Till Stefan!
Tack för all hjälp och uppmuntran!
Johannes

Mathematical Modeling of Light Distribution in Turbid Media in Terms of Photon Hitting Densities

Master's Thesis
by
Johannes Swartling

Lund Reports on Atomic Physics, LRAP-211
Lund, January 1997

ABSTRACT
The idea of using near infra-red light for transillumination of biological tissue for the purpose of medical imaging is attractive, because light of this wavelength is easily obtained from inexpensive diode lasers, and it has no known negative influence on the human organism at moderate intensities. The method would require an advanced image reconstruction algorithm, because of the very high scattering of light in tissue. An important step toward such an algorithm is the ability of calculating the probability distribution of possible photon paths traversing from a light source to a detector through tissue. This probability distribution was calculated, using the concept of photon hitting density, for simple geometries by utilizing two models: Monte Carlo simulation and diffusion theory. In the first case, an existing computer program was modified to perform the calculations. In the second, a new program was designed. The results were then compared to extract the validity as well as advantages and disadvantages of each model. Finally, possible use of the results as part of an image reconstruction algorithm for optical tomography was investigated.

Preface

This Master's thesis presents work performed at the Medical Group at the Department of Atomic Physics, Lund Institute of Technology, Sweden, during the summer and fall of 1996. The work represents the final step toward my Master of Science degree in Engineering Physics.

I would like to thank my supervisor Claes af Klinteberg for guidance through my work, and Stefan Andersson-Engels for an inspiring leadership. I also want to thank all the other staff at the division, especially Prof. Sune Svanberg, whose lectures strongly encouraged me to find my way to the atomic physics department.

Lund, January 1997

Johannes Swartling

Contents

1. Introduction

Developing methods for imaging internal structures in living tissue using near infra-red (NIR) light is a field of research that has drawn much attention during the last decade. The main benefit of such a technique would be the possibility to have an alternative technique to x-ray imaging without having to use the potentially mutagenic x-rays. This should be especially important in cases where routine examination of large groups of the population is desired, such as mammography screening for breast cancer. When using traditional x-ray mammography for this purpose, the examination itself will statistically induce growth of malignant tumors in a few cases. Another drawback of the method is that it is not sensitive to all kinds of tumors. Between 10 and 30 per cent of all breast cancer cases are initially diagnosed as negative on mammograms[1]. Interest has also been focused on the possibility of using NIR imaging for examining the brain and other internal organs, as a less costly and perhaps more movable complement to the well-established CT and MRI techniques.

Living tissue has a relatively low absorption in the NIR wavelength region (~700-1000 nm). For shorter wavelengths the hemoglobin starts to absorb heavily. Early attempts of trans-illuminating breasts utilized continuous-wave light in an approach similar to the x-ray imaging, i.e. to obtain a shadow image. Tissue, however, scatters NIR light to the extent that the images were so blurred that they carried little or no information of the internal structure. Materials characterized by this kind of low absorption and high scattering are often referred to as *turbid media*. Later developments proposed a time resolved technique, in which short laser pulses (~1 ps) are injected in the tissue. By using time resolved detection, it is possible to extract the portion of light that arrives first on the other side[2,3,4]. This light has traveled the shortest and straightest paths through the tissue. Any variation in this signal thus contains spatial information about the tissue optical properties, and should thus be suitable for imaging. This works fairly well as long as the sample is kept thin (~1 cm for tissue), but for thicker samples the amount of non-scattered light is so small that the input laser pulse has to be of an intolerably high intensity to produce a detectable signal.

Lately, many attempts have focused on the possibility of combining the time resolved detection technique with an image reconstruction algorithm, a novelty which can be termed *optical tomography*. Not only the early arriving light would then be employed, but light from different time intervals in the detected pulse, thus increasing the signal. The assumption that the photons follow straight paths is then no longer valid. Instead, a theory that gives the distribution of the light between the source and the detector is needed. The image reconstruction algorithm requires a model which makes it possible to compute this distribution. The model should not only describe the light propagation fairly accurately, but must also be computationally fast to be of any practical use. Modeling of light propagation in highly scattering tissue, with the aim of image reconstruction, is the concern of this thesis.

The problem of calculating the distribution of light in tissue can be approached in several ways. Two of the most common methods are *diffusion theory* and *Monte Carlo simulation*. Both methods are based on the more general transport theory, which can be used to describe many types of transport in physical systems, e.g. mass, heat, electromagnetic energy or sound. When diffusion theory is applied on light scattering, it is assumed that two criteria are fulfilled. Firstly, that every time a photon changes its direction, due to scattering, it may take off in any new direction with a uniform probability. This is called *isotropic* scattering. Secondly, that if a detector is placed at a point inside the tissue, it will detect an equal amount of photons regardless of which direction it is pointed in, no matter where the light source is

located. These assumptions are not generally true, but, as will be shown later, they are valid in many situations of interest for medical applications.

Monte Carlo simulation uses an entirely different approach. Where diffusion theory relies on solving a differential equation to calculate the photon density in the scattering medium, the Monte Carlo method explicitly simulates individual photons as they move through the medium. Scattering is a highly stochastic process, and thus a random number generator must be used to accomplish the simulation. A large number of photons are simulated in this manner, and eventually a distribution will form. The restrictions of the diffusion theory do not apply to the Monte Carlo method. It is possible to determine the probability distributions from which the random numbers are drawn, e.g. to favor a certain direction of scattering for the photons.

This thesis reviews the means of calculating the distribution of light in a scattering medium, by using diffusion theory and Monte Carlo simulation, respectively. The concept of the *photon hitting density* is used, which is a measure of the probability that a detected photon has been at a certain position in the medium for a particular geometry. The results from the computer computations are presented and discussed, and the two models are compared and evaluated. Finally, in a somewhat stand-alone section, the problem of image reconstruction is defined and possible approaches to solve it are described. Special attention is given to the *perturbation approach*, which is the method adopted by most researchers in the area today, and in which the photon hitting density plays an important role.

2. Light Propagation in Tissue

2.1 Optical Properties of Tissue

As light interacts with matter many processes may occur. The light can, for example, be absorbed, elastically or inelastically scattered, or reflected. All of these processes are wavelength dependent. In this thesis, absorption, elastic scattering and, when necessary, reflection are considered.

In the NIR wavelength region the dominating process in tissue is elastic scattering of photons. Elastic means that the photons do not change their energy in the process. As a whole, no light energy is lost due to this type of scattering. However, a light beam traversing a scattering medium will be attenuated in the forward direction, because some of the photons will find new directions of propagation. This attenuation can be described by an exponential law (the well-known *Lambert-Beer law*), and a measure of the scattering is given by the macroscopic quantity, the *scattering coefficient* μ_s. The unit of the scattering coefficient is [length^{-1}], often expressed in mm^{-1}. $1/\mu_s$ can then be said to be a measure of the mean distance a photon travels between scattering events. The value of the scattering coefficient in tissue is in the order of 10 mm^{-1} in the NIR region.

The quantity used to describe the absorption is called the *absorption coefficient*, μ_a, and is a measure of the probability of a photon being absorbed by the material. The value of the absorption coefficient in tissue is in the order of 0.1 mm^{-1} for NIR light. The total attenuation coefficient is defined as

$$\mu_t = \mu_a + \mu_s. \qquad [2.1]$$

The third quantity used to describe the optical properties of tissue is the *scattering anisotropy factor g*,

$$g = \int_{4\pi} \cos\theta \cdot p(\cos\theta) d\Omega \qquad [2.2]$$

The g-factor is the mean cosine of the scattering angle θ and can vary between -1 and +1. $g = -1$ means complete back scattering, $g = 0$ means isotropic scattering and $g = 1$ means complete forward scattering. Light subjected to scattering by particles larger than the wavelength, which is the case in tissue, is forward scattered. The anisotropy of light scattering can be approximated by the *Henyey-Greenstein phase function*, originally used to describe scattering of light in space by interstellar dust[5],

$$p(\Omega'\cdot\Omega) = p(\cos\theta) = \frac{1}{4\pi} \frac{1-g^2}{(1+g^2-2g\cos\theta)^{3/2}} \qquad [2.3]$$

where $p(\Omega'\cdot\Omega)$ is the probability that a photon traversing along direction Ω' will continue in the direction Ω after the scattering event, and θ is the deflection angle. Typical values of g in tissue are in the range 0.7-0.95.

Another useful quantity is the *reduced scattering coefficient*, defined as

$$\mu_s' = (1 - g)\,\mu_s. \qquad [2.4]$$

$1/\mu_s'$ is a measure of the diffusive effective mean free path between scattering events, or in other words: the mean distance a photon travels between scattering events so that the scattering can be regarded as isotropic (Fig. 2.1). This quantity will prove useful later, when the *diffusion equation* is introduced.

Finally, the *linear transport coefficient* is defined as

$$\mu_{tr} = \mu_s' + \mu_a. \qquad [2.5]$$

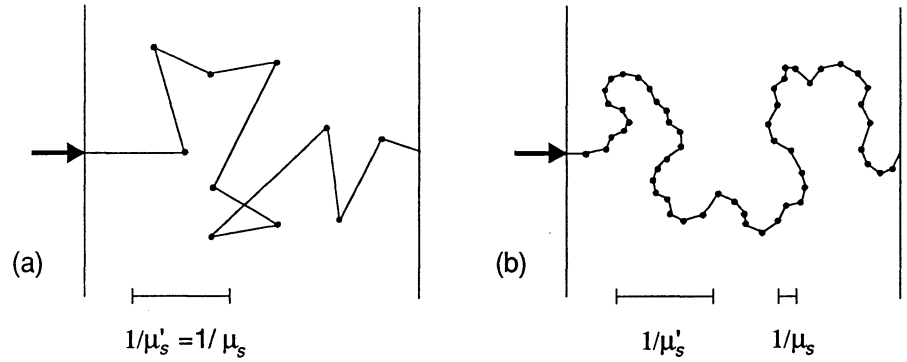


Fig. 2.1 The significance of the reduced scattering coefficient μ_s'. In (a), scattering is isotropic. In (b), photons are forward scattered, but on the scale given by $1/\mu_s'$ scattering may been seen as isotropic in both (a) and (b).

2.2 Measurements

The time resolved measurement technique proposes the utilization of short laser pulses, in the picosecond regime, and a detector capable of resolving the signal also in the order of picoseconds. Present semiconductor-detector technology is just on the border of such performance, so usually a *single photon counting* setup is used for detection. This technique utilizes a fast detector, e.g. a microchannel plate, for detection,[2] which has a temporal resolution of ~50 ps. It may also be possible to use a streak-camera for detection. The detected pulse is often referred to as the *time point-spread function* (TPSF) (Fig. 2.2).

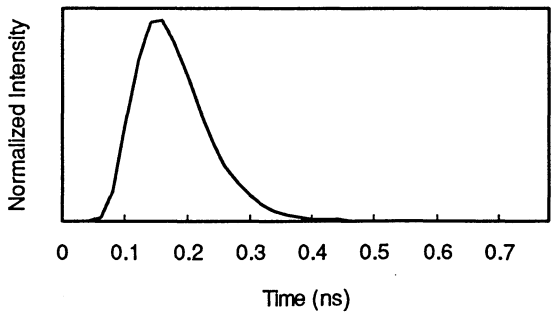


Fig. 2.2 Typical curve shape of the TPSF. Incident light pulse is assumed to have been applied at $t = 0$, and the TPSF corresponds to the detected intensity at a point at some distance from the source.

Another possibility is to carry out the measurements in the frequency domain. The laser source is then sinusoidally modulated, giving rise to *photon density waves* in the medium. The amplitude and phase at the detector are measured for different frequencies.

Mathematically, these two measurement techniques are equivalent, coupled by the Fourier transform. Thus, the TPSF contains information corresponding to measurements for all frequencies in the frequency domain, which then can be obtained in one single measurement. Because of the equivalency, the choice of measurement is determined by which detection setup is easiest to accomplish and provides the best signal-to-noise ratio.

2.3 THE DIFFUSION EQUATION

A very common way of describing light propagation in tissue is to employ transport theory. The transport equation, in its general form, is a basic equation of continuity. It is very complicated to solve other than in some simple cases. By expanding the functions in the transport equation in spherical harmonics and then truncating the expansion, one can derive the diffusion equation[6,7], which is valid under certain circumstances:

$$\frac{1}{v}\frac{\partial}{\partial t}\rho - \nabla D\nabla\rho + \mu_a\rho = \frac{1}{v}q_0 \qquad [2.6]$$

where $\rho = \rho(\mathbf{r},t)$ is the photon density at position $\mathbf{r}$ at time t,
$v = c/n$ is the speed of light in the medium,

$$D = \frac{1}{3\mu_{tr}} = \frac{1}{3(\mu_a + (1-g)\mu_s)}$$ is the *diffusion coefficient*,

q_0 represents a source.

The diffusion equation is valid under the following conditions:

- The reduced scattering coefficient must be much larger than the absorption, i.e. $\mu_s' >> \mu_a$.
- The position where ρ is to be calculated must be far away from the light source, i.e. several effective mean free path lengths.

Although tissue is generally forward scattering, with $g \sim 0.9$, the use of the reduced scattering coefficient μ_s' in the diffusion equation ensures that scattering can be seen as isotropic (see Fig. 2.1), and if the two conditions above are fulfilled the propagation can be regarded as diffusive. In a homogeneous medium D is independent of $\mathbf{r}$ and the second term in Eq. 2.6 can be replaced by $D\Delta\rho$. The validity of the diffusion approximation has been confirmed in many experiments.

By using *Fick's law* the *photon current density* is obtained:

$$\mathbf{J}(\mathbf{r},t) = -vD\nabla\rho(\mathbf{r},t) \qquad [2.7]$$

The measured quantity is the photon density ρ when the detector is located inside the medium, but the photon current density $\mathbf{J}$ when the detector is located outside the medium[8].

The way to treat the boundary conditions has been, and is, a matter of much debate. Simply setting the photon density to zero at the sample boundary seems unphysical, because

at a boundary between two media with different indices of refraction n_1 and n_2, there is a reflection r, given by Fresnel's law[9],

$$r = \frac{1}{2}\left(\frac{\sin^2(\varphi_1 - \varphi_2)}{\sin^2(\varphi_1 + \varphi_2)} + \frac{\tan^2(\varphi_1 - \varphi_2)}{\tan^2(\varphi_1 + \varphi_2)}\right) \qquad [2.8]$$

where the angles φ_1 and φ_2 are given by Snell's law:

$$n_1 \sin \varphi_1 = n_2 \sin \varphi_2. \qquad [2.9]$$

This implies that the photon density on the boundary is non-zero. Different approaches for treating the boundary conditions were developed in the 19th century for heat conduction. Similar methods were later used for modeling neutron diffusion in nuclear reactors. The most common way is to introduce an extrapolated boundary, so that the photon density is zero at some distance outside the physical boundary (Fig. 2.3). The distance to the extrapolated boundary z_e is then a function of the Fresnel reflection,[10]

$$z_e = \frac{2D}{\mathrm{K}} \qquad [2.10]$$

where

$$\mathrm{K} = \frac{(1-R_o)(1-\mu_c^{\,2})}{(1+R_o)(1-R_o)\mu_c^{\,3}}, \qquad [2.11]$$

$$R_o = \left(\frac{n_2 - n_1}{n_2 + n_1}\right)^2, \qquad [2.12]$$

$$\mu_c = \cos\left[\sin^{-1}\left(\frac{n_1}{n_2}\right)\right] \qquad [2.13]$$

For a tissue/air boundary, $n_t = 1.4$, $n_a = 1.0$, the distance is $z_e \approx 2z_o$, where $z_o = 1/\mu_{tr}$. Other means of treating the boundary conditions have also been investigated[10].

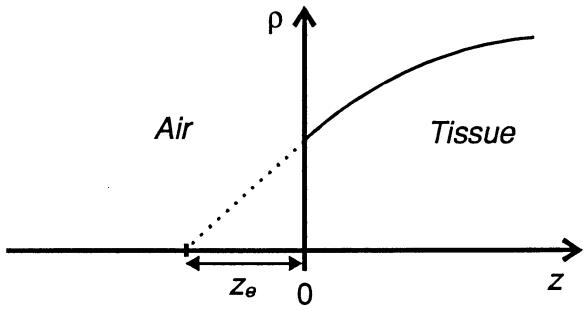


Fig. 2.3 The principle of the 'extrapolated boundary' method for treating the boundary conditions. The photon density ρ is non-zero on the physical boundary because of Fresnel reflection, so in the calculations an extrapolated boundary at a distance of z_e outside the physical boundary is assumed, where the condition of zero photon density can be applied.

The diffusion equation can be solved analytically for some simple geometries, such as a homogeneous semi-infinite half-space or a homogeneous slab, by introducing image sources.

Standard numerical methods for solving partial differential equations can be used for more complicated geometries. These include the finite element method (FEM)[11], and the Crank-Nicolson method, which can be enhanced to obtain the alternating direction implicit (ADI) method[6].

One of the two models for time resolved photon propagation later examined in this thesis is based on the diffusion equation. An advantage of this theory is its potential for designing an algorithm that is computationally relatively fast. A disadvantage might be the many approximations made deriving the equation. Extensive testing therefore has to be made to verify the validity of the results.

2.4 Monte Carlo Simulation

Another widely used technique for describing light propagation in tissue is Monte Carlo simulation. Owing its name from the famous casino, it is a statistical method based on a sophisticated type of random walk. A photon packet, with an initial weight, is launched into the simulated tissue. In each interaction point, the scattering direction, the absorption and the distance to the next interaction point are computed as stochastic variables based on the optical properties of the tissue. The weight is then updated and the photon packet moved to the next interaction point. The physical parameters of the photon packet can be logged during this walk, and the behavior of a large number of photon packets results in a statistical estimation of the macroscopic property of interest.

Monte Carlo simulations have proved to be a very powerful way of modeling light propagation in tissue. The method does not suffer from the inaccuracies of the diffusion equation. It is also possible to simulate any geometry and still obtain valid results. However, the method has one major drawback that makes it less attractive for practical use: it is computationally very time consuming. Since it is a statistical method, the more photon packets added, the lower the variance and the better the result. Often many hours even on relatively fast computers are required to obtain good statistics.

Treating the boundary conditions in the case of Monte Carlo simulation is a simple task: just make use of Fresnel's and Snell's laws explicitly when the photon packet hits the boundary.

The other model investigated in this thesis is based on Monte Carlo simulation.

2.5 Photon Hitting Density

A fundamental problem when designing an image reconstruction algorithm is knowing where the photons have been on their way from the source to the detector. An absolute knowledge of this is of course impossible to obtain, because of the stochastic nature of the propagation process, but what can be calculated is the distribution of photon paths. As a measure of this quantity, Schotland *et al.* have proposed the concept of the photon hitting density[12]. This is a measure of the expected local time that photons spend at different positions when migrating from source to detector within a given time interval.

Consider a point source applied at time $t = 0$ in $\mathbf{r}_1$, and that the light is detected at time t in $\mathbf{r}_2$ (Fig. 2.4). The probability that a photon has been in a point $\mathbf{r}$ somewhere between the source and the detector should depend on two things. Firstly, on the photon density ρ in $\mathbf{r}$ during a time window given by the time the first light reaches $\mathbf{r}$, to the time the last light has to leave $\mathbf{r}$ to have a chance to reach $\mathbf{r}_2$ in time t. Secondly, on the probability E that a photon in $\mathbf{r}$ at a time t' within the time window will reach $\mathbf{r}_2$ at time t. There has been some debate

over how to define E, the *escape function*, as we will see later. Apparently, one has to multiply these two factors and integrate over the time window. This would yield the photon hitting density in $\mathbf{r}$ for the given source-detector configuration for time t. The integration interval will depend on t, and thus the integration is a convolution. These are the basics of the approach when determining the photon hitting density utilizing the diffusion equation, which will be more apparent later.

Monte Carlo simulation lends itself to a simple way of determining the photon hitting density. Since the simulation logs the position of the photon packets, the photon paths that traverse from the source to the detector in time t directly build up the photon hitting density distribution.

The photon hitting density may be interpreted as a probability distribution, where places with high values correspond to high probabilities of finding the photon for a given geometry and optical properties. In chapter 5, the concept of the photon hitting density is discussed within the framework of image reconstruction algorithms, and a somewhat different interpretation is introduced. Its potential usefulness will then hopefully be more apparent. In the next chapter, methods for calculating the photon hitting density using the diffusion approximation and the Monte Carlo model are described.

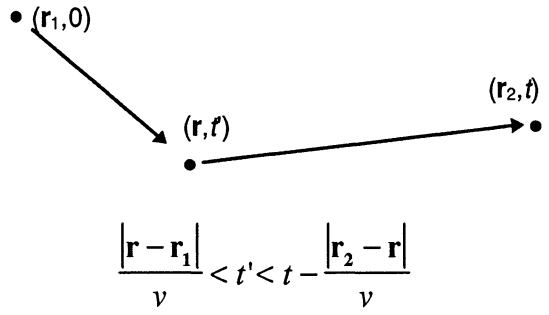


Fig. 2.4 Integration parameter t' starts when the first light reaches $\mathbf{r}$ and ends when the last light has to leave $\mathbf{r}$ in order to still have a chance of reaching $\mathbf{r}_2$ at time t.

3. Calculating the Photon Hitting Density

3.1 The Diffusion Approximation Approach

Let us first consider the simplest case: a single source in an infinite, homogeneous medium, Dirac-distributed in both space and time, given by

$$q_0(\mathbf{r},t) = \delta(\mathbf{r})\delta(t). \qquad [3.1]$$

The diffusion equation is then easily solved by using Green's functions:

$$\rho(\mathbf{r},t) = v(4\pi D)^{-3/2} t^{-3/2} \exp(-\mu_a vt) \exp\left[-\frac{\mathbf{r}^2}{4Dvt}\right] \qquad [3.2]$$

Eq. 3.2 is the *Green's function for free diffusion.*

For other geometries, as long as they are not too complicated, the standard method is to introduce mirrored image sources, so that the boundary conditions are met and the calculation can be performed over infinity (e.g. Fig. 3.1). The solution is then generalized accordingly into

$$\rho(\mathbf{r},t) = v(4\pi D)^{-3/2} t^{-3/2} \exp(-\mu_a vt) \sum_k \left\{ \exp\left[-\frac{\mathbf{r}_{pk}^{\,2}}{4Dvt}\right] - \exp\left[-\frac{\mathbf{r}_{nk}^{\,2}}{4Dvt}\right] \right\} \qquad [3.3]$$

where $\mathbf{r}_{pk}$ and $\mathbf{r}_{nk}$ represent the distances to the components of the k:th order image source dipole. Fortunately, in the cases usually at interest, the terms for high values of k in Eq. 3.3 will never affect the result, so the sum can be truncated at relatively low values of k.

In an even more general form, the photon density may be written as

$$\rho(\mathbf{r},t) = \int G(\mathbf{r}',\mathbf{r};t)\rho(\mathbf{r}',0)d^3\mathbf{r}' \qquad [3.4]$$

where $G(\mathbf{r}',\mathbf{r};t)$, the Green's function for the geometry, can be seen as an impulse response function. Note that the "$\mathbf{r}'$ " in this expression only means that we allow for the impulse source to be applied anywhere, not just in the origin as in Eq. 3.1 and 3.2. The photon hitting density can now be written as[12]

$$v(\mathbf{r};\mathbf{r}_1,\mathbf{r}_2;t) = \frac{1}{G(\mathbf{r}_1,\mathbf{r}_2;t)} \int_{\frac{|\mathbf{r}-\mathbf{r}_1|}{v}}^{t-\frac{|\mathbf{r}_2-\mathbf{r}|}{v}} G(\mathbf{r}_1,\mathbf{r};t')G(\mathbf{r},\mathbf{r}_2;t-t')dt' \qquad [3.5]$$

Eq. 3.5 may need some explanation. The inverted Green's function outside the integral is merely a normalization factor. The integration limits are the same as were discussed earlier in section 2.5. The first Green's function in the integral represents the photon density in $\mathbf{r}$ due to

the impulse source in $\mathbf{r}_1$. The second represents the probability that the photons in $\mathbf{r}$ will reach $\mathbf{r}_2$ given the specific time criteria, the escape function. That this probability is also given by a Green's function may not be obvious, but viewing the point $\mathbf{r}$ as the position of a new source and $\mathbf{r}_2$ as the position where the photon density from this source is to be calculated might be helpful. The difference is that in the first case (from $\mathbf{r}_1$ to $\mathbf{r}$) the source was an impulse in both space and time, and in the second case (from $\mathbf{r}$ to $\mathbf{r}_2$) the 'source' is only an impulse in space. The spreading in time is taken care of by the convolution integral.

This is the definition of the photon hitting density presented in Ref. 12. Here, the first Green's function represents ρ in $\mathbf{r}$, and the second, the escape function, represents ρ in $\mathbf{r}_2$ due to a source in $\mathbf{r}$. One might argue that the escape function should instead be defined by the photon current density multiplied by a normal vector, $\mathbf{n}\cdot\mathbf{J}(\mathbf{r}_2,t-t')$, when the detector is located outside the medium[13]. This quantity would then be a measure of the photon flux over a small area dA on the boundary at $\mathbf{r}_2$, due to a source in $\mathbf{r}$. Nevertheless, using the photon density ρ as the escape function has the advantage that some degree of symmetry is evident in the problem, which makes calculation simpler.

After this theoretical survey, we conclude that the photon hitting density in which we are interested of computing is proportional to

$$v(\mathbf{r};\mathbf{r}_1,\mathbf{r}_2;t) \propto \int_{\tau_1}^{t-\tau_2} \rho(\mathbf{r}_1,\mathbf{r};t')E(\mathbf{r},\mathbf{r}_2;t-t')dt' \qquad [3.6]$$

where the escape function E is either given by ρ as in Eq. 3.3 or $\mathbf{J}$ as given by Fick's law, and τ_1 and τ_2 represent the time spans given in Eq. 3.5.

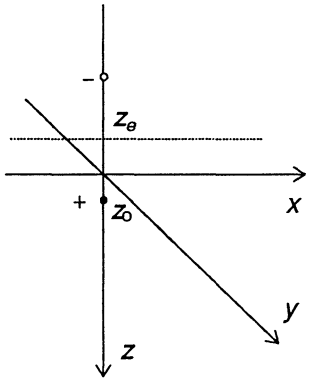


Fig. 3.1 The geometry for the semi-infinite half-space case. Positive source at z_0, the extrapolated boundary at $-z_e$ and a negative image source mirrored in the extrapolated boundary.

3.2 Computer Implementation of the Diffusion Approximation Approach

Calculations of the photon hitting density were performed for three simple geometries:

- A semi-infinite half-space, with the source and the detector placed on the boundary at some distance from each other.
- A slab with thickness d in the z-direction and infinite in the x,y-plane, with the source and the detector placed on opposite sides of the slab.
- A semi-infinite slab with thickness d and infinite in the y-direction but cut off at some x. The source and the detector were placed on opposite sides of the slab at some distance x_o from the edge.

The source was placed one transport mean free path length, z_o, from the boundary inside the sample. This emulates the actual physical situation where the incoming light beam is thought to enter the sample at the boundary, and then penetrate to some depth before the scattering can be regarded as isotropic. It is presumed that this depth is equal to z_o .

The 'extrapolated boundary' method was used to treat the boundary conditions. The photon density may have non-zero value on the physical boundary, which descends toward zero on the extrapolated boundary situated at a distance of z_e outside the physical boundary (Fig. 2.3). Placing the image sources is demonstrated in Fig. 3.2 for the most complicated of the three cases. The positive image sources are here situated in

$$(x = 0,\ z = 2kd + 4kz_e + z_o),\ \ k = 0,\pm1,\pm2,\pm3... \qquad [3.7]$$

and

$$(x = 2x_o + 2z_e,\ z = 2kd + (4k-2)z_e - z_o),\ \ k = 0,\pm1,\pm2,\pm3... \qquad [3.8]$$

and the negative in

$$(x = 0,\ z = 2kd + (4k-2)z_e - z_o),\ \ k = 0,\pm1,\pm2,\pm3... \qquad [3.9]$$

and

$$(x = 2x_o + 2z_e,\ z = 2kd + 4kz_e + z_o),\ \ k = 0,\pm1,\pm2,\pm3... \qquad [3.10]$$

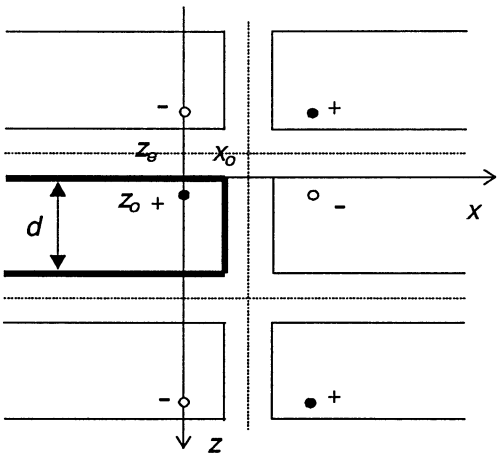


Fig. 3.2 Image sources for the case of a semi-infinite slab. Dotted lines represent the extrapolated boundary, where the photon density is equal to zero. Thin lines represent 'imaginary' slabs. Only the lowest order source dipoles are shown.

A piece of computer code for each case was written in C to carry out the calculations (App. A). The program first gets its input parameters from an input file. These include optical properties of the medium, the size of the region where to perform the calculations, time step size and total time (note: the mean cosine of the scattering angle g is not included, since we are dealing with the diffusion approximation and presume isotropic scattering, on a scale given by μ_s').

To be able to compare the results to those obtained by the Monte Carlo simulations, the photon hitting density should be integrated over small time intervals. The program evaluates the integral of the photon hitting density, in each point of a grid set up by the input parameters, over a number of time steps also given by the input. This means that for each time

step, and for each grid point, the program will have to carry out a double integral; the inner being the convolution integral as given by Eq. 3.6, and the outer being the integration over the time step size. This large amount of integration puts some strain on the numerical algorithm chosen to carry out the computation. *Numerical Recipes in C* recommends a ten-point Gauss-Legendre integration scheme, which has the advantage of only evaluating the integrand function value ten times[14]. For the double integral that means 100 function calls. The solution is also accurate, at least for 'well behaving' functions.

When calculating the integral there are two possible approaches. The integrand consists, as we have seen, of an infinite sum, which has to be truncated (Eq. 3.3). The integrand may be evaluated 'as is' for each call by the numerical algorithm, with an adequate number of terms in the sum taken into account,

$$\int_{t_1}^{t_2}\int_{\tau_1}^{t-\tau_2}\left(\sum_k \rho_k \sum_l E_l\right)dt'dt \qquad [3.11]$$

where ρ_k and E_l represent the k:th and the l:th terms in the photon density and escape functions, respectively.

The other possibility is to move the sum-signs out of the integral and perform the summation after the integration,

$$\sum_k \sum_l \left(\int_{t_1}^{t_2}\int_{\tau_1}^{t-\tau_2} \rho_k E_l dt'dt\right) \qquad [3.12]$$

The latter approach has the advantage that full control of the integration is possible, since comparison between two successive steps directly yields an estimation of the error. That makes it possible to know exactly when to truncate the sum. Nevertheless, it has one major drawback. The integration always requires a lot more computation than summing, and in this approach a full integral has to be computed for each term in the sum. The former method only needs one integration, but on the other hand there is no way to know if a sufficient number of terms in the sum have been used. A little physical insight can solve this problem, however. Looking at Fig. 3.2, we see that we are only interested in what happens between $z = 0$ and $z = d$, in a short moment after the pulse has been induced at $t = 0$. The image sources for higher values of k, which are situated far away from this area, cannot possibly have any influence here. That is partly because the 'light' simply cannot reach this area in time, and partly because the attenuation at long distances is too great. Using Eq. 3.3 for calculating the contribution from image sources of different orders, with realistic values of the optical properties, shows that even the contributions from the second order images are negligible.

In the case of the slab geometry ρ_k is given by

$$\rho_k = const \times \exp(-\mu_a ct)\left(\exp\left[-\frac{(x^2+y^2+(z-z_{pk})^2)^{1/2}}{4Dvt}\right] - \exp\left[-\frac{(x^2+y^2+(z-z_{nk})^2)^{1/2}}{4Dvt}\right]\right)$$

where the constant is given in Eq. 3.2, and

$$z_{pk} = 2kd + 4kz_e + z_o$$

and

$$z_{nk} = 2kd + (4k-2)z_e - z_o \qquad [3.13]$$

If E_l, the escape function, is chosen to be represented by ρ due to the photon density in $\mathbf{r} = (x,y,z)$ the expression for E_l is completely analogous, except that z_{pk} and z_{nk} then are replaced by

$$z^e_{pl} = (1-2l)d - 4lz_e$$

and

$$z^e_{nl} = (2l-1)d + (4l-2)z_e \qquad [3.14]$$

(see Fig. 3.3). Should the escape function be chosen to be represented by the flux across the boundary, $\mathbf{n}\cdot\mathbf{J}(\mathbf{r}_2,t-t')$, instead, the derivative of Eq. 3.13 on the boundary has to be taken.

In the case of the semi-infinite slab the expressions for ρ_k and E_l become slightly more complicated, since two more terms for each k and l are added and the positions of the image sources depend on x as well, as in Eqs. 3.7 - 3.10.

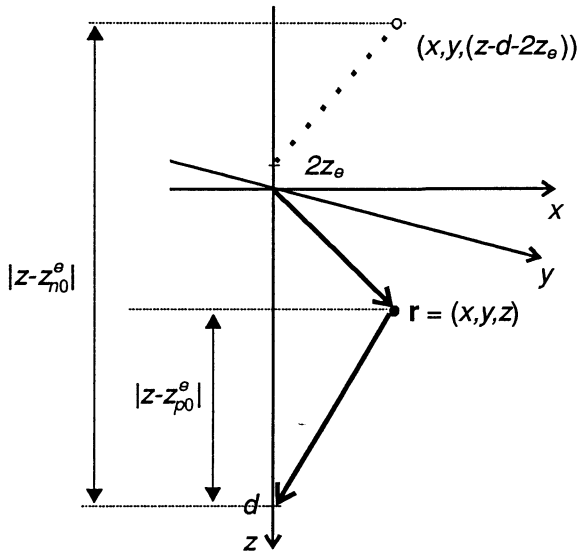


Fig. 3.3 The geometry in the case of a slab. $|z-z^e_{pl}|$ and $|z-z^e_{nl}|$ represent the distances along the z-axis from the images of the induced sources in every point (x,y,z) to the detection point. Only the zeroth order is shown, i.e. a 'real' induced source somewhere between $z = 0$ and $z = d$, and its negative counterpart.

3.2.1 Reciprocity Approach

In the discussion so far we have assumed that the proper way of calculating the photon hitting density at a certain position is a two step process: first calculate the photon density in the point due to the original source, then multiply this by an escape function which is given by the photon density (or photon current density) at the detector due to a source in the point. As we have seen, this leads to a convolution integral that has to be evaluated for every position (x,y,z) for every time t. In order to make a computationally efficient algorithm, it would be

interesting to investigate whether this scheme can be enhanced in some way. Calculating the photon density due to the source seems inevitable, but the escape function may be a candidate for improvement.

Consider the role played by the escape function in the scheme described above. The escape function fetches the probability of escape at the detector from a certain point in the sample. This is repeated over and over again; in practice a complete new calculation of the photon density at the detector for each point. When dealing with analytical forms this is no problem, since the photon density at any time, in any point, due to a source in any other point, is given directly by an analytical expression. In a practical case, however, a numerical integration scheme must be used. Such a scheme always starts with a set of initial values and iteratively works its way to the desired solution. Carrying out this process for every single point is computationally intensive.

The solution might be to reverse the calculation of the escape function, i.e. regard the detector as a source and calculate the photon density in every point, starting at time t going backwards to time $t = 0$ (Fig. 3.4). It may not be obvious that this approach yields the escape function, but the calculation is performed over the same set of optical properties, although in opposite directions, and should thus contain the same information. This is also indicated by the symmetry of the photon hitting density. Switching positions of the source and the detector and then calculating the photon hitting density renders exactly the same result. This *reciprocity theorem* has been more thoroughly shown by Gordon[15], and formally the equality involves multiplication with a certain factor.

The reciprocity assumes diffusive scattering, and can thus not be applied for models which include forward scattering, at least not directly. However, using *hybrid-scattering*, described in section 4.2.2, there may be a possibility of using reciprocity in combination with the Monte Carlo method.

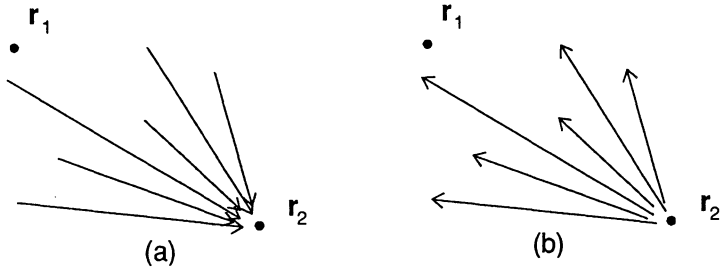


Fig. 3.4 Computing the escape function. Instead of calculating the photon density at r_2 due to sources in every point (a), the reciprocity theorem states that it is possible to calculate the photon density in each point due to a source in r_2 and still obtain the same information (b).

3.3 The Monte Carlo Simulation Approach

As stated earlier, calculating the photon hitting density by using Monte Carlo simulation is a conceptually simple method. Log the photon packets as they traverse through the medium, and if the packet hits the position that is appointed as the detection point, add the photon path to a result matrix. Eventually, when an adequate number of photons has been 'detected', the sum of photon paths has formed a distribution which should approximately equal the photon hitting density (Fig. 3.5).

In practice, however, a number of difficulties become apparent, all of which have their origin in the computationally time consuming nature of the Monte Carlo method. Since we are only interested in the photons that actually hit the detector, a large part of the computation done by the computer is a waste. The fraction of photons that contribute depends on the optical properties of the simulated medium and the geometry, primarily the distance between the source and the detector, but even under very favorable conditions this fraction is no more

than perhaps 1:1000 or 1:10000. This means that the total number of photon paths that has to be calculated is typically several millions. As the distance to the detector increases, the number of detected photon packets decreases rapidly. At some distance it is no longer useful to perform calculations, because the statistics become too poor for any reasonable computation times.

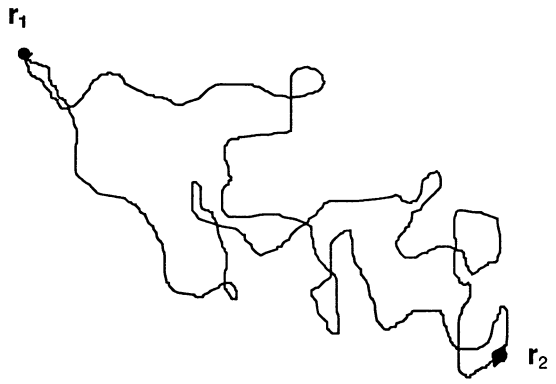


Fig. 3.5 Two photon paths have begun to build up the distribution that will eventually yield the photon hitting density.

Closely connected to the problem with the distance to the detector are the optical properties of the medium. The scattering mean free path here acts as a sort of scaling factor. Increasing the mean free path by a factor of ten (i.e., decreasing μ_s by the same amount) means that only a tenth of the calculations is needed for a photon packet to reach a certain point. This opens a possibility of performing faster simulation, by varying the optical properties so that μ_s has a minimum value while μ_s' still has the same value. That can be accomplished by setting $g = 0$, something which is discussed in more detail in section 4.2.2.

Another problem is the finite nature of the method. The result has to be presented as some sort of discrete matrix or array. The grid element size must be a compromise between on the one hand high accuracy, which calls for small elements, and on the other hand larger elements to obtain acceptable statistics within each element.

For certain simple geometries, such as the homogeneous slab, the statistics can be significantly enhanced by taking advantage of symmetries. In the case of the slab, introducing cylindrical coordinates reduces the problem from three spatial dimensions to two when it comes to plotting the result. In the general case, when the distribution of photon paths is a function of x, y, z, and t, determining the grid element size (i.e., Δx, Δy, Δz and Δt) may be a delicate matter.

The way Monte Carlo simulation is normally implemented, the photon packets are not actually logged along the whole of the traversed path, but rather merely in the interaction points. This implies that there is not much point in having the spatial grid elements much smaller than the mean free path length, because otherwise the distribution would be built from scattered points rather than connected paths, requiring yet more computation to get good statistics. Larger elements thus smooth the curve. As far as accuracy is concerned, larger grid elements are allowed as long as their size is small compared to the over all variation of the distribution. The problem might here arise near the source and detector, where the distribution may be expected to vary rather sharply. The same goes for the time steps. Since a complete spatial distribution is required for each time step, these steps must be large enough to give good statistics, i.e. enough photons must be detected within each time interval. The

upper limit of the step size is determined by the condition that it must be small compared to the over all variation in detected intensity.

The finite size of the detector also has to be considered. The simplest way of implementing detection is to assign detector status to a certain (spatial) grid element, and thus make the detector size equal to the element size. It is tempting to make the detector larger, in order to obtain a higher fraction of detected photons. In reality, the detector size would be determined by the diameter of an optical fiber or in the future perhaps the pixel size of an arrayed semiconductor-detector, in any case not much larger than around 1 mm^2. Also, when comparing the results to those obtained by the diffusion approximation method, one must keep in mind that that method assumes an infinitely small detection point. Setting the detector size equal to the grid element size seems sensible for the following reason: the photon hitting density at the source and at the detector must be the same, because, per definition, every photon packet that has contributed to the photon hitting density distribution has been both at the source and at the detector. Now, in the result matrix, the photon hitting density at the source will be represented by the number of photon interactions within the grid element that contains the source. To get the same value at the detector, the detector area should then be of the same size as the element containing the source.

3.4 Computer Implementation of the Monte Carlo Approach

The computer code used to perform the Monte Carlo simulations is a modified version of the public domain program 'Monte Carlo Simulation in Multi-layered Turbid Media' (MCML), written in C, by Wang and Jacques[16]. Modifications include implementation of time resolution, possibility of using other geometries than multiple layers, and customization of the code for computing the photon hitting density. A brief description of the program flow is given in the following section. See also App. B.

3.4.1 Program Flow of MCML

At first, MCML gets its input parameters from a text file, prepared in advance, and the simulation is initialized. These parameters are: optical and geometrical properties of the medium, result array size and grid spacing, time resolution, number of time steps, position of photon source and detector and number of photons to include in the computation. The input parameters are stored in a special structure, `InputStruct`.

Another structure, `OutStruct`, is assigned to handle the output parameters. One array is used for logging the photon packet, one for storing the result (by *array*, in this context, is meant an array of any dimension). The sum of the detected photon weight, for each time step, is stored in one array; this gives a measure of the detected intensity as a function of time. The number of detected photon packets in each time step is also stored, to give a hint of the quality of the statistics.

A third structure, `PhotonStruct`, is used to keep track of the properties of the photon packet during the simulation. It contains the current position (x,y,z), time t and direction (ux,uy,uz) of the photon packet, as well as its current weight. It also carries a Boolean variable to tell whether the photon packet is 'dead' or 'alive', used for terminating the photon packet if it crosses the boundary or is absorbed totally by the medium.

When the simulation starts, the photon packet is assigned an initial weight of unity, together with its input position and direction. Then the distance to the next interaction point is calculated as

$$S = -\frac{\ln R}{\mu_t} \qquad [3.15]$$

where R is a random number between 0 and 1. To make sure that the 'random' number really is random, a special random generator is included in the code, rather than depending on the internal system random generator, which may or may not fill the requirements of generating adequate random numbers. The random generator was picked for this task because it gives proven fair random numbers and, not least important, it is relatively fast[14].

The next step is to check whether the photon packet hits a boundary on its way to the next interaction point. Assuming it does not, the packet is moved, and its weight w is reduced by the amount

$$\Delta w = w\frac{\mu_a}{\mu_t} \qquad [3.16]$$

The corresponding position in the logging array is increased by one, flagging that the packet has been there. The total elapsed time is updated by adding S/v to the previous time, where v is the speed of light in the medium.

To close the loop, the final step is to calculate a new direction for the packet. The azimuthal scattering angle is calculated as $\psi = 2\pi R$, with R a random number like before. The deflection scattering angle θ is calculated using the Henyey-Greenstein distribution. In the case of isotropic scattering, calculating the deflection angle is a particularly simple task: $\theta = \pi R$.

The three steps described above, i.e. moving the photon packet to a new interaction point, updating the photon packet properties and calculating a new direction of propagation, are the spine of the simulation and are what keep the processor busy most of the time when running `MCML`. When optimizing program performance, the largest part of the efforts was concentrated on the routines that control these three steps.

Should the packet hit a boundary, two possibilities occur. Either the medium on the other side of the boundary is scattering, or it is non-scattering. In the first case, the distance S is split and the packet is moved to the boundary, where it is put up to the possibility of Fresnel reflection. Again, a random number is drawn, and then compared to the reflectance r, given by Eqs. 2.8 and 2.9. The packet is then moved the remaining part of S, either reflected back into the medium which it came from or transmitted into the new medium, depending on the outcome of the Fresnel reflection test.

If the medium on the other side is non-scattering, the photon packet is also moved to the boundary and put up to the Fresnel reflection test. Should it be transmitted it escapes the sample and is terminated, unless of course it happens to escape at the position of the detector. In that case the arrays in `OutStruct` are updated. The logging array now consists mostly of zeros, with a path of ones stretching from the source to the detector (and perhaps twos, threes, etc. if the path crosses itself). This array is multiplied by the final value of the photon weight w and added to the result array, for the appropriate time step, as exemplified by:

$$[\text{Result}(ix,iy,iz,it)] \rightarrow [\text{Result}(ix,iy,iz,it)] + \begin{bmatrix} 0 & 0 & 0 & 0 & . & 0 \\ 0 & 1 & 0 & 0 & . & 0 \\ S & 1 & 2 & 0 & . & 1 \\ 0 & 0 & 1 & 1 & . & D \\ . & . & . & . & . & . \\ 0 & 0 & 1 & 0 & . & 0 \end{bmatrix} * w \qquad [3.17]$$

S and D here stand for source and detector, respectively. The logging array is generally in three dimensions, except when symmetries can be used to reduce one variable, and it is indexed by the integers *ix, iy* and *iz* in units of the grid element size.

Multiplication by w has to be done because the weight in a sense represents the probability that a single photon traveling a particular path will actually hit the detector, as compared to all other possible paths. Thus, a photon packet that has traveled a long path will have a small weight left because of absorption, indicating that a lot of the photons trying to follow this path will be absorbed, and accordingly that this path will have less influence on the resulting distribution than photons following shorter paths. Eventually, the logging array is set to all zeros and a new photon packet is launched.

The possibility of the photon packet being absorbed by the medium also has to be dealt with. When the weight has fallen below some lower limit, the packet is terminated, since it would only be a waste of computing effort, and thus time, to continue the simulation for very small photon weights. This implementation is unphysical in the sense that some 'energy' will be lost in the process. Therefore, the termination is handled in a special routine called the Roulette. There is a large probability p, e.g. 0.9, that the packet just vanishes. If the packet survives, here with the probability 0.1, its weight is increased by $1/(1 - p)$, here 10, and it is allowed to continue.

When all photon packets have been launched and either detected or terminated, the simulation is finished and all that is left for MCML to do is to write the result-array, detected intensity-array and number of detected photons-array to an output file.

3.4.2 Output From MCML

Two versions of the code were designed. In the first, the only allowed geometry is a homogeneous slab, with non-scattering media on either side. The thickness of the slab has no restrictions however, so the semi-infinite geometry can be simulated by making the slab thick enough. The code makes use of the cylindrical symmetry if the detector is placed opposite to the source and stores the result in an (*ir,iz,it*)-array, where *ir* is the integer part of $r = (x^2 + y^2)^{½}$ in units of the grid element size. This implementation uses all the detected photons, which have traversed three-dimensional paths, to plot the result in a two-dimensional matrix. This procedure is however impossible if the source and the detector are placed on the same surface, as in the case of the semi-infinite medium, because the cylindrical symmetry is lost. When presenting the result as a function of *ir* the output array has to be scaled in order to obtain the same sampling density for all values of *ir* (Fig. 3.6). For higher values of *ir* the sampling volume increases, and to obtain the true photon hitting density each column in the array (representing *ir*) has to be divided by $(2ir + 1)$. This factor comes from normalization of the sampling volume with respect to the innermost sampling region, i.e. $ir = 0$.

The second version also utilizes a slab, but any number of inhomogeneities may be introduced within it. These are restricted to being shaped as cubes, spheres or cylinders and must all possess the same refractive index as the slab. They may otherwise have any dimen-

sions and may carry other scattering and absorption coefficients. It is also possible to have one inhomogeneity within another or make them overlap. The inhomogeneities are defined with the same resolution as the grids in `OutStruct`.

Presenting the results is more problematic for this version of the program. The result array in `OutStruct` is of four dimensions, (*ix,iy,iz,it*). The ideal presentation would perhaps be a three-dimensional density plot, but that is difficult to show on a computer screen and even harder to print on paper. Another way is to cut slices of the distribution to display in two dimensions. The problem is getting good enough statistics within a relatively thin slice, as compared to the statistics obtained from the full 3D-space in the first version described above. An additional problem is that statistics tend to get worse when introducing inhomogeneities, because the inhomogeneities often act as obstacles that lower the probability of the photons reaching the detection point.

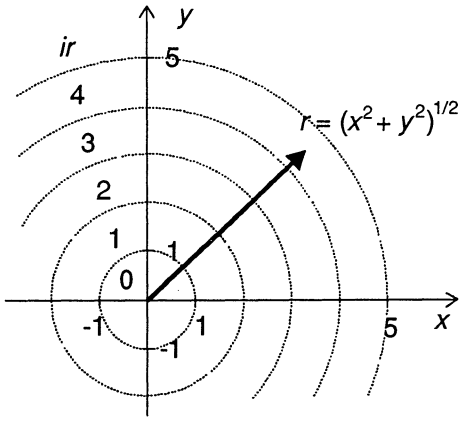


Fig. 3.6 When presenting the result as a function of the radius, the sampling volume for element *ir* is a factor $(2ir + 1)$ larger than for the innermost element, i.e. $ir = 0$. Since the desired property is a density, the result array has to be scaled accordingly. The coordinate *ir* represents the array index and is given by the integer part of *r*.

4. Results and Discussion

The results from the numerical calculations using the diffusion approximation and the Monte Carlo model are presented below. The photon hitting density is shown as intensity plots in two or three dimensions, and the features of these plots are discussed. Finally, a comparison between the two models is made.

The values of the optical properties used in the calculations are chosen so that they are within the range of those which have been measured for biological tissue[6].

4.1 Diffusion Approximation

Although calculated numerically, the results from the diffusion approximation calculations represent analytical solutions. The model is deterministic, which makes it easy to produce a great variety of plots with different input parameters. Computation time is typically in the order of one minute on a Pentium 75 MHz processor, and it does not depend much on the input parameters. The photon density ρ in the detection point is also shown as a function of time for each case. This represents the measured light intensity when using a time resolved detection technique.

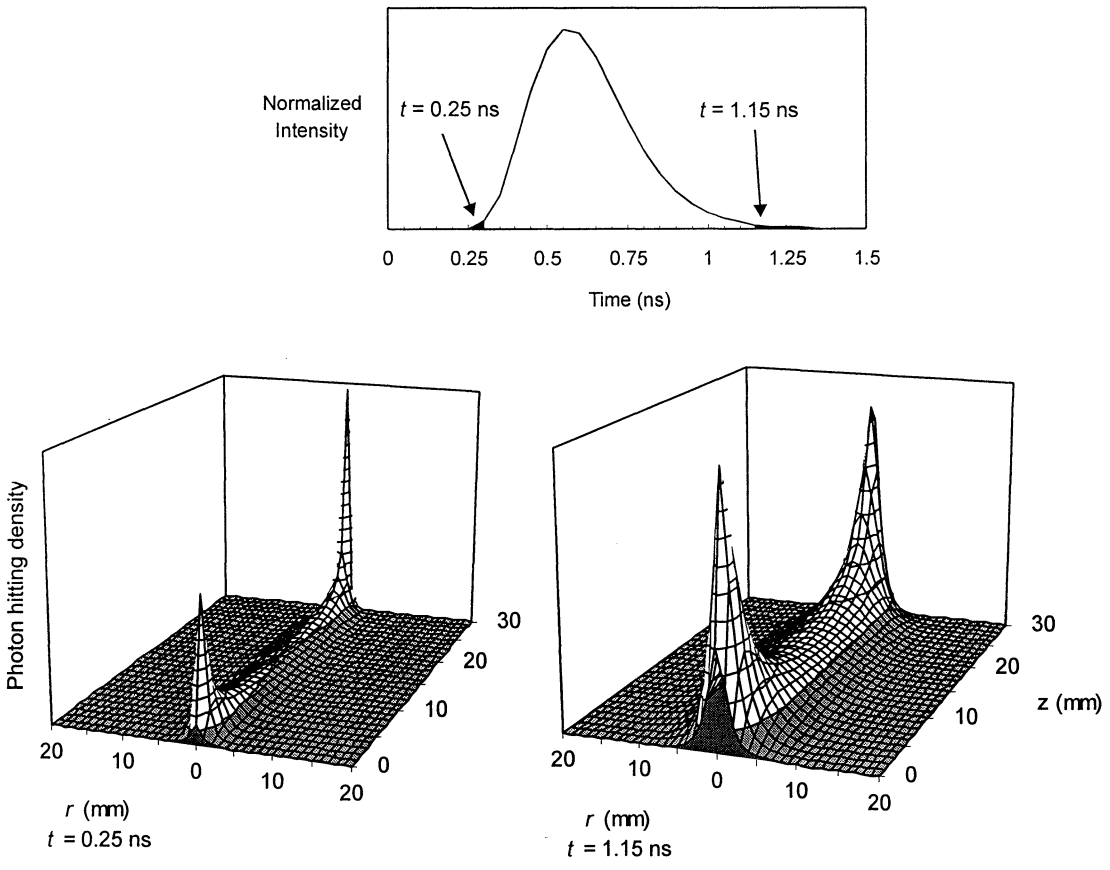


Fig. 4.1 In the lower part of the figure, the photon hitting density in the case of a slab with thickness 30 mm is shown for two time intervals: the early arriving light between 0.25 and 0.30 ns, and the late arriving light between 1.15 and 1.20 ns. The source is placed at $r = 0$, $z = z_0$, and the detector at $r = 0$, $z = 30$. The optical properties used in the calculations were $n = 1.4$, $\mu_a = 0.05$ mm^{-1}, $\mu_s' = 1.5$ mm^{-1}. The contributing parts of the TPSF are shown above.

Fig. 4.1 shows two typical plots of the photon hitting density for a slab geometry. In the first, the early arriving light is detected and the distribution is centered to a narrow corridor along the line between the source and the detector. In the second, light that has traversed a long path through the medium is detected, and there is a large probability that the photons have been far away from the center line. As can be expected the distribution is strongly peaked near the source and the detector. This means that the probability of the photons having been there is high. The validity of the details in these regions must be regarded as uncertain, because the diffusion approximation does not stand near the source, and the results can thus not be regarded as accurate in this region. Valid conclusions, so far, are that the photon hitting density has high values near the source and the detector, and possibly that the peaks are less pronounced for the late arriving light. In some of the following 3D-plots, the logarithm of the photon hitting density is shown, to restrain the peaks and emphasize regions far away from the source and detector.

The cross-section of the photon hitting density for constant values of z is an interesting property. This cross-section might be expected to be approximately constant along the z-axis, because the same number of photons should have passed through such a cross-section for any value of z. This is, however, not entirely true, because the photons may travel back and forth across a cross-section of constant z, and the probability of their doing so could vary depending on the distances from the source, the detector and the boundaries. Nevertheless, calculations show that the cross-section in fact is fairly constant along the z-axis, something which might not be evident from just looking at the plots in Fig. 4.1.

Fig. 4.2 shows a comparison between the cross-sections of the photon hitting densities in Fig. 4.1 for $z = 15$ mm, i.e. half way between the source and the detector. The diagram clearly shows that the early arriving light traverses a much narrower corridor than the late arriving light, although the distribution still spreads over several millimeters even for the early light.

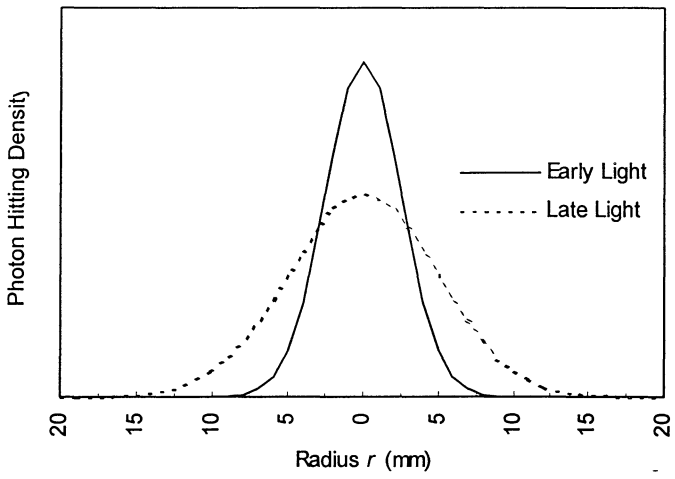


Fig. 4.2 The cross-section of the photon hitting density taken at $z = 15$, for the early arriving light and the late arriving light. The data is the same as in Fig. 4.1.

4.1.1 Some Notes on the TPSF

The shape of the TPSF-curve directly carries some information on the dominating optical properties of the medium. The first part of the curve, approximately from the beginning of the pulse to its peak, is largely determined by the scattering coefficient of the medium. Thus, the rule is the higher the scattering, the longer the pulse rise time, giving a more leveled out early portion of the curve. The last part of the curve, the decay, on the other hand, is princi-

pally determined by the absorption. It shows a typical exponential decay, which can be associated with the absorption coefficient[17].

4.1.2 Semi-infinite Half-space

The results from the semi-infinite half-space are the characteristic 'banana-shapes' reported by many authors, see Fig. 4.3. The early light shows a narrow distribution near the surface. Later in the pulse, the distribution becomes thicker and extends farther inside the medium, because photons spending much time near the surface have a large probability of escaping across the boundary. This may be useful if one is interested in depth-selective monitoring.

Method	Diffusion Approx.	*Abs.* μ_a (mm^{-1})	0.03
Geometry	Half-space	*Scatt.* μ_s' (mm^{-1})	5.0
Grid Spacing (mm)	1	*Mean Free Path* z_0 *(mm)*	0.2
Time Step (ps)	200	*Extrapol. Bound.* z_e *(mm)*	0.37
Ref. Index ***n***	1.4	*Surrounding Ref. Index*	1.0

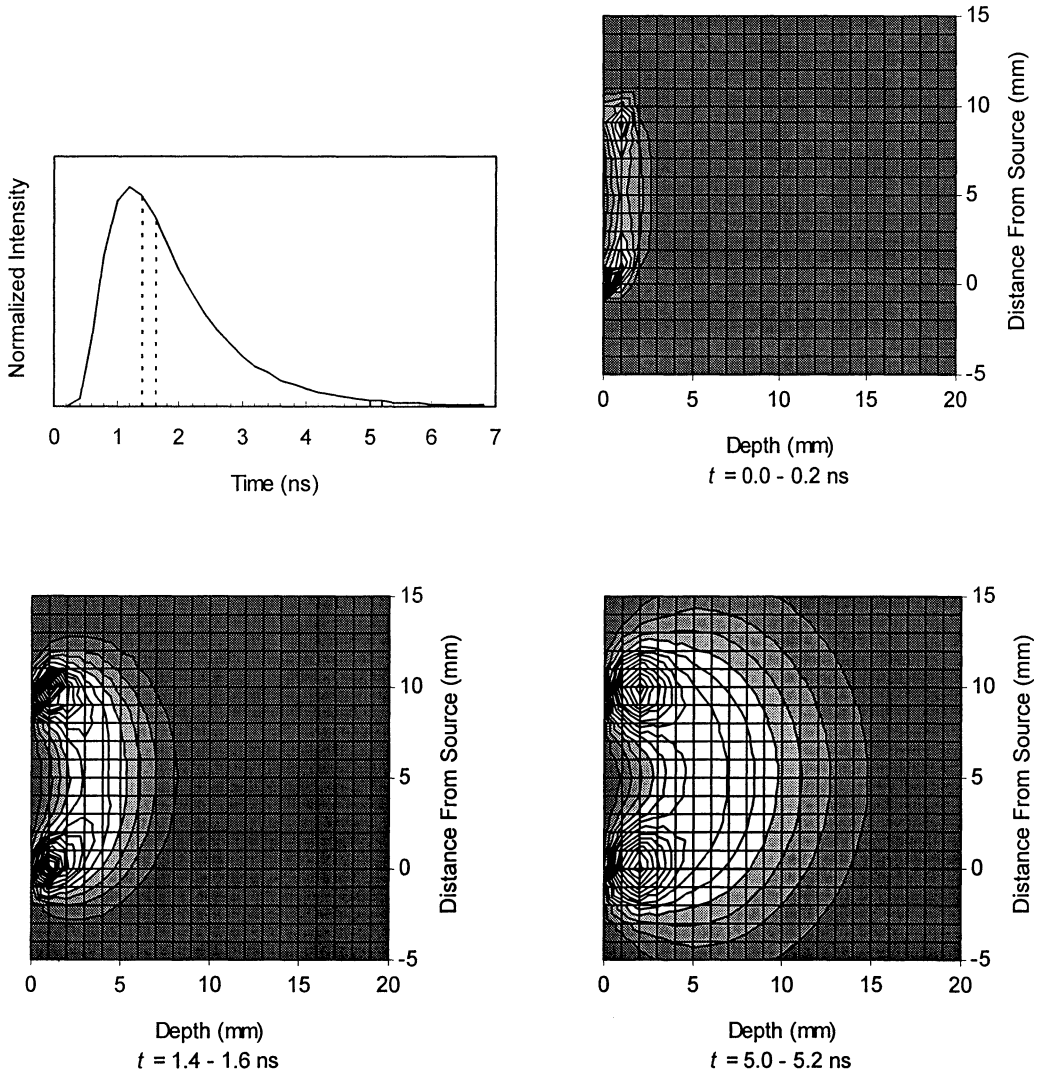


Fig. 4.3 The photon hitting density for a semi-infinite half-space, with a distance of 10 mm between the source and the detector. Three different time intervals are shown: the early arriving light, light from the peak of the pulse, and the very late arriving light.

4.1.3 Infinite Slab

Most calculations were performed in the slab geometry, since this is easy to compare with the Monte Carlo results. In Figs. 4.4 and 4.5 the photon hitting density is shown for a 15 mm slab with different sets of optical parameters; high scattering and low absorption in the first case, lower scattering and higher absorption in the second. The most obvious difference is the time scale; the detected pulse is much faster in the low-scattering case.

Method	Diffusion Approx.	*Abs.* μ_a *(mm^{-1})*	0.05
Geometry	Slab, 15 mm	*Scatt.* μ_s' *(mm^{-1})*	4.0
Grid Spacing (mm)	1	*Mean Free Path* z_0 *(mm)*	0.25
Time Step (ps)	50	*Extrapol. Bound.* z_e *(mm)*	0.45
Ref. Index ***n***	1.4	*Surrounding Ref. Index*	1.0

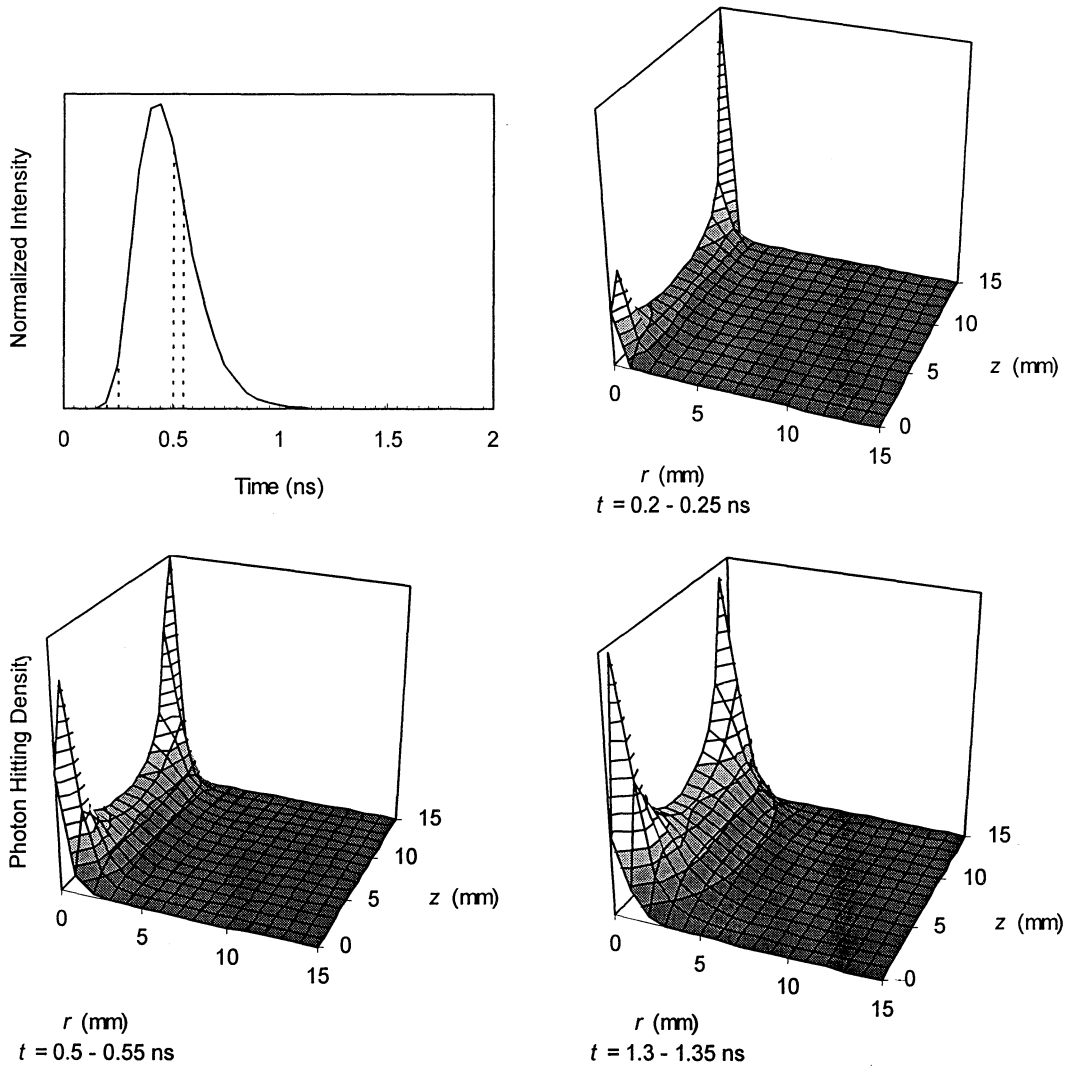


Fig. 4.4 The photon hitting density for a 15 mm slab with relatively high scattering and low absorption.

Method	Diffusion Approx.	*Abs.* μ_a (mm^{-1})	0.1
Geometry	Slab, 15 mm	*Scatt.* μ_s' (mm^{-1})	1.0
Grid Spacing (mm)	1	*Mean Free Path* z_0 *(mm)*	1.0
Time Step (ps)	20	*Extrapol. Bound.* z_e *(mm)*	1.66
Ref. Index ***n***	1.4	*Surrounding Ref. Index*	1.0

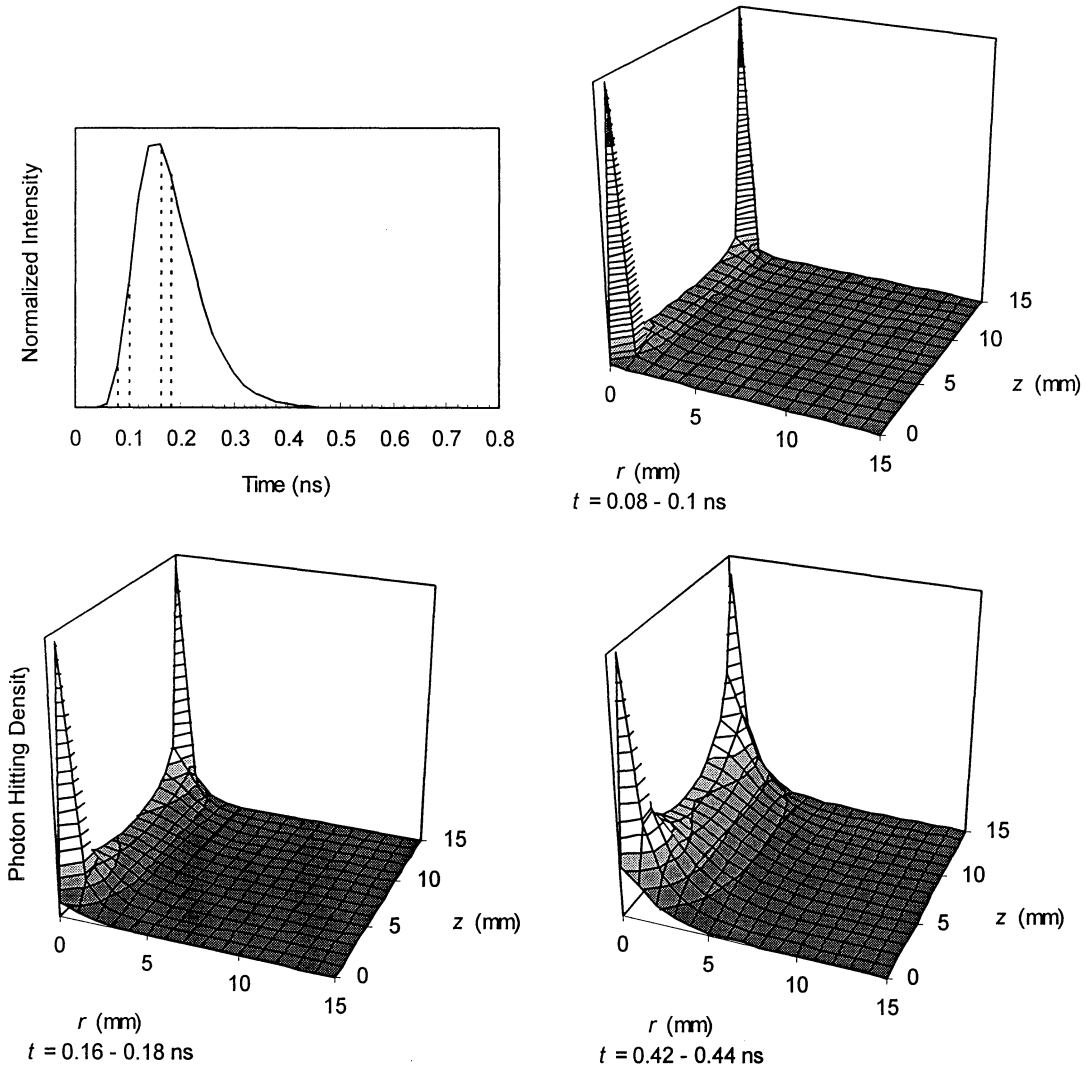
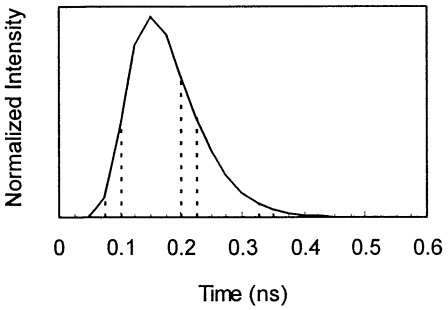


Fig. 4.5 The photon hitting density for a 15 mm slab with relatively low scattering and high absorption.

The hitting density has maxima at the source and the detector, which is expected. However, the hitting density drops rapidly near the boundary (as it must do, because of the zero value condition on the extrapolated boundary just some millimeter outside the physical boundary). This seems unphysical, because in reality all the incident photons cross the boundary and the hitting density should have a maximum there. The phenomenon is discussed further when comparing with the Monte Carlo results in section 4.3.

A set of plots of the photon hitting density for a slab geometry is shown in Fig. 4.6, which is also used as comparison against the Monte Carlo results in section 4.2.2.

Method	Diffusion Approx.
Geometry	Slab, 15 mm
Grid Spacing (mm)	1.0
Time Step (ps)	25
Ref. Index n	1.4
Abs. μ_a *(mm*$^{-1}$*)*	0.1
Scatt. μ_s' *(mm*$^{-1}$*)*	1.0
Mean Free Path z_o *(mm)*	1.0
Extrapol. Bound. z_e *(mm)*	1.66
Surrounding Ref. Index	1.0



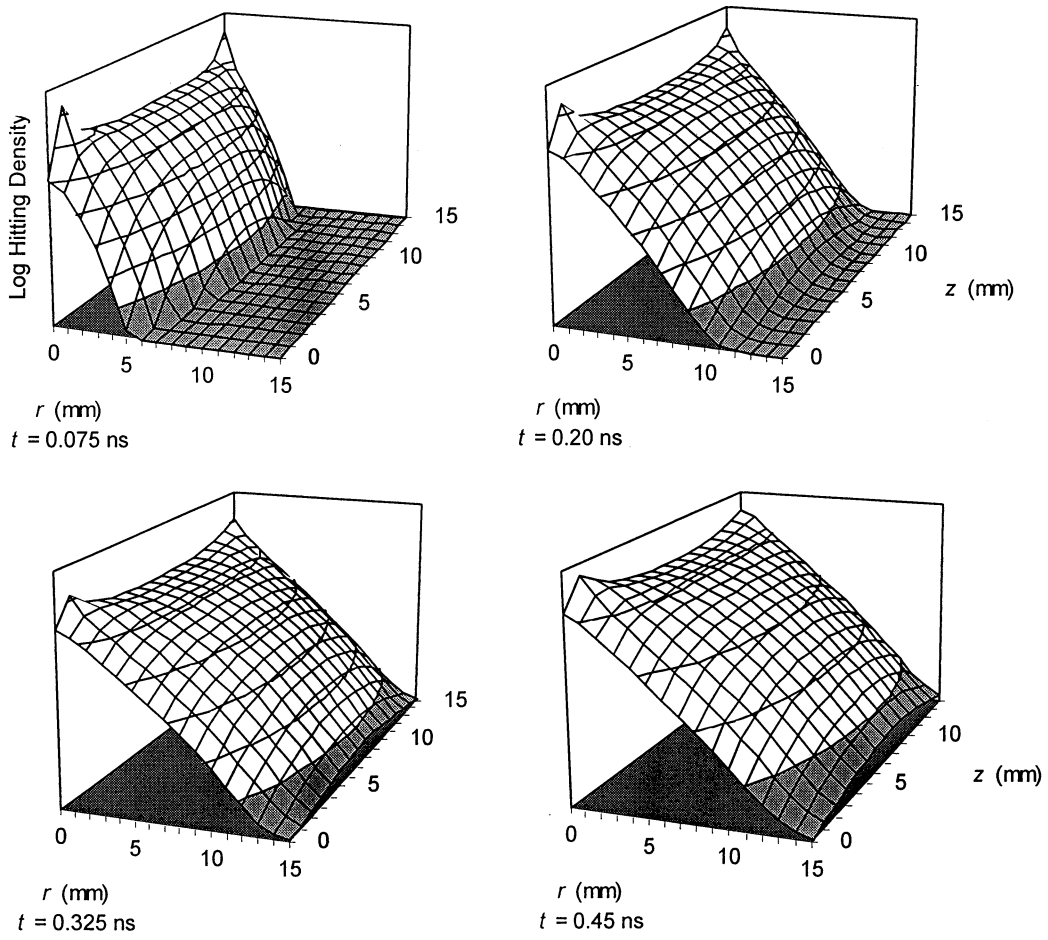
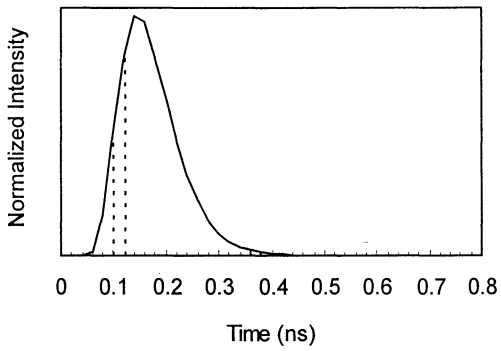


Fig. 4.6 The photon hitting density for a 15 mm slab, calculated in four different time intervals. Note that the hitting density is here presented using a logarithmic scale. The corresponding Monte Carlo results can be seen in Fig. 4.10.

4.1.4 Semi-infinite Slab

Fig. 4.7 shows plots of the photon hitting density for a semi-infinite slab geometry. The distribution is not affected at all by the presence of the slab edge for the early arriving light, but it gets more so the longer the light has traveled through the medium. Eventually the distribution resembles more the banana path in Fig. 4.3. The corresponding results from the Monte Carlo simulations may be viewed in Fig. 4.13 in section 4.2.4.

Method	Diffusion Approx.
Geometry	Semi-slab
Grid Spacing (mm)	1.0
Time Step (ps)	20
Ref. Index n	1.4
Abs. μ_a *(mm*$^{-1}$*)*	0.1
Scatt. μ_s' *(mm*$^{-1}$*)*	1.0
Mean Free Path z_o *(mm)*	1.0
Extrapol. Bound. z_e *(mm)*	1.66
Surrounding Ref. Index	1.0



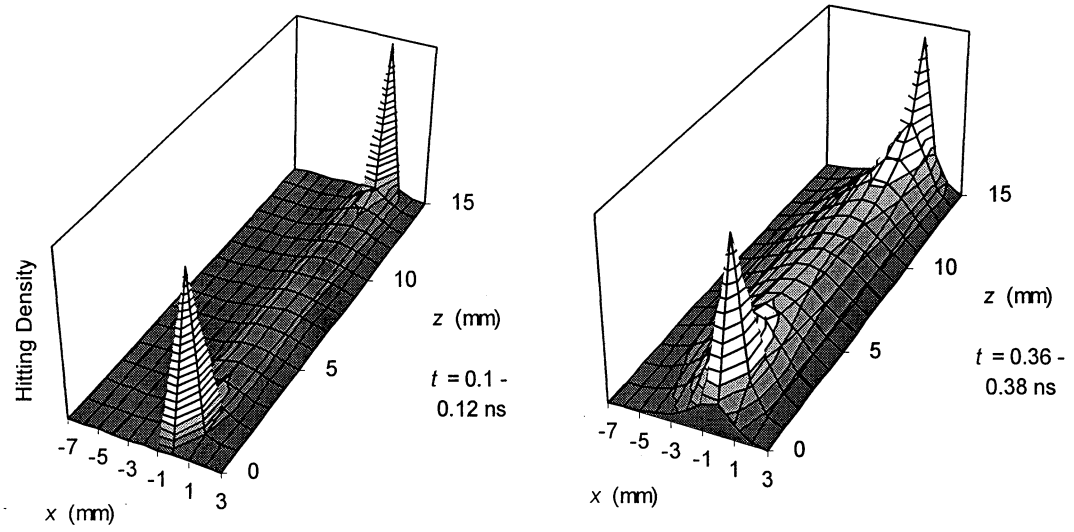


Fig. 4.7 The photon hitting density for a semi-infinite slab, calculated using the diffusion approximation, where the source and detector are placed on opposite sides at a distance of 3 mm from the edge. Fig. 4.13 shows the corresponding Monte Carlo results.

4.2 Monte Carlo Simulation

The statistical Monte Carlo method allows virtually any geometry and set of optical parameters, but in practice this freedom is radically reduced by the slowness of the method. Simulations were performed on a DEC-Alpha 150 MHz processor. Typically, several hours, and in many cases even days, were necessary to obtain good statistics. With some exception, each simulation is unique and there is no way of using the result from one run to quickly compute what the result from another set of input parameters would be. (When employing *white* (or *condensed*) *Monte Carlo simulation*, one single run is performed where the absorption coefficient is set to zero. If the total path length of each photon packet is logged, the effects of different absorption can then be added later[18,19]. Here, the effects of different absorption coefficients are not particularly considered, and for the purpose of this thesis the method was not regarded useful as a general means of reducing computation time.)

To make a comparison to the results from the diffusion approximation possible, simulations with simple geometries were emphasized.

4.2.1 General Features of the Monte Carlo Generated Results

The plots generated by Monte Carlo simulation all have a typical irregular structure, due to the stochastic nature of the method. When the statistics are poor, which is often the case, these irregularities are even more pronounced. This makes it difficult to draw quantitative conclusions regarding the results obtained. Some interesting properties can be observed, however, giving physical insight in the processes that are modeled.

Consider Figs. 4.8 and 4.9, showing the photon hitting density in a specific situation in three ways: as 3D-plots using linear and logarithmic scales, respectively, and as a 2D linear-scale plot. First, we observe that the hitting density peaks at the boundary, where the photon packets are launched, and at the detection point. This is a general feature of the Monte Carlo results; the distribution always has its peaks at the boundary, not at some distance inside the sample as could occur when applying the diffusion approximation. Different aspects on this will be discussed later when comparing the two models.

An asymmetry is evident in the 2D-plot. The peak at the photon entrance side is stretched out as compared to the one at the detection side. This is expected, and explainable from a physical point of view. The photon packets enter the sample as from a beam, all with the same initial direction. It will then take some distance before they lose all information of the initial direction and start to propagate diffusely. On average, that distance equals the transport mean free path length, but some of the photons will travel farther in the forward direction before their 'direction memory' is lost (described by the exponential law), hence the stretched out structure. At the detection side, on the other hand, the photons propagate in an almost perfectly diffusive manner, and may hit the detector from other directions than the one defined by the normal. The possible angles of incidence for the photons to reach the detector depend on the refractive index – for a certain angle total reflection occurs – and also the numerical aperture of the detector. The smallest of these determines the maximum angle of incidence.

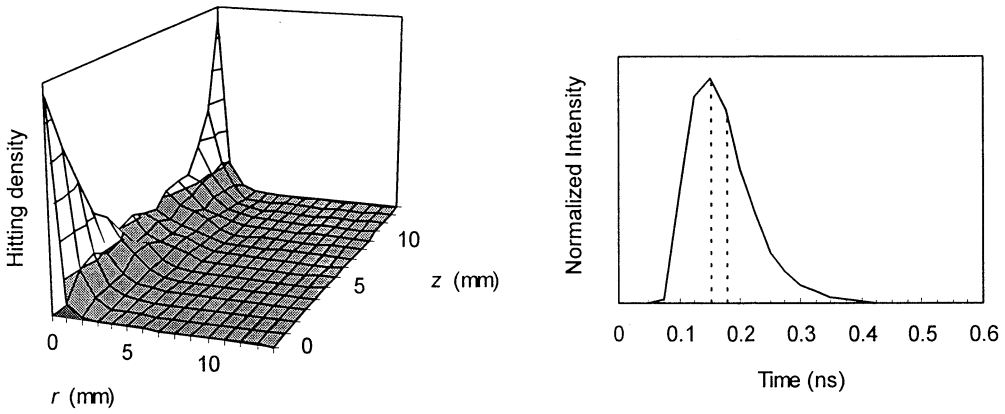


Fig. 4.8 The photon hitting density in transmitting of a 15 mm slab, sampled in the time interval 0.15 - 0.175 ns. The distribution is formed from 740 photon paths. $\mu_a = 0.1$ mm^{-1}, $\mu_s = 10$ mm^{-1}, $g = 0.9$, $n = 1.4$. The normalized intensity (the TPSF-curve) is obtained by summation of the weights of the detected photon packets for each time step.

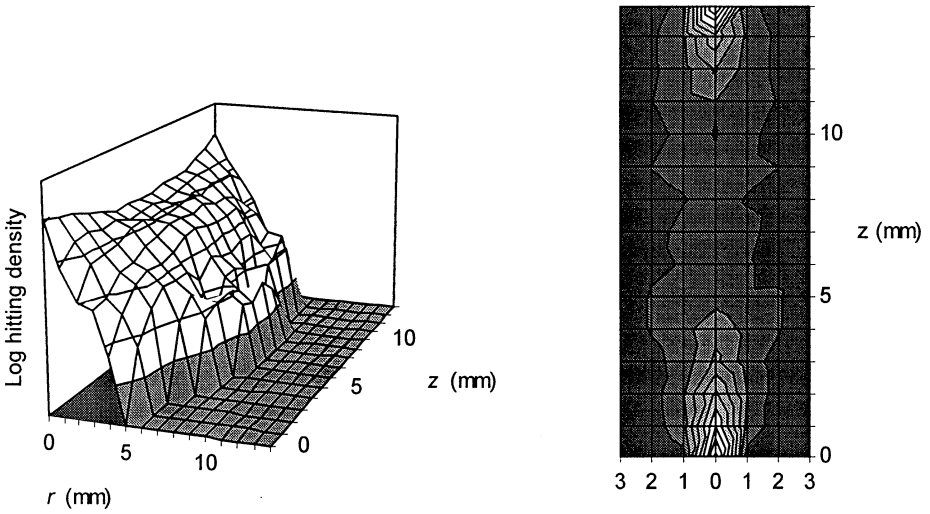


Fig. 4.9 The same photon hitting density as in Fig. 4.8, but presented in logarithmic 3D- and linear 2D-plots, respectively. The 2D-plot is mirrored in $r = 0$ to give a better view of the features near the photon entrance point and the detection point.

4.2.2 *Effects of Different Values of* g

A unique feature of the Monte Carlo method, as compared to the diffusion approximation, is the possibility of varying the value of the parameter g, i.e. the amount of forward scattering. In the diffusion approximation, recall that the parameters μ_s and g are lumped together into the effective scattering coefficient μ_s'. It is interesting to see whether this simplification makes the model inaccurate. This can be accomplished by investigating the results from Monte Carlo simulations where the values of μ_s and g are changed so that the value of μ_s' is kept a constant. The results of such computations can be viewed in Figs. 4.10 and 4.11. Fig. 4.6 shows the corresponding result when using the diffusion approximation. As explained in section 3.3, the computation time increases greatly for values of g near unity, because the photon packet interactions occur at much smaller distances when the value of μ_s is increased to keep μ_s' a constant.

The desired conclusion would be that the value of g has no significant influence on the photon hitting density. This would not only be important for the validity of the diffusion approximation, but it would also mean that isotropic scattering can be employed for the Monte Carlo simulations and thus save much computation time in the further investigations.

It is difficult to draw any firm quantitative conclusions from the plots in Figs. 4.10 and 4.11, because of the poor statistics, and a thorough investigation should include computations using many different values of the parameters μ_s' and μ_a. Nevertheless, there are apparently no great differences between the two cases $g = 0$ and $g = 0.9$. A tendency of the distribution being cut off at large radiuses appear for $g = 0.9$, but that might be explained by the poor statistics in those regions, amplified by the logarithmic scale. In any case, the photon hitting density is so low for the values of r in question, that the phenomenon should be of less importance.

The plots in Fig. 4.11, where $g = 0.9$, have a smoother appearance than those in Fig. 4.10, where $g = 0$. This is due to the larger number of interaction points when $g = 0.9$. Over all, the statistics are better in Fig. 4.10, because more photon paths contribute to the distribution.

In the following simulations, especially those already made computationally intense by for example complex geometries, isotropic scattering is generally applied to bring down the computation time.

Note When the distance between the entry point and the detector becomes very small, less than about 10 mean free path lengths, the above approximation does not hold. Some of the photon packets will traverse the medium with no interactions at all, creating a peak of *ballistic*, non-scattered, photons. This peak would not be apparent if the appropriate forward scattering were applied. To deal with this, Yamada *et al.* have investigated the possibility of *hybrid-scattering* Monte Carlo simulation[31]. When applying this method, a specified number (e.g. 10) of the first interactions the photon packets encounter are calculated using forward scattering, while the succeeding interactions proceed using isotropic scattering. This gives accurate results for thin samples while still giving a substantial reduction in computation time.

Method	Monte Carlo	*Mean Cosine* g	0.0
Geometry	Slab, 15 mm	*Red. Scatt.* μ_s' (mm^{-1})	1.0
Grid Spacing (mm)	1	*Mean Free Path* z_o *(mm)*	1.0
Time Step (ps)	25	*Surrounding Ref. Index*	1.0
Ref. Index n	1.4	*Total No. of Photons*	60 000 000
Abs. μ_a (mm^{-1})	0.1	*No. of Detected Photons*	27 302
Scatt. μ_s (mm^{-1})	1.0	*Simulation Time (h)*	14.4

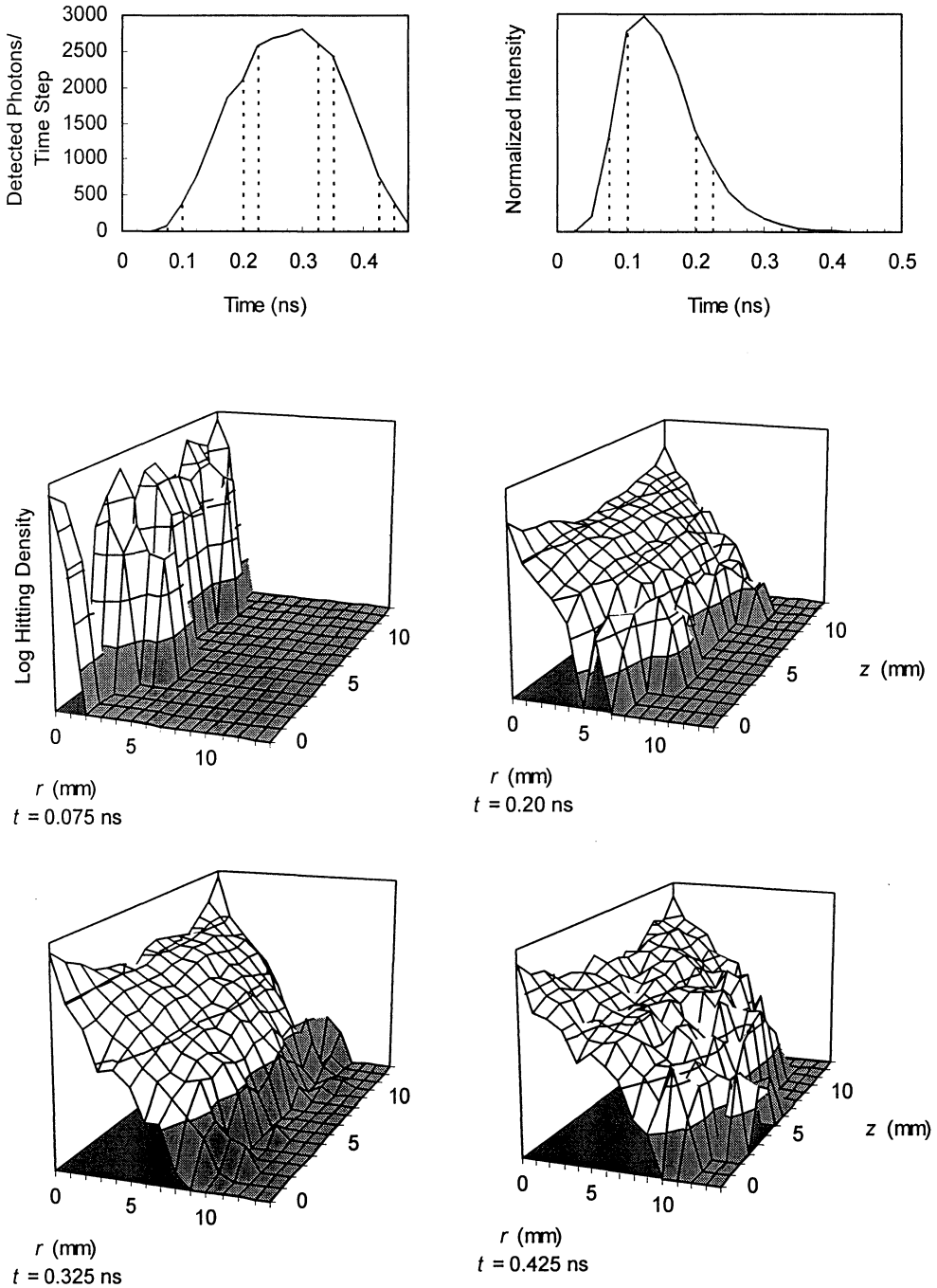


Fig. 4.10 The photon hitting density for a 15 mm slab, calculated using Monte Carlo simulation, displayed for four different time intervals. The isotropic scattering case. For the corresponding diffusion approximation results see Fig. 4.6.

Method	Monte Carlo	*Mean Cosine* g	0.9
Geometry	Slab, 15 mm	*Red. Scatt.* μ_s' *(mm*$^{-1}$*)*	1.0
Grid Spacing (mm)	1	*Mean Free Path* z_o *(mm)*	1.0
Time Step (ps)	25	*Surrounding Ref. Index*	1.0
Ref. Index n	1.4	*Total No. of Photons*	30 000 000
Abs. μ_a *(mm*$^{-1}$*)*	0.1	*No. of Detected Photons*	15 783
Scatt. μ_s *(mm*$^{-1}$*)*	10.0	*Simulation Time (h)*	80

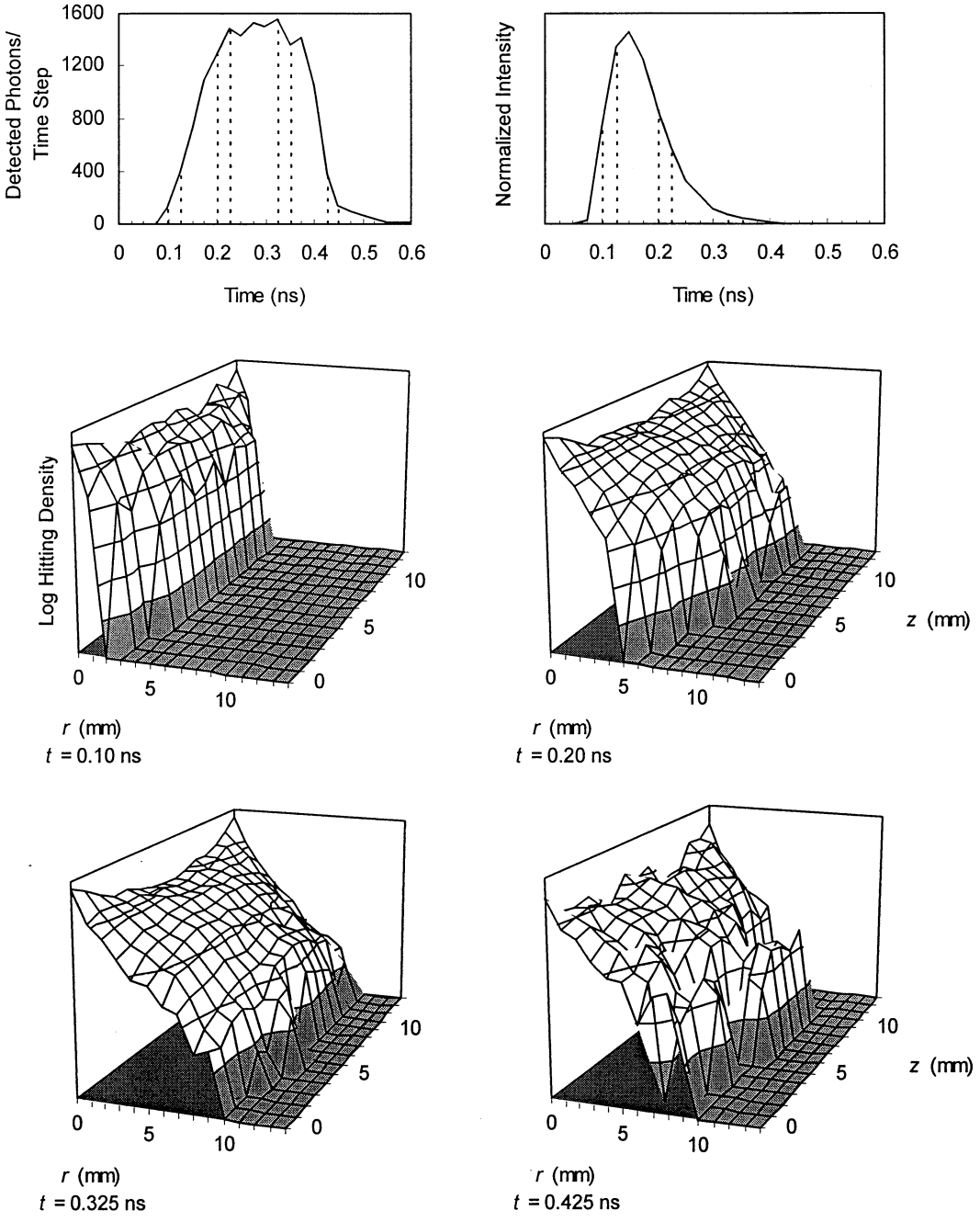


Fig. 4.11 Same geometry and μ_s' as in Fig. 4.10, but forward scattering with g = 0.9.

4.2.3 Single Absorbing Inhomogeneity

Fig. 4.12 shows the results from photon hitting density computations where a totally absorbing 3×3×3 mm inhomogeneity was introduced halfway between the source and the detector in a 15 mm slab. When comparing the TPSF in Fig. 4.12 with the one in Fig. 4.10 , the same case without inhomogeneity, it may not be obvious that the ballistic component is lost in the presence of the single absorber, but the results in Fig. 4.12 suffer from poor statistics. Also, in a true comparison the relative height of the peaks must be taken into account, and not just normalized versions of the curves.

Method	Monte Carlo	*Mean Cosine* ***g***	0.0
Geometry	Slab, 15 mm	*Red. Scatt.* μ_s' *(mm*$^{-1}$*)*	1.0
Grid Spacing (mm)	1	*Mean Free Path* z_0 *(mm)*	1.0
Time Step (ps)	25	*Surrounding Ref. Index*	1.0
Ref. Index ***n***	1.4	*Total No. of Photons*	100 000 000
Abs. μ_a *(mm*$^{-1}$*)*	0.1	*No. of Detected Photons*	7 536
Scatt. μ_s *(mm*$^{-1}$*)*	1.0	*Simulation Time (h)*	60

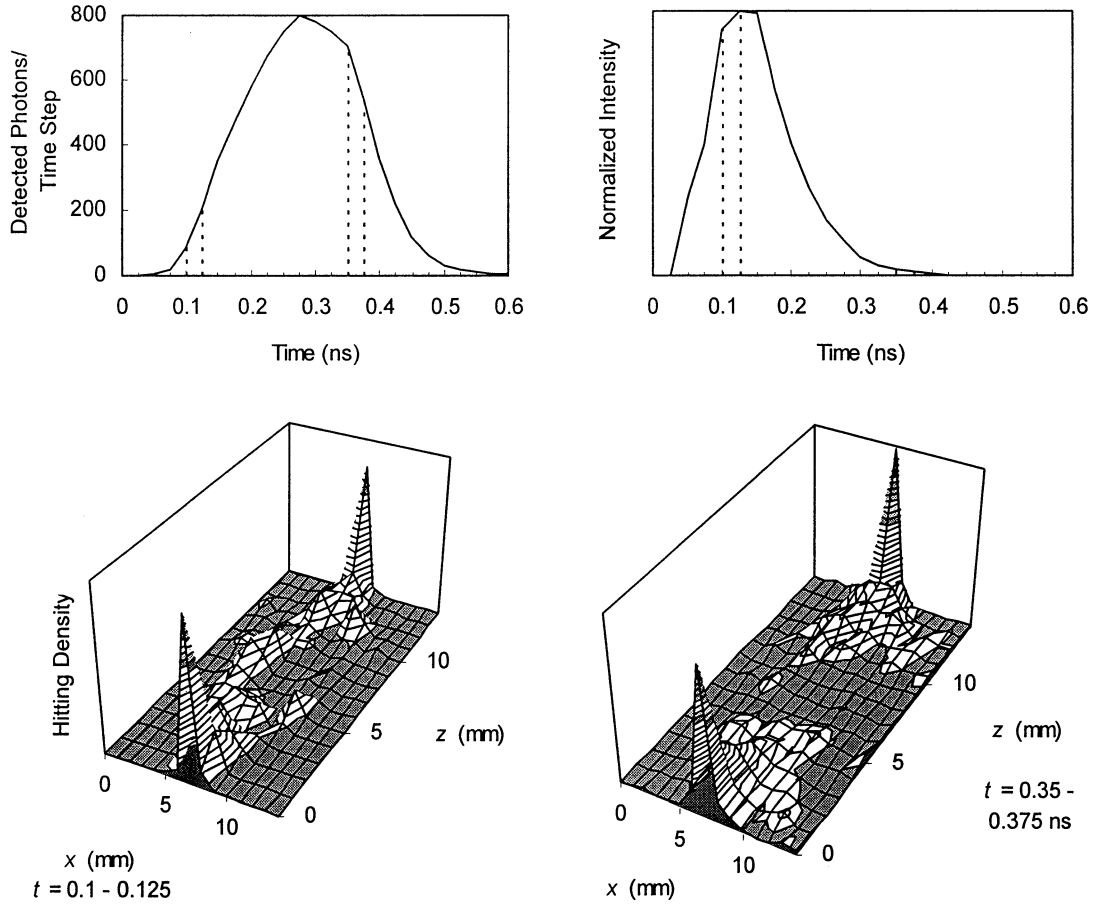


Fig. 4.12 Hitting density for a 15 mm slab where a 3×3×3 mm inhomogeneity is placed halfway between the source and the detector.

4.2.4 Semi-infinite Slab

Fig. 4.13 shows plots of the photon hitting density for a semi-infinite slab computation. Because of the large loss of photons across the edge boundary, the statistics are poor, in spite of the long simulation time. The plots shown are taken in 5 mm thick slices integrated for three successive time steps to enhance the statistics. There is a fair resemblance with the corresponding diffusion approximation results presented in Fig. 4.7.

Method	Monte Carlo	*Mean Cosine* g	0.0
Geometry	Semislab, 15mm	*Red. Scatt.* μ_s' (mm^{-1})	1.0
Grid Spacing (mm)	1	*Mean Free Path* z_o *(mm)*	1.0
Time Step (ps)	25	*Surrounding Ref. Index*	1.0
Ref. Index n	1.4	*Total No. of Photons*	140 000 000
Abs. μ_a (mm^{-1})	0.1	*No. of Detected Photons*	8 914
Scatt. μ_s (mm^{-1})	1.0	*Simulation Time (h)*	65

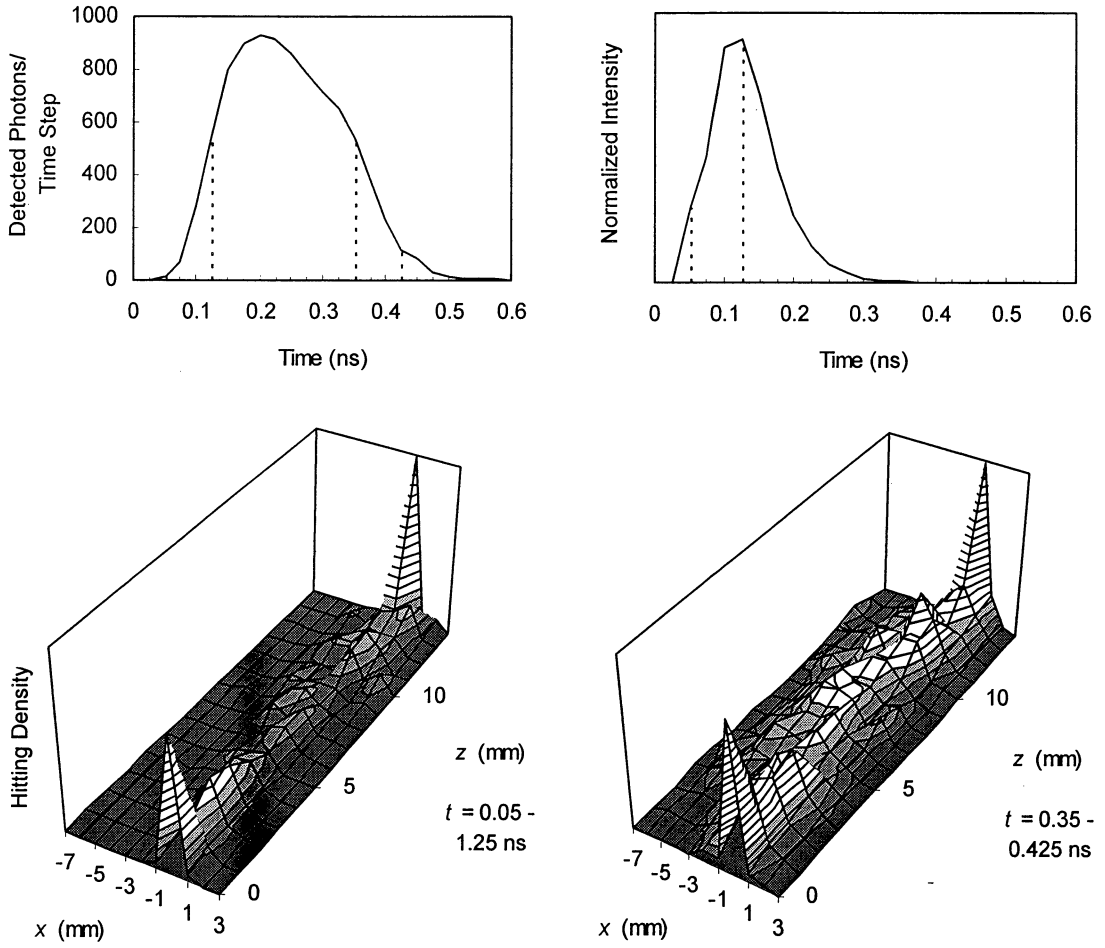


Fig. 4.13 Photon hitting density for a 15 mm thick semi-infinite slab where source and detector were placed 3 mm from the edge, shown for two time intervals.

4.3 Comparison and Discussion

With some minor exceptions, the results from the calculations performed using the diffusion approximation and the Monte Carlo simulations appear to be quite similar, although no rigorous quantitative comparison has been made. The exceptions can be explained by the limited validity of the diffusion approximation.

The optimum regime for performing photon hitting density calculations differs between the two models. Due to the approximations made, the diffusion approximation is best for large scattering and low absorption, and when the detector is located far away from the source. Such conditions are not at all favored by the Monte Carlo model, because of the computation time. Instead, the Monte Carlo model is best when scattering is relatively lower, absorption higher, and the distance between photon entry point and detector is small. A compromise between these two standpoints has to be made when comparing the two models. The optical properties of the medium are given by those of real tissue, i.e. $\mu_a \sim 0.1$ mm^{-1}, $\mu_s' \sim 1.0$ mm^{-1}. A detector-source distance of 15-20 mm was used in the calculations when the models were to be compared.

The diffusion approximation is not valid near the source; the photons must be allowed to travel at least a few mean free path lengths before the propagation may be regarded as diffusive. It is also near the source that the two models differ mostly. There is a large discrepancy in the results in the region just inside the boundary near the source. The diffusion results have peaks at z_o which descend rapidly near the boundary, whereas the Monte Carlo results have maxima immediately on the boundary and a descending slope away from the boundary. With the interpretation that the Monte Carlo results represent the 'true' solution, this indicates that the diffusion model might have to be modified in some way near the source.

In the diffusion approximation results, there is a tendency that the peaks become less pronounced and move toward the center of the sample for the very late arriving light. This can be interpreted as an indication of that the photons that have survived a long time within the sample have small probabilities of having been near the boundaries. It should then be expected that the same is true for the Monte Carlo model, but since the statistics for the very late arriving light are so poor, it is not possible to validate the assumption. In practice, the intensity of the very late arriving light is so low that the problem should be of less significance.

Although Monte Carlo simulations perhaps give the most realistic results, with current computer technology, the method in the form presented in thesis is not practically useful for computing the photon hitting density. Should the concept be incorporated in an image reconstruction algorithm, the proper model probably is the diffusion approximation. However, there is one possibility concerning the Monte Carlo model that is worth further investigation. From above we know that the hybrid-scattering approach gives valid results with reduced computation time. The bottle-neck of the Monte Carlo method is the very small fraction of 'detected' photons. Applying the reciprocity approach as described in section 3.2.1 is a possibility of getting around that problem. A combination of these two time reducing techniques may be an interesting approach.

4.3.1 Hybrid Model

The problem of the inaccuracies of the diffusion approximation near the source can be handled by using a hybrid model of the Monte Carlo and diffusion approximation approaches. Monte Carlo simulation is then employed within a radius of a few transport mean free path lengths from the source, which is the critical regime for diffusion theory.

Farther away from the source, when scattering can be regarded as diffuse, the diffusion approximation takes over with its greater computation speed[20].

4.3.2 Non-diffusive Photon Migration

The detected light pulse is sometimes said to be composed of three regimes with different characteristics. The first photons that reach the detector are called *ballistic*. These have encountered no interactions at all on their passage through the tissue. As the tissue sample gets thicker, the amount of ballistic photons that emerges on the detection side decreases exponentially. For tissue samples thicker than ~1 cm practically no ballistic photons are detected. They are not expected to show up within the limited statistics of the Monte Carlo simulations either if the sample is thicker than a few mean free path lengths.

The *snake* photons have been scattered a few times, but they essentially follow smooth paths from the light source to the detector and still move principally in the forward direction when they hit the detector.

The third regime is the diffusive photons, which have been scattered many times and have lost all information of their initial direction. In an experiment with a highly scattering medium, several mean free path lengths thick, almost all the detected light comes from diffusely scattered photons.

The diffusion approximation is not valid for the ballistic and snake photons, since these do not migrate diffusely. The ballistic photons present a trivial case – straight lines – and may be useful in some cases where the sample is thin. The snake photons, which have been scattered a few times, can be modeled using path integrals. Some authors have reported preliminary results from employing path integrals on the early arriving light, but no substantial information on this topic seems to be available at present[21,22]. This type of non-diffusive photon migration is not considered further in this thesis.

4.3.3 TPSF Comparison

Figure 4.14 shows the TPSF curves for Monte Carlo and diffusion approximation computations using identical input parameters. As expected, the two curves correspond fairly well.

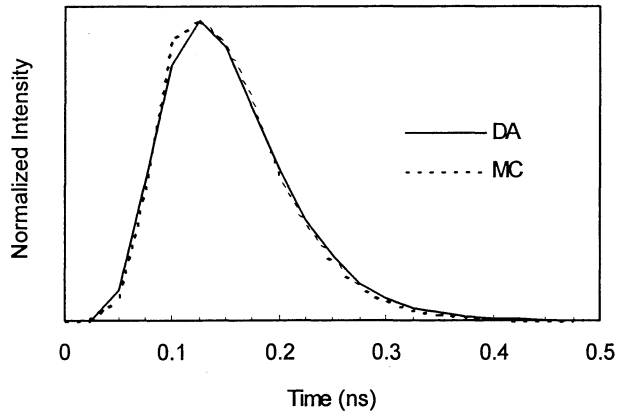


Fig. 4.14 The TPSF for a 15 mm slab, using the same input parameters for the diffusion approximation as for the Monte Carlo computations, μ_a = 0.1 mm^{-1}, μ_s' = 1.0 mm^{-1}, n = 1.4. The result was sampled in 25 ps time steps for both models. The results were normalized with respect to the peak values.

5. Tomographic Reconstruction

The concept of tomographic reconstruction has been around since 1917, when Radon first presented a solution to the problem[23]. Since the 1960's, with the birth of computers, tomographic imaging has found many applications, especially in medicine, but also for example in astronomy and in industrial applications such as material testing.

The first techniques used collimated x-rays to transilluminate the sample, with detectors placed on the opposite side. The reduction in intensity as the beams passed through the sample was measured, for a number of source-detector positions. This is still today the principle of the most common of tomographic imaging systems, the CT (*computed tomography*) scanner, in use at almost every hospital in the western countries[24]. The tomographic reconstruction algorithm most often used, the so called *filtered back projection*, relies on filtering of the measured data in the Fourier domain. It is a standard algorithm, available commercially in various software packages.

Other medical tomographic applications include the *positron emission tomography* (PET) and *magnetic resonance imaging* (MRI) methods. These rely on different principles, where it is not the question of transilluminating the sample, but rather detecting signals that have been induced to emanate from within the sample. The reconstruction is in these cases still based on straight lines, however.

A third class of tomographic methods embodies *ultra-sound diffraction tomographic imaging*[25,26] and *electrical impedance tomography*[27]. These are in a sense closer to the desired NIR tomographic method, since the signals, as for NIR, do not represent straight lines between two points outside the object. Consequently, they are newer methods and under development.

This chapter will review possible approaches to the optical (NIR) tomographic reconstruction problem. The simplest approach would be to apply the already available reconstruction algorithms used for CT, as will be discussed in the next section. A more sophisticated method which is tailored toward the specific needs of diffusive tomography will then be presented. At last, descriptions of the methods examined in the most current research will be given.

5.1 Using Filtered Back Projection for Reconstruction

The simplest means of tomographic reconstruction is to perform a mere *back projection.* The attenuation in the medium can be described by a density function $f(x,y)$, and for the detected intensity we have

$$\begin{aligned} I_{\text{det}} &= I_{in} \exp\left(-\int_{beam} f \cdot d\ell\right) \\ \Rightarrow \int_{beam} f \cdot d\ell &= \ln \frac{I_{in}}{I_{\text{det}}} \end{aligned} \qquad [5.1]$$

where the last quantity is measurable. The line integral is called the *Radon transform.* The object is to reconstruct the function f. Performing measurements for a fixed angle yields a *projection* of the sample onto an array of data (Fig. 5.1). Such measurements are made for several angles. Back projection means taking each measurement value, drawing a line with

constant 'intensity' (corresponding to the value) back along the line, and summing all lines for all measurements. This approach is obviously very simple but gives highly blurred images and it is of no practical use.

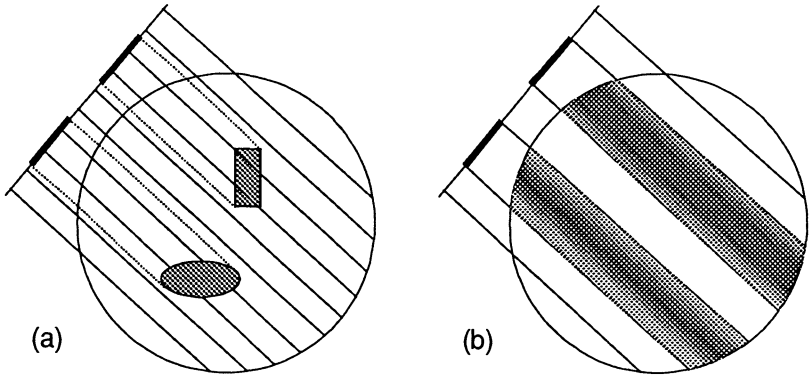


Fig. 5.1 The projection of two objects for one angle (a) and the back projection for the same angle (b).

A better technique is the *Fourier method*, in which each data array is Fourier transformed using the fast Fourier transform (FFT). This is repeated for all angles, yielding the 2D-FT of the image, where each 1D-FT array is represented on a line through the origin like spokes on a wheel. The tomographic image is rendered by 2D-FFT inversion. The problem is that the obtained 2D-FT is in polar coordinates, while the desired image usually is in Cartesian co-ordinates, requiring transformation from polar to Cartesian coordinate systems in the Fourier domain. This can be achieved using an interpolation scheme, but a generally better approach is to employ the filtered back projection method. The 1D-FT acquired for each angle is then multiplied by a ramp function, which acts as a high-pass filter, and is inverse transformed back again. The arrays are eventually back projected to render the image.

In medical applications, the FFT is frequently not performed at all and the filtering is instead accomplished by convolution in the spatial domain (*convolution back projection*). The convolution kernel, which is the inverse transform of the high-pass filter, essentially acts as a derivative operator on the data array. Since derivation amplifies noise, the kernel has to be modified. In the Fourier domain, this is accomplished by *apodisation*, i.e. the ramp function is multiplied by a window to cut off very high spatial frequencies. The apodisation also reduces aliasing, which otherwise causes the characteristic ringing artifacts that often appear when using discrete Fourier transforms.

The filtered back projection method was developed for x-rays following straight lines, attenuated when passing through tissue. How could this be applied for NIR tomographic reconstruction, where the photons definitely do not traverse straight lines, and where the attenuation is not given by just one parameter but two or three? As we have seen earlier in this thesis, with the time resolved detection technique, a large part of the detected light actually does follow fairly straight lines, or at least traverses in a narrow 'corridor'. This is especially true near the source and the detector, and of course for the early arriving light.

Various researchers have performed experiments with reconstruction using the filtered back projection, with some degree of success[28,29]. The reason for trying this was partly the availability of standard software, and partly to obtain results to compare to other methods. The method produced fairly accurate images of inhomogeneities near the surface, but as could be expected it was less successful in detecting more deeply embedded irregularities.

The back projection technique can be modified to take the diffusion into account by using deconvolution methods if the (spatial) point-spread function of the system is known. This yields somewhat better results[30]. The problem is how to obtain knowledge of the point-spread function.

To conclude, the use of standard CT algorithms does not seem to be the future solution of the problem of achieving image reconstruction in NIR applications.

5.1.1 The TEAM Approach

Yamada *et al.*[31] have developed a method for extracting the attenuation due to absorption only from time resolved measurements, for the early light. This opens the possibility of obtaining better results when using standard CT algorithms, since the measured data then no longer is a result of both scattering and absorption, and the early arriving light is used. The problem may be expected to be less non-linear. The approach is called the *temporally extrapolated absorbance method* (TEAM).

The principal idea of TEAM is to have a reference sample with the same geometry and scattering properties, with a known absorption, and compare the measurements, to calculate the absorbance for the early arriving light. The absorbance is defined as $\ln(T_{reference}/T_{object})$ where T denotes the transmitted intensity. This quantity cannot be evaluated directly, because the transmitted intensity for the early arriving light is too low. Therefore, the absorbance is evaluated as a function of time for the whole TPSF, and then extrapolated to the earliest arriving light.

Fairly good results have been reported when performing filtered back projection on TEAM data obtained from phantom measurements[31,32]. Nevertheless, the method appears to have major drawbacks, such as the need for a reference sample and the fact that the scattering is assumed to be homogeneous throughout the medium.

5.2 MATHEMATICAL BASIS OF THE RECONSTRUCTION PROBLEM

The experimental process of measuring NIR light that has traversed a medium may be considered as a mapping from a space of optical parameters to a space of measurements[33]. A linear model would be

$$\mathbf{y} = \mathbf{A}\mathbf{x} + \mathbf{r} \qquad [5.2]$$

where $\mathbf{x}$ is a vector containing the optical properties, $\mathbf{y}$ is a vector of the measurements, $\mathbf{A}$ is a projection matrix and $\mathbf{r}$ a vector describing measurement errors, noise, etc., the *residual vector*. The reconstruction would then be the inversion of Eq. 5.2 with some means of taking care of the vector $\mathbf{r}$, which is generally unknown, but from the measurements there may be some information on the noise, such as its variance. This model is applicable in the case of CT imaging, which is essentially linear. The problem in NIR imaging is, however, highly non-linear, and the above equation is generally not useful. One way around this is given by the iterative method as described by Singer *et al.*[34], reviewed below in section 5.3. Other means of controlling the non-linearity have been proposed; in the *perturbation approach*, described in section 5.4, the derivative of the mapping is defined and this is shown to be linear.

The projection matrix $\mathbf{A}$ is in general ill-conditioned, which in a physical sense means that a large variation in the optical properties results only in a small variation in detected intensity. As far as reconstruction is concerned, a small variation in the data then leads to a large change in the solution. Although, for the linearity reasons described above, the direct inversion of $\mathbf{A}$ is not performed, the reconstruction problem almost certainly will involve inversion of some ill-conditioned matrix. This makes the reconstruction sensitive to noise (the residual vector $\mathbf{r}$), so a successful reconstruction algorithm probably needs to incorporate some sort of noise control, such as application of some filter.

Concerning terminology, most proposed reconstruction methods employ a *forward model* for computing the light distribution in the medium, an *inverse model* for determining the optical parameters of a sample from light transmission measurements, and these are usually incorporated in an *iterative* scheme. Controlling the ill-posed nature of the inversion is called *regularization.*

5.3 A Proposed Method for Diffusive Tomographic Reconstruction

A method for tomographic reconstruction when the information-carrying beam has traversed a diffusive medium, as in the case of NIR probing of tissue, was presented by Singer *et al.*[34] The method is applicable to any diffusive process, although NIR tissue characterization may seem to be the strongest candidate for development at present.

The method is based on an iterative scheme, and utilizes a forward and an inverse algorithm. The process begins with an initial guess of the properties of the sample. The forward algorithm then uses this to calculate what the detected intensity at a given position would be if the sample really did look like the initial estimate. This value is compared to the actual, detected intensity, and the inverse algorithm changes the values of the estimate so that in the next iterative step the value from the forward calculation will be a little closer to the real. This is repeated until a satisfactory agreement is reached. The problem has generally no unique solution, i.e. many different sets of input parameters (optical properties) may produce the same detected intensity. This is where the tomographic part of the algorithm comes in, so that data from many different source-detector configurations are combined to produce a unique solution.

Perhaps simple in principle, the difficulties arise when it comes to implementing the algorithm. From a mathematical point of view, difficulties arise principally because the inverse problem is non-linear and ill-posed. When an adequate model has been developed, the next obstacle occurs: because of the complexity, it will require a much computation. The first step is choosing the forward and inverse models.

5.3.1 The Forward Problem

The 'forward problem' is the problem of calculating light intensity in an arbitrary position after light has been introduced in a medium with known optical properties. Different means of doing this have been discussed earlier in this thesis. These are various types of numerical solutions to the diffusion equation, and the Monte Carlo simulation method. Since the chosen model is to be incorporated into an iterative scheme, speed is crucial. This should disqualify Monte Carlo simulation from further discussion, because it is as we have seen painfully slow in most cases. That leaves us with the problem of solving the diffusion equation, or, more generally, the transport equation, for any arbitrary geometry.

In the original paper Singer *et al.* used a 'six-way flux model' as the forward model. The object is then divided into discrete volume elements, *voxels*, and each voxel is assigned a set of probabilities for scattering and absorption. Other models should also be possible, such as the finite element method or the alternating direction implicit method.

5.3.2 The Inverse Problem

The 'inverse problem' is the problem of how to change the optical properties of the initial guess to make the iteration converge. Ultimately this is an optimization problem; minimizing the sum E of the errors obtained for all source-detector configurations:

$$E = \sum_{l=1}^{N_d} \sum_{k=1}^{N_s} (P_{lk} - c_{lk})^2 \qquad [5.3]$$

where P_{lk} is the measured intensity at detector position l with the source at position k, c_{lk} the corresponding computed intensity, N_d and N_s are the total number of detection and source points, respectively. This is the least-squares error norm, but other error norms are possible, e.g. the χ^2 norm, which also takes noise into account. Arridge *et al.* [35] has pointed out that the definition of the error norm has a large significance for convergence.

Different minimization algorithms have been examined by various researchers. In the original paper, the so called *gradient descent method* class of algorithms was proposed[34]. Another interesting method is *simulated annealing*. This is the algorithm that was able to solve the famous 'traveling salesman' problem, in which the object is to minimize the distance of a journey between a number of cities visited in a successive order. Simulated annealing has the great advantage that it can circumvent local minima, a classic problem in optimization theory.

It is also possible to reformulate the problem to make the algorithm more efficient. Many researchers today use a perturbation approach, as will be described below.

5.4 Perturbation Approach

Arridge *et al.*[33, 36], Barbour *et al.*[37] and Feng *et al.*[38] have proposed a perturbation approach to the reconstruction problem. The sample, which consists of a set of optical parameters, can be represented by a vector field $\mathbf{p}(\mathbf{r})$ (the notation used in this section is adopted from Arridge). This simply means that $\mathbf{p}(\mathbf{r})$ returns the optical properties in the point $\mathbf{r}$. The forward problem can be seen as a (non-linear) mapping $\mathbf{X}_p \rightarrow \mathbf{Y}_M$ from a space $\mathbf{X}_p$ of optical parameters to a space $\mathbf{Y}_M$ of measurements. Let us define a measurement at $\mathbf{r}$ due to a source $\mathbf{q}$ as

$$F^{(M)}[\mathbf{r},\mathbf{q};\mathbf{p}] \qquad [5.4]$$

where F represents the forward model (which is now allowed to be non-linear) and M the type of measurement.

As far as perturbation is concerned, the interesting property would be the infinitesimal change in the measurement due to an infinitesimal change in the optical parameter vector field, i.e. the derivative of the above mapping with respect to $\mathbf{p}$:

$$\lim_{\Delta p \to 0} \frac{F^{(M)}[\mathbf{r},\mathbf{q};\mathbf{p}+\Delta\mathbf{p}(\mathbf{r}')] - F^{(M)}[\mathbf{r},\mathbf{q};\mathbf{p}]}{\Delta\mathbf{p}(\mathbf{r}')} \qquad [5.5]$$

This property is called the *Jacobian* and is denoted $J_p^{(M)}(\mathbf{r},\mathbf{q};\mathbf{r}')$. The Jacobian depends on three things: the type of forward model, e.g. the diffusion approximation or the Monte Carlo method; the type of measurement M, e.g. time varying intensity or time integrated intensity; and the type of perturbation p, e.g. variation in the absorption coefficient or in the scattering coefficient. In the case of using the diffusion approximation as the forward model and a time resolved measurement, the Jacobian equals the photon hitting density, which is an important result. Thus in a sense it represents a generalization of the photon hitting density. Arridge

calls this *photon-measurement density functions*[36]. These are not restricted to the time domain; they may equally well be derived in the frequency domain, something that makes the concept very powerful.

The significance of the Jacobian can be understood more intuitively. A large value of the Jacobian in a certain point means that a small variation of the optical properties in that point (a 'perturbation') results in a large variation in the detected intensity. The Jacobian thus 'pinpoints' those areas where the optical properties have a great influence on the measurement. This can be seen from the plots of the photon hitting density in the previous chapter. A general feature in all these plots is that the photon hitting density has high values near the source and the detector. Earlier, we interpreted this as an indication of that many photons had traversed those regions. If we instead view the plots as the Jacobian, the interpretation is that if we introduce perturbations in these regions, there will be dramatic changes in the detected intensity. This is intuitively clear: placing an absorbing obstacle just in front of the source, even if it is just a tiny obstacle, will cause a large decrease in the intensity at the detection point. If the small obstacle, on the other hand, is placed far away from the source and the detector, where the photon hitting density is low, the variation of the intensity will be insignificant.

The Jacobian has another interpretation that is especially important from an image reconstruction point of view. Consider a case with a set of source-detector pairs, and the sample is discretized into small volume elements by introducing a grid, then the Jacobian is also discretized and can be written on a matrix form. If the changes in measured intensity are denoted $\Delta\mathbf{I}$ (a vector), we have from the definition of the Jacobian the matrix equation

$$\Delta\mathbf{I} = \mathbf{J}_p^{(M)}\Delta\mathbf{p} \qquad [5.6]$$

which constitutes the linear matrix equation we sought above. The rows in $\mathbf{J}_p^{(M)}$ represent the photon measurement density functions for each source-detector configuration, and the columns represent the effect of a single perturbation in each grid position. Inversion of the matrix equation is then the basis for image reconstruction, because this gives the variation of the optical properties that is necessary when implementing the iterative scheme[39]. In this interpretation the Jacobian is said to be a *kernel of the inverse transformation.*

5.4.1 Inversion

Inversion of the Jacobian can be achieved in a number of ways. The literature on inverse theory is vast, and a thorough description would be outside the scope of this thesis. Inversion imposes two major difficulties that have to be dealt with. Firstly, the matrix is usually very large and the inversion has to be computed quickly to be of practical use in an iteration. Secondly, the matrix is probably ill-conditioned. A commonly used method in medical imaging is the *algebraic reconstruction technique* (ART) and related techniques. Other methods utilize more novel approaches such as the *wavelet transform*[40]. To handle the ill-posed nature of the problem methods such as *linear regularization* (also called *Tikhonov filter*) may be used. An overview of different inversion techniques is given in Ref. 33. See also Ref. 14.

5.4.2 Newton-Raphson Iteration

The perturbation approach may be used in a multi-dimensional Newton-Raphson solver[33,42]. The error norm is then defined, and the zero of its derivative is calculated using a Newton-Raphson algorithm, seeking out the minimum. The Jacobian has to be inverted in each iteration. The Newton-Raphson solver cannot differentiate between local and global minima, so the initial estimate of the optical properties has to be sufficiently close to the real. This may be achieved by some *a priori* knowledge, or possibly by employing a rougher but more stable algorithm, such as the one proposed by Singer *et al.*[34], to serve the Newton-Raphson solver a good starting estimate.

5.4.3 Measurements

In this thesis, the photon hitting density is defined and calculated for a time t in the TPSF, or a small time interval. In the more general context of the photon measurement density functions, other measurements are possible. Integrating the TPSF has the advantage that no time resolved detection equipment is needed, but the benefits of using the information in the early arriving light is then partly lost,

$$M = \int_0^\infty I_{TPSF}(t)dt \qquad [5.7]$$

The mean time of flight in the TPSF is a better choice, but still has the disadvantage that not all of the information in the TPSF is used,

$$M = \frac{\int_0^\infty t \cdot I_{TPSF}(t)dt}{\int_0^\infty I_{TPSF}(t)dt} \qquad [5.8]$$

The purpose of other means of measurement than the photon hitting density is the ease of computation. Computing the photon hitting density for small time steps over the TPSF is perhaps the best approach in theory, but the amount of computation might call for a compromise. Arridge *et al.* [36,41] theoretically investigated the two cases above, integrated TPSF and mean time of flight. Preliminary results show that the mean time of flight is the better choice for detecting deeply imbedded structures in tissue. Also, although integrated TPSF measurements in principle do not require time-resolved detection, there will be problems if the input laser intensity fluctuates so that there are variations in the detected intensity which do not depend on the optical properties of the tissue.

5.4.4 Summary

The main concept of the perturbation approach is a measure of the change of detected intensity due to an infinitesimal perturbation of the optical properties, called the Jacobian. A successful image reconstruction algorithm utilizing this approach should be able to compute

the Jacobian in an efficient way as well as inverting it. Arridge *et al.* used a FEM model to perform the computation for arbitrary cases[41,42].

5.5 Future Research

NIR image reconstruction by using a perturbation approach seems to have a great potential. Many problems need to be solved, however, before optical tomography can be used as a clinical diagnostic tool. A computationally efficient forward model must be developed. Closely connected to the forward model is the type of measurements – the geometries of the source-detector configurations, whether to perform time- or frequency-domain measurements, how much information to use from the acquired data. These may all have significant influence on the efficiency of the algorithm. The fundamental ill-posed nature of the reconstruction problem has to be dealt with both theoretically, within the framework of inversion theory, as well as practically, to obtain as noise-free signals as possible. Inversion itself is beside the forward model the computationally most demanding step in the process, requiring a fast method, which is perhaps yet to be developed, if none of the presently available schemes proves to be satisfactory. Inversion is in turn the consequence of error minimization in an iterative algorithm, where fast convergence is imperative. Optimization theory here comes in as an important topic.

Obviously, all of these theoretical issues must be closely linked to the physical reality during the development process, such as the performance of the technical equipment, the properties of the tissue being investigated, and the necessity of implementing the method so that it easily can be used in a clinical environment.

6. References

[1]M. L Giger, C.A Pelizzari, "Advances in tumor imaging", Sci. Am. September 1996, 76-78 (1996)

[2]S. Andersson-Engels, R Berg, O. Jarlman, S. Svanberg, "Time-resolved transillumination for medical diagnostics", Opt. Lett. **15**, 1179-1181 (1990)

[3]R. Berg, O. Jarlman, S. Svanberg, "Medical transillumination imaging using short-pulse diode lasers", Appl. Opt. **32**, 574-579 (1993)

[4]F. Liu, K. M. Yoo, R. R. Alfano, "Ultrafast laser-pulse transmission and imaging through biological tissues", Appl. Opt. **32**, 554-558 (1993)

[5]L. G. Henyey, J. L. Greenstein, "Diffuse radiation in the galaxy", Astrophysics Journal **93**, 70-83, (1943)

[6]R. Berg, "Laser-based cancer diagnostics and therapy - tissue optics considerations", Ph.D. thesis, LRAP-184, (Lund Institute of Technology, Sweden, 1995)

[7]K. M. Case, P. F. Zweifel, *Linear Transport Theory* (Addison-Wesley Publishing Co., 1967)

[8]F. Liu, K. M. Yoo, R. R. Alfano, "Should the photon flux or the photon density be used to describe the temporal profiles of scattered ultrashort laser pulses in random media?", Opt. Lett. **18**, 432-434 (1993)

[9]E. Hecht, *Optics*, (Addison-Wesley Publishing Co., 1987)

[10]R. C. Haskell, L. O. Svaasand, T.-T. Tsay, T.-C. Feng, M. S. McAdams, B. J. Tromberg, "Boundary conditions for the diffusion equation in radiative transfer", J. Opt. Soc. Am. A **11**, 2727-2741 (1994)

[11]S. R. Arridge, M. Schweiger, M. Hiraoka, D. T. Delpy, "A finite-element approach for modeling photon transport in tissue", Med. Phys. **20**, 299-309 (1993)

[12]J. C. Schotland, J. C. Haselgrove, J. S. Leigh, "Photon hitting density", Appl. Opt. **32**, 448-453 (1993)

[13] M. S. Patterson, S. Andersson-Engels, B. C. Wilson, E. K. Osei, "Absorption spectroscopy in tissue-simulating materials: a theoretical and experimental study of photon paths", Appl. Opt. **34**, 22-30 (1995)

[14]W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, *Numerical Recipes: The Art of Scientific Computing*, (Cambridge University Press, 1992), available in the C, Fortran, Pascal and Basic programming languages.

[15]H. R. Gordon, "Equivalence of the point- and beam-spread functions of scattering media: a formal demonstration", Appl. Opt. **33**, 1120-1122 (1994)

[16]L. Wang, S. L. Jacques, "Monte Carlo modeling of light transport in multi-layered tissues in standard C", (Laser Biology Research Laboratory, M. D. Andersson Cancer Center, Univ. of Texas, Houston, Tx., 1992)

[17] S. Andersson-Engels, R. Berg, S. Svanberg, "Effects of optical constants on time-gated transillumination of tissue and tissue-like media", J. Photochem. Photobiol. B: Biol., **16**, 155-167 (1992)

[18]A. Pifferi, R. Berg, P. Taroni, S. Andersson-Engels, "Fitting of time-resolved reflectance curves with a Monte Carlo model", in *Advances in Optical Imaging and Photon Migration,* 1996 Technical Digest (Opt. Soc. of Am., Wash. DC, 1996), pp. 128-130

[19]R. Graff, M. H. Koelink, F. F. de Mul, W. G. Zijlstra, A. C. M. Dassel, J. G. Arnoudse, "Condensed Monte Carlo simulation for the description of light transport", Appl. Opt. **32**, 426-434 (1993)

[20]L. Wang, S. L. Jacques, "Hybrid model of Monte Carlo simulation and diffusion theory for light reflectance by turbid media", J. Opt. Soc. Am. **10**, 1746-1752 (1993)

[21]A. Ya. Polischuk, R. R. Alfano, "Fermat photons in turbid media: an exact analytic solution for most favorable paths - a step toward optical tomography", Opt. Lett. **20**, 1937 (1995)

[22]L. T. Perelman, "Time-dependent photon migration using path integrals", Phys. Rev. **51**, 6134 (1995)

[23]J. H. Radon, "Über die Bestimmung von Funktionen durch ihre Integralwärte längs gewisser Mannigfaltigkeiten", Ber. vor. Sächs. Akad. Wiss. **69**, 262 (1917)

[24]W. A. Kalender, "X-ray computed tomography - state of the art", in *Medical Optical Tomography: Functional Imaging and Monitoring,* vol. 11 of SPIE Institute Series, G. Müller *et al.* eds. (SPIE Press, Bellingham, Wash. DC, 1993), pp. 10-27

[25]A. J. Devaney, "A computer simulation study of diffraction tomography", IEEE Trans. on Biomed. Eng. BME-**30**, 377-86 (1983)

[26] N. Sponheim, L. J. Gelius, I. Johansen, J. J. Stamnes, "Ultrasonic tomography of biological tissue", Ultrason. Imaging **16**, 19-32 (1994)

[27]J. G. Webster, ed., *Electrical Impedance Tomography,* (Adam Hilger, Bristol, UK, 1990)

[28]I. Oda, Y. Ito, H. Eda, T. Tamura, M. Takada, R. Abumi, K. Nagai, H. Nakagawa, M. Tamura, "Non-invasive hemoglobin oxygenation monitor and computed tomography by NIR spectroscopy", SPIE vol. **1431**, 284-293 (1991)

[29]J. C. Hebden, K. S. Wang, "Time-resolved optical tomography", Appl. Opt. **32**, 372-380 (1993)

[30]S. B. Colak, H. Schomberg, "Optical back-projection tomography in heterogeneous diffusive media", in *Advances in Optical Imaging and Photon Migration*, 1996 Technical Digest (Opt. Soc. Am., Wash. DC, 1996), pp. 147-149

[31]Y. Yamada, Y. Hasegawa, Y. Yamashita, "Simulation of fan-beam-type optical computed tomography imaging of strongly scattering and weakly absorbing media", Appl. Opt. **32**, 4808-4814 (1993)

[32]H. Eda, I. Oda, Y. Ito, Y. Wada, Y. Tsunazawa, M. Takada, "Image reconstruction of phantom with plural absorbing rods in optical CT", in *Advances in Optical Imaging and Photon Migration*, 1996 Technical Digest (Opt. Soc. Am., Wash. DC, 1996), pp. 150-152

[33]S. R. Arridge, "The forward and inverse problems in time resolved infra-red imaging", in *Medical Optical Tomography: Functional Imaging and Monitoring*, vol. **11** of SPIE Institute Series, G. Müller *et al.* eds. (SPIE Press, Bellingham, Wash. 1993), pp. 35-64

[34]J. R. Singer, F. A. Grünbaum, P. Kohn, J. P. Zubelli, "Image reconstruction of the interior of bodies that diffuse radiation", Science **248**, 990-993 (1990)

[35]S. R. Arridge, M. Schweiger, M. Hiraoka, D. T. Delpy, "Performance of an iterative reconstruction algorithm for near infrared absorption and scatter imaging", in *Proc. Photon Migration and Imaging in Random Media nad Tissues*, B. Chance, R. R. Alfano, A. Katzir eds., SPIE vol. **1888** (1993) pp. 360-371

[36]S. R. Arridge, "Photon-measurement functions. Part 1: Analytical forms", Appl. Opt. **34**, 7395-7409 (1995)

[37]R. L. Barbour, H. L. Graber, "A perturbation approach for optical diffusion tomography using continous-wave and time-resolved data", in *Medical Optical Tomography: Functional Imaging and Monitoring*, vol. **11** of SPIE Institute Series, G. Müller *et al.* ed. (SPIE Press, Bellingham, Wash. DC, 1993), pp. 87-143.

[38]S. Feng, F. Zeng, B. Chance, "Photon migration in the prescence of a single defect: a perturbation analysis", Appl. Opt. **34**, 3826-3837 (1995)

[39]S. R. Arridge, P. van der Zee, M. Cope, D. T. Delpy, "Reconstruction methods for infrared absorption imaging", in *Time-Resolved Spectroscopy and Imaging in Tissues*, B. Chance, A. Katzir eds., SPIE vol. **1431** (1991) pp. 204-215

[40]W. Zhu, Y. Wang, Y. Yao, R. L. Barbour, "Wavelet based multigrid reconstruction algorithm for optical tomography", in *Advances in Optical Imaging and Photon Migration*, 1996 Technical Digest (Opt. Soc. Am., Wash. DC, 1996), pp. 143-146.

[41]S. R. Arridge, M. Schweiger, "Photon-measurement functions. Part 2: Finite-element calculations", Appl. Opt. **34**, 8026-8037 (1995)

[42]M. Schweiger, S. R. Arridge, "A system for solving the forward and inverse problems in optical spectroscopy and imaging", in *Advances in Optical Imaging and Photon Migration*, 1996 Technical Digest (Opt. Soc. Am., Wash. DC, 1996) pp. 137-139

Appendix A: Diffusion Approximation Computer Code

Three short pieces of computer code were designed for the diffusion approximation calculations. The integration is carried out using a Gauss-Legendre scheme. The three programs differ mainly in the routines `fluence` and `escape`.

A.1 PROGRAM CODE SEMI

Computes the photon hitting density as sampled in time steps for a semi-infinte half-space geometry.

A.1.1 semi.cpp

```
/*          Semi: this code calculates the number of photons that has passed a small
            volume element at (r,z) under the following conditions:
            i) the geometry of the model sample is a semi-infinte space given by z>0
            ii) the photons are introduced as a point source (Dirac function in both
                        space and time) at z=z0 and t=0 (where z0 is the mean free path)
            iii) the photons are detected by a detector at r=x0, z=0 during a time interval
                        t1<t<t2

            The code utilizes a formula based on the diffusion equation with image
            sources introduced to create an infinte space to perform the calculation in.
            The numerical integration is done using a 10-point Gauss-Legendre algorithm.*/

#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>

void   calculate(FILE *);
FILE   *OpenInFile(void);
            // I/O: Data from Slab.in
FILE   *OpenOutFile(char *,char *,char *);            // I/O: Data to [name].TXT
inline double sqr(double);
int    initialize(void);
inline double fluence(double,double,double);
inline double escape(double,double,double);
double integrate(double,double,double,double);
inline double f(double,double,double,double);
inline double conv(double,double,double);

double pi = 3.1415926536;
double c0 = 3e8;                    // Speed of light in vaccuum, (m/s)
double mys,mya,n;                   // Opt. prop. of the sample   (1/m)
double dz,dr0;              // Depth in sample; dist. to det.  (mm)
double dr;                  // Border of calculation on r-axis (mm)
double x0;                  // Distance to detector  (m)
double start_time;          // Integration time parrameter (ns)
double time_step,stop_time;     // Integration time parameters (ns)
double c;                   // Speed of light in sample (m/s)
double ze, z00, zp0, diff; // Extrapol. bound., 1st pos. s., 1st neg. s.,Diff.koeff
double gamma, eta;          // Const. in exp-fctn, Const. in fluence/escape
double r2;                  // Sqr(distance from z-axis of vol. element)
double abscissa[5],weight[5];   // Used in Gauss-Legendre integr.
double tau1,tau2;           // Const. times in conv. integral limits

void main(void) {

            int ok;
            FILE *out;
            char *Name;
```

```
                ok=initialize();
                if (ok==0) {
                        out=OpenOutFile (Name,".TXT","wt");
                        if (out != NULL) {

                fprintf(out,"mys:\t%f\tmya:\t%f\tn:\t%f\n",mys,mya,n);
                                fprintf(out,"dz:\t%f\tdr0:\t%f\n",dz,dr0);
                                fprintf(out,"ze:\t%f\n\n",ze);
                                calculate(out);
                                fclose(out);
                        }
                }
}


void calculate(FILE *out) {

                double time,t1,delta,r1,z1,sqrz;
                double photons;
                int r,z;

                delta=time_step*1e-9;
                for (time=start_time;time<stop_time;time+=time_step) {
                        t1=time*1e-9;
                        printf("Time:\t%fns\t\tStop time:\t%fns\n",time,stop_time);

                fprintf(out,"Time:\t%f\tns\tto:\t%f\tns\n\n",time,(time+time_step));
                        for (r=-dr;r<=(dr0+dr);r++)             // From -dr to dr0+dr
                                fprintf(out,"\t%d",r);
                        fprintf(out,"\n");
                        for (z=0;z<=dz;z++) {       // Here's where the main loop begins
                                z1=z*0.001;
                                sqrz=sqr(z1);
                                fprintf(out,"%d\t",z);
                                for (r=-dr;r<=(dr0+dr);r++) {
                                        r1=r*0.001;
                                        r2=sqr(r1);
                                        tau1=sqrt(r2+sqrz)/c;
                        // From (0,0,0) to (r,z)
                                        tau2=sqrt(sqr(r1-x0)+sqrz)/c;
                        // From (r,z) to (x0,0)
                                        photons=integrate(t1,t1+delta,r1,z1);
                                        fprintf(out,"%e\t",photons);
                                }
                                fprintf(out,"\n");
                        }
                        fprintf(out,"\n");
                }
                fprintf(out,"\nTime\tPhoton fluence at detector\n\n"); // Photon fluence curve
                for (time=start_time;time<stop_time;time+=time_step) {
                        t1=time*1e-9;
                        fprintf(out,"%f\t%f\n",time,fluence(x0,0,t1));
                }
}


FILE *OpenInFile(void){

                FILE *fptr;

                if ((fptr = fopen("semi.in","rt")) == NULL) {
                        printf("\n\nError number : %d\n", errno);
                        perror("Could not open file: semi.in");
                }
                return(fptr);
}


FILE *OpenOutFile(char *Name,char *Extention,char *Type) {
```

```
        FILE *fptr;
        char FileName[40],Ext[10],*answer;
        char *pos;
        int no;

        strcpy(Ext,Extention);
        printf("\nEnter name of output file (q = quit): ");
        gets(FileName);
        if ((strlwr(FileName)[0]=='q')&&(FileName[1]=='\0')){
                printf("\n\nProgram aborted!");
                return(NULL);
        }
        pos = strchr(FileName,'.');
        if (pos == NULL){
                strcpy(Name,FileName);
                strcat(FileName,Ext);
        } else {
                no = pos - FileName;
                strncpy(Name, FileName, no);
                Name[no] = '\0';
                strcpy(Ext,pos);
        }
        strcpy(FileName,Name);
        strcat(FileName,Extention);
        fptr = fopen(FileName,"rt");
        if (fptr != NULL) {
                printf ("\n\nA file called %s already exists.",FileName);
                printf ("\n\tOverwrite ? (n/*) : ");
                gets(answer);
                if (strlwr(answer)[0] == 'n')
                        return(NULL);
                fptr = freopen(FileName, Type, fptr);
        } else
                fptr = fopen(FileName, Type);
        return(fptr);
}


inline double sqr(double x){
        return(x*x);
}


int initialize(void) {

        double  myc, r0, kappa;
        int i,ok;
        char line[50];
        FILE *fptr;

        ok = -1;
        fptr = OpenInFile();
        if (fptr != NULL) {
                for (i=1; i<=2; i++);
                        fgets(line,49,fptr);
                fscanf(fptr,"%s%lf",line,&mys);  // Reading data from Semi.in
                fscanf(fptr,"%s%lf",line,&mya);
                fscanf(fptr,"%s%lf",line,&n);
                fscanf(fptr,"%s%lf",line,&dz);
                fscanf(fptr,"%s%lf",line,&dr);
                fscanf(fptr,"%s%lf",line,&dr0);
                fgets(line,49,fptr);
                fscanf(fptr,"%s%lf",line,&start_time);
                fscanf(fptr,"%s%lf",line,&time_step);
                fscanf(fptr,"%s%lf",line,&stop_time);
                fclose(fptr);
                x0    = 0.001*dr0;
                // Source/det. distance from edge (m)
                diff  = 1/(3*(mys+mya));    // Diffusion coefficient    (m)
```

```
                        c     = c0/n;
                        // Speed of light in sample,  (m/s)
                        myc   = cos(asin(c/c0));
                        r0    = sqr(((c0-c)/(c0+c)));
                        kappa = ((1-r0)*(1-myc*myc))/(1+r0+(1-r0)*myc*myc*myc);
                        ze    = 2*diff/kappa;       // Extrapolated boundary, (m)
                        z00   = 1/mys;              // The position of the source,(m)
                        zp0   = -(2*ze + z00);      // Image source, z-coord      (m)
                        gamma = 4*diff*c; // Constant in exp-terms in fluence/escape
                        eta   = pow (gamma*pi,-1.5); // Constant in fluence/escape
                        abscissa[0]=0.1488743389;
                        abscissa[1]=0.4333953941;
                        abscissa[2]=0.6794095682;
                        abscissa[3]=0.8650633666;
                        abscissa[4]=0.9739065285;
                        weight[0]=0.2955242247;
                        weight[1]=0.2692667193;
                        weight[2]=0.2190863625;
                        weight[3]=0.1494513491;
                        weight[4]=0.0666713443;
                        ok    = 0;
                }
                return(ok);
}

// Photon-fluence: in (r,z) at time t due to point sources in a semi space geometry

inline double fluence (double r,double z,double t) {

                        double ro2,rp2;
                        // Distances between (r,z) and
                        double sum,dummy;
                // pos. source for r=0; neg. source

                                                // for r=0
                        if (t>0.0) {
                                        dummy = gamma*t;
                                        ro2 = r2+sqr(z00-z);
                                        rp2 = r2+sqr(zp0-z);
                                        sum = exp(-ro2/dummy) - exp(-rp2/dummy);
                                        sum *= c*eta*pow(t,-1.5)*exp(-mya*c*t);
                        }
                        return(sum);
}

// Escape-fuction: probability of a photon in (r,z) reaching detector at time t

inline double escape(double r,double z,double t) {

                        double k2i,l2i;   // Distances between det. (r0,0) and
                        double sum,dummy; // pos. vol. elem. source for r; neg. source
                                                // for r
                        if (t>0.0) {
                                        dummy = gamma*t;
                                        k2i = sqr(r-x0)+sqr(z);
                                        l2i = sqr(r-x0)+sqr(z+2*ze);
                                        sum = z*exp(-k2i/dummy) + (z+2*ze)*exp(-l2i/dummy);
                                        sum *= 0.5*eta*pow(t,-2.5)*exp(-mya*c*t);
                        }
                        return(sum);
}

double integrate(double a,double b, double r,double z) { // Outer (time interval)
                                                // integral
                double ss,tm,tr,dt;
                int j;
```

```
            ss = 0.0;
            // 10 point Gauss-Legendre integration
            tm = 0.5*(b+a);
            tr = 0.5*(b-a);
            for (j=0;j<=4;j++) {        // Symmetric in interval => 5 iterations
                        dt = tr*abscissa[j];
                        ss += weight[j]*(conv(r,z,tm+dt)+conv(r,z,tm-dt));
            }
            return(ss*tr);
            // tr = scaling factor
}

inline double conv(double r,double z,double t) {        // Inner (convolution)
                                                        // integral
            double ss,tm,tr,dt,a,b;
            int j;

            a  = tau1;
            b  = t-tau2;
            ss = 0.0;
            if (a<b) {
                        tm = 0.5*(b+a);
                        tr = 0.5*(b-a);
                        for (j=0;j<=4;j++) {
                                    dt = tr*abscissa[j];
                                    ss += weight[j]*(f(r,z,tm+dt,t)+f(r,z,tm-dt,t));
                        }
            }
            return(ss*tr);
}

inline double f(double r,double z,double tprim,double t) { // Integrand

            return(fluence(r,z,tprim)*escape(r,z,t-tprim));
}
```

A.1.2 Template Input File for `Semi`

```
Data to calculate light fluence in a slab

Scat_coeff(1/m):   5000.0
Abs_coeff_(1/m):   3.0
Refractiveindex:   1.4
Thick.___z_(mm):   20
Radius___x_(mm):   5
Det._dist_x(mm):   10

Start_time_(ns):   0.0
Time_step__(ns):   0.2
Stop_time__(ns):   7.0
```

A.2 PROGRAM CODE SLAB

Computes the photon hitting density as sampled in time steps for a slab geometry.

A.2.1 slab.cpp

```
/*          Slab: this code calculates the number of photons that has passed a small
            volume element at (r,z) (cyl.coord.) under the following conditions:
            i) the geometry of the model sample is an infinte slab given by 0<z<d
            ii) the photons are introduced as a point source (Dirac function in both
                        space and time) at z=z0 and t=0 (where z0 is the mean free path)
            iii) the photons are detected by a detector at z=d during a time interval
                        t1<t<t2

            The code utilizes a formula based on the diffusion equation with image
            sources introduced to create an infinte space to perform the calculation in.
            The numerical integration is done using  a 10-point Gauss-Legendre algorithm.
*/


#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>

void   calculate(FILE *);
FILE   *OpenInFile(void);
            // I/O
FILE   *OpenOutFile(char *,char *,char *);                          // I/O
inline double sqr(double);
int    initialize(void);
inline double z0(int);                                  // Position of pos. sources
inline double zp(int);                                  // Position of neg. sources
inline double q0(int,double);           // Position of pos. vol. element sources
inline double qp(int,double);           // Position of neg. vol. element sources
inline double fluence(double,double,double);
inline double escape(double,double,double);
double integrate(double,double,double,double);
inline double f(double,double,double,double);
inline double conv(double,double,double);

double pi = 3.1415926536;
double c0 = 3e8;                // Speed of light in vaccuum, (m/s)
double mys,mya,n;               // Opt. prop. of the sample   (1/m)
double dz,dr;                   // Borders of the sample       (mm)
double timestep,stoptime;       // Integration time parameters (ns)
double c,d;                 // Speed of light in medium (m/s), Thickness in mm
double ze, z00, zp0, diff;  // Extrapol. bound., 1st pos. s., 1st neg. s.,Diff.co.
double gamma, eta;          // Const. in exp-fctn, Const. in fluence/escape
double r2;                  // Sqr(distance from z-axis of vol. element)
double abscissa[5],weight[5];   // Used in Gauss-Legendre integr.
double tau1,tau2;           // Const. times in conv. integral limits

void main(void) {

            int ok;
            FILE *out;
            char *Name;

            ok=initialize();
            if (ok==0) {
                        out=OpenOutFile (Name,".TXT","wt");
                        if (out != NULL) {

            fprintf(out,"mys:\t%f\tmya:\t%f\tn:\t%f\n",mys,mya,n);
                                    fprintf(out,"dz:\t%f\tdr:\t%f\n",dz,dr);
```

```
                                        fprintf(out,"ze:\t%f\n\n",ze);
                                        calculate(out);
                                        fclose(out);
                                }
                        }
        }

        void calculate(FILE *out) {

                        double time,t1,delta,r1,z1,sqrz;
                        double photons;
                        int r,z;

                        delta=timestep*1e-9;
                        fprintf(out,"Detected intensity\n\n");
                        for (time=0;time<stoptime;time+=timestep)
                                        fprintf(out,"%f\t%f\n",time, fluence(0,d,time*1e-9));
                        fprintf(out,"\n");
                        for (time=0;time<stoptime;time+=timestep) {
                                        t1=time*1e-9;
                                        printf("Time:\t%fns\t\tStop time:\t%fns\n",time,stoptime);

                        fprintf(out,"Time:\t%f\tns\tto:\t%f\tns\n\n",time,(time+timestep));
                                        for (r=0;r<=dr;r++)
                                                        fprintf(out,"\t%d",r);
                                        fprintf(out,"\n");
                                        for (z=0;z<=dz;z++) {       // Here's where the main loop begins
                                                        z1=z*0.001;
                                                        sqrz=sqr(z1-z00);
                                                        fprintf(out,"%d\t",z);
                                                        for (r=0;r<=dr;r++) {
                                                                        r1=r*0.001;
                                                                        r2=sqr(r1);
                                                                        tau1=sqrt(r2+sqrz)/c;
                                        // From (0,0,z00) to (x,y,z)
                                                                        tau2=sqrt(r2+sqr(d-z1))/c;
                        // From (x,y,z) to (0,0,d)
                                                                        photons=integrate(t1,t1+delta,r1,z1);
                                                                        fprintf(out,"%e\t",photons);
                                                        }
                                                        fprintf(out,"\n");
                                        }
                                        fprintf(out,"\n");
                        }
        }


        FILE *OpenInFile(void){

                        FILE *fptr;

                        if ((fptr = fopen("slab.in","rt")) == NULL) {
                                        printf("\n\nError number : %d\n", errno);
                                        perror("Could not open file: slab.in");
                        }
                        return(fptr);
        }


        FILE *OpenOutFile(char *Name,char *Extention,char *Type) {

                        FILE *fptr;
                        char FileName[40],Ext[10],*answer;
                        char *pos;
                        int no;

                        strcpy(Ext,Extention);
                        printf("\nEnter name of output file (q = quit): ");
                        gets(FileName);
                        if ((strlwr(FileName)[0]=='q')&&(FileName[1]=='\0')){
```

```
                printf("\n\nProgram aborted!");
                return(NULL);
        }
        pos = strchr(FileName,'.');
        if (pos == NULL){
                strcpy(Name,FileName);
                strcat(FileName,Ext);
        } else {
                no = pos - FileName;
                strncpy(Name, FileName, no);
                Name[no] = '\0';
                strcpy(Ext,pos);
        }
        strcpy(FileName,Name);
        strcat(FileName,Extention);
        fptr = fopen(FileName,"rt");
        if (fptr != NULL) {
                printf ("\n\nA file called %s already exists.",FileName);
                printf ("\n\tOverwrite ? (n/*) : ");
                gets(answer);
                if (strlwr(answer)[0] == 'n')
                        return(NULL);
                fptr = freopen(FileName, Type, fptr);
        } else
                fptr = fopen(FileName, Type);
        return(fptr);
}


inline double sqr(double x){
        return(x*x);
}


int initialize(void) {

        double  myc, r0, kappa;
        int i,ok;
        char line[50];
        FILE *fptr;

        ok = -1;
        fptr = OpenInFile();
        if (fptr != NULL) {
                for (i=1; i<=2; i++);
                        fgets(line,49,fptr);
                fscanf(fptr,"%s%lf",line,&mys);// Reading data from Slab.in
                fscanf(fptr,"%s%lf",line,&mya);
                fscanf(fptr,"%s%lf",line,&n);
                fscanf(fptr,"%s%lf",line,&dz);
                fscanf(fptr,"%s%lf",line,&dr);
                fgets(line,49,fptr);
                fscanf(fptr,"%s%lf",line,&timestep);
                fscanf(fptr,"%s%lf",line,&stoptime);
                fclose(fptr);
                d     = 0.001*dz;
                diff  = 1/(3*(mys+mya));          // Diffusion coefficient  (m)
                c     = c0/n;            // Speed of light in sample,  (m/s)
                myc   = cos(asin(c/c0));
                r0    = sqr(((c0-c)/(c0+c)));
                kappa = ((1-r0)*(1-myc*myc))/(1+r0+(1-r0)*myc*myc*myc);
                ze    = 2*diff/kappa;       // Extrapolated boundary, (m)
                z00   = 1/mys;              // The position of the source, (m)
                zp0   = -(2*ze + z00);
        // Image source, z-coord       (m)
                gamma = 4*diff*c;
                eta   = pow (gamma*pi,-1.5);
                abscissa[0]=0.1488743389;
                abscissa[1]=0.4333953941;
                abscissa[2]=0.6794095682;
```

```
                        abscissa[3]=0.8650633666;
                        abscissa[4]=0.9739065285;
                        weight[0]=0.2955242247;
                        weight[1]=0.2692667193;
                        weight[2]=0.2190863625;
                        weight[3]=0.1494513491;
                        weight[4]=0.0666713443;
                        ok     = 0;
                }
                return(ok);
}


inline double z0(int m) {               // Position of the positive part
                return(2*m*(d+2*ze)+z00); // of the source dipole
}


inline double zp(int m) {               // Position of the negative part
                return(2*m*(d+2*ze)+zp0); // of the source dipole
}


inline double q0(int m,double z) {      // Position of the positive part
                return(2*m*(d+2*ze)+z);   // of the volume element dipole
}


inline double qp(int m,double z) {                  // Position of the negative part
                return(2*m*d+2*(2*m-1)*ze-z);       // of the volume element dipole
}


inline double fluence (double r,double z,double t) {

                        int i;
                        double ro2i,rp2i;
                        double sum,dummy;

                        sum = 0.0;
                        if (t>0.0) {
                                        dummy = gamma*t;
                                        for (i=-2; i<=2; i++) {
                // i = order of image dipole
                                                        ro2i = sqr(r)+sqr(z0(i)-z);
                                                        rp2i = sqr(r)+sqr(zp(i)-z);
                                                        sum += exp(-ro2i/dummy) - exp(-
rp2i/dummy);
                                        }
                                        sum *= c*eta*pow(t,-1.5)*exp(-mya*c*t);
                        }
                        return(sum);
}


inline double escape(double r,double z,double t) {

                        int i;
                        double ro2i,rp2i;
                        double sum,dummy;

                        sum = 0.0;
                        if (t>0.0) {
                                        dummy = gamma*t;
                                        for (i=-2; i<=2; i++) {
                // i = order of image dipole
                                                        ro2i = sqr(r)+sqr(q0(i,z)-d);
                                                        rp2i = sqr(r)+sqr(qp(i,z)-d);
                                                        sum += exp(-ro2i/dummy) - exp(-
rp2i/dummy);
```

```
                                        }
                                        sum *= eta*pow(t,-1.5)*exp(-mya*c*t);
                                }
                                return(sum);
}


double integrate(double a,double b, double r,double z) { // Outer integral

        double ss,tm,tr,dt;
        int j;

        ss = 0.0;
        // 10 point Gauss-Legendre integration
        tm = 0.5*(b+a);
        tr = 0.5*(b-a);
        for (j=0;j<=4;j++) {
                        dt = tr*abscissa[j];
                        ss += weight[j]*(conv(r,z,tm+dt)+conv(r,z,tm-dt));
        }
        return(ss*tr);
}

inline double conv(double r,double z,double t) {       // Inner (conv.) integral

        double ss,tm,tr,dt,a,b;
        int j;

        a  = tau1;
        b  = t-tau2;
        ss = 0.0;
        if (a<b) {
                        tm = 0.5*(b+a);
                        tr = 0.5*(b-a);
                        for (j=0;j<=4;j++) {
                                        dt = tr*abscissa[j];
                                        ss += weight[j]*(f(r,z,tm+dt,t)+f(r,z,tm-dt,t));
                        }
        }
        return(ss*tr);
}

inline double f(double r,double z,double tprim,double t) { // Integrand

        return(fluence(r,z,tprim)*escape(r,z,t-tprim));
}
```

A.2.2 Template Input File for `Slab`

```
Data to calculate light fluence in a slab

Scat_coeff(1/m):   4000.0
Abs_coeff_(1/m):   50.0
Refractiveindex:   1.4
Thick.___z_(mm):   15
Radius___r_(mm):   15

Time_step__(ns):   0.05
Stop_time__(ns):   2.0
```

A.3 PROGRAM CODE SEMISLAB

Computes the photon hitting density as sampled in time steps for a semi-infinte slab geometry.

A.3.1 semislab.cpp

```
/*          SemiSlab: this code calculates the number of photons that has passed a small
            volume element at (x,yc,z) under the following conditions:
            i) the geometry of the model sample is a semi-infinte slab given by 0<z<d,
                        x<x0
            ii) the photons are introduced as a point source (Dirac function in both
                        space and time) at z=z0, x=0 and t=0 (where z0 is the mean free
path)
            iii) the photons are detected by a detector at z=d, x=0 during a time interval
                        t1<t<t2

            The code utilizes a formula based on the diffusion equation with image
            sources introduced to create an infinte space to perform the calculation in.
            The numerical integration is done using a 10-point Gauss-Legendre algorithm.*/

#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>

void   calculate(FILE *);
FILE   *OpenInFile(void);
            // I/O: Data from Slab.in
FILE   *OpenOutFile(char *,char *,char *);                          // I/O: Data to
[name].TXT
inline double sqr(double);
int    initialize(void);
inline double z0(int);                                  // z-position of pos. sources
inline double zp(int);                                  // z-position of neg. sources
inline double q0(int,double);           // z-position of pos. vol. element sources
inline double qp(int,double);           // z-position of neg. vol. element sources
inline double fluence(double,double,double);
inline double escape(double,double,double);
double integrate(double,double,double,double);
inline double f(double,double,double,double);
inline double conv(double,double,double);

double pi = 3.1415926536;
double c0 = 3e8;                // Speed of light in vaccuum, (m/s)
double mys,mya,n;               // Opt. prop. of the sample   (1/m)
double dz,dx0,dyc;              // Borders of the sample, Plane of calc.  (mm)
double dx;                  // Border of calculation on x-axis (mm)
double x0,yc;               // x-border, Plane of calculation y=yc   (m)
double start_time;          // Integration time parameter (ns)
double time_step,stop_time;       // Integration time parameters (ns)
double c,d;                 // Speed of light in sample (m/s), Thickness (m)
double ze, z00, zp0, diff; // Extrapol. bound., 1st pos. s., 1st neg. s.,Diff.koeff
double x_image;             // Position of image sources in x-direction
double gamma, eta;          // Const. in exp-fctn, Const. in fluence/escape
double r2;                  // Sqr(distance from z-axis of vol. element)
double abscissa[5],weight[5];   // Used in Gauss-Legendre integr.
double tau1,tau2;           // Const. times in conv. integral limits

void main(void) {

            int ok;
            FILE *out;
            char *Name;

            ok=initialize();
            if (ok==0) {
```

```
                    out=OpenOutFile (Name,".TXT","wt");
                    if (out != NULL) {

        fprintf(out,"mys:\t%f\tmya:\t%f\tn:\t%f\n",mys,mya,n);
                                fprintf(out,"dz:\t%f\tdx0:\t%f\n",dz,dx0);
                                fprintf(out,"ze:\t%f\tx_image:\t%f\n\n",ze,x_image);
                                calculate(out);
                                fclose(out);
                    }
        }
}


void calculate(FILE *out) {

        double time,t1,delta,x1,z1,sqrz;
        double photons;
        int x,z;

        delta=time_step*1e-9;
        fprintf(out,"Detected intensity\n\n");
        for (time=0;time<stop_time;time+=time_step)
                fprintf(out,"%f\t%f\n",time, fluence(0,d,time*1e-9));
        fprintf(out,"\n");
        for (time=start_time;time<stop_time;time+=time_step) {
                t1=time*1e-9;
                printf("Time:\t%fns\t\tStop time:\t%fns\n",time,stop_time);

        fprintf(out,"Time:\t%f\tns\tto:\t%f\tns\n\n",time,(time+time_step));
                for (x=-dx;x<=dx0;x++)                  // From -dx to the edge
of the sample
                                fprintf(out,"\t%d",x);
                fprintf(out,"\n");
                for (z=0;z<=dz;z++) {                   // Here's where the main
loop begins
                                z1=z*0.001;
                                sqrz=sqr(z1-z00);
                                fprintf(out,"%d\t",z);
                                for (x=-dx;x<=dx0;x++) {
                                        x1=x*0.001;
                                        r2=sqr(x1)+sqr(yc);
                                        tau1=sqrt(r2+sqrz)/c;
                // From (0,0,0) to (x,y,z)
                                        tau2=sqrt(r2+sqr(d-z1))/c;
        // From (x,y,z) to (0,0,d)
                                        photons=integrate(t1,t1+delta,x1,z1);
                                        fprintf(out,"%e\t",photons);
                                }
                                fprintf(out,"\n");
                }
                fprintf(out,"\n");
        }
        fprintf(out,"\nTime\tPhoton fluence at detector\n\n"); // Photon fluence curve
        for (time=start_time;time<stop_time;time+=time_step) {
                t1=time*1e-9;
                fprintf(out,"%f\t%f\n",time,fluence(0,d,t1));
        }
}


FILE *OpenInFile(void){

        FILE *fptr;

        if ((fptr = fopen("semislab.in","rt")) == NULL) {
                printf("\n\nError number : %d\n", errno);
                perror("Could not open file: semislab.in");
        }
        return(fptr);
}
```

```
FILE *OpenOutFile(char *Name,char *Extention,char *Type) {

        FILE *fptr;
        char FileName[40],Ext[10],*answer;
        char *pos;
        int no;

        strcpy(Ext,Extention);
        printf("\nEnter name of output file (q = quit): ");
        gets(FileName);
        if ((strlwr(FileName)[0]=='q')&&(FileName[1]=='\0')){
                printf("\n\nProgram aborted!");
                return(NULL);
        }
        pos = strchr(FileName,'.');
        if (pos == NULL){
                strcpy(Name,FileName);
                strcat(FileName,Ext);
        } else {
                no = pos - FileName;
                strncpy(Name, FileName, no);
                Name[no] = '\0';
                strcpy(Ext,pos);
        }
        strcpy(FileName,Name);
        strcat(FileName,Extention);
        fptr = fopen(FileName,"rt");
        if (fptr != NULL) {
                printf ("\n\nA file called %s already exists.",FileName);
                printf ("\n\tOverwrite ? (n/*) : ");
                gets(answer);
                if (strlwr(answer)[0] == 'n')
                        return(NULL);
                fptr = freopen(FileName, Type, fptr);
        } else
                fptr = fopen(FileName, Type);
        return(fptr);
}


inline double sqr(double x){
        return(x*x);
}


int initialize(void) {

        double  myc, r0, kappa;
        int i,ok;
        char line[50];
        FILE *fptr;

        ok = -1;
        fptr = OpenInFile();
        if (fptr != NULL) {
                for (i=1; i<=2; i++);
                        fgets(line,49,fptr);
                fscanf(fptr,"%s%lf",line,&mys); // Reading data from SemiSlab.in
                fscanf(fptr,"%s%lf",line,&mya);
                fscanf(fptr,"%s%lf",line,&n);
                fscanf(fptr,"%s%lf",line,&dz);
                fscanf(fptr,"%s%lf",line,&dx);
                fscanf(fptr,"%s%lf",line,&dx0);
                fscanf(fptr,"%s%lf",line,&dyc);
                fgets(line,49,fptr);
                fscanf(fptr,"%s%lf",line,&start_time);
                fscanf(fptr,"%s%lf",line,&time_step);
                fscanf(fptr,"%s%lf",line,&stop_time);
```

```
                        fclose(fptr);
                        x0     = 0.001*dx0;
                        // Source/det. distance from edge (m)
                        yc     = 0.001*dyc;
                        // Plane y = yc   (m)
                        d      = 0.001*dz;
                        // Slab thickness  (m)
                        diff   = 1/(3*(mys+mya));    // Diffusion coefficient  (m)
                        c      = c0/n;               // Speed of light in sample(m/s)
                        myc    = cos(asin(c/c0));
                        r0     = sqr(((c0-c)/(c0+c)));
                        kappa  = ((1-r0)*(1-myc*myc))/(1+r0+(1-r0)*myc*myc*myc);
                        ze     = 2*diff/kappa;       // Extrapolated boundary  (m)
                        z00    = 1/mys;              // The position of the source(m)
                        zp0    = -(2*ze + z00);      // Image source, z-coord (m)
                        x_image= 2*x0+2*ze;     // x-position of image source dipoles(m)
                        gamma  = 4*diff*c;     // Constant in exp-terms in fluence/escape
                        eta    = pow (gamma*pi,-1.5); // Constant in fluence/escape
                        abscissa[0]=0.1488743389;
                        abscissa[1]=0.4333953941;
                        abscissa[2]=0.6794095682;
                        abscissa[3]=0.8650633666;
                        abscissa[4]=0.9739065285;
                        weight[0]=0.2955242247;
                        weight[1]=0.2692667193;
                        weight[2]=0.2190863625;
                        weight[3]=0.1494513491;
                        weight[4]=0.0666713443;
                        ok     = 0;
                }
                return(ok);
}


inline double z0(int m) {                // z-position of the positive part
                return(2*m*(d+2*ze)+z00);  // of the source dipole for x=0,
}                                        // negative for x = x_image


inline double zp(int m) {                // z-position of the negative part
                return(2*m*(d+2*ze)+zp0);  // of the source dipole for x=0,
}                                        // positive for x = x_image


inline double q0(int m,double z) {       // z-position of the positive part
                return(2*m*(d+2*ze)+z);    // of the volume element dipole for x,
}                                        // negative for qx_image


inline double qp(int m,double z) {       // z-position of the negative part
                return(2*m*d+(4*m-2)*ze-z); // of the volume element dipole for x,
}                                        // positive for qx_image

// Photon-fluence: in (x,yc,z) at time t due to point sources in a semi slab geometry

inline double fluence (double x,double z,double t) {

                int i;
                double ro2i,rp2i,rxo2i,rxp2i;          // Distances between (x,yc,z) and i:th
                double sum,dummy           ;           // pos. source for x=0; neg. source
                                                       // for x=0; neg. source for x=x_image;
                sum = 0.0;                          // pos. source for x=x_image, respectively
                if (t>0.0) {
                        dummy = gamma*t;
                        for (i=-2; i<=2; i++) {     // i = order of image dipole
                                ro2i = sqr(x)+sqr(yc)+sqr(z0(i)-z);
                                rp2i = sqr(x)+sqr(yc)+sqr(zp(i)-z);
                                rxo2i = sqr(x-x_image)+sqr(yc)+sqr(z0(i)-z);
                                rxp2i = sqr(x-x_image)+sqr(yc)+sqr(zp(i)-z);
```

```
                                        sum += exp(-ro2i/dummy) - exp(-rp2i/dummy);
                                                                  // From x=0 dipoles
                                        sum += -exp(-rxo2i/dummy) + exp(-rxp2i/dummy);
                                                                  // From x_image dipoles
                        }
                        sum *= c*eta*pow(t,-1.5)*exp(-mya*c*t);
                }
                return(sum);
}


// Escape-fuction: probability of a photon in (x,yc,z) reaching detector at time t

inline double escape(double x,double z,double t) {

                int i;
                double k2i,l2i,kx2i,lx2i;  // Distances between det. (0,0,d) and i:th
                double sum,dummy;          // pos. vol. elem. source for x; neg. source
                                           // for x; neg. source for x_image-x; pos. source
                sum = 0.0;                 // for x_image-x, respectively
                if (t>0.0) {
                        dummy = gamma*t;
                        for (i=-2; i<=2; i++) {     // i = order of image dipole
                                k2i = sqr(x)+sqr(yc)+sqr(q0(i,z)-d);
                                l2i = sqr(x)+sqr(yc)+sqr(qp(i,z)-d);
                                kx2i = sqr(x_image-2*x)+sqr(yc)+sqr(q0(i,z)-d);
                                lx2i = sqr(x_image-2*x)+sqr(yc)+sqr(qp(i,z)-d);
                                sum += exp(-k2i/dummy) - exp(-l2i/dummy);
                                sum += -exp(-kx2i/dummy) + exp(-lx2i/dummy);
                                }
                        sum *= eta*pow(t,-1.5)*exp(-mya*c*t);
                }
                return(sum);
}


double integrate(double a,double b, double x,double z) { // Outer (time interval)
                                                         // integral
                double ss,tm,tr,dt;
                int j;

                ss = 0.0;
                // 10 point Gauss-Legendre integration
                tm = 0.5*(b+a);
                tr = 0.5*(b-a);
                for (j=0;j<=4;j++) {       // Symmetric in interval => 5 iterations
                        dt = tr*abscissa[j];
                        ss += weight[j]*(conv(x,z,tm+dt)+conv(x,z,tm-dt));
                }
                return(ss*tr);
                // tr = scaling factor
}

inline double conv(double x,double z,double t) {       // Inner (convolution)
                                                       // integral
                double ss,tm,tr,dt,a,b;
                int j;

                a  = tau1;
                b  = t-tau2;
                ss = 0.0;
                if (a<b) {
                        tm = 0.5*(b+a);
                        tr = 0.5*(b-a);
                        for (j=0;j<=4;j++) {
                                dt = tr*abscissa[j];
                                ss += weight[j]*(f(x,z,tm+dt,t)+f(x,z,tm-dt,t));
                        }
                }
                return(ss*tr);
```

```
}

inline double f(double x,double z,double tprim,double t) { // Integrand

            return(fluence(x,z,tprim)*escape(x,z,t-tprim));
}
```

A.3.2 Template Input File for `SemiSlab`

```
Data to calculate light fluence in a slab

Scat_coeff(1/m):   1000.0
Abs_coeff_(1/m):   100.0
Refractiveindex:   1.4
Thick.___z_(mm):   15
Radius___x_(mm):   7
Edge_dist_x(mm):   3
Plane____y_(mm):   0

Start_time_(ns):   0.0
Time_step__(ns):   0.02
Stop_time__(ns):   0.8
```

Appendix B: Monte Carlo Computer Code

B.1 PROGRAM MCML

Modifications from the original MCML are:

- Time resolved calculation
- Customization for photon hitting density calculation as described in section 3.4.
- Option of saving intermediate results during simulation to prevent loss of all data in case of power black-out

MCML comprises of four modules: `mcmlmain.c` (main program), `mcmlio.c` (module handling in/out operations), `mcmlgo.c` (module handling Monte Carlo simulation), and `mcmlnr.c` (module containing numerical routines). For further details see Ref. 16.

B.1.1 mcml.h

```
/*************************************************************
 *
 *            Monte Carlo simulation of photon distribution in
 *            turbid media in Borland C++.
 ****
 *            Starting Date:              10/1991.
 *            Current Date: 08/1996.
 *
 *            Lihong Wang, Ph. D.
 *            Steven L. Jacques, Ph. D.
 *            Roger Berg
 *            Johannes Swartling
 *
 ****
 *            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
#define CVAC 3E10                       /* Speed of light in vaccum
[cm/s]. */
#define WEIGHT 1E-4                     /* Critical weight for
roulette. */
#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.
*/
  double s;                             /* current step size. [cm]. */
  double sleft;                         /* step size left.
dimensionless [-]. */
  double ttotal;    /* total elapsed time */
} PhotonStruct;

/****
 *            Structure used to describe the geometry and optical
```

```
 *           properties of a layer.
 ****/
typedef struct {
  double z0, z1;           /* z coordinates of a layer. [cm] */
  double n;                          /* refractive index of a
layer. */
  double nambient;         /* ref. index of surrounding media*/
  double mus;
  double mua;
  double g;                    /* anisotropy. */

  double cos_crit0,        cos_crit1;
} LayerStruct;

/****
 *           Input parameters for each independent run.
 ****/
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 dt;                                             /* t grid
separation [s] */

  short nz;                                                      /*
array range 0..nz-1. */
  short nr;                                                      /*
array range 0..nx-1. */
  short nt;                                                      /*
array range 0..nt-1. */
  short Detr;                                                    /* Detector
position */
  short TR;                                                      /*
Transmission=1, reflectance=0 */

  short save_intermediate;  /* 1=save, 0=don't */

  LayerStruct layerspecs;  /* layer parameters. */
} InputStruct;

/****
 *           Structures for scoring physical quantities.
 ****/
typedef struct {

  double    Rsp;           /* specular reflectance. [-] */

  double ** A_rz;          /* Abs. matrix for logging photons */

  double ***A_rzt;         /* 2D probability density in turbid */
                                                                    /*
media over r, z & t. [1/cm3] */

  double * Det_Int;        /* detected intensity as a fcn of t */

  long * Det_Photons;      /* # of detected photons per time interval
*/

} OutStruct;

/*************************************************************
 *           Routine prototypes for dynamic memory allocation and
 *           release of arrays and matrices.
 ****/
double  *AllocVector(short, short);
double **AllocMatrix(short, short,short, short);
double ***Alloc3Matrix(short,short,short,short,short,short);
double
****Alloc4Matrix(short,short,short,short,short,short,short,short);
void          FreeVector(double *, short, short);
void          FreeMatrix(double **, short, short, short, short);
void          Free3Matrix(double
***,short,short,short,short,short,short);
```

```
  }  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 ' ', '\f', '\t' 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(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 "#".
 *
 *      Also return 1 if this line is space line.
 ****/
Boolean CommentLine(char *Buf)
{
  size_t spn, cspn;

  spn = strspn(Buf, " \t");
  /* 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);
}

/*************************************************************
 *      Skip file version, then read number of runs.
```

```
 ****/
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);
}
/*************************************************************
 *      Allocate the arrays in OutStruct for one run, and
 *      array elements are automatically initialized to zeros.
 ****/
void InitOutputData(InputStruct * In_Ptr,

             OutStruct * Out_Ptr)
{
  short nz = In_Ptr->nz;
  short nr = In_Ptr->nr;
  short nt = In_Ptr->nt;

  if(nz<=0 || nr<=0)
             nrerror("Wrong grid parameters.\n");

  /* Init pure numbers. */
  Out_Ptr->Rsp = 0.0;

  /* Allocate the arrays and the matrices. */
  Out_Ptr->Det_Int = AllocVector(0,nt-1);
  Out_Ptr->Det_Photons = (long *)AllocVector(0,nt-1);

  Out_Ptr->A_rz = AllocMatrix(0,nr-1,0,nz-1);

  Out_Ptr->A_rzt = Alloc3Matrix(0,nr-1,0,nz-1,0,nt-1);

}


/*************************************************************
 *      Read the file name and the file format.
 *
 *      The file format can be either A for ASCII or B for
 *      binary.
 ****/
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 number of photons.
 ****/
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.\n");
  sscanf(buf, "%ld", &In_Ptr->num_photons);
  if(In_Ptr->num_photons<=0)
             nrerror("Nonpositive number of photons.\n");
}


/*************************************************************
 *      Read the members dz and dr.
 ****/
void ReadDrDzDt(FILE *File_Ptr, InputStruct *In_Ptr)
{
  char buf[STRLEN];

  /** read in dz, dr,dt. **/
```

```
  strcpy(buf, FindDataLine(File_Ptr));
  if(buf[0]=='\0') nrerror("Reading dr,dz,dt.\n");
  sscanf(buf, "%lf%lf%lf", &In_Ptr->dr,&In_Ptr->dz,&In_Ptr->dt);
  if(In_Ptr->dr<=0) nrerror("Nonpositive dr.\n");
  if(In_Ptr->dz<=0) nrerror("Nonpositive dz.\n");
  if(In_Ptr->dt<=0) nrerror("Nonpositive dt.\n");
}


/***********************************************************
 *      Read the members nz, nr, na.
 ****/
void ReadNrNzNt(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 dr, dz, dt's.\n");
  sscanf(buf, "%hd%hd%hd",
             &In_Ptr->nr, &In_Ptr->nz, &In_Ptr->nt);
  if(In_Ptr->nr<=0)
             nrerror("Nonpositive number of dr's.\n");
   if(In_Ptr->nz<=0)
             nrerror("Nonpositive number of dz's.\n");
  if(In_Ptr->nt<=0)
             nrerror("Nonpositive number of dt's.\n");
}



/***********************************************************
 *      Read the parameters of one layer.
 *
 *      Return 1 if error detected.
 *      Return 0 otherwise.
 *
 *      *Z_Ptr is the z coordinate of the current layer, which
 *      is used to convert thickness of layer to z coordinates
 *      of the two boundaries of the layer.
 ****/
Boolean ReadOneLayer(FILE *File_Ptr,
LayerStruct * Layer_Ptr)
{
  char buf[STRLEN], msg[STRLEN];
  double d, n, mua, mus, g;     /* d is thickness. */

  strcpy(buf, FindDataLine(File_Ptr));
  if(buf[0]=='\0') return(1);

  sscanf(buf, "%lf", &Layer_Ptr->nambient );
  if(Layer_Ptr->nambient<=0) return(1);

  strcpy(buf, FindDataLine(File_Ptr));
  if(buf[0]=='\0') return(1);   /* error. */

  sscanf(buf, "%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 = 0.0;
  Layer_Ptr->z1 = d;

  return(0);
}

/***********************************************************
 *      Compute the critical angles for total internal
 *      reflection according to the relative refractive index
 *      of the layer.
 *      All layers are processed.
 ****/
void CriticalAngle( LayerStruct * Layer_Ptr)
{
  double n1, n2;

             n1 = Layer_Ptr->n;
             n2 = Layer_Ptr->nambient;
             Layer_Ptr->cos_crit0 = n1>n2 ?
                          sqrt(1.0 - n2*n2/(n1*n1)) : 0.0;
```

```
            Layer_Ptr->cos_crit1 = n1>n2 ?
                        sqrt(1.0 - n2*n2/(n1*n1)) : 0.0;
}
/*********************************************************
 *********************************************************/


void ReadDetrTR(FILE *File_Ptr, InputStruct *In_Ptr)
{
  char buf[STRLEN];

  /** read in TR, Exdr. **/
  strcpy(buf, FindDataLine(File_Ptr));
  if(buf[0]=='\0') nrerror("Reading DetrTR.\n");
  sscanf(buf, "%hd %hd %hd", &In_Ptr->Detr, &In_Ptr->TR,
                        &In_Ptr->save_intermediate);

}
/*************************************************************/


void InitInParm(InputStruct *In_Ptr){

  short nz = In_Ptr->nz;
  short nr = In_Ptr->nr;

  if(nz<=0 || nr<=0)
            nrerror("Wrong grid parameters.\n");

}


/*************************************************************
 *      Read in the input parameters for one run.
 ****/
void ReadParm(FILE* File_Ptr, InputStruct * In_Ptr, int Run)
{
  char buf[STRLEN];

  In_Ptr->Wth = WEIGHT;

  ReadFnameFormat(File_Ptr, In_Ptr);
  ReadNumPhotons(File_Ptr, In_Ptr);
  ReadDrDzDt(File_Ptr, In_Ptr);
  ReadNrNzNt(File_Ptr, In_Ptr);
  ReadDetrTR(File_Ptr, In_Ptr);

  ReadOneLayer(File_Ptr, &In_Ptr->layerspecs);
  if(Run){
            InitInParm(In_Ptr);
  }
  CriticalAngle(&In_Ptr->layerspecs);
}

/*************************************************************
 *      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);
}

/*************************************************************
 *      Add the name to the name list.
 ****/
void 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;
  }
  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.
 ****/
void FreeFnameList(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,0);

            name_taken = FnameTaken(In_Ptr->out_fname, &head);
            if(name_taken)
              sprintf(msg, "file name %s duplicated.\n",

            In_Ptr->out_fname);

            if(name_taken) nrerror(msg);
  }
  FreeFnameList(head);
  rewind(File_Ptr);
}



/*********************************************************
 *      Undo what InitOutputData did.
 *  i.e. free the data allocations.
 ****/
void FreeData(InputStruct * In_Ptr, OutStruct * Out_Ptr)
{
  short nz = In_Ptr->nz;
  short nr = In_Ptr->nr;
  short nt = In_Ptr->nt;

  Free3Matrix(Out_Ptr->A_rzt, 0, nr-1, 0,nz-1,0,nt-1);
  FreeMatrix(Out_Ptr->A_rz, 0, nr-1, 0, nz-1);
}
/*********************************************************/

void SumScaleResult(InputStruct * In_Ptr, OutStruct * Out_Ptr){

  short nz = In_Ptr->nz;
  short nr = In_Ptr->nr;
  short nt = In_Ptr->nt;
  short ir,iz,it;
```

```
  double dr= In_Ptr->dr;
  double dz= In_Ptr->dz;
  long  n_photons=In_Ptr->num_photons;

  for(ir=0;ir<nr;ir++)
            for(iz=0;iz<nz;iz++)
                        for(it=0;it<nt;it++){
                                    Out_Ptr-
>A_rzt[ir][iz][it]/=(dr*dz*(double)n_photons);
                        /* Scale so that sampling density is the
same for every */
                        /* radius, i.e. division by (2*ir + 1) */
                                    Out_Ptr-
>A_rzt[ir][iz][it]/=(2*ir+1);
               }

}

/*************************************************************
 *      Write the input parameters to the file.
 ****/
void WriteInParm(FILE *file, InputStruct * In_Ptr)
{
  short i;

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

  fprintf(file,
            "%s \tA\t\t# output file name, ASCII.\n",
            In_Ptr->out_fname);
  fprintf(file,
            "%ld \t\t\t# No. of photons\n", In_Ptr->num_photons);

  fprintf(file,
            "%G\t%G\t%G\t\t# dr, dz [cm], dt [s]\n", In_Ptr-
>dr,In_Ptr->dz,In_Ptr->dt);
  fprintf(file, "%hd\t%hd\t%hd\t# No. of dr, dz, dt.\n",
              In_Ptr->nr, In_Ptr->nz, In_Ptr->nt);
  fprintf(file, "%hd\t%hd\t# Detr, TR.\n", In_Ptr->Detr, In_Ptr->TR);

  fprintf(file,
            "%G\t\t\t\t\t# n for medium above\n",
            In_Ptr->layerspecs.nambient);
  fprintf(file, "%G\t%G\t%G\t%G\t%G\t%G\n",
                        In_Ptr->layerspecs.n, In_Ptr-
>layerspecs.mua, In_Ptr->layerspecs.mus,
                        In_Ptr->layerspecs.g, In_Ptr-
>layerspecs.z0, In_Ptr->layerspecs.z1);
  fprintf(file, "\t\t# n, mua, mus, g, z0, z1.\n\n");
}

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


//*******************************   */
void WriteDet_Photons(FILE * file,
                                                short Nt,
                                                      OutStruct
*Out_Ptr)
{
  short it;
  long tot;

  tot=0;

  fprintf(file,"Det_Photons\n");

  for(it=0;it<Nt;it++){
        fprintf(file, "%ld\n", Out_Ptr->Det_Photons[it]);
        tot+=Out_Ptr->Det_Photons[it];
  }
  fprintf(file, "\nTotal no. of detected photons:\t%ld\n\n",tot);

}

//*******************************   */
void WriteDet_Int(FILE * file,
                                                short Nt,
                                                      OutStruct
*Out_Ptr)
{
  short it;

  fprintf(file,"Det_Int\n");

  for(it=0;it<Nt;it++){
```

```
        fprintf(file, "%G\n", Out_Ptr->Det_Int[it]);
  }
  fprintf(file, "\n");
}

//*********************************    */
void WriteA_rzt(FILE * file,
                                                    short Nr,
                                                    short Nz,
                                                    short Nt,
                                                    OutStruct
*Out_Ptr)
{
  short ir,iz,it;

  fprintf(file,"A_rzt\n\n");

  for(it=0;it<Nt;it++){
             fprintf(file,"Time %d \n",it);
             for(iz=0;iz<Nz;iz++){
             for(ir=0; ir<Nr; ir++)
                                    fprintf(file, "%12.4E\t",
                                     Out_Ptr->A_rzt[ir][iz][it]);
                                    fprintf(file,"\n");
       }
  }

  fprintf(file, "\n");
}

/*********************************************************
 ****/
void WriteResult(long Pi, InputStruct * In_Ptr,
                                                     OutStruct *
Out_Ptr,
                                                     char *
TimeReport)
{
  long phot_left=0;
  FILE *file;

  file = fopen(In_Ptr->out_fname, "w");
  if(file == NULL) nrerror("Cannot open file to write.\n");
  fprintf(file, "# %s", TimeReport);
  fprintf(file, "\n");

  if(Pi>0){
             phot_left = Pi - In_Ptr->num_photons/10;
             fprintf(file,
             "# ERROR! Simulation interrupted externally with more
than\n");
             fprintf(file,"# %ld photons still left to
go.\n\n",phot_left);
  }

  WriteInParm(file, In_Ptr);
  WriteDet_Photons(file, In_Ptr->nt, Out_Ptr);
  WriteDet_Int(file, In_Ptr->nt, Out_Ptr);
  WriteA_rzt(file, In_Ptr->nr, In_Ptr->nz, In_Ptr->nt, Out_Ptr);

  fclose(file);
}
```

B.1.4 mcmlgo.c

```
/*********************************************************
 *  Copyright Univ. of Texas M.D. Anderson Cancer Center
 *  1992.
 *
 *      Launch, move, and record photon weight.
 ****/

#include "mcml.h"

#define STANDARDTEST 0
  /* testing program using fixed rnd seed. */

#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 < 0 ? -*idum : *idum);
            mj %= MBIG;
            ma[55]=mj;
            mk=1;
            for (i=1;i<=54;i++) {
              ii=(21*i) % 55;
              ma[ii]=mk;
              mk=mj-mk;
              if (mk < MZ) mk += MBIG;
              mj=ma[ii];
            }
            for (k=1;k<=4;k++)
              for (i=1;i<=55;i++) {
            ma[i] -= ma[1+(i+30) % 55];
            if (ma[i] < MZ) ma[i] += MBIG;
              }
            inext=0;
            inextp=31;
            *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;
}

#undef MBIG
#undef MSEED
#undef MZ
#undef 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)
{
  double r1, r2;
             /* direct reflections from the 1st and 2nd layers. */
  double temp;

  temp =(Layerspecs.n - Layerspecs.nambient)
               /(Layerspecs.n + Layerspecs.nambient);
  r1 = temp*temp;

  return (r1);
}

/*************************************************************/
void SetAbsZero(InputStruct * In_Ptr,
                          OutStruct * Out_Ptr)
{
  short ir, iz;

       /* set abs. array to zero before every new photon packet*/
  for(ir=0; ir<In_Ptr->nr; ir++)
             for(iz=0; iz<In_Ptr->nz; iz++)
                          Out_Ptr->A_rz[ir][iz]=0;
}

/*************************************************************
 *      Initialize a photon packet.
 ****/
void LaunchPhoton(InputStruct  * Inp_Ptr,
                                          PhotonStruct * Photon_Ptr,
                                          OutStruct * Out_Ptr)
{
  double Rspecular=Out_Ptr->Rsp;
  double x=Photon_Ptr->x, y=Photon_Ptr->y;
  short ir, iz;

  Photon_Ptr->w         = 1.0 - Rspecular;

  Photon_Ptr->dead      = 0;
  Photon_Ptr->s = 0;
  Photon_Ptr->sleft= 0;
  Photon_Ptr->ttotal= 0;

  Photon_Ptr->x         = 0.0;
  Photon_Ptr->y         = 0.0;
  Photon_Ptr->z         = 0.0;
  Photon_Ptr->ux        = 0.0;
  Photon_Ptr->uy        = 0.0;
  Photon_Ptr->uz        = 1.0;

  Out_Ptr->A_rz[0][0]=1;

}

/*************************************************************
 *      Choose (sample) a new theta angle for photon propagation
 *      according to the anisotropy.
 *
 *      If anisotropy g is 0, then
 *              cos(theta) = 2*rand-1.
 *      otherwise
 *              sample according to the Henyey-Greenstein function.
 *
 *      Returns the cosine of the polar deflection angle theta.
 ****/
double SpinTheta(double g)
{
  double cost,ddump,temp;

  if(g == 0.0)
             cost = 2*RandomNum() -1;
  else {
             ddump=(1-g+2*g*RandomNum());
             if(ddump==0){printf("SpinTheta div zero!\n");exit(1);}
             temp = (1-g*g)/ddump;
             cost = (1+g*g - temp*temp)/(2*g);
  }
  return(cost);
}


/*************************************************************
 *      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 Spin(double g,
                              PhotonStruct * Photon_Ptr)
{
  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;
  double uy = Photon_Ptr->uy;
  double uz = Photon_Ptr->uz;
  double psi,ddump;

  cost = SpinTheta(g);
  sint = sqrt(1.0 - cost*cost);
            /* sqrt() is faster than sin(). */

  psi = 2.0*PI*RandomNum(); /* spin psi 0-2pi. */
  cosp = cos(psi);
  if(psi<PI)
            sinp = sqrt(1.0 - cosp*cosp);
              /* sqrt() is faster than sin(). */
  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*cosp*temp + uz*cost;
  }
}

/*************************************************************
 *      Move the photon s away in the current layer of medium.
 ****/
void Hop(PhotonStruct * Photon_Ptr,
                              InputStruct * In_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;
  Photon_Ptr->ttotal+=s*In_Ptr->layerspecs.n/CVAC;
}

/*************************************************************
 *      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)
{
  double mus,mua;

  mua = In_Ptr->layerspecs.mua;
  mus = In_Ptr->layerspecs.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 the boundary.
 *      Return 1 if hit boundary.
 *      Return 0 otherwise.
 *
 *      If the projected step hits the boundary, the members
 *      s and sleft of Photon_Ptr are updated.
 ****/
Boolean HitBoundary(PhotonStruct *  Photon_Ptr,

            InputStruct  *  In_Ptr)
{
  double mut;
  double dl_b;  /* length to boundary. */
  double uz = Photon_Ptr->uz;
  Boolean hit;
  short ix,iy,iz;


  /* Distance to the boundary. */
  if(uz>0.0)
            dl_b = (In_Ptr->layerspecs.z1
                                    - Photon_Ptr->z)/uz;     /*
dl_b>0. */
  else if(uz<0.0)
            dl_b = (In_Ptr->layerspecs.z0
                                    - Photon_Ptr->z)/uz;     /*
dl_b>0. */

  if(uz != 0.0 && Photon_Ptr->s > dl_b) {
              /* not horizontal & crossing. */
                mut = In_Ptr->layerspecs.mua+In_Ptr->layerspecs.mus;

            Photon_Ptr->sleft = (Photon_Ptr->s - dl_b)*mut;
            Photon_Ptr->s     = dl_b;
            hit = 1;
  }
  else
            hit = 0;

  return(hit);
}
/*************************************************************
 *    Update A_rzt if photon hits detector
 *
 */
void HitDetector(InputStruct * In_Ptr,
                                          PhotonStruct * Photon_Ptr,
                                          OutStruct * Out_Ptr)
{

  short ir, iz, it; .

  it = (short)(Photon_Ptr->ttotal/In_Ptr->dt);
  if(it>In_Ptr->nt-1) it=In_Ptr->nt-1;
  if(it<0) it=0;

  Out_Ptr->Det_Photons[it]++;
  Out_Ptr->Det_Int[it] += Photon_Ptr->w;
  for(ir=0; ir<In_Ptr->nr; ir++)
            for(iz=0; iz<In_Ptr->nz; iz++)
                        Out_Ptr->A_rzt[ir][iz][it] +=
                          Photon_Ptr->w*Out_Ptr->A_rz[ir][iz];
  }

/*************************************************************
 *      Drop photon weight inside the tissue (not glass).
 *
 *  The photon is assumed not dead.
 *
 *      The weight drop is dw = w*mua/(mua+mus).
 *
 *      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 mua, mus;
  double x=Photon_Ptr->x ,y=Photon_Ptr->y;
  short ir,iz,it;

  ir = (short)floor((sqrt(x*x+y*y))/In_Ptr->dr);
  if(ir>In_Ptr->nr-1) ir=In_Ptr->nr-1;
  if(ir<0) ir=0;

  iz = (short)(Photon_Ptr->z/In_Ptr->dz);
  if(iz>In_Ptr->nz-1) iz=In_Ptr->nz-1;
  if(iz<0) iz=0;

  it = (short)(Photon_Ptr->ttotal/In_Ptr->dt);
  if(it>In_Ptr->nt-1) it=In_Ptr->nt-1;
  if(it<0) it=0;

  mua = In_Ptr->layerspecs.mua;
  mus = In_Ptr->layerspecs.mus;

  dwa = Photon_Ptr->w * mua/(mua+mus);
  Photon_Ptr->w -= dwa;

  /* assign dwa to the absorption array element. */
  Out_Ptr->A_rz[ir][iz]++;              /* One instead of 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 a1
 *      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 n1,      /* incident refractive index.*/
                                                    double n2,
/* transmit refractive index.*/
                                                    double ca1,
/* cosine of the incident */

                          /* angle. 0<a1<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 = 0.0;
  }
  else if(ca1>COSZERO) {        /** normal incident. **/
            *ca2_Ptr = ca1;
            r = (n2-n1)/(n2+n1);
            r *= r;
  }
  else if(ca1<COS90D)  {        /** very slant. **/
            *ca2_Ptr = 0.0;
            r = 1.0;
  }
  else  {                                           /** general. **/
            double sa1, sa2;
              /* sine of the incident and transmission angles. */
            double ca2;
```

```
            sa1 = sqrt(1-ca1*ca1);
            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+ = cc - ss. */
              cam = ca1*ca2 + sa1*sa2; /* c- = cc + ss. */
              sap = sa1*ca2 + ca1*sa2; /* s+ = sc + cs. */
              sam = sa1*ca2 - ca1*sa2; /* s- = sc - cs. */
              r = 0.5*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                     Refl,   /* reflectance. */
                                          InputStruct  * In_Ptr,
                                          PhotonStruct * Photon_Ptr,
                                          OutStruct *    Out_Ptr)
{
  double x=Photon_Ptr->x, y=Photon_Ptr->y;
  short ir,iz;

  ir = (short)(sqrt(x*x+y*y)/In_Ptr->dr);
  if(ir>In_Ptr->nr-1) ir=In_Ptr->nr-1;
  if(ir<0) ir=0;

  iz = 0;

    /* add abs. array to result array if photon hits detector. */
  if(ir==In_Ptr->Detr && !In_Ptr->TR)  {
            Out_Ptr->A_rz[ir][iz]++;
            HitDetector(In_Ptr, Photon_Ptr, Out_Ptr);
            }
  SetAbsZero(In_Ptr, Out_Ptr); /* set A_rz to zero */
  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
 *      transmittance array.
 *
 *      Update the photon weight as well.
 ****/
void RecordT(double             Refl,
                                          InputStruct  * In_Ptr,
                                          PhotonStruct * Photon_Ptr,
                                          OutStruct *    Out_Ptr)
{
  double x=Photon_Ptr->x, y=Photon_Ptr->y;
  short ir,iz;

  ir = (short)(sqrt(x*x+y*y)/In_Ptr->dr);
  if(ir>In_Ptr->nr-1) ir=In_Ptr->nr-1;
  if(ir<0) ir=0;

  iz = In_Ptr->nz-1;

    /* add abs. array to result array if photon hits detector. */
  if(ir==In_Ptr->Detr && In_Ptr->TR)  {
            Out_Ptr->A_rz[ir][iz]++;
            HitDetector(In_Ptr, Photon_Ptr, Out_Ptr);
            }
  SetAbsZero(In_Ptr, Out_Ptr); /* set A_rz to zero */
```

```
  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 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 the "layer" is not the first layer and the photon
 *      packet is transmitted, move the photon to "layer-1".
 *
 *      Update the photon parmameters.
 ****/
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=0.0; /* reflectance */
  double ni = In_Ptr->layerspecs.n;
  double nt = In_Ptr->layerspecs.nambient;

  /* Get r. */
  if( - uz <= In_Ptr->layerspecs.cos_crit0)
            r=1.0;              /* total internal reflection. */
  else
            r = RFresnel(ni, nt, -uz, &uz1);

  if(RandomNum() > r) {         /* transmitted to layer-1. */
              Photon_Ptr->uz = -uz1;
              RecordR(0.0, In_Ptr, Photon_Ptr, Out_Ptr);
              Photon_Ptr->dead = 1;
  }
  else                                          /* reflected. */
            Photon_Ptr->uz = -uz;
}

/*************************************************************
 *      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 layer, record the
 *      transmitted weight as transmittance. See comments for
 *      CrossUpOrNot.
 *
 *      Update the photon parmameters.
 ****/
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 */
  double ni = In_Ptr->layerspecs.n;
  double nt = In_Ptr->layerspecs.nambient;

  /* Get r. */
  if( uz <= In_Ptr->layerspecs.cos_crit1)
            r=1.0;          /* total internal reflection. */
  else
            r = RFresnel(ni, nt, uz, &uz1);

  if(RandomNum() > r) {         /* transmitted to layer+1. */
              Photon_Ptr->uz = uz1;
              RecordT(0.0, In_Ptr, Photon_Ptr, Out_Ptr);
              Photon_Ptr->dead = 1;
  }
  else                                          /* reflected. */
            Photon_Ptr->uz = -uz;
}
```

```
/*********************************************************
 ****/
void CrossOrNot(InputStruct  *  In_Ptr,
                                                    PhotonStruct *
Photon_Ptr,
                                                    OutStruct    *
Out_Ptr)
{
  if(Photon_Ptr->uz < 0.0)
            CrossUpOrNot(In_Ptr, Photon_Ptr, Out_Ptr);
  else
            CrossDnOrNot(In_Ptr, Photon_Ptr, Out_Ptr);
}

/*********************************************************
 *      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)
{
  StepSizeInTissue(Photon_Ptr, In_Ptr);

  if(HitBoundary(Photon_Ptr, In_Ptr)) {
            Hop(Photon_Ptr,In_Ptr); /* move to boundary plane. */
            CrossOrNot(In_Ptr, Photon_Ptr, Out_Ptr);
  }
  else {
            Hop(Photon_Ptr,In_Ptr);
            Drop(In_Ptr, Photon_Ptr, Out_Ptr);
            Spin(In_Ptr->layerspecs.g,
                        Photon_Ptr);
  }
}

/*********************************************************
 ****/
void HopDropSpin(InputStruct  *  In_Ptr,
                                                    PhotonStruct *
Photon_Ptr,
                                                    OutStruct    *
Out_Ptr)
{
  HopDropSpinInTissue(In_Ptr, Photon_Ptr, Out_Ptr);

  if( Photon_Ptr->w < In_Ptr->Wth && !Photon_Ptr->dead)
            Roulette(Photon_Ptr);
  if(Photon_Ptr->dead)
            SetAbsZero(In_Ptr, Out_Ptr);
}
```

B.1.5 mcmlnr.c

```
/*********************************************************
*  Copyright Univ. of Texas M.D. Anderson Cancer Center
*  1992.
*
*            Some routines modified from Numerical Recipes in C,
*            including error report, array or matrix declaration
*            and releasing.
****/
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

/*********************************************************
*            Report error message to stderr, then exit the program
*            with signal 1.
****/
void nrerror(char error_text[])
```

```
{
fprintf(stderr,"%s\n",error_text);
fprintf(stderr,"...now exiting to system...\n");
exit(1);
}

/*************************************************************
*             Allocate an array with index from nl to nh inclusive.
*
*             Original matrix and vector from Numerical Recipes in C
*             don't initialize the elements to zero. This will
*             be accomplished by the following functions.
****/
double *AllocVector(short nl, short nh)
{
double *v;
short i;

(double *)malloc((unsigned) (nh-nl+1)*sizeof(double));
if (!v) nrerror("allocation failure in vector()");

v -= nl;
for(i=nl;i<=nh;i++) v[i] = 0.0;          /* init. */
return v;
}

/*************************************************************
*             Allocate a matrix with row index from nrl to nrh
*             inclusive, and column index from ncl to nch
*             inclusive.
****/
double **AllocMatrix(short nrl,short nrh,
short ncl,short nch)
{
short i,j;
double **m;

(double **) malloc((unsigned) (nrh-nrl+1)
*sizeof(double*));
if (!m) nrerror("allocation failure 1 in matrix()");
m -= nrl;

for(i=nrl;i<=nrh;i++) {
=(double *) malloc((unsigned) (nch-ncl+1)
*sizeof(double));
if (!m[i]) nrerror("allocation failure 2 in matrix()");
-= ncl;
}

for(i=nrl;i<=nrh;i++)
for(j=ncl;j<=nch;j++) m[i][j] = 0.0;
return m;
}
/*************************************************************/

double ***Alloc3Matrix(short nrl,short nrh,
short ncl,short nch,
short ntl,short nth)
{
short i,j,k;
double ***m;

(double ***) malloc((unsigned) (nrh-nrl+1)
*sizeof(double**));
if (!m) nrerror("allocation failure 1 in matrix3()");
m -= nrl;

for(i=nrl;i<=nrh;i++) {
=(double **) malloc((unsigned) (nch-ncl+1)
*sizeof(double*));
if (!m[i]) nrerror("allocation failure 2 in matrix3()");
-= ncl;
}

for(i=nrl;i<=nrh;i++) {
for(j=ncl;j<=nch;j++){
[j]=(double *) malloc((unsigned) (nth-ntl+1)
*sizeof(double));
if (!m[i][j]) nrerror("allocation failure 3 in matrix3()");
[j] -= ntl;
}
}

for(i=nrl;i<=nrh;i++)
for(j=ncl;j<=nch;j++)
for(k=ntl;k<=nth;k++)
[j][k] = 0.0;
return m;
```

```
}

double ****Alloc4Matrix(short nrl,short nrh,
short ncl,short nch,
short ntl,short nth,
short nll,short nlh)
{
short i,j,k,l;
double ****m;

(double ****) malloc((unsigned) (nrh-nrl+1)
*sizeof(double***));
if (!m) nrerror("allocation failure 1 in matrix4()");
m -= nrl;

for(i=nrl;i<=nrh;i++) {
=(double ***) malloc((unsigned) (nch-ncl+1)
*sizeof(double**));
if (!m[i]) nrerror("allocation failure 2 in matrix4()");
-= ncl;
}

for(i=nrl;i<=nrh;i++) {
for(j=ncl;j<=nch;j++){
[j]=(double **) malloc((unsigned) (nth-ntl+1)
*sizeof(double*));
if (!m[i][j]) nrerror("allocation failure 3 in matrix4()");
[j] -= ntl;
}
}

for(i=nrl;i<=nrh;i++) {
for(j=ncl;j<=nch;j++){
for(k=ntl;k<=nth;k++){
[j][k]=(double *) malloc((unsigned) (nlh-nll+1)
*sizeof(double));
if (!m[i][j][k]) nrerror("allocation failure 4 in matrix4()");
[j] -= ntl;
}
}
}

for(i=nrl;i<=nrh;i++)
for(j=ncl;j<=nch;j++)
for(k=ntl;k<=nth;k++)
for(l=nll;l<=nlh;l++)
[j][k][l] = 0.0;
return m;
}

/*************************************************************
*             Release the memory.
****/
void FreeVector(double *v,short nl,short nh)
{
free((char*) (v+nl));
}

/*************************************************************
*             Release the memory.
****/
void FreeMatrix(double **m,short nrl,short nrh,
short ncl,short nch)
{
short i;

for(i=nrh;i>=nrl;i--) free((char*) (m[i]+ncl));
free((char*) (m+nrl));
}

/*************************************************************
*             Release the memory.
****/
void Free3Matrix(double ***m,short nrl,short nrh,
short ncl,short nch,short ntl,short nth)
{
short i,j;

for(i=nrh;i>=nrl;i--)
for(j=nch;j>=ncl;j--)
free((char*) (m[i][j]+ntl));

for(i=nrh;i>=nrl;i--) free((char*) (m[i]+ncl));
free((char*) (m+nrl));
}

void Free4Matrix(double ****m,short nrl,short nrh,
short ncl,short nch,short ntl,short nth,short nll, short nlh)
```

```
{
short i,j,k;

for(i=nrh;i>=nrl;i--)
for(j=nch;j>=ncl;j--)
for(k=nth;k>=ntl;k--)
free((char*) (m[i][j][k]+nll));

for(i=nrh;i>=nrl;i--)
for(j=nch;j>=ncl;j--)
free((char*) (m[i][j]+ntl));

for(i=nrh;i>=nrl;i--)
free((char*) (m[i]+ncl));

free((char*) (m+nrl));
}
```

B.1.6 Template Input File for MCML

```
####
# Template of input files for Monte Carlo simulation (mcml).
# Anything in a line after "#" is ignored as comments.
# Space lines are also ignored.
# Lengths are in cm, mua and mus are in 1/cm.
#
# The 'save_intermediate' parameter should be set to 1
# for long runs, to prevent all data from being lost
# in case of computer power loss etc. For shorter runs
# when testing this is not necessary and slows down the
# program.
####

                                        # file version
                                          # number of runs

### Specify data for run 1
test.mco      A                                              #
output filename, ASCII/Binary
                                                             #
No. of photons
0.1           25E-12                                         #
dr, dz, dt
10            20                                             #
No. of dr, dz & dt.
1             1                                              #
Detr, TR: 1=tr.,0=ref. sav.int.

# n     mua     mus     g       d               # One line for each
layer
                                                  # n for medium above.
1.0           10.0          0.0           1                  #
layer
```

B.2 PROGRAM MCMAT

Modifications from the above version of MCML are a matrix representation of the simulated object rather than multiple layers, and the option of inhomogeneities.

B.2.1 mcmat.h

```
/*************************************************************
 *
 *            Monte Carlo simulation of photon distribution in
 *            turbid media in Borland C++.
 ****
 *            Starting Date:              10/1991.
 *            Current Date: 08/1996.
 *
 *            Lihong Wang, Ph. D.
 *            Steven L. Jacques, Ph. D.
 *            Roger Berg
 *            Johannes Swartling
 *
 ****
 *            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
#define CVAC 3E10                       /* Speed of light in vaccum
[cm/s]. */
#define WEIGHT 1E-4                     /* Critical weight for
roulette. */
#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.
*/
  double s;                             /* current step size. [cm]. */
  double sleft;                         /* step size left.
dimensionless [-]. */
  double ttotal;    /* total elapsed time */
} PhotonStruct;

/****
 *            Structure used to describe the geometry and optical
 *            properties of a layer.
 ****/
typedef struct {
  double z0, z1;            /* z coordinates of a layer. [cm] */
  double n;                             /* refractive index of a
layer. */
  double nambient;          /* ref. index of surrounding media*/
  double ***mua;                /* absorption coefficient. [1/cm] */
  double ***mus;                /* scattering coefficient. [1/cm] */
  double musbulk;
  double muabulk;
  double g;                     /* anisotropy. */

  double cos_crit0,         cos_crit1;
} LayerStruct;

/****
 *            Input parameters for each independent run.
 ****/
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 dx;                                            /* x grid
separation.[cm] */
  double dy;                                            /* y grid
separation.[cm] */
  double dt;                                            /* t grid
separation [s] */

  short nz;                                                         /*
array range 0..nz-1. */
```

```
  short nx;                                                    /*
array range 0..nx-1. */
  short ny;                                                    /*
array range 0..ny-1. */
  short nt;                                                    /*
array range 0..nt-1. */

  double Inx;                 /* Input position in x-coord. */
  double Iny;
  double Inz;

  short Detx;               /* Detector position */
  short Dety;
  short TR;                 /* Transmission=1, reflectance=0 */
  short wrx;                /* X plane to be written to output file */
  short width;           /* Width of slice to be written to output file
*/

  short save_intermediate;  /* 1=save, 0=don't */

  LayerStruct layerspecs;  /* layer parameters. */
} InputStruct;

/****
 *            Structures for scoring physical quantities.
 ****/
typedef struct {

  double    Rsp;            /* specular reflectance. [-] */

  double *** A_xyz;         /* Abs. matrix for logging photons */

  double ****A_xyzt;        /* 2D probability density in turbid */
                                                               /*
media over r, z & t. [1/cm3] */

  double * Det_Int;         /* detected intensity as a fcn of t */

  long * Det_Photons;       /* # of detected photons per time interval
*/

} OutStruct;

/*************************************************************
```

```
 *            Routine prototypes for dynamic memory allocation and
 *            release of arrays and matrices.
 ****/
double  *AllocVector(short, short);
double **AllocMatrix(short, short,short, short);
double ***Alloc3Matrix(short,short,short,short,short,short);
double
****Alloc4Matrix(short,short,short,short,short,short,short,short);
void           FreeVector(double *, short, short);
void           FreeMatrix(double **, short, short, short, short);
void           Free3Matrix(double
***,short,short,short,short,short,short);
void           Free4Matrix(double
****,short,short,short,short,short,short,short,short);
void           nrerror(char *);
```

B.2.2 mcmatmain.c

```
/*************************************************************
 *  Copyright Univ. of Texas M.D. Anderson Cancer Center
 *  1992.
 *
 *            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

#if THINKCPROFILER
#include <profile.h>
#include <console.h>
#endif
```

```
#include "mcmat.h"

/*            Declare before they are used in main(). */
FILE *GetFile(char *);
short ReadNumRuns(FILE* );
void ReadParm(FILE* , InputStruct * ,int);
void CheckParm(FILE* , InputStruct * );
void InitOutputData(InputStruct *, OutStruct *);
void SumScaleResult(InputStruct *, OutStruct *);
void FreeData(InputStruct *, OutStruct *);
double Rspecular(LayerStruct );
void SetAbsZero(InputStruct *, OutStruct *);
void LaunchPhoton(InputStruct *, PhotonStruct *, OutStruct *);
void HopDropSpin(InputStruct  *,PhotonStruct *,OutStruct *);
void WriteResult(long, InputStruct * , OutStruct * , char *);
void ShowVersion(char *);


/*************************************************************
 *            If F = 0, reset the clock and return 0.
 *
 *            If F = 1, pass the user time to Msg and print Msg on
 *            screen, return the real time since F=0.
 *
 *            If F = 2, same as F=1 except no printing.
 *
 *            Note that clock() and 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.
 *
 *            clock() only hold 16 bit integer, which is about 32768
 *            clock ticks.
 ****/
time_t PunchTime(char F, char *Msg)
{
#if 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));
  else return(0);
#endif
}

/*************************************************************
 *            Print the current time and the estimated finishing time.
 *
 *            P1 is the number of computed photon packets.
 *            Pt is the total number of photon packets.
 ****/
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)P1);
  date = localtime(&done_time);
  strftime(s, 80, "%H:%M %x", date);
```

```
  printf("End %s\n", s);
}

/*************************************************************
 *            Report time and write results.
 ****/
void ReportResult(long Pi,InputStruct * In_Ptr, OutStruct * Out_Ptr)

{
  char time_report[STRLEN];

  strcpy(time_report, " Simulation time of this run.");
  PunchTime(1, time_report);

  SumScaleResult(In_Ptr, Out_Ptr);
  WriteResult(Pi, In_Ptr, Out_Ptr, time_report);
}

/*************************************************************
 *            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.
 *            Pi is the index to the current photon, counting down.
 *            Pt is the total number of photons.
 *            If In_Ptr->save_intermediate is set the intermediate
 *            result will be written to the out file.
 ****/
void ReportStatus(short Num_Runs, long Pi,
                                        InputStruct * In_Ptr,
OutStruct * Out_Ptr)
{
  long Pt=In_Ptr->num_photons;

  if(Pt-Pi == 10 || Pi*10%Pt == 0 && Pi != Pt) {
    printf("%ld photons & %hd runs left, ", Pi, Num_Runs);
    PredictDoneTime(Pt-Pi, Pt);
    if(In_Ptr->save_intermediate)
            ReportResult(Pi,In_Ptr,Out_Ptr);
  }
}

/*************************************************************
 *            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;

#if THINKCPROFILER
  InitProfile(200,200); cecho2file("prof.rpt",0, 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, &out_parm);
            LaunchPhoton(In_Ptr, &photon, &out_parm);
            do  HopDropSpin(In_Ptr, &photon, &out_parm);
            while (!photon.dead);
  } while(--i_photon);

#if THINKCPROFILER
```

```
  exit(0);
#endif

  ReportResult(0, 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 96-09-26");
  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,1);
            DoOneRun(num_runs, &in_parm);
  }

  fclose(input_file_ptr);
  return(0);
}
```

B.2.3 mcmatio.c

```
/************************************************************
 *  Copyright Univ. of Texas M.D. Anderson Cancer Center
 *  1992.
 *
 *      Input/output of data.
 ****/

#include "mcmat.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.
 ****/
char * CenterStr(short  Wid,
                                                    char * InStr,
                                                    char * OutStr)
{
  size_t nspaces;       /* number of spaces to be filled */
                                                              /*
before InStr. */

  nspaces = (Wid - strlen(InStr))/2;

  strcpy(OutStr, "");
  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[STRLEN];
```

```
  CenterStr(COLWIDTH,
            "mcmat - Monte Carlo Simulation of Matrix defined Turbid
Media",
            str);
  puts(str);
  puts("");

  CenterStr(COLWIDTH, "Lihong Wang", str);
  puts(str);

  CenterStr(COLWIDTH, "Steven L. Jacques", str);
  puts(str);

  CenterStr(COLWIDTH,
            "Modified for temporal and matrix applications by Roger
Berg",str);
  puts(str);

  CenterStr(COLWIDTH,
            "Modified for photon hitting density calculation by
Johannes Swartling",str);
  puts(str);

  puts("");

  CenterStr(COLWIDTH, version, str);
  puts(str);
  puts("\n\n\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):");
              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);
}

/************************************************************
 *      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 ' ', '\f', '\t' 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(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 "#".
 *
 *      Also return 1 if this line is space line.
 ****/
Boolean CommentLine(char *Buf)
{
  size_t spn, cspn;

  spn = strspn(Buf, " \t");
  /* 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);
}

/*************************************************************
 *      Skip file version, then read number of runs.
 ****/
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);
}
/*************************************************************
 *      Allocate the arrays in OutStruct for one run, and
 *      array elements are automatically initialized to zeros.
 ****/
void InitOutputData(InputStruct * In_Ptr,

            OutStruct * Out_Ptr)
{
  short nz = In_Ptr->nz;
  short ny = In_Ptr->ny;
  short nx = In_Ptr->nx;
  short nt = In_Ptr->nt;

  if(nz<=0 || nx<=0)
            nrerror("Wrong grid parameters.\n");
```

```
  /* Init pure numbers. */
  Out_Ptr->Rsp = 0.0;

  /* Allocate the arrays and the matrices. */
  Out_Ptr->Det_Int = AllocVector(0,nt-1);
  Out_Ptr->Det_Photons = (long *)AllocVector(0,nt-1);

  Out_Ptr->A_xyz = Alloc3Matrix(0,nx-1,0,ny-1,0,nz-1);

  Out_Ptr->A_xyzt = Alloc4Matrix(0,nx-1,0,ny-1,0,nz-1,0,nt-1);

}


/*************************************************************
 *      Read the file name and the file format.
 *
 *      The file format can be either A for ASCII or B for
 *      binary.
 ****/
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 number of photons.
 ****/
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.\n");
  sscanf(buf, "%ld", &In_Ptr->num_photons);
  if(In_Ptr->num_photons<=0)
            nrerror("Nonpositive number of photons.\n");
}


/*************************************************************
 *      Read the members dz and dr.
 ****/
void ReadDxDyDzDt(FILE *File_Ptr, InputStruct *In_Ptr)
{
  char buf[STRLEN];

  /** read in dz, dr,dt. **/
  strcpy(buf, FindDataLine(File_Ptr));
  if(buf[0]=='\0') nrerror("Reading dx,dy,dz,dt.\n");
  sscanf(buf, "%lf%lf%lf%lf", &In_Ptr->dx, &In_Ptr->dy,&In_Ptr-
>dz,&In_Ptr->dt);
  if(In_Ptr->dx<=0) nrerror("Nonpositive dx.\n");
  if(In_Ptr->dy<=0) nrerror("Nonpositive dy.\n");
  if(In_Ptr->dz<=0) nrerror("Nonpositive dz.\n");
  if(In_Ptr->dt<=0) nrerror("Nonpositive dt.\n");
}


/*************************************************************
 *      Read the members nz, nr, na.
 ****/
void ReadNxNyNzNt(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 dx, dy, dz, dt's.\n");
  sscanf(buf, "%hd%hd%hd%hd",
            &In_Ptr->nx, &In_Ptr->ny, &In_Ptr->nz, &In_Ptr->nt);
  if(In_Ptr->nx<=0)
            nrerror("Nonpositive number of dx's.\n");
  if(In_Ptr->ny<=0)
            nrerror("Nonpositive number of dy's.\n");
```

```
  if(In_Ptr->nz<=0)
            nrerror("Nonpositive number of dz's.\n");
  if(In_Ptr->nt<=0)
            nrerror("Nonpositive number of dt's.\n");
}

// Reading regions

void ReadRegions(FILE *File_Ptr, InputStruct *In_Ptr,int Run)
{
  char buf[STRLEN];
  int X,Y,Z,Regions,type,i,j,k,l,m,n,o;
  short maxy = In_Ptr->ny;
  double scattc,absc;

  strcpy(buf, FindDataLine(File_Ptr));
  if(buf[0]=='\0') nrerror("Reading Regions.\n");
  sscanf(buf, "%d", &Regions);

            if(Regions>0){
                        for(i=1;i<=Regions;i++){

                                    strcpy(buf,
FindDataLine(File_Ptr));
                                    if(buf[0]=='\0')
nrerror("Region scatt. abs.\n");
                                    sscanf(buf, "%lf%lf", &scattc,
&absc);

                                    strcpy(buf,
FindDataLine(File_Ptr));
                                    if(buf[0]=='\0')
nrerror("Reading reg.type.\n");
                                    sscanf(buf, "%d", &type);

                                    if(type==0){
                                                strcpy(buf,
FindDataLine(File_Ptr));
                                                if(buf[0]=='\0')
nrerror("Reading cube region coord.\n");
                                                sscanf(buf, "%d
%d %d %d %d %d",&j,&k,&l,&m,&n,&o);
                                                if(Run)

            for(X=j;X<=m;X++)

            for(Y=k;Y<=n;Y++)

                        for(Z=l;Z<=o;Z++){

                                    In_Ptr-
>layerspecs.mus[X][Y][Z]=scattc;

                                    In_Ptr-
>layerspecs.mua[X][Y][Z]=absc;

                        }
                                    }

                                    if(type==1){
                                                strcpy(buf,
FindDataLine(File_Ptr));
                                                if(buf[0]=='\0')
nrerror("Reading sphere region coord.\n");
                                                sscanf(buf, "%d
%d %d %d",&j,&k,&l,&m);
                                                if(Run)

            for(X=j-m;X<=j+m;X++)

            for(Y=k-m;Y<=k+m;Y++)

                        for(Z=l-m;Z<=l+m;Z++)

                                    if((int)sqrt(pow((double)(X-
j),2)+pow((double)(Y-k),2)+pow((double)(Z-l),2))<=m){

                                                In_Ptr-
>layerspecs.mus[X][Y][Z]=scattc;

                                                In_Ptr-
>layerspecs.mua[X][Y][Z]=absc;

                                    }
                                    }

                                    if(type==2){
```

```
                                                    strcpy(buf,
FindDataLine(File_Ptr));
                                                    if(buf[0]=='\0')
nrerror("Reading cylinder region coord.\n");
                                                    sscanf(buf, "%d
%d %d",&j,&k,&l);
                                                    if(Run)

            for(X=j-l;X<=j+l;X++)

            for(Y=0;Y<=maxy;Y++)

                        for(Z=k-l;Z<=k+l;Z++)

                                    if(sqrt(pow((double)(X-
j),2)+pow((double)(Z-k),2))<=l){

                                                    In_Ptr-
>layerspecs.mus[X][Y][Z]=scattc;

                                                    In_Ptr-
>layerspecs.mua[X][Y][Z]=absc;

                                    }
                                    }
                        }
            }
}

/*************************************************************
 *      Read the parameters of one layer.
 *
 *      Return 1 if error detected.
 *      Return 0 otherwise.
 *
 *      *Z_Ptr is the z coordinate of the current layer, which
 *      is used to convert thickness of layer to z coordinates
 *      of the two boundaries of the layer.
 ****/
Boolean ReadOneLayer(FILE *File_Ptr,

LayerStruct * Layer_Ptr)
{
  char buf[STRLEN], msg[STRLEN];
  double d, n, mua, mus, g;     /* d is thickness. */

  strcpy(buf, FindDataLine(File_Ptr));
  if(buf[0]=='\0') return(1);

  sscanf(buf, "%lf", &Layer_Ptr->nambient );
  if(Layer_Ptr->nambient<=0) return(1);

  strcpy(buf, FindDataLine(File_Ptr));
  if(buf[0]=='\0') return(1);   /* error. */

  sscanf(buf, "%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->muabulk = mua;
  Layer_Ptr->musbulk = mus;
  Layer_Ptr->g   = g;
  Layer_Ptr->z0 = 0;
  Layer_Ptr->z1 = d;

  return(0);
}

/*************************************************************
 *      Compute the critical angles for total internal
 *      reflection according to the relative refractive index
 *      of the layer.
 *      All layers are processed.
 ****/
void CriticalAngle( LayerStruct * Layer_Ptr)
{
  double n1, n2;

            n1 = Layer_Ptr->n;
            n2 = Layer_Ptr->nambient;
            Layer_Ptr->cos_crit0 = n1>n2 ?
                        sqrt(1.0 - n2*n2/(n1*n1)) : 0.0;

            Layer_Ptr->cos_crit1 = n1>n2 ?
                        sqrt(1.0 - n2*n2/(n1*n1)) : 0.0;
}
/*************************************************************
```

```
 ************************************************************/

void ReadInxyz(FILE *File_Ptr, InputStruct *In_Ptr)
{
  char buf[STRLEN];

  /** read in TR, Exdr. **/
  strcpy(buf, FindDataLine(File_Ptr));
  if(buf[0]=='\0') nrerror("Reading TR, Exdr.\n");
  sscanf(buf, "%lf %lf %lf", &In_Ptr->Inx, &In_Ptr->Iny, &In_Ptr->Inz);
  if(In_Ptr->Inz<0) nrerror("Nonpositive Inz.\n");
}
/************************************************************/

void ReadDetxyTR(FILE *File_Ptr, InputStruct *In_Ptr)
{
  char buf[STRLEN];

  /** read in TR, Exdr. **/
  strcpy(buf, FindDataLine(File_Ptr));
  if(buf[0]=='\0') nrerror("Reading DetxyTR.\n");
  sscanf(buf, "%hd %hd %hd", &In_Ptr->Detx, &In_Ptr->Dety, &In_Ptr-
>TR);

}
/************************************************************/

void ReadWrxWidth(FILE *File_Ptr, InputStruct *In_Ptr)
{
  char buf[STRLEN];

  /** read in TR, Exdr. **/
  strcpy(buf, FindDataLine(File_Ptr));
  if(buf[0]=='\0') nrerror("Reading WrxWidth.\n");
  sscanf(buf, "%hd %hd %hd", &In_Ptr->wrx,
                        &In_Ptr->width, &In_Ptr-
>save_intermediate);

}
/************************************************************/

void InitInParm(InputStruct *In_Ptr){

  short nz = In_Ptr->nz;
  short ny = In_Ptr->ny;
  short nx = In_Ptr->nx;

  if(nz<=0 || nx<=0)
            nrerror("Wrong grid parameters.\n");

  /* Allocate the matrices. */
  In_Ptr->layerspecs.mua = Alloc3Matrix(0,nx-1,0,ny-1,0,nz-1);

  In_Ptr->layerspecs.mus = Alloc3Matrix(0,nx-1,0,ny-1,0,nz-1);
}


void SetMusMua(InputStruct *In_Ptr){

            int i,j,k;

            for(i=0;i<In_Ptr->nx;i++)
                        for(j=0;j<In_Ptr->ny;j++)
                                    for(k=0;k<In_Ptr->nz;k++){
                                                In_Ptr-
>layerspecs.mus[i][j][k]=In_Ptr->layerspecs.musbulk;
                                                In_Ptr-
>layerspecs.mua[i][j][k]=In_Ptr->layerspecs.muabulk;
                                    }
}

/************************************************************
 *      Read in the input parameters for one run.
 ****/
void ReadParm(FILE* File_Ptr, InputStruct * In_Ptr,int Run)
{
  char buf[STRLEN];

  In_Ptr->Wth = WEIGHT;

  ReadFnameFormat(File_Ptr, In_Ptr);
  ReadNumPhotons(File_Ptr, In_Ptr);
  ReadDxDyDzDt(File_Ptr, In_Ptr);
  ReadNxNyNzNt(File_Ptr, In_Ptr);
  ReadInxyz(File_Ptr, In_Ptr);
  ReadDetxyTR(File_Ptr, In_Ptr);
  ReadWrxWidth(File_Ptr, In_Ptr);
```

```
  ReadOneLayer(File_Ptr, &In_Ptr->layerspecs);
  if(Run){
            InitInParm(In_Ptr);
            SetMusMua(In_Ptr);
  }
  ReadRegions(File_Ptr, In_Ptr,Run);
  CriticalAngle(&In_Ptr->layerspecs);
}

/*************************************************************
 *      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);
}

/*************************************************************
 *      Add the name to the name list.
 ****/
void 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;
  }
  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.
 ****/
void FreeFnameList(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,0);

             name_taken = FnameTaken(In_Ptr->out_fname, &head);
             if(name_taken)
               sprintf(msg, "file name %s duplicated.\n",

             In_Ptr->out_fname);

             if(name_taken) nrerror(msg);
  }
  FreeFnameList(head);
  rewind(File_Ptr);
}



/************************************************************
 *      Undo what InitOutputData did.
 *  i.e. free the data allocations.
 ****/
void FreeData(InputStruct * In_Ptr, OutStruct * Out_Ptr)
{
  short nz = In_Ptr->nz;
  short ny = In_Ptr->ny;
  short nx = In_Ptr->nx;
  short nt = In_Ptr->nt;

  Free4Matrix(Out_Ptr->A_xyzt, 0, nx-1,0,ny-1, 0,nz-1,0,nt-1);

  Free3Matrix(In_Ptr->layerspecs.mua, 0,nx-1,0,ny-1,0,nz-1);

  Free3Matrix(In_Ptr->layerspecs.mus, 0,nx-1,0,ny-1,0,nz-1);
}
/************************************************************/

void SumScaleResult(InputStruct * In_Ptr, OutStruct * Out_Ptr){
  short nz = In_Ptr->nz;
  short ny = In_Ptr->ny;
  short nx = In_Ptr->nx;
  short nt = In_Ptr->nt;
  short ix,iy,iz,it;
  double dx= In_Ptr->dx;
  double dy= In_Ptr->dy;
  double dz= In_Ptr->dz;
  long  n_photons=In_Ptr->num_photons;

  for(ix=0;ix<nx;ix++)
             for(iy=0;iy<ny;iy++)
                        for(iz=0;iz<nz;iz++)
                                   for(it=0;it<nt;it++)
                                              Out_Ptr-
>A_xyzt[ix][iy][iz][it]/=(dx*dy*dz*(double)n_photons);

}

/************************************************************
 *      Write the input parameters to the file.
 ****/
void WriteInParm(FILE *file, InputStruct * In_Ptr)
{
  short i;

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

  fprintf(file,
             "%s \tA\t\t# output file name, ASCII.\n",
             In_Ptr->out_fname);
  fprintf(file,
             "%ld \t\t\t# No. of photons\n", In_Ptr->num_photons);

  fprintf(file,
             "%G\t%G\t%G\t%G\t\t# dx, dy, dz [cm], dt [s]\n", In_Ptr-
>dx,In_Ptr->dy,In_Ptr->dz,In_Ptr->dt);
  fprintf(file, "%hd\t%hd\t%hd\t%hd\t# No. of dx, dy, dz, dt.\n",
               In_Ptr->nx, In_Ptr->ny, In_Ptr->nz, In_Ptr->nt);
  fprintf(file, "%hd\t%hd\t%hd\t# Detx, Dety, TR.\n", In_Ptr->Detx,
In_Ptr->Dety, In_Ptr->TR);
  fprintf(file, "%hd\t%hd\t\t# wrx and width.\n\n", In_Ptr->wrx,
In_Ptr->width);
```

```
  fprintf(file,
            "%G\t\t\t\t\t# n for medium above\n",
            In_Ptr->layerspecs.nambient);
  fprintf(file, "%G\t%G\t%G\t%G\t%G\t%G\n",
                          In_Ptr->layerspecs.n, In_Ptr-
>layerspecs.muabulk, In_Ptr->layerspecs.musbulk,
                          In_Ptr->layerspecs.g, In_Ptr-
>layerspecs.z0, In_Ptr->layerspecs.z1);
  fprintf(file, "\t\t# n, muabulk, musbulk, g, z0, z1.\n\n");
}

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

//*********************************   */
void WriteDet_Photons(FILE * file,
                                          short Nt,
                                                    OutStruct
*Out_Ptr)
{
  short it;
  long tot;

  tot=0;

  fprintf(file,"Det_Photons\n");

  for(it=0;it<Nt;it++){
        fprintf(file, "%ld\n", Out_Ptr->Det_Photons[it]);
        tot+=Out_Ptr->Det_Photons[it];
  }
  fprintf(file, "\nTotal no. of detected photons:\t%ld\n\n",tot);

}

//*********************************   */
void WriteDet_Int(FILE * file,
                                          short Nt,
                                                    OutStruct
*Out_Ptr)
{
  short it;

  fprintf(file,"Det_Int\n");

  for(it=0;it<Nt;it++){
        fprintf(file, "%G\n", Out_Ptr->Det_Int[it]);
  }
  fprintf(file, "\n");
}

//*********************************
void WriteA_xyzt(FILE * file,
                                                    short Nx,
                                                    short Ny,
                                                    short Nz,
                                                    short Nt,
                                                    short Wrx,
                                                    short Width,
                                                    OutStruct
*Out_Ptr)
{
  short ix,iy,iz,it,xmin,xmax;

  xmin=Wrx-Width;
  if (xmin<0) xmin=0;

  xmax=Wrx+Width+1;
  if (xmax>Nx) xmax=Nx;

  fprintf(file,"A_xyzt for x= %d to %d\n", xmin, (xmax-1));

  for(it=0;it<Nt;it++){
        for(ix=xmin; ix<xmax; ix++){
                    fprintf(file,"Time %d, X= %d\n",it,ix);
                    for(iz=0;iz<Nz;iz++){
                                for(iy=0;iy<Ny;iy++)
                                            fprintf(file,
"%12.4E\t",
                                                  Out_Ptr-
>A_xyzt[ix][iy][iz][it]);
                                fprintf(file,"\n");
                    }
        }
  }

  fprintf(file, "\n");
```

```
}

/*********************************************************
 ****/
void WriteResult(long Pi, InputStruct * In_Ptr,
                                                    OutStruct *
Out_Ptr,
                                                    char *
TimeReport)
{
  long phot_left=0;
  FILE *file;

  file = fopen(In_Ptr->out_fname, "w");
  if(file == NULL) nrerror("Cannot open file to write.\n");

  fprintf(file, "# %s", TimeReport);
  fprintf(file, "\n");

  if(Pi>0){
            phot_left = Pi - In_Ptr->num_photons/10;
            fprintf(file,
            "# ERROR! Simulation interrupted externally with more
than\n");
            fprintf(file,"# %ld photons still left to
go.\n\n",phot_left);
  }

  WriteInParm(file, In_Ptr);
  WriteDet_Photons(file, In_Ptr->nt, Out_Ptr);
  WriteDet_Int(file, In_Ptr->nt, Out_Ptr);
  WriteA_xyzt(file, In_Ptr->nx, In_Ptr->ny, In_Ptr->nz,
                          In_Ptr->nt, In_Ptr->wrx, In_Ptr->width,
Out_Ptr);

  fclose(file);
}
```

B.2.4 mcmatgo.c

```
/*********************************************************
 *  Copyright Univ. of Texas M.D. Anderson Cancer Center
 *  1992.
 *
 *      Launch, move, and record photon weight.
 ****/

#include "mcmat.h"

#define STANDARDTEST 0
  /* testing program using fixed rnd seed. */

#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 < 0 ? -*idum : *idum);
            mj %= MBIG;
            ma[55]=mj;
            mk=1;
            for (i=1;i<=54;i++) {
              ii=(21*i) % 55;
              ma[ii]=mk;
              mk=mj-mk;
```

```
            if (mk < MZ) mk += MBIG;
            mj=ma[ii];
          }
          for (k=1;k<=4;k++)
            for (i=1;i<=55;i++) {
          ma[i] -= ma[1+(i+30) % 55];
          if (ma[i] < MZ) ma[i] += MBIG;
            }
          inext=0;
          inextp=31;
          *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;
}

#undef MBIG
#undef MSEED
#undef MZ
#undef 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)
{
  double r1, r2;
          /* direct reflections from the 1st and 2nd layers. */
  double temp;

  temp =(Layerspecs.n - Layerspecs.nambient)
              /(Layerspecs.n + Layerspecs.nambient);
  r1 = temp*temp;

/*  if((Layerspecs_Ptr[1].mua == 0.0)
  && (Layerspecs_Ptr[1].mus == 0.0))  {
          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);
```

```
}

/*****************************************************/
void SetAbsZero(InputStruct * In_Ptr,
                        OutStruct * Out_Ptr)
{
  short ix, iy, iz;

     /* set abs. array to zero before every new photon packet*/
  for(ix=0; ix<In_Ptr->nx; ix++)
            for(iy=0; iy<In_Ptr->ny; iy++)
                        for(iz=0; iz<In_Ptr->nz; iz++)
                                    Out_Ptr->A_xyz[ix][iy][iz]=0;
}

/*********************************************************
 *      Initialize a photon packet.
 ****/
void LaunchPhoton(InputStruct  * Inp_Ptr,
                                        PhotonStruct * Photon_Ptr,
                                        OutStruct * Out_Ptr)
{
  double Rspecular=Out_Ptr->Rsp;
  short ix, iy, iz;

  if(Inp_Ptr->Inz==0.0)
              Photon_Ptr->w         = 1.0 - Rspecular;
  else
              Photon_Ptr->w         = 1.0;
  Photon_Ptr->dead      = 0;
  Photon_Ptr->s = 0;
  Photon_Ptr->sleft= 0;
  Photon_Ptr->ttotal= 0;

  Photon_Ptr->x         = Inp_Ptr->Inx;
  Photon_Ptr->y         = Inp_Ptr->Iny;
  Photon_Ptr->z         = Inp_Ptr->Inz;
  Photon_Ptr->ux        = 0.0;
  Photon_Ptr->uy        = 0.0;
  Photon_Ptr->uz        = 1.0;

  ix = (short)(Photon_Ptr->x/Inp_Ptr->dx);
  iy = (short)(Photon_Ptr->y/Inp_Ptr->dy);
  iz = (short)(Photon_Ptr->z/Inp_Ptr->dz);
  Out_Ptr->A_xyz[ix][iy][iz]=1;

}

/*********************************************************
 *      Choose (sample) a new theta angle for photon propagation
 *      according to the anisotropy.
 *
 *      If anisotropy g is 0, then
 *              cos(theta) = 2*rand-1.
 *      otherwise
 *              sample according to the Henyey-Greenstein function.
 *
 *      Returns the cosine of the polar deflection angle theta.
 ****/
double SpinTheta(double g)
{
  double cost,ddump,temp;

  if(g == 0.0)
              cost = 2*RandomNum() -1;
  else {
              ddump=(1-g+2*g*RandomNum());
              if(ddump==0){printf("SpinTheta div zero!\n");exit(1);}
              temp = (1-g*g)/ddump;
              cost = (1+g*g - temp*temp)/(2*g);
  }
  return(cost);
}


/*********************************************************
 *      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 Spin(double g,
                              PhotonStruct * Photon_Ptr)
{
  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;
  double uy = Photon_Ptr->uy;
  double uz = Photon_Ptr->uz;
  double psi,ddump;

  cost = SpinTheta(g);
  sint = sqrt(1.0 - cost*cost);
            /* sqrt() is faster than sin(). */

  psi = 2.0*PI*RandomNum(); /* spin psi 0-2pi. */
  cosp = cos(psi);
  if(psi<PI)
            sinp = sqrt(1.0 - cosp*cosp);
              /* sqrt() is faster than sin(). */
  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*cosp*temp + uz*cost;
  }
}
```

```
/*************************************************************
 *      Move the photon s away in the current layer of medium.
 ****/
void Hop(PhotonStruct * Photon_Ptr,
                              InputStruct * In_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;
  Photon_Ptr->ttotal+=s*In_Ptr->layerspecs.n/CVAC;
}

/*************************************************************
 *      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 ix,iy,iz;
  double mus,mua;

  ix = (short)(Photon_Ptr->x/In_Ptr->dx);
  if(ix>In_Ptr->nx-1) ix=In_Ptr->nx-1;
  if(ix<0) ix=0;

  iy = (short)(Photon_Ptr->y/In_Ptr->dy);
  if(iy>In_Ptr->ny-1) iy=In_Ptr->ny-1;
  if(iy<0) iy=0;

  iz = (short)(Photon_Ptr->z/In_Ptr->dz);
  if(iz>In_Ptr->nz-1) iz=In_Ptr->nz-1;
  if(iz<0) iz=0;

  mua = In_Ptr->layerspecs.mua[ix][iy][iz];
```

```
  mus = In_Ptr->layerspecs.mus[ix][iy][iz];

  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 the boundary.
 *      Return 1 if hit boundary.
 *      Return 0 otherwise.
 *
 *      If the projected step hits the boundary, the members
 *      s and sleft of Photon_Ptr are updated.
 ****/
Boolean HitBoundary(PhotonStruct *  Photon_Ptr,

             InputStruct  *  In_Ptr)
{
  double mut;
  double dl_b;  /* length to boundary. */
  double uz = Photon_Ptr->uz;
  Boolean hit;
  short ix,iy,iz;


  /* Distance to the boundary. */
  if(uz>0.0)
             dl_b = (In_Ptr->layerspecs.z1
                                        - Photon_Ptr->z)/uz;     /*
dl_b>0. */
  else if(uz<0.0)
             dl_b = (In_Ptr->layerspecs.z0
                                        - Photon_Ptr->z)/uz;     /*
dl_b>0. */
  if(uz != 0.0 && Photon_Ptr->s > dl_b) {
                /* not horizontal & crossing. */
                ix = (short)(Photon_Ptr->x/In_Ptr->dx);
                if(ix>In_Ptr->nx-1) ix=In_Ptr->nx-1;
                if(ix<0) ix=0;

                iy = (short)(Photon_Ptr->y/In_Ptr->dy);
                if(iy>In_Ptr->ny-1) iy=In_Ptr->ny-1;
                if(iy<0) iy=0;

                iz = (short)(Photon_Ptr->z/In_Ptr->dz);
                if(iz>In_Ptr->nz-1) iz=In_Ptr->nz-1;
                if(iz<0) iz=0;

                mut = In_Ptr->layerspecs.mua[ix][iy][iz]+In_Ptr-
>layerspecs.mus[ix][iy][iz];

             Photon_Ptr->sleft = (Photon_Ptr->s - dl_b)*mut;
             Photon_Ptr->s     = dl_b;
             hit = 1;
  }
  else
             hit = 0;

  return(hit);
}
/*************************************************************
 *    Update A_xyzt if photon hits detector
 *
 */
void HitDetector(InputStruct * In_Ptr,
                                          PhotonStruct * Photon_Ptr,
                                          OutStruct * Out_Ptr)
{

  short ix, iy, iz, it;

  it = (short)(Photon_Ptr->ttotal/In_Ptr->dt);
  if(it>In_Ptr->nt-1) it=In_Ptr->nt-1;
  if(it<0) it=0;

  Out_Ptr->Det_Photons[it]++;
  Out_Ptr->Det_Int[it] += Photon_Ptr->w;
  for(ix=0; ix<In_Ptr->nx; ix++)
```

```
            for(iy=0; iy<In_Ptr->ny; iy++)
                        for(iz=0; iz<In_Ptr->nz; iz++)
                                    Out_Ptr-
>A_xyzt[ix][iy][iz][it] +=
                                        Photon_Ptr->w*Out_Ptr-
>A_xyz[ix][iy][iz];
  }

/************************************************************
 *      Drop photon weight inside the tissue (not glass).
 *
 *  The photon is assumed not dead.
 *
 *      The weight drop is dw = w*mua/(mua+mus).
 *
 *      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 mua, mus;
  short ix,iy,iz,it;

  ix = (short)(Photon_Ptr->x/In_Ptr->dx);
  if(ix>In_Ptr->nx-1) ix=In_Ptr->nx-1;
  if(ix<0) ix=0;

  iy = (short)(Photon_Ptr->y/In_Ptr->dy);
  if(iy>In_Ptr->ny-1) iy=In_Ptr->ny-1;
  if(iy<0) iy=0;

  iz = (short)(Photon_Ptr->z/In_Ptr->dz);
  if(iz>In_Ptr->nz-1) iz=In_Ptr->nz-1;
  if(iz<0) iz=0;

  it = (short)(Photon_Ptr->ttotal/In_Ptr->dt);
  if(it>In_Ptr->nt-1) it=In_Ptr->nt-1;
  if(it<0) it=0;

  mua = In_Ptr->layerspecs.mua[ix][iy][iz];
  mus = In_Ptr->layerspecs.mus[ix][iy][iz];

  dwa = Photon_Ptr->w * mua/(mua+mus);
  Photon_Ptr->w -= dwa;

  /* assign dwa to the absorption array element. */
  Out_Ptr->A_xyz[ix][iy][iz]++;                         /* One instead
of 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 a1
 *      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 n1,      /* incident refractive index.*/
                                                        double n2,
/* transmit refractive index.*/
                                                        double ca1,
/* cosine of the incident */

                            /* angle. 0<a1<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 = 0.0;
  }
  else if(ca1>COSZERO) {        /** normal incident. **/
             *ca2_Ptr = ca1;
             r = (n2-n1)/(n2+n1);
             r *= r;
  }
  else if(ca1<COS90D)  {        /** very slant. **/
             *ca2_Ptr = 0.0;
             r = 1.0;
  }
  else  {                                       /** general. **/
             double sa1, sa2;
               /* sine of the incident and transmission angles. */
             double ca2;

             sa1 = sqrt(1-ca1*ca1);
             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+ = cc - ss. */
               cam = ca1*ca2 + sa1*sa2; /* c- = cc + ss. */
               sap = sa1*ca2 + ca1*sa2; /* s+ = sc + cs. */
               sam = sa1*ca2 - ca1*sa2; /* s- = sc - cs. */
               r = 0.5*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                     Refl,   /* reflectance. */
                                          InputStruct  * In_Ptr,
                                          PhotonStruct * Photon_Ptr,
                                          OutStruct *    Out_Ptr)
{
  short ix,iy,iz;

  ix = (short)(Photon_Ptr->x/In_Ptr->dx);
  if(ix>In_Ptr->nx-1) ix=In_Ptr->nx-1;
  if(ix<0) ix=0;

  iy = (short)(Photon_Ptr->y/In_Ptr->dy);
  if(iy>In_Ptr->ny-1) iy=In_Ptr->ny-1;
  if(iy<0) iy=0;

  iz = 0;

    /* add abs. array to result array if photon hits detector. */
  if(ix==In_Ptr->Detx && iy==In_Ptr->Dety && !In_Ptr->TR)  {
             Out_Ptr->A_xyz[ix][iy][iz]++;
             HitDetector(In_Ptr, Photon_Ptr, Out_Ptr);
             }
  SetAbsZero(In_Ptr, Out_Ptr); /* set A_xyz to zero */
  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
 *      transmittance array.
 *
 *      Update the photon weight as well.
 ****/
void RecordT(double             Refl,
                                InputStruct  * In_Ptr,
                                PhotonStruct * Photon_Ptr,
                                OutStruct *    Out_Ptr)
{
  short ix,iy,iz;

  ix = (short)(Photon_Ptr->x/In_Ptr->dx);
  if(ix>In_Ptr->nx-1) ix=In_Ptr->nx-1;
  if(ix<0) ix=0;

  iy = (short)(Photon_Ptr->y/In_Ptr->dy);
  if(iy>In_Ptr->ny-1) iy=In_Ptr->ny-1;
  if(iy<0) iy=0;

  iz = In_Ptr->nz-1;

    /* add abs. array to result array if photon hits detector. */
  if(ix==In_Ptr->Detx && iy==In_Ptr->Dety && In_Ptr->TR)  {
             Out_Ptr->A_xyz[ix][iy][iz]++;
             HitDetector(In_Ptr, Photon_Ptr, Out_Ptr);
             }
  SetAbsZero(In_Ptr, Out_Ptr); /* set A_xyz to zero */
  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 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 the "layer" is not the first layer and the photon
 *      packet is transmitted, move the photon to "layer-1".
 *
 *      Update the photon parmameters.
 ****/
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=0.0; /* reflectance */
  double ni = In_Ptr->layerspecs.n;
  double nt = In_Ptr->layerspecs.nambient;

  /* Get r. */
  if( - uz <= In_Ptr->layerspecs.cos_crit0)
             r=1.0;               /* total internal reflection. */
  else
             r = RFresnel(ni, nt, -uz, &uz1);

  if(RandomNum() > r) {         /* transmitted to layer-1. */
              Photon_Ptr->uz = -uz1;
              RecordR(0.0, In_Ptr, Photon_Ptr, Out_Ptr);
              Photon_Ptr->dead = 1;
  }
  else                                          /* reflected. */
             Photon_Ptr->uz = -uz;
}

/*************************************************************
 *      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 layer, record the
 *      transmitted weight as transmittance. See comments for
 *      CrossUpOrNot.
 *
```

```
 *      Update the photon parmameters.
 ****/
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 */
  double ni = In_Ptr->layerspecs.n;
  double nt = In_Ptr->layerspecs.nambient;

  /* Get r. */
  if( uz <= In_Ptr->layerspecs.cos_crit1)
             r=1.0;          /* total internal reflection. */
  else
             r = RFresnel(ni, nt, uz, &uz1);

  if(RandomNum() > r) {          /* transmitted to layer+1. */
               Photon_Ptr->uz = uz1;
               RecordT(0.0, In_Ptr, Photon_Ptr, Out_Ptr);
               Photon_Ptr->dead = 1;
  }
  else                                          /* reflected. */
             Photon_Ptr->uz = -uz;
}

/*****************************************************
 ****/
void CrossOrNot(InputStruct  *  In_Ptr,
                                                      PhotonStruct *
Photon_Ptr,
                                                      OutStruct    *
Out_Ptr)
{
  if(Photon_Ptr->uz < 0.0)
             CrossUpOrNot(In_Ptr, Photon_Ptr, Out_Ptr);
  else
             CrossDnOrNot(In_Ptr, Photon_Ptr, Out_Ptr);
}

/*****************************************************
```

```
 *      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)
{
  StepSizeInTissue(Photon_Ptr, In_Ptr);

  if(HitBoundary(Photon_Ptr, In_Ptr)) {
             Hop(Photon_Ptr,In_Ptr); /* move to boundary plane. */
             CrossOrNot(In_Ptr, Photon_Ptr, Out_Ptr);
  }
  else {
             Hop(Photon_Ptr,In_Ptr);
             Drop(In_Ptr, Photon_Ptr, Out_Ptr);
             Spin(In_Ptr->layerspecs.g,
                          Photon_Ptr);
  }
}

/*****************************************************
 ****/
void HopDropSpin(InputStruct  *  In_Ptr,
                                                      PhotonStruct *
Photon_Ptr,
                                                      OutStruct    *
Out_Ptr)
{ short ix, iy, iz;

  HopDropSpinInTissue(In_Ptr, Photon_Ptr, Out_Ptr);
```

```
  if( Photon_Ptr->w < In_Ptr->Wth && !Photon_Ptr->dead)
            Roulette(Photon_Ptr);
  if(Photon_Ptr->dead)
            SetAbsZero(In_Ptr, Out_Ptr);
}
```

B.2.5 mcmatnr.c

```
/*************************************************************
 *  Copyright Univ. of Texas M.D. Anderson Cancer Center
 *  1992.
 *
 *            Some routines modified from Numerical Recipes in C,
 *            including error report, array or matrix declaration
 *            and releasing.
 ****/
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

/*************************************************************
 *            Report error message to stderr, then exit the program
 *            with signal 1.
 ****/
void nrerror(char error_text[])

{
  fprintf(stderr,"%s\n",error_text);
  fprintf(stderr,"...now exiting to system...\n");
  exit(1);
}

/*************************************************************
 *            Allocate an array with index from nl to nh inclusive.
 *
 *            Original matrix and vector from Numerical Recipes in C
 *            don't initialize the elements to zero. This will
 *            be accomplished by the following functions.
 ****/
double *AllocVector(short nl, short nh)
{
  double *v;
  short i;

  v=(double *)malloc((unsigned) (nh-nl+1)*sizeof(double));
  if (!v) nrerror("allocation failure in vector()");

  v -= nl;
  for(i=nl;i<=nh;i++) v[i] = 0.0;        /* init. */
  return v;
}

/*************************************************************
 *            Allocate a matrix with row index from nrl to nrh
 *            inclusive, and column index from ncl to nch
 *            inclusive.
 ****/
double **AllocMatrix(short nrl,short nrh,

short ncl,short nch)
{
  short i,j;
  double **m;

  m=(double **) malloc((unsigned) (nrh-nrl+1)

            *sizeof(double*));
  if (!m) nrerror("allocation failure 1 in matrix()");
  m -= nrl;

  for(i=nrl;i<=nrh;i++) {
    m[i]=(double *) malloc((unsigned) (nch-ncl+1)

            *sizeof(double));
    if (!m[i]) nrerror("allocation failure 2 in matrix()");
    m[i] -= ncl;
  }

  for(i=nrl;i<=nrh;i++)
    for(j=ncl;j<=nch;j++) m[i][j] = 0.0;
  return m;
}
/*************************************************************/

double ***Alloc3Matrix(short nrl,short nrh,
```

```
            short ncl,short nch,

            short ntl,short nth)
{
  short i,j,k;
  double ***m;

  m=(double ***) malloc((unsigned) (nrh-nrl+1)

            *sizeof(double**));
  if (!m) nrerror("allocation failure 1 in matrix3()");
  m -= nrl;

  for(i=nrl;i<=nrh;i++) {
            m[i]=(double **) malloc((unsigned) (nch-ncl+1)

            *sizeof(double*));
            if (!m[i]) nrerror("allocation failure 2 in matrix3()");
            m[i] -= ncl;
  }

  for(i=nrl;i<=nrh;i++) {
            for(j=ncl;j<=nch;j++){
                        m[i][j]=(double *) malloc((unsigned) (nth-
ntl+1)

            *sizeof(double));
                        if (!m[i][j]) nrerror("allocation failure 3
in matrix3()");
                        m[i][j] -= ntl;
            }
  }

  for(i=nrl;i<=nrh;i++)
            for(j=ncl;j<=nch;j++)
                        for(k=ntl;k<=nth;k++)
                                    m[i][j][k] = 0.0;
  return m;
}

double ****Alloc4Matrix(short nrl,short nrh,

            short ncl,short nch,

            short ntl,short nth,

            short nll,short nlh)
{
  short i,j,k,l;
  double ****m;

  m=(double ****) malloc((unsigned) (nrh-nrl+1)

            *sizeof(double***));
  if (!m) nrerror("allocation failure 1 in matrix4()");
  m -= nrl;

  for(i=nrl;i<=nrh;i++) {
            m[i]=(double ***) malloc((unsigned) (nch-ncl+1)

            *sizeof(double**));
            if (!m[i]) nrerror("allocation failure 2 in matrix4()");
            m[i] -= ncl;
  }

  for(i=nrl;i<=nrh;i++) {
            for(j=ncl;j<=nch;j++){
                        m[i][j]=(double **) malloc((unsigned) (nth-
ntl+1)

            *sizeof(double*));
                        if (!m[i][j]) nrerror("allocation failure 3
in matrix4()");
                        m[i][j] -= ntl;
            }
  }

  for(i=nrl;i<=nrh;i++) {
            for(j=ncl;j<=nch;j++){
                        for(k=ntl;k<=nth;k++){
                                    m[i][j][k]=(double *)
malloc((unsigned) (nlh-nll+1)

            *sizeof(double));
                                    if (!m[i][j][k])
nrerror("allocation failure 4 in matrix4()");
                                    m[i][j] -= ntl;
```

```
                        }
                }
        }

        for(i=nrl;i<=nrh;i++)
                for(j=ncl;j<=nch;j++)
                        for(k=ntl;k<=nth;k++)
                                for(l=nll;l<=nlh;l++)
                                        m[i][j][k][l] =
0.0;
        return m;
}

/*************************************************************
 *              Release the memory.
 ****/
void FreeVector(double *v,short nl,short nh)
{
        free((char*) (v+nl));
}

/*************************************************************
 *              Release the memory.
 ****/
void FreeMatrix(double **m,short nrl,short nrh,
                                        short ncl,short
nch)
{
        short i;

        for(i=nrh;i>=nrl;i--) free((char*) (m[i]+ncl));
        free((char*) (m+nrl));
}

/*************************************************************
 *              Release the memory.
 ****/
void Free3Matrix(double ***m,short nrl,short nrh,
                                        short ncl,short
nch,short ntl,short nth)
{
        short i,j;

        for(i=nrh;i>=nrl;i--)
                for(j=nch;j>=ncl;j--)
                        free((char*) (m[i][j]+ntl));

        for(i=nrh;i>=nrl;i--) free((char*) (m[i]+ncl));
        free((char*) (m+nrl));
}

void Free4Matrix(double ****m,short nrl,short nrh,
                                                short ncl,short
nch,short ntl,short nth,short nll, short nlh)
{
        short i,j,k;

        for(i=nrh;i>=nrl;i--)
                for(j=nch;j>=ncl;j--)
                        for(k=nth;k>=ntl;k--)
                                free((char*)
(m[i][j][k]+nll));

        for(i=nrh;i>=nrl;i--)
                for(j=nch;j>=ncl;j--)
                        free((char*) (m[i][j]+ntl));

        for(i=nrh;i>=nrl;i--)
                free((char*) (m[i]+ncl));

                free((char*) (m+nrl));
}
```

B.2.6 Template Input File for MCMAT

```
####
# Template of input files for Monte Carlo simulation (mcmat).
# Anything in a line after "#" is ignored as comments.
# Space lines are also ignored.
# Lengths are in cm, mua and mus are in 1/cm.
#
# The 'save_intermediate' parameter should be set to 1
# for long runs, to prevent all data from being lost
# in case of computer power loss etc. For shorter runs
# when testing this is not necessary and slows down the
```

```
# program.
####

1.0                                          # file version
1                                            # number of runs

### Specify data for run 1
test.mco        A                           # output filename,
ASCII/Binary
50000                                         # No. of photons
0.1       0.1       0.1     25E-12           # dx, dy, dz, dt
10        10        10        20                  # No. of dx, dy,
dz & dt.
0.5       0.5       0.0                      # In x, y, z.
5             5             1
              # Detx, Dety, TR: 1=tr.,0=ref.
5             1             0
              # wrx, width, save int. if 1

# n     mua     mus     g       d             # One line for each
layer
1.4                                          # n for medium above.
1.4     1.0     10.0      0.0     1           # layer
# Regions
0
              # No. of regions
50    10000                                                  #
mus, mua
0
              # 0=cube, 1=sphere, 2=cyl.
0 7 0 9 9 9                                                  #
x1, y1, z1, x2, y2, z2.
```