Comment on “A study on tetrahedron-based inhomogeneous Monte-Carlo optical simulation”

Qianqian Fang*
Martinos Center for Biomedical Imaging, Massachusetts General Hospital, 149 13th Street, Charlestown, Massachusetts, 02129, USA
*fangq@nmr.mgh.harvard.edu

Abstract: The Monte Carlo (MC) method is a popular approach to modeling photon propagation inside general turbid media, such as human tissue. Progress had been made in the past year with the independent proposals of two mesh-based Monte Carlo methods employing ray-tracing techniques. Both methods have shown improvements in accuracy and efficiency in modeling complex domains. A recent paper by Shen and Wang [Biomed. Opt. Express 2, 44 (2011)] reported preliminary results towards the cross-validation of the two mesh-based MC algorithms and software implementations, showing a 3–6 fold speed difference between the two software packages. In this comment, we share our views on unbiased software comparisons and discuss additional issues such as the use of pre-computed data, interpolation strategies, impact of compiler settings, use of Russian roulette, memory cost and potential pitfalls in measuring algorithm performance. Despite key differences between the two algorithms in handling of non-tetrahedral meshes, we found that they share similar structure and performance for tetrahedral meshes. A significant fraction of the observed speed differences in the mentioned article was the result of inconsistent use of compilers and libraries.

©2011 Optical Society of America

OCIS codes: (170.3660) Light propagation through tissue; (170.5280) Photon migration; (170.7050) Turbid media.

References and links
1. H. Shen and G. Wang, “A tetrahedron-based inhomogeneous Monte Carlo optical simulator,” Phys. Med. Biol. 55(4), 947–962 (2010).
2. Q. Fang, “Mesh-based Monte Carlo method using fast ray-tracing in Plücker coordinates,” Biomed. Opt. Express 1(1), 165–175 (2010).
3. E. Margallo-Balbás and P. J. French, “Shape based Monte Carlo code for light transport in complex heterogeneous Tissues,” Opt. Express 15(21), 14086–14098 (2007).
4. C. Wächter, “Quasi-Monte Carlo light transport simulation by efficient ray tracing,” Ph.D. dissertation (Ulm University, 2007)
5. N. Ren, J. Liang, X. Qu, J. Li, B. Lu, and J. Tian, “GPU-based Monte Carlo simulation for light propagation in complex heterogeneous tissues,” Opt. Express 18(7), 6811–6823 (2010).
6. J. Wald, “Realtime ray tracing and interactive global illumination,” Ph.D. dissertation (Saarland University, 2004).
7. D. Badouel, “An efficient ray-polygon intersection,” Graphics Gems, S.A. Glassner, ed. (Academic Press Professional, 1990), pp. 390–393.
8. N. Platis and T. Theoharis, “Fast ray-tetrahedron intersection using Plücker coordinates,” J. Graph. Tools 8(4), 37–48 (2003).
9. M. Shevtsov, A. Soupikov, and A. Kapustin, “Ray-triangle intersection algorithm for modern CPU architectures,” in Proceedings of GraphiCon 2007, (2007), pp. 33–39.
10. H. Shen and G. Wang, “TIM-OS Project Site,” https://sites.google.com/a/imaging.sbes.vt.edu/tim-os/.
11. Q. Fang, “Mesh-based Monte Carlo—the software,” http://mcx.sourceforge.net/mmc/.
12. H. Shen and G. Wang, “A study on tetrahedron-based inhomogeneous Monte Carlo optical simulation,” Biomed. Opt. Express 2(1), 44–57 (2011).
13. J. Havel and A. Herout, “Yet faster ray-triangle intersection (using SSE4),” IEEE Trans. Vis. Comput. Graph. 16(3), 434–438 (2010).
Comment

The recent proposals of 3D mesh-based Monte Carlo (MC) photon transport algorithms, i.e. TIM-OS (Tetrahedron-based Inhomogeneous MC Optical Simulator) by Shen & Wang [1] and MMCM (mesh-based MC method) by Fang [2], clearly echoed the increasing needs for accurate and computationally efficient modeling of complex domains in bio-optics applications. In both methods, a 3D volumetric mesh is utilized to represent complex domains as well to facilitate photon propagation. Compared to previous MC works using triangular surface meshes [3–5], both methods showed significantly reduced computational overhead in ray-triangle intersection tests by tracking the tetrahedron enclosing the photon at any given moment. This results in on average 2.5 and 4 ray-triangle intersection tests per propagation step for MMCM and TIM-OS, respectively. By comparison, the surface-mesh based approaches require testing of all triangles inside a “partition” of the surface [3–5], where the ray-triangle test count is typically greater than 4 per step in addition to the even more costly overhead of traversing the partition data structure [6].

Despite independent development paths, both mesh-based algorithms show clear roots in the classic ray-tracing techniques used in numerous computer graphics applications. The algorithm developed for TIM-OS is a partial implementation of the Badouel’s [7] and Wald’s methods [6], without the computation related to Barycentric coordinates; the algorithm in MMCM, on the other hand, was inspired by the Plücker-coordinate-based ray-tracing technique [8], which potentially resembles the Shevtsov’s method [9]. Because of the generality of Plücker-based tests, MMCM can theoretically handle not only tetrahedrons, but also other shapes with polyhedral faces. In the case of a tetrahedral mesh, MMCM can use piece-wise-linear (C1) basis functions to sample the continuous fluence and optical property distributions, while TIM-OS focuses on piece-wise-constant (C0) basis for better speed.

Both teams had released software implementations, namely timos (binary-only) [10] and mmc (open-source) [11], based on the algorithms they published (in this comment, we use lower-case names for implementations and capital names for algorithms). A rigorous cross-validation between the two codes could reveal key differences and trade-offs which may benefit the further development of the method. In a recent paper published in Biomedical Optics Express [12], Shen and Wang took the first step in this direction and reported their preliminary findings comparing the computational speed and accuracy of the two codes, along with two other MC packages, using the binary builds of the software.

In this comment, we discuss several additional issues related to this comparison that were not discussed in [12]. In the following sections, we share our opinions on the theoretical differences of the two algorithms and details on various issues that may lead to potential biases in a software comparison. Our findings are summarized at the end of this comment.

Comparison of the ray-tracing algorithms

Given the generalized conclusion in [12], the authors of the article showed clear interests in exploring the theoretical performance of the two algorithms. However, in the FLOPS (floating-point operations) analysis presented in the Appendix of [12], the authors used “textbook” formulas [8] to compare the ray-tracing kernels in TIM-OS and MMCM. In practice, this may lead to a suboptimal estimate of performance, since many contemporary ray-tracing algorithms extensively use pre-computed data to accelerate the computation [4,6,9,13]. This is especially the case when using Plücker-coordinate-based ray-tracing, such as in Shevtsov’s method [9].

Recently, Havel and Herout [13] published a brief review on the ray-tracing algorithm state of the art. They also proposed a hybrid approach that combines the strengths of Wald’s
and Shevtsov’s methods. Given the connection between mesh-based MC and ray-tracing algorithms, we found this paper particularly helpful to understand the algorithmic differences between TIM-OS and MMCM.

Combining Eqs. (9) and (10) in [13], we can reconstruct a pseudo-code that describes Shevtsov’s method in a format similar to that in the Appendix of [12]. The pseudo-code, the “apparent” FLOPS count of each step and the probability of each execution path are listed in the following table. For consistency, we count division as 1 FLOP as in [12]; in practice, division is typically more expensive than other operations.

### Table 1. The Pseudo-Code for a Ray-Tetrahedron Intersection Test using Shevtsov’s (Plücker-Based) Method

| Calculation                                                                 | Apparent FLOPS | Execution path & probability |
|-----------------------------------------------------------------------------|----------------|------------------------------|
| $\det = D_p N_p + D_q N_q + D_r$                                           | 4              | 0.40                         |
| $t' = d - (O_p N_p + O_q N_q + O_r)$                                       | 5              | 0.09                         |
| if (sign($t'') == sign(det * $t_{max} - t''))                               | 4              | 0.11                         |
| $T_p = t' D_p - \det (A_p - O_p)$                                          | 4              |                              |
| $T_q = t' D_q - \det (A_q - O_q)$                                          | 4              |                              |
| $u' = E1 \cdot T_p - E1 \cdot T_q$                                         | 3              |                              |
| if (sign($u'') == sign(det - $u''))                                        | 3              | 0.40                         |
| $v' = E0 \cdot T_q - E0 \cdot T_p$                                         | 3              |                              |
| if (sign($v'') == sign(det - $u' - v''))                                   | 4              |                              |
| $r\det = 1/\det$                                                          | 1              |                              |
| $t = r\det * t''$                                                         | 1              |                              |
| $u = r\det * u''$                                                         | 1              |                              |
| $v = r\det * v''$                                                         | 1              |                              |

*aAll symbols are defined in [9] and [13].

For simplicity, we numerically estimate the probabilities of different execution paths by simulating a large number of photons using the above method at various scattering settings. We find that these numbers quickly converge to the listed values independent of the scattering and anisotropy settings.

By summing up the FLOPS counts weighted by the execution probabilities, we estimate the average FLOPS cost per ray-triangle intersection test as 26.6. Because on average MMCM requires $2.5 = (1 + 4)/2$ tests per tetrahedron, the average “apparent” FLOPS count for Shevtsov’s method is 66.4 per ray-tetrahedron test. This number is smaller than the 78 FLOPS found in [12] using the “text-book” formula. If we save the signs of the $N_i$ component [9], we can insert “if(det>0)” after the second row of Table 1 to skip the back-facing triangles [13]; this can further reduce the “apparent” FLOPS count. Compared with the 44 “apparent” FLOPS for TIM-OS [12], Shevtsov’s method needs roughly 50% more FLOPS in theory.

In the above discussion, we highlighted the fact that the FLOPS estimates were taken as the “face values”. In modern CPUs, these values are not necessarily proportional to the execution time. Memory latency is another important factor and proper memory pre-fetching can significantly impact the efficiency [13]. Also, the if-branches in Table 1 and Appendix of [12] can hurt efficiency by causing pipeline stalls in modern CPUs [9]. In [9], the authors had shown that a branchless structure can be more efficient despite the slight increase in FLOPS.

In addition, we found Eq. (13) in [13] very helpful in understanding the key differences between TIM-OS and MMCM in the tetrahedral case. Comparing the equations, we found that the core ray-tracing kernel in TIM-OS requires the first, second and the sixth equations; for...
MMCM, all equations are needed. From this perspective, we can see that the two algorithms are not really “quite different” as concluded in [12]. Rather one is a partial version of the other: MMCM performs a complete ray-tracing calculation while TIM-OS bypasses the calculations for Barycentric coordinates.

We want to note particularly that Plücker-coordinate-based MC is not limited to tetrahedral meshes. MMCM can be extended to support general polyhedral elements, such as hexahedrons, or incorporate high-order basis functions, i.e. the p-method, to enhance accuracy [2]. In [12], the authors proposed to use refined elements, i.e. the h-method, to reduce discretization error. Based on the error analysis for mesh-refinement, the discretization error of the p-method diminishes much faster than the h-method [14]. To produce solutions with a comparable discretization error, the computational expense of the h-method can quickly exceed that of the p-method as we refine the mesh.

Comparison of interpolation strategies

Although the fluence is calculated using the piece-wise-constant basis functions in timos, the authors also proposed to use a linear interpolation as a post-process [12]. To a certain extent, this strategy allows timos to approximate the continuous fluence profile, similar to mmc, where the interpolation is done during the energy deposit phase. However, we want to point out that these two strategies are not equivalent. More specifically, we found two issues related to the post-interpolation strategy. First of all, the linear interpolation between the centroids of all tetrahedra is typically achieved by first constructing a 3D tessellation. Thus, the interpolation is only defined inside the convex hull of the centroids, which, in most cases, is smaller than the original mesh. Secondly, the space where the fluence is calculated is different from where it is interpolated. This discrepancy can cause an increase in discretization error.

To further illustrate the second issue, we use a 1D example to show the difference in accuracy of the two interpolation strategies. For simplicity, we select a continuous function \( f(x) = 1/x \) and define a set of equally spaced nodes at \( x_i = i \Delta x \) \((i = 1, 2, \ldots)\). Our goal is to estimate the integrals in the form of

\[
y(i) = \int_{i\Delta x}^{(i+1)\Delta x} f(x)dx.
\]

If we choose the mmc strategy, we first define the 1D linear basis as \( \phi_i(x) = 1 - |x - i\Delta x|/\Delta x \) and integrate \( f(x)\phi_i(x) \) in the intervals \([i\Delta x, (i+1)\Delta x] \) and \([(i-1)\Delta x, i\Delta x] \); alternatively, we can integrate \( f(x) \) in both intervals and assign the results to the middle points. We then approximate \( y(i) \) by taking the average, which is essentially a post-interpolation. It is not difficult to analytically show that

\[
y_{\text{mmc}} = \frac{1}{2} \left( \int_{(i-1)\Delta x}^{i\Delta x} f(x)dx + \int_{i\Delta x}^{(i+1)\Delta x} f(x)dx \right) = \frac{1}{2} \ln((i+1)/(i-1))
\]

\[
y_{\text{timos}} = \frac{1}{2} \int_{(i-1)\Delta x}^{i\Delta x} f(x)dx = \ln\left( (2i+1)/(2i-1) \right)
\]

In Fig. 1, we plot the absolute and relative errors for \( y_{\text{mmc}} \) and \( y_{\text{timos}} \) at two \( \Delta x \) settings. Several observations can be made from this plot: (1) the \( \text{mmc} \) strategy produces a better approximation than the post-interpolation approach, especially when \( f(x) \) has large variations, and (2) the error of the post-interpolation becomes greater as \( \Delta x \) increases. Similar analysis can be done in 2D and 3D spaces. For practical photon transport simulations, we expect the solutions produced by post-interpolation to have lower accuracy near the source and with coarse meshes.
Impact of compilers and optimization settings

By running benchmarks for several configurations with the pre-compiled binaries, the authors of [12] showed that timos software is 3–6 times faster than mmc. However, no analysis was given. With further investigation, we found that this reported difference was largely dominated by differences in compilers and libraries instead of the algorithms themselves.

To illustrate the above finding, we recompile the mmc source code with various compiler and optimization settings and report the run-times in Table 2.

Table 2. Comparison of Run-Times for Various mmc and timos Builds with Different Compilers and Optimizations

| Binary           | Machine-A (2x Intel E5530) | Machine-B (Intel Core i7-870†) |
|------------------|----------------------------|--------------------------------|
|                  | Run-time (s) | Normalized time | Run-time (s) | Normalized time |
| mmc-gcc          | 529.6 (440.0*) | 3.9 (4.4*)       | 569.5       | 3.9             |
| mmc-icc          | 331.2         | 2.5              | 390.9       | 2.7             |
| mmc-gcc2         | 342.7         | 2.5              | 383.2       | 2.6             |
| mmc-icc2         | 286.0         | 2.1              | 335.4       | 2.3             |
| mmc-icc-SSE      | 209.0         | 1.5              | 217.0       | 1.5             |
| timos            | 135.0 (100.0*) | 1.0              | 147.0       | 1.0             |

*Numbers reported in [12].
†Hyper-threading enabled.

Here, all mmc binaries were compiled on a computer (Machine-A) running 64bit CentOS 5.5 with either gcc 4.1.2 or Intel C++ compiler (icc) 2011.0.084. This computer has two Intel Xeon E5530 CPUs (8 cores total), similar to the one used in [12]. The timos executable was compiled with the Intel C++ compiler and was downloaded from its website [10] on Dec. 10, 2010. All binaries were compiled with the -O3 flag for both compilers except for mmic ICC-SSE where “-fast” and “-msse4.1” options were appended. We chose the “sph2” example [12] as the benchmark, and simulated $3 \times 10^7$ photons for each executable using 32 parallel threads. To estimate the run-time, we ran each binary 4 times consecutively, and took the average from the last 3 runs. We repeated the simulations on a second computer (Machine-B) running on a single Intel Core i7-870 CPU under the hyper-threading mode.

In Table 2, mmc-gcc and mmc-icc were built with the original mmc v0.2 source code released on Aug. 29, 2010 [11]. The difference between mmc-gcc and mmc-icc is solely in the compiler used - gcc or icc - as the suffixes indicate. In mmc-gcc2 and mmc-icc2, we replaced the exponential function expf() defined in the GNU C library (glibc) with an approximation function using the 9th order Taylor’s expansion; in addition, we also added the “static inline”
prefix to enable expansion of inline functions across multiple source units. In mmc-icc-SSE, we further optimized the ray-tracing kernel using the Streaming SIMD Extensions (SSE).

Comparing the run-times in rows 2 and 3, we can clearly see the dramatic difference in speed by simply using a different compiler: the mmc-icc binary is 1.6 times faster than its gcc counterpart! To identify the cause of this difference, we profiled both binaries and found that the slow math functions (exp, log, sin and cos) in the GNU C Library were responsible for the poor performance. As a result, in mmc-gcc2, we replaced expf() with a fast approximation. The run-time of mmc-gcc2 became close to mmc-icc, indicating that the speed enhancement of the icc binaries benefited from a more efficient implementation of the math functions. The normalized run-times (divided by that of timos) for various mmc binaries show a wide swing from 1.5 to 4, indicating the large susceptibility of run-time with respect to the choices of compiler and library settings. Similar findings were observed on Machine-B.

Other related issues

In addition to the compiler and library differences, there are several other issues that also deserve investigation. Insufficient consideration of these issues may also lead to inaccurate estimates of run-time.

From private communication with the timos authors, we learned that the Russian roulette settings were different between the two codes. In timos, the Russian roulette has a built-in threshold of 0.00001 and is used for both static and time-resolved simulations. By comparison, mmc only enables Russian roulette for static simulations. This configuration is based on our belief that using Russian roulette in a time-resolved simulation introduces time-dependent statistical characteristics for the temporal point spread function (TPSF). This is not desired in certain rigorous analysis even though the solutions may be visually similar. The use of Russian roulette can give a slight speed advantage for timos. Nevertheless, the results from the two solvers possess different characteristics. Thus, this comparison is not entirely valid.

Moreover, timos calls the SIMD-oriented Fast Mersenne Twister (SFMT) random number generator (RNG) from the Intel Math Kernel Library (MKL) [15] while mmc uses a plain multi-thread 48bit RNG from glibc. Accelerated by the SSE instructions, SFMT algorithm is known to be fast and it is reasonable to believe that the contribution from RNG differences is non-negligible. In addition to speed, using identical RNG settings in timos and the reference code can also introduce biases in the accuracy assessment in Table 3 of [12]. Nonetheless, the diminishing absorption fraction shown in Table 4 in [12] does reveal a bug in the mmc implementation. By changing the absorption accumulator variable type from “float” to “double,” we find that the above issue disappears.

In [12], the authors focused on simulation speed, but memory utilization, another important aspect of an algorithm, was not fully discussed. Because timos saves fluence at the centroid of a tetrahedral element while mmc records fluence at each node, the memory cost for timos per time-gate is on average 6-8 times larger than mmc (in a 3D tetrahedral mesh, a node is typically shared by 6 to 8 tetrahedra). When running the benchmark in the previous section, we noticed that both timos output file size and time-to-save are several times larger than mmc.

Conclusions

With the up-to-date knowledge of the contemporary ray-tracing algorithms, we compared the core calculations in TIM-OS and MMCM, and concluded that the two algorithms are rather similar when simulating photon transport with tetrahedral meshes. TIM-OS uses a partial ray-tracing scheme for better speed performance; while MMCM performs the full set of ray-tracing calculations to gain accuracy and generality. With better utilization of pre-computed data, MMCM theoretically requires only 50% more computations than TIM-OS for the ray-tracing calculations. The large speed difference shown in [12] between the two implementations, timos and mmc, were found to be related to the selection of compilers, efficiency of the math libraries, expansion of inline functions, Russian roulette settings and...
the selection of RNGs. Guided by this knowledge, we were able to fine-tune the performance of mmc by using optimized math functions and SSE instructions.

From the above discussions, we believe that identifying the detailed causes of performance differences is more informative to the community than reporting running times and error metrics without in-depth analysis. Caution must be exercised to avoid common pitfalls in performance analysis where external components can easily introduce biases to the results. Summarizing the above findings, we believe the main conclusion in [12], “TIM-OS is superior to MMC in terms of efficiency for the general heterogeneous models,” is not sufficiently justified and thus deserves further debate.

Acknowledgment

QF would like to thank Jiří Havel for his help in understanding the SSE implementation in [IEEE Trans. Vis. Comput. Graph. 16, 434 (2010)]. QF also appreciates the kind communications with Haiou Shen and Ge Wang to learn the details regarding the compilation and execution settings for the timos binary. This study was partially supported by the Harvard Catalyst Pilot Grant.