University of Twente

# From Diffusion of Light to Free-Form Scattering

Bachelor Thesis
Applied Physics and Applied Mathematics

**K.W. Fokkema**

Supervisors:
M.L. Meretska M.Sc. (COPS)
Dr. M. Schlottbom (SACS)
Prof. Dr. W.L. Vos (COPS)
August 22, 2017

## Abstract

In this thesis, a Monte Carlo simulation is presented for radiative transfer through free-form scattering materials. It is validated using diffusion theory in the case of a slab and used to examine the effect of a corrugation on the slab. The corrugation changes the angular distribution of transmitted light and the total transmission.

# Contents

# 1 Introduction

As the most efficient source of ambient light, white LEDs are widely used and of importance for everyday life and biological and medical applications. Therefore, it is important to thoroughly understand the physical processes that are present in white LEDs. A white LED typically consists of a blue LED surrounded by a layer of fluorescent material (phosphor) [1]. This layer absorbs a part of the light emitted by the blue LED and emits light with longer wavelengths (green and red). The part of the blue light that was not absorbed contributes together with the green and red light to the spectrum of the white LED. After the light comes out of the phosphor, free-form optical elements are used to change the angular distribution of the light. It is interesting to investigate whether the fluorescent material and the free-form optical elements can be combined by changing the shape of the interfaces of the fluorescent material.

In this thesis, light propagation through scattering elements with curved interfaces is investigated by means of numerical methods. The results can be used as a starting point for further research. In chapter 2, all necessary background knowledge is discussed, including light transport by means of the radiative transfer equation, the diffusion approximation and the principles of a Monte Carlo simulation. In chapter 3, the implementation and structure of the Monte Carlo simulation that was written in C++ is discussed. In chapter 4, the code is validated by means of comparison to the diffusion approximation and in chapter 4.2, the results of the program are analyzed for a slab with curved boundaries.

# 2 Theory

When light propagates in a medium filled with inhomogeneities of the order of the wavelength of the light, a *scattering medium*, it can be modeled as a photon. These photons travel in straight lines and can interact with the inhomogeneities by scattering and absorption. Furthermore, when photons move between media with different refractive indices, they can be reflected and refracted.

In this chapter, first an overview is given of all possible interactions between a photon and a scattering medium in sections 2.1 and 2.2. In section 2.3 the radiative transfer equation is shown, which describes the propagation of light in a scattering medium. The radiative transfer equation can be approximated in the limit of multiple scattering by the diffusion approximation (section 2.4). The slab geometries that are used in the thesis are defined in section 2.5 and the diffusion equation is applied to a slab in section 2.6. Finally, the principles of a Monte Carlo simulation are explained in section 2.7 and a method for sampling random variables with a probability density function is shown in section 2.8.

## 2.1 Scattering and absorption

A scattering medium is modeled as a homogeneous medium with small inhomogeneities (particles) that are randomly distributed. The probability per unit length for a photon to scatter is given by the scattering coefficient

$$\mu_s = n_s \sigma_s, \tag{1}$$

where $n_s$ is the number density of the scatterers in the medium and $\sigma_s$ is the scattering cross sections of those scatterers, a measure for how efficiently the particles scatter the light [2]. The scattering mean free path $l_s$, the average distance a photon travels before it is scattered, is given by

$$l_s = 1/\mu_s. \tag{2}$$

Similar to scattering, the probability per unit length for a photon to be absorbed is given by the absorption coefficient

$$\mu_a = n_a \sigma_a \tag{3}$$

where $n_a$ is the number density of the absorbers in the medium and $\sigma_a$ is the absorption cross section, a measure for how efficiently the particles absorb the light. The absorption mean free path $l_a$, the average distance a photon travels before it is absorbed, is given by

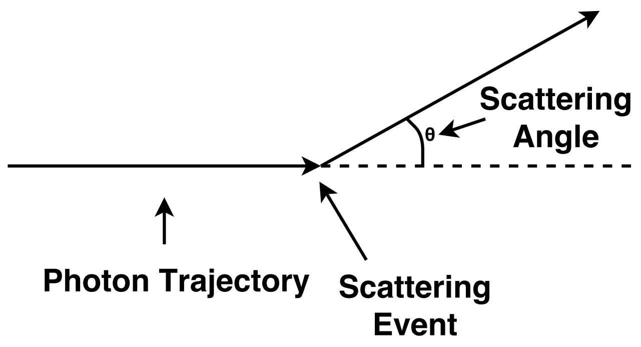


Figure 1: Schematic representation of the scattering angle $\theta$

$$l_a = 1/\mu_a. \tag{4}$$

When a beam of light propagates through a scattering medium where both absorption and scattering are present, its intensity decreases according to the Beer-Lambert law given by

$$T(x) = e^{-(\mu_s+\mu_a)x} = e^{-s}. \tag{5}$$

The variable $s = (\mu_s + \mu_a)x$ is called the optical depth.

When a photon scatters, it is deflected from its straight path by the *scattering angle* $\theta \in [0, \pi]$ (see Fig.1). The direction of this deflection in the plane perpendicular to the direction of propagation is given by the *azimuthal angle* $\phi \in [0, 2\pi)$. The anisotropy $g$ is given by the average cosine of the scattering angle

$$g = \langle \cos\theta \rangle \,. \tag{6}$$

In the case of isotropic scattering, the direction of propagation in which a photon travels after scattering is uniformly distributed on the unit circle, which means that $g = 0$. If the anisotropy is close to one, predominant forward scattering is observed. The *transport mean free path* $l$ is a measure for the distance over which the propagation direction of the photon is randomized. It is given by

$$l = \frac{l_s}{1-g}. \tag{7}$$

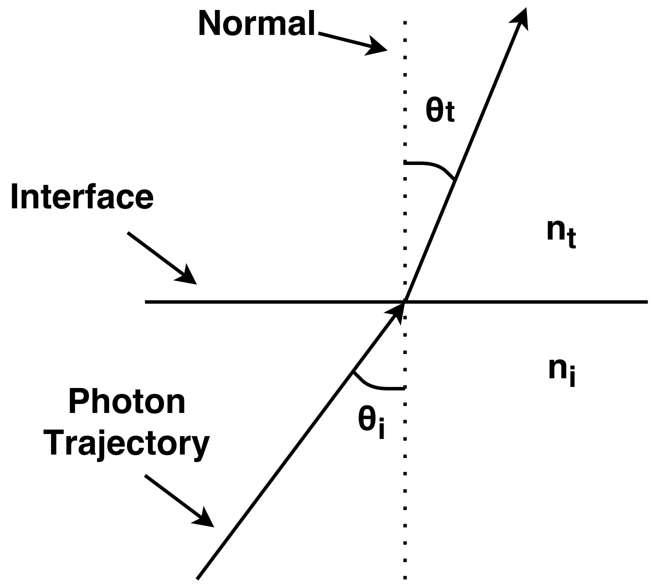


Figure 2: Schematic representation of Snell's law at an interface between two media with refractive indices $n_i$ and $n_t$

## 2.2 Interfaces

Interfaces between media are modeled by the Fresnel equations, which depend on the refractive indices of the material, the polarization of the light and the angle of incidence. Reflectance is the probability that a photon is reflected when it propagates from a medium with refractive index $n_i$ to refractive index $n_t$ with incidence angle $\theta_i$. For s-polarization, the reflectance is given by [3]:

$$R_s = \left| \frac{n_i \cos\theta_i - n_t \cos\theta_t}{n_i \cos\theta_i + n_t \cos\theta_t} \right|^2 \tag{8}$$

and for p-polarization the reflectance is

$$R_p = \left| \frac{n_i \cos\theta_t - n_t \cos\theta_i}{n_i \cos\theta_t + n_t \cos\theta_i} \right|^2 . \tag{9}$$

In these equations, $\theta_t$ is the angle of refraction (see Fig. 2), given by Snell's law:

$$n_i \sin\theta_i = n_t \sin\theta_t. \tag{10}$$

If the polarization of the light is random, the average reflectance is equal to

$$R = (R_s + R_p)/2. \tag{11}$$

## 2.3 Radiative Transfer Equation

The radiative transfer equation (RTE) describes the propagation of the intensity of light. The variable in the RTE is the radiance $L$, which is energy flux per unit normal area per unit solid angle per unit time with units $\mathrm{Wm^{-2}sr^{-1}}$. Here the normal area is perpendicular to the direction of the flow [4]. The amount of energy $dE$ transported across an area $dA$ within a solid angle $d\Omega$ in a time $dt$ is then given by

$$dE = L(\vec{r}, \hat{s}, t)\,(\hat{s} \cdot \hat{n})\,dA\,d\Omega\,dt \quad \text{(J)}. \tag{12}$$

Here $\vec{r}$ is the position, $\hat{s}$ is the unit vector that specifies the direction, t is the time and $\hat{n}$ is the unit vector normal to the surface $dA$. The RTE can be derived from energy conservation [4] and is given by:

$$\begin{aligned} \frac{\partial L(\vec{r}, \hat{s}, t)}{\partial t} =& -\hat{s} \cdot \nabla L(\vec{r}, \hat{s}, t) - n\sigma L(\vec{r}, \hat{s}, t) \\ & + n_s\sigma_s \int_{4\pi} L(\vec{r}, \hat{s}', t) \cdot p(\hat{s}' \cdot \hat{s}) d\Omega' + S(\vec{r}, \hat{s}, t). \end{aligned} \tag{13}$$

The term on the left hand side indicates the change in energy per unit time. On the right hand side, the first term describes the diffusion of light, the second term indicates the energy that is redirected from direction $\hat{s}$ by scattering and by absorption, and the third term indicates the energy redirected to direction $\hat{s}$ by scattering from other directions $\hat{s}'$. The function $p(\hat{s}' \cdot \hat{s})$ in that term is the probability distribution of $\cos\theta$, and is often modeled by the Henyey-Greenstein function [5]:

$$p(\cos\theta) = \frac{1}{2} \frac{1 - g^2}{[1 + g^2 - 2g\cos\theta]^{\frac{3}{2}}}. \tag{14}$$

## 2.4 Diffusion Equation

The RTE is difficult to solve since it has 6 independent variables. Therefore, it is often simplified to the diffusion equation [4]. The physical quantity typically measured in experiments is the fluence rate $\Phi$, the energy flow per unit area per unit time [4]. It can be obtained from the radiance by integrating over all solid angles:

$$\Phi(\vec{r},t) = \int_{4\pi} L(\vec{r},\hat{s},t)\, d\Omega \quad (\mathrm{Wm}^{-2}). \tag{15}$$

In the absence of absorption, the diffusion equation is then given by [6]:

$$\frac{\partial \Phi(\vec{r},t)}{c\partial t} - D\nabla^2 \Phi(\vec{r},t) = S(\vec{r},t). \tag{16}$$

Here $S$ is a source of diffuse light and $D$ is the *diffusion coefficient* that is given by:

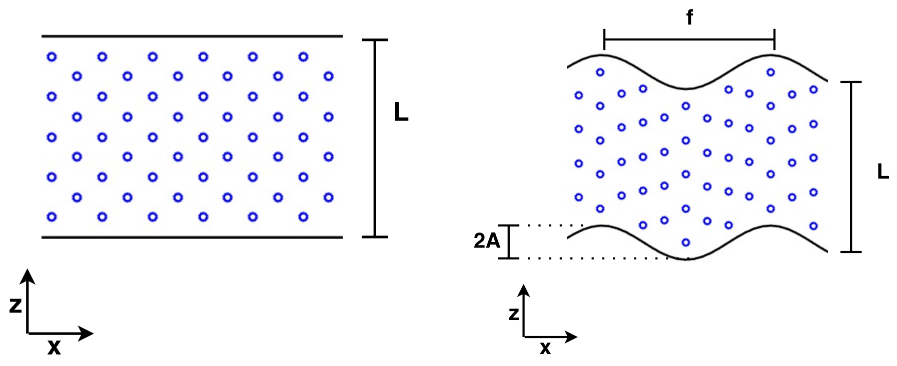
$$D = \frac{l}{3} \tag{17}$$

in the absence of absorption. In the derivation of the diffusion approximation, it is assumed that the radiance is isotropic everywhere, which is why Eq. 16 does not contain $\hat{s}$.

## 2.5 Slab Geometry

The sample geometries analyzed in this thesis are the (regular) *slab* and the *corrugated slab*, which both consist of a homogeneous scattering medium. A slab of thickness $L$ is modeled to be infinite in the $x$- and $y$-directions and bounded by $z = 0$ and $z = L$ in the $z$-direction (Fig.3a). The slab is surrounded by vacuum on both sides. The scattering mean free path in the slab is given by $l_s$. A corrugated slab is one of the simplest sample structures in the emergent topic of "free-form scattering optics". In a corrugated slab with amplitude $A$ and frequency $f$, the equations for the boundaries are replaced by $z = A\sin(2\pi f x)$ and $z = L + A\sin(2\pi f x)$ (Fig.3b).

Light is incident on the slabs from $z = -\infty$, its propagation direction parallel to the $z$-axis. Light is transmitted or reflected through a slab. The total transmission $T_t$ is divided into *ballistic light* (photons that did not scatter) and *diffuse transmission* (photons that did scatter). Likewise, total reflection $R_t$ is divided into *specular reflection* of photons that are Fresnel reflected and *diffuse reflection* for backscattered photons.



(a) (b)

Figure 3: (a) A slab of thickness $L$ (b) a corrugated slab of thickness $L$, with both surfaces modulated by an amplitude of $A$ and a frequency of $f$. Note that f is the frequency and not the period of the corrugation.

## 2.6 Diffusion in a Slab

For a slab, light propagation in the limit of multiple scattering can be calculated analytically using the diffusion equation (Eq. 16). The diffusion theory results in the following equation for the total transmission [6]:

$$T_t = \frac{l + z_{e_1}}{L + z_{e_1} + z_{e_2}}. \tag{18}$$

Here, $z_{e_1}$ and $z_{e_2}$ denote the *extrapolation length* of the first and second surface respectively (as seen from $z = -\infty$). The extrapolation length depends linearly on $l$ and can be computed as the ratio of the refractive indices of the material and the air [6]. Here, $z_{e_1} = z_{e_2} = z_e$ because both interfaces are bounded by the same media.

In absence of absorption, the reflection and transmission are related by

$$R_t = 1 - T_t. \tag{19}$$

Using diffusion theory, the angular distribution of transmitted light can also be calculated. This distribution is called the *escape function* $E$ and assuming light is polarized randomly, it is given by [6]:

$$E(\mu_a) = \frac{3}{2}\left(\frac{n_a}{n_m}\right)^2 \mu_a \left(\frac{z_e}{l} + \mu_m\right)[1 - R(\mu_m)], \tag{20}$$

here $n_m$ is the refractive index of the material and $n_a$ the refractive index of the air, $\mu_a = \cos\theta_a$ and $\mu_m = \cos\theta_m$, where $\theta_a$ and $\theta_m$ are the exit and incidence angles, respectively, with respect to the normal of the surface. $R(\mu_m)$ is the Fresnel reflection coefficient given by Eq. 11.

## 2.7 Monte Carlo

Besides the diffusion approximation, another method of acquiring a solution to the RTE is a *Monte Carlo* simulation. A Monte Carlo simulation relies on the sampling of random numbers to obtain numerical results [7] and it can solve the RTE with any desired accuracy, assuming the computational load is affordable [8].

In the case of radiative transfer, single photons are tracked as they move through the medium and random numbers are sampled to determine scattering lengths, scattering angles and transmission and reflection at boundaries. In this thesis, polarization of light, absorption and interference will be neglected.

## 2.8 Inverse Transform Sampling

In Monte Carlo simulations for radiative transfer, it is necessary to sample random variables (scattering angles and the lengths of scattering paths) with a certain probability density function [4]. *Inverse Transform Sampling* is a basic method for sampling variables with a probability density function $f$ given its cumulative distribution function $F$. In this method, a random number $\xi$ is taken from a uniform distribution on the interval [0,1]. A random number from the distribution $f$ is then given by $x = F^{-1}(\xi)$, since the cumulative distribution function of $F^{-1}(\xi)$ is given by $F$:

$$P(F^{-1}(\xi) < x) = P(\xi < F(x)) = F(x) \tag{21}$$

Inverse Transform Sampling is useful because it is an efficient method to sample random variables according to a probability distribution, when the cumulative distribution function is invertible.

# 3 Implementation

In this section, the implementation of the Monte Carlo code for scattering in a 3-dimensional medium with curved interfaces is explained. The structure of the code and the used equations are heavily based on the code 'MCML' [4]. In this section, $\xi$ denotes a random variable from the uniform distribution on the interval $[0, 1]$.

In the simulation, single photons are tracked as they travel through the scattering medium and its environment. The main properties associated with the photon are its step size $s$, which indicates the optical depth it is going to travel before scattering, its position $\hat{r}$ and its propagation direction $\hat{v}$, where $|\hat{v}| = 1$. The environment through which the photon travels consists of a convex domain bounded by edges (see Fig. 4). When a photon reaches an edge, it is terminated. The domain is divided by interfaces into a scattering medium and its environment. The main properties of the scattering medium are the refractive index $n_m$ and the scattering coefficient $\mu_s$.

The main structure of the program is shown in Fig. 5. In the remaining part of this chapter, various parts of the program will be explained in more detail:

- Sampling the step size $s$ and determining if the photon will cross an interface or reach the edge of the domain (section 3.1). ((2), (3), (4) and (7) in Fig. 5.)
- Interaction with a surface (section 3.2). ((6) in Fig. 5.)
- Scattering of a photon (section 3.3). ((5) in Fig. 5.)

## 3.1 Traveling

Interfaces between the scattering material and the environment are defined by functions of $\hat{r}$, which are 0 at the interface. The function that defines an interface $i$ is given by

$$f_i(\vec{r}) = 0. \tag{22}$$

The implicit function $f_i(\vec{r})$ should be chosen such that its gradient $\nabla f_i(\vec{r})$ is not 0 when $f_i(\vec{r}) = 0$, to avoid division by 0 in the code. The direction of the normal $\hat{n}_i$ of the interface $i$ is given by

$$\hat{n}_i = \frac{\nabla f_i(\vec{r})}{||\nabla f_i(\vec{r})||}. \tag{23}$$

Because the gradient chosen not to be 0 at an interface, the direction of $\hat{n}$ is always to the same side of the interface and the function $f_i(\vec{r})$ changes signs across the interface.

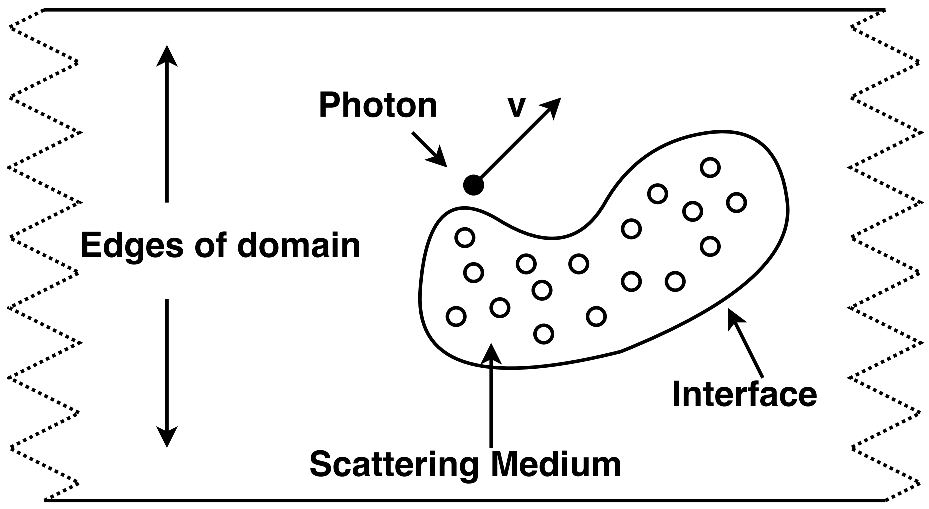


Figure 4: Schematic of the environment in which a photon is tracked, including the edges of a convex domain, the 'material' and 'air' layers and the direction of propagation of a photon.

When a photon travels, it is necessary to determine if it will hit an interface or if it will scatter in the medium. Therefore, the optical depth $s$ to which photon travels is calculated using inverse transform sampling. The cumulative distribution function $F_s$ for $s$ is given by

$$F_s(s) = 1 - T(x), \tag{24}$$

where T(x) is given by the Lambert-Beer probability in Eq. 5. According to the inverse transform sampling method,

$$F_s(s) = 1 - T(x) = 1 - e^{-s} = \xi, \tag{25}$$

where $\xi$ is a uniformly distributed random variable on the interval [0,1]. Inverting this function gives

$$s = -\log(1 - \xi) = -\log(\xi) \tag{26}$$

where $1 - \xi$ is simplified to $\xi$ since $1 - \xi$ also represents a uniformly distributed random variable on the interval [0,1].

The distance that the photon should travel is given by $d = \frac{s}{n\sigma}$. The new position of the photon $\vec{r}_{new} = \vec{r} + d\hat{v}$ is determined, assuming that it does not cross an interface. Then, for each $i$, the signs of $f_i(\vec{r})$ and $f_i(\vec{r}_{new})$ are compared to determine if an interface $i$ is crossed. When the signs are different, an interface

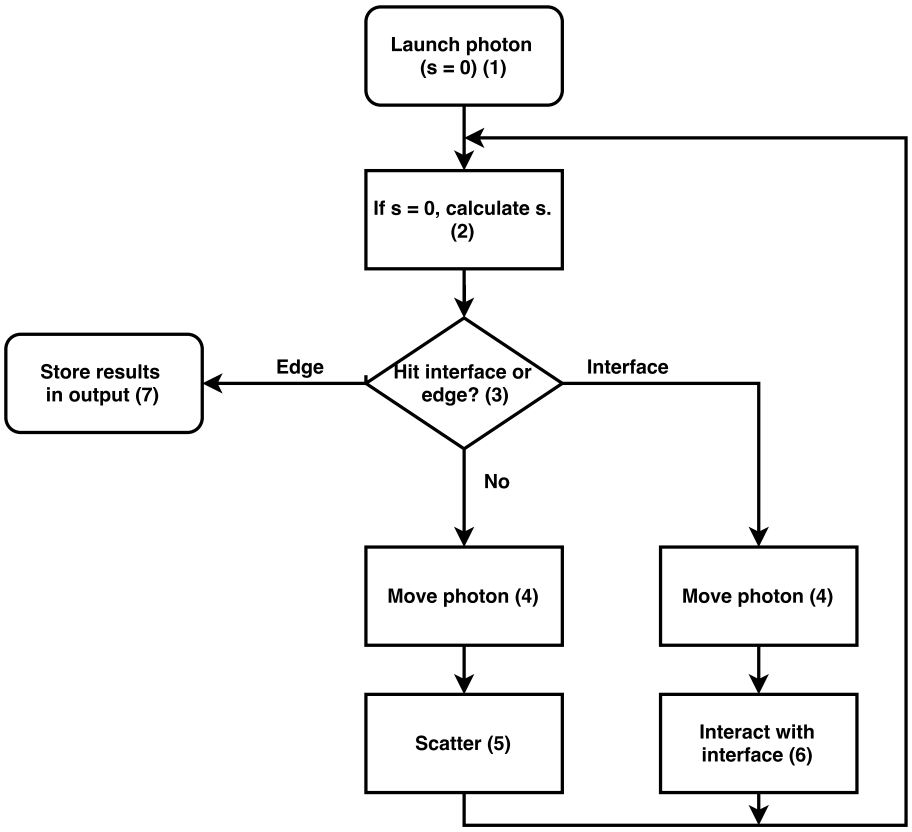

Launch photon (s = 0) (1)
If s = 0, calculate s. (2)
Hit interface or edge? (3)
Edge
Interface
No
Store results in output (7)
Move photon (4)
Move photon (4)
Scatter (5)
Interact with interface (6)

Figure 5: Structure of the program. Here, s is the step size of the photon.

has been crossed (Fig. 6a). When the signs are equal, an interface may have been crossed multiple times (Fig. 6b). To detect whether an interface has been crossed in the case where the signs are equal, the sign of $f_i(\hat{r})$ is calculated at intervals of length $\epsilon$ along the path of the photon to decide whether the interface was crossed multiple times or not at all. The value of $\epsilon$ should be small compared to the typical length scale of the geometry that is studied. When an interface has been crossed, the *false position method* [9] is used to determine $\vec{r}_{b_i}$, the location at which interface $i$ was crossed with an accuracy of $dx$. Finally, the photon travels to the nearest interface or, if no interface has been crossed, it travels the distance $d$. After traveling, the optical distance traveled is subtracted from $s$. When a photon reaches the edge of the domain, its properties are stored as output and a new photon is launched.

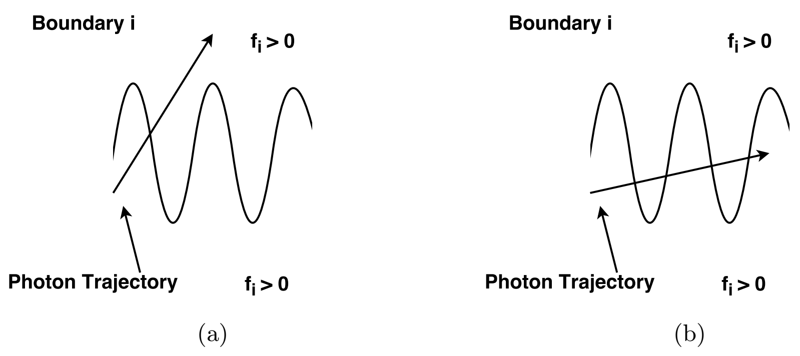


Figure 6: (a) Example of a photon that crosses a curved interface a single time. (b) Example of a photon that crosses a curved interface multiple times.

## 3.2 Interfaces

When a photon hits an interface, the angle of incidence is determined by $\cos\theta = |\hat{v}\cdot\hat{n}|$. Then, the angle of transmission is calculated using Snell's law (Eq. 10) and the probability of reflection is calculated using the Fresnel equations assuming random polarization (Eq. 11). If $\xi < R$, the photon is reflected, otherwise it is transmitted. Then the direction of propagation of the photon is updated. After interacting with the interface, the photon is moved a distance d$x$ away from the interface to avoid rounding errors when $s$ is small.

## 3.3 Scattering

When a photon scatters, the polar scattering angle is calculated by applying inverse transform sampling to the Henyey-Greenstein function given by Eq. 14. Integrating the Henyey-Greenstein function with respect to $\cos\theta$ and equating it to $\xi$ gives

$$\frac{1-g^2}{2g\sqrt{1+g^2-2g\cos\theta}} = \xi \tag{27}$$

Solving this equation for $\cos\theta$ gives

$$\cos\theta = \begin{cases} \frac{1}{2g}\left(1+g^2-\left(\frac{1-g^2}{1-g+2g\xi}\right)^2\right), & \text{if } g \neq 0 \\ 2\xi - 1, & \text{if } g = 0 \end{cases} \tag{28}$$

The azimuthal scattering angle $\phi$ is sampled using

$$\phi = 2\pi\xi \tag{29}$$

These angles are used to determine the new direction of propagation $\hat{v}'$ according to

$$\begin{aligned} v'_x &= \frac{\sin\theta(v_x v_z \cos\phi - v_y \sin\phi)}{\sqrt{1-v_z^2}} + v_x \cos\theta \\ v'_y &= \frac{\sin\theta(v_y v_z \cos\phi + v_x \sin\phi)}{\sqrt{1-v_z^2}} + v_y \cos\theta \\ v'_z &= -\sqrt{1-v_z^2}\sin\theta\cos\theta + v_z\cos\theta, \end{aligned} \tag{30}$$

except for the case when $|v_z| \approx 1$, when the following equations are used to avoid dividing by zero:

$$\begin{aligned} v'_x &= \sin\theta\cos\phi \\ v'_y &= \sin\theta\sin\phi \\ v'_z &= \operatorname{sgn} v_z \cos\theta \end{aligned} \tag{31}$$

## 4 Results

### 4.1 Slab

To validate the Monte Carlo code, the results of Monte Carlo simulations are compared to the diffusion theory (section 2.6) for several cases. First we compare the total reflection to the diffusion theory as a function of $L/l_s$ and $n_m$, then the angular dependence of the diffuse transmission is compared to diffusion theory.

For $n_m = n_a$ and $g = 0$, the total reflection $R_t$ is plotted as a function of $L/l_s$ in Fig. 7. For an optically thick slab ($L/l_s \geq 5$), the diffusion equation approximates $R_t$ within a relative error of 1 percent, but for thin samples ($L/l_s < 5$), the diffusion approximation is not valid because it only holds in the limit of multiple scattering [6].

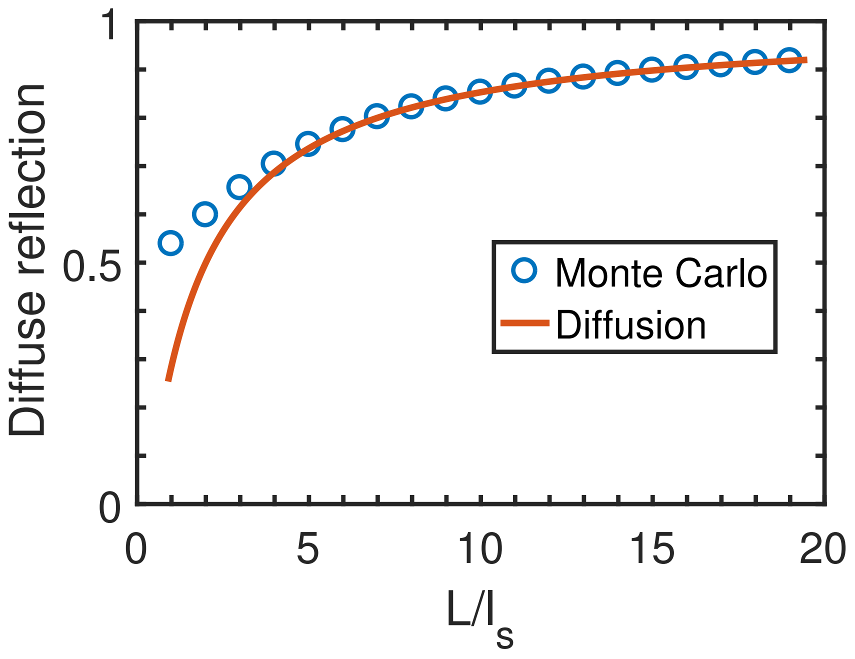


Figure 7: Total reflection as a function of slab width. Here, the refractive index of the slab and its environment are equal, and the anisotropy $g = 0$. The 'Diffusion' results are given by Eq. 19. The error bars of the Monte Carlo results are of the order $10^{-4}$, well within the symbol size.

For $L/l_s = 10$, $n_a = 1$ and $g = 0$, the total reflection and the diffuse reflection are plotted as a function of $n_m$ in Fig. 8. Since specular reflection is not taken into account in the diffusion equation, the diffusion approximation does not represent the total reflection, but the diffuse reflection. The diffusion theory agrees with the diffuse reflection with a rms deviation of 0.0038.

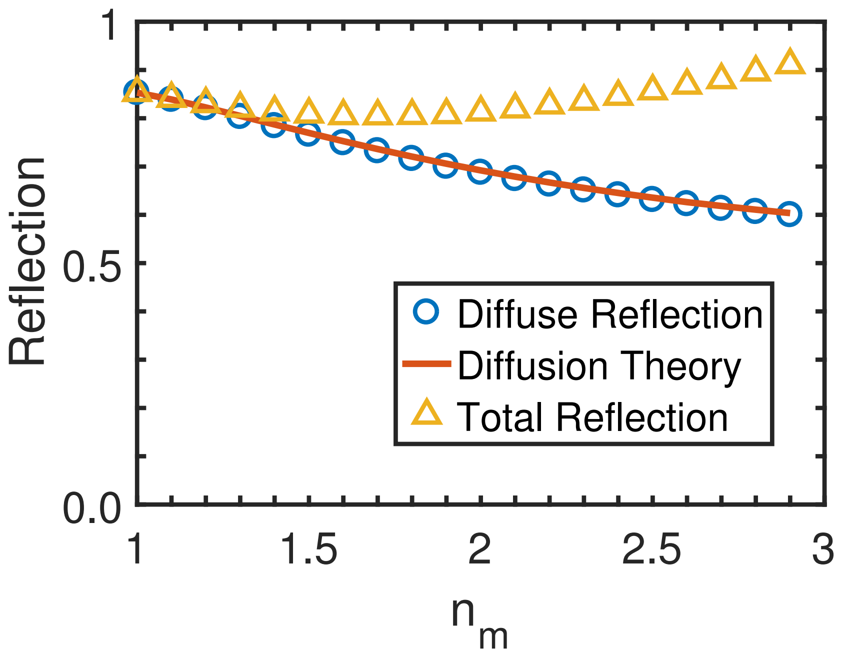


Figure 8: Reflection as a function of the refractive index of the slab, where the refractive index of the environment $n_a = 1$ and the thickness of the slab $L/l_s = 10$. The 'Diffuse Reflection' and the 'Total Reflection' are the results from the Monte Carlo code and the 'Diffusion Theory' is given by Eq. 19. The error bars are of the order $10^{-4}$, well within the symbol size.

Finally, the angular distribution of diffuse transmission is compared to the escape function (Eq. 20). In Fig. 9, the diffuse transmission is compared to the escape function in the case of a large thickness ($L/l_s = 10$) and in the absence of anisotropy. The results match well, with a rms deviation of 0.03. When for the same parameters the anisotropy is increased, the diffuse transmission deviates from the escape function since the transport mean free path is large compared to the slab and the assumption of isotropic propagation of light does not hold. An example is given in Fig. 10 for $g = 0.9$.

We have shown here that the code for Monte Carlo simulations for light transport agree well with the diffusion theory in the case of a small transport mean free path.

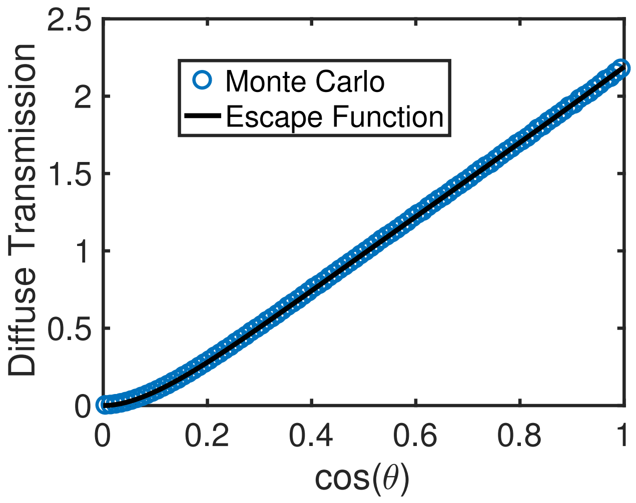


Figure 9: Diffuse transmission as a function of angle. Here, the width $L = 1$, the anisotropy $g = 0$, the refractive index of the slab is $n_m = 1.49$, the refractive index of the environment is $n_a = 1$ and the scattering length is $l_s = 0.1$. The 'Escape Function' is given by Eq. 20. The error bars of the Monte Carlo results are of the order $10^{-4}$, well within the symbol size.

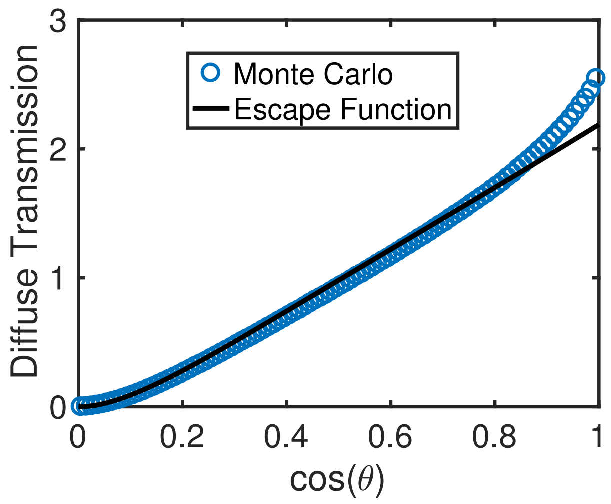


Figure 10: Diffuse transmission as a function of angle. Here, the width $L = 1$, the anisotropy $g = 0.9$, the refractive index of the slab is $n_m = 1.49$. The 'Escape Function' is given by Eq. 20. The error bars of the Monte Carlo results are of the order $10^{-4}$, well within the symbol size.

## 4.2 Corrugated Slab

In Fig. 11, the angular transmission is compared for a slab and a corrugated slab. The parameter values were chosen to represent the typical design used in white LEDs. The main difference between the slab and the corrugated slab is that at small $\cos\theta$, the corrugated slab has a higher transmission than the slab. The difference in angular transmission between the rectangular slab and the corrugated is shown in Fig. 12. As a measure of the size of the deviation, the absolute difference is integrated for all angles, which results in a difference of 2.1%. In addition to this effect, the total transmission through the corrugated slab is lower compared to the slab by 0.36%.

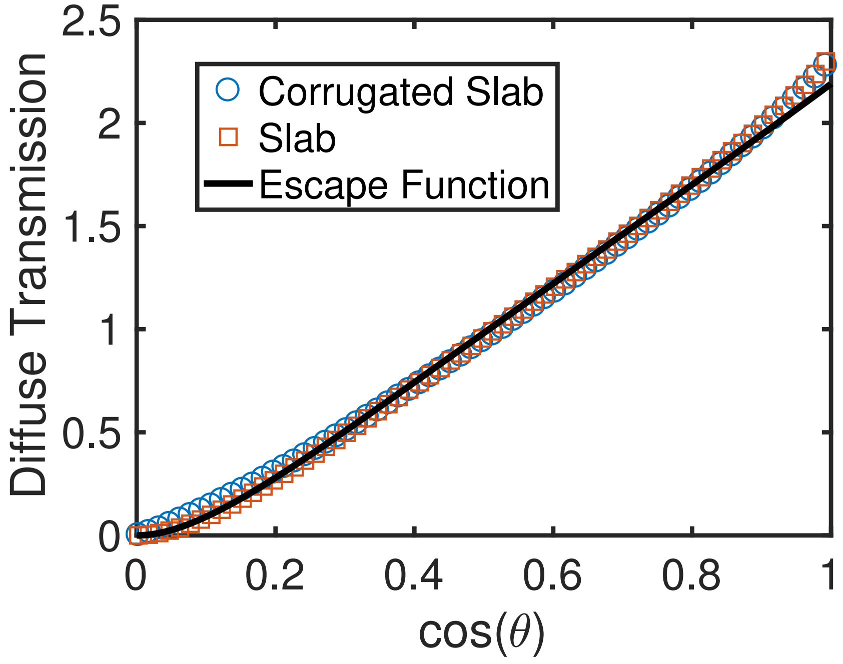


Figure 11: Diffuse reflection as a function of angle. The error bars of the Monte Carlo results are within the symbol size. Here, $L = 1$, $g = 0.9$, $n_m = 1.49$, $n_a = 1$ and $l_s = 0.08$. The properties of the corrugation are $A = 0.05$ and $f = 1$.

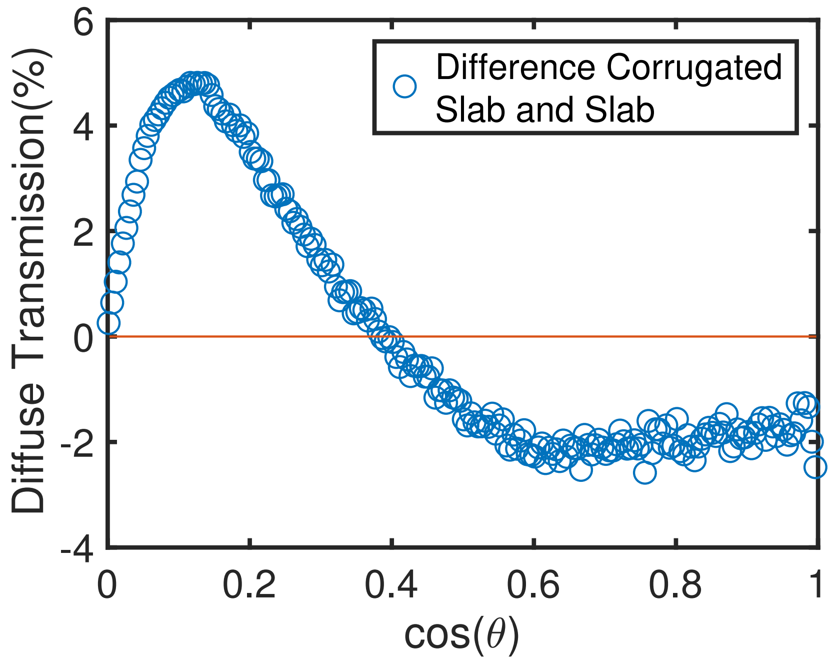


Figure 12: Difference between the results for the corrugated slab and the regular slab from Fig. 11.

A possible explanation for the difference in the angular distribution of diffusely transmitted light is that according to the escape function, light escapes mostly at angles perpendicular to the surface. Because in a corrugated slab, the surface is not perpendicular to the direction in which the light propagates, more light emerges from the surface at larger $\theta$, or smaller $\cos\theta$, which would explain the difference observed in Fig. 12.

An explanation for the decrease in total transmission is that the light is refracted at the first curved surface, which causes the average distance traveled to the second surface to be greater. To test this hypothesis, a simulation was run for the same parameters where the first surface was a flat surface and the second surface was corrugated (Fig. 13). The result was an increase (instead of a decrease) of transmission with respect to two flat surfaces, by 0.16%. This suggests that when a photon arrives at the neighbourhood of a corrugated surface from inside the material, the chance that it transmits to the air is higher than when it arrives at the neighbourhood of a flat surface.

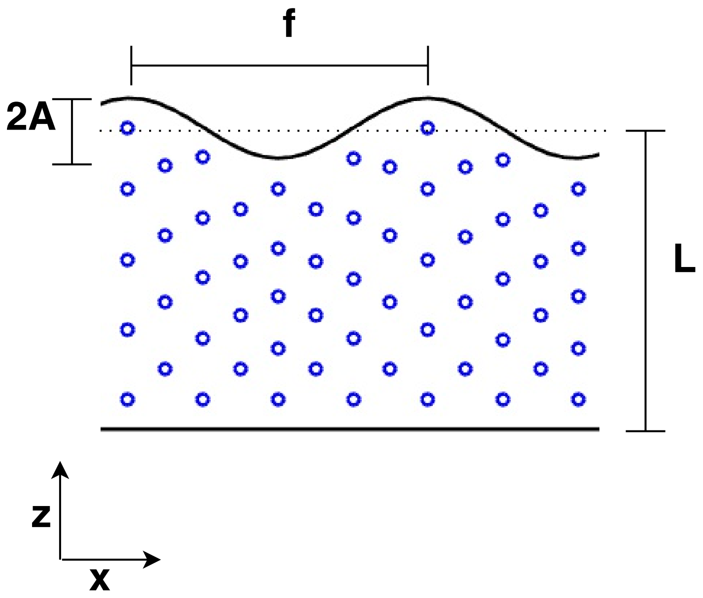


Figure 13: A slab of thickness L, which is flat on one side and corrugated with amplitude $A$ and frequency $f$ on the other side. Note that f is the frequency and not the period of the corrugation.

It is interesting to know how the deviation in the angular distribution of the light depends on the geometry of the sample. In Fig. 14, the dependence of the deviation as a function of the amplitude of the corrugation is shown. When the amplitude is larger, the deviation also becomes larger as expected. In Fig. 15, the dependence of the deviation as a function of the frequency is shown. When the frequency is larger the deviation also becomes larger. An analytical expression has not been derived yet for these graphs.

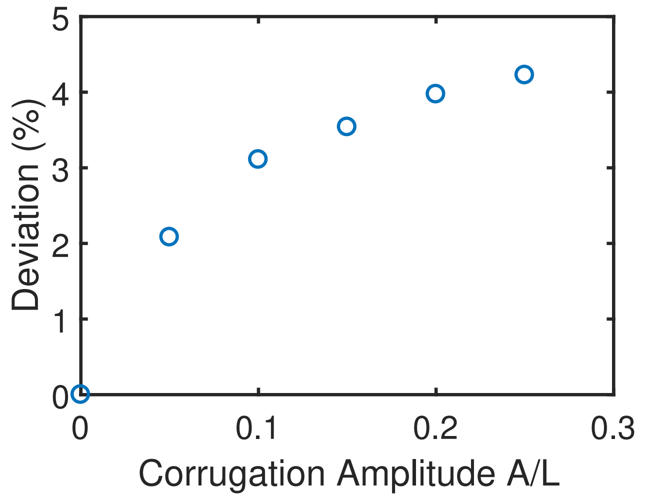


Figure 14: Deviation of the angular distribution of the light as a function of amplitude of corrugation, gauged as an integral over the absolute difference between the angular distribution of a slab and a corrugated slab. The error bars are within the symbol size. Here, $L = 1$, $g = 0.9$, $n_m = 1.49$, $n_a = 1$, $l_s = 0.08$ and and $f = 1$.

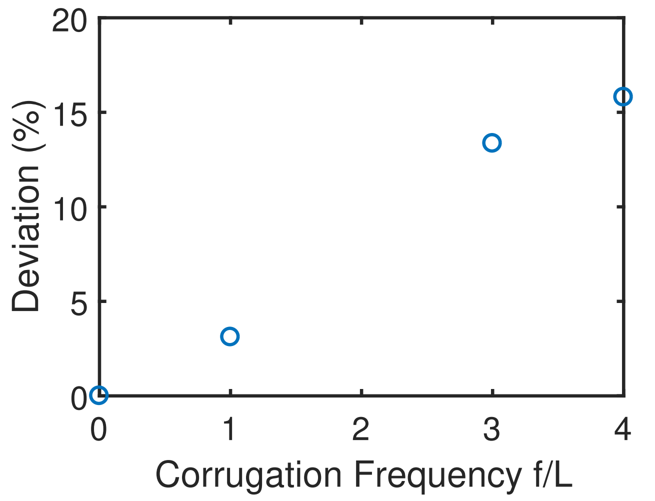


Figure 15: Deviation of the angular distribution of the light as a function of frequency of corrugation, gauged as an integral over the absolute difference between the angular distribution of a slab and a corrugated slab. The error bars are within the symbol size. Here, $L = 1$, $g = 0.9$, $n_m = 1.49$, $n_a = 1$, $l_s = 0.08$ and and $A = 0.1$.

# 5 Conclusion and Recommendations

We have studied the effect of corrugating the surfaces of a slab which consists of a scattering medium on light propagation through such a slab. It is shown that the corrugation influences the total transmission through the slab and the angular distribution of the transmission. The total transmission through a slab where both surfaces are corrugated is less then the total transmission through a slab with two straight surfaces, but when the surface on which the light is incident is straight and the other surface is corrugated, the total transmission is even higher. The difference in the angular distribution is amplified by increasing the amplitude and the frequency of the corrugation.

The Monte Carlo simulation that was written and used, can be modified in many ways depending on the goals of the user. Absorption, polarization and interference of light could be included, as well as structural anisotropy. The shapes of the boundaries can be changed as well as the number of different kinds of materials and interfaces between them. Finally, parallel computing could be used to speed up simulations.

## Acknowledgements

I would like to thank my daily supervisor Maryna Meretska for guiding me through this project. Furthermore I would like to thank my other supervisors Matthias Schlottbom and Willem Vos for their experience and useful directions. Finally, thanks to everyone at COPS for useful discussions and being an example to me; in particular my fellow students.

## References

[1] The Optoelectronics Research Centre. The life and times of the led — a 100-year history. April 2007.

[2] Jens Als-Nielsen and Des McMorrow. *Elements of modern x-ray physics.* Wiley, New York, NY, 2001.

[3] E. Hecht. *Optics.* Pearson education. Addison-Wesley, 2002.

[4] L.V. Wang and H. Wu. *Biomedical Optics: Principles and Imaging.* John Wiley & Sons, Inc., Hoboken, New Jersey, 2007.

[5] L.C. Henyey and J.L. Greenstein. Diffuse radiation in the galaxy. *Astrophysical Journal*, 93:70–83, 1941.

[6] B.P.J. Bret. *Multiple light scattering in porous gallium phosphide.* PhD thesis, University of Twente, 2005.

[7] N. Metropolis and S. Ulam. The Monte Carlo Method. *Journal of the American Statistical Organization*, pages 335–341, 1949.

[8] C. Zhu and Q. Liu. Review of Monte Carlo modeling of light transport in tissues. *Journal of Biomedical Optics*, 2013.

[9] W. T. Vetterling W. H. Press, S. A. Teukolsky and B.P. Flannery. *Numerical Recipes in C (2Nd Ed.): The Art of Scientific Computing.* Cambridge University Press, New York, NY, USA, 1992.

# A Code (C++)

## A.1 mcff.cpp

```cpp
#include <iostream>
#include <time.h>
#include "mcff_structures.h"
#include "mcff_io.h"
#include "mcff_functions.h"

int main()
{
    std::cout << "Number of photons to simulate: " << double(M*N)/1000000 << " million." << std::endl; // At the start of the simulation, show how many photons have to be simulated

    clock_t time, begintime = clock();      // Keep track of the execution time to give an estimation for the time it takes to finish the simulation

    double T;                               // The current execution time (as a double)

    if (refoutput){removerefoutput();}      // Remove reference output if the simulation is for reference output
    else{removeoutput();}                   // Remove output if the  simulation is for output

    for (int m = 0; m < M; m++)             // For each sub-simulation
    {
    OutputStruct O = initialize_output();   // Initialize the output data

    for (long n = 0; n < N; n++)            // For each photon
    {
        if (((m*N+n)%50000 == 0) & (m+n != 0))  // For certain photons, make an estimation of time remaining and show that in the console
        {
            time = clock();
            T = double(time - begintime)*double((M*N-(m*N+n))/double(m*N+n))/1000;
            std::cout << "Number of photons to go: " <<  double((N-n) +(M-m-1)*N)/1000000 << " million, estimated time remaining: " << floor(T/3600) << " hours, " << floor((T-3600*floor(T/3600))/60) << " minutes and " << floor(int(T)%60) << " seconds." << std::endl;
        }

        PhotonStruct P = initialize_photon();       // Initialize the data of the photon, (position, velocity, weight, etcetera)
        P.layer = &air;                             // The photon starts in the air layer

        while(!P.dead)                              // While the photon needs to be simulated
        {
            double closest_boundary = 0;            // Distance to closest boundary or edge is initialized as 0
            int found_boundary = 0, found_edge = 0; // Keep track of whether the photon hits a boundary or edge before it has travelled it's travel_distance
            travel(P, closest_boundary, found_boundary, found_edge);    // Calculate closest event and travel the photon
            if (found_boundary)                                         // If the photon encounters a boundary
            {
                boundary_action(P, closest_boundary, found_boundary, O);// Interact with the boundary
                     (Reflection/Transmission and refraction)
            }
            else if (found_edge)                    // If the photon reaches the edge of the domain
            {
                edge_action(P, found_edge, O,n);    // Interact with the edge
            }
            else if ((P.layer) -> mu != 0)          // If nothing else happens, the photon scatters
            {
                scatter(P, g, O);                   // Scatter the photon (update direction of
                    propagation)
            }
            else
            {
                P.pos[0] += diam*P.vel[0];
                P.pos[1] += diam*P.vel[1];
                P.pos[2] += diam*P.vel[2];
            }
        }
    }
    write_output(O,refoutput);    std::cout << "Writing output..." << std::endl;    // Write to all
        the output files after each sub-simulation
    }
}
```

## A.2 mcff_structures.h

```
#ifndef MCFF_IO_H
#define MCFF_IO_H
#include "mcff_structures.h"

/*---------------------------------------- Simulation parameters ---------------------------------------*/

const long N = 10000;         // The number of photons simulated per subsimulation
const int M = 10;               // The number of subsimulations
const double dx = 0.0001;       // The precision used to determine the location of boundaries
const int checks = 5;           // The number of checks that is done to check if a photon went outside the material and then inside again in a distance diam
const double diam = 1;          // The distance photons travel if there is no boundary or scattering event
const bool refoutput = false;    // Can be used to make reference output for the same parameters

/*---------------------------------------- Physical parameters -----------------------------------------*/

const double n_material = 1.49;     // Refractive index of the material
const double n_air = 1;             // Refractive index of the surrounding material
const double g = 0.9;               // The scattering anisotropy
const double mfp = 0.8*(1-g);       // Scattering mean free path
const double mu_material = 1/mfp;   // Optical density of the material
const double mu_air = 0;            // Optical density of air

/*---------------------------------------------- Output ----------------------------------------------*/

struct OutputStruct
{
int transmission = 0;                           // Total transmission (amount of photons that exit at edge 2)
int reflection = 0;                             // Total reflection (amount of photons that exit at edge 1)
static const int thetabins = 200;               // Number of bins in the theta-direction (angle with respect to z-axis)
static const int phibins = 200;                 // Number of bins in the phi-direction (angle in the (x,y)-plane)
int angtransmission[thetabins][phibins] = {};   // 2-dimensional array for angular resolved diffuse transmission
int baltransmission[thetabins][phibins] = {};   // 2-dimensional array for angular resolved ballistic transmission
int angreflection[thetabins][phibins] = {};     // 2-dimensional array for angular resolved diffuse reflection
int specreflection[thetabins][phibins] = {};    // 2-dimensional array for angular resolved specular reflection
int unscatteredref = 0;                         // The total number of photons from specular reflection
int unscatteredtrans = 0;                       // The total number of photons from ballistic transmission
};

/*--------------------------------------------- Geometry -----------------------------------------------*/
```

```
const double slab_width = 1;                // The width of the slab of material
const double A = refoutput ? 0.00 : 0.1;   // The corrugation amplitude (if refoutput is true, the
    boundaries are flat)
const double f = 1;                         // The corrugation frequency (length one complete
    oscillation in the x-direction)

/*--------------- Boundaries ---------------*/

const int num_boundaries = 2; // Number of boundaries of the material. When changing this, also
    change the boundary functions and the boundary_gradient functions accordingly.

/* Functions for the boundaries of the material. The zeros of these functions define the boundaries
    of the material.
   When changing these functions, also change the gradient functions in boundary_gradient and the
       number of boundaries accordingly. */

double boundary(double x, double y, double z, int boundary_number)
{
    if      (boundary_number == 1)
    {
        return z + A*sin(2*pi*f*x) - 1;             // Lower boundary for a corrugated slab
    }
    else if (boundary_number == 2)
    {
        return z + A*sin(2*pi*f*x) - slab_width - 1; // Upper boundary for a corrugated slab
    }
    else {std::cout << "Error, boundary doesn't exist" << std::endl; exit(1);}
}

/* Functions for the gradients of the boundaries of the material.
   When changing these functions, also change the functions in boundary and the number of edges
       accordingly. */

void boundary_gradient(double x, double y, double z,int boundary_number,double* gradient)
{
    if (boundary_number == 1)
    {
        gradient[0] = 2*pi*f*A*cos(2*pi*f*x); // Gradient of a corrugated slab (lower boundary)
        gradient[1] = 0;
        gradient[2] = 1;
    }
    else if (boundary_number == 2)
    {
        gradient[0] = 2*pi*f*A*cos(2*pi*f*x); // Gradient of a corrugated slab (upper boundary)
        gradient[1] = 0;
        gradient[2] = 1;
    } else {std::cout << "Warning, boundary_gradient doesn't exist" << std::endl; exit(2);}
}

/*----------------- Edges -----------------*/

const int num_edges = 2;    // Number of boundaries of the material. When changing this, also change
    the boundary functions and the boundary_gradient functions accordingly.

/* Functions for the gradients of the boundaries of the edge of the simulation ('edges').
```

```
   When changing these functions, also change the functions in edge_gradient and the number of edges
       accordingly. */

double edge(double x, double y, double z, int edge_number)
{
    if      (edge_number == 1) return z + A + 0.01;                                 // Lower edge (A
        is added to prevent the boundaries from touching the edge
    else if (edge_number == 2) return -z + slab_width + A + 1.01;                   // Upper edge (A
        is added to prevent the boundaries from touching the edge
    else {std::cout << "Warning, edge doesn't exist" << std::endl; exit(3);}
}

/* Functions for the gradients of the edge of the simulation.(NORMALIZED?)
   When changing these functions, also change the functions in edge and the number of edges
       accordingly. */

void edge_gradient(double x, double y, double z,int edge_number,double* gradient)
{
    if (edge_number == 1)
    {
        gradient[0] = 0; // Gradient for the lower edge
        gradient[1] = 0;
        gradient[2] = 1;
    }
    else if (edge_number == 2)
    {
        gradient[0] = 0; // Gradient for the upper edge
        gradient[1] = 0;
        gradient[2] = -1;
    }
}

#endif // MCFF_IO_H
```

## A.3 mcff_io.h

```
#ifndef MCFF_IO_H
#define MCFF_IO_H
#include "mcff_structures.h"

/*---------------------------------------- Simulation parameters ---------------------------------------*/

const long N = 10000;         // The number of photons simulated per subsimulation
const int M = 10;               // The number of subsimulations
const double dx = 0.0001;       // The precision used to determine the location of boundaries
const int checks = 5;           // The number of checks that is done to check if a photon went outside the material and then inside again in a distance diam
const double diam = 1;          // The distance photons travel if there is no boundary or scattering event
const bool refoutput = false;    // Can be used to make reference output for the same parameters

/*---------------------------------------- Physical parameters ----------------------------------------*/

const double n_material = 1.49;     // Refractive index of the material
const double n_air = 1;             // Refractive index of the surrounding material
const double g = 0.9;               // The scattering anisotropy
const double mfp = 0.8*(1-g);       // Scattering mean free path
const double mu_material = 1/mfp;   // Optical density of the material
const double mu_air = 0;            // Optical density of air

/*---------------------------------------------- Output ---------------------------------------------*/

struct OutputStruct
{
int transmission = 0;                            // Total transmission (amount of photons that exit at edge 2)
int reflection = 0;                              // Total reflection (amount of photons that exit at edge 1)
static const int thetabins = 200;                // Number of bins in the theta-direction (angle with respect to z-axis)
static const int phibins = 200;                  // Number of bins in the phi-direction (angle in the (x,y)-plane)
int angtransmission[thetabins][phibins] = {};   // 2-dimensional array for angular resolved diffuse transmission
int baltransmission[thetabins][phibins] = {};   // 2-dimensional array for angular resolved ballistic transmission
int angreflection[thetabins][phibins] = {};     // 2-dimensional array for angular resolved diffuse reflection
int specreflection[thetabins][phibins] = {};    // 2-dimensional array for angular resolved specular reflection
int unscatteredref = 0;                          // The total number of photons from specular reflection
int unscatteredtrans = 0;                        // The total number of photons from ballistic transmission
};

/*--------------------------------------------- Geometry ----------------------------------------------*/
```

```

const double slab_width = 1;                 // The width of the slab of material
const double A = refoutput ? 0.00 : 0.1;    // The corrugation amplitude (if refoutput is true, the boundaries are flat)
const double f = 1;                          // The corrugation frequency (length one complete oscillation in the x-direction)

/*--------------- Boundaries ---------------*/

const int num_boundaries = 2; // Number of boundaries of the material. When changing this, also change the boundary functions and the boundary_gradient functions accordingly.

/* Functions for the boundaries of the material. The zeros of these functions define the boundaries of the material.
   When changing these functions, also change the gradient functions in boundary_gradient and the number of boundaries accordingly. */

double boundary(double x, double y, double z, int boundary_number)
{
    if      (boundary_number == 1)
    {
        return z + A*sin(2*pi*f*x) - 1;             // Lower boundary for a corrugated slab
    }
    else if (boundary_number == 2)
    {
        return z + A*sin(2*pi*f*x) - slab_width - 1; // Upper boundary for a corrugated slab
    }
    else {std::cout << "Error, boundary doesn't exist" << std::endl; exit(1);}
}

/* Functions for the gradients of the boundaries of the material.
   When changing these functions, also change the functions in boundary and the number of edges accordingly. */

void boundary_gradient(double x, double y, double z,int boundary_number,double* gradient)
{
    if (boundary_number == 1)
    {
        gradient[0] = 2*pi*f*A*cos(2*pi*f*x); // Gradient of a corrugated slab (lower boundary)
        gradient[1] = 0;
        gradient[2] = 1;
    }
    else if (boundary_number == 2)
    {
        gradient[0] = 2*pi*f*A*cos(2*pi*f*x); // Gradient of a corrugated slab (upper boundary)
        gradient[1] = 0;
        gradient[2] = 1;
    } else {std::cout << "Warning, boundary_gradient doesn't exist" << std::endl; exit(2);}
}

/*----------------- Edges ------------------*/

const int num_edges = 2;    // Number of boundaries of the material. When changing this, also change the boundary functions and the boundary_gradient functions accordingly.

/* Functions for the gradients of the boundaries of the edge of the simulation ('edges').
```

```
   When changing these functions, also change the functions in edge_gradient and the number of edges
       accordingly. */

double edge(double x, double y, double z, int edge_number)
{
    if      (edge_number == 1) return z + A + 0.01;                                 // Lower edge (A
        is added to prevent the boundaries from touching the edge
    else if (edge_number == 2) return -z + slab_width + A + 1.01;                   // Upper edge (A
        is added to prevent the boundaries from touching the edge
    else {std::cout << "Warning, edge doesn't exist" << std::endl; exit(3);}
}

/* Functions for the gradients of the edge of the simulation.(NORMALIZED?)
   When changing these functions, also change the functions in edge and the number of edges
       accordingly. */

void edge_gradient(double x, double y, double z,int edge_number,double* gradient)
{
    if (edge_number == 1)
    {
        gradient[0] = 0; // Gradient for the lower edge
        gradient[1] = 0;
        gradient[2] = 1;
    }
    else if (edge_number == 2)
    {
        gradient[0] = 0; // Gradient for the upper edge
        gradient[1] = 0;
        gradient[2] = -1;
    }
}

#endif // MCFF_IO_H
```

## A.4 mcff_functions.h

```
#ifndef MCFF_FUNCTIONS_H
#define MCFF_FUNCTIONS_H
#include <fstream>
#include <iostream>
#include "mcff_io.h"
#include "mcff_structures.h"

/* Function that initializes each photon */

PhotonStruct initialize_photon()
{
    PhotonStruct P;             // P is the structure that contains the data associated with the photon
    P = {};                     // Initialize P by setting everything to zero
    P.pos[0] = (1/f)*random();  // Choose a random position in the frequency of the corrugation
    P.vel[2] = 1;               // Set the velocity of the photon in the z-direction
    P.w = 1;                    // Set the weight of the photon equal to 1.
    return P;
}

/* Function that initializes the output */

OutputStruct initialize_output()
{
    OutputStruct O;             // O is the structure that contains the data associated with the output
    O = {};                     // Initialize O by setting everything to zero
    return O;
}

/* --- Initialize Layers --- */

LayerStruct air = {1,mu_air,n_air,num_boundaries,num_edges};        // Initialize the 'air' layer
LayerStruct material = {2,mu_material,n_material,num_boundaries,0}; // Initialize the 'material' layer

/* Returns the cosine of the scattering angle according to Henyey-Greenstein function */

double get_scatter_angle(double g,double ksi)
{
    if (g == 0) return 2*ksi - 1;                               // This is the limit of the Henyey-Greenstein function for g goes to 0.
    else
    {
        return 1/(2*g) * (1+g*g-pow(((1-g*g)/(1-g+2*g*ksi)),2));    // Henyey-Greenstein function
    }
}

/* Calculate where the photon will travel and if it encounters an edge or boundary while travelling */

void travel(PhotonStruct & P, double & closest_boundary, int & found_boundary, int & found_edge)
{
    if (P.s == 0)                                               // If step size of the photon equals zero, sample it
    {
        P.s = -log(random());                                          // Sample dimensionless step size of photon
    }

    double final_position[3], travel_distance;                          // Position the photon travels to and Distance the photon travels

    if (P.layer -> mu == 0)                                             // If the optical density of the layer equals 0
    {
        travel_distance = diam;                                         // Update travel distance (high)
        final_position[0] = P.pos[0] + travel_distance * P.vel[0];  // Calculate the final position of the photon after it would travel travel_distance.
        final_position[1] = P.pos[1] + travel_distance * P.vel[1];
        final_position[2] = P.pos[2] + travel_distance * P.vel[2];
    }
    else
    {
        travel_distance = P.s/P.layer -> mu;                            // Update travel distance
        final_position[0] = P.pos[0] + travel_distance * P.vel[0];  // Calculate the final position of the photon after it would travel travel_distance.
        final_position[1] = P.pos[1] + travel_distance * P.vel[1];
        final_position[2] = P.pos[2] + travel_distance * P.vel[2];
    }

    double ref_boundary[P.layer -> num_boundaries], ref_edge[P.layer -> num_edges];     // Keeps track of the values of the boundary functions at the current position
    int sgnref_boundary[P.layer -> num_boundaries], sgnref_edge[P.layer -> num_edges];  // Pre-calculated signs of the values in ref_boundary

    double boundary_crossing = 0;                                       // Keeps track of the distance to the boundary, as a fraction of the travel_distance
    bool edge_crossing = false;                                         // Keeps track of whether the photon crosses an edge

    for (int i = 1; i <= P.layer -> num_boundaries; i++)
    {
        ref_boundary[i-1] = boundary(P.pos[0],P.pos[1],P.pos[2],i); // Store the values of the boundary functions
        sgnref_boundary[i-1] = sgn(ref_boundary[i-1]);                  // Store the signs of ref_boundary
        if (sgn(boundary(final_position[0],final_position[1],final_position[2],i)) != sgnref_boundary[i-1])
        {
            boundary_crossing = 1;                                      // If the sign of a boundary function changes, it has been crossed
        }
    }

    for (int i = 1; i <= P.layer -> num_edges; i++)
    {
        ref_edge[i-1] = edge(P.pos[0],P.pos[1],P.pos[2],i);             // Store the edge functions at previous location
        sgnref_edge[i-1] = sgn(ref_edge[i-1]);
        if (sgn(edge(final_position[0],final_position[1],final_position[2],i)) != sgnref_edge[i-1])    // If the sign of an edge function changes, an edge has been crossed
        {
            edge_crossing = true;
        }
    }

    if (boundary_crossing == 0)                                 // If no boundary is found, do extra checks to determine if there is a boundary crossing somewhere else.
    {
        for (int i = 1; i<= P.layer -> num_boundaries; i++)         // For every boundary
        {
            for (double theta = checks; theta >= 1; theta--)        // Do 5 checks. Theta is a double since it has to be divided to give a double instead of an integer
            {
                if (sgn(boundary((theta/(checks + 1)) * final_position[0] + (checks + 1 - theta) / (checks + 1) * P.pos[0],     //If the sign changes, update boundary_crossing
                                 (theta/(checks + 1)) * final_position[1] + (checks + 1 - theta) / (checks + 1) * P.pos[1],
                                 (theta/(checks + 1)) * final_position[2] + (checks + 1 - theta) / (checks + 1) * P.pos[2],i)) != sgnref_boundary[i-1])
                {
                    if (boundary_crossing == 0)                     // If this is the first boundary crossing that is found
                    {
                        boundary_crossing = theta/(checks + 1);     // Set the relative distance of the boundary crossing
                    }
                    else                                            // else take the minimum boundary distance found until now
                    {
                        boundary_crossing = std::min(boundary_crossing,theta/checks + 1);
                    }
                }
            }
        }
    }

    if (boundary_crossing)                                      // If at least one boundary is crossed, determine which boundary is closest
    {
        for (int i = 1; i<= P.layer -> num_boundaries; i++)         // For each boundary
        {
            double y_high = boundary(boundary_crossing * final_position[0] + (1 - boundary_crossing) * P.pos[0],
                                     boundary_crossing * final_position[1] + (1 - boundary_crossing) * P.pos[1],
                                     boundary_crossing * final_position[2] + (1 - boundary_crossing) * P.pos[2],i); // The value of the boundary function at the final position of the photon
            if (sgn(y_high) != sgnref_boundary[i-1])                // If there is a boundary crossing somewhere: use False Position method to determine the location of the boundary
            {
                double x_low = 0;                                   // Lower bound for position of boundary
                double y_low = ref_boundary[i-1];                   // Value of function at lower bound
                double x_high = boundary_crossing;                  // Upper bound for position of boundary
                double x_new = 0;                                   // Guess for position of boundary
                double y_new = y_low;                               // Value of function at guess
                while ((x_high - x_low) * travel_distance > dx)     // While the upper and lower bounds are not close enough
                {
                    x_new = (x_low * y_high - x_high * y_low)/(y_high - y_low); // Make a better guess (linear interpolation)
                    if ((x_new - x_high)*travel_distance < dx/2)    // if x_new and x_high are close, do a bisection step to avoid getting stuck
                    {
                        x_new = (x_high + x_low)/2;
                    }
                    else if ((x_new - x_low)*travel_distance < dx/2)     // if x_new and x_high are close, do a bisection step to avoid getting stuck
                    {
                        x_new = (x_high + x_low)/2;
                    }
                    y_new = boundary(x_new * final_position[0] + (1 - x_new) * P.pos[0],
                                     x_new * final_position[1] + (1 - x_new) * P.pos[1],
                                     x_new * final_position[2] + (1 - x_new) * P.pos[2],i); // Update value at guess
                    if (sgn(y_new) != sgn(y_low))                   // Always keep two points with different signs, so throw away the point with the same sign as the new point.
                    {
                        x_high = x_new;
                        y_high = y_new;
                    }
                    else
                    {
                        x_low = x_new;
                        y_low = y_new;
                    }
                }
                if (closest_boundary == 0)                      // If the closest_boundary is not updated yet
                {

                    closest_boundary = x_new * travel_distance; // The closest boundary is the distance we just found
                    found_boundary = i;                         // Keep track of the boundary that the photon hits.
                }
                else                                            // If the closest boundary has been updated before
                {
                    if (x_new * travel_distance < closest_boundary) // If the new boundary is closer
                    {
                        closest_boundary = x_new * travel_distance; // The closest boundary is the distance we just found
                        found_boundary = i;                         // Keep track of the boundary that the photon hits.
                    }
                }
            }
        }
```

```
    }

    if (edge_crossing)                                 // Determine the distance to the closest edge
    {
        for (int i = 1; i<= P.layer -> num_edges; i++) // For each edge
        {
            double y_high = edge(final_position[0],final_position[1],final_position[2],i); // The value of the boundary function at the final position of the photon
            if (sgn(y_high) != sgnref_edge[i-1])    // If there is a crossing
            {
                double x_low = 0;                   // Lower bound position of edge
                double y_low = ref_edge[i-1];       // Value edge function at lower bound
                double x_high = 1;                  // Upper bound position of edge
                double x_new = 0;                   // Guess for position of edge
                double y_new = y_low;               // Value of edge function at guess
                while ((x_high - x_low) * travel_distance > dx) // While the upper and lower bounds are not close enough
                {
                    x_new = (x_low * y_high - x_high * y_low)/(y_high - y_low); // Make a better guess (linear interpolation)
                    if ((x_new - x_high)*travel_distance < dx/2)
                    {
                        x_new = (x_high + x_low)/2;
                    };
                    if ((x_new - x_low)*travel_distance < dx/2)
                    {
                        x_new = (x_high + x_low)/2;
                    };
                    y_new = edge(x_new * final_position[0] + (1 - x_new) * P.pos[0],
                                 x_new * final_position[1] + (1 - x_new) * P.pos[1],
                                 x_new * final_position[2] + (1 - x_new) * P.pos[2],i); // Update value at guess
                    if (sgn(y_new) != sgn(y_low))   // Always keep two points with different signs, so throw away the point with the same sign as the new point.
                    {
                        x_high = x_new;
                        y_high = y_new;
                    }
                    else
                    {
                        x_low = x_new;
                        y_low = y_new;
                    }
                }
                if (closest_boundary == 0)          // If the closest_boundary is not updated yet
                {
                    closest_boundary = x_new * travel_distance; // The closest boundary is the distance we just found
                    found_edge = i;                 // Keep track of the edge that the photon hits.
                }
                else
                {
                    if (x_new * travel_distance < closest_boundary) // If the new boundary is closer
                    {
                        closest_boundary = x_new * travel_distance; // Update closest boundary
                        found_edge = i;                             // Set the edge we just found
                        found_boundary = 0;                          // Reset the found_boundary (Because that is checked first below)
                    }
                }
            }
        }
    }
}

/* Scatter the photon */

void scatter(PhotonStruct & P, const double anisotropy, OutputStruct & O)
{
    P.scattered = true;                         // Keep track of the fact that the photon has been scattered

    P.pos[0] += P.s/(P.layer) -> mu*P.vel[0];   // Update the position of the P
    P.pos[1] += P.s/(P.layer) -> mu*P.vel[1];
    P.pos[2] += P.s/(P.layer) -> mu*P.vel[2];
    P.s = 0;                                    // Set the step size to zero (so next loop, a new step size will be sampled)
    double costheta = get_scatter_angle(anisotropy,random());   // Get the cosine of the scattering angle
    double sintheta = sqrt(std::abs(1-costheta*costheta));      // Calculate the sine (always positive.
    double phi = 2*pi * random();               // The second angle of scattering (sampled uniform from 0 to 2 pi)
    double cosphi = cos(phi);
    double sinphi = sgn(pi - phi) * sqrt(std::abs(1-cosphi*cosphi));

    if (std::abs(P.vel[2]) > 0.999)             // If the P flies in the z-direction, use these equations (to prevent dividing by 0)
    {
        P.vel[0] = sintheta*cosphi;             // Update direction of velocity due to scattering
        P.vel[1] = sintheta*sinphi;
        P.vel[2] = sgn(P.vel[2])*costheta;
    }
    else
    {
        double ux_new = (sintheta * (P.vel[0] * P.vel[2] * cosphi - P.vel[1] * sinphi))/(sqrt(1 - P.vel[2] * P.vel[2])) + P.vel[0] * costheta; // Update direction of velocity due to scattering
        double uy_new = (sintheta * (P.vel[1] * P.vel[2] * cosphi + P.vel[0] * sinphi))/(sqrt(1 - P.vel[2] * P.vel[2])) + P.vel[1] * costheta;
        double uz_new = -sqrt(1 - P.vel[2] * P.vel[2]) * sintheta * cosphi + P.vel[2] * costheta;

        P.vel[0] = ux_new;
        P.vel[1] = uy_new;
        P.vel[2] = uz_new;
    }
}

/* The photon interacts with a boundary*/

void boundary_action(PhotonStruct & P, double closest_boundary, int found_boundary,OutputStruct & O)
{
    P.pos[0] += closest_boundary*P.vel[0];  // Update the coordinates of the photon
    P.pos[1] += closest_boundary*P.vel[1];
    P.pos[2] += closest_boundary*P.vel[2];

    P.s -= closest_boundary* (*P.layer).mu; // Update the dimensionless step size of the P

    double g[3];                                                    // Initialize the normal of the
        boundary

    boundary_gradient(P.pos[0],P.pos[1],P.pos[2],found_boundary,g); // Get the normal of the boundary

    double sizeg = g[0]*g[0] + g[1]*g[1] + g[2]*g[2];
    if (sizeg == 0)
    {
        std::cout << "Error, gradient equals 0 on boundary" << std::endl;
    }

    g[0] = g[0]/sqrt(sizeg);                    // Normalize the normal
    g[1] = g[1]/sqrt(sizeg);
    g[2] = g[2]/sqrt(sizeg);

    LayerStruct * current_layer = P.layer;      // Keep track of layer in which the photon resides
    LayerStruct * transmission_layer;           // Keep track of layer on the other side of the
        boundary

    if (current_layer -> i == air.i)            // If the photon is in air right now
    {
        transmission_layer = &material;         // transmission layer is the other side of the
            boundary
    }
    else                                        // if the photon is in material right now
    {
        transmission_layer = &air;              // transmission layer is the other side of the
            boundary
    }

    double n1 = current_layer -> n;             // save the refractive indices of the layers
    double n2 = transmission_layer -> n;

    double ip = g[0] * P.vel[0] + g[1] * P.vel[1] + g[2] * P.vel[2]; // Inner product of velocity of
        photon with normal

    int sgnip = sgn(ip);

    double R = 0; // Reflection probability

    if (ip == 0)
    {
        std::cout << "Photon parallel to wall. Assuming reflection." << std::endl;
        R = 1;
    }

    double costheta_i = sgnip*ip;                                   // Cosine of angle of velocity
        with respect to the normal (always positive)
    double sintheta_i = sqrt(std::abs(1-costheta_i*costheta_i));    // Calculate the sine from the
        cosine. Take the absolute value because of rounding errors

    if (n1/n2 * sintheta_i >= 1)    // If there is total internal reflection according to Snell's law
```

```cpp
    {
        R = 1;                      // The reflection coefficient is one.
    }
    else
    {
        double S = sqrt(1-pow(n1/n2 * sintheta_i,2));   // Pre-calculate the square root
        R = (pow((n1*costheta_i-n2*S)/(n1*costheta_i+n2*S),2) + pow((n1*S-n2*costheta_i)/(n1*S+n2*costheta_i),2))/2;    // Calculate the reflection coefficient averaged over polarizations
        if (R > 1)
        {
            std::cout << "Error, reflection coefficient greater than 1" << std::endl;
        }
    }

    if (random() < R) // If the P is reflected
    {
        P.pos[0] -= sgnip * 2 * dx * g[0];  // Move the photon a bit back from the wall to avoid rounding errors
        P.pos[1] -= sgnip * 2 * dx * g[1];
        P.pos[2] -= sgnip * 2 * dx * g[2];

        P.vel[0] -= 2*ip*g[0];              // Update the velocity of the P
        P.vel[1] -= 2*ip*g[1];
        P.vel[2] -= 2*ip*g[2];
    }
    else // if the P is transmitted
    {
        double theta_t = asin(n1/n2 * sintheta_i);  // Calculate the angle of transmission (Snell's law)

        P.pos[0] += sgnip * 2 * dx * g[0];          // Move the photon across the wall to avoid rounding errors
        P.pos[1] += sgnip * 2 * dx * g[1];
        P.pos[2] += sgnip * 2 * dx * g[2];

        double costheta = cos(theta_t);             // Cosine of transmission angle

        P.vel[0] = costheta * sgnip * g[0] + n1 / n2 * (P.vel[0] - ip * g[0]); //Update the velocity of the Photon
        P.vel[1] = costheta * sgnip * g[1] + n1 / n2 * (P.vel[1] - ip * g[1]);
        P.vel[2] = costheta * sgnip * g[2] + n1 / n2 * (P.vel[2] - ip * g[2]);

        P.layer = transmission_layer;               // Update the layer of the photon
    }
}

void edge_action(PhotonStruct & P, int found_edge, OutputStruct & O,long n)
{
    if (found_edge == 2)        // If the photon went through the material to edge 2
    {
        O.transmission += 1;    // Increase the total  transmission
        int thetabin = std::min(O.thetabins-1,int(floor(P.vel[2]*(O.thetabins))));  // Calculate bins for the angles
        int phibin = std::min(O.phibins-1,int(floor(atan2(P.vel[1],P.vel[0])*(O.phibins/2)/pi+(O.phibins/2))));
        if (!P.scattered)       // If the photon didn't scatter put it in ballistic transmission, otherwise diffuse transmission
        {
            O.unscatteredtrans += 1;
            O.baltransmission[thetabin][phibin] += 1;
        }
        else
        {
            O.angtransmission[thetabin][phibin] +=1;
        }
    }
    else if (found_edge == 1)   //  If the photon reflected to the side it came from
    {
        O.reflection += 1;      // Increase the total reflection
        int thetabin = std::min(O.thetabins-1,int(floor(-P.vel[2]*(O.thetabins))));     // Calculate bins for the angles
        int phibin = std::min(O.phibins-1,int(floor(atan2(P.vel[1],P.vel[0])*(O.phibins/2)/pi+(O.phibins/2))));
        if (!P.scattered)       // If the photon didn't scatter put it in specular reflection, otherwise diffuse reflection
        {
            O.unscatteredref += 1;
            O.specreflection[thetabin][phibin] += 1;
        }
        else
        {
            O.angreflection[thetabin][phibin] +=1;
        }
    }
    else
    {
        std::cout << "Error, edge doesn't exist" << std::endl;
    }
    P.dead = true;              // Stop simulating this photon
}

void write_output(OutputStruct & O,bool refoutput)
{
    std::ofstream myfile;       // Use myfile as output
    if (refoutput)
    {
        myfile.open ("angtrans_ref.txt", std::ios_base::app);
    }
    else
    {
        myfile.open ("angtrans.txt", std::ios_base::app);
    }
    for (int i = 0; i < O.thetabins; i++)
    {
        for (int j = 0; j < O.phibins; j++)
        {
            myfile << O.angtransmission[i][j];
            if (j != O.phibins-1)
            {
                myfile << ", ";
            };
        }
        myfile << std::endl;
    }
    myfile.close();
    if (refoutput)
    {
        myfile.open ("baltrans_ref.txt", std::ios_base::app);
    }
    else
    {
        myfile.open ("baltrans.txt", std::ios_base::app);
    }
    for (int i = 0; i < O.thetabins; i++)
    {
        for (int j = 0; j < O.phibins; j++)
        {
            myfile << O.baltransmission[i][j];
            if (j != O.phibins-1)
            {
                myfile << ", ";
            };
        }
        myfile << std::endl;
    }
    myfile.close();

    if (refoutput)
    {
        myfile.open ("angref_ref.txt", std::ios_base::app);
    }
    else
    {
        myfile.open ("angref.txt", std::ios_base::app);
    }

    for (int i = 0; i < O.thetabins; i++)
    {
        for (int j = 0; j < O.phibins; j++)
        {
            myfile << O.angreflection[i][j];
            if (j != O.phibins-1)
            {
                myfile << ", ";
            };
        }
        myfile << std::endl;
    }
    myfile.close();

    if (refoutput)
    {
        myfile.open ("specref_ref.txt", std::ios_base::app);
    }
    else
    {
        myfile.open ("specref.txt", std::ios_base::app);
    }

    for (int i = 0; i < O.thetabins; i++)
    {
        for (int j = 0; j < O.phibins; j++)
        {
            myfile << O.specreflection[i][j];
            if (j != O.phibins-1)
            {
                myfile << ", ";
            };
        }
        myfile << std::endl;
    }
    myfile.close();

    if (refoutput)
    {
        myfile.open("output_ref.txt", std::ios_base::app);
    }
    else
    {
        myfile.open("output.txt", std::ios_base::app);
    }
    myfile << "Number of photons: " << N << std::endl;
    myfile << "Total transmission: " << O.transmission << std::endl;
    myfile << "Total reflection: " << O.reflection << std::endl;
    myfile << "Unscattered transmission: " << O.unscatteredtrans << std::endl;
    myfile << "Unscattered reflection: " << O.unscatteredref << std::endl << std::endl;
    myfile.close();
    if (refoutput)
    {
        myfile.open("parameters_ref.txt");
    }
    else
    {
        myfile.open("parameters.txt");
    }
    myfile << "N " << N <<std::endl;
    myfile << "M " << M <<std::endl;
    myfile << "f " << f <<std::endl;
    myfile << "A " << A <<std::endl;
    myfile << "thetabins " << O.thetabins <<std::endl;
    myfile << "phibins " << O.phibins <<std::endl;
    myfile << "ls " << mfp <<std::endl;
    myfile << "anisotropy " << g <<std::endl;
    myfile << "lt " << mfp/(1-g) <<std::endl;
    myfile << "slab_width " << slab_width <<std::endl;
    myfile << "n_material " << n_material <<std::endl;
    myfile << "n_air " << n_air <<std::endl;
    myfile.close();
}

void removerefoutput()  // Remove previous output in the folder
{
    remove("angtrans_ref.txt");
    remove("baltrans_ref.txt");
    remove("angref_ref.txt");
    remove("specref_ref.txt");
    remove("output_ref.txt");
}
```

```

void removeoutput()     // Remove previous output in the folder
{
    remove("angtrans.txt");
    remove("baltrans.txt");
    remove("angref.txt");
    remove("specref.txt");
    remove("output.txt");
}

#endif // MCFF_INPUT_H
```