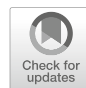
Profiling General-Purpose Fast Multipole Method (FMM) Using Human Head Topology



Dung Ngoc Pham

1 Introduction

Recently, a quasistatic boundary element method (BEM) solution has been proposed [1] that combines the adjoint double-layer formulation of the boundary element method [2–4] which utilizes surface charges at the boundaries, the zeroth-order (piecewise constant) basis functions with accurate near-field integration, and the FMM accelerator [5–7]. This approach does not require explicit forming of the BEM matrix; an iterative solution with M iterations requires $O(MN)$ operations. The fast multipole method speeds up computation of a matrix-vector product of a numerical iterative solution via the boundary element method (BEM) by many orders of magnitude. In the past, it was successfully applied for modeling high-frequency electromagnetic [8, 9] and acoustic [10–12] scattering problems. It has also been applied to modeling transcranial magnetic stimulation (TMS) and demonstrated a fast computational speed and superior accuracy for high-resolution head models as compared to both the standard boundary element method and the finite element method of various orders [1, 13, 14]. The rapid increase in the use of FMM in such numerical modeling schemes calls for an accurate and thorough study of the performance of FMM in a wide range of scenarios.

The goal of this study is to benchmark the performance (both speed and memory consumption) of the fast multipole method or FMM [5, 6]. Here, we will use the established head collection and its barycentrically refined versions to perform the profiling of the FMM library provided by Z. Gimbutas and L. Greengard [7] and employed in [1, 12]. Such profiling implies running the FMM for all head geometries at different frequencies including the static case and averaging the respective results. One FMM runtime essentially corresponds to one iteration step of an iterative

D. N. Pham (✉)
ECE Department, Worcester Polytechnic Institute, Worcester, MA, USA
e-mail: dnpham@wpi.edu

© The Author(s) 2021

S. N. Makarov et al. (eds.), *Brain and Human Body Modeling 2020*,
https://doi.org/10.1007/978-3-030-45623-8_21

BEM-FMM solution [1, 12]. Therefore, the data reported in the present study could be used to estimate the performance of a rather generic BEM-FMM algorithm if the number of iterations is approximately known or could be estimated a priori.

2 Materials and Methods

2.1 FMM Library of 2017

The core FMM algorithm is taken from the FMM library provided by Gimbutas and Greengard [7]. The latest version, last updated on November 8, 2017, is downloaded from the GitHub database to use in this study. We focus specifically in the function *fmm3d* which is used to solve Laplace and Helmholtz equations for a large number of target points. The compiled MEX versions of this function, namely, *fmm3d.mexw64* and *fmm3d.mexw64* for MATLAB compatibility in Windows and Linux, respectively, are used for all FMM calculations within the MATLAB environment. Depending on whether a solution for the Laplace or Helmholtz equation is desired, a wrapper function, either *lfmm3dpart* or *hfmm3dpart* – both available in the FMM library, is employed. A sample MATLAB command that calls *hfmm3dpart* to compute the Helmholtz equation is given by the following:

```
[U]=hfmm3dpart(iprec,k,nsource,source,ifcharge,charge,ifdipole,
dipstr,dipvec,ifpot,iffld,ntarget,target,ifpottarg,iffldtarg)
```

In the command above, the inputs parameters are as follows:

- `iprec`: precision flags for FMM
- `k`: wave number (Helmholtz parameter)
- `nsource`: number of source points
- `source`: source locations
- `ifcharge`: charge flag
- `charge`: charge values
- `ifdipole`: dipole flag
- `dipstr`: dipole magnitudes
- `dipvec`: dipole orientations
- `ifpot`: potential flag
- `iffld`: filed flag
- `ntarget`: number of targets
- `target`: target locations
- `ifpottarg`: target potential flag
- `iffldtarg`: target field flag

The output parameter is the struct U that contains the following fields:

- `U.pot:` the computed potential at source locations
- `U.fld:` the field at source location
- `U.pottarg:` potential at target locations
- `U.fldtarg:` field at target locations

In a similar manner, a sample MATLAB command that calls *lfmm3dpart* to compute the Laplace equation is given by the following:

```
[U]=lfmm3dpart(iprec,nsource,source,ifcharge,charge,ifdipole,
dipstr,dipvec,ifpot,iffld,ntarget,target,ifpottarg,iffldtarg)
```

where the input and output variables are similar to that of *hfmm3dpart*, except that for *lfmm3dpart* there is no wave number *k*. A more recent FMM library, developed by Flatiron Institute [15], is also investigated and compared with the library provided by Gimbutas and Greengard [7].

2.2 CAD Human Head Models

Every CAD human head model [8] has seven objects: the skin, skull, CSF, GM, cerebellum, WM, and ventricles head compartments. The models have an "onion" topology: the gray matter shell is a container for white matter, ventricles, and cerebellum objects; the CSF shell contains the gray matter shell; the skull shell contains the CSF shell; and finally, the skin or scalp shell contains the skull shell. The models have an average of 866,000 triangular facets and an average triangle quality of 0.25. The average edge length is 1.48 mm, and the average surface mesh density or resolution is 0.57 points per mm^2. A sample image of such a head model is shown in Fig. 1. Finer meshes with ~3,464,000 facets, obtained through one iteration of subdivision on the original CAD models, are also obtained for more intensive examinations on the scaling of timings and hardware resources.

2.3 Hardware Information

Windows server:

- 2 CPUs: Intel(R) Xeon(R) CPU E5–2683 v4 at 2.10GHz, 16 cores, 32 logical processors
- Physical memory (RAM): 256 GB
- OS: Microsoft Windows Server 2008 R2 Enterprise

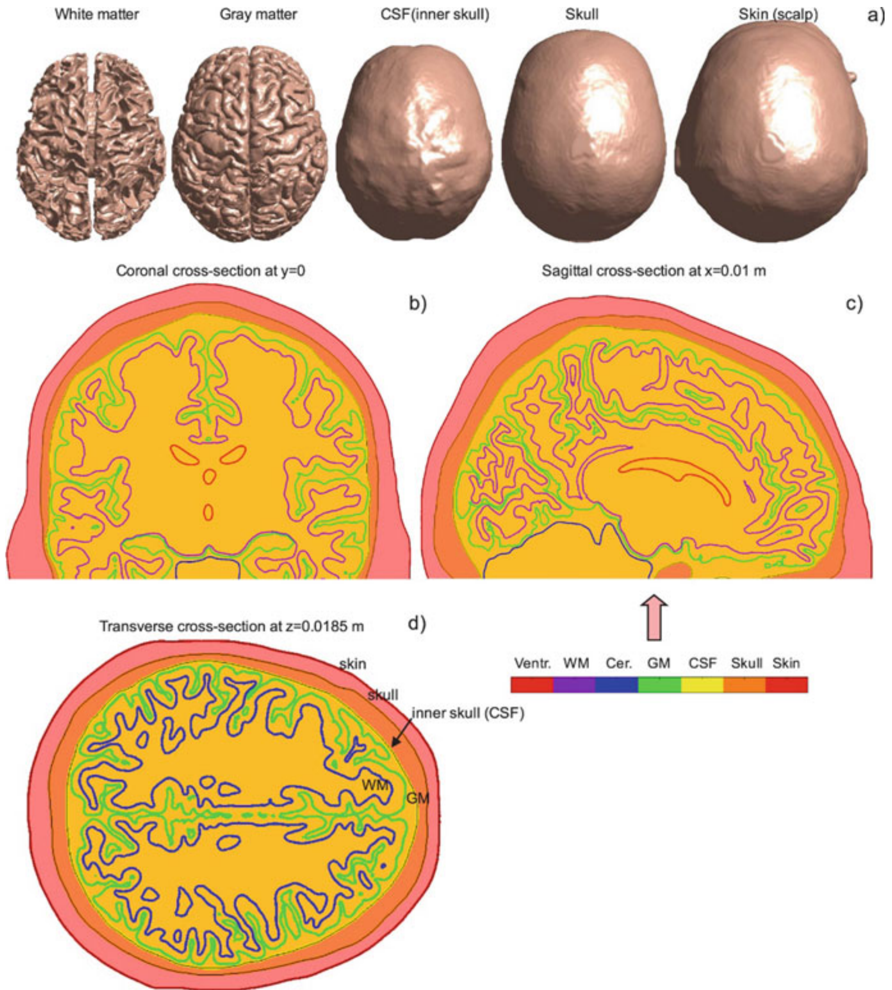


Fig. 1 Compartments of a sample brain model used in the testing of FMM software. (Image adapted from Htet et al. [8])

Linux server:

- 2 CPUs: Intel(R) Xeon(R) CPU E5–2690 0 at 2.90GHz, 64 bits
- Physical memory (RAM): 192 GB
- OS: Red Hat Enterprise Linux Server release 7.5 (Maipo)

2.4 Charge Assembly

For each of the 16 CAD models, a set of monopole charges are distributed over the surfaces of the triangular mesh so that at each triangle centers, a charge of random

magnitude q is assigned. The electric potential generated at each triangle centroids, excluding self-contribution from the local charge, is given by the following:

$$\varphi(\boldsymbol{r}) = \sum_i \frac{1}{4\pi\varepsilon_0} \frac{q}{|\boldsymbol{r}-\boldsymbol{r}_i|} e^{-jk|\boldsymbol{r}-\boldsymbol{r}_i|} \tag{1}$$

where ε_0 is permittivity of vacuum, q is electric charge of the source, k is the wave number, $\boldsymbol{r}$ is the target location at which the potential is sought, and $\boldsymbol{r}_i$ is the source location. The resultant electric field is given by the following:

$$\boldsymbol{E}(\boldsymbol{r}) = -\nabla\varphi = \sum_i + \frac{q}{4\pi\varepsilon_0} \left(\frac{\boldsymbol{r}-\boldsymbol{r}_i}{|\boldsymbol{r}-\boldsymbol{r}_i|^3} + jk \frac{\boldsymbol{r}-\boldsymbol{r}_i}{|\boldsymbol{r}-\boldsymbol{r}_i|^2} \right) e^{-jk|\boldsymbol{r}-\boldsymbol{r}_i|} \tag{2}$$

As a measure of FMM's performance, both the potential and the electric field, given by Eqs. (1) and (2), *are computed for all models, at the same triangle centroids, and excluding the self-contribution*. Through the function *hfmm3dpart* (and *lfmm3dpart* for the case ka = 0), the potential and the field are obtained simultaneously. With each head models, the calculations are done for three levels of accuracy:

- 2 digits (iprec = 0)
- 3 digits (iprec = 1)
- 6 digits (iprec = 2)

The frequencies for which the FMM algorithm is tested span over a wide range, which corresponds to ***ka*** values varying from 0 to 500. Here, ***a*** is the maximum of the x, y, and z coordinates of the model. Average value of ***a*** is 107.5754.

3 Results

3.1 Windows Platform (FMM 2017)

3.1.1 Original CAD Models

The relationship between runtimes of FMM calculations on Windows server, averaged over all 16 models, and *ka* is shown in Figs. 2, 3 and 4, with precisions 0 (two digits), 1 (three digits), and 2 (six digits), respectively. The discrete step for values of *ka* is 50, starting from $ka = 0$ and ending at $ka = 500$, with a more refined resolution within the low-frequency domain, from 0 to 50, where the step is 2.5. As can be seen in the insets in these figures, where the plot for low-frequency domain is magnified, there is always a sharp jump from the runtime for $ka = 0$ (Laplace case) to the very next value $ka = 2.5$. After the abrupt jump, FMM time increases steadily in a linear manner within the small *ka* domain (low frequencies) before growing exponentially

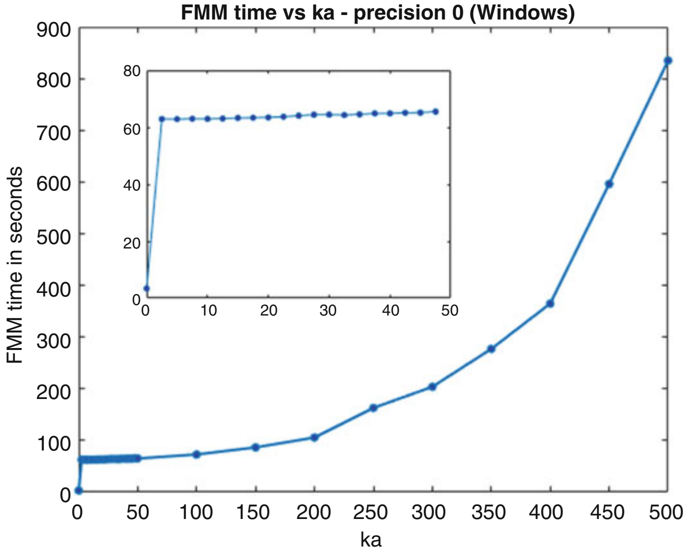


Fig. 2 FMM runtime within MATLAB platform (averaged over all sixteen heads) on Windows vs *ka*. The demanded precision is two digits accuracy. The Laplace case takes on average 3.41 s to complete with precision 0 (two digits)

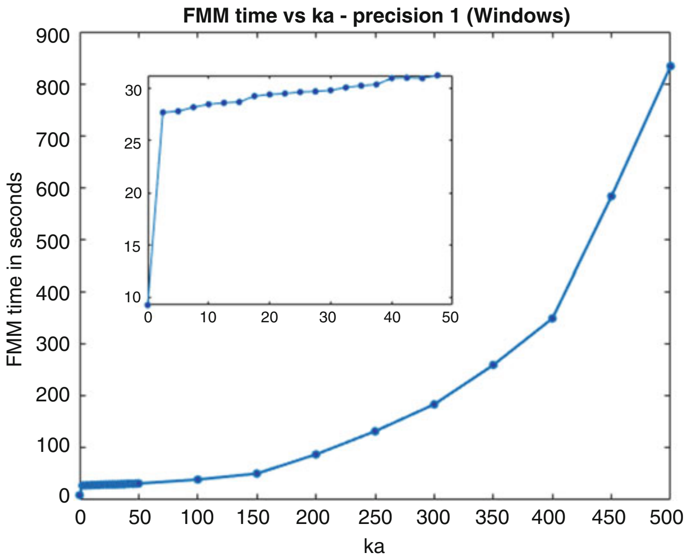


Fig. 3 FMM runtime within MATLAB platform (averaged over all sixteen heads) on Windows vs ka. The demanded precision is three digits accuracy. The Laplace case takes on average 9.23 s to complete with precision 1 (three digits)

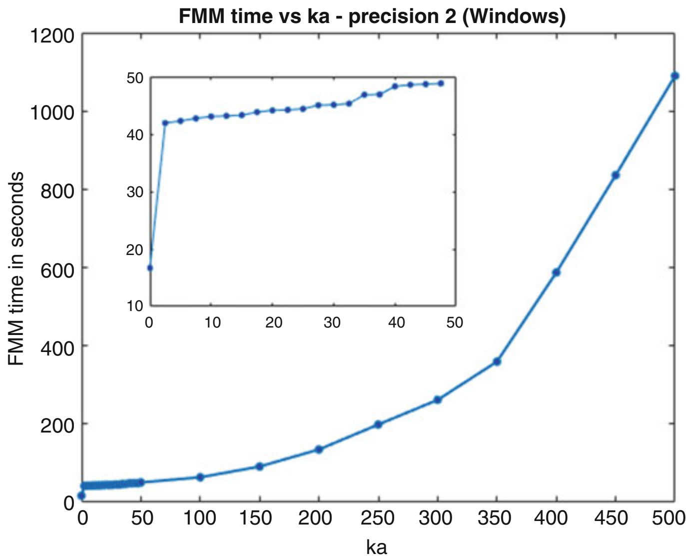


Fig. 4 FMM runtime within MATLAB platform (averaged over all sixteen heads) on Windows vs ka. The demanded precision is six digits accuracy. The Laplace case takes on average 16.70 s to complete with precision 3 (six digits)

at medium and large *ka*. The slope of the time-*ka* dependence in low *ka* with precision 0 (Fig. 2) is $(6.04 \pm 0.19) \times 10^{-2}$, whereas with precision 1, the slope is $(7.86 \pm 0.13) \times 10^{-2}$, and with precision 2, it is $(16.81 \pm 1.03) \times 10^{-2}$. From these numerical estimations, it can be concluded that the higher the demanded precision is, the steeper the time-*ka* slope becomes, and the runtime increases with *ka* in a higher rate.

In Fig. 5, the FMM runtimes for all three precision choices are plotted. *For the Laplace case (ka = 0), it takes on average 3.41 s for the computations to complete with precision 0. If higher level of accuracy is requested, the time taken increases to 9.23 s with precision 1 and 16.70 with precision 2.* This trend, however, is not replicated in the Helmholtz case, particularly at the low-frequency domain. As shown in Figs. 2 and 5, in the small *ka* domain, except for $ka = 0$, FMM is longest with precision 0, the lowest level of accuracy of all. More specifically, with precision 0, FMM runtime increases from an average of 62.88 s at $ka = 2.5$ to 65.33 s at $ka = 50$ (Fig. 1 or 4). Precision 2, the highest accuracy level tested, only takes the second longest amount of time, with 41.92 s for $ka = 2.5$, and rises to 50.77 s at $ka = 50$. Calculations within the low-frequency domain are fastest with precision 1, as it only takes 27.73 s to finish calculating for $ka = 2.5$ and 31.67 s for $ka = 50$. This rather unexpected behavior continues as far as $ka = 150$, where FMM time with precision 2, due to its rapid exponential rise, surpasses the runtime of precision 0. Toward the high end of the frequency range, precision 1 runtime, whose exponential rate is also higher than precision 0 (but not as high as 2), starts approaching before surpassing precision 0's runtime at $ka = 500$. Therefore, at very large values for *ka*, a more intuitively expected trend is observed, where FMM runtime with

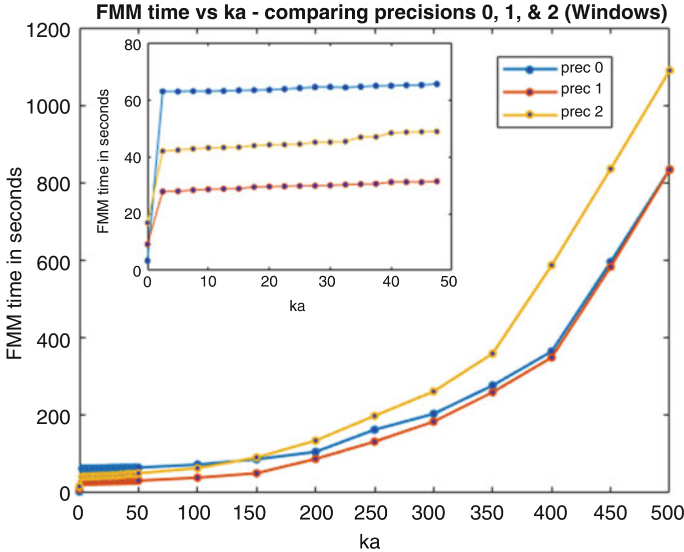


Fig. 5 Comparison among average FMM runtimes within MATLAB platform on Windows for when two, three, and six digits accuracy are demanded

precision 0 is lowest and has the slowest exponential rise as *ka* increases, followed by precision 1, and lastly, precision 2 is most time-consuming and has the quickest exponential rate.

3.1.2 First-Order Mesh Subdivision

In Figs. 6, 7 and 8, FMM runtimes on Windows server with precisions 0, 1, and 2, respectively, averaged over all 16 refined meshes obtained through one iteration of barycentric subdivision done on the original CAD models, are presented. Average mesh size quadruples; it is now 3.464 M facets. *Similar to when the calculations were done on the original head models, for ka = 0 (Laplace case), precision 0 takes the least amount of time, 10.41 s, compared to 41.50 s with precision 1 and 70.86 s with precision 2.* Also similar to the original head models, there are abrupt jumps in runtime from the Laplace case to the Helmholtz calculations, as shown in Figs. 6, 7 and 8. Within low-frequency limit, FMM runtime increases linearly with ka. The higher the requested accuracy is, the steeper the slope is; with precision 0, FMM time increases at the linear rate of $(11.02 \pm 1.90) \times 10^{-2}$ for ka in the low-frequency domain (0–50), while FMM time for calculations done with precision 1 increases at a higher rate, $(21.30 \pm 0.61) \times 10^{-2}$, and precision 2 calculation time increases most rapidly with the rate $(39.21 \pm 6.75) \times 10^{-2}$.

In Fig. 9, the runtimes of FMM applied to the first-order-refined meshes with all three precision choices are plotted. Again, within the low-frequency domain, precision 0 does not yield the fastest runtime. As shown in Fig. 9, at $ka = 2.5$, runtime

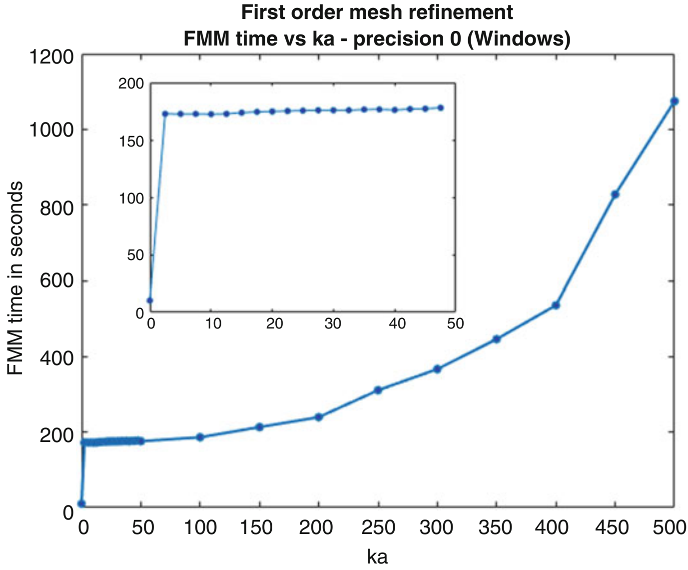


Fig. 6 FMM runtime for the refined models within MATLAB platform (averaged over all sixteen heads) on Windows vs ka. The demanded precision is two digits accuracy. The Laplace case takes on average 10.41 s to complete with precision 0 (two digits)

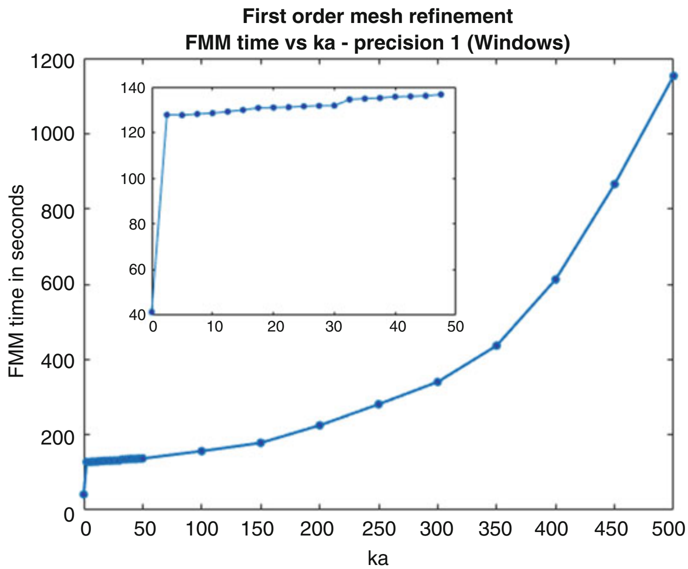


Fig. 7 FMM runtime for the refined models within MATLAB platform (averaged over all sixteen heads) on Windows vs ka. The demanded precision is three digits accuracy. The Laplace case takes on average 41.50 s to complete with precision 1 (three digits)

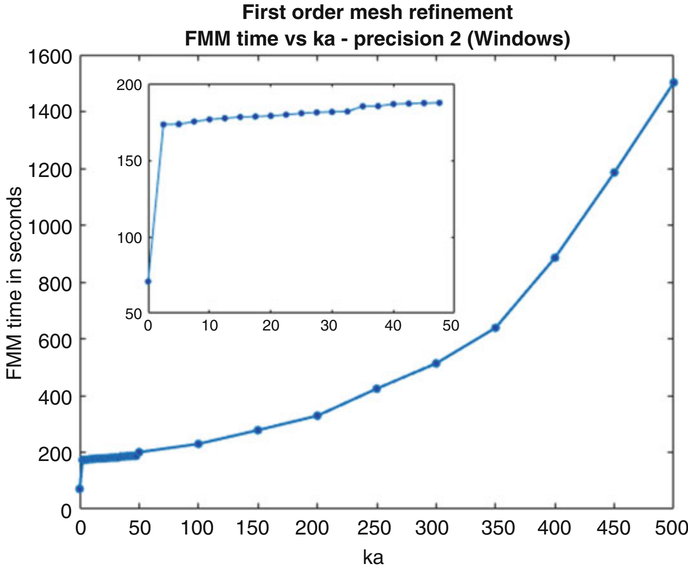


Fig. 8 FMM runtime for the refined models within MATLAB platform (averaged over all sixteen heads) on Windows vs ka. The demanded precision is six digits accuracy. The Laplace case takes on average 70.86 s to complete with precision 2 (six digits)

with precision 0 (172.62 s) is essentially comparable to precision 2 (173.26 s), and both are significantly slower than precision 1 (127.76 s). As *ka* increases, out of the three options, precision 0 has its runtimes increase at the slowest rate. Therefore, by ka = 350, FMM runtimes of precision 0 is surpassed by precision 1, and from then on, its runtimes are quickest, followed by precision 1, and precision 2 takes up the most time.

On the scaling of FMM runtime from the original CAD models that have average of $N_0 = 866{,}000$ facets to first-order-refined meshes with $N_1 = 3{,}464{,}000$ facets, the theoretical factor is as follows:

$$S = \frac{N_1 \log N_1}{N_0 \log N_0} = 4.4 \tag{3}$$

In Fig. 10, throughout the tested range for *ka*, the time ratio for precision 0 is always the smallest out of the three choices for precision. Quite surprisingly, the ratio for precision 2, for most of the time, is smaller than that of precision 1. Also interestingly, all three precision options start at $ka = 0$ with scaling factors relatively close to the theoretical values (perhaps with an exception with precision 0) and then decrease significantly as *ka* increases. Therefore, it appears that the higher the frequency is, the better the scaling in runtime is.

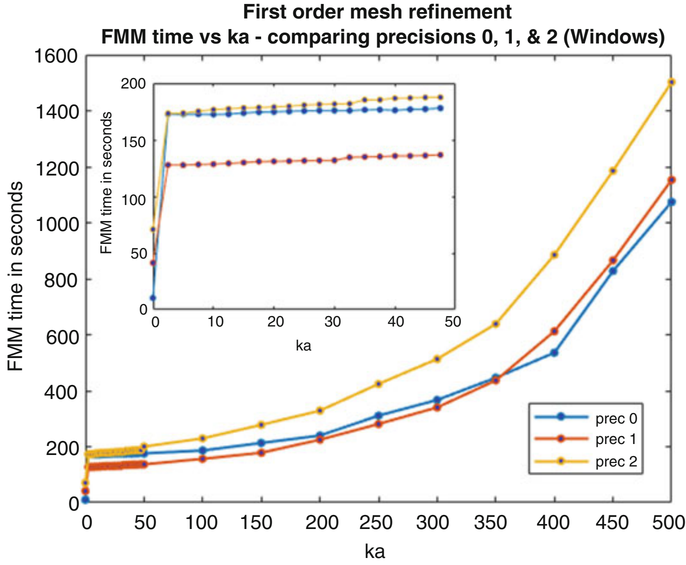


Fig. 9 Comparison among average FMM runtimes for the refined models in MATLAB on Windows for when two, three, and six digits accuracy is demanded

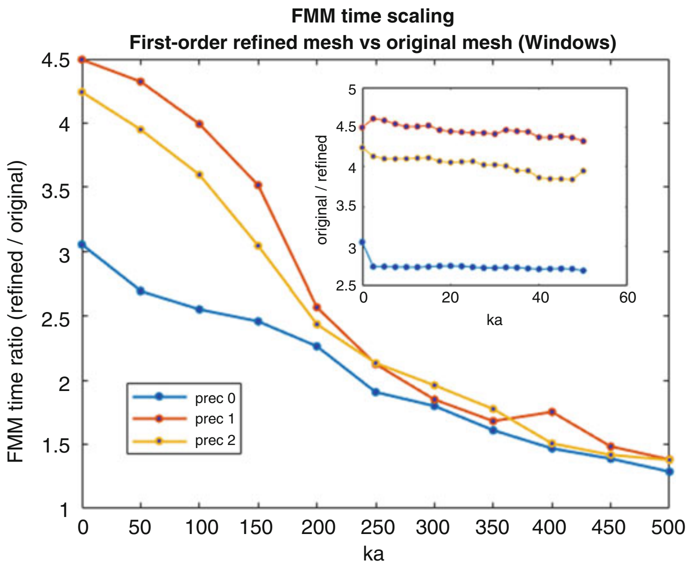


Fig. 10 Ratio between FMM time (Windows) for the refined models and FMM time for the original CAD models is plotted with respect to *ka*

3.2 Linux Platform (FMM 2017)

3.2.1 Original CAD Models

The relationship between runtimes of FMM calculations on Linux server, averaged over all 16 models, and *ka* is shown in Figs. 11, 12 and 13 with precisions 0, 1, and 2, respectively. A summary of runtimes for all three precisions is plotted in Fig. 14. Different from the same calculations done on the Windows platform, the runtime on Linux with precision 0, the lowest level of accuracy tested in this study, takes the least amount of time, while calculations with precision 1 are second, and precision 2, the highest level of accuracy with six digits, consumes the most amount of time. This order is held consistently throughout the entire frequency range from 0 to 500. It is also noticeable that runtimes with precisions 0 and 1, which guarantee accuracy within two and three digits, respectively, are comparable to each other, with calculations with precision 1 take slightly longer than 0. FMM runtime with precision 2, which demands six digits accuracy, takes significantly more time to finish. This rather intuitive behavior, however, is not present in the runtime profiling for Windows, which was discussed in Sect. 3.1. *As a note on how the two servers compare to each other in computing Laplace equation, it takes the Linux server 5.14 s to finish the calculation with precision 0, while for Windows, it is only 3.41 s. The Linux server, however, is notably faster on precision 1, taking 5.33 s to finish as compared*

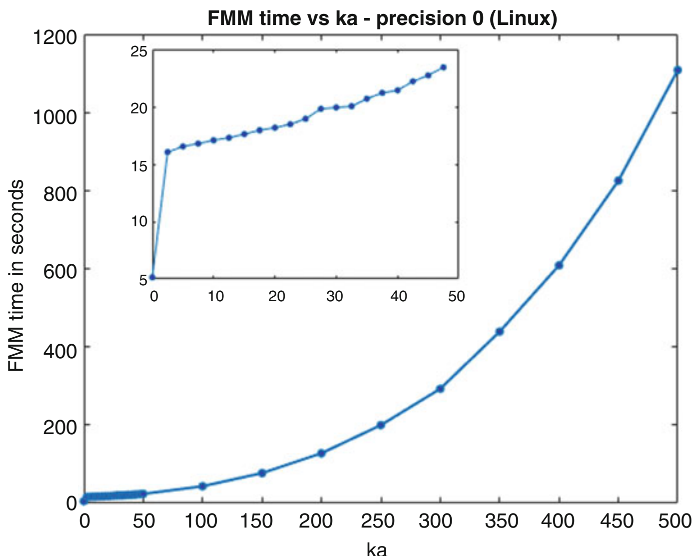


Fig. 11 FMM runtime within MATLAB platform (averaged over all sixteen heads) on Linux vs ka. The demanded precision is two digits accuracy. The Laplace case takes on average 5.14 s to complete with precision 0 (two digits)

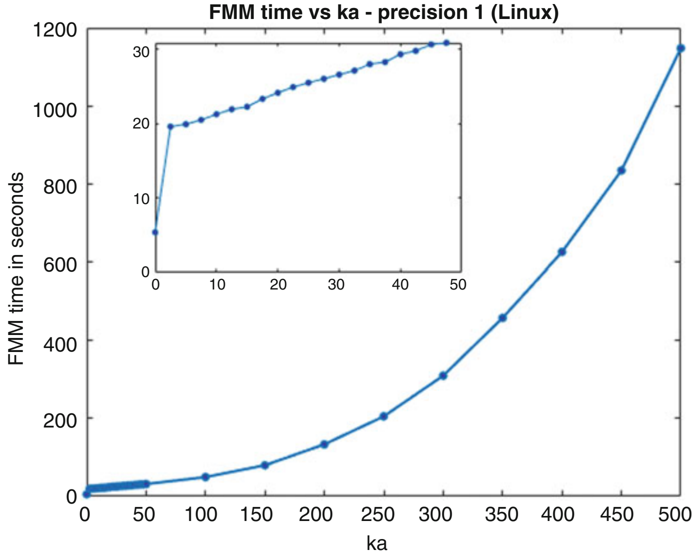


Fig. 12 FMM runtime within MATLAB platform (averaged over all sixteen heads) on Linux vs ka. The demanded precision is three digits accuracy. The Laplace case takes on average 5.33 s to complete with precision 1 (three digits)

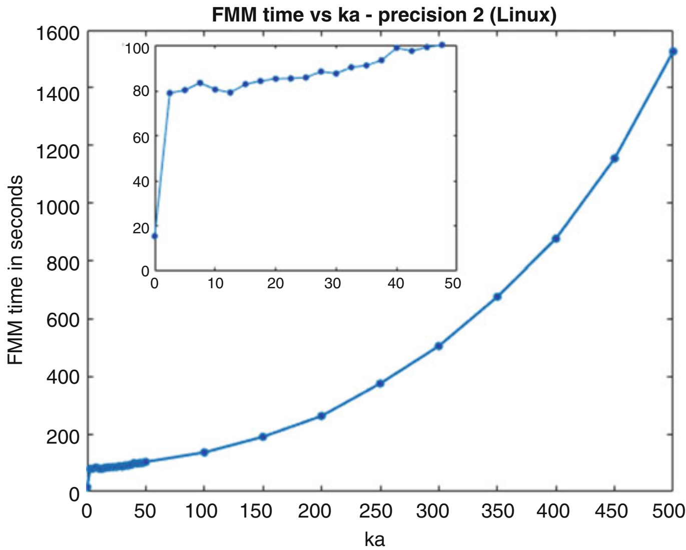


Fig. 13 FMM runtime within MATLAB platform (averaged over all sixteen heads) on Linux vs ka. The demanded precision is six digits accuracy. The Laplace case takes on average 15.31 s to complete with precision 2 (six digits)

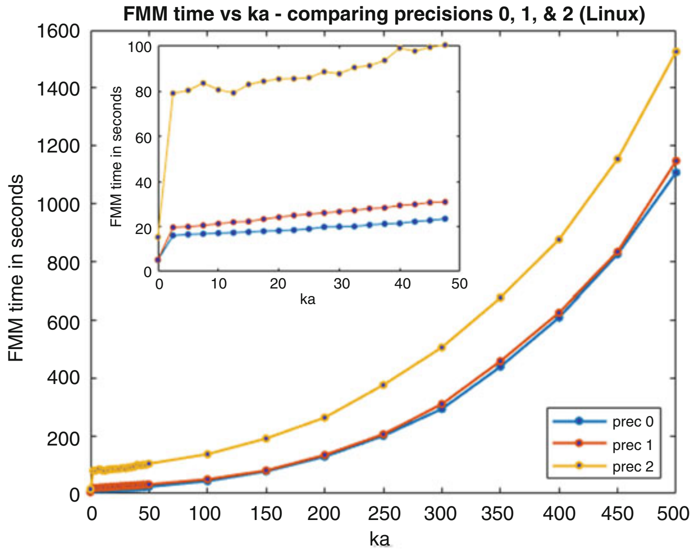


Fig. 14 Comparison among average FMM runtimes in MATLAB on Linux for when two, three, and six digits accuracy are demanded

to 9.23 s by the Windows platform. And finally, the Linux server is slightly better at precision 2, with 15.31 s, whereas Windows takes 16.70 s.

Comparing the calculations done on two platforms, Linux and Windows, we also observe major differences in how the runtime evolves as *ka* varies. First of all, FMM calculations within low-frequency domain are generally faster on Linux than on Windows, especially for low-to-medium accuracies (precisions 0 and 1). In particular, with precision 0, on Linux, it takes on average 19.51 s for *hfmm3dpart* to finish solving the Helmholtz equation on the original CAD models for *ka* within the range 0–50, whereas it takes on average 64.03 s to complete the same task on the Windows server. Similarly, it takes only 25.59 s on Linux to finish the calculations in the low-frequency domain with precision 1. Calculations on Windows, although not drastically slower than Linux as in the case of precision 0, still take 29.71 s to complete. If higher precisions are in demand, in fact, runtimes on Windows will catch up with Linux, and eventually, the speed on Windows will exceed. Evidently, for low-frequency calculations demanding precision 2 (six digits accuracy), it takes only 45.35 s for *hfmm3dpart* to complete computing, while a similar task takes the Linux server 88.88 s to complete.

Scaling of runtime as the frequency (or *ka*) is increased is another important metric. For the Linux server, within the range 0–50 for *ka*, FMM runtime increases linearly with the slope of $(16.06 \pm 0.31) \times 10^{-2}$ with precision 0. This is a significantly faster rate compared to the slope $(6.04 \pm 0.19) \times 10^{-2}$ (already mentioned in Sect. 3.1.1) for the same precision but on Windows. This comparison also holds with precisions 1 and 2, as on Linux the linear rates are $(26.20 \pm 0.07) \times 10^{-2}$ with precision 1 and a whopping $(50.09 \pm 3.81) \times 10^{-2}$

with precision 2. These values are far inferior than the rates $(7.86 \pm 0.13) \times 10^{-2}$ with precision 1 and $(16.81 \pm 1.03) \times 10^{-2}$ with precision 2 on Windows. Therefore, although the Linux server shows an edge over the Windows platform in computing low-frequency Helmholtz equation, the fact that runtimes on Linux increase too quickly with frequency makes it eventually get surpassed by Windows server at medium- and high-frequency domains. At $ka = 500$ (the highest value for ka tested in this study), runtimes on Windows are 834.69 s with precision 0, 833.91 s with precision 1, and 1090 s with precision 2, while on Linux, the numbers are 1108 s, 1148 s, and 1526 s, respectively.

3.2.2 First-Order Mesh Subdivision

In Figs. 15, 16, and 17, FMM runtimes on Linux server with precisions 0, 1, and 2, respectively, averaged over all 16 refined mesh (obtained through one iteration of barycentric subdivision done on the original CAD models), are presented. A summary of runtimes for all three precisions is plotted in Fig. 18. Comparing FMM done for the refined meshes on Linux and Windows, we obtain trends that are mostly similar to what was observed in the calculations done on the original models. *First, comparisons on Laplace calculations yield the same results: Linux with precision 0 takes 16.41 s, considerably slower than Windows, which takes only 10.41 s. For precision 1, the time is 17.7 s on Linux, again significantly better than Windows'*

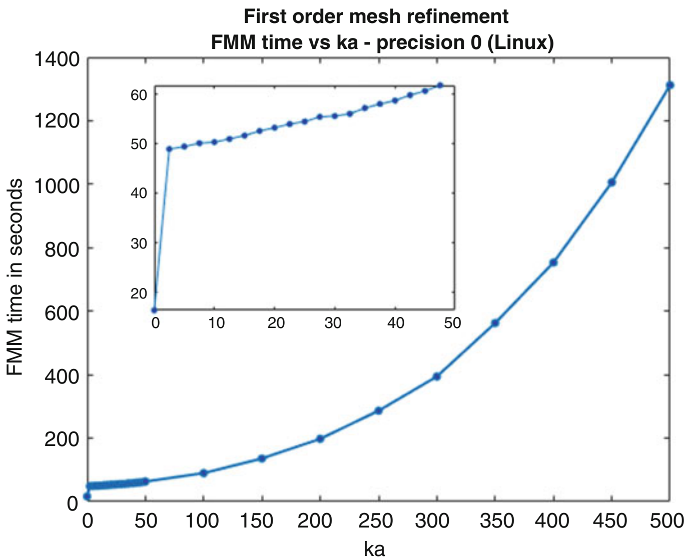


Fig. 15 FMM runtime for the refined models within MATLAB platform (averaged over all sixteen heads) on Linux vs ka. The demanded precision is two digits accuracy. The Laplace case takes on average 16.41 s to complete with precision 0 (two digits)

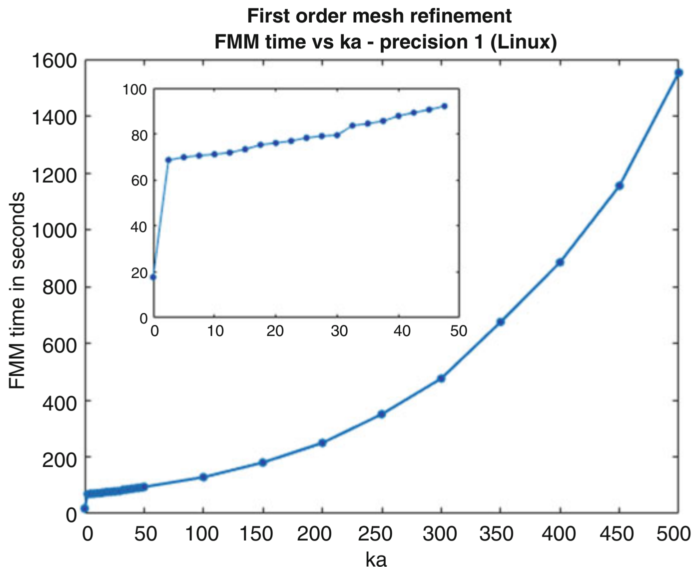


Fig. 16 FMM runtime for the refined models within MATLAB platform (averaged over all sixteen heads) on Linux vs ka. The demanded precision is two digits accuracy. The Laplace case takes on average 17.7 s to complete with precision 1 (three digits)

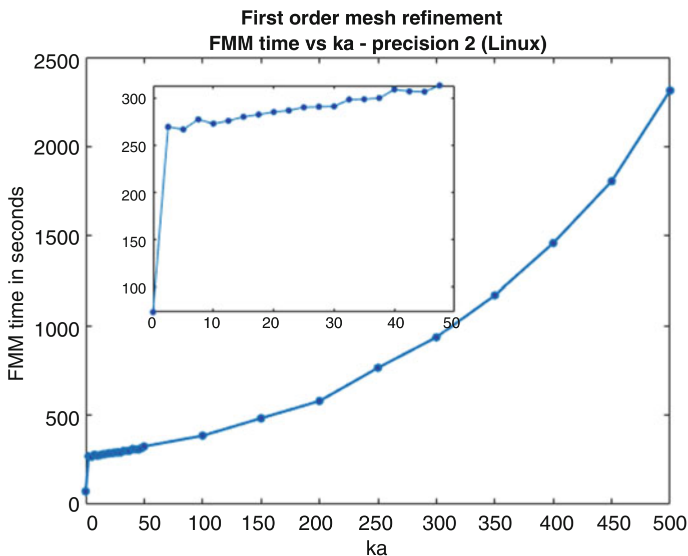


Fig. 17 FMM runtime for the refined models within MATLAB platform (averaged over all sixteen heads) on Linux vs ka. The demanded precision is two digits accuracy. The Laplace case takes on average 73.29 s to complete with precision 2 (six digits)

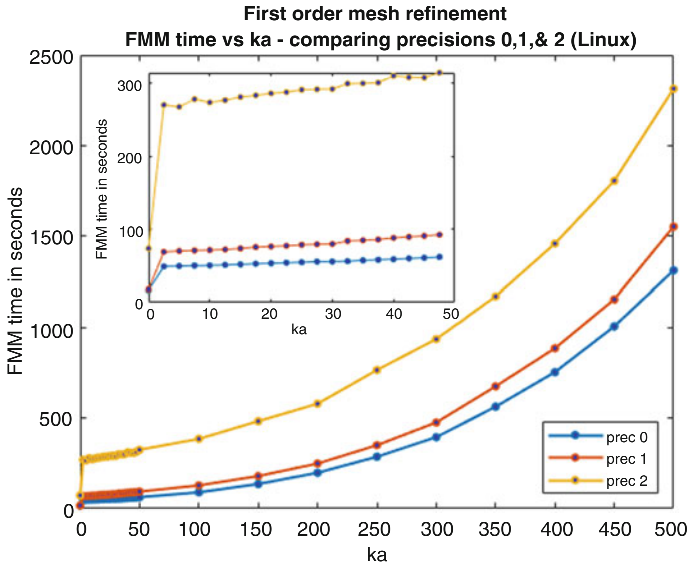


Fig. 18 Comparison among average FMM runtimes for the refined models in MATLAB on Linux for when two, three, and six digits accuracy are demanded

41.50 s, and finally, runtimes for precision 2 of the two platforms are comparable, 73.29 s for Linux and 70.86 s for Windows.

In terms of how runtimes of the three accuracy options compare to each other, as can be seen in Fig. 18, precision 0 takes the least amount of time, tightly followed by precision 1, while precision 2 is a lot more time-consuming. The same behavior was already discussed in Sect. 3.2.1 for calculations done with the original CAD models on Linux. The same conclusion, however, cannot be drawn for calculations done on Windows, as mentioned in Sects. 3.1.1 and 3.1.2.

The rates at which FMM runtimes increase with *ka* are significantly higher on Linux than on Windows. Within the low-frequency domain, where the FMM time-*ka* dependence appears to be linear, FMM runtime (with refined meshes) for precision 0 on Linux has the linear rate of $(28.90 \pm 0.45) \times 10^{-2}$, by a large margin higher than the rate $(11.02 \pm 1.9) \times 10^{-2}$ on Windows for the same precision level. Similarly, precision 1's runtimes increase at the rate $(53.91 \pm 0.90) \times 10^{-2}$ in the low-frequency range, while on windows, it is only $(21.30 \pm 0.61) \times 10^{-2}$. And finally, for precision 2, the rate is $(104.5 \pm 4.30) \times 10^{-2}$ on Linux and $(39.21 \pm 6.75) \times 10^{-2}$. Such steep slopes on the runtime-ka dependence in the low-frequency domain of the Linux platform are continued by the rapid exponentiations in the medium- and high-frequency ranges, which result in the Linux server being far inferior to Windows in computing the Helmholtz equations in high-frequency domain.

In Fig. 19, the actual ratio between FMM time (on Linux) of the refined models and the original model is plotted. Unlike the time ratio plot for Windows (Fig. 10), here, we see a more expected trend; the time ratio for precision 0 is lowest, followed

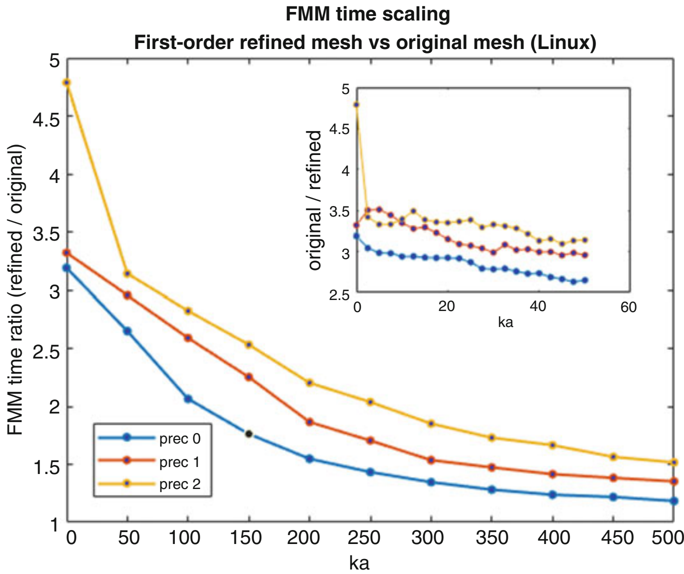


Fig. 19 Ratio between FMM time (Linux) for the refined models and FMM time for the original CAD models is plotted with respect to ka

by precision 1, while precision 2 has the largest ratio, and this behavior is maintained over the entire range *ka* = 0–500. As also shown in Fig. 19, except for only the Laplace case precision 2, all the ratios of the three precisions are below the theoretical scaling factor (see Eq. (3)). As *ka* increases, a decreasing trend is observed for all three plots. A similar result can be seen in Fig. 10 for Windows.

3.3 Memory Requirements

3.3.1 Original CAD Models

In this section, we discuss the memory consumed by FMM. Due to limitations on tools available, as well as Windows' uncompromising memory recording scheme, only memory information for calculations done on Linux is profiled and analyzed here. However, given the same FMM task, the (approximately) same amount of memory consumption is expected in both platforms. Therefore, valuable insights in memory requirements for performing FMM on Windows can still be drawn. In Figs. 20, 21, and 22, peak physical memory over FMM runtime is plotted with *ka* for precisions 0, 1, and 2, respectively. In Fig. 23, a summary of memory vs *ka* is plotted for all three precision choices. As can be seen in the figures, the general shapes of the curves are similar to the runtime plots displayed in previous sections; there is an abrupt jump from memory needed for solving Laplace equation to the Helmholtz

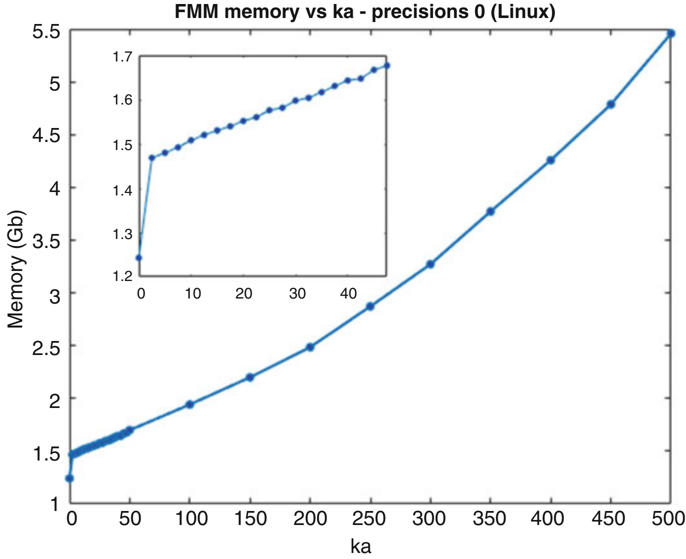


Fig. 20 Average peak memory consumption in MATLAB on Linux is plotted with respect to values of ka. The demanded precision is two digits accuracy

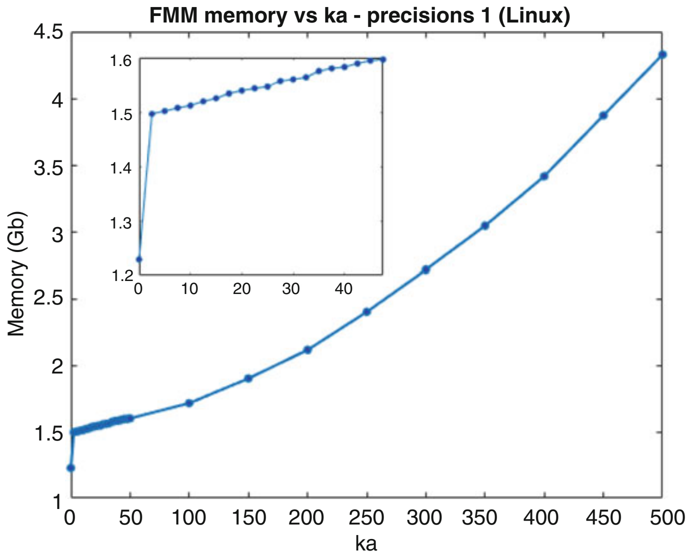


Fig. 21 Average peak memory consumption in MATLAB on Linux is plotted with respect to values of ka. The demanded precision is three digits accuracy

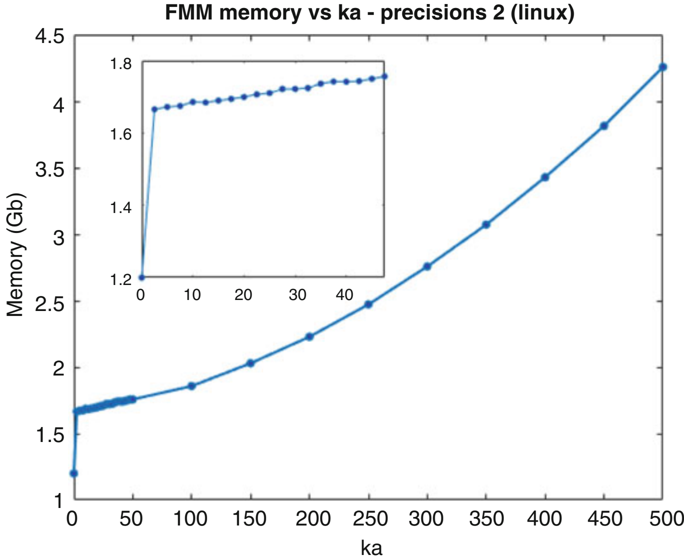


Fig. 22 Average peak memory consumption in MATLAB on Linux is plotted with respect to values of ka. The demanded precision is six digits accuracy

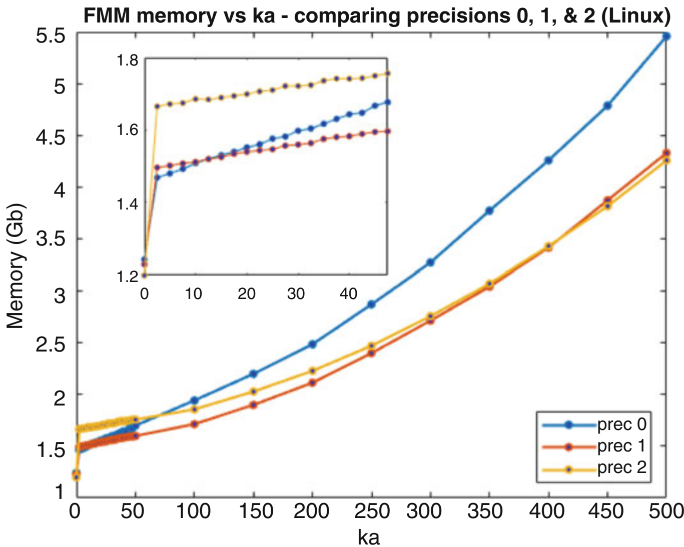


Fig. 23 A comparison among average peak memory consumption in MATLAB on Linux for when two, three, and six digits accuracy are demanded

case. For the Laplace case, calculations with all three levels of precisions require roughly the same amount of memory (approximately 1.2 Gb). Within the

low-frequency domain, the memory-*ka* dependence is linear before evolving into an exponential growth in higher frequency ranges.

Perhaps the most astonishing results are that precision 0, the lowest level of accuracy, requires the most amount of memory, especially at high-frequency domain. As shown in Fig. 23, starting with very small values of ka, calculations for precision 0 consume the least amount of memory. However, both its linear rate within the low-frequency range and its exponential rate in the higher-frequency domain exceed that of precisions 1 and 2, resulting in the memory needed for precision 0 to somehow outgrow the supposedly more computationally demanding precision options. The memory plots for precisions 1 and 2, on the other hand, evolve in a more relaxed manner and, over the entire ka range from 0 to 500, tend to stay close to each other.

3.3.2 First-Order Mesh Subdivision

In Figs. 24, 25 and 26, peak physical memory over FMM runtime (performed on refined meshes) is plotted with *ka* for precisions 0, 1, and 2, respectively. In Fig. 27, a summary of memory vs *ka* is plotted for all three precision choices. Similar to the memory recorded for FMM done on the original meshes, here, we again observe that it is precision 0 that consumes the most memory, particularly at high frequencies (Fig. 28).

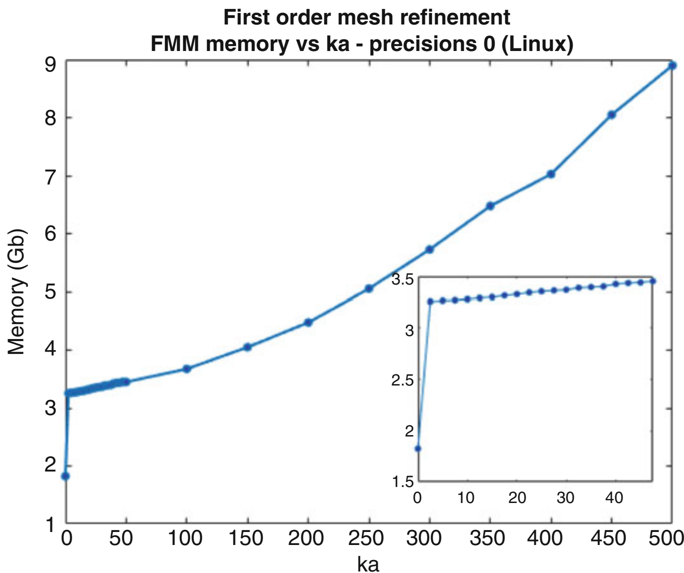


Fig. 24 Average peak memory consumption in MATLAB for the refined models on Linux is plotted with respect to values of ka. The demanded precision is two digits accuracy

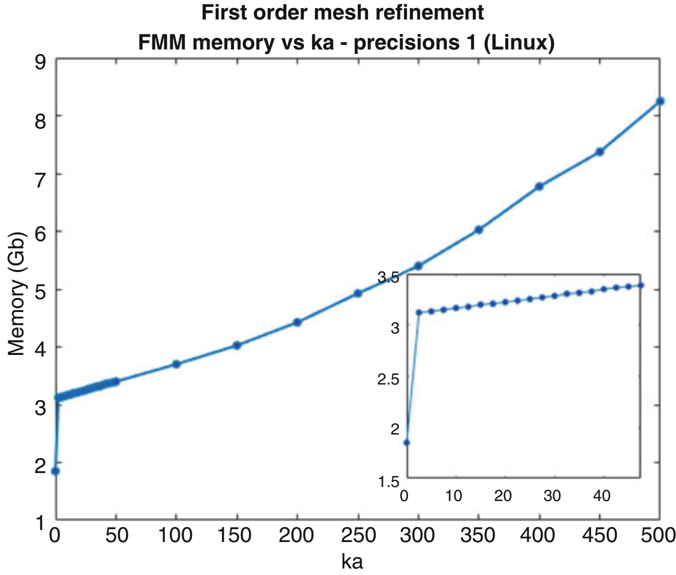


Fig. 25 Average peak memory consumption in MATLAB for the refined models on Linux is plotted with respect to values of ka. The demanded precision is three digits accuracy

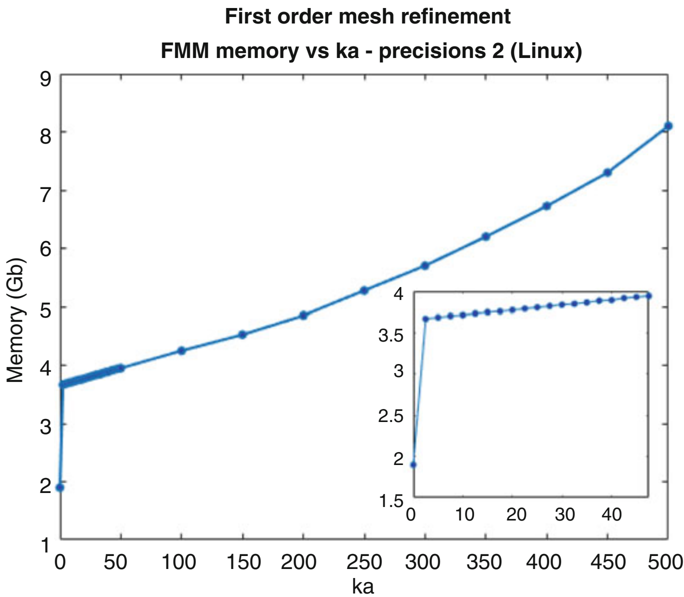


Fig. 26 Average peak memory consumption in MATLAB for the refined models on Linux is plotted with respect to values of ka. The demanded precision is six digits accuracy

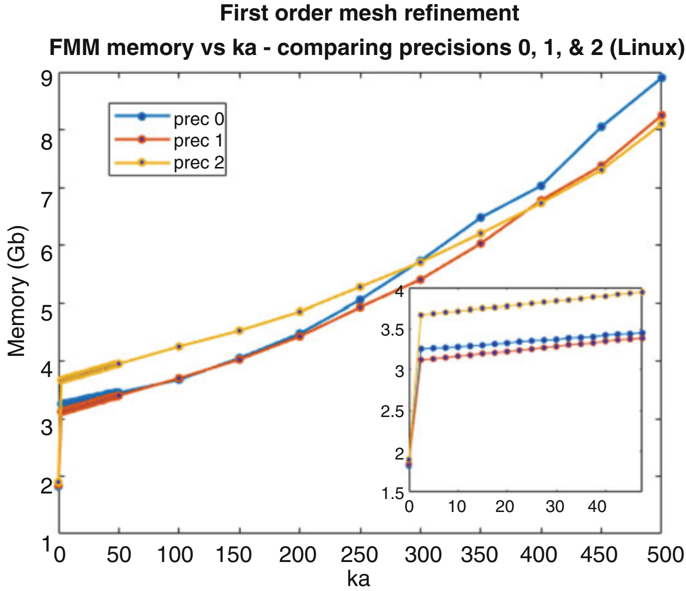


Fig. 27 A comparison among average peak memory consumption in MATLAB for the refined models on Linux for when two, three, and six digits accuracy are demanded

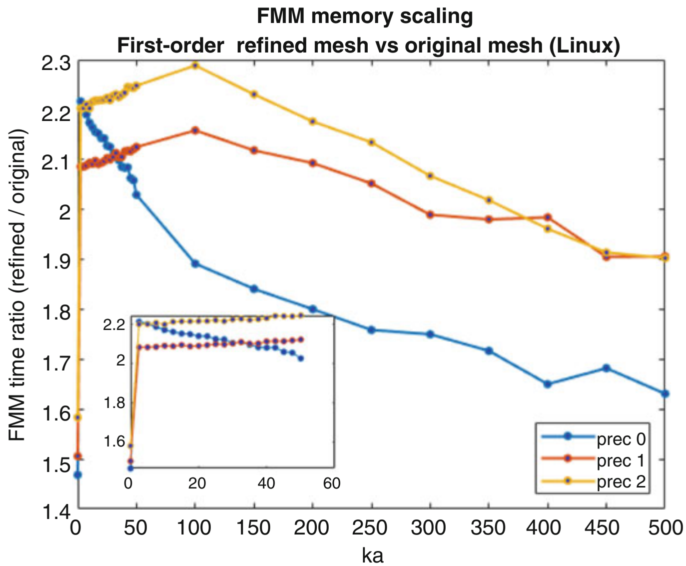


Fig. 28 Ratio between memory requirement (Linux) for the refined models and FMM time for the original CAD models is plotted with respect to ka

3.4 Second-Order Mesh Refinement (FMM 2017)

3.4.1 Windows Platform

In this section, we study the performance of FMM in calculations that use CAD models that have more refined meshes. These models have an average of 13,800,000 triangles and are obtained by performing two levels of barycentric subdivisions on the original head models. In Figs. 29, 30, and 31, FMM runtimes on Windows server with precisions 0, 1, and 2, respectively, averaged over all 16 refined meshes, are presented. In Fig. 32, a summary of runtime vs ka is plotted for all three precision choices. For the studies in this section, due to limited resources in computation, we restrict ourselves with ka values only from 0 to 50. As seen in Figs. 29 and 32, the runtimes for dense meshes are rather unpredictable, as there are no observable patterns for how the runtime of FMM evolves when ka is increased from 0 to 50. This volatility can be seen in the plots for all precisions 0, 1, and 2. In Fig. 33, the ratio between FMM time for the refined models (second level of mesh refinement) and FMM time for the original CAD models is plotted with respect to ka. As seen in Fig. 33, the scaling is best for calculations that require precision 0, followed by precision 2. FMM calculations for precision 1 has the largest scaling factor. This unintuitive scaling result was also seen for first-order mesh refinement (see Fig. 9) and was discussed in Sect. 3.1.2.

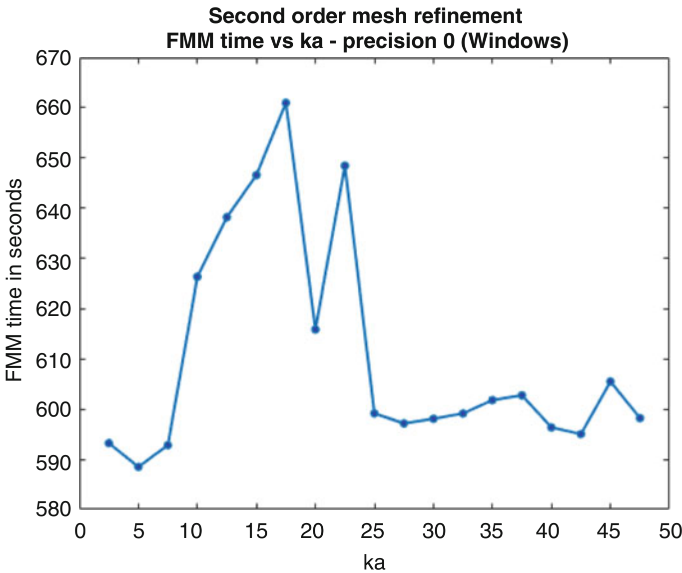


Fig. 29 Average FMM runtime for the doubly refined models in MATLAB on Windows is plotted with respect to values of ka. The demanded precision is two digits accuracy

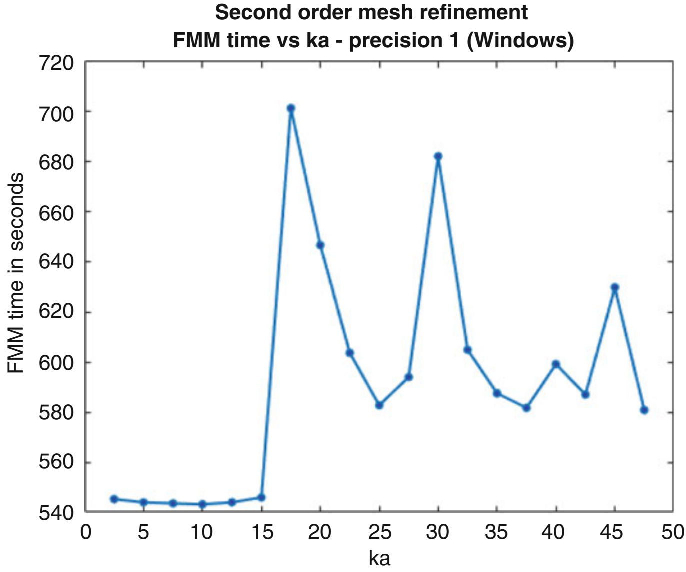


Fig. 30 Average FMM runtime for the doubly refined models in MATLAB on Windows is plotted with respect to values of ka. The demanded precision is three digits accuracy

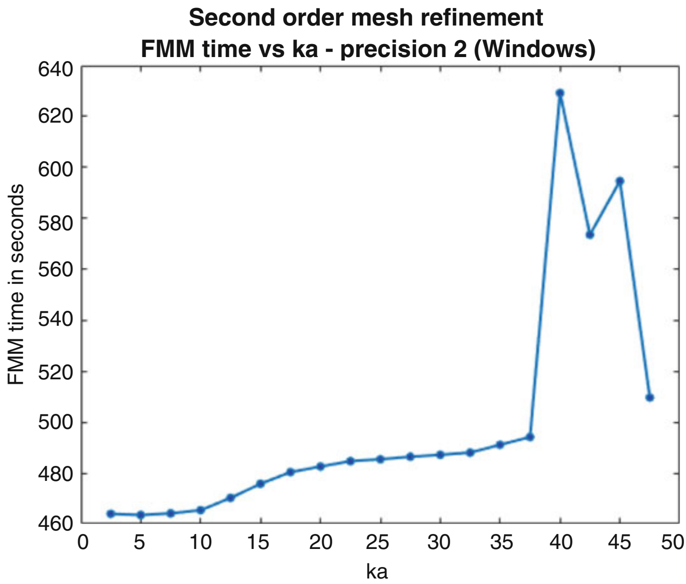


Fig. 31 Average FMM runtime for the doubly refined models in MATLAB on Windows is plotted with respect to values of ka. The demanded precision is six digits accuracy

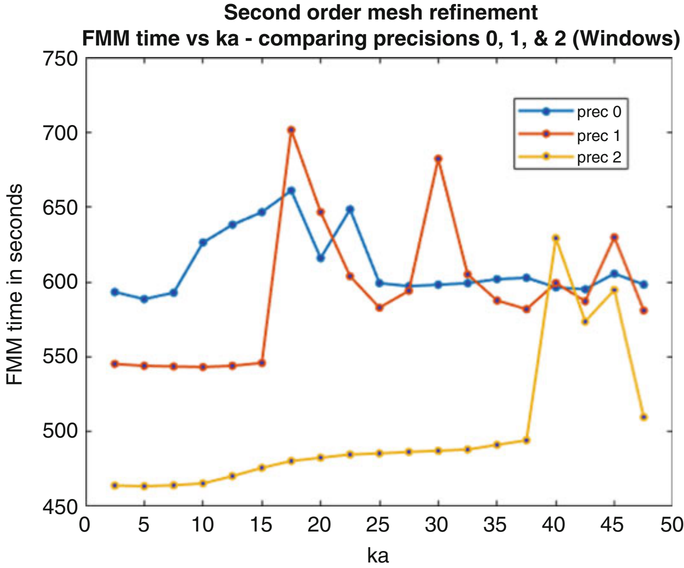


Fig. 32 A comparison among average FMM runtimes for the doubly refined models in MATLAB on Windows for when two, three, and six digits accuracy are demanded

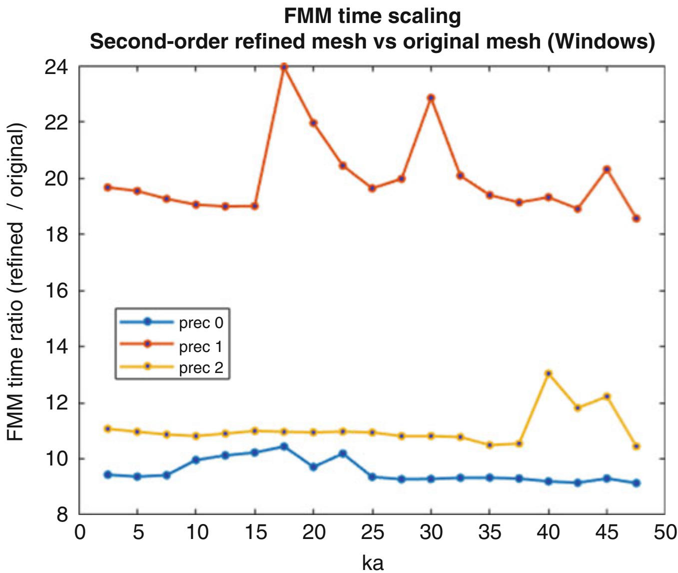


Fig. 33 Ratio between FMM time (Windows) for the doubly refined models and FMM time for the original CAD models is plotted with respect to ka

3.4.2 Linux Platform

In Figs. 34, 35, and 36, FMM runtimes on Linux server with precisions 0, 1, and 2, respectively, averaged over all 16 refined meshes (second order), are presented. In Fig. 37, a summary of runtime vs ka is plotted for all three choices of precision. Unlike the volatile behavior seen in the results for Windows, the FMM runtimes in Linux for meshes of second-order refinement increase linearly as ka increases, as expected. In Fig. 38, the ratio between FMM time for the refined models (second level of mesh refinement) and FMM time for the original CAD models is plotted with respect to ka. Again, we see that the scaling for precision 0 is lowest, while precision 1 has the highest scaling factor.

3.5 Comparisons with the New FMM Package (Summer 2019)

Very recently, a new version of the FMM library was published by the Flatiron Institute [15]. This package was downloaded for testing on June 11, 2019. Shortly after that, the online library was updated; this newer version was downloaded on June 20, 2019. In this section, we compare the performances of these two new versions of the FMM software with the original FMM library by Gimbutas and Greengard [7], last updated on November 8, 2017. Here, we focus on the

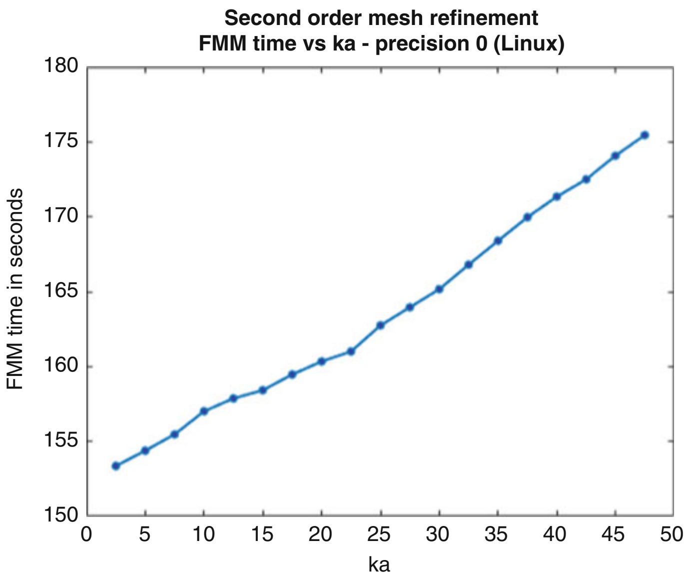


Fig. 34 Average FMM runtime for the doubly refined models in MATLAB on Linux is plotted with respect to values of ka. The demanded precision is two digits accuracy

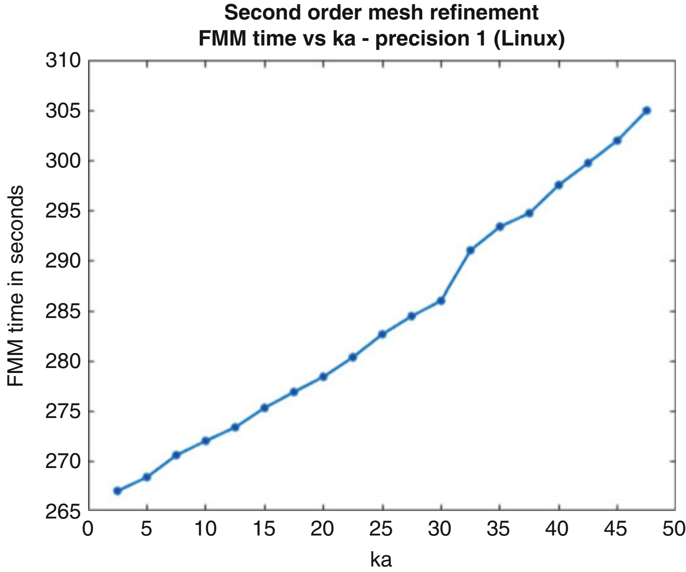


Fig. 35 Average FMM runtime for the doubly refined models in MATLAB on Linux is plotted with respect to values of ka. The demanded precision is three digits accuracy

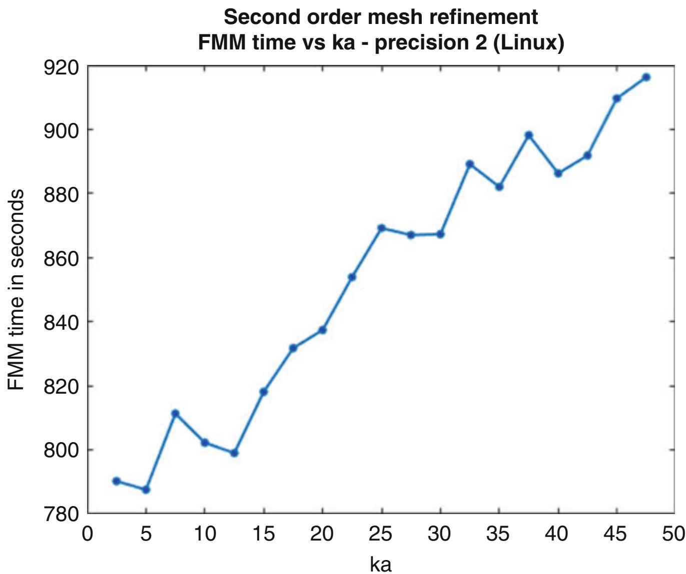


Fig. 36 Average FMM runtime for the doubly refined models in MATLAB on Linux is plotted with respect to values of ka. The demanded precision is six digits accuracy

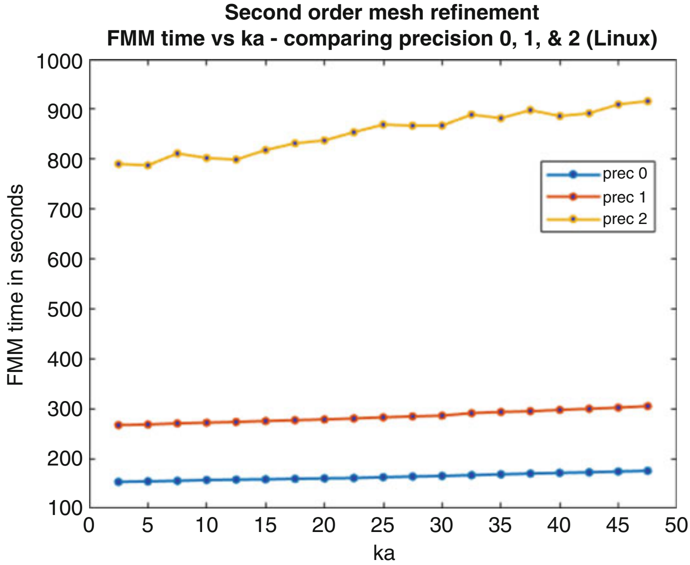


Fig. 37 A comparison among average FMM runtimes for the doubly refined models in MATLAB on Linux for when two, three, and six digits accuracy are demanded

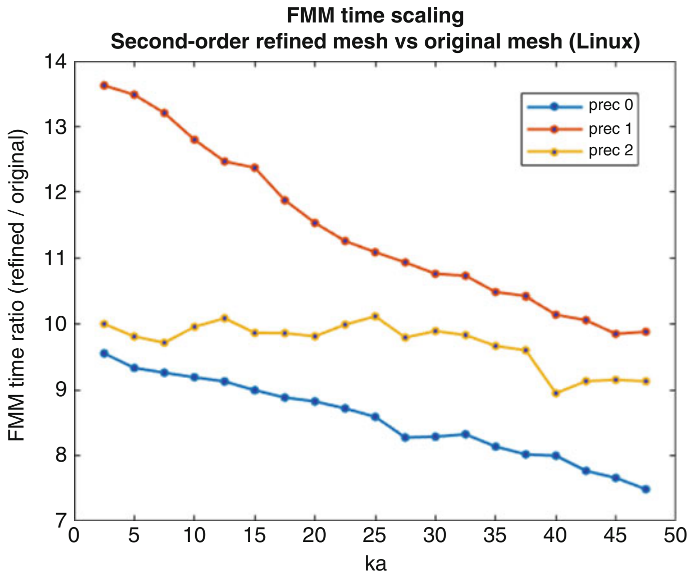


Fig. 38 Ratio between FMM time (Linux) for the doubly refined models and FMM time for the original CAD models is plotted with respect to ka

performance in Linux. *Note the legends in the plots*: "new2" = newly updated MEX file (June 20, 2019), "new1" = MEX file in the original new FMM library (downloaded June 11, 2019), "old" = MEX file in the old FMM library (2017), and zero time = program fails.

In Figs. 39, 40, and 41, FMM runtimes of the new and old libraries on Linux server with precisions 0, 1, and 2, respectively, averaged over all 16 refined meshes, are presented. A few comments are in order:

- The newest FMM library of the three tested crashed at high frequencies. The reason for these crashes is due to memory leaking. This issue has been fixed meanwhile.
- At low frequencies (ka $\leq$50), the newest library has the best runtime (June 20, 2019), followed by the second newest (June 11, 2019), while the old library is slowest. However, the old FMM library shows significant superiority in runtime over the new libraries as the frequency increases.
- Both new libraries show improvements over the old FMM codes when the Laplace solver is in used, with the second newest FMM library having the best runtime.

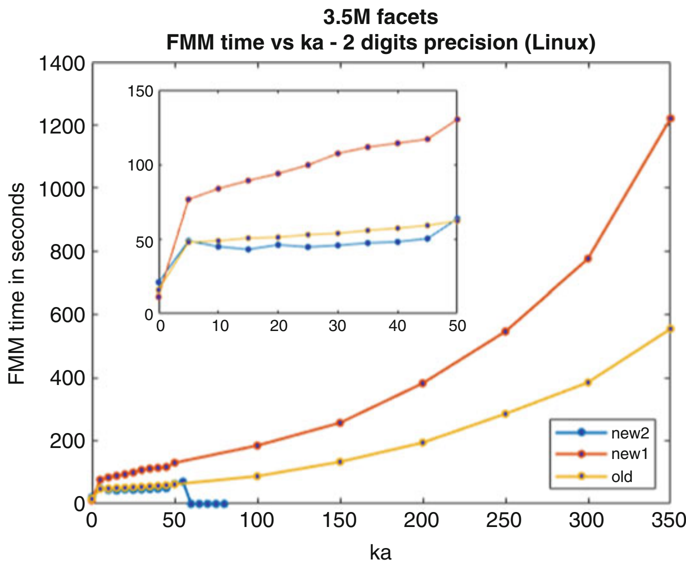


Fig. 39 FMM runtime of the new and old libraries for the original CAD models within MATLAB platform (averaged over all sixteen heads) on Linux vs ka. The demanded precision is two digits accuracy

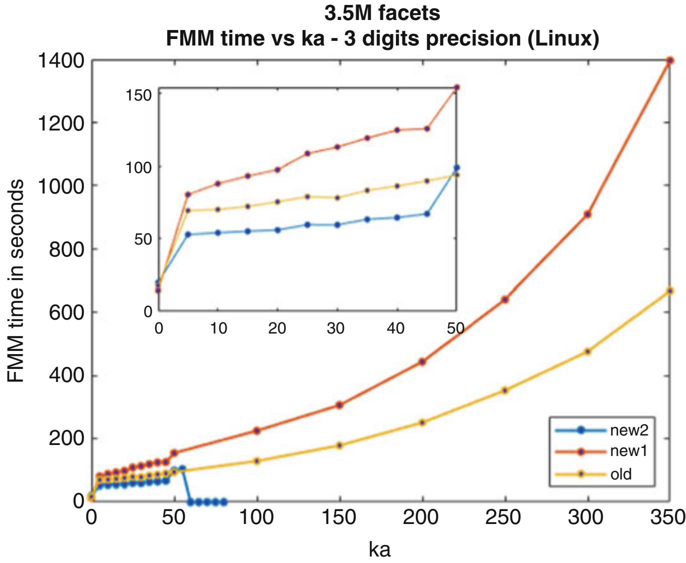


Fig. 40 FMM runtime of the new and old libraries for the original CAD models within MATLAB platform (averaged over all sixteen heads) on Linux vs ka. The demanded precision is three digits accuracy

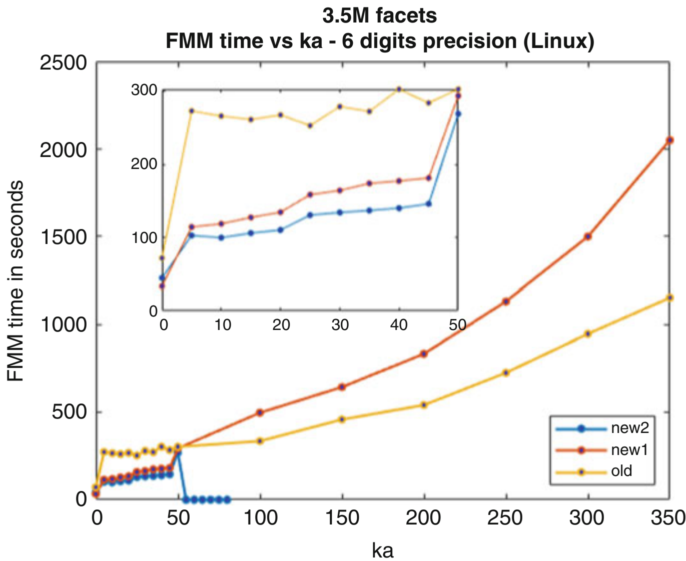


Fig. 41 FMM runtime of the new and old libraries for the original CAD models within MATLAB platform (averaged over all sixteen heads) on Linux vs ka. The demanded precision is six digits accuracy

Table 1 FMM runtimes in Linux and Windows for different number of sets of charge distributions are shown

Number of sets, N	Total runtime, T_N		
	Linux1	Linux2	Windows
1	5.37	5.66	2.99
2	6.18	5.64	3.87
4	9.10	7.69	6.43
8	12.82	10.75	11.90
16	19.72	17.94	22.89
32	35.46	35.43	43.70
64	66.76	75.58	90.07
128	X	X	X

3.6 Solving for Multiple Solutions in Parallel

The FMM software updated by the Flatiron Institute also allows one to solve multiple right-hand sides at a time. In this section, we report the scaling of the FMM code when it is used to solve the Laplace equation with multiple sets of charges simultaneously. The original CAD models are used. The performances of the MEX function compiled for Windows, Linux (June 11, 2019), and the updated version for Linux (June 20, 2019) are compared against each other. These three MEX options are, respectively, called Windows, Linux1, and Linux2 in the following table. Runtimes are in seconds, and "X" indicates program failures due to memory issues. In Table 1, we show the runtimes of FMM run on different platforms, when different number of sets are included. The scale curves are shown in Figs. 42 and 43. The most significant result is that FMM run on Windows has the shortest runtime when a small number of sets are computed. The Linux platform, on the other hand, has a much better scaling rate, and therefore, the runtimes on Linux (for both versions of the library) are progressively better than those on Windows as the number of sets increases.

4 Discussion and Conclusions

In this paper, we have studied the performance of the fast multipole method in computing the Laplace and Helmholtz source-to-source potentials within human head topology. The FMM software used for this study was developed by Gimbutas and Greengard [7], and we have profiled the method in a wide range of frequency values, mesh density, in both Linux and Windows frameworks, and with all choices of precision available. We showed that for problems that have reasonably "small" sizes (up to 3–4 million facets), the FMM runtime and memory consumption evolve in a predictable manner when run on both Linux and Windows. In particular, the runtime (and memory usage) varies exponentially as the frequency increases, with a small linear dependence at small values of frequency. We also observe universally a sharp, discrete increase in both runtime and memory from when the FMM software is using the Laplace solver to when the Helmholtz solver is used.

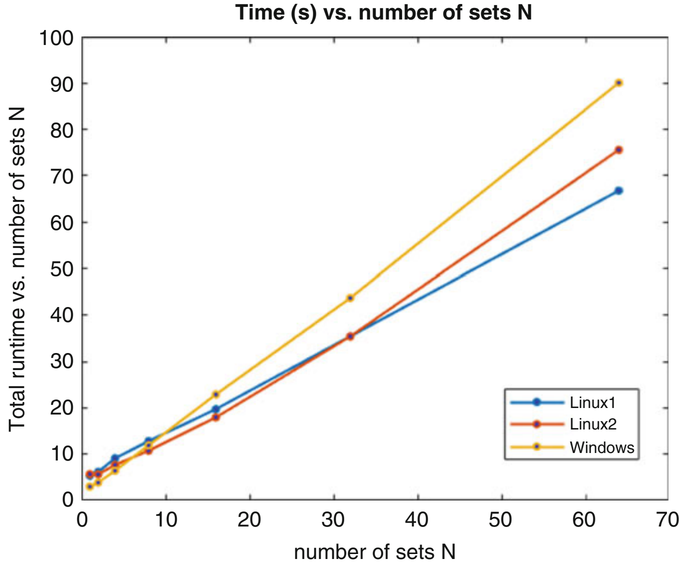


Fig. 42 The plots of runtime vs number of sets of charges are shown for Windows and Linux. Original CAD head models are used

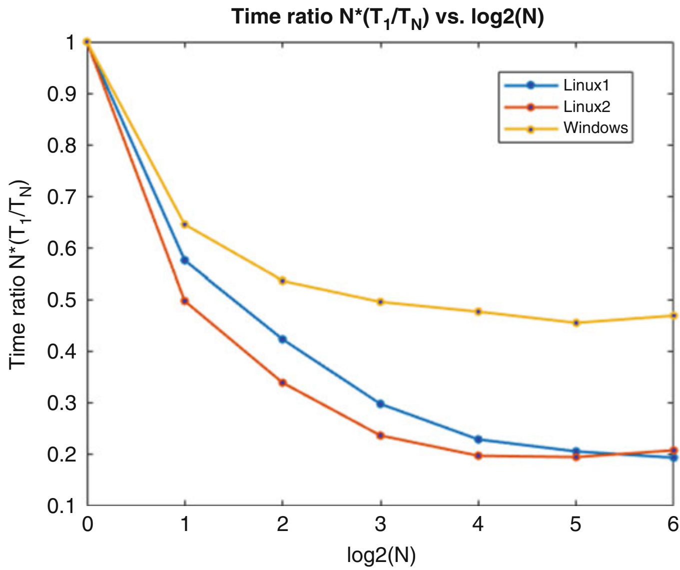


Fig. 43 The log plots of how runtimes on Windows and Linux scale with the number of sets of charges are shown. Original CAD head models are used

Upon studying the scaling efficiency of FMM, we showed that the algorithm deviates slightly from the theoretically expected scaling factors for runtime and memory, along with decreasing trends as the problem size increases. We also observed a number of interesting and unexpected results in terms of comparisons in runtimes required by different level of accuracy. In particular, we showed that for calculations run in Windows, the runtime needed for the three levels of accuracy tested did not follow any particular order at low frequency and only formed a pattern (low accuracy needed less time than high accuracy) when the frequency is sufficiently high. Resources needed for calculations performed in Linux were shown to have much more predictable patterns among different choices of accuracy and problem sizes, as well as smoother evolutions as the frequency changes.

We also compare the performance of this FMM library with a newer package (downloaded June 11, 2019) and its updated version (downloaded June 20, 2019). The results show that although the new library has better performance at low frequency, in Laplace calculations, it scales poorly compared to the old library and therefore is time-wise less efficient than the old FMM codes at high frequencies. Finally, we investigate the scaling rate of the new library with increasing number of right-hand sides (rhs) being solved simultaneously. The overall results show that while the Windows platform has the shorter runtime for small number of rhs, the FMM code compiled for Linux has better scaling rate and therefore has better runtime when the number of rhs increases.

With this study, we have benchmarked the performance of the general-purpose FMM, provided new insights to the behavior of the algorithms in various scenarios, and effectively offered a means to pre-estimate the efficiency of FMM-based or FMM-accelerated numerical methods.

References

1. Makarov, S. N., Noetscher, G. M., Raij, T., & Nummenmaa, A. (2018). A quasi-static boundary element approach with fast multipole acceleration for high-resolution bioelectromagnetic models. *IEEE Transactions on Biomedical Engineering, 65*(12), 2675–2683. https://doi.org/10.1109/TBME.2018.2813261.
2. Barnard, A. C. L., Duck, I. M., & Lynn, M. S. (1967). The application of electromagnetic theory to electrocardiology: I. derivation of the integral equations. *Biophysical Journal, 7*(5), 443–462. https://doi.org/10.1016/S0006-3495(67)86598-6.
3. Kybic, J., Clerc, M., Abboud, T., Faugeras, O., Keriven, R., & Papadopoulo, T. (2005). A common formalism for the integral formulations of the forward EEG problem. *IEEE Transactions on Medical Imaging, 24*(1), 12–28.
4. Makarov, S. N., Noetscher, G. M., & Nazarian, A. (2016). *Low-Frequency Electromagnetic Modeling of Electrical and Biological Systems Using MATLAB*. New York: Wiley. ISBN: 978-1-119-05256-2.
5. Rokhlin, V. (1985). Rapid solution of integral equations of classical potential theory. *Journal of Computational Physics, 60*(2), 187–207. https://doi.org/10.1016/0021-9991(85)90002-6.
6. Greengard, L., & Rokhlin, V. (1987). A fast algorithm for particle simulations. *Journal of Computational Physics, 73*(2), 325–348. https://doi.org/10.1016/0021-9991(87)90140-9.

7. Gimbutas, Z., & Greengard, L. (2015). Simple FMM Libraries for electrostatics, slow viscous flow, and frequency-domain wave propagation. *Communications in Computer Physics, 18*(2), 516–528. https://doi.org/10.4208/cicp.150215.260615sw.
8. Htet, A. T., Noetscher, G. M., Burnham, E. H., Pham, D. N., & Nummenmaa, A. (2019). Makarov SN. Collection of CAD human head models for electromagnetic simulations and their applications. *Biomedical Physics & Engineering Express, 5*(6). https://doi.org/10.1088/2057-1976/ab4c76.
9. Song, J., Lu, C. C., & Chew, W. C. (1997). Multilevel Fast Multipole Algorithm for Electromagnetic Scattering by Large Complex Objects. *IEEE Transactions on Antennas and Propagation, 45*(10), 1488–1493. doi: r S 0018-926X(97)07215–3.
10. Burgschweiger R, Ochmann M, Schäfer I, Nolte B. (2012). Performance-Optimierung und Grenzen eines Multi-Level Fast Multipole Algorithmus für akustische Berechnungen. *38. Jahrestagung für Akustik* (DAGA 2012), März 2012, Darmstadt, Germany.
11. Burgschweiger R, Schäfer I, Ochmann M, & Nolte, B. (2012). Optimization and Limitations of a Multi-Level Adaptive-Order Fast Multipole Algorithm for Acoustical Calculations. *Acoustics* 2012, Mai 2012, Hong Kong.
12. Burgschweiger R, Schäfer I, Ochmann M, & Nolte B (2013). The Combination of a Multi-Level Fast Multipole Algorithm with a Source-Clustering Method for higher expansion orders. *39. Jahrestagung für Akustik* (AIA/DAGA 2013), März 2013, Meran, Italy.
13. Gomez, L., Dannhauer, M., Koponen, L., & Peterchev, A. V. (2018). Conditions for numerically accurate TMS electric field simulation. *bioRxiv*, 505412. https://doi.org/10.1101/505412.
14. Htet, A. T., Saturnino, G. B., Burnham, E. H., Noetscher, G., Nummenmaa, A., & Makarov, S. N. (2019). Comparative performance of the finite element method and the boundary element fast multipole method for problems mimicking transcranial magnetic stimulation (TMS). *Journal of Neural Engineering, 16*, 1–13. https://doi.org/10.1088/1741-2552/aafbb9.
15. Documentation: https://fmm3d.readthedocs.io/en/latest/. Source code: https://github.com/flatironinstitute/FMM3D.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.