Simulation of photon distribution in tissues

Xaver Kohlbrenner

Contents

1 Introduction

This paper is a documentation about my internship in the team Numerical Analysis and Uncertainty Quantification of the University Heidelberg.
The goal of this project is to simulate the light propagation in a multi-environment tissue with python.
Over the period of the project, I translated the programs in c "mc321.c" and "mc-sub" from www.omlc.org/software/mc/ into python and modified them to adjust to my requirements and faster execution. The final program simulates the distribution of a photon in a 3-dimensional space as measured by cylindrical coordinates.The space can contain different environments, which are defined by their radius and height. The classes and data structures are built in such a way that an increase in different variables, such as the number of environments or more accurate solutions, has a less severe influence on execution time.
While developing the program, I followed three design philosophies: everywhere executable, easy to expand, and reasonable running time:

- Everywhere executable: I did most of the development on my desktop PC, which has superior specs compared to a normal office laptop. Therefore, the program should not rely on a fast CPU and especially not on huge RAM.
- Easy to expand: The next developer should easily implement further improvements. To this end, I made the code very modular. Most of the classes could be used with small changes in different projects because the class and its functions stand alone.
- Reasonable running time: I aimed for a running time of around 60-90 seconds for a simple input. Especially, the condition of the environments can lead to more steps and, through this, to more calculations. This should be a balance between faster execution and flexible programming (see point 2). Python is, of course, slower than C, but I used some methods to reduce the running time, e.g., multiprocessing.

2 Framework conditions

The data for the framework is set in the input file. It includes the number of launched photons, predefined conditions for the beam, and which version is taken. The size of the tissue and number of bins, determining the sharpness of the output, are chosen along with the chances for the roulette. Also, the tissue is built by defining the variables of the environments and their place in the tissue with height and radius. One environment has to be declared as default. This is then used if a coordinate is assigned to no environment.
To test my program I chose the following parameters:
Photons: 100.000
Size of the tissue: 0.003 cm x 0.003 cm
Number of bins: 128
Weight Threshold: 0.0001
Roulette Chance: 10

3 General Usage

This section focus around the everyday use of the program. While the output is handled in a separated section, here the input and execution is explained.
In the input file $input_Data.py$ all predefined variables are set. They give full flexibility in designing the simulation to your own desires. All length measurements are in declared in centimeter.

- *photons*: defines the number of launched photons, in a Monte Carlo Simulation a higher number of events give a more precissive outcome of the distribution.
- $mcflag$: choose between the three different beams collimated uniform, Gaussian and isotropic point
- *radius*: radius of a beam
- *waist* and $zfocus$: width of the Gaussian beam at its narrowest point and the respective depth
- xs, ys and zs: isotropic point source
- $radialSize$, $depthSize$: size of the tissue in width and depth
- *bins*: number of elements in row and column of the bin matrix, needs to be 2^n to ensure the building of a proper quad tree
- $tissueExtMedium$: refractive index of the external medium
- *threshold* and *chance*: used in roulette, when upper weight threshold is reached, there is a chance to survive

Also, for all used environments data needs to be provided:

- *name*: name of the tissue, used for identification
- *mua*: absorption coefficient
- *mus*: scattering coefficient
- $excitAntisotropy$: excitation anisotropy (dimensionless)
- $refractiveInde$: refractive index, needed when the environment is at the bottom of the tissue
- *height*: the height placement of the environment in the tissue from $h1$ to $h2$; only used, if the tissue is not default
- *radius*: radius of the environment; only used, if the tissue is not default
- $default$: set to 1, if default environment, else to 0; only one environment can be default

It is important to consider that more and denser environments can have a significant influence on the execution time of the program. The simulation is performed by invoking the "mcsubLIB.py" file in the console. As output, a ".dat" file is received, which can be analyzed with a Matlab script. The output file displays the collected data about the specular reflectance, the absorbed and escaped fraction, and most importantly, the bin matrix with the collected weight.

4 mc321.c and mcsub.c

The project is based on "mcsub.c" from www.omlc.org/software/mc/, but as an introduction, I started with the simpler version "mc321.c". This simulation sends photons in an infinite homogeneous medium with no boundaries and collects the weight loss in bins based on spherical, cylindrical, and planar radial positions. The specific variables of the medium can be set before starting the program. After each step, it scatters the photon into a new trajectory based on the Henyey-Greenstein scattering function. If the weight is below a certain threshold at the end of the step, it has a chance to multiply its weight by $1/chance$ or vanish. The photon takes a new step until it is declared dead because we are in an infinite medium. The output is the weight of the bins subdivided into spherical, cylindrical, and planar versions.

mcsub.c simulates a semi-infinite homogeneous medium with a surface boundary and collects the weight loss in a planar position, the vector of escaping photons, the reflected photons, the absorbed fraction, and the escaped fraction. All data are measured with cylindrical coordinates. It can be chosen between three different beams: uniform collimated, Gaussian, and isotropic point. The launch of the photon is similar to mc321.c with the difference that at a negative z value, it is calculated whether the photon is reflected into the tissue or escapes.

5 Execution - Way Of A Photon

In the following section the whole path of a photon is described, from the chosen start location to its death. For better readability some variable names are different to code, but the structure is kept the same. An u before a coordinate variable describes its trajectory. The function $RFresnel$ calculates the specular reflectance and the outgoing angle of photon going from one medium into the other. As input the reflectance of both mediums as well the costheta needs to be provided.

5.1 Starting point: Beam variants

There are three possible beam variants for the starting point of the photon.

Uniform collimated beam: The uniform collimated beam is a type of light beam in which the light rays are parallel to each other and the beam maintains a consistent intensity across its cross-section. Thus, the intensity does not vary from one point to another within the beam.

Gaussian Beam: The Gaussian beam is a type of electromagnetic wave that has form characterized by its Gaussian distribution. It is commonly used in optics and laser physics due to its predictable and manageable properties. The intensity of a Gaussian beam follows a Gaussian distribution along the x-Axis due to the cylindrical symmetry. This means that the intensity is highest at the center and decreases exponentially towards the edges. The focal point is the location where light rays converge and the beam waist is the range where the beam diameter is at its minimum, placed around the focal point.

Isotropic Point Source: An isotropic point source is a theoretical model to describe a idealized source of waves that radiates energy equally in all directions. The source is considered a point, meaning it has no physical dimensions and as it is isotropic the energy radiates uniformly with same intensity in all directions.

5.2 A Step

The journey of a photon consists of multiple steps. The length of a step is randomly, logarithmically chosen, so short steps are more common, and it is based on the sum of the scattering and absorption coefficients. The higher the sum of the coefficients, the shorter the step. At the destination, the photon loses weight calculated with the absorption percentage. If the weight is high enough, a spin in the trajectory is executed, and the next step begins.
In a homogeneous tissue, every step is exactly like the previous one besides the different trajectory. We only have to check after every step if the weight is still high enough. In a non-homogeneous tissue, we are confronted with completely new problems:

- after every step there is the possibility for a change in the environment
- we need a way to find out, if we are in a new environment
- the crossing point is important, because in a new environment the lasting step size can be greater or smaller than in old one

Previously, a step was a linear calculation; now, there are many things to check, which result in longer code and expensive calculations.
To address the third point, the step is a loop that stops only if no step size is left over. If the photon reaches a new environment, the rest of the step size is converted to fit with the new environment variables. So the loop is only used if a crossing is happening. Finding the environment for every exact point is very expensive and not needed because we collect the results in bins. So we assign every bin to an environment, and the boolean function checkSameBin returns if the photon is in a new bin, which resolves the first point. By a negative result, we can proceed for this step like in homogeneous tissue.

When a border is crossed, first, we need to know if the photon is in a negative height and therefore tries to leave. Remember, the tissue is semi-infinite and has a surface boundary at the entrance side of the beam. In this case, the function RFresnel returns a [0,1] value based on the refractive indices of the two media and the z-trajectory. This value is compared to a uniformly normally distributed random number, and the outcome decides whether the photon reflects or escapes. In the case of escape, the rest of the weight is saved in its own variable escapeFlux. Reflecting the photon just inverts its z-position from negative to positive. Assume the environment at the bin of the destination is the same as at the start. Then we can proceed as if we never crossed a border and proceed as if we were in the same bin. In the other case, we calculate the distance to the border, make a small step inside the new bin, save the data for the partial step in the previous bin, and set the new environment variables. The next partial step is then done in the new environment.

At the end of every step, we scatter the photon in a new trajectory and do a roulette for all photons with not enough weight to conclude if they die or regain weight.

5.3 Collecting Data

During the journey of the photon, there are four different kinds of data that can be collected:

- absorb information: in every bin, the photon travels through, the lost weight is saved. This data is collected at every step and partial step.
- escaped weight: if the photon escapes, the remaining weight is saved with the information of the last bin. This data is collected, when a photon lefts the tissue.
- specular reflectance at start: when the photon enters the tissue, for some beams there can be an initial lost in weight at this point. This data is collect after setting up the initial weight of a photon.
- total absorbed weight: total weight of all weight collected by the absorb information

5.4 Spin And Roulette

To calculate the trajectory of the next step, a so-called spin is taken upon the photon. This spin uses the Henyey-Greenstein scattering function to describe the angular distribution of the light scattering.

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

where:

- θ is the scattering angle.
- $\cos\theta$ is the cosine of the scattering angle.
- g is the excitation anisotropy parameter, which ranges from -1 to 1.

Further the azimuthal scattering angle ψ is randomly chosen and $sin(\psi)$ as well as $cos(\psi)$ calculated. The combination of the sine and cosine-modified results from the Henyey-Greenstein scattering function and the azimuthal scattering angle with the previous trajectory gives the new trajectory.

Before the next step is taken, it also has to be ensured that the photon has enough weight. When the photon is under the threshold predefined in the input file, it undergoes roulette to determine if it survives. The roulette contains a chance, likewise defined in the input file, to multiply its weight by the reciprocal of the chance. An example with a threshold of 0.0001 and a chance of 0.1: A photon with weight of 0,00005 has a $\frac{1}{10}$ chance to survive and then having the weight of $0.00005 * 10 = 0.0005$ instead.

6 Core Components

6.1 Multiprocessing

The goal is to make the execution time faster by running multiple photon launches in parallel. In general, Python uses a global interpreter lock to ensure that only one thread executes Python bytecode at a time. With the multiprocessing package, we can sidestep

the global interpreter lock by using subprocesses. This package spawns multiple processes, which take advantage of multiple cores on a computer. Each process runs independently, so a calculation-heavy journey of a photon does not hinder the progress of other photons. Also, each process has its own memory space, which results in more RAM usage but reduces the communication between parent and child processes.

For the data exchange between the main process and the spawned process queues are used. They give an easy and fast accessibility with the commands push, add an item at the end, and pop, take the first item of the queue. There are two queues in usage:

- queue for photons: has the numbers from 1 to number of photons, ensures that the exact number photons get launched, at the end there are "DONE" times number of processes added to tell the processes to finish
- queue of results: all collected data of a process are pushed in here, when a photon dies. A seperate function sorts in this result of all processes into a data set.

To start the multiprocessing we create a process with a targeted function to execute and the input variables. These are than copied into an own memory, if they were not declared prior as communication channel. With a daemon all subroutines are eliminated, when the program finishes. This ensures that there are no left overs. All processes are saved in a array, which we need later to use the join method. This ensures that every process has finished at this point, so all data is collected and there are no more changes in variables. Therefore the daemon and the method join are safety precautions for correct results and a properly execution.

The main problem with multiprocessing is the overhead. It always takes time to reserve the memory and other system resources for every process, but the similar execution should make up for it. However, there are cases, particularly with big quadtrees and low photon numbers, where the creation takes more time than the simulation. For efficient execution, some tinkering with the input needs to be done.

6.2 Quadtree

With multiple environments it gets important to know in which environment the photon currently moves. It is to expensive to check after every step, if the coordinates of a photon lie in an environment, especially because we would have to check every environment. Every point of the tissue is represented of a bin and so it is every position of a photon. Now the idea is to find quickly the right bin for this position.

A quadtree is a tree where every internal node has exactly four children by recursively subdividing the space of a node into four quadrants or regions. The root node represents the whole tissue, and every leaf represents one bin. Therefore, the selected bins for height and radius have to be always the same. Every leaf is assigned to an environment.

To build the tree, we first create a list that represents every bin in the simulation, and these bins are assigned to the default environment. Then, the environment manager is called to assign all environments to a bin with its center inside their space. The quadtree is built recursively with the input of the length, width, and depth. Every time a leaf is added, the node also gets an environment by the list fitting to the representing bin. The time for searching a bin is $O\log(n)$ because the tree is balanced. This time can even be improved by a small adjustment: Every time when all children are assigned to the same environment, the parent also gets assigned to the environment, and our search ends

the first time the environment variable of a node is not empty. In areas with only one environment, far fewer comparisons need to be done for a response by the quadtree.

In conclusion, the quadtree is a data structure with all needed information about every point in the tissue and can be expanded by adding more attributes to the nodes. It is used by just providing the coordinates of a point.

6.3 Environment Manager

The purpose of the environment manager is twofold: First, it should make it easier for the developer to consume the environment data by providing a database with corresponding functions over all environments. Second, the input file should only be read once and the data properly saved in the program. The environment manager creates its data with the "environment" sub-class and saves it. It currently has functions to add an environment, find an environment by variables, assign environments to a 1D-list based on the location of the list, and return the default variables, because exactly one environment has to be declared as default by the user.

7 Explanation Of Classes And Functions

7.1 Function "launch"

Purpose: The launch function simulates the course of a photon through a tissue model. It initializes photon positions and trajectories based on various light source models and handles the scattering and absorption of the photon within the medium.

Input Parameters:

- queuePhotons (multiprocessing.Queue): A queue containing the total of one up to the numbers of photons.
- environmentGeneral (dictionary): A dictionary containing general parameters needed for every simulation.
- queueResult (multiprocessing.Queue): A queue to store the results and collected data of the photon simulation. Another function sorts them.
- ID (int): Identifier of the process.
- tree (QuadTree): A tree data structure for efficiently querying environmental properties at given coordinates.

Workflow: The process sets the general parameters, that are always the same for every simulation like the size of the tissue or the number of bins. Then it pops a photon of queuePhoton and sets the starting position and trajectory based on the type of light source. If it reads "DONE", it puts an "DONE" in queueResult to indicate the sort process it has finished and terminates itself. As long as the photon is indicated as alive, it executes the following steps:

- Calculate randomly the step length

- As long as the size of the step is greater than zero, if the photon is in the same voxel we calculate the new position and deposit the lost power in the corresponding cylindrical bin. Else we calculate the smallest step over the border of the next voxel and deposit only the power of the partial step.
- If the photon lefts a voxel we have to check, if it is the lower bound and the so it based on chance and refraction, if it is lost or reflected into the tissue
- A spin gets applied at the photon to determine the new trajectory
- For low-weight photons a roulette technique gets applied to roll, if it gets terminated or gains weight

7.2 Function "writer":

Purpose: The writer function creates the photon queue for the launchers to read.

Input Parameters:

- count (int): The number of integers to write to the queue.
- numOfReaderProcs (int): The number of reader processes.
- queue (multiprocessing.Queue): The queue to which the integers will be written.

Workflow: It writes the numbers of 0 to the numbers of photon minus 1 to the queue and puts for every reader process a "DONE" at the end.

7.3 Function "sort"

Purpose: The sort function collects and aggregates results from the launcher processes.

Input Parameters:

- queue (Queue): The queue from which to read results.
- processes (int): The number of processes to wait for.
- escapeFlux (list): A list to store the escape flux data.
- absorbInfo (list): A list to store absorption information.
- tempRsptot (float): A variable to store total specular reflectance.
- Atot (float): A variable to store total absorbed photon weight.

Workflow: It reads results from the queue until it receives "DONE" from all processes. It sorts in the absorption information and sums up escape flux, total specular reflectance and total absorbed weight. Then the aggregated results are returned.

7.4 Function “checkSameBin“

Purpose: The checkSameBin function checks if two points lie within the same bin

Input Parameters:

- x1, y1 (float): Coordinates of the first point.
- x2, y2 (float): Coordinates of the second point.
- dx (float): Width of the voxel in the x-direction.
- dy (float): Height of the voxel in the y-direction.

Workflow: Returns a boolean if the voxel indices of the two points to determine are in the same voxel.

7.5 Function „FindVoxelFace“

Purpose: The FindVoxelFace function calculates the step size to the next voxel boundary for a photon.

Input Parameters:

- r1, z1 (float): Initial radial and depth positions of the photon.
- dr (float): Radial bin size.
- dz (float): Depth bin size.
- ur (float): Radial direction cosine of the photon’s trajectory.
- uz (float): Depth direction cosine of the photon’s trajectory.

Workflow: It determines the indices of the current and next voxels based on the photon’s direction Then calculates the distances to the next voxel boundaries in the radial and depth directions and returns the smaller of the two distances as the step size.

7.6 Function: RFresnel

Purpose: The RFresnel function calculates the Fresnel reflectance based of the transmission angle for a photon between the tissue environment and the outer environment.

Input Parameters:

- n1 (float): Refractive index of the incident medium.
- n2 (float): Refractive index of the transmitting medium.
- ca1 (float): Cosine of the angle of incidence.

Workflow: First it handles special cases for matched boundaries and normal incidence. Then it calculates the sine and cosine of the transmission angle. It determines the reflectance based on the Fresnel equations and returns the reflectance and the cosine of the transmission angle.

7.7 Function: SaveFile

Purpose: The SaveFile function saves the simulation results to a file.

Input Parameters:

- Nfile (int): File identifier for naming the output file.
- J (float): escaped Flux
- F(float): absorbed weight
- S(float): specular reflectance
- A(float): absorbed fraction
- E (float): escaped fraction
- environmentGeneral (dict): general environmental parameters.
- Nphotons (int): Number of photons simulated.

Workflow: It opens a file named "mcOUT (Nfile).dat" and writes the simulation parameters and results to the file,

7.8 Function Main:

Purpose: The main function builds the data sets and calls the subfunctions to process the data.

Worklflow: Based on the input file, the environment manager creates all needed environments. Further, the quadtree is built, and the queues for the multiprocessing are declared. First, all launch processes are started in the background, and then the writer and sort functions are called. The launch processes are on hold until the writer function puts the number of photons in the photon queue. Also, the program does not proceed further until the sort function has terminated. In this way, it is ensured that no information is left over. After the sorting, we also check that all launcher processes have correctly finished. This way, we know that the allocated memory of the processes is again free. Now all relevant data is collected, and the variables are modified for the save file. The program finishes by printing some general information to the console, especially the execution time.

7.9 Class environment

Purpose: The environment class saves information about a environment.

Functions:

- InEnvironment(radius, height): returns a boolean, if the variable is inside of the environment.
- GetVariables(): returns the name, absorption coefficient, scattering coefficient and excitation anisotropy

7.10 Class manageEnv

Purpose: The manageEnv class saves all environment classes and manages general request to all environments. It contains also a default environment, which has no specific location in the tissue and is therefore always taken, when no other environment fits at a position.

Functions:

- addEnvironment(name, mua, mus, excitAnisotropy, height, radius, default): Creates the environment „name“ with all the corresponding variables and sets it as default environment, if default is true. Only one environment can be set as default.
- FindEnv(x, y, z): returns all variables of the environment for the given location data.
- assignEnv(envList, pixel): calls every environment and updates a 1D-Array with the environment variables based on the location data.
- GetDefaultVariables(): returns the variables of the default environment

7.11 Class Node

Purpose: The class Node saves information of a node in the quadtree, like radius, height, centre, assigned environment, children and ist ID.

Functions:

- SetChild(child): adds the node “child” to the array “children”
- SetEnv(env): set the environment variable to the dictionary “env”
- GetEnv(x, y): returns the environment of the node. If it has no environment set it searches the right child for the coordinates.
- GetSquares(x, y): called by getEnv. Returns the right square for the coordinates based on the centre coordinates of the node.

7.12 Class photon

Purpose: The class photon contains the position and the weight of a photon.

Functions:

- UpdatePosition(x, y, z): adds variables to current coordinates
- getPosition(): returns current coordinates
- getSphericalPosition(): returns spherical position
- getPlanarPosition(): returns planar position
- setWeight(w): set weight to w
- updateWeight(w): adds w to the weight of the photon
- getWeight(): returns weight of the photon
- updateDead(): set status to 0
- getStatus(): returns, if the photon is dead (0) or alive (1)

7.13 Class quadTree

Purpose: The class quadtTree creates a quadtree array by calling the class node recursively and manages it.

Functions:

- createTree(r1, r2, h1, h2, pixel, depth, envList): creates recursively a tree with with radius space r1 to r2 and height space h1 to h2, the number of pixels in height and radius, the depth of the tree and a list envList with the environment information for every leave
- getEnv(x, y): returns the environment for the coordinates (x,y)
- getNode(n): returns the n-th node of the tree

8 Output

The program gives some output in the command line, mainly for controlling its process and to determine possible errors or heavy time losses. All collected data is saved in the structured file "mcout1.dat", which can be simulated with MatLab and a program provided on this website: omlc.org/software/mc/mcsub/index.html.

The command line output consists of start and finish information of the subprocesses from the multiprocessing and time stamps for the creation of the environment list, the quadtree and the creation of all subprocesses, as well as the total execution time. If the creation of the subprocesses takes to much time, there is to much overhead. You should consider to reduce the number of bins.

In the file "mcOUT1.dat" are saved:

- general information about the simulation like the used beam or the number of launched photons

- the total specular reflectance, the absorbed and excaped fraction
- most important a 2-dimensional list with depth values of the bins and their respective absorbed photon distribution.

9 Benchmarks

The the benchmarks should prove the efficiency and robustness against changes in the input variables. As standard input some challenging values are taken, to get a real comparison and all variables with possible impact on the execution time were changed. The number of photons are kept the same, because every the launch time of every photon has the same distribution, and the threshold was not changed, because the purpose of threshold is getting meaningful weight losses. All test were done at an AMD Ryzen 7 5800X 8-Core Processor and 48 GB RAM. The standard variables are:

- photons: 100.000, higher amount chosen to minimize the overhead of the program and compare the calculations
- bins: 128, more bins means are deeper quadtree and more border crossings of a photon
- chance: 0.1, survival rate and therefore more calculations
- threshold: 0.0001, predefined numbers kept
- Beam: Uniform Collimated
- tissue: height and radius of 0.003 cm
- environments: standard tissue from height 0 to 0.001 and 0.002 to 0.003, blood form height 0.002 to 0.003; this simulates a pseudo three environment tissue, because the are two internal borders

Bins: More bins results in a deeper quadtree and more border crossings, a small increase in time with more bins make sense.

Bins	64	128	256
Time	28.48 s	29.60 s	31.69 s

Chance: The chance for getting back weight has no big influence at the execution time.

Chance	0.01	0.1	0.2
Time	28.80 s	29.60 s	29.70 s

Beam: The beam is only considered for the starting point of the photon. Therefore the Isotropic Point has significant less calculations to do, then the other two beams

Beam	Uniform	Gaussian	Isotropic Point
Time	29.60 s	29.26 s	24.88 s

One environment: 27.43 s
In this test only the default environment was taken. The time savings are possible come from the replacement of the denser blood tissue with the lighter default one. Because every bins has the same environment the quadtree must never be called.

Skull instead of standard: 141.10 s
This test shows how much more calculation time a denser environment takes.

Results: The most influence at the execution time takes the chosen environments. The simulation itself is very resilient against changing the standard values and keeps a steady time over all the different framework conditions beside the environments. The building time of the mulitprocessing is constant between 7 and 8 seconds a small cost with big regains in the parallel execution of the photon launchs.

10 Failed Implementations

In my execution of the project I tried different approaches for a fast, flexible and low memory-low implementation. Often parts of the initial idea were in the end used. For example I started with a Octtree, because the photons are travelling through 3-dimensional space. But the input and the measurements are done in two dimensional space, so I reduced the tree to a quadtree, while keeping the structere of the tree for now always four instead of eight children. The following topics were implemented in some way with the hope of time and memory improvements, but were in the end not advantageous.

10.1 Raytracing

Ray tracing is a rendering technique used in computer graphics to simulate how light interacts with objects to produce realistic images. The fundamental idea behind ray tracing is to model the path of photons as they travel from a light source, interact with surfaces, and finally reach the viewer's eye. The process begins with rays being cast from the eye into the scene, rather than from the light source. These rays traverse the scene, interacting with objects to determine the color and intensity of the light. For each ray, it is determined if and where it intersects with objects in the scene, and equations are used to calculate the scattering and absorption values of objects.

This process is quite similar to the one used in this program. We start with a light source (the eye) and send the photon out to interact with atoms (objects). By calculating the different possibilities for each potential collision, we can simulate the path of a photon using random numbers after each step.

The problem is that there are so many possible paths for a photon that each journey is nearly unique. Ray tracing calculations are very expensive, especially when many objects are involved. In graphic design, the number of objects is manageable, but the number of steps (collisions) is difficult to estimate. Ultimately, this problem does not align perfectly with the use case of ray tracing, even though they share a similar starting position.

10.2 Multiple Functions For Multiprocessing

In multiprocessing, each process reserves its own memory. This can lead to significant overhead and memory issues, especially with a large quadtree. I first encountered this problem when I used an octree. My solution was to reduce the number of tree calls by adding additional processes ("searchers") with a function that searches the tree for given variables from the "launchers." With one-fifth of all processes being searchers, there were also only one-fifth of the tree calls, drastically reducing memory usage. To reduce lock time, a queue was used where all launchers placed their search requests, and each launcher had its own response variable. The searchers only had to wait for new requests, and the launchers received their responses directly.

On paper, this approach made sense, and the reduction in memory usage was a significant success. However, the running time suffered considerably. It took the same amount of time to process 1,000 photons as it previously did to process 100,000 photons. I believe the locking mechanisms, which were necessary to ensure no data corruption, cost too much time, especially due to the forward and backward exchange of information. In the current version, the result queue forms a pipeline in only one direction, from the launchers to the sort function.

11 Conclusion

The program was successfully translated into Python, and additional features were added to make it practically usable. Compared to its original C version, it lost some execution time but is now much more flexible for further development or modifications. Additionally, with the implementation of multiprocessing, the program utilizes the full CPU power of a computer. Potential further improvements could include:

- user friendly: The program should check, if the inputs are right and fitting.
- optimization for dense environments: most of the testing while development were done with the standard test variables. When changing to denser environment, there are heavy execution costs.
- picture output: instead of relying on the matlab program, this one could get its own visualisation.

12 Code

The code of this project can be found in this public github repository:

https://github.com/xkohlbrenner/praktikum_mcls

Multiple test outputs are in the test folder. If not different described, the stnadard variables are taken:

mcOUT_std.dat: output of with standard variables mcOUT_gaussian.dat: output with gaussian beam mcOUT_isotropic.dat: output with isotropic point beam mcOUT_256.dat: output with 256 bin instead of 128 mcOUT_blood.dat: output with the environment "blood"