Abstract

abstrakt pls

Preface

This project report was written as a part of the obligatory project in the second to last semester of a Master of Science in Engineering Physics at NTNU. The work was conducted at the Department of Electronics and Telecommunications. The motivation behind the project is to provide an important processing block in what will in the long run end up to be a diagnostic instrument to be used in medicine. The specifications are rather cybernetical, with deadlines and real time requirements, and the main topic matter either photonics-related or very computer scientific, as low-level programming of GPUs is used. All in all, a mish-mash of interdisciplinarity, all in the hands of a physicist.

I wish to thank my supervisor Lise Lyngsnes Randeberg for giving me this oppourtunity to do something worthwile with my project, and for all the help, advice, guidance and sharing of frustrations. Further thanks goes to Martin Denstedt for being available and bringing me up to speed at the start of the project.

I'd also like to thank Terje for probably/most likely reading through my report.

Contents

Chapter 1

Introduction

Hyperspectral imaging is a technique where images are taken and each pixel represents the whole visible spectrum instead of intensity values for red, green and blue. Hyperspectral imaging is widely used for remote sensing, and various techniques have been developed for information extraction. Hyperspectral imaging has also recently been adopted for diagnostic purposes in imaging of skin.

Human skin is a large structure covering the entire human body. Skin is subdivided into three layers, epidermis, dermis and hypodermis [12]. The epidermis protects the skin against water loss and disease-causing organisms, and is a dynamic structure which will protect itself when affected by external stimuli [12]. For instance, it will produce melanin to protect itself against UV-radiation, providing a tan [12]. Dermis consists mainly of connective tissue and blood vessels supplying nourishment to the skin [12]. Hypodermis contains fat cells [12]. Light can penetrate through mainly the first two of these layers and be reflected back. The amount of light reflected back at different wavelengths will reveal information about the structures and properties in the different layers.

A hyperspectral image of a skin sample will therefore contain important information about different materials at different depths. This information can be extracted, and used for various purposes in medicine, in particular by medical doctors with limited time for each patient, provided the information extraction procedure is fast. The hyperspectral community will use various spectral unmixing algorithms for matching different absorption spectras against the hyperspectral pixels, but in the case of skin, wavelength-dependent scattering and different penetration depths have been a problem.

Light transport models may describe the light transport through layered models approximating human skin. Scattering can be eliminated through inversion of these models. This inverse-modelling needs to be fast with little to no waiting time, and deliver its results as fast as the hyperspectral camera can process its surroundings. There are mainly two methods for doing forward-modelling of light transport.

Monte Carlo methods for calculating light transport are regarded to be a more accurate solutions to the Boltzmann transport equation than approximations like the diffusion approximation. In particular is MCML (Monte Carlo for Multi-Layered media) [45] widely regarded to be the so-called gold standard in the biomedical optics-field. The MCML package is freely available on the Internet [2], which might be part of the reson for its wide regard. However, Monte Carlo simulations are slow, since only one photon packet at a time can be simulated. Typically, the number of photon packets need to be in the order of magnitude of 30000 for the results to be accurate. A calculation time of 20 minutes will be typical for the full visible spectrum, from 400 nm to 800 nm with a spectral resolution of 160 wavelengths. Without any parallellization, the running time is unlikely to be reduced with newer processors, as the processor producers seem to focus on increasing the number of cores rather than the processing capabilities of each core.

Given the diffuse reflectance spectrum from a skin sample, the properties resulting in this particular diffuse reflection spectrum cannot be directly determined. As the Monte Carlo method is a black box forward model where the processes determining the final results are rather unknown, the properties may

be determined only through iteration. This will mean many slow evaluations of the same Monte Carlo routine.

However, recent developments show that the computation time of the Monte Carlo model can drastically be reduced through the use of GPUs. CPU parallellization without using supercomputers will not decrease the computation time significantly. The normal number of cores ranges from 2 to 5 on a normal CPU, which will only double the speed, and even with clever CPU scheduling, it will only take it so far. Asymptotically, it will be the same. 200 minutes is no better than 2000 minutes in terms of patient care. However, where a normal CPU is not optimized for this kind of task, a GPU (graphical processing unit) will be optimized for parallellization. GPU-MCML is a General Public License-licensed GPU-implementation of MCML freely available on the Internet [1].

Another method is to use the diffusion theory, which is an approximation. It has however successfully been used by many different researchers [41, 36, 29, 28, 24, 23], also for inverse modelling. The diffusion theory has an analytical solution to the problem, evaluateable using simple arithmetics.

This project will try to implement an inverse model based on the GPU-MCML program and evaluate its feasibility in terms of running time. This project will also implement an inverse model for hyperspectral images by use of the diffusion model and GPU parallellization, striving for real-time performance. The implementation should output results as fast as the camera can output its data. The motivation for this is providing a diagnostic instrument for use by doctors and the like, who will be able to scan skin using the hyperspectral camera and get easily understandable images of the different properties as inverse-modelled by the inverse model. The inverse model, earlier employed by Randeberg and others (see for example [28]), will be evaluated in each step.

This report will first go through GPU programming theory and theory related to photon transport in tissue. The theoretical foundations of Monte Carlo and the diffusion model are presented. The report will then move on to a presentation of the GPU-implementations of Monte Carlo and the diffusion model and go through the inversion steps of a diffuse reflectance spectrum. After this, the results will be presented, with benchmarking of some of the steps in the inversion strategy, comparisons of GPU-MCML and diffusion theory and benchmarking of GPU-MCML and the GPU-implementation of the diffusion model in terms of inversions.

Chapter 2

Theory and background

2.1 GPU-architecture

The problem of rendering and displaying graphics is in its nature a problem which, if to be done efficiently, requires parallelization and hyperthreading [42]. Due to this fact, Graphics Unit Processors (GPUs) are designed to handle multiple calculations at the same time, not only by employing multiple processor cores but also by enabling the different cores to process multiple threads at the same time. GPUs can also be put to use to other, parallellizable problems than graphics rendering. This was very difficult to do until the different GPU manufacturers recently released frameworks for enabling more easy programming of the devices. NVIDIA has a framework called CUDA, which essentially is C with some extensions and requirements to think low-level [4]. The GPU architecture as exposed by the CUDA framework will be presented, and some major optimization possibilities will be pointed out.

The GPU is very specifically targeted towards employing the same processor instructions on different datasets, as opposed to the CPU (Central Processing Unit), which will be designed to be more multi-purpose and handle a wide array of different tasks [4]. More specifically will each processing core on the GPU be able to handle 32 threads at once - provided that all the 32 threads do the same calculations and operations in the same sequence, only on different data. This puts some limits on which problems can be parallellized well by the GPU. For optimal parallellization, they need to follow the SIMD-principle - Single Instruction. All the threads need to be completely independent, since the GPU will try to run a different thread if one of the threads stall or lag behind. The handling of 32 threads at once for one instruction is referred to as a *warp* [4]. CUDA will organize its threads in blocks, as shown in figure 2.1. The blocks are distributed over the available GPU multiprocessors, as is shown in figure 2.2. Each multiprocessor will run the threads of its assigned block until completion, in warps of 32 threads each [4]. A neccessary requirement for maximum multiprocessor utilization is to have a number of threads per block divisible by 32. The blocks will be organized in a larger grid. Both the blocks and the grid can be one- or two-dimensional, but this has no performance benefits and is just for convenience when accessing any array data [4].

Each thread will run its own instance of a *kernel* [4]. This is a C-like function with some CUDA extensions, compiled for the GPU processor. The kernel is instanziated for a grid of blocks, and each thread in each block of threads will process the instructions in the kernel, on different data accessed by using thread- and block indices. Should any of the threads try to process different instructions, the warp will break down and the multiprocessor will run the threads sequentially instead of in parallell in groups of 32. Such divergence should be avoided [4].

Additionally, there are some memory requirements [3]. Each multiprocessor has a cache that may load a given amount of bytes. When one thread tries to read, say, two bytes from memory, the cache will typically also load the subsequent bytes from memory in one chunk as far as there is space left in the cache. If the threads in a warp all need data from subsequent memory locations, they will benefit from the same chunk of cache instead of ordering new cache reads. On the other hand, if they do not need

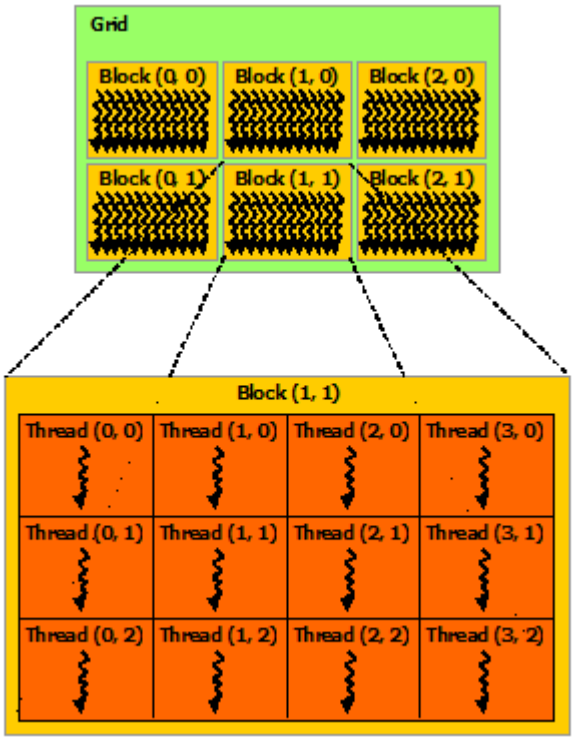

Figure 2.1: The relation between CUDA blocks, the grid and each thread. The figure is taken from [4].

data from subsequent memory locations or if the needed memory is not properly aligned with the cache reading lines, the warps will break down since sequential cache reads need to be made [3]. When memory reads can be done in parallell, they are called *coalesced* [4].

GPUs have different kinds of memory [4]. The GPU has some DRAM, called *global memory*, and some multiprocessor-associated caches and shared memory. Compared to the last two, the DRAM is very slow and transfers to and from this will likely be the largest bottleneck in the entire CUDA-enabled application [3]. Kernels can define local variables which will be local only to the thread, but these will be put in the scarce shared memory, which is distributed evenly amongst all the threads in the block. This may put an upper limit on the allowable number of threads per block, depending on the amount of local variables [4].

If any threads are to update the same memory location at the same time, there will be additional concerns about ensuring that the threads wait in turn. This will introduce additional latency [4].

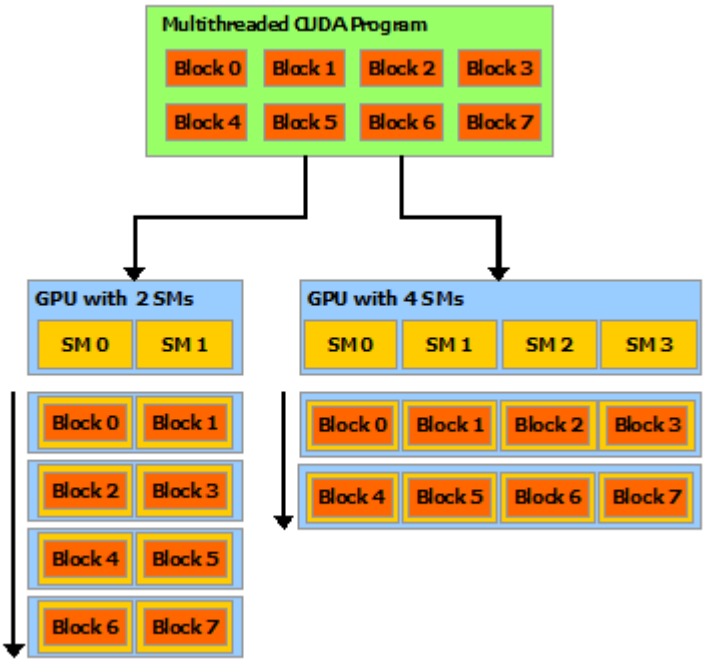

Figure 2.2: Distribution of blocks over the multiprocessors. Here, they are called *streaming multiprocessors*. The figure is taken from [4].

2.2 Absorption and scattering mechanisms of photons in human tissue

Photon scattering will happen if photons encounter materials with different refraction indices than the rest of tissue, for example collagen fibres or red blood cells.

Single-scattering theory is assumed to be valid. The scattering from a single particle can be described by its scattering cross section, σ_s, meant to denote the amount of power scattered in all directions [17]. If the mean distance between scattering particles is greater than both scatterer size and the wavelength, the scattering events can be considered independent and single-scattering theory can be assumed [46]. In case of a homogenous medium, the scattering can be described by its scattering coefficient, $\mu_s = \sigma_s \cdot N_s$, where N_s is the number density of scatterers. Cells are the largest structures in tissue, 10 microns in size [46], the packing of which will greatly depend on the cellular soup in question, but biological materials are in general assumed to be sparsely distributed materials for simplicity. Rayleigh and Mie theories can give measures of the scattering cross section [46]. Rayleigh theory can be applied if the particles are much smaller than the wavelength, and Mie theory will be valid regardless of the particle size [46].

The energy of a photon may be absorbed by a particle and re-emitted as a new photon, or in part cause the particle to enter a vibrational state. This process can similarly be described by an absorption cross section σ_a and an absorption coefficient μ_a, but with no reservations about sparse distributions [46]. If the medium in question is a non-scattering medium, the transmittance through the medium can be expressed as

$$T(x) = e^{-\mu_a x}, \tag{2.1}$$

but this will generally not be valid if the medium is a scattering medium. Absorbing materials will generally be referred to as chromophores throughout this report.

When the light undergoes many scattering events, it will lose coherence and polarization can be neglected. Taking the above properties into account and accounting for all scattering and absorption losses, the situation may be summed up in the Boltzmann equation for photon transport, alternatively known as the radiative transfer equation. Its derivation can be found in [46], only the time-independent result will

be presented here:

$$\hat{s} \cdot \nabla L(\vec{r}, \hat{s}) = \mu_s \cdot \int_{4\pi} L(\vec{r}, \hat{s}') p(\hat{s}' \cdot \hat{s}) d\Omega' - \mu_t L(\vec{r}, \hat{s}) + S(\vec{r}, \hat{s}) \tag{2.2}$$

The radiance L is the energy flow per unit normal area per unit solid angle directed in direction $\hat{s}$ at the position $\vec{r}$. The probability of scattering in the direction $\hat{s}'$ when the photon originally had the direction $\hat{s}$ is described by p, the phase function. The extinction coefficient μ_t is equal to $\mu_a + \mu_s$, and the last term, S, is the source term.

The left-hand side is loss due to divergence of the beam. The integration involving L and p is the main scattering term, wherein photons are scattered in some direction as according to the phase function. The extinction coefficient μ_t is loss due to extinction, and the source term represents new photons entering the position $\vec{r}$ in the direction $\hat{s}$.

The Henyey-Greenstein phase function is often used for biological materials to describe the phase function p. This was originally used by Henyey and Greenstein [16] to describe diffuse radiation in galaxies, but was found by Jacques et al. [33] to be a good fit also for the angular dependence of the scattering in biological tissues, at least at the wavelength 632 nm. This phase function is given as

$$p(\hat{s}' \cdot \hat{s}) = p(\cos\theta) = \frac{1 - g^2}{2(1 + g^2 - 2g\cos\theta)^{3/2}}. \tag{2.3}$$

θ is the angle between the original photon direction and the scattered photon direction, while g is the anisotropy factor, defined as

$$g = \int \cos\theta p(\cos\theta) d(\cos\theta), \tag{2.4}$$

i.e. the average of the cosine of the scattering angle. Using the Henyey-Greenstein phase function for tissue is widely accepted to be correct and has never been cause for much scrutiny. Criticism was attempted by Binzoni et al. [8], but it turned out they were mistaken in their approach [7].

The two most popular solutions to (2.2) are Monte Carlo and diffusion theory. Monte Carlo will rely on the phase function, while whether or not it will be used in the diffusion model depends on the chosen source function. Both solutions will in any case involve g.

2.3 Monte Carlo

The derivation of the Monte Carlo program is best described by referring to other sources like Wang et al. [45] and [46], but it will be summarized.

In essence, a Monte Carlo-implementation of (2.2) will involve tracking each photon packet and accounting for absorption and scattering losses as according to the mechanisms described by equation (2.2). Any new photon directions are decided by picking random numbers following the probability distribution in (2.3). A figure displaying the program flow for the Monte Carlo package MCML is shown in figure 2.3. The step size is represented by s. This is sampled from a random distribution which describes the likelihood of encountering an absorption or scattering event, hence the logarithm of an uniformly distributed number ξ between 0 and 1. Fresnel's equations will be used to describe the reflection and transmission if the particles hit the boundary between two layers. Otherwise, they are moved as according to the step size. The weight of the photon packet is reduced as according to absorption and back-scattering determined by μ_t, and the new scattering direction is determined by sampling a random variable from (2.3). The program moves on to the next photon if the photon weight is zero. If the photon weight only is small, the photon will go through a russian roulette deciding whether it will live or be taken out of the simulation. The roulette is a part of the program because following photons with small weights yields little interest and should be terminated as early as possible. The most of the photons are eliminated from the simulation once the photon weight becomes small, and the roulette keeps a few of them alive. This ensures energy conservation, compensating for the energy lost.

If the photons come back up through the first layer, they are registered in a reflection array. If they are absorbed, they are registered in an absorption array. If they end up at the other side of the simulated slab, they are registered in a transmission array. At the end of the day, these arrays are summed up to yield diffuse reflectance, absorbance and transmittance of the simulated light.

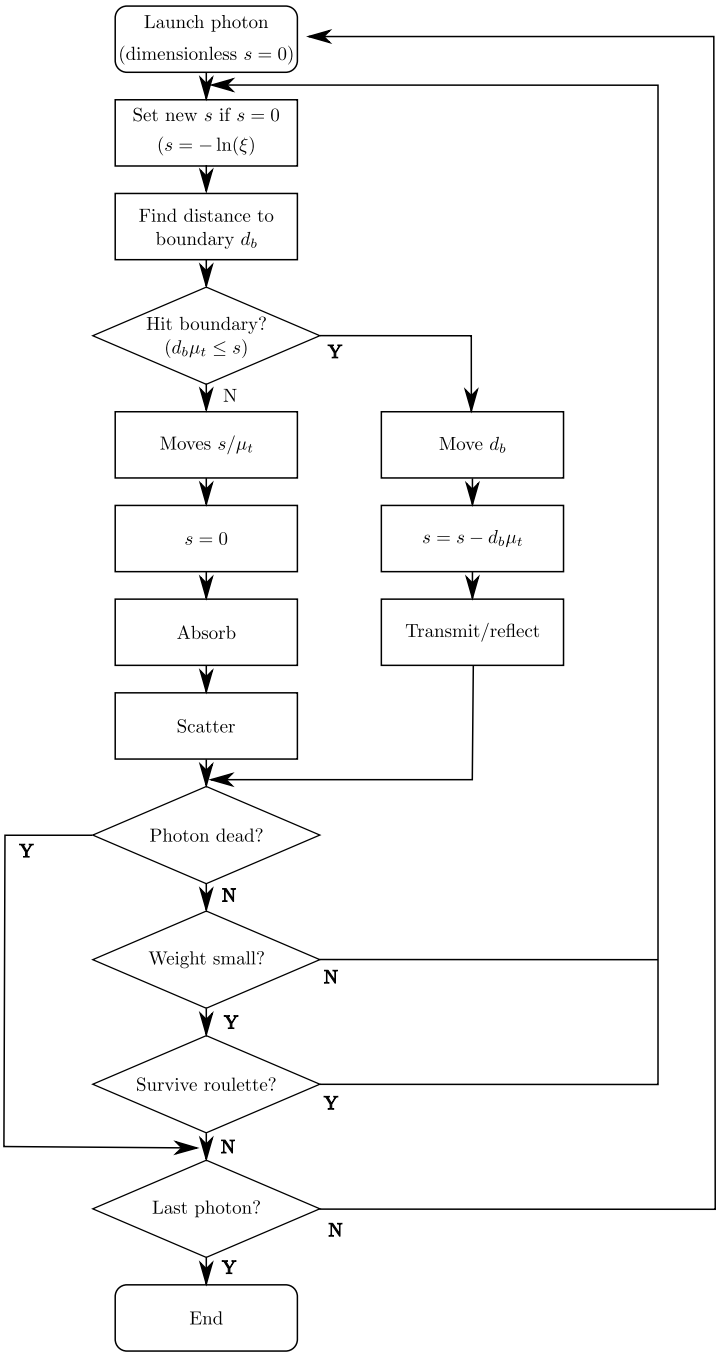

Figure 2.3: The program flow in MCML. Reproduced in Inkscape from Wang et al. [45].

2.4 Diffusion model

Equation (2.2) can be integrated over all solid angles in order to yield a continuity equation which may used later on. The integrated L in the phase function integral will be independent of Ω since it will become the isotropic ϕ after integration, and the phase function will be integrated away to 1 as ϕ can be taken outside of the integral [14]:

$$\nabla \vec{j}(\vec{r}) = -\mu_a \phi(\vec{r}) + q(\vec{r}) \tag{2.5}$$

The integrated L and $L\hat{s}$ have respectively been replaced by the fluence rate ϕ and the diffuse photon flux vector $\vec{j}$, while the integrated source function S is has been replaced by q [41].

Equation (2.2) has no analytical solutions, but approximations can be made. The radiance L can be assumed to be almost isotropic if μ_s is much larger than μ_a, and the radiance can then be written as the first two terms of a Legendre polynomial expansion [41]:

$$L = \frac{\phi}{4\pi} + \frac{3}{4\pi}\vec{j} \cdot \hat{s}. \tag{2.6}$$

The first term represents the isotropic part while the second term is the deviation from isotropy in the direction given by the unit directional vector $\hat{s}$ [41]. One main assumption is that the deviation from isotropy is not too large, or else higher order terms would have to be included. The photon flux j should therefore not be too large when compared to ϕ.

If this is inserted into (2.2), the resulting equation multiplied by $\hat{s}$ and integrated over all solid angles will be [14]

$$\vec{j}(\vec{r}) = -D\nabla\phi(\vec{r}, t), \tag{2.7}$$

where D is defined as $\frac{1}{3[(1-g)\mu_s + \mu_a]}$. This is Fick's law [14], hence "diffusion" in "diffusion model". The source term disappears from (2.2) when multiplied by $\hat{s}$ and integrated over all solid angles if it is assumed to be isotropic, which is why it no longer is a part of the equation.

Equations (2.7) and (2.5) can be combined to yield [41]

$$\nabla^2\phi - \frac{\phi}{\delta^2} = -\frac{q}{D}, \tag{2.8}$$

where $\delta = \sqrt{\frac{1}{D\mu_a}}$, also known as the optical penetration depth. This equation can be applied for each layer in the skin model, each with its own depth-varying source function. The solution will depend on some boundary conditions at the skin-air surface:

When L from earlier on is integrated over half the solid angle, the result is the irradiance E, the power passing through an unit area. Integrating (2.6) results in [41]

$$E = \frac{\phi}{4} \pm \frac{j}{2}. \tag{2.9}$$

The plus or minus sign depends on whether the photon flux $\vec{j}$ is pointing along or opposite of the surface normal for the surface over which the irradiance is passing through. This can be used for constructing boundary conditions between differently scattering layers, since the irradiance must be the same on both sides of the layer boundary [41]:

$$\frac{\phi_1}{4} + \frac{j_1}{2} = \frac{\phi_2}{4} + \frac{j_2}{2} \tag{2.10}$$

$$\frac{\phi_1}{4} - \frac{j_1}{2} = \frac{\phi_1}{4} + \frac{j_1}{2} \tag{2.11}$$

j_1 and ϕ_1 are the flux and fluence rate in layer 1, and j_2 and ϕ_2 the flux and fluence rate in layer 2. The irradiance should be continuous regardless of which way the surface normal is pointing, which is why there are two boundary conditions, one for the case where the surface normal has the same direction as $\vec{j}$, and one for the case where the surface normal points in the opposite direction.

The start boundary condition coupling everything together is the boundary condition at the air-skin interface. By integrating the Fresnel reflection coefficient over all angles of incidence, an R_ϕ and R_j can be found [14]. The reflected part of the irradiance at the air-skin interface must be related to the irradiation propagating back into the skin by [41]

$$R_\phi \frac{\phi}{4} - R_j \frac{j}{2} = \frac{\phi}{4} + \frac{j}{2} \tag{2.12}$$

This can be summed up in a boundary condition $j = A\phi$, where A depends on R_ϕ and R_j. By using the formulas for R present in [14], A can be calculated to be 0.14 for $n = 1.4$.

The criterion that the flux should be much smaller than the fluence rate is therefore fulfilled only to a limited degree at the skin surface [41], but it can be argued that it still will work nicely out [41].

One important concern is the source function. If isotropic source functions throughout the layers is assumed, it may be expressed, for a two-layer model, as [41]

$$q_1 = P_0 \mu_{s,1}(1 - g_1)e^{-\mu_{t,1}x} \tag{2.13}$$

for $0 \leq x \leq d_1$, and

$$q_2 = P_0 \mu_{s,2}(1 - g_2)e^{-\mu_{t,1}d_1} e^{-\mu_{t,2}(x-d_1} \tag{2.14}$$

for $d_2 \leq x \leq \infty$. $\mu_s(1-g)$ is often abbreviated as μ_s', the reduced scattering coefficient. After enough scattering events and little to no absorption, this will be the effective scattering coefficient and can in such cases be used instead of the actual scattering coefficient. Indices 1 and 2 refer to the upper skin layer and the lower skin layer, respectively. P_0 is the incident intensity after specular reflection at the air-skin surface. The solution to (2.8) may now be found for each layer by inserting the source functions and applying the boundary conditions. The solution to ϕ will not be presented here, but is displayed in [41].

The diffuse reflection coefficient will be expressed by

$$\gamma = \frac{j(x=0)}{P_0} \tag{2.15}$$

The solution for γ using the isotropic source function is in its entirety presented in [41]. This is used in the implementation of the diffusion model on the GPU, since it is an analytic expression and differentiable. The MATLAB implementation of the diffusion model implemented by Spott, Randeberg and others will use inversion of matrices presented in [34], which while instructive and more manageable, will not be optimal in a GPU implementation, as will be seen.

There are also alternatives to using the isotropic source functions. One alternative is the Delta Eddington source function, presented by Spott and Svaasand [35]. The diffusion approximation is to neglect the higher orders of non-isotropy, which when μ_s is no longer much larger than μ_a will no longer be valid. The Delta-Eddington source functions seek to remedy this by keeping some of its non-isotropy. It is derived using the Henyey-Greenstein phase function, shown in [35]. It is also shown to have better correspondence with MCML [35].

Implementation-wise, the analytic function for γ derived using this source function is not as available as for the isotropic case, and only the isotropic variant of the diffusion model was implemented on the GPU for this project. However, Delta-Eddington was still tested against GPU-MCML for different parameters, as will be seen. Delta-Eddington will deposit its photon deeper into the tissue. When the absorption becomes too strong, it is thought that Delta-Eddington will break down, and it will be more fair to use the isotropic variant for all cases.

2.5 Skin model

The two-layer skin model employed is described by Spott et al. [34], Randeberg et al. [28, 21] and Svaasand et al. [41], but will be summed up here.

Two or three-layered skin models are usually deemed sufficient, even if the skin models used for simulating diffuse reflectance might sometimes have as many as seven layers [5]. Due to errors in the simulation methods and many shortcuts in the description of the scattering and absorption mechanisms, the extra amount of layers will only cause uneccessary complications and not extra realism. Two layers were used throughout this project report for simplicity. The two layers in question are epidermis and dermis, and in case of a three-layer model, dermis is subdivided into two new layers. Hypodermis is never taken into account as photons scattered back from such depths is negligible [21].

In reality, both epidermis and dermis will be sub-divided into different layers with different properties, but as an approximation, these properties are assumed uniformly distributed in each layer. Each layer is flat and uniform, but the epidermis will in reality reach partially into the dermis through the papillae. A small fraction of blood, 0.02%, is therefore included in the epidermis even if epidermis does not contain blood. See figure 2.4.

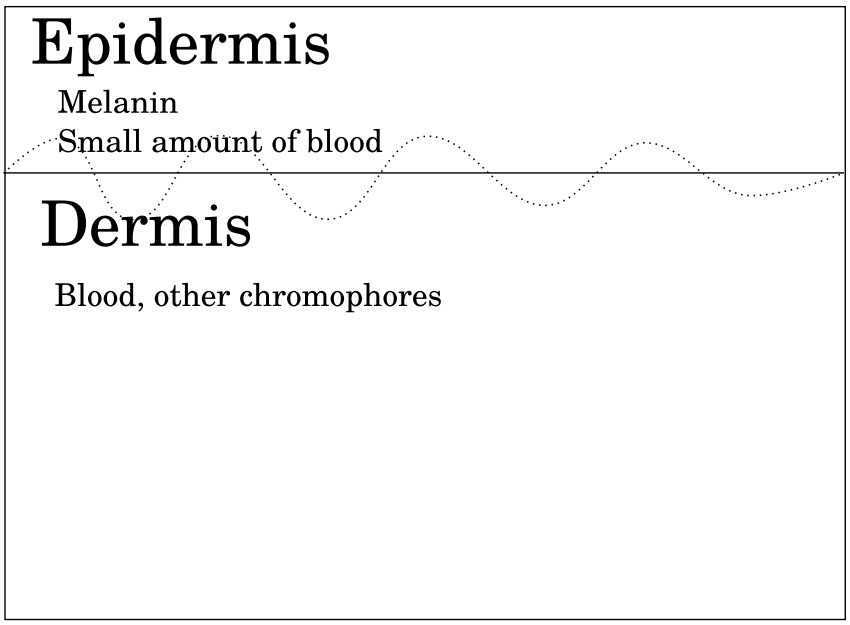

Figure 2.4: Two-layer skin model. The papillae is shown as the dotted line. The papillae are not present in the model, but are accounted for through the small amount of blood in the epidermis.

2.5.1 Scattering

Scattering will predominantly rise from the presence of collagen fibers in the skin, which will induce both Rayleigh and Mie scattering. The wavelength dependency was found by Saidi et al. [31] to be

$$\mu'_{s,t} = C_{Mie}(1 - 1.745 \cdot 10^{-3}\lambda + 9.843 \cdot 10^{-7}) + C_{Rayleigh}\lambda^{-4}. \tag{2.16}$$

This is the reduced scattering coefficient. C_{Mie} and $C_{Rayleigh}$ are some constants. In order to get the non-reduced scattering coefficient, one will have to divide the Mie-part of the scattering by $1-g$ and the Rayleigh-part by 1.0, since Rayleigh scattering is isotropic and will not be reduced when considering the effective scattering coefficient. The anisotropy factor was for skin found by van Gemert et al. [43] to be

$$g = 0.62 \cdot 29 + 10^{-5}\lambda. \tag{2.17}$$

Age-dependencies of C_{Mie} and $C_{Rayleigh}$ are determined by Saidi et al. [31] for newborns. The reported curves were extrapolated into adulthood by Randeberg and the values respectively found to be 10500 m^{-1} and $1.05{\cdot}10^{14}$ $\mathrm{nm}^4\mathrm{m}^{-1}$. Newer versions of this scattering function exist, but they were not implemented for this project.

There will also be scattering from red blood cells, as presented by Ishimaru [17]. The scattering coefficient may be written as [17, 40, 39]

$$\mu_{s,b} = \sigma_{s,b} \frac{H(1-H)(1.4-H)}{v_e} \left(\frac{685}{\lambda}\right)^{0.37}. \tag{2.18}$$

The scattering cross section $\sigma_{s,b}$ was reported by [17] to be $55.09 \cdot 10^{-12}$. The volume of the red blood cells, v_e, is claimed to be $1.26 \cdot 10^{-16}$ m^2 by Spott et al. [36]. H is the haematocrit, the fraction of red blood cells to the total volume of blood. The haematocrit will be 40-52% for males and 38-48% for females [30]. Blood is sometimes reported to have a specific g_b, but for fairness, the g in (2.17) is used for everything throughout this project. Besides, as will be seen, the Monte Carlo simulations will have no way of differentiating between different gs. The blood scattering will also be incorrect for bruises or higher blood concentrations, since there will be different packing of the red blood cells. Using the exact, correctly reduced version of (2.18) will matter less.

The total scattering coefficient for each layer will be

$$\mu_{s,e} = B_e \mu_{s,b} + (1 - B_e \mu_{s,t}) \tag{2.19}$$

$$\mu_{s,d} = B_d \mu_{s,b} + (1 - B_d \mu_{s,t}). \tag{2.20}$$

B_d and B_e are the blood volume fractions (BVF) for dermis and epidermis respectively. B_d will later be referred to as the BVF.

2.5.2 Absorption

Absorption in skin happens mainly due to deoxygenated hemoglobin, oxygenated hemoglobin, methemoglobin, melanin, bilirubin, betacarotene and water. Throughout this project report, methemoglobin, bilirubin and betacarotene are not included in the simulations or used at all since the constituent-separating part of the inverse chain was not finished.

The presence of the absorption spectrum in the visible of each possible constituent will still be mentioned. Bilirubin absorption is strong from 400 nm to 520 nm and resides in the dermis. Betacarotene absorption is strong from 400 nm to about 520 nm and resides in the fatty parts of the dermis and in stratum corneum, the outer-most part of the epidermis. It will however, once included, be included in the dermis only since the stratum corneum is too thin to be implemented in the diffusion model. Water has a very low absorption in the visible parts of the spectrum and can mostly be neglected.

Zijlstra [50] will report absorption spectra for both deoxy and oxy hemoglobin which are thought to be accurate, but these were not used since they are only available from 450 to 800 nm. Instead, values measured by Thorsten Spott some time in the past were used, and are displayed in figure 2.5. The ratio of oxy hemoglobin to deoxy hemoglobin is called the oxygen saturation and will be denoted as O. The absorption of blood will be written as

$$\mu_{a,b} = \mu_{a,Hb} \cdot (1 - O) + \mu_{a,HbO_2} \cdot O. \tag{2.21}$$

An isosbestic point is where the absorption coefficients of deoxygenated and oxygenated hemoglobin are the same. The total absorption coefficient of blood will here depend only on the total fraction of blood to the rest of the constituents.

Melanin is the main absorber in epidermis. The melanin content will be quantified by the melanin absorption coefficient at 694 nm. Some will report melanin contents by using fractions, but since the melanin will not be evenly distributed, going from fractions to the actual melanin absorption will be difficult. Instead, the melanin absorption at 694 nm is by itself used to indicate the amount of melanin present in the epidermis. Sun-protected european skin will, for example, have a melanin absorption coefficient at 694 nm in the range of 280 to 325 m^{-1} [41]. The wavelength dependence of melanin absorption in skin was investigated by [47], and again observed by Spott et al. [36] to be

$$\mu_{a,m} = \mu_{a,m,694} \cdot \left(\frac{694}{\lambda}\right)^{3.46}. \tag{2.22}$$

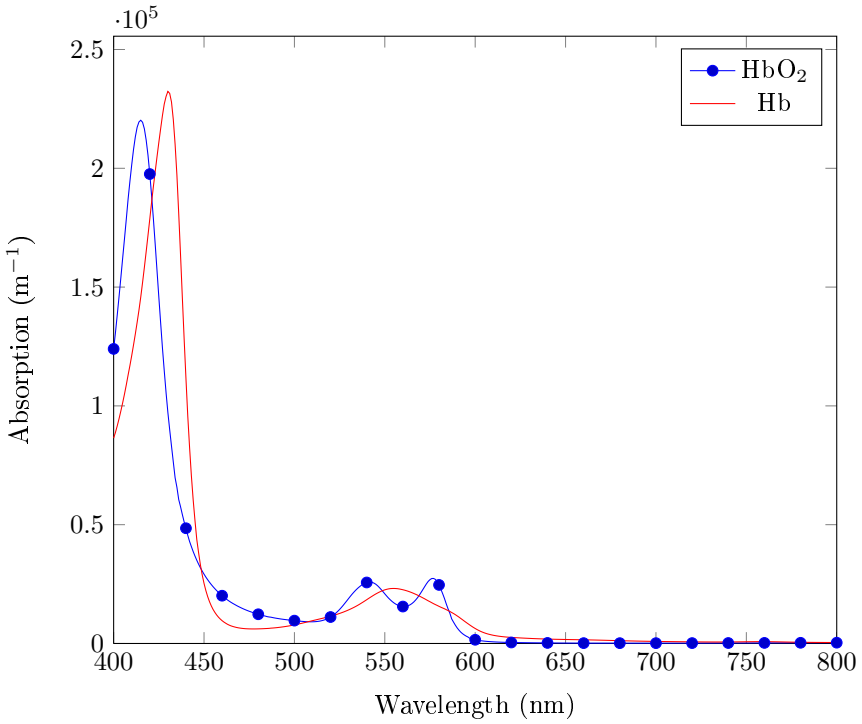

Figure 2.5: Absorption spectra for deoxy and oxy hemoglobin, measured by Thorsten Spott.

In addition, a constant background absorption of $\mu_{a,o} = 25$ m^{-1} will be assumed, as Spott et al. [36] did. Later developments has some wavelength-dependence incorporated in this factor, but this was not implemented for this project report.

The absorption properties of each layer can be summed up as

$$\mu_{a,e} = \mu_{a,b} \cdot B_e + \mu_{a,m} + \mu_{a,o} \cdot (1 - B_e) \tag{2.23}$$

$$\mu_{a,d} = \mu_{a,b} \cdot B_d + \mu_{a,o} \cdot (1 - B_d) \tag{2.24}$$

Other chromophores will be located in dermis.

2.6 Iteration methods

Monte Carlo is a black box, and iteration can only be done in a less optimal way. Intuitively, the diffuse reflectance will be lowered when the absorption is heightened, and this represents the entire philosophy behind the iteration strategy employed. The absorption is changed in a way that should lower or heighten the diffuse reflectance towards the desired value, and once the calculated diffuse reflectance jumps over the desired value, the algorithm will jump back and reduce the step length and continue until some kind of convergence.

For the diffusion model iterations, the analytical derivative is available and used in Newton's method. This is given as [26]

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}. \tag{2.25}$$

Newton's method can have some convergence problems if the estimate x_n is far from the true value x or when $f(x)$ has a very strange behavior or local maximums or minimums, and special care might possibly have to be taken depending on the problem [26].

Chapter 3

Method

The inversion method, GPU-MCML and the implementation of the diffusion model inverse chain for hyperspectral images is presented. First, the GPU-MCML framework developed by Alerstam et al. [5] is presented, along with the modifications. Then the modifications made to MCML are summed up, and then GPU-DM, the inverse chain implemented on the GPU is presented by going through the philosophy behind, the optimizations and the program flow.

After the simulation methods have been presented, the inversion strategy is presented by explaining the philosophy and the unknown parameters needing determination. The parameters are the blood volume fraction, the melanin content and the dermal absorption for each wavelength. There exists several methods for determining the former two parameters, and several of these are presented.

3.1 GPU-MCML

Several attempts at developing an inverse Monte Carlo model has been attempted [49, 48, 32, 25, 15, 13, 9, 6], but none of these rely on the Monte Carlo model directly, they all rely either on scaling pre-simulated results, on look-up tables or on empirical models.

Monte Carlo is too slow for anything else. Either, one will need to make the Monte Carlo method faster, or one will need to employ different and faster, but allegedly less accurate models like the diffusion model. Iterating the Monte Carlo model directly has proved less than feasible in the past. The approach of [49, 48, 32, 25, 15, 13, 9, 6] has also been to assume all parameters unknown, and define a scheme where the model needs to be iterated with respect to more parameters than one, increasing the required computation time. Regardless, even with only one parameter, iterating the original Monte Carlo model is not feasible. One simulation will take 20 minutes, one iteration will never be enough. A hundred iterations is more likely, increasing the computation time to 2000 minutes.

Parallelizations of Monte Carlo simulations on GPUs have previously been worked on by various research groups. The main motivation for parallelization on GPUs is that each photon package can be simulated independently. The work done on each photon package will be very similar, and this behaviour is ideal for GPU parallelization. A description of the implementation used in this project report can be found in Alerstam et al. [5]. They have also identified the main bottleneck of this Monte Carlo approach, which is writing to the absorption array. As photons are terminated throughout the simulation, their weights will be recorded in a large absorption array which will be shared for all the threads. The large absorption array has to be saved in the slowest memory block, the global device memory. Access will be slow.

The threads cannot write to the same array positions at the same time, which they will do if two photons are absorbed at the same position. One thread at a time can write to it and the rest have to be stalled. Since the global memory is rather slow, this will be slow, and although the problem is easily parallelizable and the GPU is more than up for the processing task itself, the memory writes will slow down the whole thing. The solution Alerstam et al. [5] proposes and implements, is to allocate some of the faster registers

to the most frequently written parts of the array close to the source, or dropping the absorption array altogether if only the reflectance or transmittance is desired.

As only the reflectance will be measured, only the reflectance will be needed in the simulations, and this solution can be employed in order to maximize the processing speeds. Memory writes will happen only to the reflection array. Unfortunately, it is on the other hand difficult to do this in a coalesced way.

The internals of GPU-MCML were not tampered much with. The **main()**-function was rewritten to a general function called **run_gpumcml()**, callable from other C-functions. It was rewritten to accept a struct describing the optical properties of the layers instead of reading them from a file. This was done by "hacking" the file-reading function to read its properties from this struct instead of from a file.

Some bottlenecks in the initialization was also removed by reading the random seeds used for the random number generator from file only once at the first wavelength. An inverse model was created as according to the inverse strategy and black box iteration.

3.2 MCML

MCML was compared against GPU-MCML. Also in this case, the function reading in the input files was "hacked" to accept the same struct as the one used by GPU-MCML, and the main function was altered to loop through the whole wavelength range instead of looking at one file only. Others have created PERL-scripts for preparing input files for the whole spectral range, but the author of this project report found changing the functions to do what he wanted easier and less time-consuming.

3.3 Camera and computer hardwaree

The hyperspectral camera the programs will be developed for is a HySpex VINIR-1600 camera, developed and manufactured by Norsk Elektro Optikk. This is a line scanning camera using a push-broom technique. Including autofocusing, the camera will use 30 ms per line. Each line consist of 1600 pixels and 160 wavelengths (bands). Camera data is transferred over the TCP/IP-protocol.

The GPU programs were tested using an NVIDIA Quadro FX 3700M graphics card. It has a lower-end GPU with 16 multiprocessors and compute capability 1.1. Low compute capability means care needs to be taken in optimizing memory access. It will also lack important features present in newer GPUs, and application runtimes are expected to decrease for GPUs with a higher compute capability.

3.4 GPU-DM

The diffusion model has an analytical expression that may be evaluated efficiently using few processor instructions, as opposed to GPU-MCML, which will have to be evaluated sequentially for each wavelength and iteration due to the fact that all GPU processing capabilities already is handling the photon simulations for one wavelength at a time. The aim is to implement the diffusion model as an inverse model on the GPU and process the full line of hyperspectral data at a time within the time limits defined by the streaming capabilities of the hyperspectral camera.

This is a parallellizable problem. Once the epidermal absorption and the blood volume fraction are estimated, all parameters except for the dermal absorption will be known. The dermal absorption coefficients can independently be derived from each wavelength and each pixel with no interference or crosstalk whatsoever. All wavelengths and pixels will be independent, and the inversion of these can evenly be distributed amongst the multiprocessors on the GPU.

The threads cannot do a complete iteration of the diffusion model with respect to the input reflectance by themselves. This will cause the different threads to do different calculations. As mentioned in earlier sections, threads cannot be divergent, or the warps will break down. Iterating in this way, while possibly

ending some iterations earlier and making room for new pixels and wavelengths, will minimize GPU performance.

However, the forward model can be implemented in a GPU-friendly way. Assuming all known parameters have been calculated for each pixel, the forward model can be run for all pixels and wavelengths in an independent and deterministic way which will cause maximum parallellization to occur. Then, again inpendently and deterministically for all pixels and wavelengths, the output reflectance and hyperspectral intensity can be compared and the next dermal absorption may be calculated. The process may then be repeated until convergence across all pixels and wavelengths.

The chosen iteration method was Newton's method, because of the availability of the derivative of the diffusion model, its fast convergence, simplicity and deterministic behavior, ideal for GPU parallellization. The reflectance will also be an optimal function for Newton's method. Once close to the target, Newton's method will be fast to converge, but if the curve that Newton's method will be iterated with respect to has local maxima or minima, it may instead choose to diverge [26]. This is not the case with the reflectance as a function of the dermal absorption, it will be well-behaved. The derivative of the diffuse reflectance with respect to the dermal absorption coefficient is too complicated for proving the monotonicity of the diffuse reflectance with respect to the dermal absorption coefficient, although any plot will show that it is well-behaved and will drop off in a monotonic way, as shown in figure 3.1.

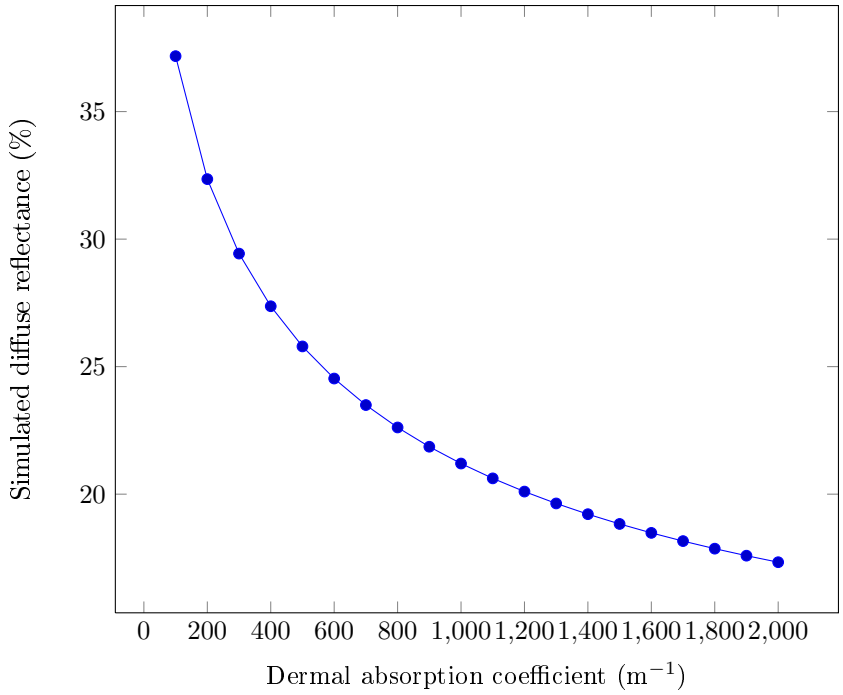

Figure 3.1: Simulated reflectance as a function of the dermal absorption coefficient with $\mu_{a,mel,694} = 250$, $o = 0.8$ and bvf $= 0.01$ at $\lambda = 500$ nm.

There are some problems with this approach. Some of the pixels and wavelengths will converge much faster and waste GPU processing power on what essentially will be a busy wait, if they have an initial dermal absorption coefficient closer to the true value than the others. If a pre-determined amount of iterations across all wavelengths and pixels is to be done, some of the values will therefore be more accurate than the others. Keeping track of the error is not desirable due to the fact that making the threads write to a common error variable will cause them to stall each other for sanity, causing too much lag. There will be less control over the error and convergence rates. However, the model is in itself inaccurate, and matching the model exactly against the measured reflectance might not have any merit. Elsewise, as argued above, Newton's method should always converge, and due to Newton's method's quadratic converging abilities [26], it will converge fast. It will be seen that 10-20 iterations should be more than good enough for a good estimate, no matter the starting point.

Checking convergence would also not be a good measure overall for the whole image. It will not be possible to let the image consist only of skin. Anything else than skin, like clothing or bedsheets, cannot be matched by the diffusion model when the input parameters are for skin only. If the image largely consists of everything else than skin and the erroneous model fit somehow converges faster than the actual skin fit, it might give a false positive regarding the convergence. Trying to separate out the skin in the image will take too much time, it will be more efficient to just inverse model the whole thing in one go and worry about model fits later on.

The system can at most manage the number of iterations achieved within the deadline as defined by the hyperspectral camera. By measuring the time taken for the calculation of one reflectance for all pixels and wavelengths and one iteration of Newton's method, this number of possible iterations can be decided in advance. The system may also continuously measure the relative time until deadline before each iteration and stop its iterations well within the deadline. In any case, the accuracy will then be determined by the power of the GPU, and with Newton's method's converging abilities in mind, this will in most cases be enough. In addition, since a new line of data is not ready at once, finishing early has no benefits.

With the philosophy of the inverse model straightened out, the details are now presented.

The line data from the hyperspectral camera will be one-dimensional and have the structure shown in figure 3.2. The array will be divided into the wavelength bands, and the sub-table within each

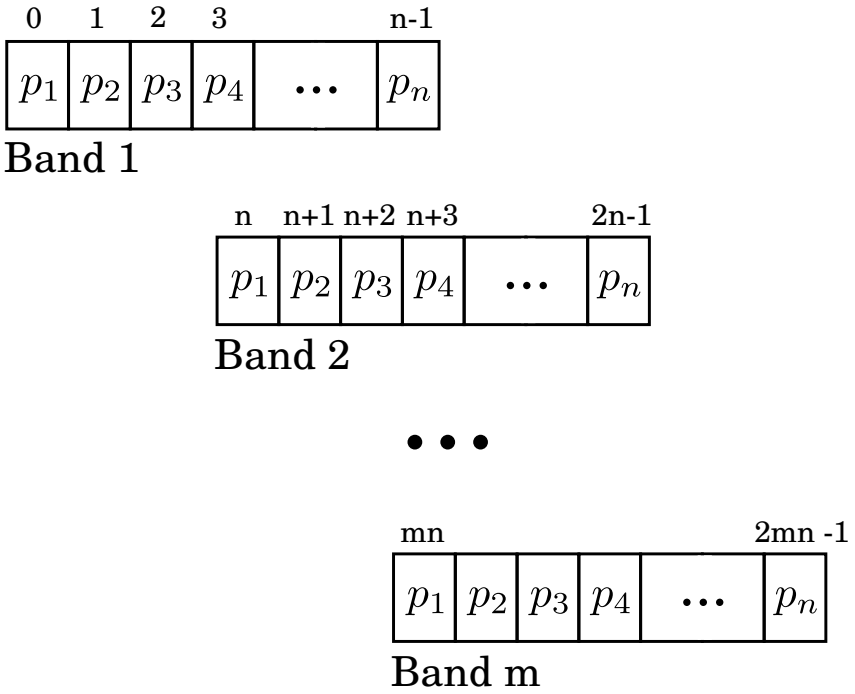

Figure 3.2: Illustration showing the distribution of bands and pixels within the one-dimensional data array arriving from the hyperspectral camera. Array indices are shown on top of the tables.

wavelength band will consist of the intensity values at each pixel. The formula for accessing pixel p at wavelength band b in an image with n pixels and m wavelengths, zero-indexation for both pixels, arrays and wavelengths provided, will be

$$index(p, b) = p + b \cdot n. \tag{3.1}$$

One main concern is how to distribute the pixels and wavelengths so that there always will be 32 reflectance values per multiprocessor, and how to ensure coalesced memory read/writes. There are many solutions to this, all involving how the the line data array should be accessed from each thread. The camera streams frames of 1600 pixels x 160 wavelengths, and 160 and 1600 are both multiples of 32, which places some constraints on how the bands and pixels may be distributed among blocks and threads.

The solution employed is illustrated in figure 3.3. The pixels in one band is distributed across the blocks in one grid row. This ensures that subsequent threads in one block also access subsequent columns in the data array, ensuring coalesced memory access. One band is associated with the blocks in one grid row,

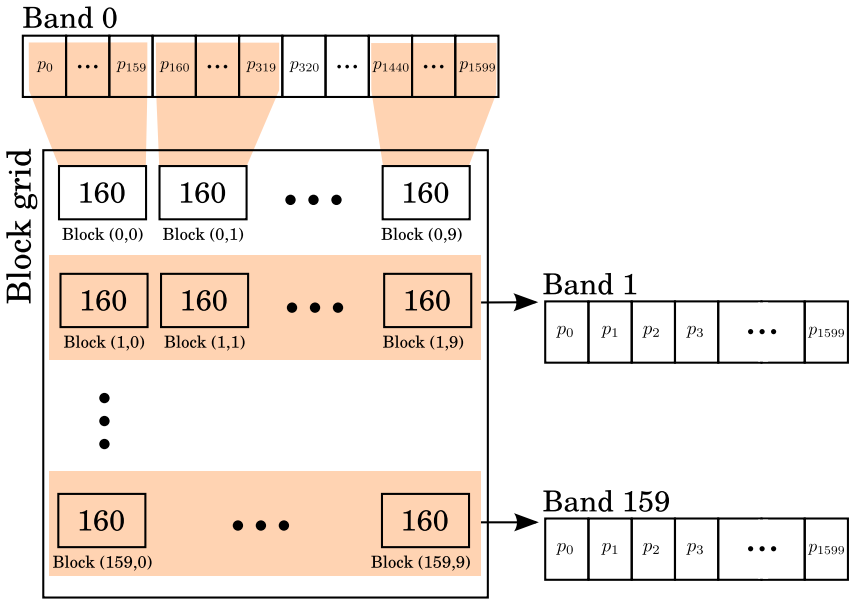

Figure 3.3: The distribution of pixels and bands across CUDA threads, blocks and grids. "160" in each block is meant to denote "160 threads". The blocks are one-dimensional, while the grid is two-dimensional.

with increasing wavelengths along the y-axis of the block grid. The blocks will each have 160 threads. The reflectance calculation requires a high amount of registers, and with the number of registers being a scarce resource, it was not possible to heighten the number of threads per block without causing the kernels to crash. 160 was also chosen because the number of pixels, 1600, will be divisible by 160. The number of threads per block will have to be modified if the number of pixels per line should change. The number of bands is of no concern.

The arrays had to be padded when the line data was transferred to the GPU and other arrays for optical properties were created. The starting address for the memory access had to be a multiple of 128, because of alignment issues with the cache lines [4]. With the above approach, this will in principle automatically be fulfilled, since all arrays in the implementation consist of the datatype float. Each float is 4 bytes. The first block will therefore access a range of 640 bytes. The next block will access the first byte after these first 640 bytes, and since 640 bytes is a multiple of 128, memory alignment is ensured. But as mentioned above, the camera specifications might be subject to change, and the chosen number of threads per block might not be optimal. Padding of the arrays by inserting some bytes after each block of memory corresponding to each block of threads to ensure memory alignment is therefore done by using **cudaMallocPitch()**. This will ensure that memory accesses still will coalesce if the number of bytes per block somehow is not a multiple of 128. It was pure chance that 160 would be a multiple of 32, $160 \cdot 4$ a multiple of 128 and 1600 a multiple of 160, and a different camera will not have the same fortunate properties.

Floating point numbers consisting of 4 bytes is also not an universal, platform-independent matter.

The program flow of the entire inverse simulation chain is shown in figures 3.4 and 3.5. The functions have been designed to be called from any framework providing hyperspectral data. Common to all frameworks is that they all will require an instance of the data structure **GPUDMArrays**, which is defined in **gpudm.h** and is a simple struct containing float-arrays that will be allocated in the GPU, along with some other useful variables describing properties concerning both camera and array structure. The arrays are allocated as pointers to some memory reference, and the functions instansiating the **GPUDMArrays** instance need not be CUDA-aware. All CUDA-awareness is contained and isolated

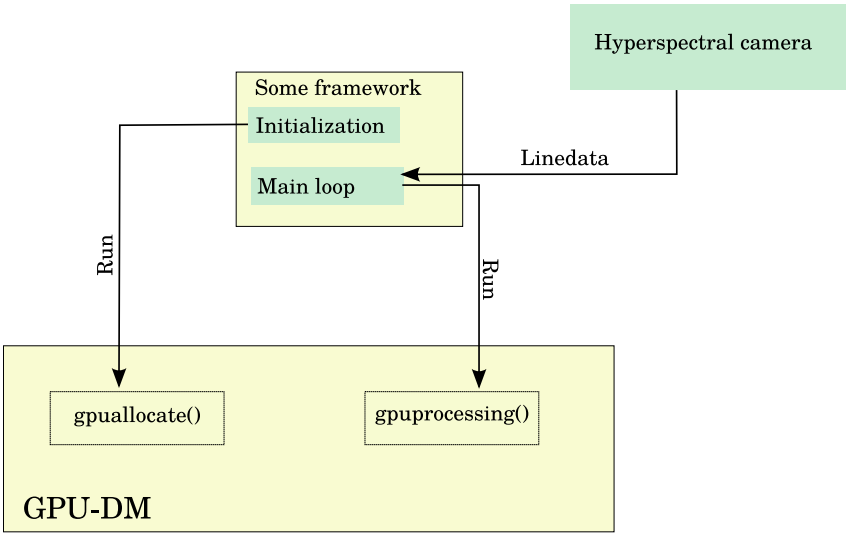

Figure 3.4: Program flow for GPU-DM in relation to a framework. Some framework will have to initialize the arrays using **gpuallocate()**, and it will have to continually input new linedata into **gpuprocessing()**, which will handle the rest.

within the file defining the GPU-DM-functions.

3.4.1 Allocation step

The CUDA functions for allocating memory in the GPU global memory, **cudaMalloc()**, **cudaMallocPitch()** and the like, are slow. The arrays needed for a complete iteration can hence not be allocated in the DRAM once every line arrives since this would produce an overhead larger than the deadline of 30 ms. This is therefore done in an initialization step, and the arrays will be reused. Some other arrays like calibration arrays and wavelength arrays are needed by **gpuallocate()**.

Two arrays for containment of hyperspectral data is allocated by **gpuallocate()**. One is for processing, and one is for buffering the next linedata.

The arrays containing the skin properties, which will be different for each pixel, are contained in arrays of the same size and structure as the hyperspectral line data. This will be a waste of memory since different wavelengths associated with the same pixel will have the same skin input properties. This was done because the skin properties would then have the same access pattern as the line data, which is optimal. As an alternative, one could have placed the skin properties in smaller arrays, but the memory access pattern would slow down the application. A solution would be to load the skin properties into a shared memory variable across the threads in the block, but this would require one thread to load the value and the rest to stall, most assuredly resulting in divergence and breakdown of GPU warps. With 160 wavelengths, 1600 pixels and 5 input parameters, this will in total be a memory waste of about 5 megabytes, but considering the major speedup of at least 32x, it is a small sacrifice to make.

The optical properties are contained in arrays of the same sizes, but this will be more natural since each wavelength and each pixel will contain different optical properties.

3.4.2 Continuous processing step

The data array containing the next line, a data array for bringing back the previous $\mu_{a,d}$ and the data structure containing the GPU-arrays is input to **gpuprocessing()**.

If this linedata is the first data line, **gpuprocessing()** will only transfer it to the DRAM.

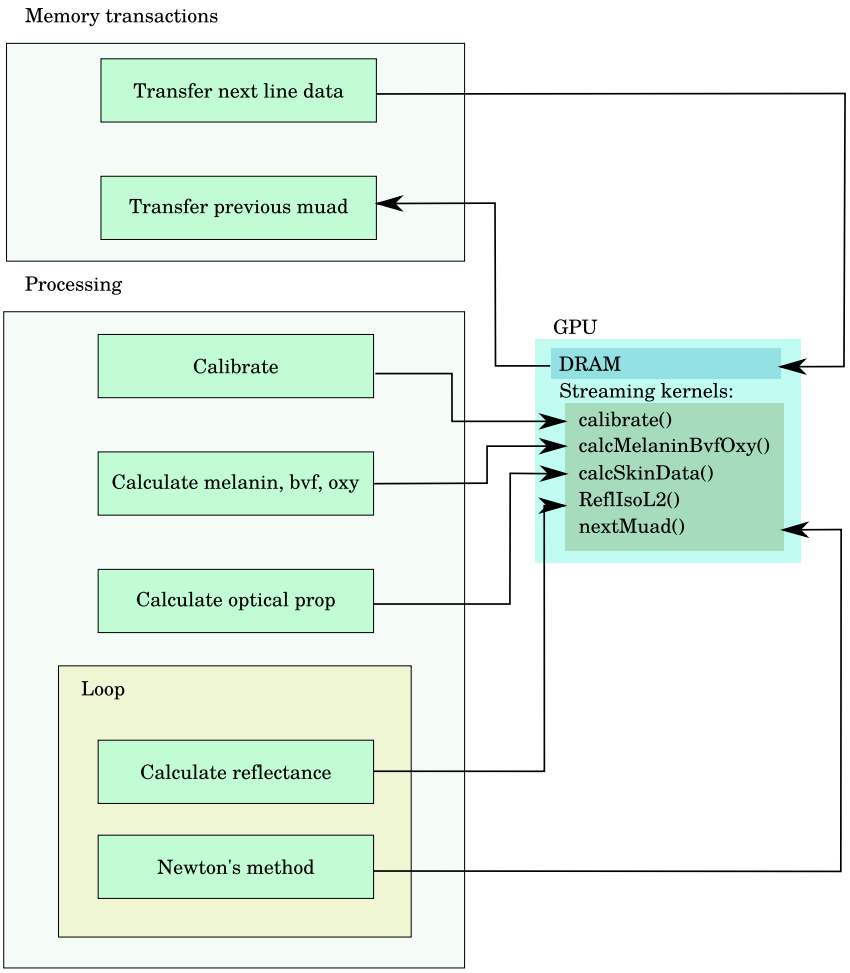

Figure 3.5: Program flow for **gpuprocessing()**. The memory transfers will be run concurrently with the processing part if possible.

If the linedata is not the first dataline, it will transfer it to the DRAM but additionally start processing the last linedata, transferred to the GPU the last time **gpuprocessing()** was called. By pagelocking the linedata to physical memory, some GPUs will be able to concurrently run kernels in addition to transferring data from the host to the GPU DRAM. Memory transferral between the host and the GPU can be slow, and concurrent memory transfer is desired. If this is not the second time **gpuprocessing()** is run, it will also transfer the previously calculated absorption coefficient back to the host, also concurrently if possible.

The function will move on to the processing once the memory transfer stage is initialized. The kernels are launched for each pixel and wavelength, in order:

1. **calibrate()** - Will divide all hyperspectral data by some calibration array
2. **calcMelaninBvfOxy()** - Will or should calculate epidermal melanin absorption, an initial estimate for the blood volume fraction and the oxygen saturation, as will be discussed.
3. **calcSkinData()** - Uses the initial estimates to calculate g, μ_a, μ_s for each layer, pixel and wavelength.
4. **ReflIsoL2()** - Calculates the simulated reflectance and its derivative.
5. **nextMuad()** - Using Newton's method, this function calculates the next $\mu_{a,d}$ using the derivative and reflectance calculated in **ReflIsoL2**.

All the kernels are heavily optimized except for **calcMelaninBvfOxy()**, in part because it never was implemented. One possible implementation is to select a few wavelengths and use these in relation to each other, which will cause the kernel to employ different memory access patterns than the rest and not achieve coalesced memory access. This kernel is likely to be a bottleneck.

Major optimizations that have been employed is first and foremost coalesced reads, but also trying to minimize as much global memory access as possible. If the same global memory position is needed more than once, its value is read into a local variable.

The optical properties will only be calculated once by **calcSkinData()**, but it is likely that minimizing calculations will result in lower throughput because of far more global memory reads. There were not enough registers available on each multiprocessor for testing whether or not the processing speed could heighten for **ReflIsoL2()** if it could calculate its own optical properties instead of having to read them from global memory.

The input data are in multiple, separate arrays. Navigating in the parameter list of each kernel function can be difficult. This was done because the data cannot be input as a struct to the GPU, as it will be very difficult to ensure correct memory alignment and coalesced memory access. The **GPUDMArrays** struct cannot be input to the GPU directly either. Only the float-arrays contained within the struct are allocated on the GPU, while the struct itself is a host memory allocated struct. The GPU will not know how to handle it.

The model will assume that the epidermal depth d_e is known. This is therefore defined as a global constant which will be compiled to appear in the GPU's fast, constant memory. The same goes for A and $\frac{1}{3}$, a common number.

The simplicity of the function calls and required data structure should enable GPU-DM to be integrated into any framework supplying hyperspectral line data. Speed concerns made the application less flexible, and when the time comes to implement the three-layer skin model or enable the model to handle other materials than skin, major changes will have to be done. Still, the changes will be within the general philosophy.

3.5 Inversion strategy

The inversion scheme employed for both models is based on previous inversion strategies applied on the diffusion model [36, 21, 28].

In order to make the optimization as simple as possible, it is desired to keep the number of possible parameters low. With the two-layer skin model, there will some unknown parameters:

- Melanin absorption in epidermis
- Oxygen saturation in the blood
- Blood volume fraction in dermis
- Depth of the layers
- Additional chromophores

The scattering coefficients are largely known and dependent on the blood volume fraction and are not mentioned as an unknown parameter.

Within the two-layer model, the depth of dermis is irrelevant as the dermis is semi-infinite. The depth of epidermis can be assumed to be known. Within the model presented by Randeberg et al. [28], it was set between 100 and 200 microns. In this model, 100 microns will be used.

Only the melanin absorption will be unknown in epidermis, since the blood volume fraction here is a skin model quirk and known beforehand. The artificial blood volume fraction in epidermis is so low that it is unlikely the oxygen saturation here does matter much. Thus, provided the scattering functions can

be trusted, the unknown parameters will be the blood volume fraction in dermis, as it is needed for the scattering functions, the associated oxygen saturation and any additional chromophores.

Except for the blood volume fraction, all of these parameters can be summed up as the dermal absorption coefficient. If the melanin content and the blood volume fraction are known, the dermal absorption coefficient will for each wavelength remain the single unknown parameter that can be inverted from the spectrum by iteration of the forward model. The constituents of the skin can be recognized by fitting the different absorption spectras against the acquired dermal absorption spectrum. Discrepancies and maladjusted fitting can be controlled and evaluated at leisure.

It might also be neccessary to somehow determine the oxygen saturation beforehand, in case forward simulations need to be done in order to obtain the epidermal melanin absorption. Oxygen saturation will then also be an unknown parameter requiring determination before inverting $\mu_{a,d}$.

A different inversion strategy is to assume all parameters unknown and fit them all at once, but this was not used since all chromophores will have to be accounted for beforehand. This approach is also less feasible in a GPU approach, since its increased complexity will cause the approach to be less deterministic.

3.5.1 Melanin absorption and blood volume fraction

The main iteration is done with respect to the absorption coefficient in dermis, and the absorption in epidermis will therefore have to be estimated in advance by using a different philosophy. Each reflectance value represents one equation with two unknowns if both the epidermal and dermal absorption coefficients are unknown, and will represent an under-determined system. Methods determining the epidermal absorption coefficient will have to use the general shape of the reflectance spectrum and be restrained by the shape of the epidermal absorption curve.

The main cromophore in epidermis is assumed to be melanin, and melanin has, as shown in equation 2.22, a simple shape which should have a recognizable impact on the diffuse reflectance spectrum. The whole diffuse reflectance will tend to be more steep when more melanin is present. There are some established methods for estimating the melanin absorption in this way. It will later be seen that some of these needed scrutiny, which is why several of them are presented here.

It turns out the result of heightened deoxy hemoglobin is interconnected with the result of heightened melanin absorption, and methods for determining this are therefore presented alongside with the melanin methods.

Dawson's indices have previously been used by Randeberg et al. for the melanin determining stage of the inverse chain.

Dawson's indices

One of the methods for determining melanin was introduced already in 1980 by Dawson et al. [10], and the most of the later methods are variations thereof. Dawson bases himself on some skin modelling theory where the logarithm of the inverse reflectance (LIR) can be written as the sum of the absorption and scattering coefficient of each layer of the skin. Dawson claims that the slope of the logarithm of the inverse reflectance from around 600 to 700 nm is proportional to the melanin content of the skin, and he develops a melanin index based on this slope. In a similar way, he also develops an erythema index for quantifying the blood content in skin. The melanin index is given as

$$MI = (-(\log(R(645)) + \log(R(650)) + \log(R(655))) + (\log(R(695)) + \log(R(700)) + \log(R(705))) + \alpha) \cdot 100, \tag{3.2}$$

where α was chosen by Dawson to be 0.015 in order to have only non-negative MI-values. The erythema index is similarly defined as the area under the LIR-spectrum around 510 to 610 nm, resulting in

$$EI = (-\log(R(560)) - 1.5(\log(R(543)) + \log(R(576))) - 2.0(\log(R(510)) + \log(R(610)))) \cdot 100. \tag{3.3}$$

Dawson verifies the linearity of the melanin index by testing it on some individuals with a variety of skin colors. He also uses the erythema index on the same individuals, and observing that the erythema index is correlated to the skin colour, the erythema index is corrected by adding the melanin index to the erythema index by the neccessary amount to make the erythema index constant for all the individuals, assuming they all will have the same blood content in the skin:

$$EI_c = EI \cdot (1 + \gamma \cdot MI) \tag{3.4}$$

Dawson sets the γ value to be 0.04 based on his subjects.

Kollias' slope-based methods

This project report will assemble different slope-based methods under the common name "Kollias' slope-based methods", even if Kollias is not the first author on some of the later articles concerning the topic matter.

Kollias and Baqer [19, 20] presented a method where the melanin content is quantified by fitting a straight line from 630 nm to 720 nm and taking the slope to be the melanin content. This is in theory and practice very similar to what Dawson et al. [10] proposed, except for using more wavelengths.

Stamatas and Kollias [37] will take the linear fit to be the melanin contribution to the diffuse reflectance spectrum, and remove the melanin contribution by subtraction. Then, they find estimates for the blood contribution to the spectrum by fitting the blood absorption spectra to the corrected LIR-spectrum.

Stamatas et al. [38] additionally aim to correct the slope-based melanin by removing the deoxy hemoglobin contribution obtained by the previous algorithm.

Stamatas' and Kollias' method summed up:

1. Convert the diffuse reflectance spectrum to LIR
2. Fit a straight line from 630 nm to 700 nm
3. Assume the straight line to be the melanin contribution to LIR, subtract it from the LIR to obtain spectrum without melanin
4. Assume the LIR to represent absorption and fit the oxy-hemoglobin and deoxy-hemoglobin absorption spectra to the corrected LIR to obtain an estimate of the apparent fractions
5. Multiply the deoxy hemoglobin absorption spectrum by the obtained deoxy fraction, subtract it from the previously fitted, straight line. Assume the slope of this to be an index for the melanin content.

A variant of this, not mentioned in Kollias' and Stamatas' articles, would be to continue correcting the spectras with the newer melanin line and then newer blood fractions until some kind of convergence. Or in other words, continue executing steps 4. and 5.

The oxy and deoxy spectra fits [oxy] and [deoxy] can be used as a blood volume fraction index by taking [BVF] = [oxy] + [deoxy], and oxygen saturation can be estimated using [oxy]/([oxy]+[deoxy]).

Iterative method

A different method, not found in any articles, could be to do iterations at the isosbestic points of the blood spectra. Ignoring possible methemoglobin interference, the isosbestic points at 548, 569 and 585 nm should give reflectance values only affected by the melanin content and the blood volume fraction without any concerns about the distribution of deoxy- and oxy hemoglobin. One would only have to fit the melanin absorption and blood volume fraction until the simulated reflectance fits the measured reflectance at these two points.

Other methods

There exists other attempts to quantify the melanin content in some way, but the most will either assume the apparent absorption spectrum to be representative of the actual absorption spectrum [22], not be feasible in a simple hyperspectral image scan due to requiring blood-less spectra to be obtained by applying pressure to the skin [11] or need wavelengths outside of the possible wavelength range in the instrument [44], or similar methods discarded for similar reasons.

Summary

Some methods for quantifying the melanin absorption has been presented. Except for the iterative method, the results of these are claimed to be only proportional to the melanin absorption and not equal to. The articles presenting them do not bridge the gap between their melanin quantification results and the actual melanin absorption or fraction.

One method is to iterate the melanin absorption with respect to the result of one of these methods, as is done by Norvang et al. [23]. Some variation of this is attempted. It will later be seen that the results of the methods for quantifying melanin content and blood volume fraction are correlated, and the chosen method was to iterate both the blood volume fraction and the melanin content sequentially.

3.5.2 Oxygen saturation

If there is need for forward simulations before the epidermal absorption coefficient is determined, not only the blood volume fraction and melanin content will be needed, but also the oxygen saturation.

Randeberg et al. [27] proposed a formula for calculating the oxygen saturation directly from the reflectance spectrum. It is described to be a very rough approximation, but sufficient for relative measurements. The question is whether this formula can be used to find the estimate of the blood oxygen saturation which is needed. The formula is based on a different formula given in the paper by Spott et al. found in [36], which again is based on a formula given in Ishimaru [17], chapter 3-5-2. Randeberg's formula is given as

$$oxy = \frac{\mu_{Hb}(\lambda_1) - \mu_{Hb}(\lambda_2) \cdot \frac{R(\lambda_2)}{R(\lambda_1)}}{\mu_{Hb}(\lambda_1) - \mu_{HbO2}(\lambda_1)}, \tag{3.5}$$

where R is the measured diffuse reflectance.

3.5.3 MCA

Given a measured absorption spectrum for a mixture of materials, and knowing the absorption spectra for each different material, the concentration of each material in the mixture can be derived using multicomponent analysis (MCA).

The total absorption for each wavelength can be assumed to be a linear combination of each absorption coefficient:

$$\mu_a(\lambda) = \sum_i \mu_{ai} \cdot c_i, \tag{3.6}$$

where c_i is the unknown concentration of component i and μ_{ai} is the known absorption coefficient for the same component. Choosing a set of wavelengths, this will form a set of linear equations. It can be assumed that the total $\mu_a(\lambda)$ will contain some noise, which will make us unable to solve this exactly. Linear least squares can be used to find the best fit of each absorption coefficient [18]. Linear least squares will however lay no restraints on the non-negativity of the coefficients.

MCA will not be used in this project report, but some mentions are made of fitting of absorption spectra against the wavelength-dependent dermal absorption coefficient and is therefore mentioned as a possible method for doing so. It is unlikely that MCA will be used later on, as better spectral unmixing algorithms designed for hyperspectral images can be employed.

Chapter 4

Results and discussion

Real time inverse modelling of hyperspectral images is a sorely needed tool if hyperspectral imaging ever should become a feasible diagnostic tool in medicine. Different methods have been evaluated. First, some spectra used for benchmarking the inverse method will be presented. The forward models are then evaluated, followed by an evaluation of the inverse modelling times for the Monte Carlo-based inversion program and the hyperspectral, diffusion-model inversion tool, and lastly an evaluation of the first step in the inverse chain.

4.1 Spectra

The inverse models were benchmarked against a select few spectras, since it would be easier to test the inversion strategy if all possible errors that might rise due to using hyperspectral data were eliminated. These are displayed in figures 4.1, 4.2, 4.3. These were collected by my supervisor from her own skin and one of her PhD-students using a spectroscope. Figures 4.1 and 4.2 are spectras with a low epidermal melanin absorption, probably in the range 190 to 250 m^{-1}. Figure 4.2 has a low oxygen saturation, evident from the lack of features in the 550 nm-region.

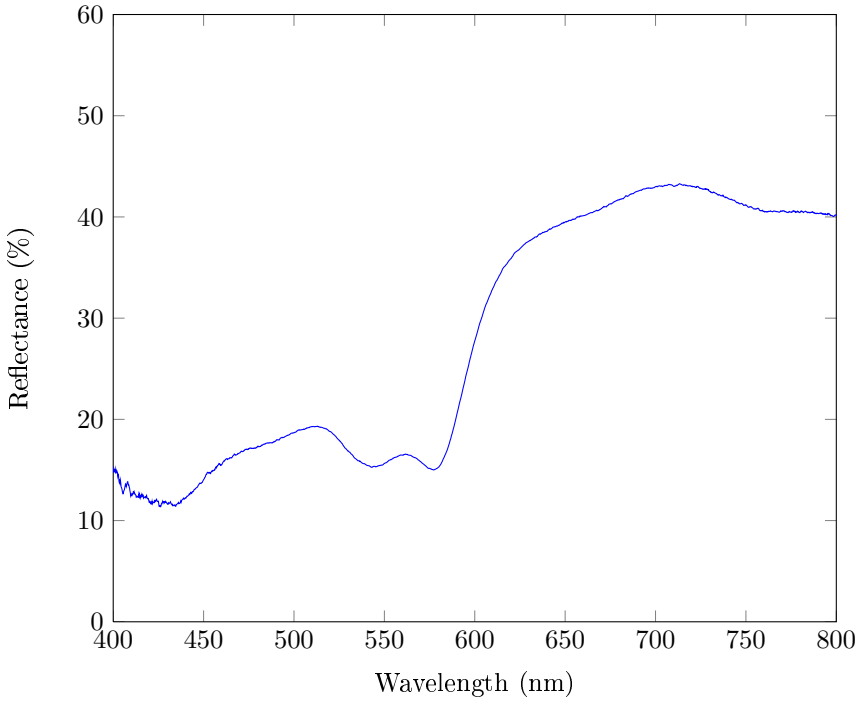

Figure 4.1: Measured diffuse reflectance spectrum from a light-skinned individual.

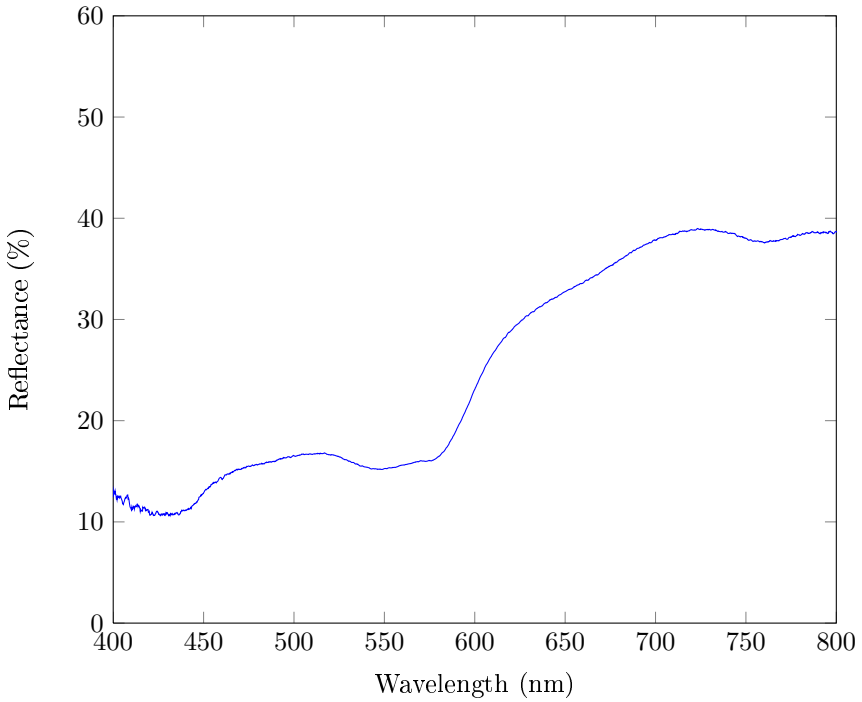

Figure 4.2: Measured diffuse reflectance spectrum from a light-skinned individual. Low oxygen saturation is evident.

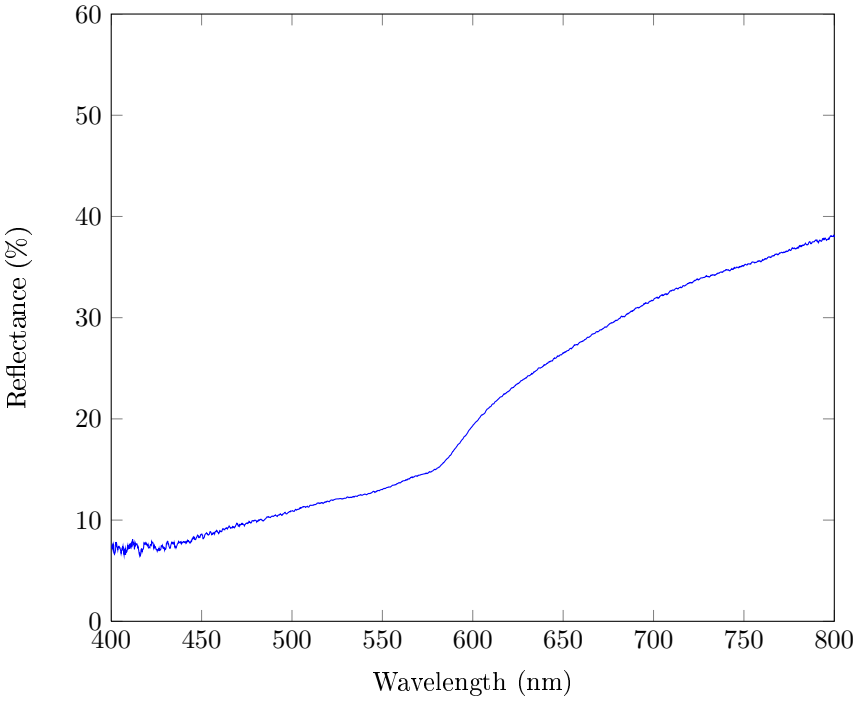

Figure 4.3: Measured diffuse reflectance spectrum from a dark-skinned individual.

4.2 Forward models

The GPU-MCML forward model was tested against the diffusion model forward model for different parameters and different source functions in the diffusion model. Results are plotted in figures 4.5, 4.6, 4.8 and 4.7. The results show that overall, the correspondence is good, but will be worse when the absorption is higher, as expected. The results are also similar to the results found by Randeberg et al. [28], namely that the error between the isotropic diffusion model and Monte Carlo will be shaped like the blood spectrum and the error can be expected to be scaled away by scaling the blood absorption spectrum by the neccessary amount.

Results from GPU-MCML is also plotted alongside with results from the more traditional MCML [45] for the same input parameters in figure 4.4. Results are the same.

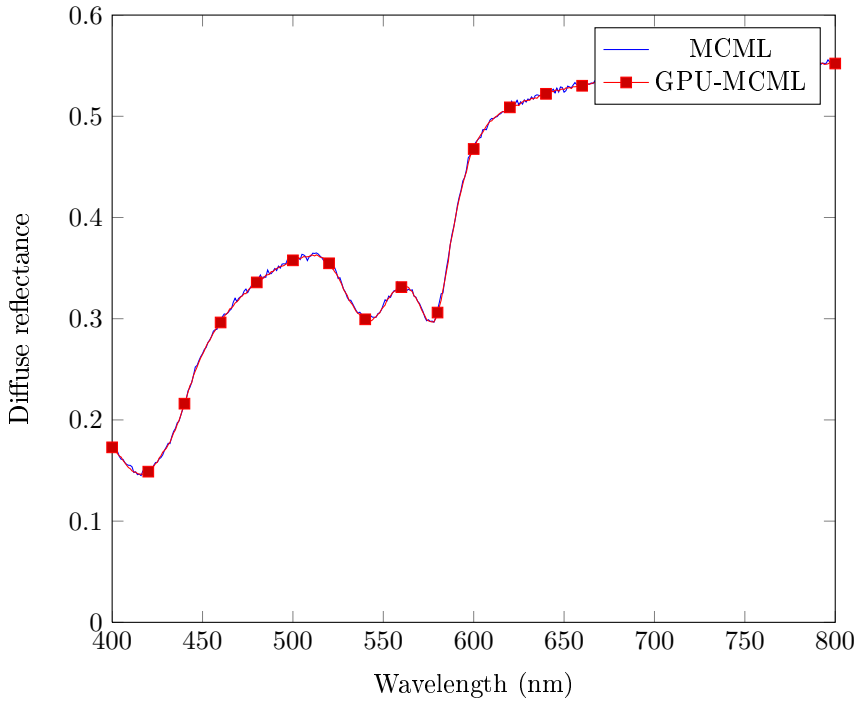

Figure 4.4: Comparison of MCML and GPU-MCML for oxy = 0.8, BVF = 0.05, $\mu_{a,m,694} = 250$.

Figures 4.5 and 4.6 both have a low absorption, and especially Delta-Eddington has a good corresponence.

On the other hand, figure 4.8, with its high blood concentration and figure 4.7, with its high melanin absorption, display a worse correspondence. It can be noted that Delta-Eddington is a worse fit than the isotropic variant above 600 nm, where the scattering is larger than the absorption and the diffusion model overall should be a better fit. Delta-Eddington "breaks down".

The scattering function in the simulations has one small error: the Rayleigh part of the scattering is assumed to have the same anisotropy factor g as the Mie part. The Rayleigh scattering is in reality isotropic, and should have an anisotropy factor equal to 0. If this is implemented for the diffusion model and MCML, the result shown in figure 4.9 will happen.

The result is worse and has a larger discrepancy between GPU-MCML and the diffusion model than the simulations done for the same input parameters in fig. 4.6. Even the error for the Delta-Eddington source function is now blood absorption spectrum shaped.

The reason for this is likely the way the diffusion model and GPU-MCML treats the g-factor. The diffusion model handles the reduced and normal scattering coefficient separatedly and will be able to

differentiate between the Rayleigh scattering and the correctly reduced Rayleigh scattering, i.e. the exact same coefficient. MCML, on the other hand, uses the non-reduced μ_s, with the Rayleigh-part set to the correct value, but uses the same g, which will not differentiate between the Rayleigh and non-Rayleigh parts of the compund expression. The end result will be an MCML which treats the Rayleigh-part as non-isotropic and a diffusion model handling it isotropically, explaining the large discrepancy.

The Rayleigh scattering has therefore been handled in the same way in both GPU-MCML and the diffusion model by pretending it has the same g as the rest of the scattering mechanisms. Strictly speaking, it will not be correct, but MCML and the diffusion model will at least agree on a common ground. On the other hand, because of this, MCML will be slightly less correct than the diffusion model since the diffusion model has the potential of handling the g correctly. Mixing gs from different scattering mechanisms purely by altering g seems largely undefined. Mixing it by using the volume fractions will not be correct. The summed reduced scattering coefficients using different gs will not agree with the value obtained by multiplying the non-reduced, combined scattering coefficient by a g combined in this way, and combining the gs in a way which does not violate this has no guarantee of actually being the correct g, even if the reduced scattering coefficients will be correct.

Conclusions are, with some reservations, that the diffusion model largely should be good enough for implementing an inverse model.

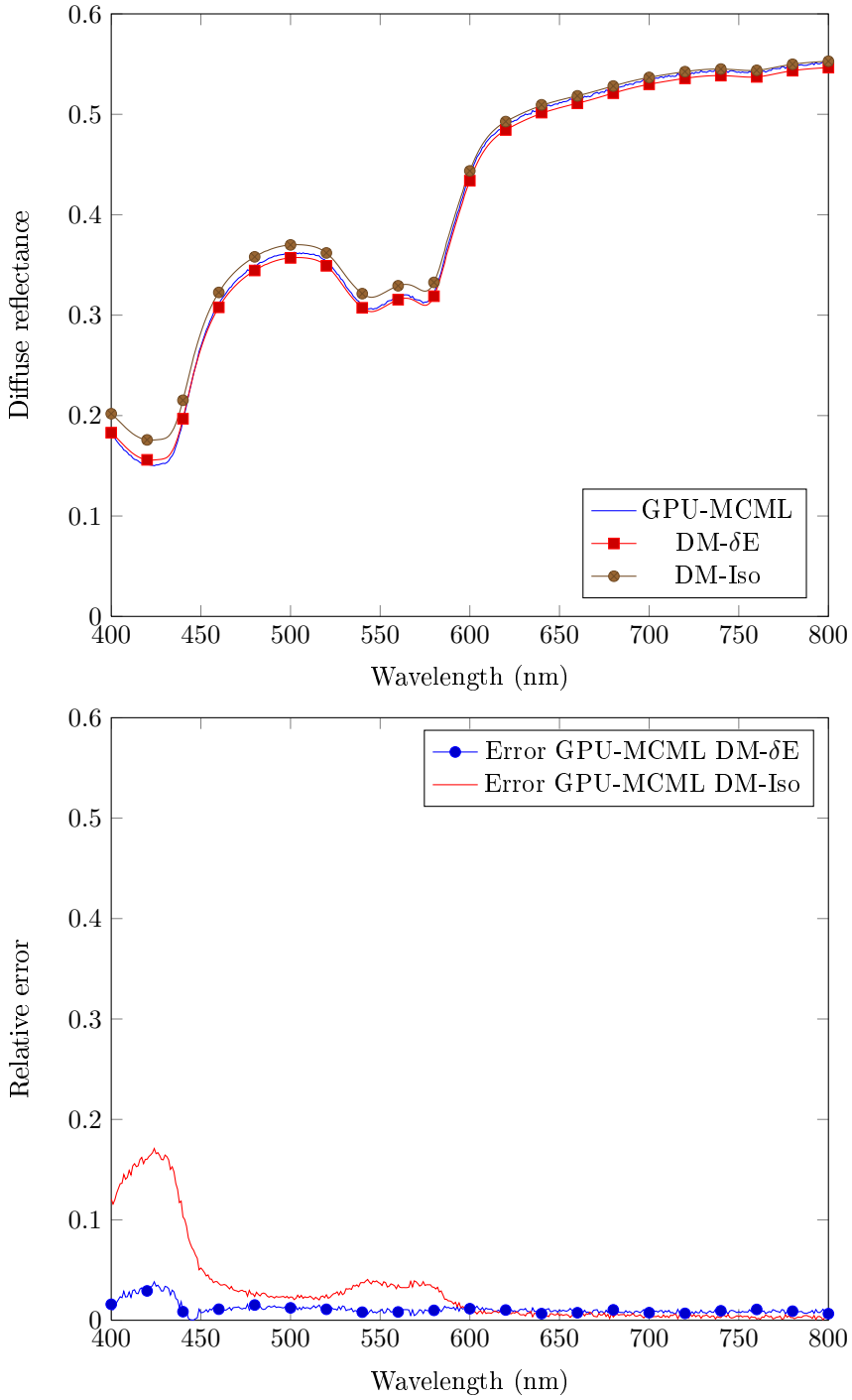

Figure 4.5: Comparison of GPU-MCML and diffusion model for oxy = 0.4, BVF = 0.01, $\mu_{a,m,694} = 250$ m^{-1}.

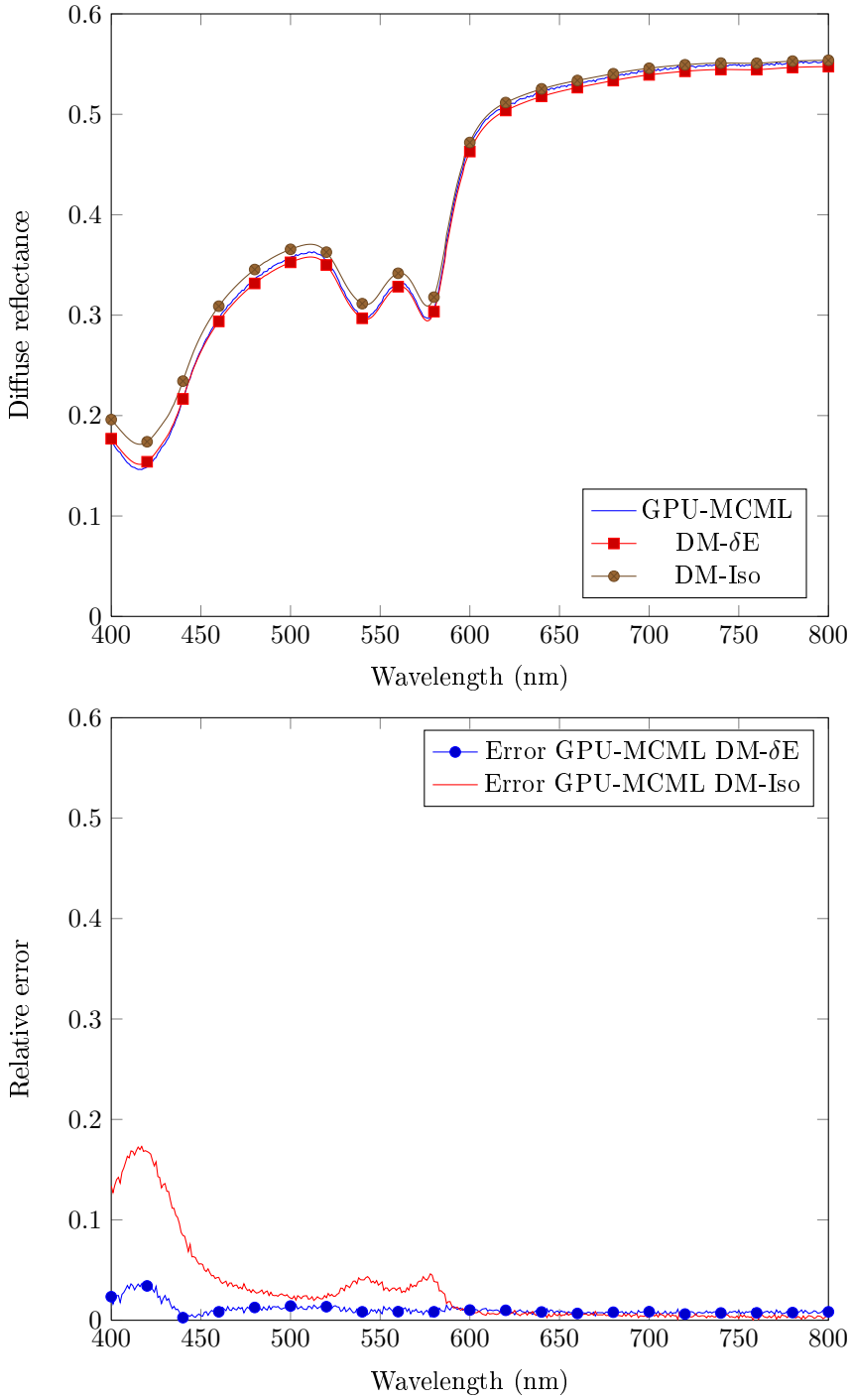

Figure 4.6: Comparison of GPU-MCML and diffusion model for oxy = 0.8, BVF = 0.01, $\mu_{a,m,694} = 250$ m^{-1}.

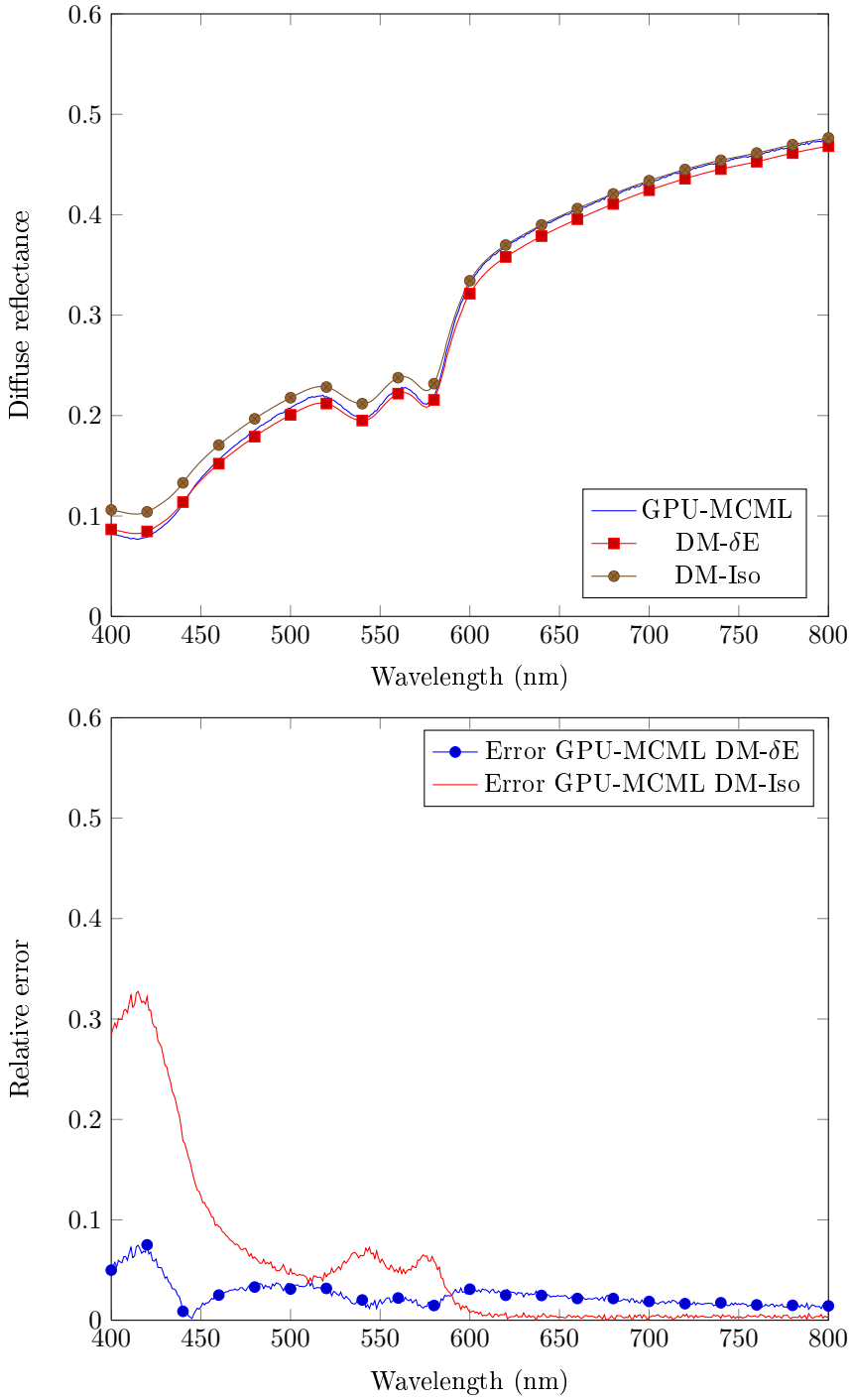

Figure 4.7: Comparison of GPU-MCML and diffusion model for oxy = 0.8, BVF = 0.01, $\mu_{a,m,694} = 700$ m^{-1}.

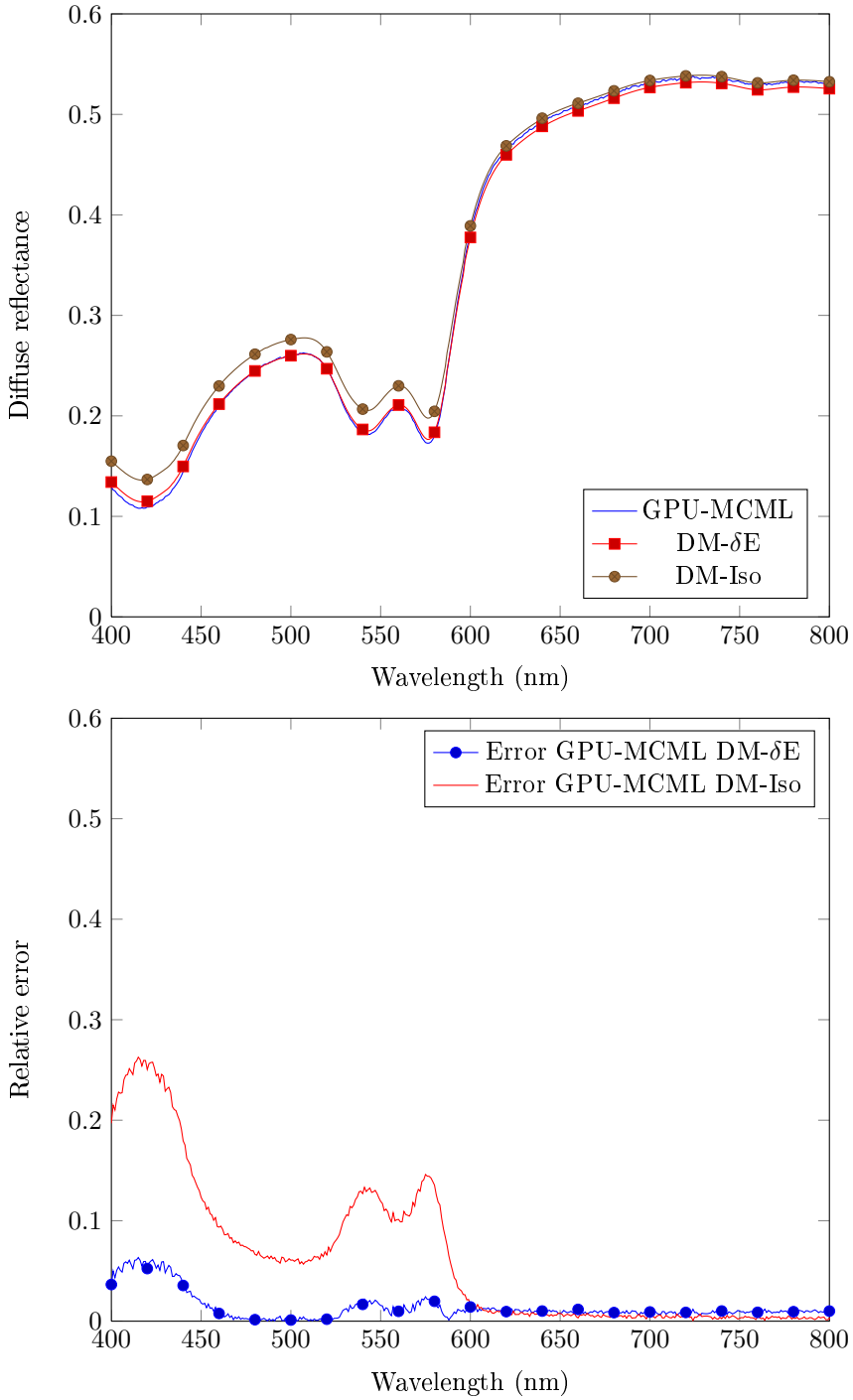

Figure 4.8: Comparison of GPU-MCML and diffusion model for oxy = 0.8, BVF = 0.05, $\mu_{a,m,694} = 250$ m^{-1}.

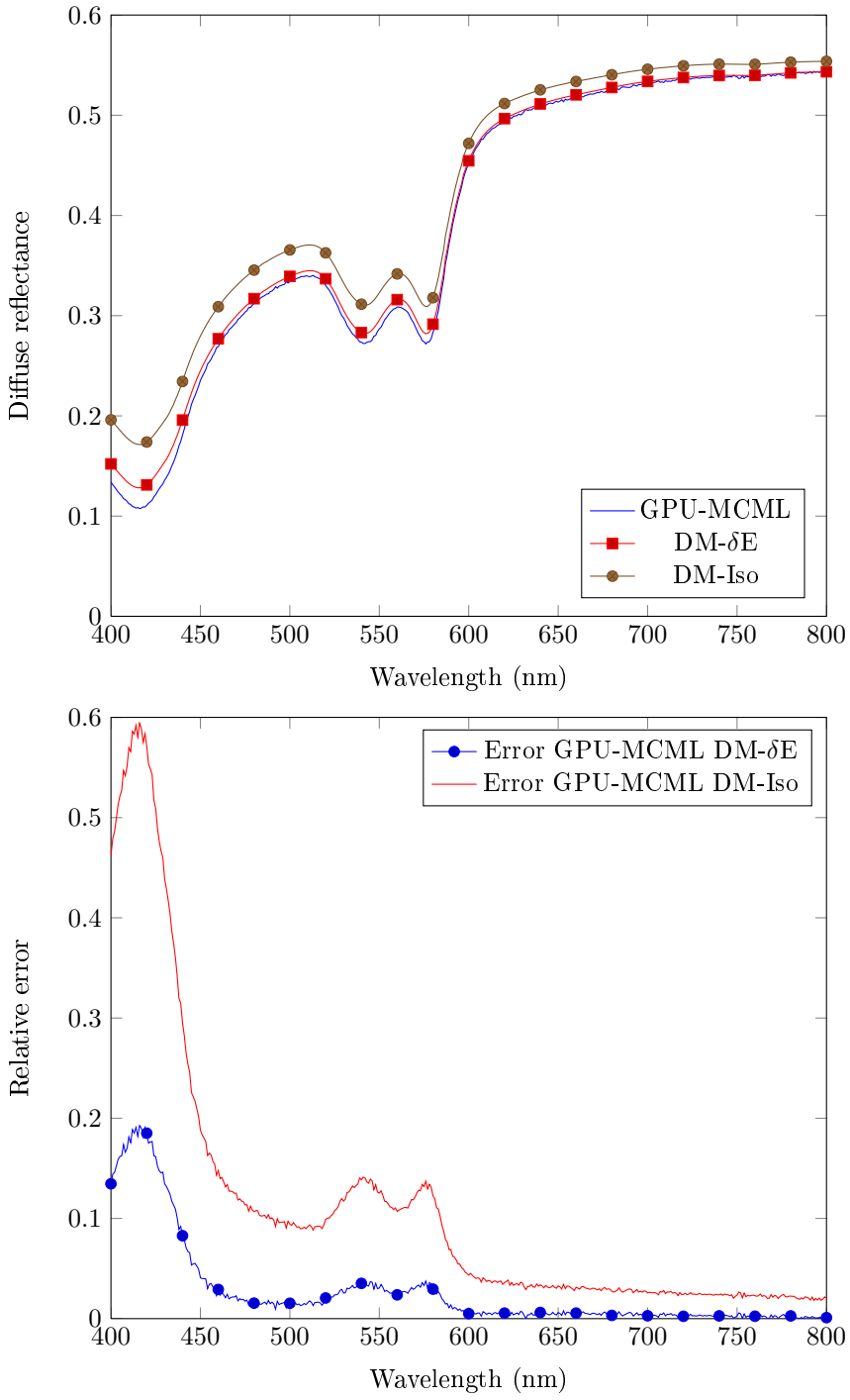

Figure 4.9: Comparison of GPU-MCML and diffusion model for oxy = 0.8, BVF = 0.01, $\mu_{a,m,694} = 250$ m^{-1}, with g_{ray} set to 0.

4.3 Benchmarking of the inverse models

Both a forward and an inverse model has been implemented for GPU-MCML, though lacking a concrete method for determining $\mu_{a,m,694}$, as will be discussed later. The forward model was benchmarked against the diffusion model above. Mainly the computation times will be benchmarked here and not the end results.

The GPU-MCML inverse model will need everything from 3 to 8 minutes for 160 wavelenghts in the range of 400 to 800 nm, depending on the number of photons used for each simulation. As the time taken to inverse model a single spectrum, this method is not feasible for a larger hyperspectral image, even if it allegedly gives more correct results as a more accurate solution to Boltzmann's transport equation.

The forward model will be slightly faster, everything from 20 to 40 seconds. This will be more useful for benchmarking the diffusion model. Results output from the diffusion model may be tested against GPU-MCML faster than what MCML would ever achieve. With a newer computer, MCML was able to finish forward-modelling the whole spectrum only after 20 minutes. Can GPU-MCML be optimized? A plot of the running times versus wavelengths for a typical situation is shown in figure 4.10. The running

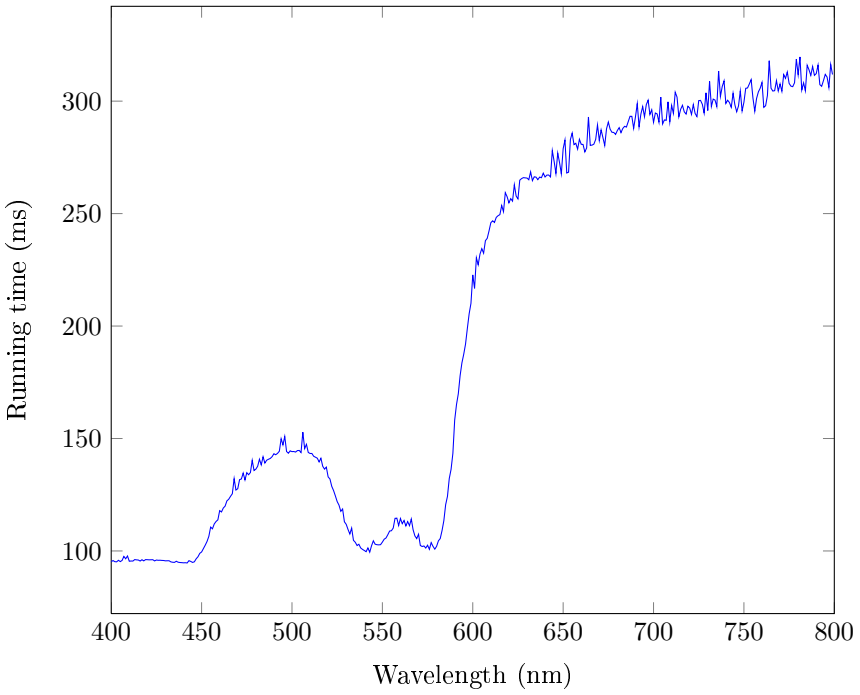

Figure 4.10: The running time for GPU-MCML as a function of wavelength, oxy = 0.8, BVF = 0.01, $\mu_{a,m,694} = 250$.

times will be dependent on the absorption-scattering ratio due to how GPU-MCML is implemented. This will pose some problems for an inverse modelling situation since each forward simulation will be dependent on how high we set our absorption, reducing the predicability of the running time. In any case, it can be seen in figure 4.10 that the running times mostly follow the shape of the reflectance spectrum, except for some areas where it constantly lies at 100 ms. This should be evidence of a bottleneck for each simulation which might be eliminated. When adapting the GPU-MCML code for spectras, the internal code was largely left untouched and mainly the input functions were hacked to accept internal data structures instead of input files. As will be demonstrated with GPU-DM, memory allocation is a point in GPU-based code which will use a large amount of time, and profiling of GPU-MCML using NVIDIA Visual Profiler will show that the memory allocation step will use about 50 ms of the total running time. The same array sizes are likely to be allocated every time a simulation is run, and the total running time may be reduced by allocating memory only once.

Yet, as all GPU parallellization capabilities are focused on simulating the results for one wavelength only, and GPUs have limitied capabilities for running more than one stream at a time, all wavelengths have no other choice but to be simulated sequentially. If an instantaneous Monte Carlo simulation of the full spectrum is desired, the single simulation will have to be optimized dramatically, which at least is not possible on the lower-end GPU which was used for testing. Reducing the running time by 50 ms for each wavelength will not help in the long run. It might be possible to reduce the running time enough on future GPU architectures, but there will be years and years before GPU-MCML is fast enough for inversion of 1600 pixels sequentially without employing 1600 different GPUs. Additionally, when that time comes, the hyperspectral camera specifications are likely to have changed and will give the results far faster than one line at a time. The real time requirements will change.

Today, GPU-MCML simply is not feasible for a real time inversion system for hyperspectral imaging, and is likely to never be. The small accuracy pay-off will not satisfy the price of long simulation times.

The diffusion model, on the other hand, has proven to be feasible as a real time inverse model for hyperspectral cameras even on a lower-end GPU. Table 4.1 shows the running times of each step in the process. The GPU allocation step is the most time-consuming step, and will only be done once at the

Table 4.1: Running times of GPU-DM

Function	Running time (ms)
gpuallocate()	117.0
Memory transactions	2.220
calibrate()	0.073
calcMelaninBvfOxy()	????
calcSkinData()	0.308
ReflIsoL2()	0.410
nextMuad()	0.120

beginning of the program initialization. The memory transactions, the calibration and the calculations of the skin data will only be done once for each line scan, a total of 2.1 ms at the beginning of the runtime and 0.5 ms at the end. The memory transactions may also, on a better GPU, be parallellized with the kernel functions. It is as of yet unknown how much time the melanin step will take. It depends on how efficiently the memory access may be implemented, and how much inverse simulating that has to be done. Anything less than 2, maybe 4 ms, can at best be estimated, and 2 ms extra time is gained if the memory transactions become concurrent. Low multiprocessor occupancy is sported by **ReflIsoL2()**, which, if somehow heightened, might reduce its running time even more. This is mainly due to a low number of threads per block, which could not be increased because **ReflIsoL2()** needs a large amount of registers per thread. The number of registers may be lowered, but the same calculations will then have to be done multiple times. Latency hidden away by a higher multiprocessor occupation will be re-introduced as added calculation latency.

The remaining time until deadline will be between 25 and 22 ms. Assuming some extra overhead between each line, the results should be prepared within 20 ms. With 0.53 ms per iteration, 37 iterations of Newton's method can be performed within the deadline.

Is this enough? The convergence status after 15 iterations is seen in figure 4.11. With the worst start estimate for $\mu_{a,d}$, it will converge within these iterations. Having $\mu_{a,d}$ start at 1.0 for all wavelengths is very artificial, a far better approximation always will be the start estimate. The method should therefore be expected to converge much faster than within 15 iterations. While the outlook is good, there is one problem. If there is something wrong with the spectrum, like noise, the spectrum might in places lie so low that the method needs to set a negative dermal absorption since the epidermal absorption is high enough that the dermal absorption will have to compensate. In that case, the forward model might fail, the derivative will fail and the method determining the next $\mu_{a,d}$ will fail. A check for negativity has been built into the Newton step, which will set the dermal absorption to 1.0. This is no better than just

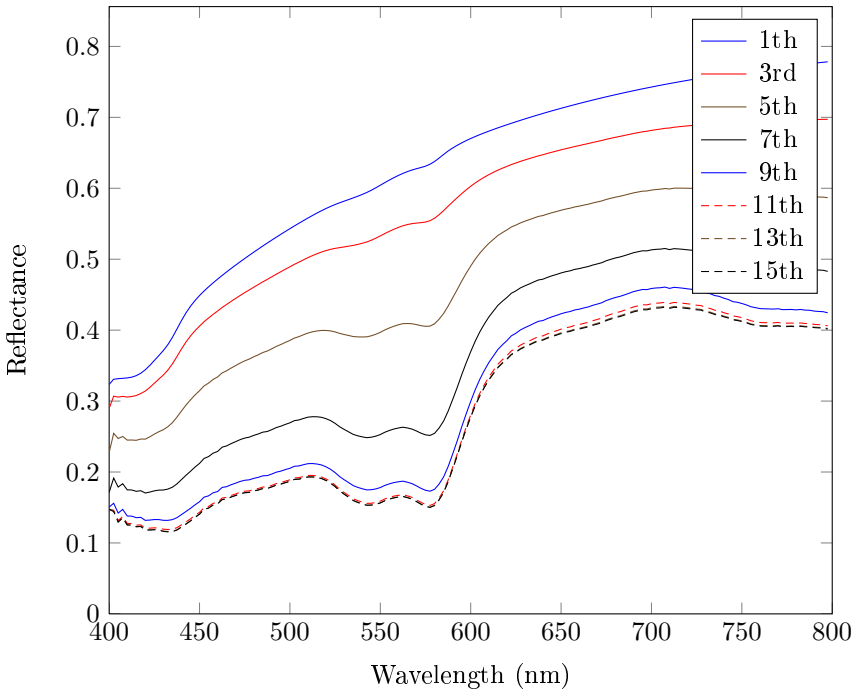

Figure 4.11: The convergence of GPU-DM using $\mu_{a,m,694} = 250$, oxy $= 0.6$, bvf $= 0.05$ and a very bad start estimate for $\mu_{a,d}$, $\mu_{a,d} = 1.0$.

outputting NaN-values, but at least the values are real. Noise removal will have to remove such noise before inverse modelling is attempted.

The GPU-DM inverse modelling of the spectrum in figure 4.1 is compared against the GPU-MCML-result in figure 4.12. They are not the same, as can be expected since the source function in use was the isotropic one and not Delta-Eddington, but the result is not that bad. Both GPU-MCML and GPU-DM agree on a common shape of the dermal absorption spectrum. When spectral unmixing some time in the future is implemented, it is likely to output the same results since the general shapes are the same.

There are however no guarantees that the running time of each step will always stay the same. No hard real time functionality is implemented in the CUDA framework, and it is unknown how the GPU will schedule different processes competing for GPU time. If some framework for instance tries to process data on the GPU from two different threads, the GPU might decide to follow some first come, first served-principle, and even if the calculation of reflectance values still is fast, the time between two reflectance value calculations will be longer since it has to wait for some other calculation to finish first. If the GPU is used for display management, this can cause additional interference. The framework handling the hyperspectral data will have to implement some mutual exclusion for the GPU resource, and the functionality supposed to handle the data in real time will have to be given higher preference.

In any case, it is not crucial that the hyperspectral processing meets the hard deadline of 30 ms, and interference might be acceptable.

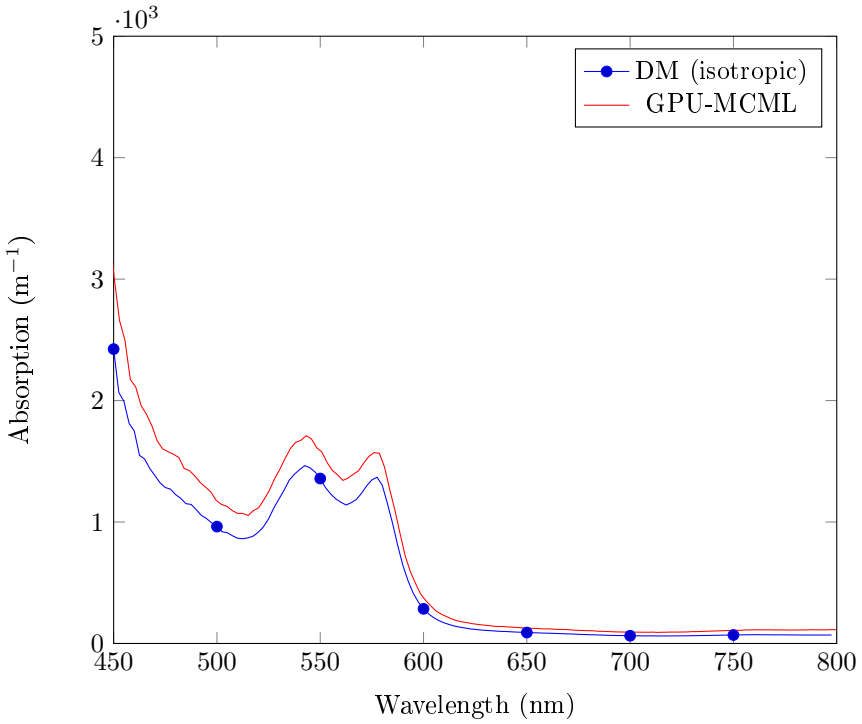

Figure 4.12: Derived $\mu_{a,d}$ from the spectrum in figure 4.1 by using GPU-MCML and the diffusion model. The same initial estimate, $\mu_m = 250$ m^{-1}, $o = 0.6$, bvf $= 0.01$.

4.4 Melanin and blood volume fraction

The part of the inverse chain needing the most scrutiny was the epidermal absorption-determining stage. The methods presented for estimating the melanin content and blood volume fraction have earlier been used, both as a part of an inverse chain and especially for quantifying relative differences. There are some known cross-talk problems in these methods, less apparent when used for quantifying relative differences. When used as a part of an inverse model requiring absolute measures of each parameter, these problems caused the end result to be wrong, and the melanin quantifying methods were therefore investigated and tested in detail. The investigation of Dawson's indices, Kollias' slope-based method will be presented along with a presentation of the problem.

4.4.1 Dawson's indices

Using forward-simulations, different blood volume fractions and melanin absorptions were input into the model and Dawson's erythema and melanin indices were calculated for each diffuse reflectance result. Dawson's erythema index as a function of the blood volume fraction for different oxygen saturations and melanin absorptions is shown in figure 4.13, and Dawson's melanin index as a function of the melanin content for varying oxygen saturations and blood volume fractions is shown in figure 4.14.

The erythema index is not linear with the blood volume fraction. In addition, the maximum and minimum possible erythema indices for each blood volume fraction are not equal. One erythema index will answer to various blood volume fractions, not ensuring a one-to-one relation.

The melanin index is, on the other hand, more or less linear, but also here is there some difference between the maximum and minimum possible melanin index for one specific input melanin absorption.

There reason for this is explained and investigated using actual human tissue by Stamatas and Kollias [37]. The melanin content is estimated from a wavelength interval in which the deoxy hemoglobin absorption is not equal to zero, namely around 620 to 700 nm. The deoxy hemoglobin absorpion curve exhibits

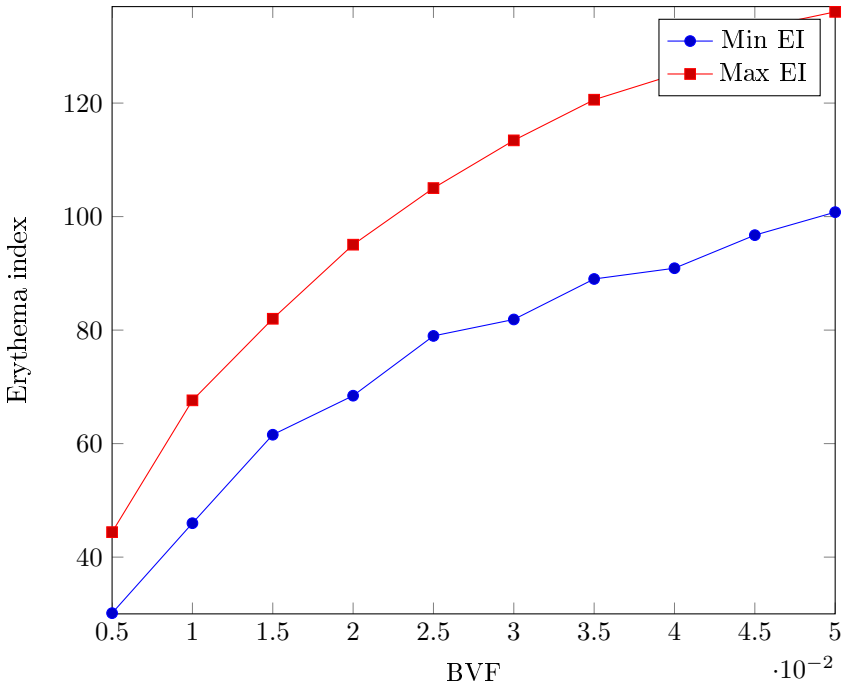

Figure 4.13: Dawson's corrected erythema index as a function of the blood volume fraction, simulated by varying $\mu_{a,m,694}$ between 100 and 1000 m^{-1} and the oxygen saturation between 0.5 and 0.8, and picking the highest and lowest resulting erythema index. Simulated using GPU-MCML.

traits very similar to the melanin absorption curve, they both slope downwards, as shown in figure 4.15. Such methods will mis-interpret the presence of the deoxy hemoglobin absorption as the presence of strong melanin absorption when the melanin content is sufficiently low. Stamatas and Kollias showed that color space methods similar to the melanin index used to quantify melanin content would claim a higher melanin content when increasing the deoxy hemoglobin content by trapping blood by applying pressure.

4.4.2 Kollias' indices

Where Dawson tried to correct his erythema index for the melanin content, but not the other way around, Stamatas et al. [38] presented a modification of Kollias' method where also the melanin content is corrected for the presence of blood.

Figure 4.17 displays the highest possible and lowest possible corrected melanin slope corresponding to a specific input melanin content. The relationship between the melanin content and the melanin slope is linear, as it was with Dawson's method. The difference between the maximum and minimum quantification value is also similar. The same melanin slope will correspond to many different melanin contents, as Dawson's index did.

In detail will figure 4.16 show how the corrected melanin slope varies with the oxygen saturation. The behavior in fig. 4.17 showed how a higher melanin content corresponded to a more negative slope. Fig. 4.16 shows that a lower oxygen saturation, that is, a higher deoxy hemoglobin content, will vary the melanin slope in the same way, resulting in cross-talk.

An iterative method where the blood coefficients and melanin slope are continually corrected against each other was also presented, and the result of this is shown in figure 4.18. The results do not deviate much from the results gained from doing only one correction, in fact, the deviations are more or less random.

Kollias' deoxy hemoglobin fit is plotted for different input melanin contents in figure 4.19. The melanin

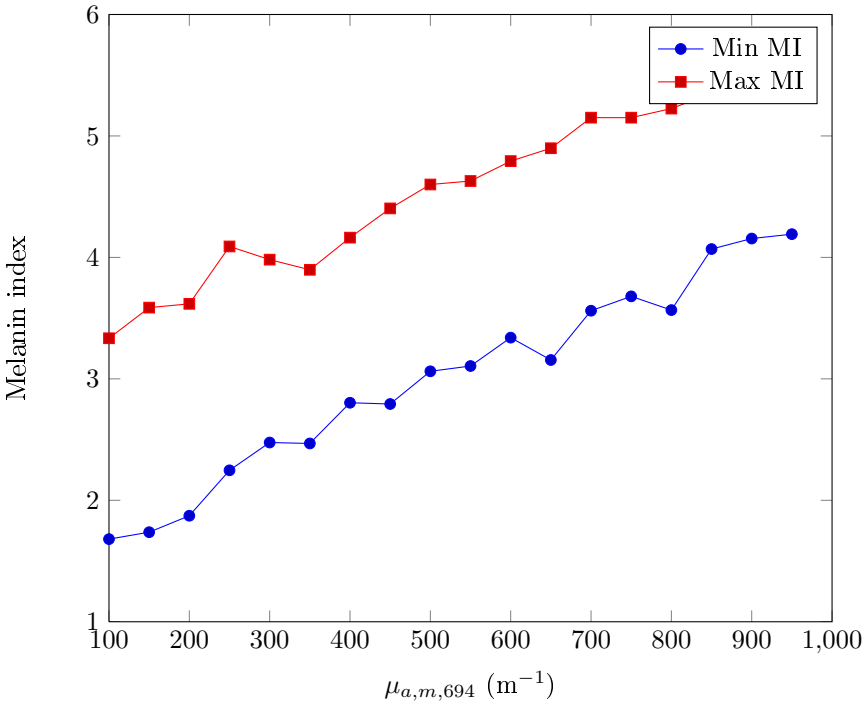

Figure 4.14: Dawson's melanin index as a function of the melanin absorption, simulated by varying the BVF between 0.0005 and 0.005 and the oxygen saturation between 0.5 and 0.8, and picking the highest and lowest resulting melanin index. Simulated using GPU-MCML.

slope was correlated with the deoxy hemoglobin amount. Similar behavior for the Kollias' deoxy hemoglobin fit is not as evident. Where the lines should be straight, they do however slope with the melanin content.

The fitted oxy and deoxy hemoglobin parameters are summed, and the possible minimum and maximum of these are plotted as a function of blood volume fraction in figure 4.20.

The behavior is very similar to what was shown for Dawson's erythema index in figure 4.13. The relationship between what should quantify the blood volume fraction and the input blood volume fraction is not linear, but sports something more similar to a logarithmic function. In addition, the same is seen here as for the rest of the indices, there is a difference between the largest and smallest possible index.

The general behavior for Stamatas' and Kollias' methods is very similar to Dawsons indices when tested using the same simulation methods, they both display similar crosstalk. Still, compared to Dawson's indices, the errors are less. The range of melanin absorption parameters that correspond to the same melanin index is less in Kollias' case than in Dawson's case, a range of 500 m^{-1} compared to a range of 700 m^{-1}.

Using Stamatas' and Kollias' method as a part of the inverse chain was investigated. This was done by iterating the melanin content and blood volume fraction seqentially as according to Stamatas' and Kollias' corrected melanin slope and blood parameter fits. An initial attempt, as shown in figure 4.21, will show that the method is able to converge towards the right order of magnitude of $\mu_{a,m,694}$ for a simulated spectrum. It is also able to converge towards the right orders of magnitude when applied on the spectrum in figure 4.1, as shown in figure 4.22. However, the method gives a too-high estimate of the melanin content when the presence of deoxy hemoglobin becomes too strong, as shown in figure 4.23 when applying the method on the spectrum in figure 4.2. The spectrum is from skin that has a low oxygen saturation and will be a perfect example of the problems presented by Stamatas and Kollias [37]. Even if the blood volume fraction and the melanin absorption coefficient at 694 nm are iterated simultaneously, they will not converge towards the correct values. The spectrum in figure 4.2 is from a very light-skinned individual, around 200 m^{-1}, and the result from the iteration, around 1000 m^{-1},

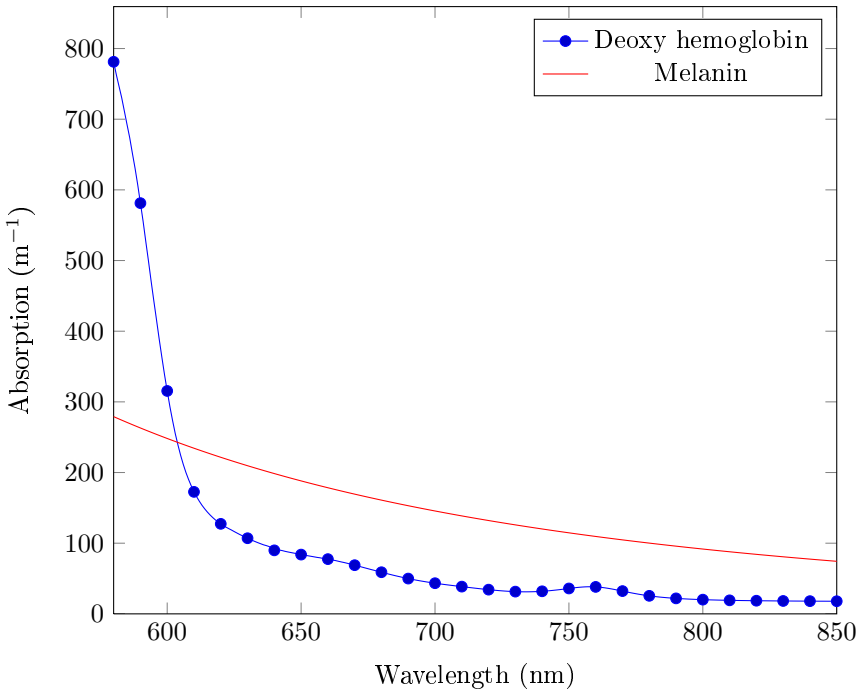

Figure 4.15: The absorption coefficients of deoxy hemoglobin and melanin. The deoxy absorption coefficient is multiplied by 0.05, while $\mu_{a,m,694} = 150$ m^{-1}. The melanin absorption coefficient is plotted using (2.22), while the blood absorption data is measured by Thorsten Spott.

corresponds to dark skin, which is not correct.

The problem can in detail be illustrated by plotting the measured spectrum together with the spectrum obtained by estimating $\mu_{a,m,694}$ and the BVF, as shown in figure 4.24. The simulation matches the measurement along the slope in the correct area, but differs too much from the measured spectrum elsewhere. The fact that the simulated spectrum lies below the measured spectrum in several places will also pose a problem, since later $\mu_{a,d}$ fitting will have too lower the dermal absorption coefficient. It might not be possible to do so while keeping the dermal absorption coefficient non-negative.

This initial attempt at using Kollias' and Stamatas' methods shows that the cross-talk problem is too strong for a simple iteration strategy. Response in one variable will cause response in a different part of the spectrum where the other variable is estimated, and the end result is a complex problem not easily solved, especially not by a simple iteration strategy. On the other hand, the measures of melanin and blood volume fraction are designed to be decorrelated, and implementing a convoluted scheme for iteration to correct for this behavior will be very ad-hoc. Something should instead be done to remedy the behavior itself.

Stamatas and Kollias seemed to have success with their method, although by quantifying relative differences. The methods were in this report mainly benchmarked using simulations. The discrepancies in this report could be due to simulation errors in these. The skin model is a very simple two-layer model which only approximates real skin. The scattering functions are approximations, the input absorption spectra are not measured in vivo. While Monte Carlo widely is regarded to be true and accurate, there are still some assumptions made that might be faulty.

However, as real skin is more complex, there are far more complex and convoluted processes behind the diffusively reflected light than what the simulations can achieve. If the methods have problems in simple situations, there is no reason they should fare any better in a more complex one.

The main problem in trying to correct the blood quantification with the melanin estimate and vice versa is how both methods rely on some extended Beer's law scheme in order to make the LIR-spectrum a valid approximation to the absorption. When light penetrates the tissue and is diffusively reflected back after

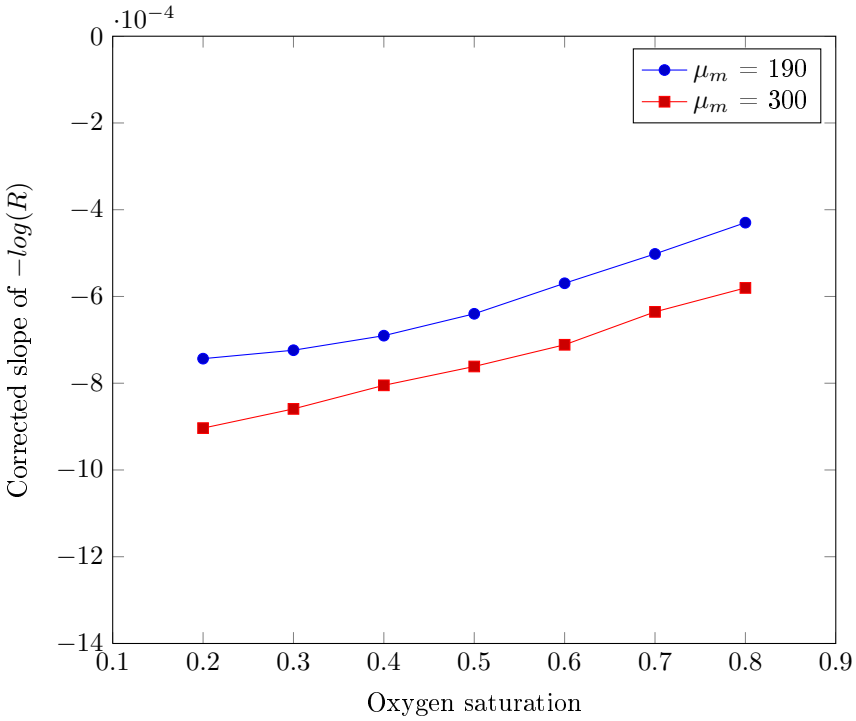

Figure 4.16: Kollias' corrected melanin slope for constant melanin content and different oxygen saturations. Simulated using GPU-MCML and BVF = 0.05.

undergoing absorption, it will in part display mechanisms similar to Beer's law, but the amount of light coming back is also dependent on scattering. The light will penetrate different depths in non-homogenous tissue, bringing back different information depending on the wavelength. Comparing different LIR-values at different wavelengths, like correcting the spectrum around the 550 nm area for an melanin absorption found in the 600-700 nm area, will be difficult.

An example can be seen in figure 4.25, where the unfitness can be observed by trying to fit the blood absorption coefficients to the melanin-corrected spectrum. The estimate could have become better if we alongside the blood spectras also tried to fit a constant factor as shown in figure 4.26, but it will not. The scattering variation is too strong.

The main problem with the cross-talk itself is how the methods rely on the spectrum from 600 to 700 nm. The thought has all along been that the blood absorption spectrum is low enough that the spectrum mainly would be affected by melanin, and it can clearly be seen in figure 4.3 compared to figure 4.1 that higher melanin content will cause the whole diffuse reflectance spectrum to slope more downwards. As demonstrated, however, melanin is not alone. When the general absorption is low enough that the scattering is dominating, the back-reflected light will come from greater depths and be affected by far more than just the shape of the melanin curve in epidermis.

This report will reach no specific conclusions other than the fact that the methods, as of today, are not sufficient. They are sufficient for relative measurements, but not for any absolute measurements of the melanin content or blood volume fraction. Work will still have to be done in order to find a method which should fare better for absolute measurements. A part of the problem is the large penetration depth, and estimating the melanin content at more shallow penetration depths can be one possible solution, as is done by Verkruysse et al. [44], perhaps in combination with the other methods.

Figure 4.27 shows that it will be important to find a near exact method. If the initial $\mu_{a,m,694}$ is varied in steps of 50 m^{-1} within the allowable range for a light-skinned individual, the derived end result for μ_d will vary too much. For some of these derived dermal absorption coefficients, the dermal chromophore absorption spectra may be fitted only when one at the same time fits the melanin curve to the result,

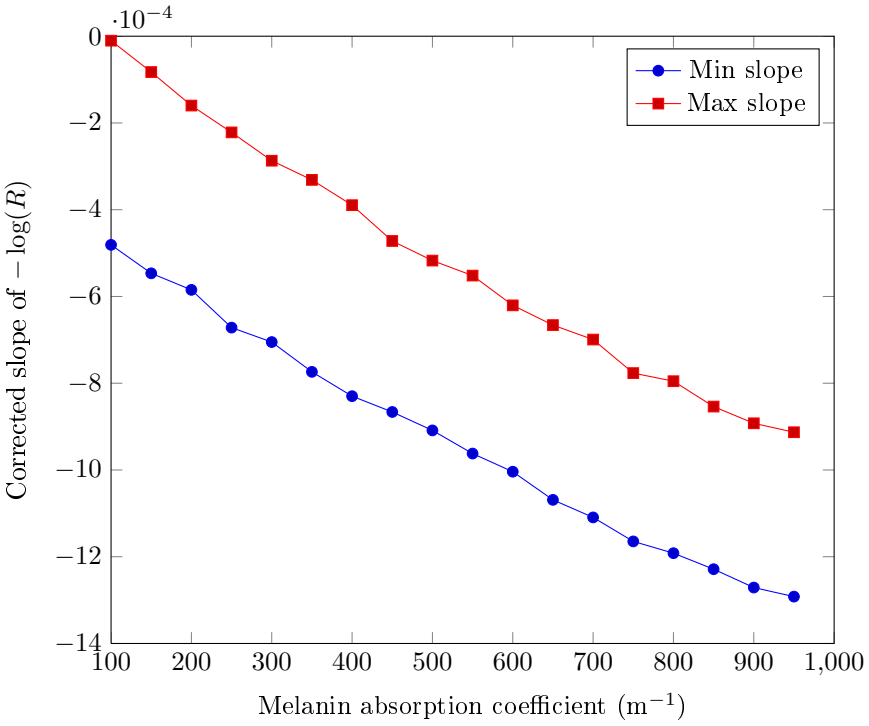

Figure 4.17: Kollias' melanin slopes as a function of the melanin absorption coefficient, the maximum and minimum slope taken from a set of oxygen saturations (0.5 and 0.8) and blood volume fractions (0.0005 and 0.005). Simulated using GPU-MCML.

which just shows that the melanin has not completely been separated from the rest and the model fit will try to compensate for the lacking melanin absorption by increasing the dermal absorption coefficient.

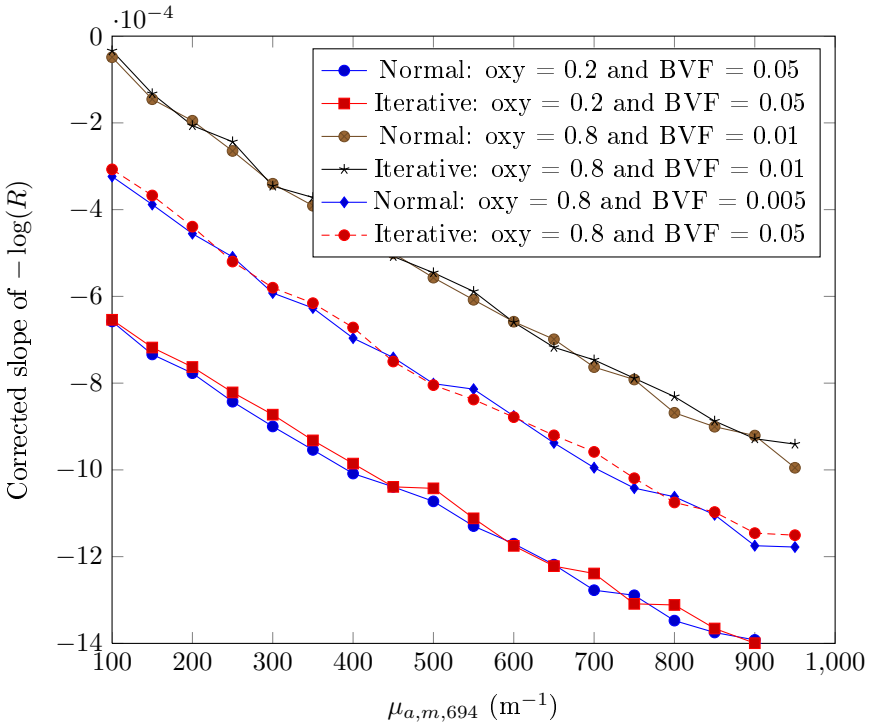

Figure 4.18: Kollias' corrected melanin slope as a function of melanin absorption for a set of different skin parameters, using Stamatas' proposed method and an iteration-based variant. Simulated using GPU-MCML.

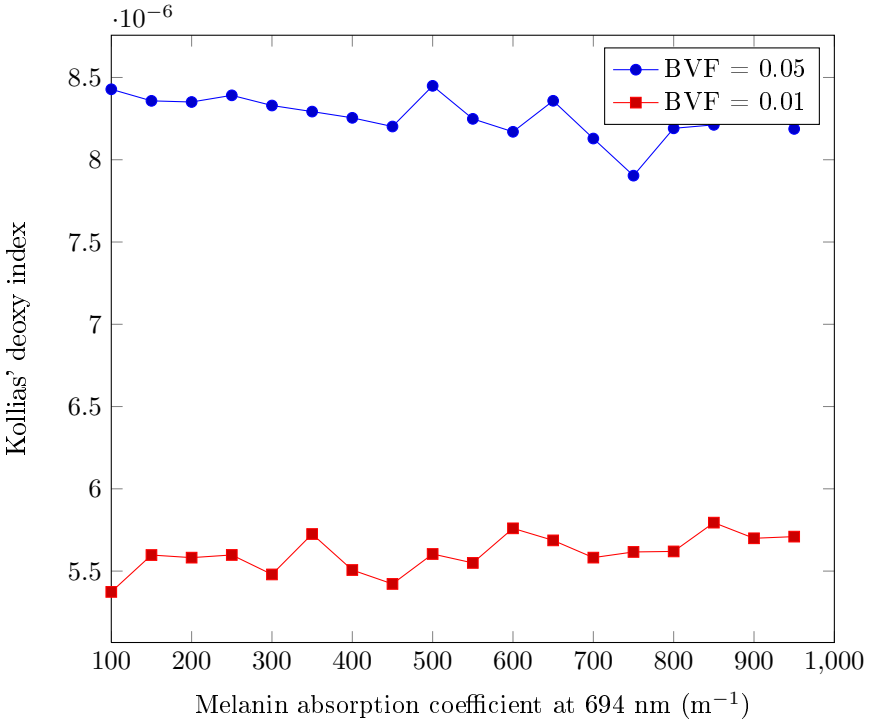

Figure 4.19: Kollias' deoxy fit for different $\mu_{a,m,694}$ for an oxygen saturation of 20%. Simulated using GPU-MCML.

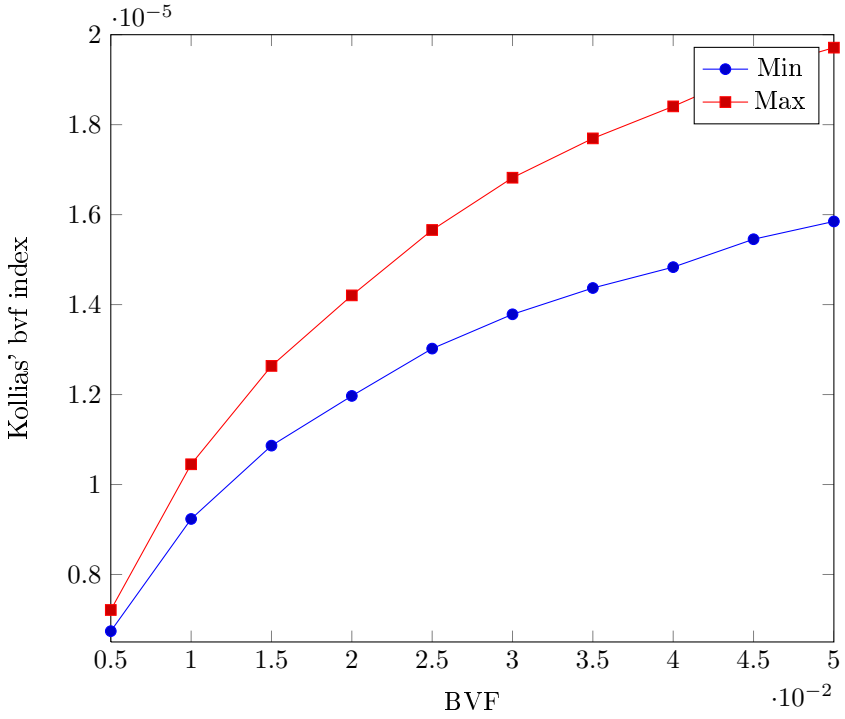

Figure 4.20: Kollias' BVF index as a function of the blood volume fraction, simulated by varying $\mu_{a,m,694}$ between 100 and 1000 m^{-1} and the oxygen saturation between 0.5 and 0.8, and picking the highest and lowest resulting BVF index. Simulated using GPU-MCML.

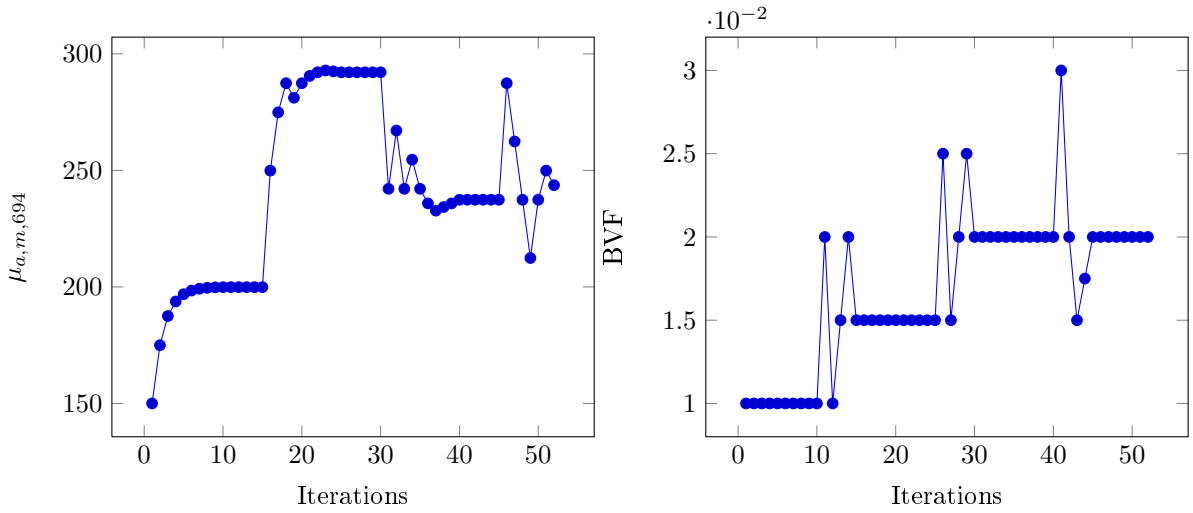

Figure 4.21: Convergence of iteration using Kollias' slopes for a simulated spectrum using $\mu_{a,m,694} = 250$ m^{-1}, oxy = 0.5, BVF = 0.02. Both simulation and iteration using GPU-MCML.

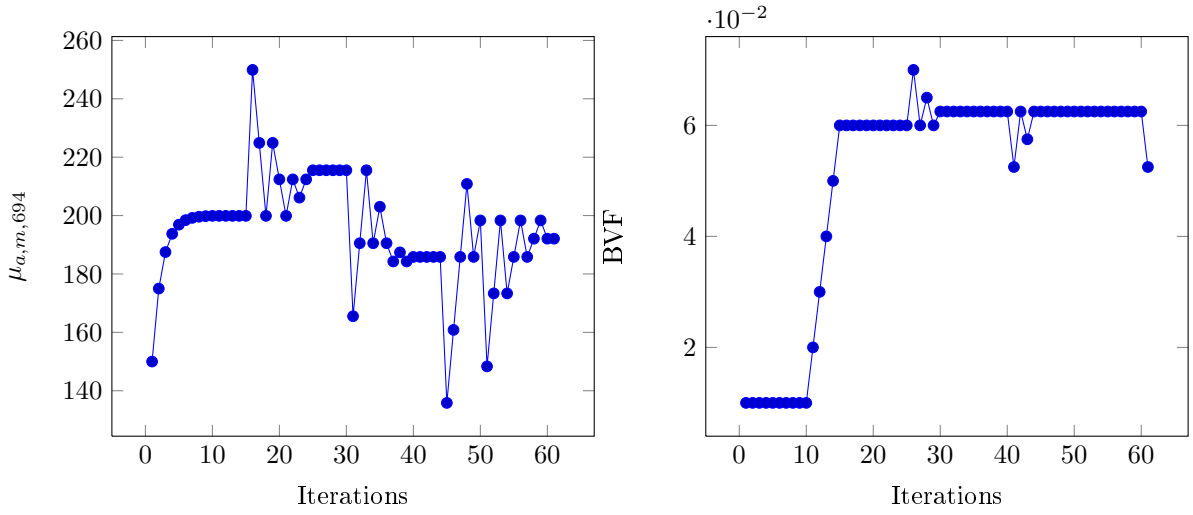

Figure 4.22: Convergence of $\mu_{a,m,694}$ and BVF using sequential iteration based on Kollias' methods for the spectrum in fig. 4.1. Iterated using GPU-MCML.

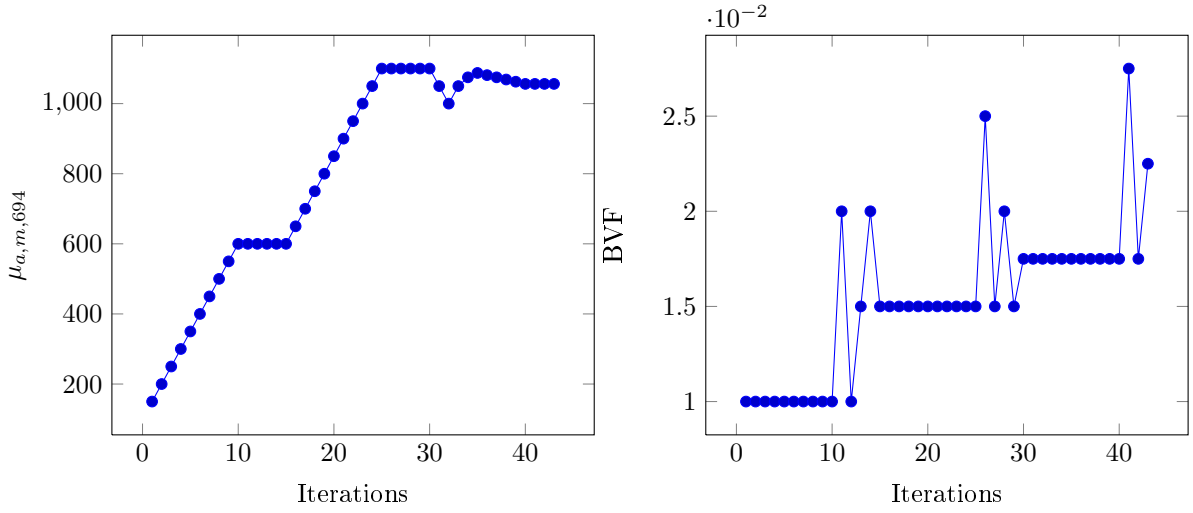

Figure 4.23: Convergence of $\mu_{a,m,694}$ and BVF using sequential iteration based on Kollias' methods for the spectrum in fig. 4.2. Iterated using GPU-MCML.

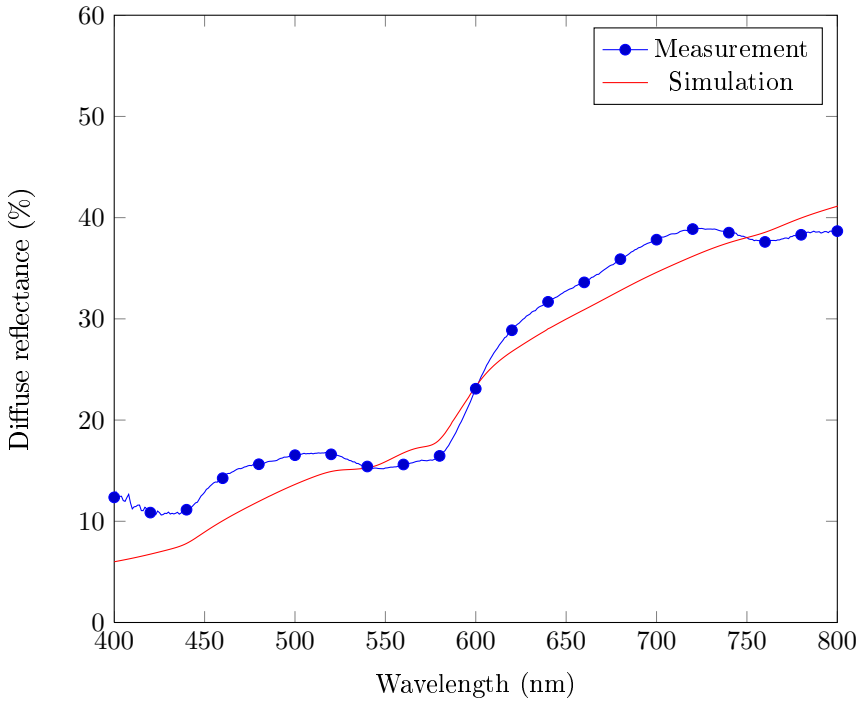

Figure 4.24: Delta-Eddington diffusion model-simulation of the parameter fit from fig. 4.23 along with the spectrum from fig. 4.2.

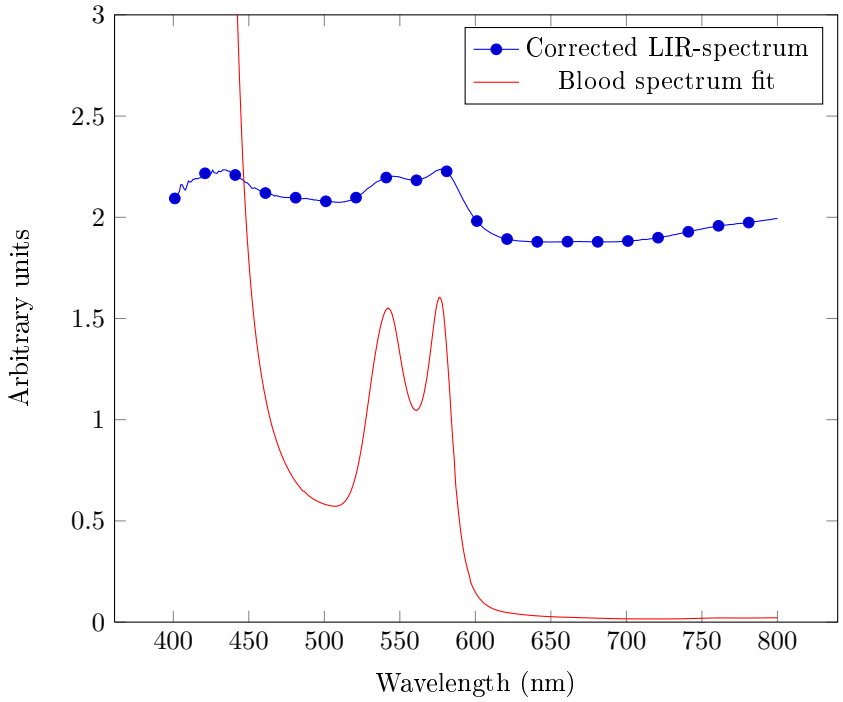

Figure 4.25: The melanin-corrected LIR values plotted alongside the blood spectrum fit.

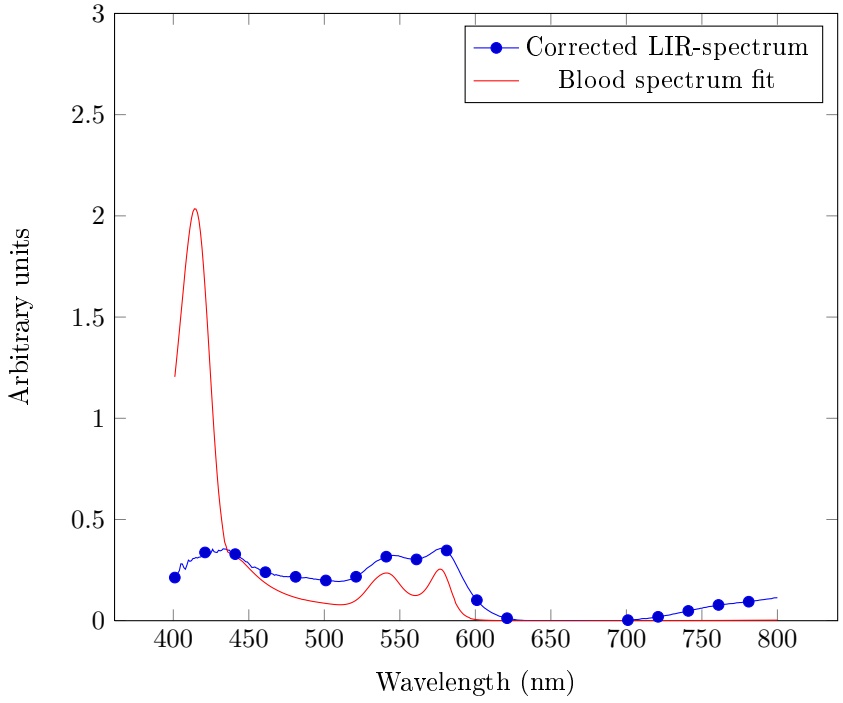

Figure 4.26: The melanin-corrected LIR values, also corrected with a constant factor, plotted alongside the blood spectrum fit.

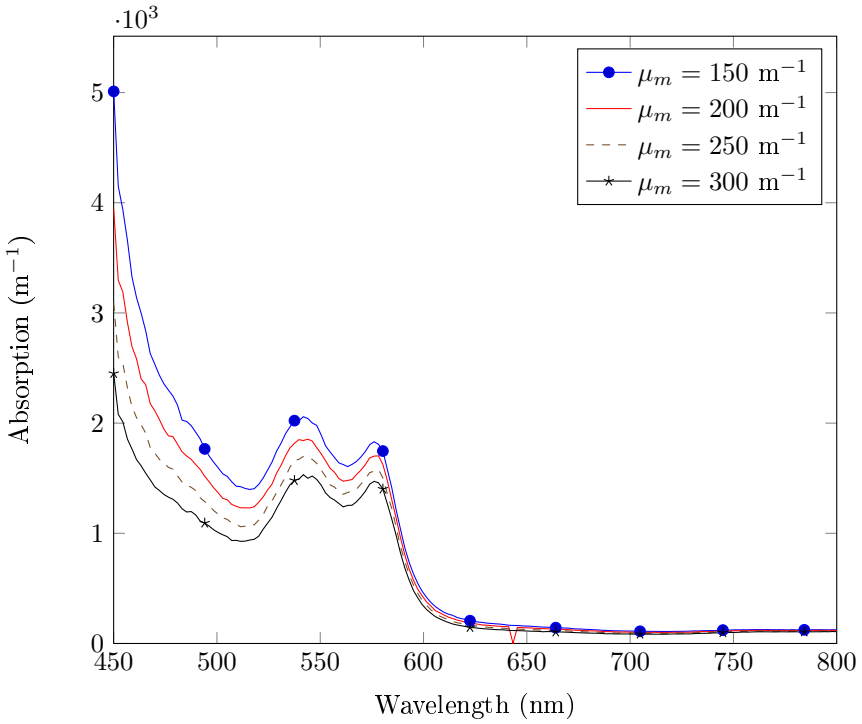

Figure 4.27: Derived μ_d with different initial $\mu_{a,m,694}$ for the spectrum in fig. 4.1.

4.4.3 Subsequential model fitting

Posed by the cross-talk problems of Kollias and Dawson, a different method was investigated, where the melanin absorption and blood volume fraction are in turn iterated with respect to two isosbestic points. Comparing a simulation of the $\mu_{a,m,694}$-fit against the spectrum in figure 4.1 will result in a simulated spectrum that almost has a constant difference to the measured spectrum (see figure 4.28). If one tries to completely bridge the gap between the simulated spectrum and the measured spectrum, either the blood volume fraction or the melanin absorption will be over-estimated in order to provide a high enough absorption to completely bridge the gap, and at the next point, one of the other parameters will be under-estimated in order to correct for an over-estimated absorption at the next point.

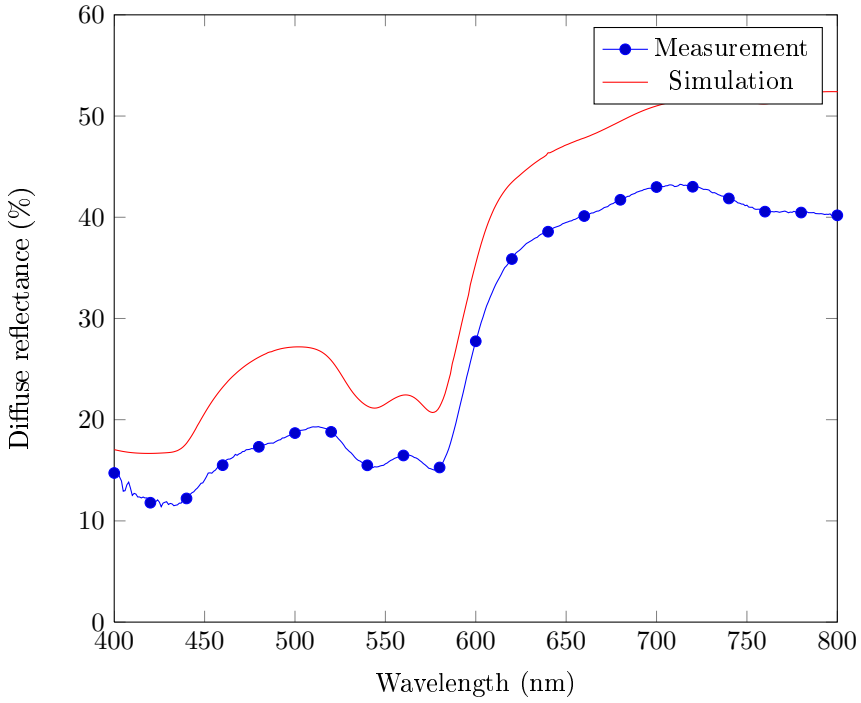

Figure 4.28: Delta Eddington diffusion model-simulation of the parameter fit from fig. 4.22 along with the spectrum from fig. 4.1.

This method should therefore fail, but the use of isosbestic points should be investigated further, in combination with the idea of using more shallow penetration depths.

4.5 Oxygen saturation

The formula in equation (3.5) was used on the Delta-Eddington diffusion model to calculate the apparent oxygen saturation for a range of input oxygen saturations in order to benchmark the method. Results are shown in figure 4.29. It is claimed [21] that the (660, 817) combination will reach greater depths in the skin and give an oxygen saturation larger than the one obtained while using the (540,548) combination, since the blood at the superficial depths is less oxygen-rich. The simulations will show an oxygen saturation that sports this behaviour and has values in the range that the blood should have at these two different depths, but the formula gives this range of oxygen saturation values regardless of the input oxygen saturation. Plus, this behaviour should not be present in the simulations as it represents a homogenous medium with the exact same oxygen saturation at all layers of the skin.

Judging purely from this, the oxygen saturation formula should not be trusted for absolute measurements. For the Kollias simulations, the oxygen saturation was estimated using Kollias' indices.

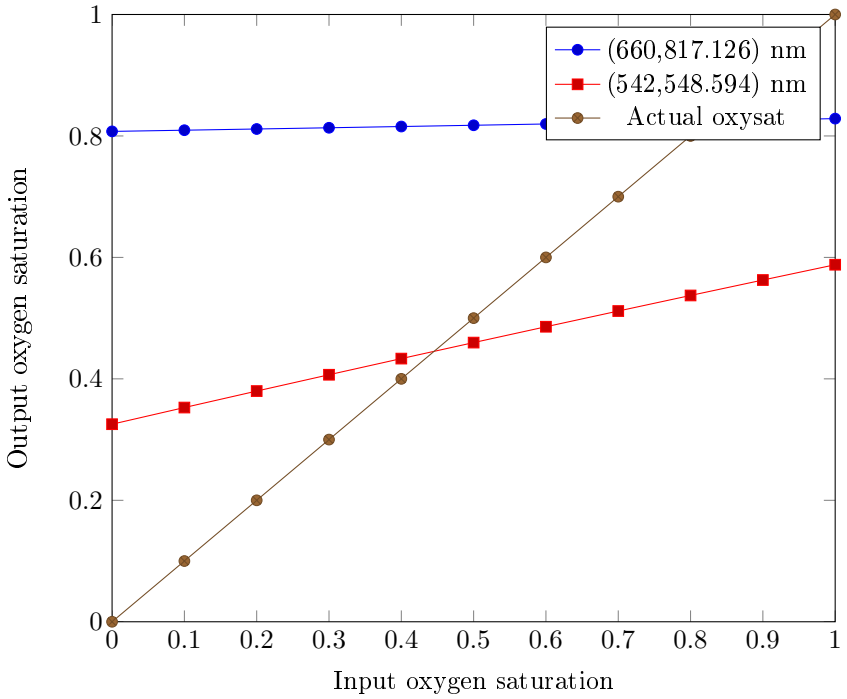

Figure 4.29: Results from the modified Ishimaru formula simulated using Delta-Eddington diffusion model for a range of input oxygen saturations. $\mu_{a,m,694} = 300$ m^{-1}, BVF $= 0.01$.

The corresponding, simulated spectra are in part shown in figure 4.30.

Ishimaru's original formula [17], upon which Randeberg's formula is based, is specifically designed for measuring the blood oxygen saturation for a purely absorbing, hemolyzed solution of blood. A blood solution is hemolyzed when the erythrocyte membranes are ruptured and the blood solution reduced to a purely absorbing medium. When posed by the absorption through such a blood solution, the formula will output the correct values, less so when scattering also is taken into account. Spott's adoption [36] uses the derived dermal absorption coefficients for skin after an inversion in place of Ishimaru's absorption values, which eliminates the scattering. Any additional wavelength-varying absorption at the wavelengths in question can be a problem since Ishimaru's formula needs an exact measure of the combined blood absorption coefficient multiplied by some constant, or else it will not work. Spott is able to solve this by scaling the dermal absorption coefficient to fit the blood absorption spectrum, in addition to the fact that he is estimating the oxygen saturation in a part of the spectrum less dependent on other chromophores than blood.

As earlier mentioned, it can be argued that the inverse of the diffuse reflectance can be estimated to be the absorption spectrum multiplied with some constants, which is the whole basis for Randeberg's formula. Now, the spectrum will be dominated by both an extra, wavelength-dependent absorption from melanin and a wavelength-dependent scattering. If the wavelength-dependence is strong, the formula should be less likely to output the correct values since it will interfere with the extraction of the oxygen saturation from the compound absorption.

However, in the 660 to 817-range, both the melanin absorption and the scattering is only slowly varying, and this combination of wavelengths should have yielded better results. As seen in figure 4.29, it is almost constant. The reasons for this are unknown, although some other strange behavior has also been spotted for the simulations. For instance, the erythema indices for both Dawson's and Kollias' cases were following some logarithmic function instead of being linear with respect to the input parameters. The method cannot completely be discarded even if the simulations show it to give less likely results, the simulations will have to be investigated in more detail first.

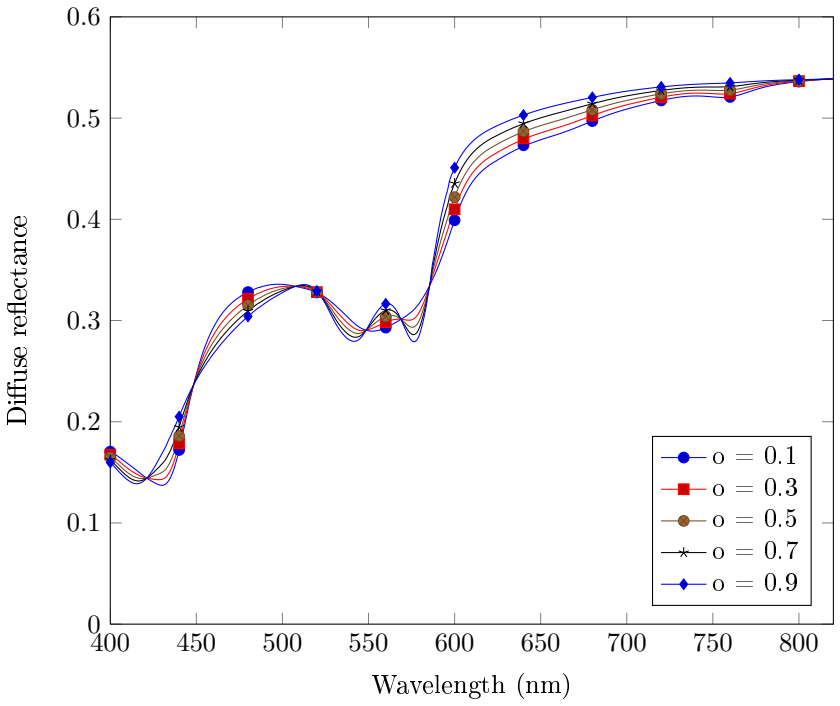

Figure 4.30: Simulations using the Delta-Eddington diffusion model for a range of input oxygen saturations. $\mu_{a,m,694} = 300$ m^{-1}, BVF $= 0.01$.

Chapter 5

Conclusion and further work

An existing GPU-implementation of Monte Carlo simulation of light transport in multi-layered tissue has successfully been altered to output the reflectance for the whole wavelength range in the visible spectrum within 40 seconds, as compared to the traditional MCML package, which will use 20 minutes with considerably fewer photons per simulation. An inverse model has also successfully been implemented using GPU-MCML, but was considered too slow for real-time use. The running time of the forward model will however be sufficient, and can be used for benchmarking of other results. Some future work here could be to provide some GUI or at least streamline the code a bit more.

The traditional methods for evaluating the melanin content of human skin was evaluated and found insufficient, and a major part of the future work will have to be to find a better solution to this problem. Some ideas have been proposed which will be looked into. There is also hyperspectral data available of skin with both different blood volume fractions and different melanin contents present in the same patch of skin, for which any new methods may be benchmarked.

A GPU-implementation of an inverse model using the analytical diffusion theory has been implemented and found to output a finished result well within the deadlines defined by the hyperspectral camera hardware. As the lowest of the low-end GPUs were used, this is likely to perform even better as the GPU in use is replaced by one with the latest compute capabilities. The inverse chain does theoretically work, and has been proven to work on single spectras. The hyperspectral images are however quite noisy, and some algorithm for good noise removal on a per-line basis is needed before the inverse chain can be tested on real hyperspectral data. There exists good noise removal tools for hyperspectral images, but these are best applied on full images and not lines. Delta-Eddington should be implemented for the inverse chain, as should the three-layer model. The scattering functions should be updated with newer findings, as should the rest of the model.

The hyperspectral images must be calibrated. The hyperspectral camera itself will provide automatic radiometric calibration, but the spectras will also have to be calibrated against and corrected for non-uniformity of the light source. A calibration slab will always be inserted in the start of each hyperspectral scan. Detecting this slab correctly and integrating over its interiors before starting the inverse processing will have to be done in a stable and fast way within some time limits. Results from real time robotics and computer vision may probably be used.

Spectral unmixing will have to be implemented for the result from the inverse chain, where the absorption spectrum of each cromophore is fitted against the output dermal absorption coefficient, and the results must be checked. It is likely that the spectras cannot be fit all at once throughout the whole wavelength range, since the penetration depths will vary with the absorption coefficient, and the absorption coefficient will represent properties gained from different depths. The skin is not homogenous, and the spectras must therefore be fit within ranges within which the penetration depth is constant.

The inverse chain must also be integrated into some framework. Right now, it is called from a stand-alone client for the hyperspectral camera which also can be supplied with images from a mock-up server, but in the end, the inverse chain is supposed to be a part of a larger hyperspectral processing framework

developed by Forsvarets Forskningsinstitutt. Due to the way the inverse chain is written, this should however not be very challenging.

The inverse model must also rigorously be tested for stability and whether it will reach its deadlines even when disturbed by other applications. Timing and actual real time issues have not been addressed in this project, this project has only ensured the existence of an inverse model fast enough for embedding in a real time environment.

Bibliography

[1] http://code.google.com/p/gpumcml/. Visited 2012-12-04.

[2] http://omlc.ogi.edu/software/mc/. Visited 2012-12-03.

[3] *CUDA C Best Practices Guide*, October 2012. http://docs.nvidia.com/cuda.

[4] *CUDA C Programming Guide*, October 2012. http://docs.nvidia.com/cuda.

[5] E. Alerstam, W. C. Y. Lo, T. D. Han, J. Rose, S. Andersson-Engels, and L. Lilge. Next-generation acceleration and code optimization for light transport in turbid media using gpus. *Biomed Opt Express*, 1(2):658–675, Sep 2010.

[6] F. Bevilacqua, D. Piguet, P. Marquet, J. D. Gross, B. J. Tromberg, and C. Depeursinge. In vivo local determination of tissue optical properties: Applications to human brain. *Appl Opt*, 38(22):4939–4950, Aug 1999.

[7] T. Binzoni, T. S. Leung, A. H. Gandjbakhche, D. Rufenacht, and D. T. Delpy. Comment on 'the use of the henyey-greenstein phase function in monte carlo simulations in biomedical optics'. *Phys Med Biol*, 51(22):L39, 2006.

[8] T. Binzoni, T. S. Leung, A. H. Gandjbakhche, D. Rufenacht, and D. T. Delpy. The use of the henyey-greenstein phase function in monte carlo simulations in biomedical optics. *Phys Med Biol*, 51(17):N313, 2006.

[9] K. W. Calabro, E. Aizenberg, and I. J. Bigio. Improved empirical models for extraction of tissue optical properties from reflectance spectra. *Proc SPIE*, 8230:82300H–82300H–7, 2012.

[10] J. B. Dawson, D. J. Barker, D. J. Ellis, J. A. Cotterill, E. Grassam, G. W. Fisher, and J. W. Feather. A theoretical and experimental study of light absorption and scattering by in vivo skin. *Phys Med Biol*, 25(4):695, 1980.

[11] M. Ferguson-Pell and S. Hagisawa. An empirical technique to compensate for melanin when monitoring skin microcirculation using reflectance spectrophotometry. *Med Eng Phys*, 17(2):104 – 110, 1995.

[12] S. I. Fox. *Human Physiology*. McGraw-Hill Higher Education, 2010.

[13] I. Fredriksson, M. Larsson, and T. Stromberg. Inverse monte carlo method in a multilayered tissue model for diffuse reflectance spectroscopy. *J Biomed Opt*, 17(4):047004–1–047004–12, 2012.

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

[15] C. K. Hayakawa, J. Spanier, F. Bevilacqua, A. K. Dunn, J. S. You, B. J. Tromberg, and V. Venugopalan. Perturbation monte carlo methods to solve inverse photon migration problems in heterogeneous tissues. *Opt Lett*, 26(17):1335–1337, Sep 2001.

[16] L.G. Henyey and J.L Greenstein. Diffuse radiation in the galaxy. *Astrophys J*, 93:70–83, 1941.

[17] A. Ishimaru. *Wave Propagation and Scattering in Random Media*. Wiley-IEEE Press, 1997.

[18] S. M. Kay. *Fundamentals of Statistical Signal Processing: Estimation theory.* Prentice Hall PTR, 1993.

[19] N. Kollias and A. Baqer. Spectroscopic characteristics of human melanin in vivo. *J Invest Dermatol*, 85:38–42, 1985.

[20] N. Kollias and A. Baqer. On the assessment of melanin in human skin in vivo. *Photochem Photobiol*, 43:49–54, 1986.

[21] L. L. Randeberg, E. B. Roll, L. T. Norvang Nilsen, T. Christensen, and L. O. Svaasand. In vivo spectroscopy of jaundiced newborn skin reveals more than a bilirubin index. *Acta Paediatr*, 94(1):65–71, 2005.

[22] I. Nishidate, Y. Aizu, and H. Mishina. Estimation of melanin and hemoglobin in skin tissue using multiple regression analysis aided by monte carlo simulation. *J Biomed Opt*, 9(4):700–710, 2004.

[23] L. Norvang, T. Milner, J. Nelson, M. Berns, and L. Svaasand. Skin pigmentation characterized by visible reflectance measurements. *Laser Med Sci*, 12:99–112, 1997. 10.1007/BF02763978.

[24] L. T. Norvang, E. J. Fiskerstrand, J. S. Nelson, M. W. Berns, and L. O. Svaasand. Epidermal melanin absorption in human skin. *Proc SPIE*, 2624:143–154, 1996.

[25] G. M. Palmer and N. Ramanujam. Monte carlo-based inverse model for calculating tissue optical properties. part i: Theory and validation on synthetic phantoms. *Appl Opt*, 45(5):1062–1071, Feb 2006.

[26] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes.* Cambridge University Press, 2007.

[27] L. L. Randeberg, E. L. P. Larsen, and L. O. Svaasand. Characterization of vascular structures and skin bruises using hyperspectral imaging, image analysis and diffusion theory. *J Biophotonics*, 3(1-2):53–65, 2010.

[28] L. L. Randeberg, A. Winnem, R. Haaverstad, and L. O.. Svaasand. Performance of diffusion theory vs monte carlo methods. In *Diagnostic Optical Spectroscopy in Biomedicine III*, page ThB3. Optical Society of America, 2005.

[29] L.L. Randeberg, J.H. Bonesronning, M. Dalaker, J.S. Nelson, and L.O. Svaasand. Methemoglobin formation during laser induced photothermolysis of vascular skin lesions. *Laser Surg Med*, 34(5):414–419, 2004.

[30] P. Tate R.R. Seeley, T.D. Stephens. *Essentials of Anatomy and Physiology.* McGraw-Hill Publishing Co., 2006.

[31] I. S. Saidi, S. L. Jacques, and F. K. Tittel. Mie and rayleigh modeling of visible-light scattering in neonatal skin. *Appl Opt*, 34(31):7410–7418, Nov 1995.

[32] C. R. Simpson, M. Kohl, M. Essenpreis, and M. Cope. Near-infrared optical properties of ex vivo human skin and subcutaneous tissues measured using the monte carlo inversion technique. *Phys Med Biol*, 43(9):2465, 1998.

[33] S.A. Prahl S.L. Jacques, C.A. Alter. Angular dependence of hene laser light scattering by human dermis. *Lasers Life Sci*, 1:309–333, 1987.

[34] T. Spott. *Characterization of layered tissue structures with diffusively propagating photon-density waves.* PhD thesis, NTNU, 1999.

[35] T. Spott and L. O. Svaasand. Collimated light sources in the diffusion approximation. *Appl Opt*, 39(34):6453–6465, Dec 2000.

[36] T. Spott, L. O. Svaasand, R. E. Anderson, and P. F. Schmedling. Application of optical diffusion theory to transcutaneous bilirubinometry. *Proc SPIE*, 3195:234–245, 1998.

[37] G. N. Stamatas and N. Kollias. Blood stasis contributions to the perception of skin pigmentation. *J Biomed Opt*, 9(2):315–322, 2004.

[38] G.N. Stamatas, B.Z. Zmudzka, N. Kollias, and J.Z. Beer. In vivo measurement of skin erythema and pigmentation: new means of implementation of diffuse reflectance spectroscopy with a commercial instrument. *Br J Dermatol*, 159(3):683–690, 2008.

[39] J.M Steinke and A. P. Shepherd. Comparison of mie theory and the light scattering of blood cells. *Appl Opt*, 27:4027–4033, 1988.

[40] J.M. Steinke and A.P. Shepherd. Diffusion model of the optical absorbance of whole blood. *J Opt Soc Am A*, 5:813–822, 1988.

[41] L. Svaasand, L. Norvang, E. Fiskerstrand, E. Stopps, M. Berns, and J. Nelson. Tissue parameters determining the visual appearance of normal skin and port-wine stains. *Laser Med Sci*, 10:55–65, 1995. 10.1007/BF02133165.

[42] T. Theoharis, G. Papaioannou, N. Platis, and N. M. Patrikalakis. *Graphics and Visualization: Principles & Algorithms*. A. K. Peters, 2007.

[43] M. J. C. van Gemert, S. L. Jacques, H. J. C. M. Sterenborg, and W. M. Star. Skin optics. *IEEE Trans Biomed Eng*, 36:1146–1154, 1989.

[44] W. Verkruysse, L. O. Svaasand, W. Franco, and J. S. Nelson. Remittance at a single wavelength of 390nm to quantify epidermal melanin concentration. *J Biomed Opt*, 14(1):014005–014005–4, 2009.

[45] L. Wang, S. L. Jacques, and L. Zheng. Mcml monte carlo modeling of light transport in multi-layered tissues. *Comput Meth Prog Bio*, 47(2):131 – 146, 1995.

[46] L.V. Wang and H. Wu. *Biomedical Optics, Principles and Imaging*. John Wiley & Sons, 2007.

[47] M. L. Wolbarsht, A. W. Walsh, and G. George. Melanin, a unique biological absorber. *Appl Opt*, 20:2184–2186, 1981.

[48] D. Yudovsky and L. Pilon. Rapid and accurate estimation of blood saturation, melanin content, and epidermis thickness from spectral diffuse reflectance. *Appl Opt*, 49(10):1707–1719, Apr 2010.

[49] C. Zhu, G. M. Palmer, T. M. Breslin, J. Harter, and N. Ramanujam. Diagnosis of breast cancer using diffuse reflectance spectroscopy: Comparison of a monte carlo versus partial least squares analysis based feature extraction technique. *Laser Surg Med*, 38(7):714–724, 2006.

[50] W. G. Zijlstra, A. Buursma, and O. W. van Assendelft. *Visible and near infrared absorption spectra of human and animal haemoglobin*. VSP, Utrecht, 2000.

Appendix A

Source code

A.1 GPU-DM

A TCP/IP-client was also developed by assembling code parts from examples provided by Norsk Elektro Optikk, but this was not included since the calibration step was not resolved, only simplified pseudo-code using some prototypes is displayed.

File A.1: GPU–DM/gpudm.h

```
1  #ifndef GPUDM_H_DEFINED
2  #define GPUDM_H_DEFINED
3
4
5  //for quick access and better control of the arrays allocated in the GPU and
       containing the optical properties
6  //references to all the GPU arrays
7  typedef struct{
8      //epidermis
9      float *muae;
10     float *muse;
11
12     //dermis
13     float *muad;
14     float *musd;
15
16     //iteration arrays
17     float *refl;
18     float *deriv;
19     float *prev_muad; //buffered, previous line of muad
20     float *lineData_gpu_1; //linedata 1
21     float *lineData_gpu_2; //linedata 2. Will switch between these two
           depending on which one is the buffer
22     int currLineData; //contains either 1 or 2, indicating lineData_gpu_1 or 2.
            Initially set to 0
23     bool safeToTransferMuad; //whether the prev_muad array contains any data or
            not
24
25     //spectras
26     //arrays containing absorption and scattering spectrum values corresponding
            to the wavelengths defined in the hyperspectral image header for
           different cromophores
27     float *muh_oxy;
28     float *muh_deoxy;
29     float *melanin_base; //contains (695/lambda)^3.46
```

```
    float *gcol; //anisotropy factor for collagen
    float *musm; //unreduced mie scattering
    float *musr; //unreduced rayleigh scattering coefficient
    float *musb_base; //unreduced blood scattering coefficient

    float *oxy;
    float *bvf;
    float *muam694;

    //memory properties
    size_t pitch;
    size_t byteWidth;
    size_t height;

    //camera properties
    int spectral_size; //number of wavelengths
    int spatial_size; //number of pixels in one line
    int threads_per_block; //threads per block
    float *cal_array; //calibration array, containing line_spec .* cal_slab
} GPUDMArrays;

extern "C"
void gpuprocessing(GPUDMArrays *a, float *nextLineData, float *prevMuad);

extern "C"
void gpuallocate(GPUDMArrays *arrays, float *wavelengths, float *cal, int
    spectral_size, int spatial_size);

extern "C"
void gpufree(GPUDMArrays *arrays);

//pointers to pointers because there is no other way the function will be able
    to change the reference of the pointer and allocate memory and keep the
    changes outside of scope

void gpuallocate_prop(int spectral_size, int spatial_size, size_t byteWidth,
    size_t height, size_t *pitch, float **muae, float **muse, float **muad,
    float **musd);
void gpuallocate_iter(int spectral_size, int spatial_size, size_t byteWidth,
    size_t height, size_t *pitch,float **refl, float **deriv, float **prev_muad,
     float **lineData_1, float **lineData_2);
void gpuallocate_spect(float *wavelengths, int spectral_size, int spatial_size,
     size_t byteWidth, size_t height, size_t *pitch,float **muh_oxy, float **
    muh_deoxy, float **melanin_base, float **gcol, float **musm, float **musr,
    float **musb_base);


#endif
```

File A.2: GPU–DM/gpudm_main.cu

```
#include <time.h>
#include "cameraHeader.h"
#include "muh_blood.h"
#include <iostream>
#include "gpudm.h"
using namespace std;

//GPU kernels
__global__ void calcMelaninBvfOxy(float *oxy, float *bvf, float *muam694, float
     *lineData, size_t inputPitch);
```

```
10 __global__ void calcSkinData(float *oxy, float *bvf, float *muam694, float *
       muae, float *muse, float *muad, float *musd,
11                 float *muh_oxy, float *muh_deoxy, float *melanin_base, float *
                       musm, float *musr, float *musb_base, size_t pitch);
12 __global__ void ReflIsoL2(float *muae, float *muse, float *muad, float *musd,
       float *res, float *gcol, float *deriv, size_t pitch); //gd and ge are read
       in as texture arrays since they do not differ from the standard gcol anyway
13 __global__ void nextMuad(float *lineData, float *refl, float *deriv, float *
       muad, size_t pitch);
14 __global__ void calibrate(float *lineData, float *cal, size_t pitch);
15
16 #define div13 1.0f/3.0f
17 #define A 0.14386f
18 #define de 100e-6*1.0f
19
20
21 //takes in the arrays and the next line, which will only be transferred and
       processed in the next round
22 void gpuprocessing(GPUDMArrays *a, float *nextLineData, float *prevMuad){
23         float *refl = new float[a->spectral_size*a->spatial_size*sizeof(float)
               ];
24     //timers for timing the first call to ReflIsoL2
25     cudaEvent_t start, preprocessing, stop;
26     cudaEventCreate(&start);
27     cudaEventCreate(&stop);
28     cudaEventCreate(&preprocessing);
29     if (a->currLineData == 0){
30         //first line of data, can only be transferred, no processing at this
               stage
31         cerr << cudaMemcpy2D(a->lineData_gpu_1, a->pitch, nextLineData, a->
               byteWidth, a->byteWidth, a->height, cudaMemcpyHostToDevice) << endl;
32         a->currLineData = 1;
33     } else {
34         //find out which array is buffered and not
35         float *currLineData; //current line to be processed
36         float *nextLineData_gpu; //reference to gpu array to which the next
               line data should be transferred
37         if (a->currLineData == 1){
38             a->currLineData = 2;
39             currLineData = a->lineData_gpu_1;
40             nextLineData_gpu = a->lineData_gpu_2;
41         } else if (a->currLineData == 2) {
42             a->currLineData = 1;
43             currLineData = a->lineData_gpu_2;
44             nextLineData_gpu = a->lineData_gpu_1;
45         } else {
46             a->currLineData = 0; //something went wrong? Try to reset
47             return;
48         }
49         //create streams
50         cudaStream_t transferStream;
51         cudaStream_t processingStream;
52         cudaStreamCreate(&transferStream);
53         cudaStreamCreate(&processingStream);
54
55         //pagelocked memory for async transfer
56         float *nextLineData_pagelocked;
57         cerr << cudaMallocHost(&nextLineData_pagelocked, a->byteWidth*a->height
               ) << endl;
58         cudaMemcpyAsync(nextLineData_pagelocked, nextLineData, a->byteWidth*a->
               height, cudaMemcpyHostToHost, transferStream);
```

```
59
60          cudaMemcpy2DAsync(nextLineData_gpu, a->pitch, nextLineData_pagelocked,
                a->byteWidth, a->byteWidth, a->height, cudaMemcpyHostToDevice,
                transferStream);
61
62          float *prevMuad_pagelocked;
63          if (a->safeToTransferMuad){
64              cudaMallocHost(&prevMuad_pagelocked, a->byteWidth*a->height);
65              cudaMemcpy2DAsync(prevMuad_pagelocked, a->byteWidth, a->prev_muad,
                    a->pitch, a->byteWidth, a->height, cudaMemcpyDeviceToHost,
                    transferStream);
66          }
67
68          float deadline = 25.0f;
69
70          //set up thread grid and blocks
71          int block_number_x = a->spatial_size/a->threads_per_block;
72          int block_number_y = a->spectral_size;
73          dim3 dimBlock(a->threads_per_block);
74          dim3 dimGrid(block_number_x, block_number_y);
75
76
77          //block and grid sizes for melanin determination
78          //FIXME: Should be more dynamic and choose the number of
                multiprocessors, not 16
79          //since the melanin and bvf determination will do uncoalesced reads
                anyway, no purpose in making the block dimension be a multiple of
                32...
80          dim3 dimBlockMel(160);
81          dim3 dimGridMel(10);
82
83          size_t inputPitch = a->pitch/sizeof(float);
84          //cudaEventRecord(start, 0);
85
86          //divide by reference
87          calibrate<<<dimGrid, dimBlock, 0, processingStream>>>(currLineData, a->
                cal_array, inputPitch);
88
89          //can't input the structs into the function since the memory won't be
                aligned correctly
90          //calcMelaninBvfOxy<<<dimGridMel, dimBlockMel>>>(a->oxy, a->bvf, a->
                muam694, currLineData, inputPitch);
91          calcMelaninBvfOxy<<<dimGrid, dimBlock>>>(a->oxy, a->bvf, a->muam694,
                currLineData, inputPitch);
92          calcSkinData<<<dimGrid, dimBlock, 0, processingStream>>>(a->oxy, a->bvf
                , a->muam694, a->muae, a->muse, a->muad, a->musd, a->muh_oxy, a->
                muh_deoxy, a->melanin_base, a->musm, a->musr, a->musb_base,
                inputPitch);
93
94
95          //cudaEventRecord(preprocessing, 0);
96
97          //first iteration
98          ReflIsoL2<<<dimGrid, dimBlock, 0, processingStream>>>(a->muae, a->muse,
                 a->muad, a->musd, a->gcol, a->refl, a->deriv, inputPitch);
99          nextMuad<<<dimGrid, dimBlock, 0, processingStream>>>(currLineData, a->
                refl, a->deriv, a->muad, inputPitch);
100         //cudaEventRecord(stop, 0);
101         //cudaEventSynchronize(stop);
102
103         float preproc_time;
```

```
104         float iter_time;
105         cudaEventElapsedTime(&preproc_time, start, preprocessing);
106         cudaEventElapsedTime(&iter_time, preprocessing, stop);
107 
108         int num_iterations = 30;//(int)(deadline - preproc_time)/iter_time; //
                the number of iterations we have time to do
109 
110 
111         for (int i=0; i < num_iterations; i++){
112             ReflIsoL2<<<dimGrid, dimBlock, 0, processingStream>>>(a->muae, a->
                    muse, a->muad, a->musd, a->gcol, a->refl, a->deriv, inputPitch);
113             nextMuad<<<dimGrid, dimBlock, 0, processingStream>>>(currLineData,
                    a->refl, a->deriv, a->muad, inputPitch);
114         }
115         if (a->safeToTransferMuad){
116             cerr << cudaMemcpyAsync(prevMuad, prevMuad_pagelocked, a->byteWidth
                    *a->height, cudaMemcpyHostToHost, transferStream) << endl;
117         }
118 
119         cudaDeviceSynchronize();
120 
121         //copy muad to prevmuad so that it may be transferred back to the host
                in the next processing step
122         cerr << cudaMemcpy2D(a->prev_muad, a->pitch, a->muad, a->pitch, a->
                byteWidth, a->height, cudaMemcpyDeviceToDevice) << endl;
123         a->safeToTransferMuad = true;
124     }
125 }
126 
127 //takes in the skin optical properties and outputs the reflectance to res, and
        the derivative of the reflectance with respect to muad to deriv
128 //isotropic source function, two-layer model
129 __global__ void ReflIsoL2(float *muae, float *muse, float *muad, float *musd,
        float *gcol, float *res, float *deriv, size_t pitch){
130     int ind = (gridDim.x*blockIdx.y + blockIdx.x)*pitch+ threadIdx.x;
131     float g = gcol[ind];
132 
133     //reduced scattering coefficients
134     float musr1 = muse[ind]*(1.0f-g);
135     float musr2 = musd[ind]*(1.0f-g);
136 
137     //move mua into shared memory
138     float curr_muae = muae[ind];
139     float curr_muad = muad[ind];
140 
141     //diffusion constant
142     float D1 = fdividef(1.0f,3.0f*(musr1 + curr_muae));
143     float D2 = fdividef(1.0f,3.0f*(musr2 + curr_muad));
144 
145     float musr2dmusr1 = fdividef(musr2,musr1); //musr2 divided by musr1, keep
            for derivative calc
146 
147 
148     //optical penetration depth
149     float del1 = sqrtf(fdividef(D1,curr_muae));
150     float del2 = sqrtf(fdividef(D2,curr_muad));
151 
152     //from Svaasand 1995
153     //calculate the reflectance value
154     float sinhval = sinhf(fdividef(de, del1));
155     float coshval = coshf(fdividef(de, del1));
```

```
156     float fact1 = del1*musr1*A;
157     float f1 = (del1*del1*del2*div13-del1*del1*D2)*coshval+(del1*del1*del1*
            fdividef(D2,D1)*div13 - del1*del2*D1)*sinhval; //keeping for derivative
            calc
158     float f2 = 1.0f+fdividef(del2, D2)*div13; //keeping for derivative calc
159     float expval = expf(-fdividef(de,D1)*div13);
160     float f3 = (musr2dmusr1*del2*del2*D1*(del1*fdividef(del1,D1*D1)*div13*div13
            -1.0f)+del1*del1*(D2-del2*fdividef(del2,D2)*div13*div13))*expval; //
            keeping for derivative calc
161     float f4 = del1*fdividef(del1,D1*D1)*div13*div13 - 1.0f; //keep for
            derivative calc, f4
162     float f5 = fdividef(del2,D2)*div13+1.0f; //keep for derivative calc, f5
163     float f6 = D1*del1*(D2+del2*A)*coshval+(D1*D1*del2 + D2*del1*del1*A)*
            sinhval; //keep for derivative calc, f6
164     float num = fact1*(f1*f2+f3);
165     float denom = f4*f5*f6;
166     res[ind] = fdividef(num, denom);
167
168     //calculate the derivative with respect to muad
169     float dD2dmuad = -3.0f*D2*D2;
170     float ddel2dmuad = (dD2dmuad*curr_muad-D2)*fdividef(1.0f, curr_muad*
            curr_muad)*fdividef(1.0f, 2.0f*del2);
171     float df2dmuad = div13*(ddel2dmuad*D2 - del2*dD2dmuad)*fdividef(1.0f, D2*D2
            );
172     deriv[ind] = (fact1*((coshval*(del1*del1*div13*ddel2dmuad - del1*del1*
            dD2dmuad) + sinhval*(del1*del1*del1*fdividef(1.0f, D1)*div13*dD2dmuad -
            del1*D1*ddel2dmuad))*f2 + f1*df2dmuad + expval*(musr2dmusr1*2.0f*del2*
            ddel2dmuad*D1*(fdividef(del1*del1,9.0f*D1*D1)-1.0f) + del1*del1*(
            dD2dmuad + fdividef(1.0f, 9.0f*curr_muad*curr_muad))))*denom - (f4*(
            df2dmuad*f6 + D1*del1*(dD2dmuad + ddel2dmuad*A)*coshval + (D1*D1*
            ddel2dmuad + dD2dmuad*del1*del1*A)*sinhval*f5)*num))*fdividef(1.0f,
            denom*denom);
173 }
174
175 //for calibration against the reference, _not_ the radiometric calibration
176 __global__ void calibrate(float *lineData, float *cal, size_t pitch){
177     int ind = (gridDim.x*blockIdx.y + blockIdx.x)*pitch+ threadIdx.x;
178     float currCal = cal[ind];
179     float currRefl = lineData[ind];
180     lineData[ind] = fdividef(currRefl, currCal);
181 }
182
183 //calculate the next muad based on the simulated reflectance, measured
        reflectance and derivative of the simulated reflectance using Newton-Rhapson
184 __global__ void nextMuad(float *lineData, float *refl, float *deriv, float *
        muad, size_t pitch){
185     int ind = (gridDim.x*blockIdx.y + blockIdx.x)*pitch + threadIdx.x;
186     float currMuad = muad[ind];
187     float currDeriv = deriv[ind];
188     float currLineData = lineData[ind];
189     float currRefl = refl[ind];
190
191     //newton's method
192     currMuad = currMuad - fdividef(currRefl-currLineData, currDeriv);
193
194     //correction in case muad wants to be negative, which we seriously don't
            want
195     currMuad = currMuad*(1-signbit(currMuad)) + signbit(currMuad);
196     muad[ind] = currMuad;
197 }
198
```

```
199 //takes in pre-allocated arrays and the arrays containing the bases of the
        different absorption coefficients, fills the skin data arrays with the
        optical properties
200 __global__ void calcSkinData(float *oxy_arr, float *Bd_arr, float *muam694_arr,
         float *muae, float *muse, float *muad, float *musd,
201                 float *muh_oxy, float *muh_deoxy, float *melanin_base, float *
                        musm, float *musr, float *musb_base, size_t pitch){
202     //walk down the lines, walk along the blocks, walk along the threads inside
            the block
203     int index = (gridDim.x*blockIdx.y + blockIdx.x)*pitch + threadIdx.x;
204
205     //absorption properties
206     float H = 0.41;
207     float H0 = 0.45;
208     float Be = 0.002;
209     float oxy = oxy_arr[index];
210     float Bd = Bd_arr[index];
211     float muam694 = muam694_arr[index];
212     float mua_other = 25; //FIXME
213     float muab_blood = (muh_oxy[index]*oxy + muh_deoxy[index]*(1-oxy))*fdividef
            (H,H0);
214     float mua_melanin = muam694*melanin_base[index];
215     muae[index] = mua_melanin + muab_blood*Be + mua_other*(1-Be);
216     muad[index] = muab_blood*Bd + mua_other*(1-Bd);
217
218     //scattering properties
219     float c_ray = 1.05e12;
220     float c_mie = 105;
221     float must = musm[index]*c_mie*100 + musr[index]*c_ray*100;
222     float musb685 = 55.09e-12;
223     float ve = 1.25e-16;
224     float musb = musb685*H*(1-H)*(1.4-H)*fdividef(1.0f,ve)*musb_base[index];
225     muse[index] = must*(1-Be)+musb*Be;
226     musd[index] = must*(1-Bd)+musb*Bd;
227 }
228
229 __global__ void calcMelaninBvfOxy(float *oxy, float *bvf, float *muam694, float
         *lineData, size_t pitch){
230     //int pixel = threadIdx.x + (blockIdx.x * blockDim.x)*pitch;
231     //int index;
232     //for (int i=0; i < 160; i++){
233     //  index = (10*pitch)*i + pixel;
234     //  oxy[index] = 0.6;
235     //  bvf[index] = 0.01;
236     //  muam694[index] = 250;
237     //}
238     int index = (gridDim.x*blockIdx.y + blockIdx.x)*pitch + threadIdx.x;
239     oxy[index] = 0.6;
240     bvf[index] = 0.05;
241     muam694[index] = 250;
242 }
243
244
245 //allocate the arrays in the GPU
246 //needs a list of wavelengths and the calibration array in which the elements
        are to divide each element in each line data array
247 void gpuallocate(GPUDMArrays *arr, float *wavelengths, float *cal_array, int
        spectral_size, int spatial_size){
248     arr->currLineData = 0;
249     arr->safeToTransferMuad = false;
```

```
250     arr->threads_per_block = 160; //limited by the GPU. FIXME: Should be more
            dynamic
251     size_t pitch;
252     size_t byteWidth = arr->threads_per_block*sizeof(float);
253     size_t height = spatial_size*spectral_size/arr->threads_per_block;
254 
255     gpuallocate_prop(spectral_size, spatial_size, byteWidth, height, &pitch, &(
            arr->muae), &(arr->muse), &(arr->muad), &(arr->musd));
256     gpuallocate_iter(spectral_size, spatial_size, byteWidth, height, &pitch, &(
            arr->refl), &(arr->deriv), &(arr->prev_muad), &(arr->lineData_gpu_1), &(
            arr->lineData_gpu_2));
257     gpuallocate_spect(wavelengths, spectral_size, spatial_size, byteWidth,
            height, &pitch, &(arr->muh_oxy), &(arr->muh_deoxy), &(arr->melanin_base)
            , &(arr->gcol), &(arr->musm), &(arr->musr), &(arr->musb_base));
258 
259     arr->pitch = pitch;
260     arr->byteWidth = byteWidth;
261     arr->height = height;
262     arr->spectral_size = spectral_size;
263     arr->spatial_size = spatial_size;
264 
265     //calibration array
266     cerr << cudaMallocPitch(&(arr->cal_array), &pitch, byteWidth, height);
267     cerr << cudaMemcpy2D(arr->cal_array, pitch, cal_array, byteWidth, byteWidth
            , height, cudaMemcpyHostToDevice);
268 
269     //oxy, bvf, muam694
270     cerr << cudaMallocPitch(&(arr->oxy), &pitch, byteWidth, height);
271     cerr << cudaMallocPitch(&(arr->bvf), &pitch, byteWidth, height);
272     cerr << cudaMallocPitch(&(arr->muam694), &pitch, byteWidth, height);
273 }
274 
275 void gpuallocate_prop(int spectral_size, int spatial_size, size_t byteWidth,
        size_t height, size_t *pitch, float **muae, float **muse, float **muad,
        float **musd){
276     //optical properties
277     cerr << cudaMallocPitch(muae, pitch, byteWidth, height);
278     cerr << cudaMallocPitch(muse, pitch, byteWidth, height);
279     cerr << cudaMallocPitch(muad, pitch, byteWidth, height);
280     cerr << cudaMallocPitch(musd, pitch, byteWidth, height);
281 }
282 void gpuallocate_iter(int spectral_size, int spatial_size, size_t byteWidth,
        size_t height, size_t *pitch, float **refl, float **deriv, float **prev_muad
        , float **lineData_1, float **lineData_2){
283     //iteration variables
284     cerr << cudaMallocPitch(refl, pitch, byteWidth, height);
285     cerr << cudaMallocPitch(deriv, pitch, byteWidth, height);
286     cerr << cudaMallocPitch(prev_muad, pitch, byteWidth, height);
287     cerr << cudaMallocPitch(lineData_1, pitch, byteWidth, height) << endl;
288     cerr << cudaMallocPitch(lineData_2, pitch, byteWidth, height) << endl;
289 }
290 void gpuallocate_spect(float *wavelengths, int spectral_size, int spatial_size,
         size_t byteWidth, size_t height, size_t *pitch, float **muh_oxy, float **
        muh_deoxy, float **melanin_base, float **gcol, float **musm, float **musr,
        float **musb_base){
291     size_t arrayByteSize = sizeof(float)*spatial_size*spectral_size;
292     //wavelength dependencies of scattering and absorption coefficients
293     cerr << cudaMallocPitch(muh_oxy, pitch, byteWidth, height);
294     cerr << cudaMallocPitch(muh_deoxy, pitch, byteWidth, height);
295     cerr << cudaMallocPitch(melanin_base, pitch, byteWidth, height);
296     cerr << cudaMallocPitch(musm, pitch, byteWidth, height);
```

```
297     cerr << cudaMallocPitch(musr, pitch, byteWidth, height);
298     cerr << cudaMallocPitch(musb_base, pitch, byteWidth, height);
299     cerr << cudaMallocPitch(gcol, pitch, byteWidth, height);
300 
301     //calculate the spectrum arrays on the host, try to transfer concurrently
            by swapping between initiating the cudaMemCpy and doing calculations on
            the CPUi
302     float *muh_oxy_host = (float*)malloc(arrayByteSize);
303     float *muh_deoxy_host = (float*)malloc(arrayByteSize);
304     float *melanin_base_host = (float*)malloc(arrayByteSize);
305     float *gcol_host = (float*)malloc(arrayByteSize);
306     float *musm_host = (float*)malloc(arrayByteSize);
307     float *musr_host = (float*)malloc(arrayByteSize);
308     float *musb_base_host = (float*)malloc(arrayByteSize);
309 
310     //prepare the wavelength dependent parts of the absorption and scattering
            coefficients
311     //it might seem inefficient to assign the exact same value to the whole
            range of spatial positions for each wavelength, but even if it is all a
            memory waste, it will pay off in speed later on because of memory
            coalescing in the GPU. Also tried to load it into shared memory across
            all the thread blocks, but had to make the threads diverge in order to
            do that and would not work well with multiple shared variables
312     //Would be the exact same latency issues. whether 32 threads load 32
            different variables in one go and in parallell, or one of the thread
            loads the whole thing into the cache and picks one variable and
            broadcasts it to the rest will give the exact same latency, plus it will
             have to stall threads.
313     float lambda;
314     int position;
315     float muh_oxy_temp, muh_deoxy_temp, gcol_temp, musm_temp, musr_temp,
            musb_temp, melanin_temp;
316     for (int i=0; i < spectral_size; i++){
317         lambda = wavelengths[i];
318         muh_oxy_temp = muh_oxy_calc(lambda);
319         muh_deoxy_temp = muh_deoxy_calc(lambda);
320         melanin_temp = pow((694/lambda),3.46);
321         gcol_temp = 0.62 + lambda*29e-5;
322         musm_temp = (1-1.745e-3*lambda + 9.843e-7*lambda*lambda)/(1-gcol_host[i
                ]);
323         musr_temp = pow(lambda, -4);
324         musb_temp = pow((685/(lambda*1.0)), 0.37);
325         for (int j=0; j < spatial_size; j++){
326             position = i*spatial_size + j;
327             muh_oxy_host[position] = muh_oxy_temp;
328             muh_deoxy_host[position] = muh_deoxy_temp;
329             melanin_base_host[position] = melanin_temp;
330             gcol_host[position] = gcol_temp;
331             musm_host[position] = musm_temp;
332             musr_host[position] = musr_temp;
333             musb_base_host[position] = musb_temp;
334         }
335     }
336 
337     //copy the wavelength dependencies to the GPU device
338     cerr << cudaMemcpy2D(*muh_oxy, *pitch, muh_oxy_host, byteWidth, byteWidth,
            height, cudaMemcpyHostToDevice);
339     cerr << cudaMemcpy2D(*muh_deoxy, *pitch, muh_deoxy_host, byteWidth,
            byteWidth, height, cudaMemcpyHostToDevice);
340     cerr << cudaMemcpy2D(*melanin_base, *pitch, melanin_base_host, byteWidth,
            byteWidth, height, cudaMemcpyHostToDevice);
```

```
    cerr << cudaMemcpy2D(*musm, *pitch, musm_host, byteWidth, byteWidth, height
        , cudaMemcpyHostToDevice);
    cerr << cudaMemcpy2D(*musr, *pitch, musr_host, byteWidth, byteWidth, height
        , cudaMemcpyHostToDevice);
    cerr << cudaMemcpy2D(*musb_base, *pitch, musb_base_host, byteWidth,
        byteWidth, height, cudaMemcpyHostToDevice);
    cerr << cudaMemcpy2D(*gcol, *pitch, gcol_host, byteWidth, byteWidth, height
        , cudaMemcpyHostToDevice);

    //free arrays that are no longer needed
    free(muh_oxy_host);
    free(muh_deoxy_host);
    free(melanin_base_host);
    free(musm_host);
    free(musr_host);
    free(musb_base_host);
    free(gcol_host);
}

//free the GPU arrays
void gpufree(GPUDMArrays *a){
    cerr << cudaFree(a->muae);
    cerr << cudaFree(a->muse);
    cerr << cudaFree(a->muad);
    cerr << cudaFree(a->musd);
    cerr << cudaFree(a->refl);
    cerr << cudaFree(a->prev_muad);
    cerr << cudaFree(a->lineData_gpu_1);
    cerr << cudaFree(a->lineData_gpu_2);
    cerr << cudaFree(a->muh_oxy);
    cerr << cudaFree(a->muh_deoxy);
    cerr << cudaFree(a->melanin_base);
    cerr << cudaFree(a->gcol);
    cerr << cudaFree(a->musm);
    cerr << cudaFree(a->musr);
    cerr << cudaFree(a->musb_base);
}
```

File A.3: pseudo–framework.c

```
#include "gpudm.h"
#include "neccessary_libraries_for_the_pseudocode_to_work.h"

//global variables
RingBuffer buffer; //circular buffer into which data is input and extracted
CameraHeader header; //camera header
Buffer muads; //buffer for the dermal absorption coefficients

Semaphore headerWait; //signalling whether the header has arrived
Semaphore calWait; //signalling whether the calibration slab has arrived
Semaphore lineWait; //signalling a line is ready

int main(){
    //initialize both semaphores to 0
    sem_init(headerWait, 0);
    sem_init(calWait, 0);
    sem_init(lineWait, 0);

    //initialize threads
    init_thread(client_thread);
    init_thread(inversion_thread);
```

```
22
23      wait_for(client_thread);
24      wait_for(processing_thread);
25
26      return 0;
27  }
28
29  void* client_thread(){
30      TcpConnection conn; //representing the TCP connection
31
32      uint16 *data;
33      receive_header_from_server(&header);
34      sem_post(headerWait); //post the semaphore
35
36      int i=0;
37      int calStop=20; //assume the first 21 lines contain the calibration slab
38
39      while (!conn.timeout()){
40          if (i == calStop){
41              sem_post(calWait); //post the semaphore
42          }
43
44          //receive data, push into buffer
45          receive_data_from_server(data);
46          buffer.push(data);
47
48          if (i > calStop){
49              sem_post(lineWait);
50          }
51      }
52  }
53
54
55  void* inversion_thread(){
56      sem_wait(headerWait); //wait for the header to become ready
57      size_t arraySize = header.spectral_size*header.spatial_size*sizeof(float);
58
59      sem_wait(calWait); //wait for the calibration slab to have fully entered
            the picture
60      float *cal = malloc(arraySize);
61      float *lineData = malloc(arraySize);
62      float *prevMuad = malloc(arraySize);
63
64      extractCalArray(buffer, cal);
65
66      GPUDMArrays a;
67      gpu_allocate(&a, header.wavelengths, cal, header.spectral_size, header.
            spatial_size);
68
69      while(true){
70          sem_wait(lineWait);
71          buffer.extractInto(lineData);
72          gpuprocessing(a, lineData, prevMuad);
73          muads.push(prevMuad);
74      }
75  }
```

A.2 GPU-MCML

While GPU-DM generally is structured, the extra functions around the GPU-MCML program for inverting the melanin contents and so is not. It was not really meant for a production situation. It still is included. A source file containing the most important modifications to GPU-MCML is also included. Some other functions were changed somewhat to save the output diffuse reflectance in an input float value instead of to file, these are not included in the listing.

File A.4: GPU–MCML/fast–gpumcml/main.cpp

```
1  #include "gpumcml.h"
2  #include <iostream>
3  #include <fstream>
4  #include <string>
5  #include <vector>
6  #include <string.h>
7  #include <cmath>
8  #include <sstream>
9  #include "opticalprop.h"
10 #include "spectrum.h"
11 #include "erythemalookup.h" //containing very ugly lookup table
12
13 //GNU scientific libraries
14 #include <gsl/gsl_multifit.h>
15 #include <gsl/gsl_vector.h>
16 #include <gsl/gsl_matrix.h>
17 using namespace std;
18
19
20
21 int findIndexMatch(vector<double> wlens, double x){
22     int index = 0;
23     //just linear search, could be optimized with binary search
24     for (unsigned int i=0; i < wlens.size(); i++){
25         if (wlens[i] > x){
26             index = i-1;
27             break;
28         }
29     }
30     if (index < 0){
31         index = 0;
32     }
33     return index;
34 }
35
36 void melaninabs_seqiter(double w1, double w2, double w3, double w4, double v1,
       double v2, double v3, double v4){
37     GPURunData data;
38     initialize_gpumcml(&data);
39
40     double bvp = 0.03;
41     double muam694 = 100;
42     double oxy = 0.5;
43     SkinStruct skinData;
44     calcSkinData(w1, oxy, bvp, muam694, &skinData);
45
46     double simRefl = run_gpumcml(data, skinData);
47     double margin = 0.001;
48
49
50     bool above = simRefl > v1;
```

```
51
52      double step;
53
54      int numIts = 10;
55      int i=0;
56      double ekstrabs = 0;
57
58
59      int j = 0;
60
61      double difference = margin;
62      while(true){
63          step = 100;
64          i = 0;
65          //med gitt bvp, forsoek tilpasning med muam paa punkt 1
66          calcSkinData(w1, oxy, bvp, muam694, &skinData);
67          skinData.muad += ekstrabs;
68
69          simRefl = run_gpumcml(data, skinData);
70          above = simRefl > v1;
71          cout << "melaninit" << endl;
72          while (i < 5){
73              if ((simRefl) <= v1){
74                  muam694 -= step;
75                  if (above){
76                      step *= 0.5;
77                  }
78              } else {
79                  muam694 += step;
80                  if (!above){
81                      step *= 0.5;
82                  }
83              }
84              cout <<  muam694 << "␣" << bvp << "␣" << simRefl << "␣" << v1 <<
                    endl;
85              calcSkinData(w1, oxy, bvp, muam694, &skinData);
86              skinData.muad += ekstrabs;
87              simRefl = run_gpumcml(data, skinData);
88
89              i++;
90          }
91
92          calcSkinData(w2, oxy, bvp, muam694, &skinData);
93          skinData.muad += ekstrabs;
94
95          simRefl = run_gpumcml(data, skinData);
96          bool above1 = simRefl > v2;
97          bool above2 = simRefl > v3;
98
99
100         step = 0.01;
101         i = 0;
102         //med nyutregnet muam, proev tilpasning paa bvp paa neste punkt
103         cout << "blodit" << endl;
104         while (i < 5){
105             //tilpass paa det foerste punktet
106             if ((simRefl) <= v2){
107                 bvp -= step;
108                 if (above1){
109                     step *= 0.5;
110                 }
```

```
111             } else {
112                 bvp += step;
113                 if (!above1){
114                     step *= 0.5;
115                 }
116             }
117             if (bvp < 0){
118                 bvp = 0.015;
119             }
120             cout  << muam694 << "␣" << bvp << "␣" << simRefl << "␣" << v2 <<
                    endl;
121
122             calcSkinData(w3, oxy, bvp, muam694, &skinData);
123             skinData.muad += ekstrabs;
124             simRefl = run_gpumcml(data, skinData);
125
126
127             //tilpass paa det andre punktet
128             if ((simRefl) <= v3){
129                 bvp -= step;
130                 if (above2){
131                     step *= 0.5;
132                 }
133             } else {
134                 bvp += step;
135                 if (!above2){
136                     step *= 0.5;
137                 }
138             }
139             cout  << muam694 << "␣" << bvp << "␣" << simRefl << "␣" << v3 <<
                    endl;
140
141             calcSkinData(w2, oxy, bvp, muam694, &skinData);
142             skinData.muad += ekstrabs;
143             simRefl = run_gpumcml(data, skinData);
144
145             i++;
146             if (bvp < 0){
147                 bvp = 0.015;
148             }
149         }
150
151         step = 10;
152         i = 0;
153         j++;
154
155     }
156 }
157
158 //a*x + b
159 void linearFit(Spectrum spect, double lowLen, double hiLen, double *a, double *
        b){
160     double xyb = 0;
161     double x2b = 0;
162     double xb = 0;
163     double yb = 0;
164     vector<double> wlens = spect.getWlens();
165     vector<double> vals = spect.getVals();
166
167     unsigned int i = 0;
168     int n = 0;
```

```
169     //fit a straight line through the available points from 630 to 700 nm
170     while ((wlens[i] <= hiLen) && (i < wlens.size())){
171         if (wlens[i] >= lowLen){
172             xyb += wlens[i]*vals[i];
173             xb += wlens[i];
174             yb += vals[i];
175             x2b += wlens[i]*wlens[i];
176             n++;
177         }
178         i++;
179     }
180 
181     xyb /= n;
182     xb /= n;
183     yb /= n;
184     x2b /= n;
185     *a = (xyb - xb*yb)/(x2b - xb*xb);
186     *b = yb - *a*xb;
187 }
188 
189 void kolliasHem(Spectrum logSpect, double mela, double melb, double *oxy,
        double *deoxy){
190     //extract the wavelengths and values in the desired area
191     double lam1 = 560;
192     double lam2 = 580;
193     vector<double> tempWlens = logSpect.getWlens();
194     vector<double> tempVals = logSpect.getVals();
195     vector<double> wlens;
196     vector<double> vals;
197     for (unsigned int i=0; i < tempWlens.size(); i++){
198         if ((tempWlens[i] >= lam1) && (tempWlens[i] <= lam2)){
199             wlens.push_back(tempWlens[i]);
200             vals.push_back(tempVals[i] - mela*tempWlens[i] - melb);
201         }
202     }
203 
204     //preparing the matrix and vectors in the fitting y = M*c
205     int param = 2;
206     int obs = wlens.size();
207     gsl_vector *y = gsl_vector_alloc(obs);
208     gsl_vector *c = gsl_vector_alloc(param);
209     gsl_matrix *mat = gsl_matrix_alloc(obs, param);
210     gsl_matrix *cov = gsl_matrix_alloc(param, param);
211     for (int i = 0; i < obs; i++){
212         gsl_matrix_set(mat, i, 0, muh_oxy(wlens[i]));
213         gsl_matrix_set(mat, i, 1, muh_deoxy(wlens[i]));
214         for (int j = 2; j < param; j++){
215             gsl_matrix_set(mat, i, j, pow(wlens[i], j-2));
216         }
217         gsl_vector_set(y, i, vals[i]);
218     }
219 
220     //setting up the workspace for multiparameter fitting
221     gsl_multifit_linear_workspace *workspace = gsl_multifit_linear_alloc(obs,
            param);
222 
223     //performing the fitting itself
224     double chisq = 0;
225     gsl_multifit_linear(mat, y, c, cov, &chisq, workspace);
226 
227     *oxy = gsl_vector_get(c,0);
```

```
228     *deoxy = gsl_vector_get(c,1);
229 
230     gsl_multifit_linear_free(workspace);
231 
232     gsl_matrix_free(mat);
233     gsl_vector_free(y);
234     gsl_vector_free(c);
235 
236 }
237 
238 //forbedret kolliasmetode
239 void kollias(Spectrum spect, double *mel, double *oxy, double *deoxy){
240     //logarithmitize
241     vector<double> wlens = spect.getWlens();
242     vector<double> vals = spect.getVals();
243     for (unsigned int i=0; i < wlens.size(); i++){
244         vals[i] = -log10(vals[i]);
245         //cout << wlens[i] << " " << vals[i] << endl;
246     }
247     Spectrum logSpect(wlens, vals);
248 
249 
250     //get linear fit from 630 to 700 nm
251     double b;
252     linearFit(logSpect, 630, 700, mel, &b);
253 
254     double prevMel = 200000;
255     double margin = 10e-12;
256     while(abs(*mel - prevMel) >= margin){
257         prevMel = *mel;
258         kolliasHem(logSpect, *mel, b, oxy, deoxy);
259 
260         //correct the spectrum for deoxy
261         vector<double> corrVals = vals;
262         for (unsigned int i=0; i < corrVals.size(); i++){
263             corrVals[i] -= *deoxy*muh_deoxy(wlens[i]);
264             //cout << wlens[i] << " " << corrVals[i] << endl;
265         }
266         Spectrum corrLogSpect(wlens, corrVals);
267         linearFit(corrLogSpect,630,700, mel, &b);
268     }
269 }
270 
271 //simuler for parametrene og returner kolliasindeksene
272 void simKollias(double oxy, double bvf, double muam694, double *kolMel, double
        *kolOxy, double *kolDeoxy){
273     GPURunData data;
274     initialize_gpumcml(&data);
275     //finn indeksene vi skal oppnaa
276     vector<double> wlens;
277     //kameltopp
278     wlens.push_back(560);
279     wlens.push_back(580);
280 
281     //melaninomraade
282     wlens.push_back(630);
283     wlens.push_back(700);
284 
285     vector<double> vals;
286     SkinStruct skinData;
287     for (unsigned int i=0; i < wlens.size(); i++){
```

```
288         calcSkinData(wlens[i], oxy, bvf, muam694, &skinData);
289         vals.push_back(run_gpumcml(data, skinData));
290     }
291     Spectrum spect(wlens, vals);
292     kollias(spect, kolMel, kolOxy, kolDeoxy);
293 }
294
295 void kolliasIterationTest(Spectrum spect, double *outMuam694, double *outBvf,
        double *outOxy, bool willPrint){
296     double oxyInd;
297     double deoxyInd;
298     double melInd;
299     double oxy = 0.5;
300     //simKollias(oxy, 0.02, 250, &melInd, &oxyInd, &deoxyInd);
301     kollias(spect, &melInd, &oxyInd, &deoxyInd);
302     double eryInd = oxyInd + deoxyInd;
303
304     double bvf = 0.01;
305     double muam694 = -9.8548e+05*melInd-193.87; //from generated lookup table
306     double bvfMax = exp((eryInd - 3.6231e-05)/5.5528e-06);
307     double bvfMin = exp((eryInd - 2.7763e-05)/4.0e-06);
308     bvf = (bvfMax + bvfMin)/2;
309     oxy = oxyInd/(oxyInd + deoxyInd);
310     cout << bvf << "␣" << muam694 << "␣" << oxy<< endl;
311     double currMel;
312     double currOxy;
313     double currDeoxy;
314     double currEry;
315     double margin;
316     int i;
317     bool above;
318     int numIts = 10;
319     int melConvs = 0;
320     int bvfConvs = 0;
321     int convs = 2;
322     while((melConvs <= convs) && (bvfConvs <= convs)){
323         simKollias(oxy, bvf, muam694, &currMel, &currOxy, &currDeoxy);
324         double step = 50;
325         i = 0;
326         //med gitt bvp, forsoek tilpasning med muam pa punkt 1
327         margin = 1.0e-6;
328         above = currMel > melInd;
329         bool isConverging = false;
330         while ((abs(currMel - melInd) >= margin) && (i < numIts)){
331             if (abs(currMel) >= abs(melInd)){
332                 muam694 -= step;
333                 if (above){
334                     step *= 0.5;
335                     isConverging = true;
336                 }
337             } else {
338                 muam694 += step;
339                 if (!above){
340                     step *= 0.5;
341                     isConverging = true;
342                 }
343             }
344             if (willPrint){
345                 cout <<  muam694 << "␣" << bvf << endl;
346             }
347             simKollias(oxy, bvf, muam694, &currMel, &currOxy, &currDeoxy);
```

```
348             i++;
349         }
350         if (isConverging){
351             melConvs++;
352         }
353         simKollias(oxy, bvf, muam694, &currMel, &currOxy, &currDeoxy);
354         currEry = currOxy + currDeoxy;
355 
356         above = currEry > eryInd;
357 
358         step = 0.01;
359         margin = 1.0e-8;
360         i = 0;
361         isConverging = false;
362         //med nyutregnet muam, proev tilpasning paa bvp paa neste punkt
363         while ((abs(currEry - eryInd) >= margin) && (i < numIts)){
364             if (currEry >= eryInd){
365                 bvf -= step;
366                 if (!above){
367                     step *= 0.5;
368                     isConverging = true;
369                 }
370             } else {
371                 bvf += step;
372                 if (above){
373                     step *= 0.5;
374                     isConverging = true;
375                 }
376             }
377             if (willPrint){
378                 cout  << muam694 << "␣" << bvf << endl;
379             }
380             simKollias(oxy, bvf, muam694, &currMel, &currOxy, &currDeoxy);
381             currEry = currOxy + currDeoxy;
382             i++;
383             if (bvf < 0){
384                 bvf = 0.015;
385             }
386             i++;
387         }
388         if (isConverging){
389             bvfConvs++;
390         }
391     }
392     *outBvf = bvf;
393     *outMuam694 = muam694;
394     *outOxy = oxy;
395 }
396 
397 //get out the full absorption spectrum, matching MC reflectance against
        measured reflectance. muam694, oxy assumed known. Assumed that bvp is
        approximately correct.
398 void getAbsorptionSpectrum(Spectrum spect, double oxy, double bvp, double
        muam694, bool willPrint, Spectrum *output, vector<double> *weights){
399     vector<double> absWlens;
400     vector<double> absVals;
401 
402     vector<double> wlens = spect.getWlens();
403     vector<double> refl = spect.getVals();
404     int n = wlens.size();
405     int indexStep = (int)n/160; //will match against 160 wavelengths
```

```
406     double margin = 0.0005;
407     double prevMuad = 0;
408     GPURunData data;
409     initialize_gpumcml(&data);
410 
411     //go through the chosen wavelengths
412     for (int i=0; i < n; i += indexStep){
413         int startClock = clock();
414         //set up absorption properties for epidermis and scattering properties
                for dermis, also approximate absorption properties for dermis
415         SkinStruct skinSetup;
416         calcSkinData(wlens[i], oxy, bvp, muam694, &skinSetup);
417         double startMuad = skinSetup.muad;
418 
419         skinSetup.muad = 0.5*(skinSetup.muad + prevMuad);
420         double step = 10;
421         double currRefl = run_gpumcml(data, skinSetup);
422 
423         //for deciding whether we initially started above or below the desired
                value
424         bool above = currRefl > refl[i];
425         bool below = !above;
426 
427         //iterate
428         while (abs(currRefl - refl[i]) >= margin){
429             if (currRefl > refl[i]){
430                 skinSetup.muad += step;
431                 if (below){
432                     step *= 0.5;
433                 }
434             } else {
435                 skinSetup.muad -= step;
436                 if (above){
437                     step *= 0.5;
438                 }
439             }
440             currRefl = run_gpumcml(data, skinSetup);
441         }
442         prevMuad = skinSetup.muad;
443         if (willPrint){
444             cout << wlens[i] << "␣" << currRefl << "␣" << skinSetup.muad << "␣"
                     << startMuad<< endl;
445         }
446         absWlens.push_back(wlens[i]);
447         absVals.push_back(skinSetup.muad);
448         weights->push_back(abs((double)clock()-startClock));
449     }
450     output->setSpect(absWlens, absVals);
451     deinitialize_gpumcml(&data);
452 }
453 
454 //do MCA in the absorption spectrum. Input: spekteret, vektor av innlastede
        kromoforer
455 void mca(Spectrum spect, vector<Spectrum> cromophores, vector<double> *
        concentrations){
456     int obs = spect.getSize(); //number of observations
457     double w = 0; //wavelength of the current observation
458     int n_cro = cromophores.size();
459     int param = n_cro+1;
460 
461     gsl_vector *abs = gsl_vector_alloc(obs); //the absorption spectrum to be
```

```
        matched
    gsl_vector *c = gsl_vector_alloc(param); //the concentrations of each
        chromophore to be determined
    gsl_matrix *cromophore_mat = gsl_matrix_alloc(obs, param); //the matrix of
        chromophores
    gsl_matrix *cov = gsl_matrix_alloc(param, param); //covariance matrix

    //set the cromophore matrix and observation matrix
    for (int i = 0; i < obs; i++){
        w = spect.wlenAt(i);
        gsl_vector_set(abs, i, spect.valAt(i));
        for (int j = 0; j < n_cro; j++){
            gsl_matrix_set(cromophore_mat, i, j, cromophores[j].val(w));
        }
        gsl_matrix_set(cromophore_mat, i, n_cro, 1);
    }

    //setting up the workspace for multiparameter fitting
    gsl_multifit_linear_workspace *workspace = gsl_multifit_linear_alloc(obs,
        param);

    //performing the fitting itself
    double chisq = 0;
    //gsl_multifit_wlinear(cromophore_mat, wei_vec, abs, c, cov, &chisq,
        workspace);
    gsl_multifit_linear(cromophore_mat, abs, c, cov, &chisq, workspace);

    for (int i=0; i < param; i++){
        concentrations->push_back(gsl_vector_get(c,i));
    }

    //free memory
    gsl_multifit_linear_free(workspace);
    gsl_matrix_free(cromophore_mat);
    gsl_vector_free(abs);
    gsl_vector_free(c);
}
```

File A.5: GPU–MCML/fast–gpumcml/opticalprop.c

```
//for generating optical properties
#include "math.h"
#include "opticalprop.h"
#include "muabo.h"

double muabEff(double r, double muab){
    return 1/(2*r)*(1-exp(-2*r*muab));
}

//translated directly from that matlab script
void calcSkinData(float lambda, float oxy, float Bd, float muam694, SkinStruct
    *output){
    output->nphotons = 300000;
    //double oxy = 0.8;
    float H = 0.41;
    float H0 = 0.45;
    float Be = 0.002;
    //double Bd = 0.01;

    output->ne = 1.4;
    output->nd = 1.4;
```

```
21
22      output->de = 100e-6;
23      output->dd = 1;
24      double r1, r2, r3, r4;
25      r1 = 10e-6;
26      r2 = 25e-6;
27      r3 = 50e-6;
28      r4 = 100e-6;
29
30
31      //absorption coefficients
32      //double muam694 = 500;   //melanin absorption coeff. at 694nm, in [m-1]
33      float mua_other = 25;  //non-blood, non-melanin absorption
34
35      float muab_blood = (muh_oxy(lambda)*oxy+muh_deoxy(lambda)*(1-oxy))*H/H0;
36      //muab_blood = 0.7*muabEff(r1, muab_blood) + 0.24*muabEff(r2, muab_blood) +
            0.04*muabEff(r3, muab_blood) + 0.02*muabEff(r4, muab_blood);
37      float muab_melanin = muam694*pow((694/lambda),3.46);
38
39      output->muae = (muab_melanin + muab_blood*Be + mua_other*(1-Be));
40      output->muad = (muab_blood*Bd + mua_other*(1-Bd));
41
42
43      //scattering coefficients
44      ScattCoeffStruct scattering;
45      calcScattCoeff(lambda, H, Be, Bd, &scattering);
46
47      output->muse = scattering.muse;
48      output->musd = scattering.musd;
49      output->ge = scattering.ge;
50      output->gd = scattering.gd;
51  }
52
53  void calcScattCoeff(float lambda, float H, float Be, float Bd, ScattCoeffStruct
        *output){
54      //av. cosine of scattering angle in tissue:
55      float gcol = 0.62 + lambda*29e-5;
56      float gery = 0.9969;
57
58      // Collagen:
59      //Mie scattering: (reduced)
60      float c_mie = 105; //voksen person, fra Winnem2004
61      float musmr = c_mie*(1 - 1.745e-3*lambda + 9.843e-7*lambda*lambda);
62
63      //Rayleigh scattering: (NOT reduced)
64      float c_ray = 1.05e12; //voksen person, fra Winnem2004
65      float musrr = c_ray*pow(lambda,-4);
66
67      //total scattering coeff. in collagen:
68      float must = (musmr + musrr)*100/(1-gcol);
69
70      //total scattering coeff. in tissue:
71      //double must = mustr/(1-g); //deler paa 1-g for aa gaa fra reduced til
            ikke reduced
72
73      // Erythrocytes:
74      //scatt. coeff. at 685nm:
75      float musb685 = 55.09e-12;
76      //Erythrocyte volume:
77      float ve = 1.25e-16;
78      //scatt. coeff.:
```

```
79      float musb = musb685*H*(1-H)*(1.4-H)/ve*pow((685/(lambda*1.0)),0.37); //
            FIXME: ok det skal mangle en gb fordi dette er jo for monte carlo og
            ikke diff
80
81      //total scattering coefficient
82      output->muse = must*(1-Be) + musb*Be;
83      output->musd = must*(1-Bd) + musb*Bd;
84
85
86      //probability that scattering was caused by collagen, epidermis
87      //float pcole = must*(1-Be)/output->muse;
88
89      //probability that scattering was caused by collagen, dermis
90      //float pcold = must*(1-Bd)/output->musd;
91
92      //probaility for scattering from Mie
93      //float pmie = (musmr/(1-gcol))/((musmr/(1-gcol))+musrr);
94
95      //anisotropy g in dermis
96      output->ge = gcol;//*pcole*pmie + gery*(1-pcole);
97
98      //g in epidermis
99      output->gd = gcol;//*pcold*pmie + gery*(1-pcold);
100 }
101
102 float muh_oxy(float l){
103     int low_ind = (int)(l-400);
104     int upp_ind = low_ind+1;
105     float lower_l = (int)l;
106     float upper_l = lower_l+1;
107     return (muabo[low_ind] + (l - lower_l)/(upper_l - lower_l)*(muabo[upp_ind]
            - muabo[low_ind]));
108 }
109
110 float muh_deoxy(float l){
111     int low_ind = (int)(l-400);
112     int upp_ind = low_ind+1;
113     float lower_l = (int)l;
114     float upper_l = lower_l+1;
115     return (muabd[low_ind] + (l - lower_l)/(upper_l - lower_l)*(muabd[upp_ind]
            - muabd[low_ind]));
116 }
```

File A.6: GPU–MCML/fast–gpumcml/opticalprop.h

```
1  #ifndef OPTICALPROP_H_DEFINED
2  #define OPTICALPROP_H_DEFINED
3
4  float ext_oxy(float l);
5  float ext_deoxy(float l);
6
7  float muh_oxy(float l);
8  float muh_deoxy(float l);
9
10
11
12 typedef struct{
13     float muse; //muh_s in epidermis
14     float musd; //muh_s in dermis
15     float ge; //anisotropy factor
16     float gd;
```

```
} ScattCoeffStruct;


typedef struct{
    //general
    unsigned long nphotons;

    //epidermis
    float ge;
    float muse;
    float muae;
    float ne;
    float de;

    //dermis
    float gd;
    float musd;
    float muad;
    float nd;
    float dd;
} SkinStruct;

void calcSkinData(float lambda, float oxy, float Bd, float muam694, SkinStruct
    *output);

void calcScattCoeff(float lambda, float H, float Be, float Bd, ScattCoeffStruct
     *output);

#endif
```

File A.7: GPU–MCML/fast–gpumcml/gpumcmlchanges.c

```

int set_simulation_data(SimulationStruct** simulations, SkinStruct skinData)
{
  int i=0;
  int ii=0;
  int n_layers = 2;
  unsigned long number_of_photons = skinData.nphotons;
  float dtot=0;
  double n1, n2, r;

  // Allocate memory for the SimulationStruct array
  *simulations = (SimulationStruct*) malloc(sizeof(SimulationStruct)*1);
  if(*simulations == NULL){perror("Failed to malloc simulations.\n");return 0;}

    //set the program to ignore detection of absorption, will be a performance
        bottleneck if we do
    (*simulations)[i].ignoreAdetection=1;
    (*simulations)[i].AorB='A';

    //set the number of photons
    (*simulations)[i].number_of_photons=number_of_photons;

    //set grid to exact one element, we don't need it since no absorption is
        saved
    (*simulations)[i].det.dz=skinData.dd*1.11; //litt stoerre enn tykkelsen
    (*simulations)[i].det.dr=0.01;
    (*simulations)[i].det.nz=1;
    (*simulations)[i].det.nr=100;
    (*simulations)[i].det.na=50;
```

```
28
29      //number of layers
30      (*simulations)[i].n_layers = n_layers;
31
32      // Allocate memory for the layers (including one for the upper and one for
            the lower)
33      (*simulations)[i].layers = (LayerStruct*) malloc(sizeof(LayerStruct)*(
            n_layers+2));
34      if((*simulations)[i].layers == NULL){perror("Failed to malloc layers.\n");
            return 0;}
35
36      //ser upper refractive index
37      (*simulations)[i].layers[0].n=1.0;
38
39      //set layer parameters
40      //epidermis
41      dtot=0;
42
43      ii=1;
44        (*simulations)[i].layers[ii].n=skinData.ne;
45        (*simulations)[i].layers[ii].mua=skinData.muae/100.0;
46
47
48        (*simulations)[i].layers[ii].g=skinData.ge;
49        (*simulations)[i].layers[ii].z_min=dtot;
50        dtot+=skinData.de*100.0;
51        (*simulations)[i].layers[ii].z_max=dtot;
52        (*simulations)[i].layers[ii].mutr=1.0f/(skinData.muae/100.0 + skinData.
              muse/100.0);
53
54      //dermis
55          ii++;
56        (*simulations)[i].layers[ii].n=skinData.nd;
57        (*simulations)[i].layers[ii].mua=skinData.muad/100.0;
58
59
60        (*simulations)[i].layers[ii].g=skinData.gd;
61        (*simulations)[i].layers[ii].z_min=dtot;
62        dtot+=skinData.dd*100;
63        (*simulations)[i].layers[ii].z_max=dtot;
64        (*simulations)[i].layers[ii].mutr=1.0f/(skinData.muad/100.0 + skinData.
              musd/100.0);
65
66
67
68      // Read lower refractive index (1xfloat)
69      (*simulations)[i].layers[n_layers+1].n=1;
70
71      //calculate start_weight
72      n1=(*simulations)[i].layers[0].n;
73      n2=(*simulations)[i].layers[1].n;
74      r = (n1-n2)/(n1+n2);
75      r = r*r;
76      (*simulations)[i].start_weight = 1.0F - (float)r;
77
78    return 1;
79  }
80
81  //do initialization of the gpu
82  void initialize_gpumcml(GPURunData *data){
83    int i;
```

```
  UINT64 seed = (UINT64) time(NULL);
  data->num_GPUs = 1; //assume only one GPU
  // Determine the number of GPUs available.
  int dev_count;
  CUDA_SAFE_CALL( cudaGetDeviceCount(&dev_count) );
  if (dev_count <= 0)
  {
    fprintf(stderr, "No GPU available. Quit.\n");
    exit(1);
  }

  cudaDeviceProp props;
  int n_threads = 0;    // total number of threads for all GPUs
  for (i = 0; i < data->num_GPUs; ++i)
  {
    data->hstates[i] = (HostThreadState*)malloc(sizeof(HostThreadState));

    // Set the GPU ID.
    data->hstates[i]->dev_id = i;

    // Get the GPU properties.
    CUDA_SAFE_CALL( cudaGetDeviceProperties(&props, data->hstates[i]->dev_id) );

    // Validate the GPU compute capability.
    int cc = (props.major * 10 + props.minor) * 10;
    if (cc < __CUDA_ARCH__)
    {
      fprintf(stderr, "\nGPU %u does not meet the Compute Capability "
          "this program requires (%d)! Abort.\n\n", i, __CUDA_ARCH__);
      exit(1);
    }

    // We launch one thread block for each SM on this GPU.
    data->hstates[i]->n_tblks = props.multiProcessorCount;

    n_threads += data->hstates[i]->n_tblks * NUM_THREADS_PER_BLOCK;
  }
  // Allocate and initialize RNG seeds (for all threads on all GPUs).
  data->x = (UINT64*)malloc(n_threads * sizeof(UINT64));
  data->a = (UINT32*)malloc(n_threads * sizeof(UINT32));

#ifdef _WIN32
    if (init_RNG(data->x, data->a, n_threads, "safeprimes_base32.txt", seed))
        exit(1);
#else
    if (init_RNG(data->x, data->a, n_threads, "executable/safeprimes_base32.txt
        ", seed)) exit(1);
#endif
  // Assign these seeds to each host thread state.
  int ofst = 0;
  for (i = 0; i < data->num_GPUs; ++i)
  {
    SimState *hss = &(data->hstates[i]->host_sim_state);
    hss->x = &data->x[ofst];
    hss->a = &data->a[ofst];

    ofst += data->hstates[i]->n_tblks * NUM_THREADS_PER_BLOCK;
  }
}

```

```
142 //free memory structures
143 void deinitialize_gpumcml(GPURunData *data){
144   int i;
145   // Free host thread states.
146   for (i = 0; i < data->num_GPUs; ++i) free(data->hstates[i]);
147 
148   // Free the random number seed arrays.
149   free(data->x); free(data->a);
150 
151 }
152 
153 
154 double run_gpumcml(GPURunData gpuData, SkinStruct skinData)
155 //int main(int argc, char* argv[])
156 {
157   int willPrint = 0;
158   SimulationStruct* simulations;
159   int n_simulations;
160   int i;
161 
162   // Read the simulation inputs.
163   n_simulations = set_simulation_data(&simulations, skinData);
164 
165   //perform all the simulations
166   double diff_reflct;
167   double spec_reflct;
168   double transmitt;
169   for(i=0;i<n_simulations;i++)
170   {
171     // Run a simulation
172     DoOneSimulation(i, &simulations[i], gpuData.hstates, gpuData.num_GPUs,
          gpuData.x, gpuData.a, &diff_reflct, &transmitt, &spec_reflct, willPrint)
          ;
173   }
174   double retVal = diff_reflct;///(1-spec_reflct); //siden Rdiff bestaar av
        gamma*(1-Rsp), vil kun ha gamma for aa sammenligne direkte med
        diffusjonsteori
175 
176 
177   FreeSimulationStruct(simulations, n_simulations);
178   return retVal;
179 }
```

A.3 General MATLAB-implementation of the diffusion model

This is not included since it is not my work, although it was changed somewhat to keep better control over the reduced and unreduced scattering coefficients. Small scripts for calculating oxygenation and the like is not included.