dthsc,180]; dthsc = 180/(nb thsc-1))
  .. nb phsc = number of phsc (phi scattering angle)
  .. ([0,dphsc,180]; dphsc = 180/(nb phsc-1))

.. MCDATA:

.. one loop for all ang int = [0,dth,90]; dth = 90/(nb rang int-1).
.. nb rang int is a new variable
.. algorithm:
.. for ang int = 0,dth,90
  .. bkscat (thsc=180)
  .. for thsc = 0,dthsc,180-dthsc dthsc=180/(nb thsc-1)
  .. if thsc = 0 then

forwscat
else
  for phsc = 0,dphsc,180 dphsc=180/(nb phsc-1)
  ac(th,ph) = probability of scattering in this direction.
  end
end
end
end

.. The precision of the output is increased.
.. The size and dimension of the array ac() is increased.
.. A new commonblock is created containing the variable
.. first in order to reset this variable for every new ang int.
.. first is declared in the main program as well

implicit real*16 (a-h,o-z)
double complex tmat,scans
logical first
common dtr,rtd,pi
common /mtxcom/ nrank,nranki,tmat(400,400)
common /mtxcom2/ cmxncm(200,201)
common /cmvcom/ nm,km
common /cmvcom2/ cmv,twm,prod
common /thtcom/ theta,sinth,costh,x
common /vccom/ angint,anginp,utheta,uphi,ic
common /vccom2/ thsc
common /outcom/ total
common /newblock/ first
dimension cmx(200),ac(361,361)
c
// set program constants

dtr = .017453292519943
rtd = 57.2957795131
pi = 3.14159265358979

.. read from file:
  the T-matrix and normalization constants
.. nrank = number of terms (matrix order)
.. nm = number of azimuthal modes m
.. x = size parameter (ka)
.. MCDATA: new input parameters
.. new variables (dtheta, dphiold, forwscat)
open(unit=12, file='case', status='unknown')
rewind 12
read(12, 100) nm, nrank, x
nrank1 = nrank + 1
write(6, 110)
write(6, '*') 'enter nbrangint, anglab, nbrthsc, nbrphsc'
read(5, *) nbrangint, anglab, nbrthsc, nbrphsc
dtheta = pi/(nbrthsc-1)
dphiold = pi/(nbrphsc-1)
forwelem = 2*pi*(1.0 - cos(dtheta/2))

... calculation for angint = 0 or 180 degrees uses m = 1 only...
... read the stored arrays once to skip over the T-matrix...
... and normalization constants for m = 0...

MCDATA:
... one loop (80) for each incident angle...
... nbrangint = 1 is used for spheres (angint doesn't matter)...
... nbrangint <> 1, angint is uniformly distributed [0, 90]...
... the t-file is opened and closed inside each loop...
... angint is written to stdout...
... first is set to 'true' in the beginning of each loop...

open(unit=9, file='mc.dat', status='unknown')
rewind 9
do 80 langint = 1, nbrangint
  if (nbrangint.eq.1) then
    angint = 0.0
  else
    angint = 90.0*(langint-1)/nbrangint-1
  end if
  anginp = 0.0
write(6, '(*') (langint)
open(unit=11, file='t', form='unformatted', status='unknown')
rewind 11
first = .true.
if (langint.eq.0.0) then
  nl = 2
  nh = 2
  ic = 1
read(11) tmat, cmx
else
  ... calculation for angint not 0 or 180 degrees (all m)...
  ... set the do 30 and do 50 loop indices for all m...
end if

... the theta (parallel) and phi (perpendicular) directions...
... the utheta and uphi are the amplitudes of the incident field in...
... coefficients and the backscatter...
... enter a loop for each azimuthal mode m...
... read the T-matrix and normalization constants for each m...

... do 30 im = nl, nh
... kmv = im-1
cmv = real(kmv)
prodm = im-1, nh
if (kmv.gt.0) then
  quanm = kmv
  do 10 i = 1, kmv
    quanm = quanm + prodm + 1.0
    prodm = quanm*prodm/2.0
  end do
  continue
else
  kmv = kmv-1
  if (ij.gt.0) then
    j1 = kmv-1
    do 10 i = 1, j1
      j1 = j1
      prodm = prodm + 1.0
    end do
    continue
endif
read(11) tmat, cmx
... do 20 k = 1, ns
... cmxnorm(k, im) = cmx(k)

... ...
calculate the scattered field expansion coefficients
and sum the backscatter over all azimuthal modes

call addprc

continue

MCDATA: no logarithm.

 calculate the differential scattering cross section

MCDATA: enter a loop to vary thsc values

do 70 ig = 1,nbrthsc-1

MCDATA: enter a loop to vary phsc values

do 60 jg = 1,nbrphsc

MCDATA: the combination thsc = 0.0 and phsc <> 0.0 is not necessary.

if (ig.ne.1.or.jg.eq.1) then
  thsc = pi*(ig-1)/(nbrthsc-1)
  phsc = pi*(jg-1)/(nbrphsc-1)
else
  thsc = 0

rotate the phi orientation angle anginp with phsc so that
the scattering at the angle phsc can be obtained in
the phi = zero degree plane in the laboratory frame

anginp = tp-phsc
if(anginp.lt.0.0) anginp = anginp+2.0*pi

set m-dependent constants

kmv = im-1
cmv = real(kmv)
prodm = 1.0
if(kmv.gt.0.0) then
  quann = cmv
  do 40 i = 1,kmv
    quann = quann+1.0*prodm
    prodm = quann*prodm/2.0
  continue
end if

twm = 2.0*cmv

if thsc = 0 : forward scattering
  forwscat = total
else : justify the intensity according to the area
  with the factor dphiold/dphinew
  dphiold: the phi angle between two phsc
  dphinew: the phi angle for a surface element
  with an area equal to the area of
  the forward (and back) scattering
  surface element (forwelem)
  dphinew is theta-dependent
  the factor 2 is multiplied to ac(*,*) while this value
  symbolizes the scattering of both phi and 360-phi
  this factor is not used for phi=0, phi=180 or theta=0
  if (ig.eq.1) then
    forwscat = total
  else if (anginp.gt.0.0) then
    i = ig - 1
    dphinew = forwelem/(cos((i-0.5)*dtheta) -
    cos((i+0.5)*dtheta))
    ac(ig,jg) = 2*total*dphiold/dphinew
  else
    if (jg.eq.1.or.jg.eq.nbrphsc) then
      ac(ig,jg) = ac(ig,jg)/2
    else
      ac(ig,jg) = ac(ig,jg)
  end if
MCDATA: increment phsc value
continue

MCDATA: increment thsc value
continue

THIS PART IS ONLY FOR MCDATA:
ac(*,*) contains differential scattering cross-sections (dscs) in different scattering angles.

bkscat and forwscat symbolizes back- and forward scattering.

sum : summation of all dscs

divide ac by sum to get a probability of scattering in different scattering angles.

check is used to be sure that the probability of scattering in all directions together is exactly 1.0

sum = bkscat
do i = 2,nbrthsc-1
   do j = 1,nbrphsc
      sum = sum + ac(i,j)
   enddo
enddo

sum = sum + forwscat
bkscat = bkscat/sum
val = bkscat

forwscat = forwscat/sum + val

write(9,120) bkscat
write(9,120) (ac(i,j),j=1,nbrphsc)
continue

format(2i4,f8.4)
format (' ....................................................',I,
1 calculate scattered intensity in all directions .',/
2 ' output is written to mc.dat .',/
3 'nbrangint: number of different incident theta angles',/
4 'anglab: angle of the E-field polarization',/
5 'nbrthsc: number of scattering angles (theta)',/
6 'nbrphsc: number of scattering angles (phi)',/
7 ' the precision of the ouput is increased .',/
8 ' calculate associated Legendre functions (argument cos(theta) divided by sin(theta) for each azimuthal mode m generated first two orders by formula and remaining orders by recursion

nc = number of orders (0 to nc-1)
the order of the associated Legendre functions is incremented by one in the pmmlg(*) array

pmmlg = associated Legendre function/sin(theta)
increased by one in the pmmlg(*) array

calculate associated Legendre functions (argument cos(theta)) divided by sin(theta) for each azimuthal mode m
generate first two orders by formula and remaining orders by recursion

nc = number of orders (0 to nc-1)
the order of the associated Legendre functions is incremented by one in the pmmlg(*) array

implied real*16 (a-h,o-z)
common /cmvcom/ nm,km
common /cmvcom2/ cmv,twm,prodm
common /thtcom/ theta,sinth,costh,x
dimension pmmlg(nc)
dtwm = twm+1.0

write(9,120) bkscat
forwscat = 1.0-ac(nbrthsc-1,nbrphsc)
write(9,120) (ac(i,j),j=1,nbrphsc)
enddo

format(e14.9)
end subroutine genlgp(pnmlg,nc)
calculate for the special case \( \theta = 0 \) degrees or 180 degrees - note that for \( \theta = 180 \) degrees, \( \cos \theta = -1 \), but \( \theta \) is set to 0 degrees prior to subroutine entry

if \( \theta = 0 \) degrees and \( m \neq 1 \) all functions are zero .

if \( \theta = 0 \) degrees and \( m = 1 \) calculate orders 0,1,2

else if \( \theta = 0 \) degrees and \( m = 0 \) calculate first two .

nonzero orders - the associated Legendre function is zero for orders less than the azimuthal mode \( m \).

else

do 20 ilg = 1,kmv

continue

end if

calculate the theta scattering angles in the particle frame .

\[ r = \sqrt{x^2 + y^2} \]
\[ \tan \theta = \frac{y}{x} \]
if(z.ne.0.o.or.r.ne.0.o) tang = atan2(r,z)

. calculate the phi scattering angles in the particle frame
. .
. pang = 0.o
if(x.ne.0.o.or.y.ne.0.o) tang = atan2(y,x)
if(pang.lt.0.o) pang = 2.o*pi-pang
.
. calculate the elements of the transform matrix
. matrix for the polarization vectors
. .
. the transform matrix is needed only for the
. . incident polarization vector i case n = 1
.
.
if(n.eq.1) then
  cosb = -cosp
  sinb = sinp
end if
return
end subroutine addprc

. calculate the scattered field expansion coefficients and
. the differential scattering cross section at angles
. tang, pang in the particle frame corresponding to
. angles thsc, phsc in the laboratory frame
. entry at addprc calculates the incident field expansion
. coefficients and scattered field expansion coefficients
. which only need to be calculated once for each m
. entry at prcess uses the previously calculated scattered
. field expansion coefficients to calculate the
. differential scattering cross section
. calculations are performed in the particle frame
.
. MCDATA: the variable first can be set in the main program
. a new commonblock for the variable first is created
. first is declared in another way
.
implicit real*16 (a-h,o-z)
double complex tmat,ab1(400),ab2(400),ci,c1,c2,cim,acans(2)
double complex fg1(400,201),fg2(400,201)
logical first
common /dtr,rtd,pi
common /mtxcom/ nrank,nranki,tmat(400,400)
common /mtxcom2/ cmonrm(200,201)
common /cmvcom/ nm,knv
common /cmvcom2/ cmv,twm,prod
common /thtcom/ theta,sinth,costh,x
common /uvccom/ angint,anginp,utheta,uphi,ic
common /uvccom2/ thsc
common /output/ total
common /newblock/ first
dimension pnmllg(201)
.
. on first entry set the constants and:
. (1) calculate the incident field direction thtinc
. in the particle frame
. (2) transform the incident field polarization
. vector from the laboratory frame to the
. particle frame
. uu1 = theta component and uu2 = phi component
. (3) set the backscatter angle thsc to 180 degrees
.
if(first) then
  ci = (0.o,0.o)
  nr2 = 2*nrank
  snorm = 1.q0/((pi*x)**2)
  thsc = 0.q0
  call genang1(tang,pang,cosp,sinp)
  thtinc = tang
  uu1 = cosp*utheta-sinp*uphi
  uu2 = sinb*utheta-cosp*uphi
  thsc = pi
  first = .false.
end if

. set indices for matrix compression when n < m
. note: useful only when m > 1
.
ij = kmv-1
if(ij.lt.0) ij = 0
ijt = 2*ij
ns = nrank-1

.. calculate the associated Legendre functions for the
.. incident angle = thtinc in the particle frame
.. theta = thtinc
.. sinth = sin(theta)
.. costh = cos(theta)
.. call genlgp(pnmllg,ns2)
.
.. calculate the incident field coefficients:
.. ab1(*) = theta polarization
.. ab2(*) = phi polarization
.. note: ab1(*) and ab2(*) used in scattered
field programs are \( j \) times \( ab1(*) \) and \( ab2(*) \) used in internal field programs.

do 10 \( n = 1,nrank \)
   if(\( n.le.ij \)) go to 10
   \( np = n+nrank \)
   \( cn = real(n) \)
   \( nl = n+1 \)
   \( cl = 4.*cn^{*}n \)
   \( c2 = 4.*cn^{*}n \)
   \( p1 = cmv*cos(th)^{*}pnmllg(nl)-(cn+cmv)^{*}pnmllg(n) \)
   \( p2 = cmv*pnmllg(nl) \)
   \( abl(n-ij) = cl*p2*uu1 \)
   \( abl(np-ijt) = -c2*p1*uu1 \)
   \( ab2(n-ij) = -cl*p1*uu2 \)
   \( ab2(np-ijt) = c2*p2*uu2 \)

continue

the scattered field coefficients = \(-[T]\) times \(\tau\). store the coefficients for all azimuthal modes.

fg1(ns+1) to fg2(ns2) is \(-g(omn)\).

do 30 \( i = 1,ns2 \)
   \( fg1(i,kmv+1) = (0.0,0.0,0.0) \)
   \( fg2(i,kmv+1) = (0.0,0.0,0.0) \)
   do 20 \( j = 1,ns2 \)
   \( fg1(i,kmv+1) = fg1(i,kmv+1)-tmat(i,j)^{*}ab1(j) \)
   \( fg2(i,kmv+1) = fg2(i,kmv+1)-tmat(i,j)^{*}ab2(j) \)

continue

continue

alternate entry to calculate the differential scattering cross section using the previously calculated scattered field expansion coefficients entry process

calculate the associated Legendre functions.

clear the result accumulators and calculate the coordinate transformation variables for each new grid point.

if(ic.eq.1.or.kmv.eq.0) then
   \( acans(l) = (0,0,0,0) \)
   \( acans(2) = (0,0,0,0) \)
   call genang(2,tang,pang,cosb,sinb)
end if

set indices for matrix compression when \( n < m \).

note: useful only when \( m > 1 \).
c . solve for the phi polarized scattered field in the particle frame .
c . \( f_g(npijt) \) is \(-g_{\text{omn}}\) per (3.12b) .
c

\[
\begin{align*}
\text{acans}(2) &= \text{acans}(2) - c_i m_l f_{g1}(nij,kmv+1)^*cc \\
&+ f_{g2}(nij,kmv+1)bb + c_i f_{g2}(npijt,kmv+1)dd) + c_m c_n r_n(m(nij,kmv+1) \\
\text{continue}
\end{align*}
\]

40

\[
\begin{align*}
\text{normalize the converged results and calculate the total differential scattering cross section (both polarizations)} \quad &. \\
\text{in the particle frame (same as in the laboratory frame) .}
\end{align*}
\]

\[
\begin{align*}
\text{if(ic.eq.1.or.kmv.eq.(nm-1)) then} \\
\text{total} &= \text{snorm} \cdot \text{abs(acans(1))}^2 + \text{abs(acans(2))}^2 \\
\text{end if} \\
\text{return} \\
\text{end}
\end{align*}
\]
A.2. Flow charts

A.2.1. EXTENDT1

```
A.2. Flow charts

A.2.1. EXTENDT1
```

set program constants
(nrank,ntheta,case,ka,a/b,n)

case=2
case=1,0

angint=\(\pi/4\)
angint=0
anginp=\(\pi/4\)
anginp=0
m azimuthal modes
one azimuthal mode

gauss (calculates the integration-points and their weights)

set m-dependent variables

\[ A(*,*) = 0, B(*,*) = 0 \]

For each integration-point:

- GENLGP calculates the associated Legendre-function
- BESH/BESJ calculates the Hankel/Bessel-function

\[ f \text{ and } g \text{ are functions of Hankel, Bessel, Legendre and trigonometric functions} \]

f for row = 1, 2*nrank
for col = 1, 2*nrank
\[ A(row,col) = A(row,col) + f(itheta) \]
\[ B(row,col) = B(row,col) + g(itheta) \]

calculate normalization factor

case=2
\[ \text{save } A,B \]

PRCSSM (calculates \( T = A^{-1}B \))

\[ \text{save } T \]

ADDPRC (calculates \( \sigma \) and angular scattering)
(checks convergence (not the first time))

\[ \text{converged or } m >= nrank \]
\[ \text{no} \rightarrow \text{first} \rightarrow \text{second} \]
\[ \text{yes} \rightarrow \text{first} \rightarrow \text{second} \]

Write result and show if a converged solution is obtained
A.2.2. EXTENDT3

set program constants

\[ \begin{align*} 
\text{theta} & = 0 \text{ or } \pi \\
\text{m} & = 1 \quad \text{else} \\
\text{m} & = nm
\end{align*} \]

for \( \text{im} = 1, m \) read T-matrix values and normalization constants

\[ \text{calculate the logarithm of ADDPRC GENLGP} \]
\[ \text{the normalized scattering (calculates the associated cross section coefficients, once for each m) Legendre function} \]
\[ \text{in back scatter direction} \]
\[ \text{in each PRCSSM} \]
\[ \text{calculates the differential scattering cross sections, once angles, corresponding to the reference frame} \]

for each grid point \( \text{theta and phi are computed} \)

The expansion coefficients are already computed. ADDPRC doesn’t need to be entered.

write the logarithm of the normalized differential scattering cross sections in file t3.dat as described in chapter 3.
A.2.3. MCDATA

set program constants

for iangint = 1, nangint

\[ \text{anginp} = 0 \]
\[ \text{angint} = \pi/2 \times \text{iangint}/(\text{nangint}-1) \]
\[ = 0 \text{ if } \text{nangint} = 1 \text{ (sphere)} \]

angint = 0 or \( \pi \)

\[ m = 1 \]

else

\[ m = nm \]

for im = 1, m

read T-matrix values and normalization constants

compute the differential scattering cross section in back scatter direction

ADDPRC (calculates the scattering expansion coefficients, once for each m)

GENLGP (calculates the associated Legendre function)

PRCSSM (calculates the differential scattering cross sections, once for each scattering angle)

GENANG (calculates the scattering angles, corresponding to the reference frame)

for each grid point

theta and phi are calculated

The expansion coefficients are already evaluated. ADDPRC does not need to be entered.

compute the scattering probability in all grid points

write the probability values in file mc.dat as described in chapter 3.
**APPENDIX B: The Monte Carlo computer programs**

**B.1. Computer code listing**

**B.1.1. BOUNMCML**

**B.1.1.1. mcm1.h**

```c
/* *************************************************************/
/* Monte Carlo simulation of photon distribution in */
/* multi-layered turbid media in ANSI Standard C. */
/* ***/
/* Starting Date: 10/1991. */
/* Current Date: 6/1992. */
/* Lihong Wang, Ph. D. */
/* Steven L. Jacques, Ph. D. */
/* Laser Biology Research Laboratory - 17 */
/* M.D. Anderson Cancer Center */
/* University of Texas */
/* 1515 Holcombe Blvd. */
/* Houston, TX 77030 */
/* USA */
/* */
/* This program was based on: */
/* (1) The Pascal code written by Marleen Keijzer and */
/* Steven L. Jacques in this laboratory in 1989, which */
/* deals with multi-layered turbid media. */
/* (2) Algorithm for semi-infinite turbid medium by */
/* S.A. Prahl, M. Keijzer, S.L. Jacques, A.J. Welch, */
/* SPIE Institute Series Vol. IS 5 (1989), and by */
/* A.N. Witt, The Astrophysical journal Supplement */
/* Series 35, 1-6 (1977). */
/* */
/* Major modifications include: */
/* . Conform to ANSI Standard C. */
/* . Removal of limit on number of array elements, */
/* because arrays in this program are dynamically */
/* allocated. This means that the program can accept */
/* any number of layers or gridlines as long as the */
/* memory permits. */
/* . Avoiding global variables whenever possible. This */
/* program has not used global variables so far. */
/* . Grouping variables logically using structures. */
/* . Top-down design, keep each subroutine clear & */
/* short. */
/* . Reflectance and transmittance are angularly */
/* resolved. */
/* ***/
/* Modifications 1996 by P. Alsholm and A. Nilsson including */
/* lateral boundaries. */
/* ***/
/* General Naming Conventions: */
/* Preprocessor names: all capital letters, */
/* e.g. #define PREPROCESSORS */
/* Globals: first letter of each word is capital, no */
/* underscores, */
/* e.g. short GlobalVar; */
/* Dummy variables: first letter of each word is capital, */
/* and words are connected by underscores, */
/* e.g. void NiceFunction(char Dummy_Var); */
/* Local variables: all lower cases, words are connected */
/* by underscores, */
/* e.g. short local_var; */
/* Function names or data types: same as Globals. */
/* ***/
/* Dimension of length: cm. */
/* ***/
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <stddef.h>
#include <time.h>
#include <string.h>
#include <ctype.h>
#define PI 3.1415926 /* Critical weight for roulette. */
#define WEIGHT 1E-4
```

83
#define CHANCE 0.1 /* Chance of roulette survival. */
#define STRLEN 256 /* String length. */

#define Boolean char
#define SIGN(X) ((X)>=0 ? 1:-1)
/***************** Stuctures **************************/
/*
** Structure used to describe a photon packet.
**
typedef struct {
  double x, y, z; /* Cartesian coordinates. [cm] */
  double ux, uy, uz; /* directional cosines of a photon. */
  double w; /* weight. */
  Boolean dead; /* 1 if photon is terminated. */
  short layer; /* index to layer where the photon */
    /* packet resides. */
  double g; /* current step size. [cm]. */
  double sleft; /* step size left. dimensionless [cm]. */
} PhotonStruct;
****
/* Structure used to describe the geometry and optical */
/* properties of a layer. */
/* z and z1 are z coordinates for the upper boundary */
/* and lower boundary respectively. */
/*
** cos_crit0 and cos_crit1 are the cosines of the */
** critical angle of total internal reflection for the */
** upper boundary and lower boundary respectively. */
/* cos_critxy is the cosine of the critical angle of */
** total internal reflection for the boundary to the */
** surrounding glass. */
/* They are set to zero if no total internal reflection */
/* exists. */
/* They are used for computation speed. */
****

typedef struct {
  double z0, z1; /* z coordinates of a layer. [cm] */
  double n; /* refractive index of a layer. */
  double mua; /* absorption coefficient. [1/cm] */
  double mus; /* scattering coefficient. [1/cm] */
  double g; /* anisotropy. */
  double cos_crit0, cos_crit1, cos_critxy;
} LayerStruct;
/**
* Input parameters for each independent run.
*
* z and r are for the cylindrical coordinate system. [cm]
* a is for the angle alpha between the photon exiting
* direction and the surface normal. [radian]
* The grid line separations in z, r, and alpha
directions are dz, dr, and da respectively. The numbers
* of grid lines in z, r, and alpha directions are
* nz, nr, and na respectively.
* The member layersspecs will point to an array of
* structures which store parameters of each layer.
* This array has (number_layers + 3] elements. One
* element is for a layer.
* The layers 0 and (num_layers + 1) are for top ambient
* medium and the bottom ambient medium respectively.
* The layer (num_layers + 2) is for the surrounding glass.
****
typedef struct {
  char out_fname[STRLEN]; /* output file name. */
  char out_fformat; /* output file format. */
  /* 'A' for ASCII, */
  /* 'B' for binary. */
  long num_photons; /* to be traced. */
  double Wth; /* play roulette if photon */
    /* weight < Wth. */
  double dz; /* z grid separation. [cm]. */
  double dr; /* r grid separation. [cm]. */
  double da; /* alpha grid separation. */
  /* [radian] */
  short nz; /* array range 0..nz-1. */
  short nr; /* array range 0..nr-1. */
  short na; /* array range 0..na-1. */
  short num_layers; /* number of layers. */
  double x_size, y_size; /* dimension of the sample. */
} InputStruct;
****
/* Structures for scoring physical quantities. */
/* z and r represent z and r coordinates of the */
/* cylindrical coordinate system. [cm]. */
/* a is the angle alpha between the photon exiting */
* direction and the normal to the surfaces. [radian]
typedef struct {
    double Rsp;  /* specular reflectance. [-] */
    double * Rd_ra; /* 2D distribution of diffuse */
    /* reflectance. [1/(cm2 sr)] */
    double * Rd_r; /* 1D radial distribution of diffuse */
    /* reflectance. [1/cm2] */
    double * Rd_a; /* 1D angular distribution of diffuse */
    /* reflectance. [1/sr] */
    double Rd; /* total diffuse reflectance. [1] */
    double ** A_rz; /* 2D probability density in turbid */
    /* media over r & z. [1/cm3] */
    double * A_z; /* 1D probability density over z. */
    /* [1/cm] */
    double * A_l; /* each layer's absorption */
    /* probability. [-] */
    double A; /* total absorption probability. [-] */
} OutStruct;

B.1.1.2. mcmlmain.c

/******************************************************************************/
/* main program for Monte Carlo simulation of photon                      */
/* distribution in multi-layered turbid media.                           */
/******************************************************************************/

#define THINKCPROFILER 0
/* GNU cc does not support difftime() and CLOCKS_PER_SEC. */
#define GNUCC 0

#include <profile.h>
#include <console.h>
#include "mcml.h"

FILE *GetFile(char *);
short ReadNumRuns(FILE*);
void ReadParm(FILE*, Inputstruct *);
void CheckParm(FILE*, Inputstruct *);
void InitOutputData(Inputstruct, Outstruct *);
void FreeData(Inputstruct, Outstruct *);
double Rspecular(Layerstruct *);
void LaunchPhoton(double, Layerstruct *, PhotonStruct *);
void HopDropSpin(Inputstruct *, PhotonStruct *, Outstruct *);
void sumScaleResult(Inputstruct, Outstruct *);
void WriteResult(Inputstruct, Outstruct, char *);

/******************************************************************************/
/* Routine prototypes for dynamic memory allocation and                   */
/* release of arrays and matrices.                                       */
/* Modified from Numerical Recipes in C.                                 */
/******************************************************************************/
double ** AllocVector(short, short);
double ** AllocMatrix(short, short, short, short);
void FreeVector(double *, short, short);
void FreeMatrix(double **, short, short, short);
void nrerror(char *);

short If F = 0, reset the clock and return 0.
short If F = 1, pass the user time to Msg and print Msg on
    screen, return the real time since F=0.
short If F = 2, same as F=1 except no printing.
short Note that clock() and time() return user time and real
time respectively.
short User time is whatever the system allocates to the
    running of the program;
short real time is wall-clock time. In a time-shared system,
Ll!~y
not
be
the
same.

clock() only hold 16 bit integer, which is about 32768
clock ticks.

```c
#ifndef GNUCC
return(0);
#else
static clock_t ut0; /* user time reference. */
static time_t rt0; /* real time reference. */
double secs;
char s[STRLEN];
if(F==0)
    ut0 = clock();
    rt0 = time(NULL);
    return(0);
else if(F==1) {
    secs = (clock() - ut0)/(double)CLOCKS_PER_SEC;
    if(secs<0) secs=0; /* clock() can overflow. */
sprintf(s, "User time: %8.0lf sec = %8.2lf hr. %s\n", secs, secs/3600.0, Msg);
    puts(s);
    strcpy(Msg, s);
    return(difftime(time(NULL), rt0));
} else if(F==2) return(difftime(time(NULL), rt0));
#endif

/***********************************************************/
* Print the current time and the estimated finishing time. *
* Pl is the number of computed photon packets. *
* Pt is the total number of photon packets. *
/***********************************************************/
void PredictDoneTime(long Pl, long Pt)
{
    time_t now, done_time;
    struct tm *date;
    char s[80];
    now = time(NULL);
    date = localtime(&now);
    strftime(s, 80, "%H:%M %x", date);
    printf("Now \%s, \%");
    done_time = now +
        (time_t) (PunchTime(2,"")*(Pt-P1)/(double)Pt);
    date = localtime(&done_time);
    strftime(s, 80, "%H:%M %x", date);
    printf("End \%s\n", s);
}

/***********************************************************/
* Report estimated time, number of photons and runs left *
* after calculating 10 photons or every 1/10 of total *
* number of photons. *
* Num_Runs is the number of runs left. *
* Pl is the index to the current photon, counting down. *
* Pt is the total number of photons. *
/***********************************************************/
void ReportStatus(short Num_Runs, long Pl, long Pt)
{
    if(Pt-Pi == 10 || Pt*10%Pt == 0 && Pi != Pt) {
        printf("%ld photons & %hd runs left, ", Pl, Num_Runs);
    }
    PredictDoneTime(Pt-Pi, Pt);
}

/***********************************************************/
* Report time and write results. *
/***********************************************************/
void ReportResult(InputStruct In_Parm, OutStruct Out_Parm)
{
    char time_report[STRLEN];
    strcpy(time_report, " Simulation time of this run.");
    PunchTime(1, time_report);
    SumScaleResult(In_Parm, &Out_Parm);
    WriteResult(In_Parm, Out_Parm, time_report);
}

/***********************************************************/
* Get the file name of the input data file from the *
* argument to the command line. *
/***********************************************************/
void GetFnameFromArgv(int argc, char * argv[], char * input_filename)
{
    if(argc>=2) { /* filename in command line */
        strcpy(input_filename, argv[1]);
    }
}
```
else
    input_filename[0] = '\0';
}

/************************************************************************
* Execute Monte Carlo simulation for one independent run.
*****
**
void DoOneRun(short NumRuns, InputStruct *In_Ptr)
{
    register long i__photon;
    /* index to photon. register for speed.*/
    OutStruct out__parm;    /* distribution of photons.*/
    Photonstruct photon;
#endif
    InitProfile(200,200);
    cecho2file("prof.rpt",0, stdout);
#if THINKCPROFILER
    register for speed.*/
    InitOutputData(*In_Ptr, &out__parm);
#endif
    out__parm.Rsp = Rspecular(In_Ptr->layerspecs);
    i__photon = In_Ptr->num__photons;
    PunchTime(0, " ");
do {
    ReportStatus(NumRuns, i__photon, In_Ptr->num__photons);
    LaunchPhoton(out__parm.Rsp, In_Ptr->layerspecs, &photon);
    do HopDropSpin(In_Ptr, &photon, &out__parm);
        while (!photon.dead);
    while(--i__photon);
    #if THINKCPROFILER
    B.l.l.3. mcmlio.c
    #endif
    ReportResult(*In_Ptr, out__parm);
    FreeData(*In_Ptr, &out__parm);
} /*
 * The argument to the command line is filename, if any.
 * Macintosh does not support command line.
******/
char main(int argc, char *argv[])
{
    char input_filename[STRLEN];
    FILE *input_file__ptr;
    short num_runs;    /* number of independent runs. */
    InputStruct in__parm;
    ShowVersion ( "Version 1.0, 1992" );
    GetFnameFromArgv(argc, argv, input_filename);
    input_file__ptr = GetFile(input_filename);
    CheckParm(input_file__ptr, &in__parm);
    num_runs = ReadNumRuns(input_file__ptr);
    while(num_runs--){
        ReadParm(input_file__ptr, &in__parm);
        DoOneRun(num_runs, &in__parm);
    }
    fclose(input_file__ptr);
    return (0);
}

B.1.1.3.  mcmlio.c

/************************************************************************
* Input/output of data.
*****
**
#include "mcm.h"

/************************************************************************
* Structure used to check against duplicated file names.
*****
**
struct NameList {
    char name[STRLEN];
    struct NameList * next;
};

typedef struct NameList NameNode;
typedef NameNode * NameLink;

/************************************************************************
* Center a string according to the column width.
**CenterStr**

```c
#define COLWIDTH 80

FILE *GetFile(char *Fname) {
    FILE *file = NULL;
    Boolean firsttime = 1;
    do {
        if (firsttime && Fname[0] == '\0') { /* use the filename from command line */
            firsttime = 0;
            return(file);
        } else { /* use input from command line */
            printf("Input filename (or . to exit): ");
            scanf("%s", Fname);
            firsttime = 0;
            if (strlen(Fname) == 1 && Fname[0] == '.')
                exit(1); /* exit if no filename entered */
            file = fopen(Fname, "r");
            while (file == NULL);
            return(file);
        }
    }
    CenterStr(COLWIDTH, "Lihong Wang, Ph. D.", str);
    puts(str);
    CenterStr(COLWIDTH, "Steven L. Jacques, Ph. D.", str);
    puts(str);
    CenterStr(COLWIDTH, "Laser Biology Research Laboratory - 17", str);
    puts(str);
    CenterStr(COLWIDTH, "M.D. Anderson Cancer Center", str);
    puts(str);
    CenterStr(COLWIDTH, "University of Texas", str);
    puts(str);
}
```
void KillChar(size_t i, char * Str)
{
    size_t sl = strlen(Str);
    for(; i<sl; i++) Str[i] = Str[i+1];
}

Boolean CheckChar(char * Str)
{
    Boolean found = 0; /* found bad char. */
    size_t sl = strlen(Str);
    while (i<sl)
    {
        if (Str[i]<0 || Str[i]>255)
            nrerror("Non-ASCII file
");
        else if(!isspace(Str[i]))
            i++;
        else {
            found = 1;
            KillChar(i, Str);
            sl--;
        }
    }
    return (found);
}

Boolean CommentLine(char * Buf)
{
    size_t spn, cspn;
    spn = strspn(Buf, " \	");
    cspn = strcspn(Buf, "#n");
    if(spn == cspn) /* comment line or space line. */
        return(1);
    else /* the line has data. */
        return(0);
}

char * FindDataLine(FILE * File_Ptr)
{
    char buf[STRLEN];
    buf[0] = ' \0';
    do {
        /* skip space or comment lines. */
        if(fgets(buf, 255, File_Ptr) == NULL) {
            printf("Incomplete data
");
            buf[0] = '\0';
            break;
        }
        CheckChar(buf);
    } while(CommentLine(buf));
    return (buf); 
}

short ReadNumRuns(FILE* File_Ptr)
{
    char buf[STRLEN];
    short n=0;
    FindDataLine(File_Ptr); /* skip file version. */
    strcpy(buf, FindDataLine(File_Ptr));
    if(buf[0]=='\0') nrerror("Reading number of runs\n");
    sscanf(buf, "%hd", &n);
    return(n);
}
Read the file name and the file format.
The file format can be either A for ASCII or B for binary.

```c
void ReadFnameFormat(FILE *File_Ptr, InputStruct *In_Ptr)
{
    char buf[STRLEN];
    /* read in file name and format. */
    strcpy(buf, FindDataLine(File_Ptr));
    if(buf[0]=='0')
        nrerror("Reading file name and format.
    In_Ptr->out_fname, & In_Ptr->out_fformat );
    if(toupper(In_Ptr->out_fformat) != 'B'
        In_Ptr->out_format = 'A';
}
```

Read the number of photons.

```c
void ReadNumPhotons(FILE *File_Ptr, InputStruct *In_Ptr)
{
    char buf[STRLEN];
    /* read in number of photons. */
    strcpy(buf, FindDataLine(File_Ptr));
    if(buf[0]=='0')
        nrerror("Reading number of photons.
    In_Ptr->num_photons);  
    if(In_Ptr->num_photons<=0)
        nrerror("Nonpositive number of photons.
    In_Ptr->num_photons = 0;  
}
```

Read the members dz and dr.

```c
void ReadDzDr(FILE *File_Ptr, InputStruct *In_Ptr)
{
    char buf[STRLEN];
    /* read in dz, dr. */
    strcpy(buf, FindDataLine(File_Ptr));
    if(buf[0]=='0')
        nrerror("Reading dz, dr.
    In_Ptr->dz, &In_Ptr->dr);  
    if(In_Ptr->dz<=0)
        nrerror("Nonpositive dz.
    if(In_Ptr->dr<=0)
        nrerror("Nonpositive dr.
        In_Ptr->da = 0.5*PI/In_Ptr->na;
}
```

Read the members nz, nr, na.

```c
void ReadNzNrNa(FILE *File_Ptr, InputStruct *In_Ptr)
{
    char buf[STRLEN];
    /* read in number of dz, dr, da. */
    strcpy(buf, FindDataLine(File_Ptr));
    if(buf[0]=='0')
        nrerror("Reading number of dz, dr, da's.
    In_Ptr->nz, &In_Ptr->nr, &In_Ptr->na);  
    if(In_Ptr->nz<=0)
        nrerror("Nonpositive number of dz's.
    if(In_Ptr->nr<=0)
        nrerror("Nonpositive number of dr's.
    if(In_Ptr->na<=0)
        nrerror("Nonpositive number of da's.
    In_Ptr->da = 0.5*PI/In_Ptr->na;
}
```

Read the members x_size and y_size.

```c
void ReadXSizeYSize(FILE *File_Ptr, InputStruct *In_Ptr)
{
    char buf[STRLEN];
    /* read in x_size, y_size. */
    strcpy(buf, FindDataLine(File_Ptr));
    if(buf[0]=='0')
        nrerror("Reading x_size, y_size.
    In_Ptr->x_size, &In_Ptr->y_size);  
    if(In_Ptr->x_size<=0)
        nrerror("Nonpositive x_size.
    if(In_Ptr->y_size<=0)
        nrerror("Nonpositive y_size.
}
```

Read the number of layers.

```c
void ReadNumLayers(FILE *File_Ptr, InputStruct *In_Ptr)
{
    char buf[STRLEN];
    /* read in number of layers. */
    strcpy(buf, FindDataLine(File_Ptr));
    if(buf[0]=='0')
        nrerror("Reading number of layers.
    In_Ptr->num_layers);  
    if(In_Ptr->num_layers<=0)
        nrerror("Nonpositive number of layers.
    In_Ptr->num_layers = 0;
}
```
if(In_Ptr->num_layers<=0)
    nerror('Nonpositive number of layers.\n');
}

/***********************************************************
* Read the refractive index n of the ambient.
****/
void ReadAmbient(FILE *File_Ptr,
LayerStruct * Layer_Ptr,
char *side)
{
    char buf[STRLEN], msg[STRLEN];
    double n;
    strcpy(buf, FindDataLine(File_Ptr));
    if(buf[0]=='\0')
        sprintf(msg, "Reading n of %s ambient.\n", side);
    nerror(msg);
    sscanf(buf, "%lf", &n);
    if(n<=0)
        nerror('Wrong n.\n');
    Layer_Ptr->n = n;
}

/***********************************************************
* Read the parameters of one layer.
****/
void ReadLayerSpecs(FILE *File_Ptr,
short Num_Layers,
Layerstruct ** Layerspecs_PP)
{
    char buf[STRLEN];
    double d, n, mua, mus, g;
    if(buf[0]=='\0') return(1); /* error. */
    sscanf(buf, "%lf\lf%lf%lf%lf%lf", &n, &mua, &mus, &g, &d);
    if(d<0 || n<=0 || mua<0 || mus<0 || g<0 || g>1)
        return(1); /* error. */
    Layer_Ptr->n = n;
    Layer_Ptr->mua = mua;
    Layer_Ptr->mus = mus;
    Layer_Ptr->g = g;
    *Layer_Ptr->z0 = *Z_Ptr;
    *Layer_Ptr->z1 = *Z_Ptr;
    return(0);
}

/***********************************************************
* Read and set the parameters of the surrounding glass.
****/
Boolean ReadSurfGlass(FILE *File_Ptr,
LayerStruct * Layer_Ptr,
double *Z_Ptr)
{
    char buf[STRLEN];
    double n;
    strcpy(buf, FindDataLine(File_Ptr));
    if(buf[0]=='\0') return(1); /* error. */
    sscanf(buf, "%lf", &n);
    if(n<=0)
        nerror('Wrong n.\n');
    return(0);
}

/***********************************************************
* Read the parameters of one layer at a time.
****/
void ReadLayerSpecs(FILE *File_Ptr,
short Num_Layers,
LayerStruct ** Layerspecs_PP)
char msg[STRLEN];
short i = 0;
double z = 0.0; /* z coordinate of the current layer. */
/* Allocate an array for the layer parameters. */
/* layer 0 and layer Num_Layers + 1 are for ambient. */
*Layerspecs_PP = (LayerStruct *)malloc((unsigned) (Num_Layers+3) *sizeof(LayerStruct));
if (!(Layerspecs_PP))
  nerror("allocation failure in ReadLayerSpecs()");
ReadAmbient(File_Ptr, &(*Layerspecs_PP)[1], "top");
for(i=1; i<=Num_Layers; i++)
  if(ReadOneLayer(File_Ptr, &(*Layerspecs_PP)[i], &z))
    sprintf(msg, "Error reading %hd of %hd layers\n", i, Num_Layers);
  nerror(msg);
ReadAmbient(File_Ptr, &(*Layerspecs_PP)[1], "bottom");
if(ReadSurfGlass(File_Ptr, &(*Layerspecs_PP)[1+1], &z))
  sprintf(msg, "Error reading surrounding glass");
nerror(msg);
/**
 * Compute the critical angles for total internal
 * reflection according to the relative refractive index
 * of the layer.
 * All layers are processed.
 **/
void CriticalAngle(short Num_Layers, LayerStruct ** Layerspecs_PP)
{
  short i=0;
double n1, n2;
for(i=1; i<=Num_Layers; i++)
  { n1 = (*Layerspecs_PP)[i].n;
    n2 = (*Layerspecs_PP)[i-1].n;
    (*Layerspecs_PP)[i].cos_crit0 = n1>n2 ?
      sqrt(1.0 - n2*n2/(n1*n1)) : 0.0;
    n2 = (*Layerspecs_PP)[i+1].n;
    (*Layerspecs_PP)[i].cos_cri t1 = n1>n2 ?
      sqrt(1.0 - n2*n2/(n1*n1)) : 0.0;
    n2 = (*Layerspecs_PP)[Num_Layers+2].n;
    (*Layerspecs_PP)[i].cos_critxy = n1>n2 ?
      sqrt(1.0 - n2*n2/(n1*n1)) : 0.0;
  }
}
if (list == NULL) { /* first node. */
    List_Ptr = list = (NameLink)malloc(sizeof(NameNode));
    strcpy(list->name, Name);
    list->next = NULL;
} else { /* subsequent nodes. */
    /* Move to the last node. */
    while (list->next != NULL)
        list = list->next;
    /* Append a node to the list. */
    list->next = (NameLink)malloc(sizeof(NameNode));
    list = list->next;
    strcpy(list->name, Name);
    list->next = NULL;
    /* Check against duplicated file names. */
    A linked list is set up to store the file names used
    in this input data file.
    Boolean FnameTaken(char *fname, NameLink *List_Ptr)
    {
        if (NameInList(fname, *List_Ptr))
            return(1);
        else
            AddNameToList(fname, List_Ptr);
        return(0);
    }
    /***********************************************************
    * Free each node in the file name list.
    * FreenameList(NameLink List)
    {
        NameLink next;
        while (List != NULL) {
            next = List->next;
            free(List);
            List = next;
        }
    }
    /***********************************************************
    * Check the input parameters for each run.
    ****/
    void CheckParm(FILE *File_Ptr, InputStruct *In_Ptr)
    {
        short i_run;
        short num_runs; /* number of independent runs. */
        NameLink head = NULL;
        Boolean name_taken; /* output files share the same */
        /* file name. */
        char msg[STRLEN];
        num_runs = ReadNumRuns(File_Ptr);
        for (i_run = 1; i_run <= num_runs; i_run++) {
            printf("Checking input data for run %hd\n", i_run);
            ReadParm(File_Ptr, In_Ptr);
            name_taken = FnameTaken(In_Ptr->out_fname, &head);
            if (name_taken)
                sprintf(msg, "file name %s duplicated.\n",
                    In_Ptr->out_fname);
                free(In_Ptr->layerspecs);
                if (name_taken) nrerror(msg);
        }
        FreeFnameList(head);
        rewind(File_Ptr);
        /***********************************************************
        * Allocate the arrays in OutStruct for one run, and
        * array elements are automatically initialized to zeros.
        ****/
        void InitOutputData(InputStruct In_Ptr,
            short nz, short nr, short na, short nl;
            OutStruct *Out_Ptr)
        {
            short nz = In_Ptr.nz;
            short nr = In_Ptr.nr;
            short na = In_Ptr.na;
            short nl = In_Ptr.nl;
            /* remember to use nl+2 because of 2 for ambient. */
            if (nz<0 || nr<0 || na<0 || nl<0)
                nrerror("Wrong grid parameters.\n");
            /* Init pure numbers. */
            Out_Ptr->Rsp = 0.0;
            Out_Ptr->Rd = 0.0;
            Out_Ptr->A = 0.0;
            Out_Ptr->Tt = 0.0;
/* Allocate the arrays and the matrices. */
Out_Ptr->Rd_ra = AllocMatrix(0,nr-1,0,na-1);
Out_Ptr->Rd_r = AllocVector(0,nr-1);
Out_Ptr->Rd_a = AllocVector(0,na-1);
Out_Ptr->A_rz = AllocMatrix(0,nr-1,0,nz-1);
Out_Ptr->A_z = AllocVector(0,nz-1);
Out_Ptr->A_l = AllocVector(0,nl+1);
Out_Ptr->Tt_ra = AllocMatrix(0,nr-1,0,na-1);
Out_Ptr->Tt_r = AllocVector(0,nr-1);
Out_Ptr->Tt_a = AllocVector(0,na-1);

/***********************************************************
* Undo what InitOutputData did. i.e. free the data allocations.
***********************************************************
void FreeData(InputStruct In_Parm, OutStruct * Out_Ptr)
{
  short nz = In_Parm.nz;
  short nr = In_Parm.nr;
  short na = In_Parm.na;
  short nl = In_Parm.num_layers;
  /* remember to use nl+2 because of 2 for ambient. */
  free(In_Parm.layerspecs);
  FreeMatrix(Out_Ptr->Rd_ra, 0,nr-1,0,na-1);
  FreeVector(Out_Ptr->Rd_r, 0,nr-1);
  FreeVector(Out_Ptr->Rd_a, 0,na-1);
  FreeMatrix(Out_Ptr->A_rz, 0,nr-1,0,nz-1);
  FreeVector(Out_Ptr->A_z, 0,nz-1);
  FreeVector(Out_Ptr->A_l, 0,nl+1);
  FreeMatrix(Out_Ptr->Tt_ra, 0,nr-1,0,na-1);
  FreeVector(Out_Ptr->Tt_r, 0,nr-1);
  FreeVector(Out_Ptr->Tt_a, 0,na-1);
}

/***********************************************************
* Get 1D array elements by summing the 2D array elements.
***********************************************************
void Sum2DRA(InputStruct In_Parm, OutStruct * Out_Ptr)
{
  short nr = In_Parm.nr;
  short na = In_Parm.na;
  short ir,ia;
  double sum;
  for(ir=0; ir<nr; ir++)
  {
    sum = 0.0;
    for(ia=0; ia<na; ia++) sum += Out_Ptr->Rd_ra[ir][ia];
    Out_Ptr->Rd_r[ir] = sum;
  }
  for(ia=0; ia<na; ia++)
  {
    sum = 0.0;
    for(ir=0; ir<nr; ir++) sum += Out_Ptr->Rd_ra[ir][ia];
    Out_Ptr->Rd_a[ia] = sum;
  }
  sum = 0.0;
  for(ir=0; ir<nr; ir++)
  {
    sum = 0.0;
    for(ia=0; ia<na; ia++) sum += Out_Ptr->Rd_ra[ir][ia];
    Out_Ptr->Rd_r[ir] = sum;
  }
  for(ia=0; ia<na; ia++)
  {
    sum = 0.0;
    for(ir=0; ir<nr; ir++) sum += Out_Ptr->Rd_ra[ir][ia];
    Out_Ptr->Rd_a[ia] = sum;
  }
  sum = 0.0;
  for(ir=0; ir<nr; ir++) sum += Out_Ptr->Rd_r[ir];
  Out_Ptr->Rd = sum;
}

/***********************************************************
* Get the index to the layer according to the index to the grid line system in z direction (Iz).
* Use the center of box.
***********************************************************
short IzToLayer(short Iz, InputStruct In_Parm)
{
  short i=1; /* index to layer. */
  short num_layers = In_Parm.num_layers;
  double dz = In_Parm.dz;
  while( (Iz+0.5)*dz >= In_Parm.layerspecs[i].z1 && i<num_layers) i++;
  return(i);
}

/***********************************************************
* Get 1D array elements by summing the 2D array elements.
***********************************************************
void Sum20Rd(InputStruct In_Parm, OutStruct * Out_Ptr)
{
  short nr = In_Parm.nr;
  short na = In_Parm.na;
  short ir,ia;
  double sum;
  for(ir=0; ir<nr; ir++)
  {
    sum = 0.0;
    for(ia=0; ia<na; ia++) sum += Out_Ptr->Rd_ra[ir][ia];
    Out_Ptr->Rd_r[ir] = sum;
  }
  for(ia=0; ia<na; ia++)
  {
    sum = 0.0;
    for(ir=0; ir<nr; ir++) sum += Out_Ptr->Rd_ra[ir][ia];
    Out_Ptr->Rd_a[ia] = sum;
  }
  sum = 0.0;
  for(ir=0; ir<nr; ir++)
  {
    sum = 0.0;
    for(ia=0; ia<na; ia++) sum += Out_Ptr->Rd_ra[ir][ia];
    Out_Ptr->Rd_r[ir] = sum;
  }
  for(ia=0; ia<na; ia++)
  {
    sum = 0.0;
    for(ir=0; ir<nr; ir++) sum += Out_Ptr->Rd_ra[ir][ia];
    Out_Ptr->Rd_a[ia] = sum;
  }
  sum = 0.0;
  for(ir=0; ir<nr; ir++) sum += Out_Ptr->Rd_r[ir];
  Out_Ptr->Rd = sum;
}
```c
Out_Ptr->A_z[iz] = sum;
}
sum = 0.0;
for(iz=0; iz<nz; iz++) {
  sum += Out_Ptr->A_z[iz];
  Out_Ptr->A[1][zzToLayer(iz, In_Parm)] += Out_Ptr->A_z[iz];
}
Out_Ptr->A = sum;
}

/*************************************************************************

Get 2D array elements by summing the 2D array elements.
*************************************************************************/
void Sum2DToC(InputStruct In_Parm, OutStruct *Out_Ptr)
{
  short nr = In_Parm.nr;
  short na = In_Parm.na;
  short ir,ia;
  double sum;
  for(ir=0; ir<nr; ir++) {
    sum = 0.0;
    for(ia=0; ia<na; ia++) sum += Out_Ptr->Tt_ra[ir][ia];
    Out_Ptr->Tt_r[ir] = sum;
  }
  sum = 0.0;
  for(ia=0; ia<na; ia++) {
    sum += Out_Ptr->Tt_ra[ir][ia];
    Out_Ptr->Tt[ia] = sum;
  }

  sum = 0.0;
  for(ia=0; ia<na; ia++) {
    sum += Out_Ptr->Tt_ra[ir][ia];
    Out_Ptr->Tt_r[ir] = sum;
  }

  sum = 0.0;
  for(ia=0; ia<na; ia++) sum += Out_Ptr->Tt[ia];
  Out_Ptr->Tt = sum;
}

/*************************************************************************

Scale Rd and Tt properly.
*************************************************************************/
void ScaleRdTt(InputStruct In_Parm, OutStruct *Out_Ptr)
{
  short nr = In_Parm.nr;
  short na = In_Parm.na;
  double dr = In_Parm.dr;
  double da = In_Parm.da;
  short ir,ia;
  double scale1, scale2;
  scale1 = 4.0*PI*dr*sin(da/2)*In_Parm.num_photons;
  /* The factor (ir+0.5)*sin(2a) to be added. */
  for(ir=0; ir<nr; ir++) {
    for(ia=0; ia<na; ia++) {
      scale2 = 1.0/((ir+0.5)*sin((2.0*ir+0.5)*da)*scale1);
      Out_Ptr->Rd_ra[ir][ia] *= scale2;
      Out_Ptr->Tt_ra[ir][ia] *= scale2;
    }
  }
  scale1 = 2.0*PI*dr*In_Parm.num_photons;
  /* area is 2*PI*(ir+0.5)*dr. */
  for(ir=0; ir<nr; ir++) {
    scale2 = 1.0/((ir+0.5)*scale1);
    Out_Ptr->Rd_r[ir] *= scale2;
    Out_Ptr->Tt_r[ir] *= scale2;
  }
  scale1 = 2.0*PI*da*In_Parm.num_photons;
  /* solid angle is 2*PI*sin(a)*da. sin(a) to be added. */
  for(ia=0; ia<na; ia++) {
    scale2 = 1.0/(sin((2.0*ia+0.5)*da)*scale1);
    Out_Ptr->Rd_a[ia] *= scale2;
    Out_Ptr->Tt_a[ia] *= scale2;
  }
  scale2 = 1.0/(double)In_Parm.num_photons;
  Out_Ptr->Rd *= scale2;
  Out_Ptr->Tt *= scale2;
}
```

---

Out_Ptr->A_z[iz] = sum;

sum = 0.0;
for(iz=0; iz<nz; iz++) {
  sum += Out_Ptr->A_z[iz];
  Out_Ptr->A[1][zzToLayer(iz, In_Parm)] += Out_Ptr->A_z[iz];
}
Out_Ptr->A = sum;

*************************************************************************

Get 2D array elements by summing the 2D array elements.
*************************************************************************/
void Sum2DToC(InputStruct In_Parm, OutStruct *Out_Ptr)
{
  short nr = In_Parm.nr;
  short na = In_Parm.na;
  short ir,ia;
  double sum;
  for(ir=0; ir<nr; ir++) {
    sum = 0.0;
    for(ia=0; ia<na; ia++) sum += Out_Ptr->Tt_ra[ir][ia];
    Out_Ptr->Tt_r[ir] = sum;
  }
  sum = 0.0;
  for(ia=0; ia<na; ia++) {
    sum += Out_Ptr->Tt_ra[ir][ia];
    Out_Ptr->Tt[ia] = sum;
  }

  sum = 0.0;
  for(ia=0; ia<na; ia++) {
    sum += Out_Ptr->Tt_ra[ir][ia];
    Out_Ptr->Tt_r[ir] = sum;
  }

  sum = 0.0;
  for(ia=0; ia<na; ia++) sum += Out_Ptr->Tt[ia];
  Out_Ptr->Tt = sum;
}

*************************************************************************

Scale Rd and Tt properly.
*************************************************************************/
void ScaleRdTt(InputStruct In_Parm, OutStruct *Out_Ptr)
{
  short nr = In_Parm.nr;
  short na = In_Parm.na;
  double dr = In_Parm.dr;
  double da = In_Parm.da;
  short ir,ia;
  double scale1, scale2;
  scale1 = 4.0*PI*dr*sin(da/2)*In_Parm.num_photons;
  /* The factor (ir+0.5)*sin(2a) to be added. */
  for(ir=0; ir<nr; ir++) {
    for(ia=0; ia<na; ia++) {
      scale2 = 1.0/((ir+0.5)*sin((2.0*ir+0.5)*da)*scale1);
      Out_Ptr->Rd_ra[ir][ia] *= scale2;
      Out_Ptr->Tt_ra[ir][ia] *= scale2;
    }
  }
  scale1 = 2.0*PI*dr*In_Parm.num_photons;
  /* area is 2*PI*(ir+0.5)*dr. */
  for(ir=0; ir<nr; ir++) {
    scale2 = 1.0/((ir+0.5)*scale1);
    Out_Ptr->Rd_r[ir] *= scale2;
    Out_Ptr->Tt_r[ir] *= scale2;
  }
  scale1 = 2.0*PI*da*In_Parm.num_photons;
  /* solid angle is 2*PI*sin(a)*da. sin(a) to be added. */
  for(ia=0; ia<na; ia++) {
    scale2 = 1.0/(sin((2.0*ia+0.5)*da)*scale1);
    Out_Ptr->Rd_a[ia] *= scale2;
    Out_Ptr->Tt_a[ia] *= scale2;
  }
  scale2 = 1.0/(double)In_Parm.num_photons;
  Out_Ptr->Rd *= scale2;
  Out_Ptr->Tt *= scale2;
void ScaleA(InputStruct In_Parm, OutStruct *Out_Ptr) {
    short nz = In_Parm.nz;
    short nr = In_Parm.nr;
    double dz = In_Parm.dz;
    double dr = In_Parm.dr;
    short nl = In_Parm.num_layers;
    short il, ir, iz;
    double scalel;

    scalel = 2.0 * PI * dr * dr * dz * In_Parm.num_photons;
    /* volume is 2*pi*(irt0.5i*dr*dr*dz.*' ir+O.S to be added. */

    for(iz=0; iz<nz; iz++)
        for(ir=0; ir<nr; ir++)
            Out_Ptr->A[iz][ir] *= scalel;

    Out_Ptr->A2[*=scalel;

    for(iz=0; iz<nz; iz++)
        for(ir=0; ir<nr; ir++)
            Out_Ptr->A_z[iz][ir] /= (ir+0.5)*scalel;

    Out_Ptr->A .*=scalel;

} void SumScaleResult(InputStruct In_Parm, OutStruct *Out_Ptr) {
    /* Get ID & OD results. */
    Sum2DRd(In_Parm, Out_Ptr);
    Sum2DA(In_Parm, Out_Ptr);
    Sum2DTt(In_Parm, Out_Ptr);
    ScaleRDt(In_Parm, Out_Ptr);
    ScaleA(In_Parm, Out_Ptr);
}

/* Write the version number as the first string in the file.
 * Use chars only so that they can be read either ASCII or binary.
 */
void WriteVersion(FILE *file, char *Version) {
    fprintf(file, "## Version number of the file format.\n\n",

    fprintf(file, "###\n# Data categories include: in\n",

    fprintf(file, "## A_l, A_z, Rd_r, Rd_a, Tt_r, Tt_a, in\n",

    fprintf(file, "## A_rz, Rd_ra, Tt_ra\n####\n
")

} void WriteInParm(FILE *file, InputStruct In_Parm) {
    short i;

    fprintf(file, "InParm \t\t\t\t\t# Input parameters. cm is used.\n",

    fprintf(file, "%s \t\t\t# output file name, ASCII.\n",

    fprintf(file, "%ld\t\t\t\t\t# No. of photons\n",

    fprintf(file, "%G\t\t\t\t\t# x_size and y_size.\n",

    fprintf(file, "%hd\t\t\t\t\t# No. of layers\n",

    fprintf(file, "%G\t\t\t\t\t# for medium above\n",

}
LayerStruct s;
s = In_Parm.layerspecs[i];
fprintf(file, "%G\t%G\t%G\t%G\t%G\t# layer %hd\n",
    s.n, s.mua, s.mus, s.g, s.zl-s.zO, i);
fprintf(file, "%G\t# n for medium below\n",
    In_Parm.layerspecs[i].n);
fprintf(file, "%G\t# n for surrounding glass\n",
    In_Parm.layerspecs[i+1].n);

/*************************************************************************/
*/ Write reflectance, absorption, transmission.  
**/  

void WriteRAT(FILE * file, OutStruct Out_Parm)
{
    fprintf(file, "RAT  
#Reflectance, absorption, transmission. \n"
);  /* flag. */
    fprintf(file, "%-14.6G 	#Specular reflectance  [-]
", Out_Parm.Rsp);
    fprintf(file, "%-14.6G 	#Diffuse reflectance  [-]
", Out_Parm.Rd);
    fprintf(file, "%-14.6G 	#Absorbed fraction  [-]
", Out_Parm.AI);
    fprintf(file, "%-14.6G 	#Transmittance  [-]
", Out_Parm.Tt);
    fprintf(file, "\n");

/*************************************************************************/
*/ Write absorption as a function of layer.  
**/  

void WriteA_layer(FILE * file, short Num_Layers,
    OutStruct Out_Parm)
{
    short i;
    fprintf(file, "A_l 
#Absorption as a function of layer. [-]
"
);  /* flag. */
    for(i=1; i<=Num_Layers; i++)
        fprintf(file, '%12.4G
', Out_Parm.A_l[i]);
    fprintf(file, "\n");

/*************************************************************************/
*/ 5 numbers each line.  
**/  

void WriteRd_ra(FILE * file, short Nr,
    short Na,  
    OutStruct Out_Parm)
{
    short ir, ia;
    fprintf(file,  
"%s\n%s\n%s\n%s\n%s\n%s
", /* flag. */  
"# Rd[ir][angle]. [1/cm2sr]",  
"# Rd[0][0], [0][1],...[0][na-1]",  
"# Rd[1][0], [1][1],...[1][na-1]",  
"# ...
",  
"# Rd[nr-1][0], [nr-1][1],...[nr-1][na-1]",  
"*Rd_ra*");
    for(ir=0;ir<Nr;ir++)
        for(ia=0;ia<Na;ia++)  
            fprintf(file, "%12.4E
", Out_Parm.Rd_ra[ir][ia]);
    fprintf(file, "\n");

/*************************************************************************/
*/ 1 number each line.  
**/  

void WriteRd_r(FILE * file, short Nr,
    short Na,  
    OutStruct Out_Parm)
{
    short ir;
    fprintf(file,  
"Rd_r 
#Rd[0],...Rd[nr-1]. [1/cm2]
"
);  /* flag. */  
    for(ir=0;ir<Nr;ir++)
        fprintf(file, "%12.4E
", Out_Parm.Rd_r[ir]);
    fprintf(file, "\n");

/*************************************************************************/
*/ 1 number each line.  
**/  

void WriteRd_a(FILE * file, 
*/
short Na,
OutStruct Out_Param;
{
    short ia;
    fprintf(file,
        "Rd_a #Rdi[0], [1]...Rd[na-1]. [sr-1]\n*n); /* flag */
    for(ia=0;ia<Na;ia++)
        fprintf(file, "%12.4E", Out_Param.Rd_a[ia]);
    fprintf(file, "\n");
}

/***********************************************************
** 5 numbers each line.
***/
void WriteTt_ra(FILE * file,
    short Nr,
    short Na,
    OutStruct Out_Param)
{
    short ir, ia;
    fprintf(file,
        "%s\n%s\n%s\n%s\n%s\n", /* flag */
        "# Tt[angle]. [1/cm2sr].",
        "# Tt[0][0], [0][1]...[0][na-1].",
        "# Tt[1][0], [1][1]...[1][na-1].",
        "# ...",
        "# Tt[nr-1][0], [nr-1][1]...[nr-1][na-1].",
        "Tt_ra";
    for(ir=0;ir<Nr;ir++)
        for(ia=0;ia<Na;ia++)
            fprintf(file, "%12.4E " Out_Param.Tt_ra[ir][ia]);
    fprintf(file, "\n");
}

/***********************************************************
** 5 numbers each line.
***/
void WriteA_rz(FILE * file,
    short Nz,
    short Na,
    OutStruct Out_Param)
{ short iz, ir;
    fprintf(file,
        "%s\n%s\n%s\n%s\n%s\n", /* flag */
        "# A[0][0], [0][1]...[0][nz-1].",
        "# A[1][0], [1][1]...[1][nz-1].",
        "# ...",
        "# A[nr-1][0], [nr-1][1]...[nr-1][nz-1].",
        "A_rz";
    for(iz=0;iz<Nz;iz++)
        for(ir=0;ir<Nr;ir++)
            fprintf(file, "%12.4E " Out_Param.A_rz[ir][iz]);
    fprintf(file, "\n");
}

/***********************************************************
** 1 number each line.
***/
void WriteA_z(FILE * file,
    short Nz,
    OutStruct Out_Param)
{ short iz;
    fprintf(file,
        "A_z #A[0], [1]...A[nz-1]. [1/cm]\n*n); /* flag */
    for(iz=0;iz<Nz;iz++)
        fprintf(file, "%12.4E", Out_Param.A_z[iz]);
    fprintf(file, "\n");
}

/***********************************************************
** 1 number each line.
***/
void WriteTt_r(FILE * file,
    short Nr,
    OutStruct Out_Param)
{ short ir;
    fprintf(file,
        "# Tt[angle]. [1/cm2sr].",
        "# Tt[0][0], [0][1]...[0][na-1].",
        "# Tt[1][0], [1][1]...[1][na-1].",
        "# ...",
        "# Tt[nr-1][0], [nr-1][1]...[nr-1][na-1].",
        "Tt_r";
    for(ir=0;ir<Nr;ir++)
        for(ia=0;ia<Na;ia++)
            fprintf(file, "%12.4E " Out_Param.Tt_r[ir][ia]);
    fprintf(file, "\n");
}
B.1.1.4. mcmlgo.c

******************************************************************************
/* Launch, move, and record photon weight. */
******************************************************************************
#include "mcml.h"
#define STANDARDTEST 0 /* testing program using fixed rnd seed. */
#define PARTIALREFLECTION 0 /* 1=split photon, 0=statistical reflection. */

#define COSZERO (1.0 - 1.0E-12) /* cosine of about 1e-6 rad. */
#define COS90D 1.0E-6 /* cosine of about 1.57 - 1e-6 rad. */

/***********************************************************
/* A random number generator from Numerical Recipes in C. */
/***********************************************************

FILE *file;
file = fopen(In_Param.out_fname, "w");
if(file == NULL) nrerror("Cannot open file to write.
");
if (toupper(In_Param.out_fformat) == 'A')
   WriteVersion(file, "A1");
else
   WriteVersion(file, "B1");
fprintf(file, "# %s", TimeReport);
fprintf(file, "%12.4E
", Out_Parm.Tt_r[ir]);
fprintf(file, "
");
for(ia=0;ia<Na;ia++)
   fprintf(file, "%12.4E
", Out_Parm.Tt_a[ia]);
 fprintf(file, "
");
WriteInParm(file, In_Param);
WriteRAT(file, Out_Parm);
WriteA_layer(file, In_Param.num_layers, Out_Parm);
WriteA_z(file, In_Param.nz, Out_Param);
WriteRd_r(file, In_Param.nr, Out_Param);
WriteTt_r(file, In_Param.nr, Out_Param);
WriteTt_a(file, In_Param.na, Out_Param);
WriteA_rz(file, In_Param.nr, In_Param.nz, Out_Param);
WriteRd_a(file, In_Param.nr, In_Param.na, Out_Param);
WriteTt_r(file, In_Param.nr, In_Param.na, Out_Param);
fclose(file);

**************
void WriteTt_a(FILE * file, short Na, OutStruct Out_Param) 
{
   short ia;
   fprintf(file, "Tt_a #Tt[0], [1],..Tt[n-1]. [sr-1]
");
   for(ia=0;ia<Na;ia++)
      fprintf(file, "%12.4E
", Out_Parm.Tt_a[ia]);
   fprintf(file, "
");
}

**************
void WriteResult(InputStruct In_Param, 
   OutStruct Out_Param, 
   char * TimeReport) 
{
define MBIG 1000000000
#define MSEED 161003398
#define MZ 0
#define FAC 1.0E-9

float ran3(int *idum)
{
    static int inext, inextp;
    static long ma[56];
    static int iff = 0;
    long mj, mk;
    int i, ii, k;
    if (*idum < 0 || iff == 0) {
        iff = 1;
        mj = MSEED - *idum <= 0 ? -*idum : *idum);
        mj %= MBIG;
        ma[55] = mj;
        mk = 1;
        for (i = 1; i <= 54; i++) {
            i = (i-1) * 55;
            ma[i] = mk;
            mk = mj - mk;
            if (mk < MZ) mk += MBIG;
            mj = ma[i];
        }
        for (k = 1; k <= 4; k++) {
            for (i = 1; i <= 55; i++) {
                ma[i] = ma[i+(i-30) % 55];
                if (ma[i] < MZ) ma[i] += MBIG;
            }
            inext = 0;
            inextp = 1;
            *idum = 1;
        }
        if (++inext == 56) inext = 1;
        if (++inext == 56) inextp = 1;
        if (mj = ma[inext] - ma[inextp];
        if (mj < MZ) mj += MBIG;
        ma[inext] = mj;
        return mj * FAC;
    }
}

double RandomNum(void)
{
    static Boolean first_time = 1;
    static int idum; /* seed for ran3. */
    if (first_time) {
        #if STANDARDTEST /* Use fixed seed to test the program. */
            idum = -1;
        #else
            idum = -(int)time(NULL) % (1<<15);
            /* use 16-bit integer as the seed. */
        #endif
        ran3(&idum);
        first_time = 0;
        idum = 1;
    }
    return (double) ran3(&idum);
}

double Rspecular(LayerStruct * Layerspecs_Ptr)
{
    double r1, r2;
    /* direct reflections from the 1st and 2nd layers. */
    double temp;
    temp = (Layerspecs_Ptr[0].n - Layerspecs_Ptr[1].n) / ([Layerspecs_Ptr[0].n] + Layerspecs_Ptr[1].n);
    r1 = temp * temp;
    * Generate a random number between 0 and 1. Take a number as seed the first time entering the function.
    * The seed is limited to 1<<15.
    * We found that when idum is too large, ran3 may return numbers beyond 0 and 1.
    *****/
    double RandomNum(void)
    {
        static Boolean first_time = 1;
        static int idum; /* seed for ran3. */
        if (first_time) {
            #if STANDARDTEST /* Use fixed seed to test the program. */
                idum = -1;
            #else
                idum = -(int)time(NULL) % (1<<15);
                /* use 16-bit integer as the seed. */
            #endif
            ran3(&idum);
            first_time = 0;
            idum = 1;
        }
        return (double) ran3(&idum);
    }

    Compute the specular reflection.
    * If the first layer is a turbid medium, use the Fresnel reflection from the boundary of the first layer as the specular reflectance.
    * If the first layer is glass, multiple reflections in the first layer is considered to get the specular reflectance.
    * The subroutine assumes the Layerspecs array is correctly initialized.
    *****/
    double Rspecular(LayerStruct * Layerspecs_Ptr)
    {
        double r1, r2;
        /* direct reflections from the 1st and 2nd layers. */
        double temp;
        temp = (Layerspecs_Ptr[0].n - Layerspecs_Ptr[1].n) / ([Layerspecs_Ptr[0].n] + Layerspecs_Ptr[1].n);
        r1 = temp * temp;
    }
if((Layerspecs_Ptr[1].mua == 0.0) 
&& (Layerspecs_Ptr[1].mus == 0.0)) {/* glass layer. */
    temp = (Layerspecs_Ptr[1].n - Layerspecs_Ptr[2].n) /
              (Layerspecs_Ptr[1].n + Layerspecs_Ptr[2].n);
    r2 = temp*temp;
    r1 = r1 + (1-r1)*(1-r1)*r2/(1-r1*r2);
}

    return (r1);

} /* End of Initialize a photon packet. */

//***********************************************************
// Choose a new direction for photon propagation by
// sampling the polar deflection angle theta and the
// azimuthal angle psi. */
//***********************************************************

double SpinTheta(double g)
{
    double cost, sint;
    if(g == 0.0)
    {
        cost = 2*RandomNum() -1;
        else 
        {
            double temp = (1-g*g)/(1-g+2*g*RandomNum());
            cost = (1+g*g) - temp*temp)/(2*g);
        }
    return(cost);
}

void Spin(double g,
          PhotonStruct * Photon_Ptr)
{
    double cost, sint, cosp, sinp;
    cost = SpinTheta(g);
    sint = sqrt(1.0 - cost*cost);
    if(psi<PI)
        cosp = cos(psi);
    else 
        cosp = cosp*cos(psi); /* sqrt() is faster than sin(). */
    psi = 2.0*PI*RandomNum(); /* spin psi 0-2pi. */
    if(psi<PI)
        sinp = sqrt(1.0 - cosp*cosp);
    else 
        sinp = - sqrt(1.0 - cosp*cosp);

    return (sinp, cosp); /* cosine and sine of the */
    /* polar deflection angle theta. */
    /* azimuthal angle psi. */

    double ux = Photon_Ptr->ux;
    double uy = Photon_Ptr->uy;
    double uz = Photon_Ptr->uz;
    double psi;
    cost = SpinTheta(g);
    sint = sqrt(1.0 - cost*cost);
    /* sqrt() is faster than sin(). */
    if(psi<PI)
        sinp = sqrt(1.0 - cosp*cosp);
    else 
        sinp = - sqrt(1.0 - cosp*cosp);
    return (sint, cosp, sinp, cosp); /* cosine and sine of the */
    /* polar deflection angle theta. */
    /* azimuthal angle psi. */

    double u0, u1, u2, u3, u4, u5, u6, u7, u8, u9;
if(fabs(uz) > COSZERO) { /* normal incident. */
  Photon_Ptr->ux = sint*cosp;
  Photon_Ptr->uy = sint*sinp;
  Photon_Ptr->uz = cost*SIGN(uz);
  /* SIGN() is faster than division. */
} else { /* regular incident. */
  double temp = sqrt(1.0 - uz*uz);
 Photon_Ptr->ux = sint*(ux*uz*cosp - uy*sinp)/temp + ux*cost;
  Photon_Ptr->uy = sint*(uy*uz*cosp + ux*sinp)/temp + uy*cost;
  Photon_Ptr->uz = -sint*cosp*temp + uz*cost;
}

/*****************************************************/
*/ Move the photon s away in the current layer of medium. */
void Hop(PhotonStruct * Photon_Ptr)
{
  double s = Photon_Ptr->s;
  Photon_Ptr->x += s*Photon_Ptr->ux;
  Photon_Ptr->y += s*Photon_Ptr->uy;
  Photon_Ptr->z += s*Photon_Ptr->uz;
}

/*****************************************************/
*/ The step size is the distance between the current */
*/ position and the nearest boundary in the photon direction. */
*/ uz ! = 0 */

short StepSizeInGlass(PhotonStruct * Photon_Ptr,
  InputStruct * In_Ptr)
{
  double dl_b; /* distance to nearest boundary. */
  double dl_xbound; /* distance to boundary perp. to x-direction. */
  double dl_ybound; /* distance to boundary perp. to y-direction. */
  double dl_zbound; /* distance to boundary perp. to z-direction. */
  short layer = Photon_Ptr->layer;
  double ux = Photon_Ptr->ux;
  double uy = Photon_Ptr->uy;
  double uz = Photon_Ptr->uz;
  short hitcase; /* the boundary which is hit. */
  double dl_zbound = (In_Ptr->layerspecs[layer].z1 - Photon_Ptr->z)/uz;
  else if(uz<0.0)
    dl_zbound = (In_Ptr->layerspecs[layer].z2 - Photon_Ptr->z)/uz;
  dl_b = dl_zbound;
  hitcase = 3;
  if(ux != 0.0) {
    dl_xbound = ((In_Ptr->x_size)*SIGN(ux)/2 - Photon_Ptr->x)/ux;
    if (dl_xbound<dl_b) {
      dl_b = dl_xbound;
      hitcase = 1;
    }
  } if(uy != 0.0) {
    dl_ybound = ((In_Ptr->y_size)*SIGN(uy)/2 - Photon_Ptr->y)/uy;
    if (dl_ybound<dl_b) {
      dl_b = dl_ybound;
      hitcase = 2;
    }
  }
  Photon_Ptr->s = dl_b;
  return(hitcase);
}

/*****************************************************/
*/ The step size is the distance between the current */
*/ position and the boundary, perp. to z-direction, */
*/ in the photon direction. */
*/ uz != 0 */

void StepSizeInSurrGlass(PhotonStruct * Photon_Ptr,
  InputStruct * In_Ptr)
{
  double dl_b; /* distance to boundary. */
  short layer = Photon_Ptr->layer;
  double uz = Photon_Ptr->uz;
  if(uz>0.0)
    dl_zbound = (In_Ptr->layerspecs[layer].z1 - Photon_Ptr->z)/uz;
  else if(uz<0.0)
    dl_zbound = (In_Ptr->layerspecs[layer].z0 - Photon_Ptr->z)/uz;
  dl_b = dl_zbound;
  return(1);
 Photon_Ptr->Z)/UZ; /* dl_b>0. */
Photon_Ptr->s = dl_b;
}
/* *********************** */
/* Pick a step size for a photon packet when it is in */
/* tissue. */
/* If the member sleft is zero, make a new step size */
/* with: -log(rnd)/(mua+mus). */
/* Otherwise, pick up the leftover in sleft. */
/* Layer is the index to layer. */
/* In_Ptr is the input parameters. */
void StepSizeInTissue(PhotonStruct * Photon_Ptr,
                      InputStruct * In_Ptr)
{
short layer = Photon_Ptr->layer;
double mua = In_Ptr->layerspecs[layer].mua;
double mus = In_Ptr->layerspecs[layer].mus;
if(Photon_Ptr->sleft == 0.0) { /* make a new step. */
    double rnd;
    do rnd = RandomNum();
      while( rnd <= 0.0 ); /* avoid zero. */
    Photon_Ptr->s = -log(rnd)/(mua+mus);
} else { /* take the leftover. */
    Photon_Ptr->s = Photon_Ptr->sleft/(mua+mus);
    Photon_Ptr->sleft = 0.0;
}
/* *********************** */
/* Check if the step will hit a boundary. */
/* Return 1, 2 or 3 if a boundary is hit. */
/* Return 0 otherwise. */
/* If the projected step hits the boundary, the members */
/* s and sleft of Photon_Ptr are updated. */
/* *****/
short HitBoundary(PhotonStruct * Photon_Ptr,
                  InputStruct * In_Ptr)
{
    double dl_b; /* distance to nearest boundary. */
    double dl_xbound; /* distance to boundary perp. to x-direction. */
    double dl_ybound; /* distance to boundary perp. to y-direction. */
    double dl_zbound; /* distance to boundary perp. to z-direction. */
    short layer = Photon_Ptr->layer;
double ux = Photon_Ptr->ux;
double uy = Photon_Ptr->uy;
double uz = Photon_Ptr->uz;
short hitcase; /* the boundary which is hit. */
/* hitcase = 0 if no boundary is hit */
double maxdl_xybound = sqrt(pow(In_Ptr->x_size,2.0) + pow(In_Ptr->y_size,2.0));
    dl_zbound = (In_Ptr->layerspecs[layer].z1 - In_Ptr->layerspecs[layer].z0)/uz;
    /* The longest possible path in this layer. */
    if(ux>0.0)
        dl_zbound = (In_Ptr->layerspecs[layer].z1 - Photon_Ptr->z)/uz;
    else if(ux<0.0)
        dl_zbound = (In_Ptr->layerspecs[layer].z0 - Photon_Ptr->z)/uz;
    dl_b = maxdl_xybound;
    if (ux != 0.0) {
        dl_xbound = (((In_Ptr->x_size)*SIGN(ux))/2 - Photon_Ptr->x)/ux;
        if (dl_xbound<dl_b) {
            dl_b = dl_xbound;
            hitcase = 1;
        }
    }
    if(uy != 0.0) {
        dl_ybound = (((In_Ptr->y_size)*SIGN(uy))/2 - Photon_Ptr->y)/uy;
        if (dl_ybound<dl_b) {
            dl_b = dl_ybound;
            hitcase = 2;
        }
    }
    if(Photon_Ptr->s > dl_b) {
        /* crossing. */
        double mut = In_Ptr->layerspecs[layer].mua
                      + In_Ptr->layerspecs[layer].mus;
        Photon_Ptr->sleft = (Photon_Ptr->s - dl_b)*mut;
    } else
        hitcase = 0;
}
/* Drop photon weight inside the tissue (not glass). */
* The photon is assumed not dead.
* The weight drop is \( dw = \frac{w \cdot \mu_a}{\mu_a + \mu_s} \).
* The dropped weight is assigned to the absorption array elements.
*/

void Drop(InputStruct * In_Ptr, PhotonStruct * Photon_Ptr, OutStruct * Out_Ptr)
{
  double dwa; /* absorbed weight. */
  double x = Photon_Ptr->x;
  double y = Photon_Ptr->y;
  short iz, ir; /* index to z & r. */
  short layer = Photon_Ptr->layer;
  double mua, mus;

  /* compute array indices. */
  iz = (short)(Photon_Ptr->z / In_Ptr->dz);
  if(iz>=In_Ptr->nz-1) iz=In_Ptr->nz-1;

  ir = (short)(sqrt(x*x+y*y) / In_Ptr->dr);
  if(ir>=In_Ptr->nr-1) ir=In_Ptr->nr-1;

  /* update photon weight. */
  mua = In_Ptr->layerspecs[layer].mua;
  mus = In_Ptr->layerspecs[layer].mus;
  dwa = Photon_Ptr->w * mua / (mua+mus);
  Photon_Ptr->w -= dwa;

  /* assign dwa to the absorption array element. */
  Out_Ptr->A[ir][iz] += dwa;
}

void Roulette(PhotonStruct * Photon_Ptr)
{
  if(Photon_Ptr->w == 0.0)
  { Photon_Ptr->dead = 1;
  }
  else if(RandomNum() < CHANCE) /* survived the roulette. */
  { Photon_Ptr->w /= CHANCE;
    else Photon_Ptr->dead = 1;
  }

/************************************************************
* Compute the Fresnel reflectance. *
* Make sure that the cosine of the incident angle \( \alpha_l \) is positive, and the case when the angle is greater than the critical angle is ruled out.
* Avoid trigonometric function operations as much as possible, because they are computation-intensive.
************************************************************

double RFresnel(double nl, /* incident refractive index. */
  double n2, /* transmit refractive index. */
  double cal, /* cosine of the incident */
  double * ca2_Ptr /* cosine of the transmission */
  /* angle. 0<\alpha_l<90 degrees. */

double r;
if(nl==n2) /* 0<\alpha_l<90 degrees, normal incident. */
  *ca2_Ptr = cal;
  r = 0.0;
else if(cal>COSZERO) /* 0<\alpha_l<90 degrees, normal incident. */
  *ca2_Ptr = cal;
  r = (n2-n1)/(n2+n1);
  r *= r;
else if(cal>COS90D) /* 0<\alpha_l<90 degrees, very slant. */
  *ca2_Ptr = 0.0;
  r = 1.0;
else /* general. */
  double sal, sa2;
  /* sine of the incident and transmission angles. */
  double ca2;
  sa1 = sqrt(1-cal*cal);
  sa2 = n1*sa1/n2;
  if(sa2>=1.0) /* double check for total internal reflection. */
    *ca2_Ptr = 0.0;
    r = 1.0;
else {
    double cap, cam; /* cosines of the sum ap or */
    /* difference am of the two */
    /* angles: ap = a1+a2 */
    /* am = a1 - a2. */
    double sap, sam; /* sines. */
    *ca2_Ptr = ca2 = sqrt(1-sa2*sa2);
    cap = ca1*ca2 - sa1*sa2;
    *C+ =
    cam = ca1*ca2 + sa1*sa2;
    *C- =
    sap = sa1*ca2 + ca1*sa2;
    *S+ =
    sam = sa1*ca2 - ca1*sa2;
    *S- =
    cc = 0.5*sam*sam*(cap*cap+cap*cap)/(sap*sap*cam*cam);
    /* rearranged for speed. */
    return(trl);
}

/***********************************************************
* Record the photon weight exiting the first layer(uz<0),
* no matter whether the layer is glass or not, to the
* reflection array.
* */
void RecordR(double Refl, InputStruct * In_Ptr,
              PhotonStruct * Photon_Ptr,
              OutStruct * Out_Ptr) {
    double x = Photon_Ptr->x;
    double y = Photon_Ptr->y;
    short ir, ia; /* index to r & angle. */
    ir = (short)(sqrt(x*x+y*y)/In_Ptr->dr);
    if(ir>In_Ptr->nr-1) ir=In_Ptr->nr-1;
    ia = (short)(acos(-Photon_Ptr->uz)/In_Ptr->da);
    if(ia>In_Ptr->na-1) ia=In_Ptr->na-1;
    /* assign photon to the reflection array element. */
    Out_Ptr->Rd_ra[ir][ia] += Photon_Ptr->w*(1.0-Refl);
    Photon_Ptr->w *= Refl;
}

/***********************************************************
* Record the photon weight exiting the last layer(uz>0),
* no matter whether the layer is glass or not, to the
* transmission array.
* */
void RecordT(double Refl, InputStruct * In_Ptr,
              PhotonStruct * Photon_Ptr,
              OutStruct * Out_Ptr) {
    double x = Photon_Ptr->x;
    double y = Photon_Ptr->y;
    short ir, ia; /* index to r & angle. */
    ir = (short)(sqrt(x*x+y*y)/In_Ptr->dr);
    if(ir>In_Ptr->nr-1) ir=In_Ptr->nr-1;
    ia = (short)(acos(-Photon_Ptr->uz)/In_Ptr->da);
    if(ia>In_Ptr->na-1) ia=In_Ptr->na-1;
    /* assign photon to the transmission array element. */
    Out_Ptr->Tt_ra[ir][ia] += Photon_Ptr->w*(1.0-Refl);
    Photon_Ptr->w *= Refl;
}

/***********************************************************
* Decide whether the photon will be transmitted or
* reflected on the upper boundary (uz<0) of the current
* layer.
* * If "layer" is the first or surrounding layer,
* the photon packet will be partially transmitted
* and partially reflected if
* PARTIALREFLECTION is set to 1,
* or the photon packet will be either transmitted or
* reflected determined statistically if PARTIALREFLECTION
* is set to 0.
* * Record the transmitted photon weight as reflection.
* * If "layer" is not the first or surrounding
* layer and the photon packet is transmitted,
* move the photon to "layer-1".
* * Update the photon parameters.
* */
void CrossUpOrNot(InputStruct * In_Ptr,
                   PhotonStruct * Photon_Ptr,
void CrossDnOrNot(InputStruct * In_Ptr, Photonstruct * Photon_Ptr, Outstruct * Out_Ptr)
{
    double uz = Photon_Ptr->uz; /* z directional cosine. */
    double uzl; /* cosines of transmission alpha. always positive. */
    double r=0.0; /* reflectance. */
    short layer = Photon_Ptr->layer;
    double ni = In_Ptr->layerspecs[layer].n;
    double nt = In_Ptr->layerspecs[layer+1].n;
    /* Get r. */
    if( - uz <= In_Ptr->layerspecs[layer].cos_crit0)
        r=1.0; /* total internal reflection. */
    else if (layer == In_Ptr->num_layers + 2) /* surr. glass. */
        r = RFresnel(In_Ptr->layerspecs[layer].n, In_Ptr->layerspecs[0].n, -uz, &uzl);
    else
        r = RFresnel(In_Ptr->layerspecs[layer].n, In_Ptr->layerspecs[layer+1].n, -uz, &uzl);
    if(uz > In_Ptr->layerspecs[layer].cos_crit1)
        r=1.0; /* total internal reflection. */
    else if(RandomNum() > r) /* reflected. */
    {
        Photon_Ptr->uz = -uz;
        RecordR(r, In_Ptr, Photon_Ptr, Out_Ptr);
        Photon_Ptr->uz = -uz;
        Photon_Ptr->dead = 1;
    }
    else /* transmitted. */
    {
        Photon_Ptr->uz = uz;
        if(RandomNum() > r) /* reflected. */
        {
            Photon_Ptr->uz = -uz;
        }
        else /* transmitted. */
        {
            Photon_Ptr->uz = uz;
            Photon_Ptr->layer--;  // Move the photon to the layer above.
            Photon_Ptr->ux *= ni/nt;
            Photon_Ptr->uy *= ni/nt;
            Photon_Ptr->uz = -uzl;
        }
    }
    else /* partially transmitted. */
    {
        Photon_Ptr->uz = -uzl;
        RecordT(r, In_Ptr, Photon_Ptr, Out_Ptr);
        Photon_Ptr->uz = -uz;
    }
    Photon_Ptr->layer--;  // Move the photon to the layer above.
    Photon_Ptr->ux *= ni/nt;
    Photon_Ptr->uy *= ni/nt;
    Photon_Ptr->uz = -uzl;
    Photon_Ptr->dead = 1;
}
#endif
/* Decide whether the photon will be transmitted or be reflected on the bottom boundary (uz>0) of the current layer. *
* If the photon is transmitted, move the photon to "layer+1". If "layer" is the last or surrounding layer, *
* record the transmitted weight as transmittance. *
* See comments for CrossUpOrNot. *
* UPDATE the photon parameters. *****/
void CrossDnOrNot(InputStruct * In_Ptr, PhotonStruct * Photon_Ptr, OutStruct * Out_Ptr)
else if(RandomNum() > r) /* transmitted. */
  Photon_Ptr->layer++;
  Photon_Ptr->ux *= ni/nt;
  Photon_Ptr->uy *= ni/nt;
  Photon_Ptr->uz = uzl;
}
else /* reflected. */
  Photon_Ptr->uz = uz;
#endif
else if(RandomNum() > r) /* transmitted. */
  if (layer == In_Ptr->num_layers ||
      layer == In_Ptr->num_layers + 2)
    Photon_Ptr->uz = uzl;
    RecordT(0.0, In_Ptr, Photon_Ptr, Out_Ptr);
    Photon_Ptr->dead = 1;
  else {
    Photon_Ptr->layer++;
    Photon_Ptr->ux *= ni/nt;
    Photon_Ptr->uy *= ni/nt;
    Photon_Ptr->uz = uzl;
  }
else /* reflected. */
  Photon_Ptr->ux = -ux;
#endif
넨
if(fabs(ux) <= In_Ptr->layerspecs[layer].cos_critxy)
  r=1.0; /* total internal reflection. */
else r = RFresnel(In_Ptr->layerspecs[layer].n, 
                 In_Ptr->layerspecs[In_Ptr->num_layers+2].n, fabs(ux), &ux1);

if(RandomNum() > r) {
  if( fabs(uy) <= In_Ptr->layerspecs[layer].cos_critxy) 
    r=1.0; /* total internal reflection. */
  else r = RFresnel(In_Ptr->layerspecs[layer].n, 
                   In_Ptr->layerspecs[In_Ptr->num_layers+2].n, fabs(uy), &uy1);

  if(RandomNum() > r) {
    if( fabs(uz) <= In_Ptr->layerspecs[layer].cos_critxy) 
      r=1.0; /* total internal reflection. */
    else r = RFresnel(In_Ptr->layerspecs[layer].n, 
                      In_Ptr->layerspecs[In_Ptr->num_layers+2].n, fabs(uz), &uz1);

    if(RandomNum() > r) {
      if( fabs(ux) <= In_Ptr->layerspecs[layer].cos_critxy) 
        r=1.0; /* total internal reflection. */
      else r = RFresnel(In_Ptr->layerspecs[layer].n, 
                        In_Ptr->layerspecs[In_Ptr->num_layers+2].n, fabs(ux), &ux1);

      if(RandomNum() > r) {
        if( fabs(uy) <= In_Ptr->layerspecs[layer].cos_critxy) 
          r=1.0; /* total internal reflection. */
        else r = RFresnel(In_Ptr->layerspecs[layer].n, 
                          In_Ptr->layerspecs[In_Ptr->num_layers+2].n, fabs(uy), &uy1);

        if(RandomNum() > r) {
          if( fabs(uz) <= In_Ptr->layerspecs[layer].cos_critxy) 
            r=1.0; /* total internal reflection. */
          else r = RFresnel(In_Ptr->layerspecs[layer].n, 
                            In_Ptr->layerspecs[In_Ptr->num_layers+2].n, fabs(uz), &uz1);

        }
else /* reflected. */
Photon_Ptr->uy = -uy;
}

void CrossOrNot(InputStruct * In_Ptr,
PhotonStruct * Photon_Ptr,
outStruct * Out_Ptr,
short hitcase)
{
    if (hitcase == 1)
        CrossXOrNot(In_Ptr, Photon_Ptr, Out_Ptr);
    else if (hitcase == 2)
        CrossYOrNot(In_Ptr, Photon_Ptr, Out_Ptr);
    else if (hitcase == 3) {
        if(Photon_Ptr->uz < 0.0)
            CrossUpOrNot(In_Ptr, Photon_Ptr, Out_Ptr);
        else
            CrossDnOrNot(In_Ptr, Photon_Ptr, Out_Ptr);
    }
}

void HopInGlass(InputStruct * In_Ptr,
PhotonStruct * Photon_Ptr,
OutStruct * Out_Ptr)
{
    short hitcase; /* the boundary which is hit. */
    if(Photon_Ptr->uz == 0.0)
        /* horizontal photon in glass is killed. */
        Photon_Ptr->dead = 1;
    else {
        if(Photon_Ptr->layer == In_Ptr->num_layers + 2)
            /* surrounding glass */
            StepSizeInSurfGlass(Photon_Ptr, In_Ptr);
            hitcase = 3; /* always in surrounding glass */
            CrossOrNot(In_Ptr, Photon_Ptr, Out_Ptr, hitcase);
        else /* glass layer which is not the surrounding glass layer */
            if(hitcase == StepSizeInGlass(Photon_Ptr, In_Ptr))
                Hop(Photon_Ptr);
            else
                CrossOrNot(In_Ptr, Photon_Ptr, Out_Ptr, hitcase);
    }
}

void HopDropSpinInTissue(InputStruct * In_Ptr,
PhotonStruct * Photon_Ptr,
OutStruct * Out_Ptr)
{
    short hitcase; /* the boundary which is hit. */
    if(hitcase == 0) /* if no boundary is hit */
        StepSizeInTissue(Photon_Ptr, In_Ptr);
    hitcase = HitBoundary(Photon_Ptr, In_Ptr);

    if (hitcase > 0) {
        Hop(Photon_Ptr); /* move to boundary plane. */
        CrossOrNot(In_Ptr, Photon_Ptr, Out_Ptr, hitcase);
    } else {
        Hop(Photon_Ptr);
        Drop(In_Ptr, Photon_Ptr, Out_Ptr);
        Spin(In_Ptr->layerspecs[Photon_Ptr->layer].mua,
            Photon_Ptr->layer,
            Photon_Ptr);
    }
}

void CrossOrNot(InputStruct * In_Ptr,
PhotonStruct * Photon_Ptr,
OutStruct * Out_Ptr)
{
    short layer = Photon_Ptr->layer;
    if((In_Ptr->layerspecs[layer].mua == 0.0)
        & (In_Ptr->layerspecs[layer].mua == 0.0))
B.1.2. TMMCML

B.1.2.1. mcml.h

```c
if( Photon_Ptr->w < In_Ptr->Wth && !Photon_Ptr->dead) 
Roulette(Photon_Ptr);
```

- Top-down design, keep each subroutine clear & short.
- Reflectance and transmittance are angularly resolved.

*** Modifications when including scattering particles made 1996 by P. Alsholm and A. Nilsson.

- General Naming Conventions:
  - Preprocessor names: all capital letters, e.g. `#define PREPROCESSORS`
  - Globals: first letter of each word is capital, no underscores,
    - e.g. `short GlobalVar;
  - Dummy variables: first letter of each word is capital, and words are connected by underscores,
    - e.g. `void NiceFunction(char Dummy_Var);`
  - Local variables: all lower cases, words are connected by underscores,
    - e.g. `short local_var;
  - Function names or data types: same as Globals.

- Dimension of length: cm.

```c
#define PI 3.1415926
#define WEIGHT 1E-4
#define CHANCE 0.1
```

- Critical weight for roulette.
- Chance of roulette survival.
#define STRLEN 256
/* String length. */
#define Boolean char
#define SIGN(X) ((X)>=0 ? 1:-1)

/* */

struct PhotonStruct {
    double x, y, z; /* Cartesian coordinates. [cm] */
    double ux, uy, uz; /* directional cosines of a photon. */
    double w; /* weight. */
    Boolean dead; /* 1 if photon is terminated. */
    short layer; /* index to layer where the photon packet resides. */
    double s; /* current step size. [cm]. */
    double sleft; /* step size left. dimensionless [-]. */
} PhotonStruct;

/* */

struct LayerStruct {
    double z0, z1; /* z coordinates of a layer. [cm]. */
    double n; /* refractive index of a layer. */
    double abscross; /* absorption cross section. [-]. */
    double meanstep; /* the step length. [cm]. */
    double meanstepdev; /* deviation of step length. [cm]. */
    double bcux; /* orientation of blood cell. */
    double bcuy; /* orientation of blood cell. */
    double bcuz; /* orientation of blood cell. */
    double dev; /* deviation of that orientation. [0.0-1.0]. */
    double cos_cril0, cos_cril1, cos_critxy;
} LayerStruct;

/* */

typedef struct {
    char out_fname[STRLEN]; /* output file name. */
    char out__fformat; /* output file format. */
    char dat_fname[STRLEN]; /* data base file name. */
    short nbrangint; /* data base parameter, number of incident angles. */
    short nbrthsc; /* data base parameter, number of scattering theta. */
    short nbrpssc; /* data base parameter, number of scattering psi. */
    double ** valmatrix; /* data base values. */
    long num__photons; /* to be traced. */
    double Wth; /* play roulette if photon */
    double dz; /* z grid separation. [cm]. */
    double dr; /* r grid separation. [cm]. */
    double da; /* alpha grid separation. [radian]. */
    double dz; /* weight < Wth. */
    double ns; /* array range 0..nz-1. */
    double nr; /* array range 0..nr-1. */
    double na; /* array range 0..na-1. */
    short num_layers; /* number of layers. */
    double x_size, y__size; /* dimension of the sample. [cm]. */
} InputStruct;

/* */

/* Input parameters for each independent run. */
/* z and r are for the cylindrical coordinate system. [cm] */
/* a is for the angle alpha between the photon exiting */
/* direction and the surface normal. [radian] */
/* The grid line separations in z, r, and alpha */
/* directions are dz, dr, and da respectively. The numbers */
/* of grid lines in z, r, and alpha directions are */
/* nz, nr, and na respectively. */
/* The member layerspecs will point to an array of */
/* structures which store parameters of each layer. */
/* This array has (num_layers + 3) elements. One */
/* element is for a layer. */
/* The layers 0 and (num_layers + 1) are for top ambient */
/* medium and the bottom ambient medium respectively. */
/* The layer (num_layers + 2) is for the surrounding glass. */
/* */

/*
 * Structures for scoring physical quantities.
 * z and r represent z and r coordinates of the
 * cylindrical coordinate system. [cm]
 * a is the angle alpha between the photon exiting
 * direction and the normal to the surfaces. [radian]
 * See comments of the InputStruct.
 * See manual for the physical quantities.
 */
typedef struct {
  double Rsp;  /* specular reflectance. [-] */
  double ** Rd_ra; /* 2D distribution of diffuse */
       /* reflectance. [1/(cm2 sr)] */
  double * Rd_r; /* 1D radial distribution of diffuse */
       /* reflectance. [1/cm2] */
  double * Rd_a; /* 1D angular distribution of diffuse */
       /* reflectance. [1/cm] */
  double ** A_rz; /* 2D probability density in turbid */
       /* media over r & z. [1/cm3] */
  double * A_z; /* 1D probability density over z. */
       /* [1/cm] */
  double Rd; /* total diffuse reflectance. [-] */
  double * Tt_ra; /* 2D distribution of total */
       /* transmittance. [1/cm2 sr] */
  double * Tt_r; /* 1D radial distribution of */
       /* transmittance. [1/cm2] */
  double * Tt_a; /* 1D angular distribution of */
       /* transmittance. [1/sr] */
  double Tt; /* total transmittance. [-] */
} OutStruct;

/B.1.2.2. mcmlmain.c

#include "mcml.h"

/*
 * main program for Monte Carlo simulation of photon
 * distribution in multi-layered turbid media.
 */

/****
 * THINKCPROFILER is defined to generate profiler calls in
 * Think C. If 1, remember to turn on 'Generate profiler
 * calls' in the options menu.
 ****/
#define THINKCPROFILER 0

/* GNU cc does not support difftime() and CLOCKS_PER_SEC. */
#define GNUCC 0

#include <profile.h>
#include <console.h>
#endif

/* Declare before they are used in main(). */
void ShowVersion(char *);
FILE *GetFile(char *);
short ReadNumRuns(FILE *);
void ReadParm(FILE *, InputStruct *);
void CheckParm(FILE *, InputStruct *);
void ReadDataBase(FILE *, InputStruct *);
void InitOutputData(InputStruct, Outstruct *);
void FreeData(InputStruct, Outstruct *);
void LaunchPhoton(double, LayerStruct *, PhotonStruct *);
void HopDropSpin(InputStruct *, PhotonStruct *, Outstruct *);
void SumScaleResult(Inputstruct, outStruct *);
void WriteResult(InputStruct, OutStruct, char *);
If \( F = 0 \), reset the clock and return 0.

If \( F = 1 \), pass the user time to \( \text{Msg} \) and print \( \text{Msg} \) on screen, return the real time since \( F=0 \).

If \( F = 2 \), same as \( F=1 \) except no printing.

Note that \( \text{clock()} \) and \( \text{time()} \) return user time and real time respectively.

User time is whatever the system allocates to the running of the program; real time is wall-clock time. In a time-shared system, they need not be the same.

\( \text{clock()} \) only hold 16 bit integer, which is about 32768 clock ticks.

```c
void PunchTime(char F, char *Msg)
{
#if GNUCC
    return(0);
#else
    static clock_t utO; /* user time reference. */
    static time_t rtO; /* real time reference. */
    double sees;
    char s[STRLEN];
    if(F==0) {
        utO = clock();
        rtO = time(NULL);
        return(0);
    } else if(F==1) {
        sees = (clock() - utO)/(double)CLOCKS_PER_SEC;
        if (sees<0) sees=0; /* clock() can overflow. */
        sprintf(s, "User time: %8.0lf sec = %8.2lf hr. %s\n", sees, sees/3600.0, Msg);
        puts(s);
        strcpy(Msg, s);
        return(difftime(time(NULL), rtO));
    } else if(F==2) return(difftime(time(NULL), rtO));
#else return(0);
#endif
}
```

Print the current time and the estimated finishing time.

\( P_i \) is the number of computed photon packets.

```c
void PredictDoneTime(long P1, long Pt)
{
    time_t now, done_time;
    struct tm *date;
    char s[80];
    now = time(NULL);
    date = localtime(&now);
    strftime(s, 80, "%H:%M %x", date);
    printf("Now %s, ", s);
    done_time = now + (time_t)(PunchTime(2,"")*(Pt-P1)/(double)Pt);
    date = localtime(&done_time);
    strftime(s, 80, "%H:%M %x", date);
    printf("End %s\n", s);
}
```

Report estimated time, number of photons and runs left after calculating 10 photons or every 1/10 of total number of photons.

```c
void ReportStatus(short Num_Runs, long Pi, long Pt)
{
    if(Pt-Pi == 10 || Pi*10%Pt == 0 && Pi != Pt) {
        printf("%ld photons & %hd runs left, \n", Pi, Num_Runs);
        PredictDoneTime(Pt-Pi, Pt);
    }
}
```

Report time and write results.

```c
void ReportResult(InputStruct In_Parm, outStruct Out_Parm)
{
    char time_report[STRLEN];
    strcpy(time_report, "Simulation time of this run.\n");
    PunchTime(1, time_report);
    SumScaleResult(In_Parm, &Out_Parm);
    WriteResult(In_Parm, Out_Parm, time_report);
}
Get the file name of the input data file from the argument to the command line.

```c
void GetFnameFromArgv(int argc, char * argv[], char * input_filename)
{
    if(argc>=2) {
        /* filename in command line */
        strcpy(input_filename, argv[1]);
    } else
        input_filename[0] = '\0';
}
```

Execute Monte Carlo simulation for one independent run.

```c
void DoOneRun(short NumRuns, InputStruct *In_Ptr)
{
    register long i_photon;
    /* index to photon. register for speed. */
    outstruct out_parm; /* distribution of photons. */
    PhotonStruct photon;
    #if THINKCPROFILER
    InitProfile(200,200); cecho2file("prof.rpt",O, stdout);
    #endif
    InitOutputData(*In_Ptr, &out_parm);
    out_parm.Rsp = Rspecular(In_Ptr->layerspecs);
    i_photon = In_Ptr->num_photons;
    PunchTime(0, "");
    do {
        ReportStatus(NumRuns, i_photon, In_Ptr->num_photons);
        LaunchPhoton(out_parm.Rsp, In_Ptr->layerspecs, &photon);
        do HopDropSpin(In_Ptr, &photon, &out_parm);
        while (!photon.dead);
    } while (--i_photon);
    #if THINKCPROFILER
    exit(0);
    #endif
    ReportResult(*In_Ptr, out_parm);
    FreeData(*In_Ptr, &out_parm);
}
```

Input/output of data.

```c
#include "mcml.h"
do HopDropSpin(In_Ptr, &photon, &out_parm);
while (!photon.dead);
} while (--i_photon);
```

Structure used to check against duplicated file names.

```c
#include "mcml.h"
```

B.1.2.3. mcmlio.c

```c
#include "mcml.h"
do HopDropSpin(In_Ptr, &photon, &out_parm);
while (!photon.dead);
} while (--i_photon);
```
typedef struct NameList NameNode;
typedef NameNode * NameLink;

/***********************************************************
* Center a string according to the column width.
****/
char * CenterStr(short Wid,
    char * InStr,
    char * Outstr)
{
    size_t nspaces; /* number of spaces to be filled */
    /* before InStr. */

    nspaces = (Wid - strlen(InStr))/2;
    if(nspaces<0) nspaces = 0;
    strcpy(Outstr, "\n");
    while(nspaces--)
        strcat(OutStr, " ");
    strcat(OutStr, InStr);
    return (Outstr);
}

/***********************************************************
* Print some messages before starting simulation.
* e.g. author, address, program version, year.
****/
#define COLWIDTH 80
void ShowVersion(char *version)
{
    char str[STRELEN];
    CenterStr(COLWIDTH, "Monte Carlo Simulation of Multi-layered Turbid Media", str);
    puts(str);
    puts("\n");
    CenterStr(COLWIDTH, "Lihong Wang, Ph. D.", str);
    puts(str);
    CenterStr(COLWIDTH, "Steven L. Jacques, Ph. D.", str);
    puts(str);
    CenterStr(COLWIDTH, "Laser Biology Research Laboratory - 17", str);
    puts(str);
    CenterStr(COLWIDTH, "M.D. Anderson Cancer Center", str);
    puts(str);
    CenterStr(COLWIDTH, "University of Texas", str);
    puts(str);
    CenterStr(COLWIDTH, "Houston, TX 77030", str);
    puts(str);
    CenterStr(COLWIDTH, "Fax: (713)792-3995", str);
    puts(str);
    puts("\n");
    CenterStr(COLWIDTH, "Modified 1996, PA - tmmcl", str);
    puts(str);
    puts("\n");
    CenterStr(COLWIDTH, "Laser Biology Research Laboratory-17", str);
    puts(str);
    CenterStr(COLWIDTH, "M.D. Anderson Cancer Center", str);
    puts(str);
    CenterStr(COLWIDTH, "University of Texas", str);
    puts(str);
    CenterStr(COLWIDTH, "Houston, TX 77030", str);
    puts(str);
    CenterStr(COLWIDTH, "Fax: (713)792-3995", str);
    puts(str);
    puts("\n");
    #undef COLWIDTH

/***********************************************************
* Get a filename and open it for reading, retry until
* the file can be opened. \"\." terminates the program.
* If Fname != NULL, try Fname first.
****/
FILE *GetFile(char *Fname)
{
    FILE * file=NULL;
    Boolean firsttime=1;
    do {
        if(firsttime && Fname[0] == '\0') {
            /* use the filename from command line */
            firsttime = 0;
        }
        else {
            printf("Input filename(or . to exit):\n");
            scanf("%s", Fname);
            firsttime = 0;
        }
        if(strlen(Fname) == 1 && Fname[0] == '.')
            exit(1); /* exit if no filename entered. */
    } while(1);
```c
file = fopen(Fname, "r");
} while(file == NULL);
return(file);
*/

//***********************************************************
/* Kill the ith char (counting from 0), push the following
/* chars forward by one.
*****/
void KillChar(size_t i, char * Str)
{
    size_t sl = strlen(Str);
    for(;i<sl;i++) Str[i] = Str[i+1];
}
//***********************************************************
/* Eliminate the chars in a string which are not printing
/* chars or spaces.
/* Spaces include ' ', '', '	' etc.
/* Return 1 if no nonprinting chars found, otherwise
/* return 0.
*****/
Boolean CheckChar(char * Str)
{
    Boolean found = 0; /* found bad char. */
    size_t sl = strlen(Str);
    size_t i=0;
    while(i<sl)
    if (Str[i]<0 || Str[i]>255)
        nerror("Non-ASCII file\n");
    else if(isprint(Str[i]) || isspace(Str[i]))
        i++;
    else /* found 1;
            KillChar(i, Str);
            sl--;
    return(found);
}
//***********************************************************
/* Return 1 if this line is a comment line in which the
/* first non-space character is ".#."
*****/
Boolean CommentLine(char *Buf)
{
    size_t spn, cspn;
    spn = strspn(Buf, " \\
"); /* length spanned by space or tab chars. */
    cspn = strcspn(Buf, ".#\n"); /* length before the 1st # or return. */
    if(spn == cspn) /* comment line or space line. */
        return(1);
    else /* the line has data. */
        return(0);
}
//***********************************************************
/* Skip space or comment lines and return a data line only.
*****/
char * FindDataLine(FILE *File_Ptr)
{
    char buf[STRLEN];
    buf[0] = '\0';
    do {
        /* skip space or comment lines. */
        if(fgets(buf, 255, File_Ptr) == NULL) {
            printf("Incomplete data\n");
            buf[0]='\0';
            break;
        }
        else
            CheckChar(buf);
    while(CommentLine(buf));
    return(buf);
}
//***********************************************************
/* Also return 1 if this line is space line.
*****/
short ReadNumRuns(FILE* File_Ptr)
{
    char buf[STRLEN];
    short n=0;
    FindDataLine(File_Ptr); /* skip the file version. */
```
strcpy(buf, FindDataLine(File_Ptr));
if(buf[0]=='\0') nrerror("Reading number of runs\n");
scanf(buf, "%hd", &n);
return(n);
}

/***************************************************************
* Place the data base values in a matrix. *
* Valmatrix(i,j): j = [1,nbrangint]  
The different incident angles [0,PI/2]. *
* i = [1,num_rows]: num_rows = (nbrthsc-2)*nbrpssc-1) *
* Valmatrix(1,*) : back scattering probability. *
* Valmatrix(2,*) - Valmatrix(1,*) : 
  probability of scattering in direction  
  theta = 1/(nbrthsc-1), psi = 0. *
* Valmatrix(n,*) - Valmatrix(n-1,*) : 
  probability of scattering in direction  
  theta = ((n-2)*nbrpssc+1)/(nbrthsc-1),  
  psi = ((n-2)/nbrpssc)/(nbrthsc-1). *
* Valmatrix((num_rows,*)  : 1 - forward scattering probability. *
* For forward and back scattering, psi is arbitrary. 
***************************************************************
void PlaceDataInMatrix(InputStruct * In_Ptr)
{
  char buf[STRLEN];
  short nbthsc = In_Ptr->nbthsc;  
  short nbspssc = In_Ptr->nbspssc;  
  short num_thetacol = In_Ptr->num_thetacol;  
  short row = 0;  
  short num_rows;  
  double value;
  FILE * Datfile_Ptr;
  num_rows = (nbthsc-2)*nbspssc + 1; /* number of data values for each incident angle */  
  /* number of data values for each incident angle according  */  
  /* to the remarks in MCDATA.FOR. */  

  In_Ptr->valmatrix = AllocMatrix(1,num_rows,1,num_thetacol);

  Datfile_Ptr = fopen(In_Ptr->dat_fname,"r");
  if (num_thetacol==1) /* sphere. */
    for (row=1; row <= num_rows; row++)
    {
     strcpy(buf, FindDataLine(Datfile_Ptr));
     if(buf[0]=='\0') nrerror("Reading data base value.\n");
     scanf(buf, "%lf", &value);
     In_Ptr->valmatrix[row][thetacol] = value;
    }
  else /* spheroid. */
    for (thetacol=1; thetacol<=num_thetacol; thetacol++)
      for (row=1; row <= num_rows; row++)
      {
       strcpy(buf, FindDataLine(Datfile_Ptr));
       if(buf[0]=='\0') nrerror("Reading data base value.\n");
       scanf(buf, "%lf", &value);
       In_Ptr->valmatrix[row][thetacol] = value;
      }
  fclose(Datfile_Ptr);
}

/***************************************************************
* Read the data base. *
***************************************************************
void ReadDataBase(FILE * File_Ptr, InputStruct *In_Ptr)
{
  char buf[STRLEN];
  /* read in file name, nbrangint. */
  strcpy(buf, FindDataLine(File_Ptr));
  if(buf[0]=='\0') nrerror("Reading data base parameters.\n");
  scanf(buf, "%s %hd", In_Ptr->dat_fname, &In_Ptr->nbrangi.ntl;
  if(In_Ptr->nbrangint<=0)
    nrerror("Nonpositive number of angles.\n");
  /** read in nbrthsc, nbspssc. **/
  strcpy(buf, FindDataLine(File_Ptr));
  if(buf[0]=='\0') nrerror("Reading data base parameters.\n");
  scanf(buf, " %hd %hd", &In_Ptr->nbthsc, &In_Ptr->nbspssc);
  if(In_Ptr->nbthsc<=1 || In_Ptr->nbspssc<=1)
    nrerror("Wrong number of angles.\n");
  PlaceDataInMatrix(In_Ptr);
}

/***************************************************************
* Read the file name and the file format. 

116
The file format can be either A for ASCII or B for binary.

```c
void ReadFnameFormat(FILE *File_Ptr, InputStruct *In_Ptr)
{
    char buf[STRLEN];
    /** read in file name and format. **/
    strcpy(buf, FindDataLine(File_Ptr));
    if(buf[0]=='\0')
        nrerror("Reading file name and format.\n");
    sscanf(buf, "%s %c", In_Ptr->out_fname, &In_Ptr->out_fformat);
    if(toupper(In_Ptr->out_fformat) != 'B')
        In_Ptr->out_fformat = 'A';
}
```
Read the refractive index $n$ of the ambient.

```c
void ReadAmbient(FILE *File_Ptr,
    LayerStruct * Layer_Ptr,
    char *side)
{
    char buf[STRLEN], msg[STRLEN];
    double n;
    strcpy(buf, FindDataLine(File_Ptr));
    if(buf[0]=='\0') { 
        sprintf(msg, "Reading n of %s ambient.\n", side);
        nrerror(msg);
    }
    sscanf(buf, "%lf", &n);
    if(n<=0 ||
        temp = sqrt(bcux*bcux+bcuy*bcuy+bcuz*bcuz); 
    if (temp == 0.0) { /* no orientation. */
        Layer_Ptr->dev = 1.0;
        Layer_Ptr->bcux = 0.0;
        Layer_Ptr->bcuy = 0.0;
        Layer_Ptr->bcuz = 0.0;
    } else {
        Layer_Ptr->bcux = bcux/temp; /* normalization. */
        Layer_Ptr->bcuy = bcuy/temp;
        Layer_Ptr->bcuz = bcuz/temp;
        Layer_Ptr->dev = dev;
    }
    Layer_Ptr->z0 = *Z_Ptr;
    *Z_Ptr += d;
    return(0);
}
```

Read and set the parameters of the surrounding glass.

```c
Boolean ReadSurrGlass(FILE *File_Ptr,
    char buf[STRLEN];
    double n;
    strcpy(buf, FindDataLine(File_Ptr));
    if(buf[0]=='\0') return(1); /* error. */
```
if(buf[0]=='\0') return(1); /* error. */
sscanf(buf, "%lf", &n);
if(n<0)
    return(1); /* error. */
Layer_Ptr->n = n;
Layer_Ptr->meanstep = 0.0;
Layer_Ptr->z0 = 0.0;
Layer_Ptr->z1 = z_Ptr;
return(1);
}

/***********************************************************
* Read the parameters of one layer at a time. ************
*/
void ReadLayerSpecs(FILE *File_Ptr, short Num_Layers,
                     LayerStruct **Layerspecs_PP)
{
    char msg[STRLEN];
    short i=0;
    double z = 0.0; /* z coordinate of the current layer. */
    /* Allocate an array for the layer parameters. */
    /* layer 0 and layer Num_Layers + 1 are for ambient. */
    /* Num_Layers + 2 is for the surrounding glass. */
    *Layerspecs_PP = (LayerStruct *)malloc((unsigned) (Num_Layers+3)*sizeof(LayerStruct));
    if (!(*Layerspecs_PP)
        nerror("allocation failure in ReadLayerSpecs()"));
ReadAmbient(File_Ptr, &(*Layerspecs_PP)[1], "top");
for(i=1; i<Num_Layers; i++)
    if (ReadOneLayer(File_Ptr, &(*Layerspecs_PP)[i], &z))
        sprintf(msg, "Error reading %hd of %hd layers\n", i, Num_Layers);
        nerror(msg);
    ReadAmbient(File_Ptr, &(*Layerspecs_PP)[i], "bottom");
if(ReadSurfGlass(File_Ptr, &(*Layerspecs_PP)[i+1], &z))
    sprintf(msg, "Error reading surrounding glass");
    nerror(msg);
}

/***********************************************************
* Compute the critical angles for total internal reflection according to the relative refractive index
* of the layer.  
* All layers are processed. 
*****/
void CriticalAngle( short Num_Layers, 
                   LayerStruct ** Layerspecs_PP)
{
    short i=0;
    double n1, n2;
    for(i=1; i<Num_Layers; i++)
        n1 = (*Layerspecs_PP)[i].n;
        n2 = (*Layerspecs_PP)[i-1].n;
        (*Layerspecs_PP)[i].cos_crit0 = n1>n2 ?
        sqrt(1.0 - n2*n2/(n1*n1)) : 0.0;
        n2 = (*Layerspecs_PP)[i+1].n;
        (*Layerspecs_PP)[i].cos_crit1 = n1>n2 ?
        sqrt(1.0 - n2*n2/(n1*n1)) : 0.0;
        n2 = (*Layerspecs_PP)[Num_Layers+2].n;
        (*Layerspecs_PP)[i].cos_critxy = n1>n2 ?
        sqrt(1.0 - n2*n2/(n1*n1)) : 0.0;
        n1 = (*Layerspecs_PP)[Num_Layers+2].n; /* surrounding glass */
        n2 = (*Layerspecs_PP)[0].n;
        (*Layerspecs_PP)[Num_Layers+2].cos_crit0 = n1>n2 ?
        sqrt(1.0 - n2*n2/(n1*n1)) : 0.0;
        n2 = (*Layerspecs_PP)[Num_Layers+1].n;
        (*Layerspecs_PP)[Num_Layers+2].cos_crit1 = n1>n2 ?
        sqrt(1.0 - n2*n2/(n1*n1)) : 0.0;
    }

/***********************************************************
* Read in the input parameters for one run. ************
*/
void ReadParm(FILE *File_Ptr, InputStruct * In_Ptr)
{
    In_Ptr->Wth = WEIGHT;
    ReadFnameFormat(File_Ptr, In_Ptr);
    ReadNumPhotons(File_Ptr, In_Ptr);
    ReadDzDr(File_Ptr, In_Ptr);
    ReadNzNrNa(File_Ptr, In_Ptr);
    ReadXSizeYSize(File_Ptr, In_Ptr);
    ReadNumLayers(File_Ptr, In_Ptr);
    ReadLayerSpecs(File_Ptr, In_Ptr->num_layers, &In_Ptr->layerspecs);
    CriticalAngle(In_Ptr->num_layers, &In_Ptr->layerspecs);
}
Boolean NameInList(char *Name, NameLink List)
{
    while (List != NULL)
    {
        if(strcmp(Name, List->name) == 0)
            return(1);
        List = List->next;
    }
    return(0);
}

int AddNameToList(char *Name, NameLink * List_Ptr)
{
    NameLink list = *List_Ptr;
    if(list == NULL) /* first node. */
    {
        *List_Ptr = list = (NameLink)malloc(sizeof(NameNode));
        strcpy(list->name, Name);
        list->next = NULL;
        return(0);
    }
    else /* subsequent nodes. */
    {
        /* Move to the last node. */
        while(list->next != NULL)
            list = list->next;
        /* Append a node to the list. */
        list->next = (NameLink)malloc(sizeof(NameNode));
        strcpy(list->name, Name);
        list->next = NULL;
    }
    return(0);
}

void CheckParm(FILE* File_Ptr, InputStruct * In_Ptr)
{
    short i_run;
    short num_runs; /* number of independent runs. */
    NameLink head = NULL;
    Boolean name_taken; /* output files share the same */
    char msg[STRLEN];
    num_runs = ReadNumRuns(File_Ptr);
    for(i_run=1; i_run<=num_runs; i_run++)
    {
        printf("Checking input data for run %hd\n", i_run);
        FindDataLine(File_Ptr); /* skip the data base parameters.*/
        FindDataLine(File_Ptr); /* */
        ReadParm(File_Ptr, In_Ptr);
        name_taken = FnameTaken(In_Ptr->out fname, &head);
        if(name_taken)
            sprintf(msg, "file name %s duplicated.\n", In_Ptr->out_fname);
        else {
            AddNameToList(fname, List_Ptr);
            return(0);
        }
    } // while
}

Boolean FnameTaken(char *fname, NameLink * List_Ptr)
{
    if(NameInList(fname, *List_Ptr))
        return(1);
    else
        AddNameToList(fname, List_Ptr);
        return(0);
}

void FreeFnameList(NameLink List)
{
    NameLink next;
    while(List != NULL)
    {
        next = List->next;
        free(List);
        List = next;
    }
}

return(1, if the name in the name list.
return 0, otherwise.****!

Boolean NameInList(char *Name, NameLink List)
{
    while (List != NULL)
    {
        if(strcmp(Name, List->name) == 0)
            return(1);
        List = List->next;
    }
    return(0);
}

int AddNameToList(char *Name, NameLink * List_Ptr)
{
    NameLink list = *List_Ptr;
    if(list == NULL) /* first node. */
    {
        *List_Ptr = list = (NameLink)malloc(sizeof(NameNode));
        strcpy(list->name, Name);
        list->next = NULL;
        return(0);
    }
    else /* subsequent nodes. */
    {
        /* Move to the last node. */
        while(list->next != NULL)
            list = list->next;
        /* Append a node to the list. */
        list->next = (NameLink)malloc(sizeof(NameNode));
        strcpy(list->name, Name);
        list->next = NULL;
    }
    return(0);
}

void CheckParm(FILE* File_Ptr, InputStruct * In_Ptr)
{
    short i_run;
    short num_runs; /* number of independent runs. */
    NameLink head = NULL;
    Boolean name_taken; /* output files share the same */
    char msg[STRLEN];
    num_runs = ReadNumRuns(File_Ptr);
    for(i_run=1; i_run<=num_runs; i_run++)
    {
        printf("Checking input data for run %hd\n", i_run);
        FindDataLine(File_Ptr); /* skip the data base parameters.*/
        FindDataLine(File_Ptr); /* */
        ReadParm(File_Ptr, In_Ptr);
        name_taken = FnameTaken(In_Ptr->out fname, &head);
        if(name_taken)
            sprintf(msg, "file name %s duplicated.\n", In_Ptr->out_fname);
        else {
            AddNameToList(fname, List_Ptr);
            return(0);
        }
    } // while
}

Boolean FnameTaken(char *fname, NameLink * List_Ptr)
{
    if(NameInList(fname, *List_Ptr))
        return(1);
    else
        AddNameToList(fname, List_Ptr);
        return(0);
}

void FreeFnameList(NameLink List)
{
    NameLink next;
    while(List != NULL)
    {
        next = List->next;
        free(List);
        List = next;
    }
}

return(1, if the name in the name list.
return 0, otherwise.****!

Boolean NameInList(char *Name, NameLink List)
{
    while (List != NULL)
    {
        if(strcmp(Name, List->name) == 0)
            return(1);
        List = List->next;
    }
    return(0);
}

int AddNameToList(char *Name, NameLink * List_Ptr)
{
    NameLink list = *List_Ptr;
    if(list == NULL) /* first node. */
    {
        *List_Ptr = list = (NameLink)malloc(sizeof(NameNode));
        strcpy(list->name, Name);
        list->next = NULL;
        return(0);
    }
    else /* subsequent nodes. */
    {
        /* Move to the last node. */
        while(list->next != NULL)
            list = list->next;
        /* Append a node to the list. */
        list->next = (NameLink)malloc(sizeof(NameNode));
        strcpy(list->name, Name);
        list->next = NULL;
    }
    return(0);
}

void CheckParm(FILE* File_Ptr, InputStruct * In_Ptr)
{
    short i_run;
    short num_runs; /* number of independent runs. */
    NameLink head = NULL;
    Boolean name_taken; /* output files share the same */
    char msg[STRLEN];
    num_runs = ReadNumRuns(File_Ptr);
    for(i_run=1; i_run<=num_runs; i_run++)
    {
        printf("Checking input data for run %hd\n", i_run);
        FindDataLine(File_Ptr); /* skip the data base parameters.*/
        FindDataLine(File_Ptr); /* */
        ReadParm(File_Ptr, In_Ptr);
        name_taken = FnameTaken(In_Ptr->out fname, &head);
        if(name_taken)
            sprintf(msg, "file name %s duplicated.\n", In_Ptr->out_fname);
        else {
            AddNameToList(fname, List_Ptr);
            return(0);
        }
    } // while
}

Boolean FnameTaken(char *fname, NameLink * List_Ptr)
{
    if(NameInList(fname, *List_Ptr))
        return(1);
    else
        AddNameToList(fname, List_Ptr);
        return(0);
}

void FreeFnameList(NameLink List)
{
    NameLink next;
    while(List != NULL)
    {
        next = List->next;
        free(List);
        List = next;
    }
}

return(1, if the name in the name list.
return 0, otherwise.****!

Boolean NameInList(char *Name, NameLink List)
{
    while (List != NULL)
    {
        if(strcmp(Name, List->name) == 0)
            return(1);
        List = List->next;
    }
    return(0);
}

int AddNameToList(char *Name, NameLink * List_Ptr)
{
    NameLink list = *List_Ptr;
    if(list == NULL) /* first node. */
    {
        *List_Ptr = list = (NameLink)malloc(sizeof(NameNode));
        strcpy(list->name, Name);
        list->next = NULL;
        return(0);
    }
    else /* subsequent nodes. */
    {
        /* Move to the last node. */
        while(list->next != NULL)
            list = list->next;
        /* Append a node to the list. */
        list->next = (NameLink)malloc(sizeof(NameNode));
        strcpy(list->name, Name);
        list->next = NULL;
    }
    return(0);
}

void CheckParm(FILE* File_Ptr, InputStruct * In_Ptr)
{
    short i_run;
    short num_runs; /* number of independent runs. */
    NameLink head = NULL;
    Boolean name_taken; /* output files share the same */
    char msg[STRLEN];
    num_runs = ReadNumRuns(File_Ptr);
    for(i_run=1; i_run<=num_runs; i_run++)
    {
        printf("Checking input data for run %hd\n", i_run);
        FindDataLine(File_Ptr); /* skip the data base parameters.*/
        FindDataLine(File_Ptr); /* */
        ReadParm(File_Ptr, In_Ptr);
        name_taken = FnameTaken(In_Ptr->out fname, &head);
        if(name_taken)
            sprintf(msg, "file name %s duplicated.\n", In_Ptr->out_fname);
        else {
            AddNameToList(fname, List_Ptr);
            return(0);
        }
    } // while
}

Boolean FnameTaken(char *fname, NameLink * List_Ptr)
{
    if(NameInList(fname, *List_Ptr))
        return(1);
    else
        AddNameToList(fname, List_Ptr);
        return(0);
}

void FreeFnameList(NameLink List)
{
    NameLink next;
    while(List != NULL)
    {
        next = List->next;
        free(List);
        List = next;
    }
}

return(1, if the name in the name list.
return 0, otherwise.****!

Boolean NameInList(char *Name, NameLink List)
{
    while (List != NULL)
    {
        if(strcmp(Name, List->name) == 0)
            return(1);
        List = List->next;
    }
    return(0);
}

int AddNameToList(char *Name, NameLink * List_Ptr)
{
    NameLink list = *List_Ptr;
    if(list == NULL) /* first node. */
    {
        *List_Ptr = list = (NameLink)malloc(sizeof(NameNode));
        strcpy(list->name, Name);
        list->next = NULL;
        return(0);
    }
    else /* subsequent nodes. */
    {
        /* Move to the last node. */
        while(list->next != NULL)
            list = list->next;
        /* Append a node to the list. */
        list->next = (NameLink)malloc(sizeof(NameNode));
        strcpy(list->name, Name);
        list->next = NULL;
    }
    return(0);
}

void CheckParm(FILE* File_Ptr, InputStruct * In_Ptr)
{
    short i_run;
    short num_runs; /* number of independent runs. */
    NameLink head = NULL;
    Boolean name_taken; /* output files share the same */
    char msg[STRLEN];
    num_runs = ReadNumRuns(File_Ptr);
    for(i_run=1; i_run<=num_runs; i_run++)
    {
        printf("Checking input data for run %hd\n", i_run);
        FindDataLine(File_Ptr); /* skip the data base parameters.*/
        FindDataLine(File_Ptr); /* */
        ReadParm(File_Ptr, In_Ptr);
        name_taken = FnameTaken(In_Ptr->outfname, &head);
        if(name_taken)
            sprintf(msg, "file name %s duplicated.\n", In_Ptr->outfname);
        else {
            AddNameToList(fname, List_Ptr);
            return(0);
        }
    } // while
}
Allocate the arrays in OutStruct for one run, and
array elements are automatically initialized to zeros.

```c
void InitOutputData(InputStruct In_Parm,
                     OutStruct * Out_Ptr)
{
    short nz = In_Parm.nz;
    short nr = In_Parm.nr;
    short na = In_Parm.na;
    short nl = In_Parm.num_layers;
    /* remember to use nl+2 because of 2 for ambient. */
    if (nz <= 0 || nr <= 0 || na <= 0 || nl <= 0)
        nrerror("Wrong grid parameters.");
    /* Init pure numbers. */
    Out_Ptr->Rsp = 0.0;
    Out_Ptr->Rd = 0.0;
    Out_Ptr->A = 0.0;
    Out_Ptr->Tt = 0.0;
    /* Allocate the arrays and the matrices. */
    Out_Ptr->Rd_ra = AllocMatrix(O,nr-1,0,na-1);
    Out_Ptr->Rd_r = AllocVector(O,nr-1);
    Out_Ptr->Rd_a = AllocVector(O,na-1);
    Out_Ptr->A_rz = AllocMatrix(O,nr-1,0,nz-1);
    Out_Ptr->A_z = AllocVector(O,nz-1);
    Out_Ptr->A_l = AllocVector(O,nl+1);
    Out_Ptr->Tt_ra = AllocMatrix(O,nr-1,0,na-1);
    Out_Ptr->Tt_r = AllocVector(O,nr-1);
    Out_Ptr->Tt_a = AllocVector(O,nl+1);
}
```

Undo what InitOutputData did.

```c
void FreeData(InputStruct In_Parm,
              OutStruct * Out_Ptr)
{
    short num_thetacol=In_Parm.nbrthsc-2;
    short num_rows = (In_Parm.nbrthsc-2)*In_Parm.nbrpssc + 1;
    num_rows = (In_Parm.nbrthsc-2)*In_Parm.nbrpssc + 1;
    /* number of data values for each incident angle according */
    /* to the remarks in MCDATA.FOR. */
    free(In_Parm.layerspecs);
    FreeMatrix(Out_Ptr->Rd_ra, 0,nr-1,0,na-1);
    FreeVector(Out_Ptr->Rd_r, 0,nr-1);
    FreeVector(Out_Ptr->Rd_a, 0,na-1);
    FreeMatrix(Out_Ptr->A_rz, 0, nr-1, 0,nz-1);
    FreeVector(Out_Ptr->A_z, 0, nz-1);
    FreeVector(Out_Ptr->A_l, 0,nl+1);
    FreeMatrix(Out_Ptr->Tt_ra, 0,nr-1,0,na-1);
    FreeVector(Out_Ptr->Tt_r, 0,nr-1);
    FreeVector(Out_Ptr->Tt_a, 0,na-1);
    FreeMatrix(In_Parm.valmatrix,1,num_rows,1,num_thetacol);
    FreeMatrix(In_Parm.valmatrix,1,num_rows,1,num_thetacol);
}
```

Get 1D array elements by summing the 2D array elements.

```c
void Sum2D2Rd(InputStruct In_Parm,
              OutStruct * Out_Ptr)
{
    short nr = In_Parm.nr;
    short na = In_Parm.na;
    short ir,ia;
    double sum;
    for(ir=0; ir<nr; ir++)
    {
        for(ia=0; ia<na; ia++)
        {
            double sum = 0.0;
            Out_Ptr->Rd_ra[ir][ia] = Out_Ptr->Rd_ra[ir][ia];
            Out_Ptr->Rd_r[ir] = Out_Ptr->Rd_r[ir];
            Out_Ptr->Rd_a[ia] = Out_Ptr->Rd_a[ia];
        }
    }
}
```

121
Return the index to the layer according to the index to the grid line system in z direction (Iz).

Use the center of box.

short IzToLayer(short Iz, InputStruct In_Parm) {
  short i=1; /* index to layer. */
  short num_layers = In_Parm.num_layers;
  double dz = In_Parm.dz;
  while((Iz+0.5)*dz >= In_Parm.layerspecs[i].zl && i < num_layers) i++;
  return(i);
}

Get 1D array elements by summing the 2D array elements.

void sum2DA(InputStruct In_Parm, OutStruct * Out_Ptr) {
  short nz = In_Parm.nz;
  short nr = In_Parm.nr;
  short iz,ir;
  double sum;
  for(iz=0; iz<nz; iz++) {
    sum = 0.0;
    for(ir=0; ir<nr; ir++) sum += Out_Ptr->A_rz[ir][iz];
    Out_Ptr->A_z[iz] = sum;
  }
}

Get 1D array elements by summing the 2D array elements.

void sum2DAtt(InputStruct In_Parm, OutStruct * Out_Ptr) {
  short nr = In_Parm.nr;
  short na = In_Parm.na;
  double dr = In_Parm.dr;
  double da = In_Parm.da;
  short ir,ia;
  double sum;
  for(ir=0; ir<nr; ir++) {
    sum = 0.0;
    for(ia=0; ia<na; ia++) sum += Out_Ptr->Tt_ra[ir][ia];
    Out_Ptr->Tt_r[ir] = sum;
  }
}

Scale Rd and Tt properly.

scale1 = 4.0*PI*PI*dr*sin(da/2)*In_Parm.num_photons;
/* The factor [ir+0.5]*sin[2a] to be added. */
for(ir=0; ir<nr; ir++)
for(ia=0; ia<na; ia++)
{
scale2 = 1.0/((ir+0.5)*sin(2.0*(ia+0.5)*da)*scale1);
Out_Ptr->Rd_ra[ir][ia] *= scale2;
Out_Ptr->Tt_ra[ir][ia] *= scale2;
}

scale1 = 2.0*PI*dr*dr*In_Parm.num_photons;
/* area is 2*PI*(ir+0.5)^2*dr. */
/* ir+0.5 to be added */
for(ir=0; ir<nr; ir++)
{
scale2 = 1.0/((ir+0.5)*scale1);
Out_Ptr->Rd_r[ir] *= scale2;
Out_Ptr->Tt_r[ir] *= scale2;
}

scale1 = 2.0*PI*da*In_Parm.num_photons;
/* solid angle is 2*PI*sin(a)*da. sin(a) to be added. */
for(ia=0; ia<na; ia++)
{
scale2 = 1.0/(sin((ia+0.5)*da)*scale1);
Out_Ptr->Rd_a[ia] *= scale2;
Out_Ptr->Tt_a[ia] *= scale2;
}

scale2 = 1.0/(double)In_Parm.num_photons;
Out_Ptr->Rd *= scale2;
Out_Ptr->Tt *= scale2;
/***********************************************************
Scale absorption arrays properly.
/***********************************************************
void ScaleA(InputStruct In_Parm, OutStruct * Out_Ptr) {
short nz = In_Parm.nz;
short nr = In_Parm.nr;
double dz = In_Parm.dz;
double dr = In_Parm.dr;
short n1 = In_Parm.num_layers;
short iz,ir;
short il;
double scale1;
/* Scale A_rz. */
scale1 = 2.0*PI*dr*dz*In_Parm.num_photons;
/* volume is 2*PI*(ir+0.5)^2*dr*dz. */
/* ir+0.5 to be added. */
for(iz=0; iz<nz; iz++)
for(ir=0; ir<nr; ir++)
Out_Ptr->A_rz[ir][iz] /= (ir+0.5)*scale1;

/* Scale A_z. */
scale1 = 1.0/(dz*In_Parm.num_photons);
for(iz=0; iz<nz; iz++)
Out_Ptr->A_z[iz] /= scale1;

/* Scale A_l. Avoid int/int. */
scale1 = 1.0/(double)In_Parm.num_photons;
for(il=0; il<nl+1; il++)
Out_Ptr->A_l[il] /= scale1;
Out_Ptr->A *= scale1;
/***********************************************************
Sum and scale results of current run.
/***********************************************************
void SumScaleResult(InputStruct In_Parm, OutStruct * Out_Ptr) {
/* Get 1D & 0D results. */
Sum2DRd(In_Parm, Out_Ptr);
Sum2DA(In_Parm, Out_Ptr);
Sum2DTt(In_Parm, Out_Ptr);
ScaleRdTt(In_Parm, Out_Ptr);
ScaleA(In_Parm, Out_Ptr);
/***********************************************************
Write the version number as the first string in the file.
Use chars only so that they can be read as either ASCII or binary.
/***********************************************************
void WriteVersion(FILE *file, char *Version) {
fprintf(file, "%s
	# Version number of the file format.
"
Version);
fprintf(file, "####
# Data categories include:
"
InParm, RAT, 
#
A_l, A_z, Rd_r, Rd_a, Tt_r, Tt_a, 
####

"};
Write the input parameters to the file.

void WriteInParm(FILE *file, InputStruct In_Parm)
{
    short i;
    fprintf(file, "InParm "
        "# Input parameters. cm is used.\n"");
    fprintf(file, "$s "
        "output file name, ASCII.\n"", In_Parm.out_fname);
    fprintf(file, "$s "
        "data base file name\n"", In_Parm.dat_fname);
    fprintf(file, "$hd "
        "data base param.\n"", In_Parm.nbrangint, In_Parm.nbrthsc, In_Parm.nbrpssc);
    fprintf(file, "$d 
        "No. of photons\n", In_Parm.num_photons);
    fprintf(file, "$G "
        "dz, dr [cm]\n", In_Parm.dz, In_Parm.dr);
    fprintf(file, "$G "
        "x_size and y_size.\n", In_Parm.x_size, In_Parm.y_size);
    fprintf(file, "$hd "
        "# Number of layers\n", In_Parm.num_layers);
    fprintf(file, "$G "
        "n for medium above\n", In_Parm.layerspecs[$0].n);
    fprintf(file, "$G "
        "n for medium below\n", In_Parm.layerspecs[$i].n);
    fprintf(file, "$G "
        "n for surrounding glass\n
", In_Parm.layerspecs[$i+1].n);
}

Write reflectance, absorption, transmission.

void WriteRAT(FILE *file, OutStruct Out_Parm)
{
    fprintf(file, "RAT "
        "# Reflectance, absorption, transmission. \n"");
    fprintf(file, "$-14.6G "
        "Specular reflectance [-]\n", Out_Parm.Rsp);
    fprintf(file, "$-14.6G "
        "Diffuse reflectance [-]\n", Out_Parm.Rd);
    fprintf(file, "$-14.6G "
        "Absorbed fraction [-]\n", Out_Parm.AI);
    fprintf(file, "$-14.6G "
        "Transmittance [-]\n", Out_Parm.Tt);
}

Write absorption as a function of layer.

void WriteA_layer(FILE *file, short Num_Layers, OutStruct Out_Parm)
{
    short i;
    fprintf(file, "A_l 
        "# Absorption as a function of layer. [-]\n", Out_Parm.A_l);
    for(i=1; i<=Num_Layers; i++)
    {
        fprintf(file, 
            "$12.4G \n", Out_Parm.A_l[i]);
    }
}

Write reflectance, absorption, transmission.

void WriteRd_ra(FILE *file, short Nr, Na, OutStruct Out_Parm)
{
    fprintf(file, 
        "$s $s $s $s $s $s $s $s $s \n", /* flag. */
        "$s $s $s $s $s $s $s $s $s $s \n", /* flag. */
        "$s $s $s $s $s $s $s $s $s $s \n", /* flag. */
        "$s $s $s $s $s $s $s $s $s $s \n", /* flag. */
        "$s $s $s $s $s $s $s $s $s $s \n", /* flag. */
        "$s $s $s $s $s $s $s $s $s $s \n", /* flag. */
        "$s $s $s $s $s $s $s $s $s $s \n", /* flag. */
        "$s $s $s $s $s $s $s $s $s $s \n", /* flag. */
        "$s $s $s $s $s $s $s $s $s $s \n", /* flag. */
        "$s $s $s $s $s $s $s $s $s $s \n", /* flag. */
        "$s $s $s $s $s $s $s $s $s $s \n", /* flag. */
    }
}

void WriteInParm(FILE *file, InputStruct In_Parm)
{
    short i;
    fprintf(file, "InParm "
        "# Input parameters. cm is used.\n"");
    fprintf(file, "$s "
        "output file name, ASCII.\n"", In_Parm.out_fname);
    fprintf(file, "$s "
        "data base file name\n"", In_Parm.dat_fname);
    fprintf(file, "$hd "
        "data base param.\n"", In_Parm.nbrangint, In_Parm.nbrthsc, In_Parm.nbrpssc);
    fprintf(file, "$d 
        "No. of photons\n", In_Parm.num_photons);
    fprintf(file, "$G "
        "dz, dr [cm]\n", In_Parm.dz, In_Parm.dr);
    fprintf(file, "$G "
        "x_size and y_size.\n", In_Parm.x_size, In_Parm.y_size);
    fprintf(file, "$hd "
        "# Number of layers\n", In_Parm.num_layers);
    fprintf(file, "$G "
        "n for medium above\n", In_Parm.layerspecs[$0].n);
    fprintf(file, "$G "
        "n for medium below\n", In_Parm.layerspecs[$i].n);
    fprintf(file, "$G "
        "n for surrounding glass\n
", In_Parm.layerspecs[$i+1].n);
}

Write reflectance, absorption, transmission.

void WriteRAT(FILE *file, OutStruct Out_Parm)
{
    fprintf(file, "RAT "
        "# Reflectance, absorption, transmission. \n"");
    fprintf(file, "$-14.6G "
        "Specular reflectance [-]\n", Out_Parm.Rsp);
    fprintf(file, "$-14.6G "
        "Diffuse reflectance [-]\n", Out_Parm.Rd);
    fprintf(file, "$-14.6G "
        "Absorbed fraction [-]\n", Out_Parm.AI);
    fprintf(file, "$-14.6G "
        "Transmittance [-]\n", Out_Parm.Tt);
}

Write absorption as a function of layer.

void WriteA_layer(FILE *file, short Num_Layers, OutStruct Out_Parm)
{
    short i;
    fprintf(file, "A_l 
        "# Absorption as a function of layer. [-]\n", Out_Parm.A_l);
    for(i=1; i<=Num_Layers; i++)
    {
        fprintf(file, 
            "$12.4G \n", Out_Parm.A_l[i]);
    }
}

Write reflectance, absorption, transmission.

void WriteRd_ra(FILE *file, short Nr, Na, OutStruct Out_Parm)
{
    fprintf(file, 
        "$s $s $s $s $s $s $s $s $s $s \n", /* flag. */
        "$s $s $s $s $s $s $s $s $s $s \n", /* flag. */
        "$s $s $s $s $s $s $s $s $s $s \n", /* flag. */
        "$s $s $s $s $s $s $s $s $s $s \n", /* flag. */
        "$s $s $s $s $s $s $s $s $s $s \n", /* flag. */
        "$s $s $s $s $s $s $s $s $s $s \n", /* flag. */
        "$s $s $s $s $s $s $s $s $s $s \n", /* flag. */
        "$s $s $s $s $s $s $s $s $s $s \n", /* flag. */
        "$s $s $s $s $s $s $s $s $s $s \n", /* flag. */
        "$s $s $s $s $s $s $s $s $s $s \n", /* flag. */
        "$s $s $s $s $s $s $s $s $s $s \n", /* flag. */
    }
}
for(ir=0;ir<Nr;ir++)
    for(ia=0;ia<Na;ia++)
    {  
        fprintf(file, "%12.4E  ", Out_Parm.Rd_ra[ir][ia]);
        if( (ir*Na + ia + 1)%5 == 0) fprintf(file, "\n");
    }
    fprintf(file, "\n");

void WriteRd_a(FILE * file, short Na, OutStruct Out_Parm)
{
    short ia;
    fprintf(file, "# Rd[a] [0], [1], .. Rd[na-1], \n");
    for(ia=0;ia<Na;ia++)
    {  
        fprintf(file, "%12.4E ", Out_Parm.Rd_a[ia]);
        if( (i*Na + ia + 1)%5 == 0) fprintf(file, "\n");
    }
    fprintf(file, "\n");
}

void WriteTt_ra(FILE * file, short Nr, short Na, OutStruct Out_Parm)
{
    short ir, ia;
    fprintf(file, "# Tt[r] \n");
    for(ir=0;ir<Nr;ir++)
    for(ia=0;ia<Na;ia++)
    {  
        fprintf(file, "%12.4E ", Out_Parm.Tt_ra[ir][ia]);
        if( (ir*Na + ia + 1)%5 == 0) fprintf(file, "\n");
    }
    fprintf(file, "\n");
}

void WriteA_rz(FILE * file, short Nr, short Nz, OutStruct Out_Parm)
{
    short ir, iz;
    fprintf(file, "# A[r][z] \n");
    for(iz=0;iz<Nz;iz++)
    for(ir=0;ir<Nr;ir++)
    {  
        fprintf(file, "%12.4E ", Out_Parm.A[rz][ir][iz]);
        if( (i*Na + iz + 1)%5 == 0) fprintf(file, "\n");
    }
    fprintf(file, "\n");
}
for (ir = 0; ir < Nr; ir++)
for (iz = 0; iz < Nz; iz++)
{
    fprintf(file, "%12.4E", Out_Parm.A_rz[ir][iz]);
    if ( (ir*Nz + iz + 1)%5 == 0) fprintf(file, "\n");
}

void WriteA_z(FILE * file,
              short Nz,
              OutStruct Out_Parm)
{
    short iz;
    fprintf(file,
            "A_z #A[0], [1],..A[nz-1], [1/cm]\n"); /* flag. */
    for(iz=0;iz<Nz;iz++)
    {
        fprintf(file, "%12.4E\n", Out_Parm.A_z[iz]);
    }
    fprintf(file, "\n");
}

void WriteTt_r(FILE * file,
               short Nr,
               OutStruct Out_Parm)
{
    short ir;
    fprintf(file,
            "#Tt[0], [1],..Tt[nr-1], [1/cm^2]\n"); /* flag. */
    for(ir=0;ir<Nr;ir++)
    {
        fprintf(file, "%12.4E\n", Out_Parm.Tt_r[ir]);
    }
    fprintf(file, "\n");
}

void WriteResult(InputStruct In_Farm,
                  FILE *file, 
                  OutStruct Out_Parm,
                  char * TimeReport)
{
    FILE *file;
    file = fopen(In_Farm.out_fname, "w");
    if (file == NULL) nrerror("Cannot open file to write.
");
    if(toupper(In_Farm.out_fformat) == 'A')
        WriteVersion(file, "AL");
    else
        WriteVersion(file, "BL");
    fprintf (file, "#");
    fprintf (file, "%s", TimeReport);
    fprintf (file, 
    WriteinParm(file, In_Parm);
    WriteRAT(file, Out_Parm);
    WriteA_layer(file, In_Farm.num_layers, Out_Parm);
    WriteA_z(file, In_Farm.nz, Out_Parm);
    WriteRd_r(file, In_Parm.nr, Out_Parm);
    WriteRd_a(file, In_Parm.na, Out_Parm);
    WriteTt_r(file, In_Parm.nr, Out_Parm);
    WriteTt_a(file, In_Parm.na, Out_Parm);
}

void WriteTt_a(FILE * file,
               short Na,
               OutStruct Out_Parm)
{
    short ia;
    fprintf(file, 
            "#Tt[0], [1],..Tt[na-1], [sr-1]\n"); /* flag. */
    for(ia=0;ia<Na;ia++)
    {
        fprintf(file, "%12.4E\n", Out_Parm.Tt_a[ia]);
    }
    fprintf(file, "\n");
}
WriteRD_ra(file, In_Parm.nr, In_Parm.na, Out_Parm);
WriteTT_ra(file, In_Parm.nr, In_Parm.na, Out_Parm);

B.1.2.4. mcmlgo.c

/***********************************************************
* Launch, move, and record photon weight.               
***/
#include "mcml.h"
#define STANDARDTEST 0
  /* testing program using fixed rnd seed. */
#define PARTIALREFLECTION 0
  /* 1=split photon, 0=statistical reflection. */
#define COSZERO (1.0-1.0E-12)
  /* cosine of about 1e-6 rad. */
#define COS90D 1.0E-6
  /* cosine of about 1.57 - 1e-6 rad. */

/***********************************************************
* A random number generator from Numerical Recipes in C.  
***/
#define MBIG 1000000000
#define MSEED 161803398
#define MZ 0
#define FAC 1.0E-9

float ran3(int *idum)
{
  static int inext,inextp;
  static long ma[56];
  static int iff=0;
  long mj,mk;
  int i,ii,k;
  if (*idum < 0 || iff == 0) {
    iff=1;
    mj=MSEED-*idum ; *idum ;
    mj %= MBIG;
    ma[55]=mj;
    mk=1;
    for (i=1;1<=S4;1++) {
      ii=(21*i) % 55;
      mk=mj-mk;
      if (mk < MZ) mk += MBIG;
      mj=ma[11];
      for (i=1;i<=55;i++) {
        ma[i] = ma[i+(i+30) % 55];
        if (ma[i] < MZ) ma[i] += MBIG;
      }
      inext=0;
      inextp=1;
      *idum=1;
    }
    if (++inext == 56) inext=1;
    if (++inextp == 56) inextp=1;
    mj=ma[inext]-ma[inextp];
    if (mj < MZ) mj += MBIG;
    ma[inext]=mj;
    return mj*FAC;
  }
}

/***********************************************************
* Generate a random number between 0 and 1. Take a     
* number as seed the first time entering the function.  
* The seed is limited to 1<<15.                         
* We found that when idum is too large, ran3 may return 
* numbers beyond 0 and 1.                                
***/
double RandomNum(void)
{
  static Boolean first_time=1;
  static int idum;
  /* seed for ran3. */
  if(first_time)
#if STANDARDTEST /* Use fixed seed to test the program. */
idum = -1;
#else
    idum = -(int)time(NULL)%1<<15;
    /* use 16-bit integer as the seed. */
#endif
ran3(&idum);
first_time = 0;
idum = 1;

return( (double) ran3(&idum) );

/*---------------------------------------------------
* Compute the specular reflection.
* If the first layer is a turbid medium, use the Fresnel
* reflection from the boundary of the first layer as the
* specular reflectance.
* If the first layer is glass, multiple reflections in
* the first layer is considered to get the specular
* reflectance.
* The subroutine assumes the Layerspecs array is correctly
* initialized.
*--------------------------------------------------
*/
double Rspecular(Layerstruct * Layerspecs_Ptr,
        Photonstruct * Photon_Ptr)
{
    double rl, r2;
    /* direct reflections from the 1st and 2nd layers. */
    double temp;

    temp = (Layerspecs_Ptr[0].n - Layerspecs_Ptr[1].n) / (Layerspecs_Ptr[0].n + Layerspecs_Ptr[1].n);
    r1 = temp*temp;

    if(Layerspecs_Ptr[1].meanstep == 0.0) /* glass layer. */
        temp = (Layerspecs_Ptr[1].n - Layerspecs_Ptr[2].n) / (Layerspecs_Ptr[1].n + Layerspecs_Ptr[2].n);
        r2 = temp*temp;
        rl = r1 + (1-r1)*r2/(1-r1*r2);
    return (rl);

/*---------------------------------------------------
* Initialize a photon packet.
*---------------------------------------------------
*/

****/

void LaunchPhoton(double Rspecular, LayerStruct * Layerspecs_Ptr,
        PhotonStruct * Photon_Ptr)
{
    Photon_Ptr->w = 1.0 - Rspecular;
    Photon_Ptr->dead = 0;
    Photon_Ptr->layer = 1;
    Photon_Ptr->x = 0;
    Photon_Ptr->ys = 0;
    Photon_Ptr->z = 0;
    Photon_Ptr->x = 0.0;
    Photon_Ptr->y = 0.0;
    Photon_Ptr->z = 0.0;
    Photon_Ptr->x = 0.0;
    Photon_Ptr->y = 0.0;
    Photon_Ptr->z = 1.0;

    if(Layerspecs_Ptr[1].meanstep == 0.0) /* glass layer. */
        Photon_Ptr->layer = 2;
        Photon_Ptr->x = Layerspecs_Ptr[2].x;

/*---------------------------------------------------
* Calculate a value from a normal (Gaussian) distribution,
* according to Numerical Recipes in C.
* Meanvalue = 0, standard deviation = dev.
*---------------------------------------------------
*/

double GaussDevStep(double dev)
{
    double fac, r1, v1, v2, gaussdev;
    static double StepGset;
    static short StepIsset = 0;

    if(StepIsset == 0) {
        do {
            v1 = 2*RandomNum() - 1.0;
            v2 = 2*RandomNum() - 1.0;
            r1 = v1*v1+v2*v2;
        } while (r1>=1.0 || r1==0.0);
    fac = sqrt(-2.0*dev*dev*log(r1)/r);
    StepGset = v1*fac;
    gaussdev = v2*fac;
    StepIsset = 1;
    }
    else {
        StepIsset = 0;
        gaussdev = StepGset;
    }
}
return(gaussdev);
}

******************************************************************************
* Calculate a value from a normal (Gaussian) distribution, according to Numerical Recipes in C. Meanvalue = 0, standard deviation = dev. *****/

double GaussDevSpin(double dev)
{
    double fac, r, v1, v2, gaussdev;
    static double SpinGset;
    static short Spiniset = 0;
    if(Spiniset == 0)
    do {
        v1 = 2*RandomNum() - 1.0;
        v2 = 2*RandomNum() - 1.0;
        r = v1*v1+v2*v2;
    }
    while (r>=1.0 || r==0.0);
    fac = sqrt(-2.0*dev*dev*log(r)/r);
    SpinGset = v1*fac;
    gaussdev = v2*fac;
    Spiniset = 1;
    else {
        Spiniset = 0;
        gaussdev = SpinGset;
    }
    return(gaussdev);
}

******************************************************************************
* Choose a new direction for photon propagation by sampling the polar deflection angle theta and the azimuthal angle psi. Note: theta: 0 - pi so sin(theta) is always positive feel free to use sqrt() for cos(theta).
* psi: 0 - 2pi for 0-pi sin(psi) is + for pi-2pi sin(psi) is - *****/

void SpinInputStruct • In_Ptr, PhotonStruct * Photon_Ptr)
{
    short layer = Photon_Ptr->layer;
    double cost, sint;
    /* cosine and sine of the polar deflection angle theta. */
    double cosp, sinp;
    /* cosine and sine of the azimuthal angle psi. */
    double ux = Photon_Ptr->ux; /* photon direction. */
    double uy = Photon_Ptr->uy; /* photon direction. */
    double uz = Photon_Ptr->uz; /* photon direction. */
    double bcux; /* spheroid direction. */
    double bcuy; /* spheroid direction. */
    double bcuz; /* spheroid direction. */
    double angin; /* the incident angle. */
    double theta; /* scattering angle. */
    double psi; /* scattering angle. */
    double dev = In_Ptr->layerspecs[layer].dev;
    short thetacol;
    short thetacolshort;
    double thetacoldouble;
    double dev = In_Ptr->layerspecs[layer].dev;
    short num_rows;
    short num_thetacol = In_Ptr->nbrangint;
    short nbrthsc = In_Ptr->nbrthsc;
    short nbcpsc = In_Ptr->nbcpsc;
    short ipsi, itheta;
    short i;
    double temp;
    num_rows = (nbrthsc-2)*nbcpsc + 1; /* number of data values for each incident angle according */
    /* to the remarks in MCDATA.FOR. */
    if (num_thetacol == 1) /* sphere. */
        thetacol = 1; /* independent of incident angle. */
    else {
        if (dev == 1.0) /* spheroid. */
            do {
                bcux = cos(PI*RandomNum());
                bcuy = cos(PI*RandomNum());
                bcuz = cos(PI*RandomNum());
                temp = sqrt(bcux*bcux+bcuy*bcuy+bcuz*bcuz);
            }
            while (temp == 0.0);
        else {
            bcux = In_Ptr->layerspecs[layer].bcux;
            bcuy = In_Ptr->layerspecs[layer].bcuy;
            bcuz = In_Ptr->layerspecs[layer].bcuz;
            temp = 1.0;
            if (dev == 0.0) {
do {
    bcux = bcux + GaussDevSpin(dev);
    bcuy = bcuy + GaussDevSpin(dev);
    bcuz = bcuz + GaussDevSpin(dev);
    temp = sqrt(bcux*bcux+bcuy*bcuy+bcuz*bcuz);
    while (temp == 0.0);
}

angin = acos((ux*bcux+uy*bcuy+uz*bcuz)/temp);
/* the incident angle angin is the angle between the photon direction */
/* and the symmetry axis of the particle. */
if (angin>PI/2)
    angin = PI-angin;

thetacolshort=(angin/(PI/2))*(num_thetacol-1)+1; /* integer value. */
if (thetacoldouble - thetacolshort < 0.5)
    thetacol = thetacolshort;
else
    thetacol = thetacolshort + 1;
/* The discrete scattering directions are evaluated and */
/* spread out continuously. */

psi = 2*PI - psi; /* symmetry property */
cost = cos(theta);
sint = sqrt(1.0 - cost*cost);
/* sqrt() is faster than sin(). */
cosp = cos(psi);
if(psi<PI)
    sinp = sqrt(1.0 - cosp*cosp);
else
    sinp = - sqrt(1.0 - cosp*cosp);
if(fabs(uz) > COSZERO) { /* normal incident. */
    Photon_Ptr->ux = sint*cosp;
    Photon_Ptr->uy = sint*sinp;
    Photon_Ptr->uz = cost*SIGN(uz);
    /* SIGN() is faster than division. */
} else { /* regular incident. */
    double temp = sqrt(1.0 - uz*uz);
    Photon_Ptr->ux = sint*(ux*uz*cosp - uy*sinp)/temp + ux*cost;
    Photon_Ptr->uy = sint*(uy*uz*cosp + ux*sinp)/temp + uy*cost;
    Photon_Ptr->uz = -sint*cost*temp + uz*cost;
}

***********************************************************************
* Move the photon s away in the current layer of medium. *******/
void Hop(PhotonStruct * Photon_Ptr) {
    double s = Photon_Ptr->s;
    Photon_Ptr->x += s*Photon_Ptr->ux;
    Photon_Ptr->y += s*Photon_Ptr->uy;
    Photon_Ptr->z += s*Photon_Ptr->uz;
}

/***********************************************************************/
short StepSizeInGlass(PhotonStruct * Photon_Ptr, InputStruct * In_Ptr) { double dl_b; /* distance to nearest boundary. */ double dl_xbound; /* distance to boundary perp. to x-direction. */ double dl_ybound; /* distance to boundary perp. to y-direction. */ double dl_zbound; /* distance to boundary perp. to z-direction. */ short layer = Photon_Ptr->layer; double ux = Photon_Ptr->UX; double uy = Photon_Ptr->uy; double uz = Photon_Ptr->uz; short hitcase; /* the boundary which is hit. */ /* hitcase = 1, 2 or 3 for boundary perp. to the */ /* x-, y- or z-directions. */ /* distance to the nearest boundary. */ /* dl_zbound>O. */ if(uz>0.0) { dl_zbound = (In_Ptr->layerspecs[layer].z1 - Photon_Ptr->z)/uz; /* dl_zbound>O. */ dl_b = dl_zbound; hitcase = 3; } else if(ux<0.0) { dl_xbound = ((In_Ptr->x_sizei*SIGN(ux)/2 - Photon_Ptr->xl)/ux; if (dl_xbound<dl_b) { dl_b = dl_xbound; hitcase = 1; } } else if(uy>0.0) { dl_ybound = ((In_Ptr->y_sizei*SIGN(uy)/2 - Photon_Ptr->yl)/uy; if (dl_ybound<dl_b) { dl_b = dl_ybound; hitcase = 2; } } Photon_Ptr->s = dl_b; return(hitcase); }

void StepSizeInSurrGlass(PhotonStruct * Photon_Ptr, InputStruct * In_Ptr) { double dl_b; /* distance to boundary. */ short layer = Photon_Ptr->layer; double uz = Photon_Ptr->uz; if(uz>0.0) { dl_b = (In_Ptr->layerspecs[layer].z1 - Photon_Ptr->z)/uz; /* dl_b>0. */ } else if(uz<0.0) { dl_b = (In_Ptr->layerspecs[layer].z0 - Photon_Ptr->z)/uz; /* dl_b>0. */ } Photon_Ptr->s = dl_b; }

void StepSizeInTissue(PhotonStruct * Photon_Ptr, InputStruct * In_Ptr) { short layer = Photon_Ptr->layer; double meanstep = In_Ptr->layerspecs[layer].meanstep; double scale = In_Ptr->layerspecs[layer].meanstep; double meanstepdev = In_Ptr->layerspecs[layer].meanstepdev; double stepdev; /* standard deviation of meanstep. */ do { stepdev = GaussDevStep(meanstepdev); while (stepdev<meanstep/2); /* meanstep+stepdev must not be too small. */ meanstep = meanstep + stepdev; } while (Photon_Ptr->sleft == 0.0) { /* make a new step. */ Photon_Ptr->s = meanstep; } else { /* take the leftover. */ Photon_Ptr->s = Photon_Ptr->sleft/scale; }}
 Photon_Ptr->sleft = 0.0;
}

/******************************************************************************
 * Check if the step will hit a boundary.
 * Return 1, 2 or 3 if a boundary is hit.
 * Return 0 otherwise.
 * If the projected step hits the boundary, the members
 * s and sleft of Photon_Ptr are updated.
 * scale is used instead of mua+mut in the original program.
 */
short HitBoundary(PhotonStruct * Photon_Ptr,
                  InputStruct * In_Ptr)
{
    double dl_b;  /* distance to nearest boundary. */
    double dl_xbound;  /* distance to boundary perp. to x-direction. */
    double dl_ybound;  /* distance to boundary perp. to y-direction. */
    double dl_zbound;  /* distance to boundary perp. to z-direction. */
    short layer = Photon_Ptr->layer;
    double ux = Photon_Ptr->UX;
    double uy = Photon_Ptr->uy;
    double uz = Photon_Ptr->uz;
    short hitcase;  /* the boundary which is hit. */
    /* hitcase = 1, 2 or 3 for boundary perp. to the x-, y- or z-directions. */
    /* hitcase = 0 if no boundary is hit. */
    double maxdl_xybound = sqrt(pow(In_Ptr->x_size,2.0)
                                + pow(In_Ptr->y_size,2.0)
                                + pow(In_Ptr->layerspecs[layer].zl - In_Ptr->layerspecs[layer].z0,2.0));
    /* The longest possible path in this layer. */
    if(UZ>0.0)
    {
        dl_zbound = (In_Ptr->layerspecs[layer].z0 - Photon_Ptr->z)/uz;
    /* dl zbound>0. */
    }
    else if(uz<0.0)
    {
        dl_zbound = (In_Ptr->layerspecs[layer].z0 - Photon_Ptr->z)/uz;
    /* dl zbound>0. */
    }
    if(uz == 0.0)
    {
        dl_zbound = maxdl_xybound;
    else
        {
            dl_b = dl_zbound;
            hitcase = 3;
        }
    }
    if(ux != 0.0) 
    {
        dl_xbound = ((In_Ptr->x_size)*SIGN(ux)/2 - Photon_Ptr->x)/ux;
        if (dl_xbound>dl_b) 
        {
            dl_b = dl_xbound;
            hitcase = 1;
        }
        else
        {
            hitcase = 0;
        }
        return(hitcase);
    }
    if(Photon_Ptr->s > dl_b)
    { /* crossing. */
        double scale = In_Ptr->layerspecs[layer].meanstep;
        Photon_Ptr->sleft = (Photon_Ptr->s - dl_b)*scale;
        Photon_Ptr->s = dl_b;
    }
    else
    {
        hitcase = 0;
        return(hitcase);
    }
}

/******************************************************************************
 * Drop photon weight inside the tissue (not glass).
 * The photon is assumed not dead.
 * The weight drop is dwa = abscross*w.
 * The dropped weight is assigned to the absorption array elements.
 */
void Drop(InputStruct * In_Ptr,
          PhotonStruct * Photon_Ptr,
          outStruct * out_Ptr)
{
    double dwa;  /* absorbed weight. */
    double x = Photon_Ptr->x;
    double y = Photon_Ptr->y;
    short iz, ir;  /* index to z & r. */
    short layer = Photon_Ptr->layer;
    double abscross;
    /* compute array indices. */
    iz = (short)((Photon_Ptr->z/In_Ptr->dzl);
    if(iz>In_Ptr->nz-1) iz=In_Ptr->nz-1;
    ir = (short)(sqrt(x*x+y*y)/In_Ptr->drl);
    if(ir>In_Ptr->nr-1) ir=In_Ptr->nr-1;
    abscross = In_Ptr->layerspecs[layer].abscross;

    if(iz<0)
    {
        iz = 0;
    }
    if(iz>=(In_Ptr->nz-1))
    {
        iz = In_Ptr->nz-1;
    }
    if(ir<0)
    {
        ir = 0;
    }
    if(ir>=(In_Ptr->nr-1))
    {
        ir = In_Ptr->nr-1;
    }
    if((iz<0) || (iz>(In_Ptr->nz-1)) || (ir<0) || (ir>(In_Ptr->nr-1)))
    {
        abscross = 0.0;
    }
    double weight = Photon_Ptr->w * dwa;
    Photon_Ptr->w = Photon_Ptr->w - weight;
    Photon_Ptr->sleft = Photon_Ptr->sleft + weight;
    out_Ptr->absorption[iz][ir] = abscross;
    out_Ptr->weight[iz][ir] = weight;
}

}
/* update photon weight. */
abscross = In_Ptr->layerspecs[layer].abscross;
dwa = Photon_Ptr->w * abscross;
Photon_Ptr->w -= dwa;
/* assign dwa to the absorption array element. */
Out_Ptr->A_rz[ir][iz] += dwa;

/***********************************************************
* The photon weight is small, and the photon packet tries *
* to survive a roulette. ****
void Roulette(PhotonStruct * Photon_Ptr)
{
if(Photon_Ptr->w == 0.0)
Photon_Ptr->dead = 1;
else if(RandomNum() < CHANCE) /* survived the roulette.*/
Photon_Ptr->w /= CHANCE;
else
Photon_Ptr->dead = 1;
}

/***********************************************************
* Compute the Fresnel reflectance. *
* Make sure that the cosine of the incident angle al *
* is positive, and the case when the angle is greater *
* than the critical angle is ruled out. *
* Avoid trigonometric function operations as much as *
* possible, because they are computation-intensive. ****
* Record the photon weight exiting the first *
* layer(uz<O), no matter whether the layer is glass or not, to the *
* reflection array. Update the photon weight as well. ****
double RFresnel(double n1, /* incident refractive index.*/
    double n2, /* transmit refractive index.*/
    double ca1, /* cosine of the incident */
    /* angle. 0<ca1<90 degrees. */
    double * ca2_Ptr) /* pointer to the */
    /* cosine of the transmission */
    /* angle. a2>0. */
{
    double r;
    if(n1==n2) { /* matched boundary. */
    "ca2_Ptr = ca1;
    r = (n2-n1)/(n2*n1);
    } else if(ca1<COS90D) { /* very slant. */
    "ca2_Ptr = 0.0;
    r = 1.0;
    } else (*ca2_Ptr = \n    double ca1, sa1, sa2;
    /* sine of the incident and transmission angles. */
    double ca2;
    sa1 = sqrt(1-ca1*ca1);
    sa2 = n1*sa1/n2;
    if(sa2>1.0) {
    "ca2_Ptr = 0.0;
    r = 1.0;
    } else (*ca2_Ptr = \n    double sal, sa2; /* transmit refractive index. */
    double * ca2_Ptr = ca2 = sqrt(1-sa2*sa2);
    cap = ca1*ca2 - sa1*sa2; /* c+ = cc + ss. */
    cam = ca1*ca2 + sa1*sa2; /* c- = cc - ss. */
    sap = sal*ca2 + cap*sa1; /* s+ = sc + cs. */
    sam = sal*ca2 - cap*sa1; /* s- = sc - cs. */
    r = 0.5*(sam*sam*sam*cam*cam+cap*cap)/(sap*sap*cam*cam);
    /* rearranged for speed. */
}
    return(r);
}

/***********************************************************
* Record the photon weight exiting the first layer(uz<0), *
* no matter whether the layer is glass or not, to the *
* reflection array. *
* Update the photon weight as well. ****
void RecordR(double InputStruct * In_Ptr,
    PhotonStruct * Photon_Ptr,
OutStruct * Out_Ptr)
{
    double x = Photon_Ptr->x;
    double y = Photon_Ptr->y;
    short ir, ia; /* index to r & angle. */
    ir = (short)(sqrt(x*x+y*y)/In_Ptr->dr);
    if(ir>In_Ptr->nr-1) ir=In_Ptr->nr-1;
    ia = (short)(acos(-Photon_Ptr->uz)/In_Ptr->da);
    if(ia>In_Ptr->na-1) ia=In_Ptr->na-1;
    /* assign photon to the reflection array element. */
    Out_Ptr->Rd_ra[ir][ia] += Photon_Ptr->w*(1.0-Refl);
    Photon_Ptr->w *= Refl;
}

void RecordT(double Refl, InputStruct * In_Ptr,
              PhotonStruct * Photon_Ptr,
              OutStruct * Out_Ptr)
{
    double x = Photon_Ptr->x;
    double y = Photon_Ptr->y;
    short ir, ia; /* index to r & angle. */
    ir = (short)(sqrt(x*x+y*y)/In_Ptr->dr);
    if(ir>In_Ptr->nr-1) ir=In_Ptr->nr-1;
    ia = (short)(acos(Photon_Ptr->uz)/In_Ptr->da);
    if(ia>In_Ptr->na-1) ia=In_Ptr->na-1;
    /* assign photon to the transmittance array element. */
    Out_Ptr->Tt_ra[ir][ia] += Photon_Ptr->w*(1.0-Refll);
    Photon_Ptr->w *= Refl;
}

void CrossUpOrNot(InputStruct * In_Ptr, PhotonStruct * Photon_Ptr,
                   OutStruct * Out_Ptr)
{
    double uz = Photon_Ptr->uz; /* z directional cosine. */
    double uz1; /* cosines of transmission alpha, always positive. */
    double r=O.O; /* reflectance. */
    short layer = Photon_Ptr->layer;
    double ni = In_Ptr->layerspecs[layer].n;
    double nt = In_Ptr->layerspecs[layer-1].n;
    /* Get r. */
    if( - uz <= In_Ptr->layerspecs[layer].cos_critO)
        r=1.0; /* total internal reflection. */
    else if (layer == In_Ptr->num_layers + 2) /* surr. glass. */
        r = RFresnel(In_Ptr->layerspecs[layer].n,
                     In_Ptr->layerspecs[0].n, -uz, &uz1);
    else r = RFresnel(In_Ptr->layerspecs[layer].n,
                      In_Ptr->layerspecs[layer-1].n, -uz, &uz1);
    if (r == 1.0 && layer == In_Ptr->num_layers + 2)
        Photon_Ptr->dead = 1; /* total internal reflection in surrounding medium. */
    else if (RandomNum() > r) {/* transmitted. */
        Photon_Ptr->uz = -uz1; /* transmitted photon. */
        RecordR(r, In_Ptr, Photon_Ptr, Out_Ptr);
        Photon_Ptr->uz = -uz; /* reflected photon. */
    } else if(RandomNum()) > r; /* transmitted. */
}
```c
Photon_Ptr->ux *= ni/nt;
Photon_Ptr->uy *= ni/nt;
Photon_Ptr->uz = -uz;
}
else /* reflected. */
Photon_Ptr->uz = -uz;
#endif

if(RandomNum() > r) {/* transmitted. */
} 
#endif

if(RandomNum() > r) {/* transmitted. */
}

/* Decide whether the photon will be transmitted or be */
/* reflected on the boundary perpendicular to the x-direction. */
/* If the photon is transmitted, move the photon to */
/* "layer+1". If "layer" is the last or surrounding layer, */
/* record the transmitted weight as transmittance. */
/* See comments for CrossUpOrNot. */
/* * Update the photon parameters. * */
void CrossDnOrNot(InputStruct * In_Ptr,
PhotonStruct * Photon_Ptr,
OutStruct * Out_Ptr)
{
    double uz = Photon_Ptr->uz; /* z directional cosine. */
    double uz1; /* cosines of transmission alpha. */
    double r=0.0; /* reflectance. */
    short layer = Photon_Ptr->layer;
    double ni = In_Ptr->layerspecs[layer].n;
    double nt = In_Ptr->layerspecs[layer+1].n;
    /* Get r. */
    if(uz <= In_Ptr->layerspecs[layer].cos_crit1)
        r=1.0; /* total internal reflection. */
    else if(layer == In_Ptr->num_layers + 2) {/* surr. glass. */
        r = RFresnel(In_Ptr->layerspecs[layer].n,
                    In_Ptr->layerspecs[layer-1].n, uz, &uz1);
    else r = RFresnel(In_Ptr->layerspecs[layer].n,
                    In_Ptr->layerspecs[layer+1].n, uz, &uz1);
#endif

/* Decide whether the photon will be transmitted or be */
/* reflected on the boundary perpendicular */
/* to the x-direction. */
/* If the photon is transmitted, move the photon to the */
/* surrounded glass layer. */
/* * Update the photon parameters. */
```
void CrossXOrNot(InputStruct * In_Ptr, PhotonStruct * Photon_Ptr, OutStruct * Out_Ptr) {
    double ux = Photon_Ptr->ux; /* x directional cosine. */
    double uxl; /* cosines of transmission alpha. */
    double r=0.0; /* reflectance. */
    short layer = Photon_Ptr->layer;
    double ni = In_Ptr->layerspecs[layer].n;
    double nt = In_Ptr->layerspecs[In_Ptr->num_layers+2].n;
    /* Get r. */
    if(fabs(ux) <= In_Ptr->layerspecs[layer].cos_critxy) r=1.0; /* total internal reflection. */
    else r = RFresnel(In_Ptr->layerspecs[layer].n, In_Ptr->layerspecs[In_Ptr->num_layers+2].n, fabs(ux), &uxl);  
    if(RandomNum() > r) { /* transmitted to surrounding glass. */
        Photon_Ptr->layer = In_Ptr->num_layers+2;
        Photon_Ptr->ux *= ni/nt;
        Photon_Ptr->uy *= ni/nt;
    } else Photon_Ptr->ux = -ux; /* reflected. */
}

void CrossYOrNot(InputStruct * In_Ptr, PhotonStruct * Photon_Ptr, OutStruct * Out_Ptr) {
    double uy = Photon_Ptr->uy; /* x directional cosine. */
    double uyl; /* cosines of transmission alpha. */
    double r=0.0; /* reflectance. */
    short layer = Photon_Ptr->layer;
    double ni = In_Ptr->layerspecs[layer].n;
    double nt = In_Ptr->layerspecs[In_Ptr->num_layers+2].n;
    /* Get r. */
    if(fabs(uy) <= In_Ptr->layerspecs[layer].cos_critxy) r=1.0; /* total internal reflection. */
    else r = RFresnel(In_Ptr->layerspecs[layer].n, In_Ptr->layerspecs[In_Ptr->num_layers+2].n, fabs(uy), &uyl);
    if(RandomNum() > r) { /* transmitted to surrounding glass. */
        Photon_Ptr->layer = In_Ptr->num_layers+2;
        Photon_Ptr->ux *= ni/nt;
        Photon_Ptr->uy *= ni/nt;
    } else Photon_Ptr->uy = -uy; /* reflected. */
}

void HopInGlass(InputStruct * In_Ptr, PhotonStruct * Photon_Ptr, OutStruct * Out_Ptr) {
    short hitcase; /* the boundary which is hit (1,2 or 3). */
    if(Photon_Ptr->ux == 0.0) /* horizontal photon in glass is killed. */
        Photon_Ptr->dead = 1;
if(Photon_Ptr->layer == In_Ptr->num_layers + 2) {
    /* surrounding glass. */
    StepSizeInSurGlass(Photon_Ptr, In_Ptr);
    hitcase = 3; /* always in surrounding glass. */
    CrossOrNot(In_Ptr, Photon_Ptr, Out_Ptr, hitcase);
} else { /* glass layer which is not the surrounding glass layer. */
    hitcase = StepSizeInGlass(Photon_Ptr, In_Ptr);
    Hop(Photon_Ptr);
    CrossOrNot(In_Ptr, Photon_Ptr, Out_Ptr, hitcase);
}

/********************************************
* Set a step size, move the photon, drop some weight, *
* choose a new photon direction for propagation. *
* When a step size is long enough for the photon to *
* hit an interface, this step is divided into two steps. *
* First, move the photon to the boundary free of *
* absorption or scattering, then decide whether the *
* photon is reflected or transmitted. *
* Then move the photon in the current or transmission *
* medium with the unfinished stepsize to interaction *
* site. If the unfinished stepsize is still too long, *
* repeat the above process. *
********************************************
void HopDropSpinInTissue(InputStruct * In_Ptr,
          PhotonStruct * Photon_Ptr,
          OutStruct * Out_Ptr)
{
    short hitcase; /* the boundary which is hit (1,2,3). */
    /* hitcase = 0 if no boundary is hit. */
    StepSizeInTissue(Photon_Ptr, In_Ptr);
    hitcase = HitBoundary(Photon_Ptr, In_Ptr);
    if(hitcase>0) { /* move to boundary plane. */
        Hop(Photon_Ptr); /* move to boundary plane. */
        CrossOrNot(In_Ptr, Photon_Ptr, Out_Ptr, hitcase);
    } else {
        Hop(Photon_Ptr);
        Drop(In_Ptr, Photon_Ptr, Out_Ptr);
        Spin(In_Ptr, Photon_Ptr);
    }
}

/********************************************
****
void HopDropSpin(InputStruct * In_Ptr,
          PhotonStruct * Photon_Ptr,
          OutStruct * Out_Ptr)
{
    short layer = Photon_Ptr->layer;
    if(In_Ptr->layerspecs[layer].meanstep == 0.0) /* glass layer. */
        HopInGlass(In_Ptr, Photon_Ptr, Out_Ptr);
    else
        HopDropSpinInTissue(In_Ptr, Photon_Ptr, Out_Ptr);
    if( Photon_Ptr->w < In_Ptr->Wth && !Photon_Ptr->dead)
        Roulette(Photon_Ptr);
B.2. List of specific modifications

B.2.1. Modifications from MCML to MCMLBOUN

The input file:

- The lateral extension of the plane layers of the sample, x_size and y_size, are defined on a row below the row defining the variables nz, nr and na.
- The refractive index of the surrounding glass is defined on a row next to the rows containing the parameters of the different layers and ambients.

mcml.h:

- A variable cos_critxy is declared (data type: double) in the structure of Layerstruct.
- The two variables x_size and y_size are declared (data type: double) in the structure of Inputstruct.

mcmlio.c:

- The function ShowVersion shows a modified version.
- A new function ReadXSizeYSize, reading the variables x_size and y_size, is written after the function ReadNzNrNa.
- An additional function ReadSurrGlass, reading and setting the parameters of the surrounding glass, is included after the function ReadOneLayer.
- The number of elements in the array named Layerspecs is increased by one due to the surrounding glass. The argument of the malloc commando in the function ReadLayerSpecs is thus changed.
- The function ReadLayerSpecs calls the function ReadSurrGlass. The surrounding glass is thus defined as the layer with index (num_layers + 2).
- The calculation of the critical angles for total internal reflection for the different layers relative the surrounding glass (cos_critxy) are added in the function CriticalAngle as well as the critical angles for the surrounding glass relative the ambient.
- The function ReadXSizeYSize is called after the function ReadNzNrNa from the function ReadParm.
- A modification of the function WriteInParm, arranges the new input parameters in the output file.

mcmlgo.c:

- The data type of the function StepSizeInGlass is changed to short. The function returns the values (1, 2, 3) due to the hit boundary (perpendicular to x-, y- or z-axis). The implementation of the path length to the boundary is not modified.
- The function HitBoundary is modified in the same way, but when no boundary was hit, the value returned is zero.
- In order to handle photon packages propagating in the surrounding glass, a function called StepSizeInSurrGlass has been written.
- The two functions CrossUpOrNot and CrossDnOrNot are modified to work in the surrounding glass too. Photon packages which are totally reflected in the surrounding glass are killed in CrossUpOrNot.
- Two new functions, similar to the two previous mentioned, check if the photon packages are reflected or transmitted when hitting the boundary to the surrounding...
glass. This functions are called CrossXOrNot and CrossYOrNot and are introduced directly after CrossDnOrNot.

- A new argument, hitcase (data type: short), is declared in the function CrossOrNot. If hitcase equals 1 or 2, CrossXOrNot or CrossYOrNot is called respectively. If hitcase equals 3, the function continues as in the original version.
- The function HopInGlass calls StepSizeInGlass or StepSizeInSurrGlass depending on the position of the photon package. When the function CrossOrNot is called, the value of the variable hitcase obtained from the function StepSizeInGlass, is included in the argument.
- The new variable hitcase is used in a similar way in the function HopDropSpinInTissue, but its value is obtained from the function HitBoundary. The function CrossOrNot is called when hitcase differs from zero.

mcmlmain.c:

- The function ShowVersion is declared in the main program in order to avoid compiler warnings.

B.2.2. Modifications from MCMLBOUN to TMMCML

The input file:

- The name of the file with the data base, and the values of the variables (datatype: short) nbrangint, nbrthsc and nbrpssc are written before the normal input parameters of each run. The three variables represent the number of incident angles ($\theta_i$), scattered deflection angles ($\theta_s$) and scattered azimuthal angles ($\phi_s$), respectively, uniformly distributed. These values must agree with those connected to the data base, to obtain the right result.
- The mean distance between the scattering objects (meanstep), the standard deviation of this value (meanstepdev), the normalized absorption cross section (abscross), the direction of the symmetry axis of the scattering objects (bcux, bcuy, bcuz) and the standard deviation of this direction (dev) are written on a new row for each layer. The length unit is cm.

mcml.h:

- The parameters belonging to the data base (dat_fname, nbrangint, nbrthsc, nbrpssc and valmatrix) are declared in the structure of lnputStruct. The array dat_fname (data type: char) contains the name of the data base file. The two-dimensional array valmatrix (data type: double) is used to store the values from the data base.
- The parameters of each layer (datatype: double) are declared in the structure of LayerStruct, replacing of the old ones (mua, mus and g).

mcmlio.c:

- The function ShowVersion shows a modified version.
- The new parameters are read instead of the old ones in the function ReadOneLayer and are written to the output file by the modified function WriteInParm.
- The function ReadSurrGlass assigns meanstep = 0 instead of $\mu_a = \mu_s = 0$.
- Two new functions, ReadDataBase and PlaceDataInMatrix, allocate memory space for a two-dimensional array (matrix), read the data base values and put them in the array.
• In the function CheckParm a commando FindDataLine is added to prevent the data base to be read twice. This saves time but the data base parameters will not be checked before the program starts.
• To free the allocated memory used to the data base values, another FreeMatrix commando is added in the function FreeData.

mcmlgo.c:

• The functions Rspecular, LaunchPhoton and HopDropSpin check if meanstep = 0 instead of $\mu_a = \mu_s = 0$.
• By using the variables meanstep and meanstepdev and the new function StepGaussDev, the function StepSizeInTissue evaluates a step length of the photon package instead of the old expression using $\mu_a$ and $\mu_s$. In this function and in the function HitBoundary a variable scale is defined, corresponding to the variable $\mu_t$ in the original program when calculating the value of the variable sleft.
• The function Drop decreases the weight of the photon package by a factor abs$\times$cross instead of the quotient $\mu_d/\mu_t$.
• The function Spin is completely changed. The function SpinTheta is removed and the new function SpinGaussDev handles the Gaussian distributed directions of the scattering objects.
• The variable g in the argument of Spin is replaced by the structure variable In_Ptr. This modification is also made in the call from the function HopDropSpinInTissue.

mcmlmain.c:

• The function ReadDataBase is declared and called.

B.3. Flow charts

Flow charts of the Monte Carlo programs are presented on the following three pages. The chart of the original Monte Carlo program is first shown, followed by the two modified versions MCMLBOUN and TMMCML. Squares symbolizing new routines in the modified versions are filled and squares for modified routines are shaded.
1. ReadNumRuns

2. ReadParm

3. Read Parn format

4. Read File

5. Check Parm

6. Run n

7. Doone Run

8. Free Data

1. Rspecu
2. Skapa och O:a matriser
3. Report II
4. Launch Status photon
5. Predict Time
6. Report Status photon
7. Hop Drop Spin 1.2.3.
8. Report Result

**Numberoflayers = Numberoflayers + 1**

**if SurrGlass**

**else**

- **StepSize inSurGlass**
- **StepSize inGlass**
- **StepSize inTissue**

**Cross Not**

**Hit Boundary**

- **Yes**
- **No**

**Spin (μx,μy,μz)**

**Spin Theta**

- **Hitcase = 3, μz < 0**
- **Hitcase = 1, 2**

**Cross DropoutNot**

**CrossDropNot**

- **Yes**
- **No**

**Hitcase = 3**

**Drop (w, A_ry)**

**RecordR (Rd_ra)**

- **If layer = layer1**
- **CrossX OrNot**

**CrossY OrNot**

- **If layer = lastlayer**
- **RecordT (Tt_ra)**
APPENDIX C: How to run the programs

A description of running the programs will be given in order to prevent mistakes and wrong solutions. The characteristics of the data base MC.DAT will be discussed in detail in order to make it easier for other users to do further modifications. An extensive description of the modifications of the Monte Carlo programs is made in appendix B.2. This appendix will present only examples of input files of these programs.

C.1. EXTENDT1

The input parameters are the same as in the original T1 program (Chapter 3.1.1.).
1. Choose values for nrank and ntheta. The experience from this work is to choose nrank slightly less than twice the size parameter of an equal volume sphere and ntheta = 2*nrank.
2. Check convergence over nrank (case 1), ntheta (case 0).
3. If convergence was not obtained, restart from 1.
4. Check convergence over m (case 2), save the T-matrix (alternative 1).
5. If convergence was obtained, the T-matrix is stored in file t.

C.2. EXTENDT3

The EXTENDT3 program was in this work only used for controls and to plot three-dimensional pictures. The input and output parameters are exactly the same as in program T3, described in chapter 3.1.3.

C.3. MCDATA

1. Write the input parameters. The variables nbrangin, nbrthsc and nbphsc are previously described (section 3.1.4). The variable anglab represents the polarization direction and has hardly any influence on the angular scattering intensities, when the difference between the refractive indices of the particle and the surrounding medium is much less than unity.
2. The differential scattering cross sections are computed in all directions. The scattering angles \( \theta \) and \( \phi \) vary between \( 0^\circ \) and \( 180^\circ \) with an interval of \( d\theta = 180^\circ/(nbrthsc-1) \) and \( d\phi = 180^\circ/(nbphsc-1) \), respectively. When \( \theta \) equals \( 0^\circ \) or \( 180^\circ \), the cross section only need to be evaluated for one azimuthal angle \( (\phi = 0^\circ) \). As these variables are discrete, one direction is assumed to represent all the directions closed by. Each direction is assigned an area corresponding to the surface area in that direction interval of a surrounding sphere. Therefore, the differential scattering cross sections are normalized by a factor:

\[
\frac{d\phi \left( \cos(\theta - \frac{d\theta}{2}) - \cos(\theta + \frac{d\theta}{2}) \right)}{2\pi} = \frac{1 - \cos(\frac{d\theta}{2})}{\text{area of this direction}} = \frac{\text{area of forward direction}}{	ext{area of forward direction}}
\]

All values representing directions when \( \phi \) differ from \( 0^\circ \) and \( 180^\circ \) are multiplied by a factor of two, as they represent both \( \phi \) and \( 360^\circ - \phi \). This implementation, according to symmetry properties is made to save time and memory allocation.
3. The modified differential cross sections are then multiplied by a new normalization factor, in order to obtain a probability function, where the sum of all variables equals one.
4. Step 2. and 3. are repeated for every new incident angle.

144
5. The probability data are stored in MC.DAT using a specific algorithm:
   for angin = 1 to nbrangin \( \theta_1 = 0 \) to 90°
   \[ \text{sum} = \text{probability of back scattering} \]
   write sum
   for theta = 2 to nbrthsc-1 \( \theta_8 = \theta_1 + 180° \) to 180°/(nbrthsc-1))
   \[ \text{for phi} = 1 \text{ to nbrphsc } \phi \text{ = 0 to 180°} \]
   \[ \text{sum} = \text{sum} + \text{probability of scattering in this direction} \]
   write sum
   end
   end
   the probability of forward scattering \( \theta_8 = 0 \) is thus 1 - sum
   end

C.4. MCMLBOUN

Input file of MCMLBOUN:

```plaintext
####
# This is the input file (bountest.mci) for testing the mcm1boun
# using fixed rnd seed. 6/25/96.
####
1.0 # file version
1 # number of runs
### Specify data for run 1
tmtest1.mco A # output filename, ASCII/Binary
1000 # No. of photons
1 0.1 # dz, dr
7 75 30 # No. of dz, dr & da.
1000 1000 # x_size and y_size
3 # No. of layers
# n  mua  mus  g  d
# One line for each layer
1.0 # n for medium above.
1.5 0 0 0 1 # glass
1.345 1 249 0.99 5 # layer 1
1.5 0 0 0 1 # glass
1.0 # n for medium below.
1.5 # n for surrounding glass
```

C.5. TMMCML

The most important thing to remember, running this program, is to write the same values for the number of different angles (nbrangin, nbrthsc, nbrphsc) in the input file as those used when obtaining the data base MC.DAT. Assign meanstep = 0 when a glass layer or another non-absorbing and non-scattering layer is required.
Input file of TMMCML:

```plaintext
###
# This is the input file (tmtest.mci) for testing the mcml
# using fixed rnd seed. 6/25/96.
###

1.0                         # file version
1                          # number of runs

### Specify data for run 1
mc.dat               1    # data base file name, nbrangin
181    37              # nbrthsc, nbrphsc
tmtestl.mco     A      # output filename, ASCII/Binary
100
1     1
8    75     30         # No. of dz, dr & da.
5000    5000          # x_size and y_size

3                          # No. of layers
# n  mstep  dev  abs  bcux  bcuy  bcuz  dev  d  # One line for each layer
# n for medium above.
1.0
1.5    0    0    0    0    0    0    0    0    1    # glass
1.345  0.1  0.02  0.004  1    1    0    0.2  10  # layer 1
1.5    0    0    0    0    0    0    0    0    1    # glass
1.0
1.5
```

# Sample explanation:
- The file version is `1.0` and the number of runs is `1`.
- The data base file name is `mc.dat` with `1` number of runs.
- The output filename is `tmtestl.mco` with file type `A`.
- The number of photons is `100`.
- The number of layers is `3`.
- The first layer consists of a medium with `n = 1.0`, thickness `8`, `dz`, `dr` of `30`, and `x_size`, `y_size` of `5000`.
- The second layer consists of a medium with `n = 1.5`, thickness `181`, `dz`, `dr` of `37`, and `x_size`, `y_size` of `181`.
- The third layer consists of a medium with `n = 1.345`, thickness `1`, `dz`, `dr` of `0.02`, and `x_size`, `y_size` of `100`.
- The fourth layer consists of a medium with `n = 1.5`, thickness `1`, `dz`, `dr` of `0.004`, and `x_size`, `y_size` of `5000`.
- The last layer consists of a medium with `n = 1.5`, thickness `1`, `dz`, `dr` of `0`, and `x_size`, `y_size` of `5000`.

---

146
APPENDIX D: Special functions

When physical problems related to wave propagation and scattering theory at spherical symmetry are to be solved mathematically, some complicated functions seem to turn up very often. Some of the functions, which have got their own names, are used in this work and a short mathematical description will be given. More detailed information and theory are given by Arfken.

D.1. Bessel and Hankel functions

The Bessel function \( j_n(z) \) is the solution of the equation

\[
\frac{d^2}{dz^2} f_n(z) + 2z \frac{d}{dz} f_n(z) + (z^2 - n(n+1)) f_n(z) = 0
\]  

(D.1)

which is finite in origo. The Hankel functions of the first and second kind are described by

\[
h_n^{(1)}(z) = j_n(z) + i n_n(z)
\]

(D.2a)

\[
h_n^{(2)}(z) = j_n(z) - i n_n(z)
\]

(D.2b)

where the Neumann function \( n_n(z) \) is the solution of the differential equation (D.1) which is infinite in origo. The Bessel function can be expressed as

\[
j_n(z) = 2^n z^n \sum_{k=0}^{\infty} \frac{(-1)^k (k + n)!}{k! (2k + 2n + 1)!} z^k
\]  

(D.3)

and has the property

\[
j_n(-z) = (-1)^n j_n(z)
\]  

(D.4)

The expression of the Neumann function is similar and the two functions can be represented in easier asymptotic expressions when the arguments are small.

For \( n = 0 \) the functions are described by

\[
j_0(z) = \sin \frac{z}{z} ; \quad n_0(z) = -\cos \frac{z}{z}
\]

and thus \( h_0^{(1)} = -\frac{i}{z} e^{i\phi} \)  

(D.5 a,b,c)

D.2. Associated Legendre functions

The associated Legendre functions \( P_n^m(\cos \theta) \) are parts of the spherical harmonic functions \( Y_n^m(\theta,\phi) \) according to the relation

\[
Y_n^m(\theta,\phi) = c_{nm} P_n^m(\cos \theta) e^{im\phi}
\]  

(D.6)

where \( P_n^m(\cos \theta) \) is the solution of the differential equation
\[ \frac{1}{\sin \theta} \frac{d}{d\theta} (\sin \theta \frac{d}{d\theta} (P_n^m(\cos \theta))) + (n(n+1) - \frac{m^2}{\sin^2 \theta})P_n^m(\cos \theta) = 0 \]  
(D.7)

which results in the following relations:

\[ P_n^m(\cos \theta) = \sin^m \theta \frac{d^m}{d(\cos \theta)^m} P_n^0(\cos \theta) \quad \theta \in [0,\pi] \]  
(D.8)

\[ P_n^m(\cos \theta) = 0 \quad \text{when } |m| > 1 \]  
(D.9)

\[ P_n^m(-\cos \theta) = (-1)^m P_n^m(\cos \theta) \]  
(D.10)

\[ P_n^m(\theta = 0) = \delta_{m0} \]  
(D.11)

Two examples of associated Legendre functions of lower orders are expressed as

\[ P_1^1(\cos \theta) = \sqrt{1 - \cos^2 \theta} = \sin \theta \]  
(D.12a)

\[ P_2^1(\cos \theta) = 105 \cos \theta (1 - \cos^2 \theta)^{3/2} = 105 \cos \theta \sin^3 \theta \]  
(D.12b)