A study of the speed and the accuracy of the Boundary Element Method as applied to the computational simulation of biological organs

Kirana Kumara P
Centre for Product Design and Manufacturing,
Indian Institute of Science,
Bangalore, Karnataka 560012, India
Email: kiranakumarap@gmail.com

Abstract

In this work, first a Fortran code is developed for three dimensional linear elastostatics using constant boundary elements; the code is based on a MATLAB code developed by the author earlier. Next, the code is parallelized using BLACS, MPI, and ScaLAPACK. Later, the parallelized code is used to demonstrate the usefulness of the Boundary Element Method (BEM) as applied to the realtime computational simulation of biological organs, while focusing on the speed and accuracy offered by BEM. A computer cluster is used in this part of the work. The commercial software package ANSYS is used to obtain the 'exact' solution against which the solution from BEM is compared; analytical solutions, wherever available, are also used to establish the accuracy of BEM. A pig liver is the biological organ considered. Next, instead of the computer cluster, a Graphics Processing Unit (GPU) is used as the parallel hardware. Results indicate that BEM is an interesting choice for the simulation of biological organs. Although the use of BEM for the simulation of biological organs is not new, the results presented in the present study are not found elsewhere in the literature. Also, a serial MATLAB code, and both serial and parallel versions of a Fortran code, which can solve three dimensional (3D) linear elastostatic problems using constant boundary elements, are provided as supplementary files that can be freely downloaded.

Keywords: *Realtime, Soft biological organs, Computer cluster, Parallelize, Constant boundary elements, Code, MATLAB, Fortran, Download.*

1 Introduction

Realtime simulation of biological organs is a necessity while building realistic surgical simulators (realtime performance greatly enhances realism). Currently, realtime simulation in most cases is achieved using deformable models, spring-mass models, or models based on the Finite Element Method (FEM). There is lot of literature on the use of each of these approaches for the realtime simulation of biological tissues and organs including soft tissues (and organs that contain soft tissues, like liver). One can refer to any review paper or any of the many sources that contain lot of references to the relevant literature (e.g., [1-10]) to get an idea of the work that has already been done in this area.

Sources in literature widely agree on the fact that continuum mechanics based models (like models based on FEM) are desirable if one is interested to get accurate simulations but achieving simulations in realtime using the finite element models is difficult (if not impossible) because of the well known fact that realtime graphics needs about 30 computations per second whereas realtime haptics needs about 1000 computations per second.

One can see that the realtime simulation of three dimensional solids is a necessity for the continuum mechanics based realtime computational simulation of biological organs. One can also note that FEM is the most widely used numerical technique employed for this purpose. It is widely reported in literature that it is difficult to obtain realtime performance with FEM if nonlinear material behaviour (that describes the material behaviour of a biological organ, say liver) is to be incorporated. However it may be enough to describe biological organs with a linear elastostatic constitutive model sometimes, and in this case, it is reported in the literature that it is possible to perform simulations in realtime.

However there have been attempts to simulate realtime nonlinear behaviour with FEM (e.g., [11,12]). Information on the outcome of these attempts is not always readily available. Of course, FEM could be a great tool if one is not worried about the realtime performance; many commercial software packages (e.g., ANSYS) can easily perform a nonlinear simulation of biological organs if realtime performance is not a concern. Also, free and open source alternatives to the commercial finite element software packages are available, and one can use these packages to perform the simulations also; one can also use these packages as a background 'engine' to perform the simulations 'online' but it would be difficult to obtain the realtime performance still. Of course, there is no lack of references on the topic of simulation of biological organs where the realtime performance is not a concern; many a times, this is the case with surgical planning (e.g., [13]).

Apart from FEM, other numerical techniques like meshfree methods, and the Boundary Element Method (BEM) have been tried out in the literature for the realtime simulations. For example, author's work [14] uses the Finite Point Method (FPM) and precomputations to obtain realtime performance, whereas [15] uses the Method of Finite Spheres (MFS); both of these numerical techniques are meshfree methods. The review paper [1] identifies BEM as a candicate which could be better than FEM when it comes to the realistic and realtime simulation of biological organs for surgical simulations. References [16-18] use BEM for simulations. Reference [16] uses linear elastostatic boundary elements together with Woodbury formula; Woodbury formula is used to modify the characteristic matrix after the geometry is changed because of cutting. Reference [17] uses linear elastostatic boundary elements together with precomputations to achieve realtime simulations. Reference [17] also suggests parallel computing to obtain realtime performance but does not implement it. Reference [18] precomputes linear elastostatic boundary element solutions to obtain realtime performance. To simulate cutting, [18] uses interpolated pre-solutions which may not provide accurate solutions every time because cutting changes geometry and hence the solutions obtained from interpolated pre-solutions could be far from accurate at times. Reference [18] also suggests parallel computing to obtain realtime performance but does not implement it. A few more references that use BEM for simulations are mentioned in the next paragraph.

Reference [19] achieves realtime performance by using linear elastostatics and updating only part of the boundary element mesh where there is a change in the mesh, and updating the boundary element system of equations for this part of the geometry only. But this method may not be suitable every time especially when there is considerable difference between the original mesh and the modified mesh. Reference [20] deals with realtime boundary elements, but it is neither related to 3D linear elastostatics nor related to the simulation of biological organs. Reference [21] also uses linear elastostatics together with precomputations and interpolating precomputed solutions to achieve realtime performance. In this work also, upon cutting, boundary element mesh is updated only where there is a change in the mesh, and the boundary element system of equations is updated for this part of geometry only. A look up table is created in the precomputation step by applying unit displacement in three mutually perpendicular directions, on all the nodes in turn, and storing the solutions; since the principle of superposition holds good for liner elastostatic problems, the solutions stored in the look up table may be suitably superimposed to obtain the solutions during realtime simulations. In the concerned simulation, since the tumour is located completely within the soft tissue, [21] is an example where the boundary element has been used to solve a problem where a homogeneous and isotropic domain is enclosed within another homogeneous and isotropic region (i.e., a problem dealing with multiple regions or two regions); here, mesh that represents the tumour and the mesh that represents the soft tissue together form nested surfaces. Reference [21] also serves as an example where biological organs (soft tissue in particular) are modelled with linear elastostatic constitutive law. Reference [22] deals with a hybrid deformable model and uses BEM together with a mass-spring model to simulate biological organs.

By carefully looking at the literature that deals with the application of BEM to the realtime simulation, one can observe that all of the works achieve the realtime performance by following one of these approaches or by following a combination of the following approaches: (i) Calculate the characteristic matrix for the system offline, and also take the inverse of the characteristic matrix offline (useful only in case of 3D linear elastostatics and also where the characteristic matrix does not change) (ii) Using a fine mesh only near the points where there is an interaction between the biological organs and the surgical tools (and also using a fine mesh where there is a contact, e.g., contact with the surrounding organs), and using a coarse mesh for the rest of the regions (iii) Whenever there is a change in the geometry, boundary element mesh is updated only where there is a change in the mesh, and the boundary element system of equations are updated for this part of geometry only (iv) Compiling a look up table during the precomputation step by applying unit displacement in three mutually perpendicular directions, on all the nodes in turn, and storing the solutions; since the principle of superposition holds good for liner elastostatic problems, the solutions stored in the look

up table may be suitably superimposed to obtain the solutions during realtime simulations (v) Obtaining solutions by interpolating the solutions from the look up table (vi) Using hybrid models, e.g., using BEM together with a spring-mass model (vii) using Woodbury formula to modify the characteristic matrix after the geometry is changed (because of cutting, say)

One can see that directly and completely solving a BEM problem in realtime (without following any of the approaches (i) to (vii) above) definitely has advantages. For example, during surgical simulations, simulating cutting reduces to solving the BEM problem for the changed geometry. Also, simulating prodding, suturing, dealing with multiple regions, simulating the interaction between the surrounding tissues etc. do not require any special techniques or any special approaches; they always reduce to just solving a linear elastostatic problem with different loads and boundary conditions (and for a different mesh sometimes). And since the time required to solve a linear elastostatic problem can be estimated beforehand (for a given number of total degrees of freedom, and a given element type), one can always be sure that the computations are performed within the allowed time limit (i.e., in realtime). One can also note that many of the sources in literature (including many of the ones mentioned above) recommend parallelization, but they mention it as 'future work'. Author has not come across any source in the literature that reports the direct parallelization (i.e., without following any of the approaches (i) to (vii) above) of a BEM solution with the intention of obtaining realtime performance, although this approach has been recommended in many places.

Hence the present work aims to obtain the realtime performance using BEM. BEM code is parallelized and run on a computer cluster (up to 256 processors). This is a systematic study which demonstrates the applicability of BEM as applied to the realtime simulation of biological organs. Biological organs are assumed to be linear elastic, but in the end, the possibility of using BEM when nonlinear material behaviour is desirable (e.g., in the simulation of soft tissues) is given a look. One can note that nobody has tried to perform nonlinear simulations in realtime using boundary elements so far, may be because on the one hand codes for nonlinear boundary element analysis are not readily available (forcing one to write ones own codes which would take considerable amount of time, and further, developing these codes are nothing but a huge developmental work which may need to be taken up by a team (not individuals)) and on the other hand 'common sense' may be telling that it would not be possible to get realtime performance with nonlinear boundary elements because of inherent limitations and delays that may exist in computer systems.

However, one can note that BEM is an established numerical technique that has already been used to solve a wide variety of problems including nonlinear problems. In the literature, one can find the application of BEM to problems which are more or less similar to the simulation of biological organs (e.g., [23]). But one can also note that these applications do not bother about realtime performance, and some of these bother about two dimensional (2D) applications only.

At this point, it is also informative to note a few points mentioned in the present paragraph. Studies on the applicability of FEM (not BEM) for the realtime simulation of biological organs are already available [24]. But such studies are not available for BEM. Since BEM has been identified in the literature as a good candidate for the simulation of biological organs, further studies on the usefulness of BEM for the simulation of biological organs is a necessity. Present work is a small step towards fulfilling this necessity. In contrast to the 'theoretical' approach used in [24] to work out whether some simulation could be carried out in realtime, present work really carries out the simulations and then observes whether the simulations can be carried out in realtime. Although the approach followed in [24] has some advantages also, author can see that the approach followed in this work has the following advantages: (i) Even if it is found that some simulation cannot be carried out in realtime, the codes and the complete simulation could be useful if one is happy with near realtime performance although simulations are not strictly realtime (ii) Even if it is proved 'mathematically' that some simulations can be carried out in realtime, from the application point of view, one must really carry out the simulations and see that the simulations can be carried out in realtime (just a mathematical proof is not sufficient but a demonstration is required).

One can observe that Reference [25] deals with the parallel implementation of the boundary element method for linear elastic problems on a MIMD parallel computer, but it deals with two dimensional (2D) problems only. Also this work does not deal with realtime simulations, and it does not worry about the applicability of BEM to the simulation of biological organs (realtime or otherwise).

Present paper is organized as follows. Next section describes the BEM codes developed by the author. The codes are provided as supplementary files. The subsequent two sections deal with speed and accuracy of BEM, as applied to realtime simulations, respectively. Results are also presented in these two sections itself. The last section concludes the paper.

2 The code

The codes that can be used to solve any three dimensional linear elastostatic problem (without considering body forces) are provided with the present paper as supplementary files. The file 'bemconst.m' is a MATLAB program. The file 'bemconst.f90' is the same program but written in Fortran 90; it is a serial code. The file 'bemconstp.f90' is the parallelized version of the serial program 'bemconst.f90'. Input for the two Fortran programs is through a file named 'inbem96.txt' while the input data for the MATLAB program is contained in the MATLAB code itself.

The codes attached as supplementary files are based on the author's earlier work [26]. Since [26] provides some theory behind constant boundary elements as applied to linear elastostatics, and since it also explains a MATLAB code in detail, those details are not mentioned here. Apart from [26], one can refer to any standard text book (e.g., [27-30]) on boundary elements to know the theory behind boundary elements. Also, good tutorials like [31] may serve as quick but effective introduction.

There are mainly three differences between the code presented in [26] and the codes provided here as supplementary files. The first difference is that while [26] does not directly address any singularity (in [26], both strong and weak singularities have to be taken care of by accurate numerical integration only), the present code addresses strong singularity by using 'rigid body modes' and weak singularity is taken care of by utilizing higher number of integration points over each of the boundary elements. Next paragraph tells a few words about 'rigid body modes'.

When an integral as encountered in the 3D linear elastostatics without body forces (with constant elements) is strongly singular, the value of the definite integral exists only in the sense of Cauchy Principal Value (CPV). Cauchy Principal Values in this case may be found either by direct evaluation [32,33] or by using 'rigid body modes' explained in [28,29]. Present work uses 'rigid body modes' or 'rigid body considerations' as explained in [28,29] to evaluate strongly singular integrals.

The second difference between the present code and the code presented in [26] is that, [26] always uses 4 by 4 numerical integration (i.e., 16 integration points over each element) while the present code uses 16 by 16 numerical integration (i.e., 256 integration points over each element). Higher number of integration points ensure accurate evaluation of weakly singular integrals. Also, the present code has a provision to use 4 by 4, 8 by 8, or 32 by 32 integration points also; location of Gauss points and the corresponding weights for these cases are listed in the code itself; here the location of Gauss points and weights are obtained from [34]. One can note that even for 4 by 4 integration, present code uses different locations for integration points and also different weights when compared to [26]. Also, one can see that (unlike [26]) the Gauss points are located within the range -1 to 1 in the present code, and these points are then mapped to the interval from 0 to 1 in the code.

The third difference between the present code and the code presented in [26] is that the present code uses simpler expressions for mapping the integration points from local (or element) coordinates to the global coordinate. While [26] uses Equation 6.26 from [27] to achieve the mapping, present work makes use of the equation given in the errata [35] (under the section 'Chapter 6') for the book [27].

Author has made use of many resources while writing the present code. A few of the many noteworthy ones are [36-41]. Author has also made extensive use of the information contained in many of the online forums, especially while running/compiling/developing codes. However, all of the codes provided as supplementary files with this paper are developed by the author from scratch; they are not in whole or in part a copy or a translation of any of the codes developed by someone else.

Comment lines in the present code may provide useful information, and they may be useful to understand the code.

Coming to the GPU implementation, the supplementary file 'BEM_constant_element_GPU.m' is a GPU implementation of the supplementary file 'bemconst.m'. One has to note that this is only one of the many GPU implementations tried out by the author. The supplementary file 'matmul_cpugpu_timecompare.m' is a code that can compare the time taken by CPU and GPU to multiply a matrix by a vector; this code is of use while one wants to know whether the multiplications can be performed in realtime, for different matrix sizes. The file 'cpugpu_timecompare.m' is a code that can compare the time taken to solve a system of equations on CPU and GPU. The files 'GPU_time_matmul_trial1.txt' and 'GPU_time_matmul_trial2.txt' contain the results of the use of 'matmul_cpugpu_timecompare.m' while the files 'GPU_time_solve_trial1.txt' and 'GPU_time_solve_trial2.txt' contain the results of the use of 'cpugpu_timecompare.m'; they report the results for two trials, for each problem considered; 'i' in these files refer to the matrix sizes, e.g., i=16000 means that the matrix size is 16000 by 16000 and the size of the vector is 16000 by 1.

Coming to the hardware and software used, the MATLAB codes (with or without GPU) are run on a desktop computer (Intel(R) Xeon(R) CPU E5405 @ 2.00GHz (8 cores), 8GB RAM, Mainboard: Intel D5400XS, Chipset: Intel 5400B, Windows xp Professional x64 Edition, SSD: Corsair CSSD-F60GB2, MATLAB2011b (32 bit version), GPU: NVIDIA Quadro 4000 (Driver Version 6.14.12.9573)). Apart from the solid-state drive (SSD), the simulations were tried out using the ordinary hard disk also. Also, 64 bit version of MATLAB2011b was tried out. But since it was found from the simulation results that the SSD is about 1.5 times faster when compared to the conventional hard disk and the 32 bit MATLAB is about 10 times faster when compared to the 64 bit MATLAB, it is decided to use the SSD and the 32 bit MATLAB.

Coming to the hardware used to run the parallelized Fortran code, a computer cluster consisting of 17 nodes is used. The cluster consists of 9 nodes with 32 cores each (2.4GHz AMD Opteron 6136 processor, 64GB RAM) and 8 nodes with 64 cores each (2.2GHz AMD Opteron 6274 processor, 128GB RAM). A 500GB SATA HDD (3Gbps) is used for OS and system software. An Infiniband Card (MHQH19B-XTR) is used for MPI communication, and Dual-port Gigabit Ethernet Connectivity is used for enabling logins and NFS. And coming to software, Intel Composer XE (Version: 2011.5.220) which includes Intel Fortran compiler together with Intel Math Kernel Library (Intel MKL) is used with MVAPICH2 (Version: 1.8-r5423). CentOS 6.2 (Linux x86_64 Platform) is the operating system, and the batch scheduler software 'Torque' is used for job scheduling and load balancing. Although the cluster has 800 processors in total, only 256 cores are used in the present work. This is because, as a policy, no simulation can use more than 256 processors in this cluster. Also, from the results to be presented later in this paper, one can see that there is no need to go for more number of processors since it may not lead to any speed up. The supplementary file 'submit' is a sample job script; similar scripts may be used to submit jobs to the cluster whenever jobs are to be submitted through the job scheduler Torque. Jobs must be routed through Torque whenever the number of processors requested is more than or equal to 16; there is no need for the jobs to be routed through Torque whenever the number of processors requested is less than 16. Whenever Torque is not required, parallel jobs are run using a command that is similar to the command 'mpirun -np 4 ./a.out'; here '4' is the number of processors requested, and 'a.out' is the executable (which in this case is located in the same directory where the 'mpirun' command is issued). Whenever Torque is required, 'mpiexec' (Release 0.84) from Ohio Supercomputer Center (OSC) is used instead of the 'mpirun', in the job script. Sequential programs are run using a command similar to './a.out' where 'a.out' is the executable (which in this case is located in the same directory where the './a.out' command is issued). Sequential Fortran code is compiled using a command similar to 'ifort *.f90 -mkl=sequential'; parallel Fortran codes are compiled using a command similar to 'mpif90 *.f90 -mkl=cluster'; '*.f90' are the Fortran source codes here.

3 Speed

Many of the subsections in this section utilize a sample problem. The supplementary files 'bemconst.m', 'bemconst.f90', 'bemconstp.f90', 'BEM_constant_element_GPU.m' all solve the same sample problem by default. Also, the input file 'inbem96.txt' for the Fortran programs contains the geometry of the sample problem. The sample problem is about taking up a 4 mm by 4 mm by 4 mm cube, and completely fixing one face of the cube, and applying traction of 4 N/mm^2 in the y-direction over the whole of the opposite face. Each face of the cube is discretized into sixteen boundary elements. Young's modulus is assumed to be equal to 200000 N/mm^2, and Poisson's ratio is assumed to be equal to 0.33; aim of the simulation is to obtain the displacements for the elements that are subjected to known tractions and also to obtain the tractions for the elements that are subjected to known displacements, using constant boundary elements, and using linear elastostatic assumption and ignoring body forces. To demonstrate the speed that can be achieved using BEM, one needs to solve a problem using BEM, and the sample problem is of help here.

The following subsections give information about speeds that can be achieved using different hardware and software; results indicate whether or not a particular hardware-software combination can offer realtime performance for simulations considered in a particular subsection.

3.1 Running the MATLAB code on a desktop computer

In this subsection, no GPU is used. Also, no manual parallelization is attempted. But of course, the automatic parallelization available in MATLAB is utilized.

Time needed to run the code 'bemconst.m' is found using the MATLAB commands 'tic' and 'toc'. It was found that it takes 2.103116 s, 2.146334 s, and 2.091901 s respectively, when the same code was run

three times, using the default option of using all 8 cores (i.e., using the default fully automatic parallelization available in MATLAB) in the desktop. Next, the MATLAB code is made to run only on a single core in the desktop using 'maxNumCompThreads(1)', and the time needed to run the code in this case was found to be 2.100657 s, 2.089562 s, and 2.100865 s respectively, during three successive trials. Now, one can see that it takes more time to run the MATLAB code on 8 cores, when compared to the time needed to run the corresponding MATLAB code on a single core. Hence, the automatic parallelization offered by MATLAB is not of use for this problem (in fact, in this case, performance offered by the automatically parallelized code is worse when compared to the performance offered by the sequential code), and one can also observe that the code takes about the same time to run on the 8 cores as it takes to run on a single core. Also, one can see that the simulation is far from being realtime.

Now a small note on the speed of MATLAB in general. There is a general opinion that MATLAB is slower when compared to 'lower level' programming languages like Fortran, C, C++; in fact, there have been many discussions in online forums on topics like "Speed of MATLAB versus speed of Fortran" etc. Upon going through these forums, one comes across varying opinions like "Present day compilers for high level languages such as FORTRAN are so good that the codes written in high level languages are almost as fast as the same codes written in an assembly language, at least when the programmer is not extremely skilled", "MATLAB used to be about 100x slower when compared to languages like FORTRAN; but once MATLAB started using the modern JIT compiler, MATLAB is about 10x slower", "MATLAB has many built-in functions for scientific applications and the functions are so optimized for speed that it is difficult to write the same functions oneself, in languages like FORTRAN, to achieve the same speed", "If the programmer is not skilled, languages like FORTRAN could be slower when compared to MATLAB", or "Whether Fortran is faster when compared to MATLAB is highly dependent on the problem in hand as well as the skill of the programmer". In the present work, instead of assuming that the MATLAB is faster when compared to Fortran or otherwise, codes are written both in MATLAB and Fortran and whether Fortran is faster is decided based on the results from the actual runs; in fact, from the results presented (or to be presented) in this section, one can conclude that Fortran is significantly faster when compared to MATLAB; further, a Fortran code can be parallelized and run on a computer cluster whereas it is not easy to find a MATLAB version that can run on a cluster, and even if a MATLAB version that can run on a cluster is found, that version has its own limitations. Of course, people make their MATLAB codes faster by making use of 'System Objects', 'MATLAB Coder', 'MEX functions' etc. (e.g., [42]), but the present work aims to achieve the realtime performance without making use of these specialized approaches; also, all simulations cannot be translated to these approaches, and one can note that the GPU implementations of these specialized features are not available often, and further, it may not be possible to achieve the realtime performance with MATLAB even after employing these specialized techniques.

3.2 Manually parallelizing the MATLAB code to run on the multiple cores on the desktop

Since it is found from the previous subsection that the realtime performance cannot be obtained through either a sequential or an automatically parallelized MATLAB code, attempt is made in this subsection to manually parallelize the code.

Now, only a portion of the MATLAB code is parallelized; idea is that if realtime performance can be obtained for this portion of the code, manual parallelization can be attempted for a larger portion of the code; and of course, there is no need to attempt to parallelize the whole MATLAB code if one cannot obtain the realtime performance even for a portion of the whole code. Manual parallelization is attempted only for the second 'for' loop in the code 'bemconst.m' now; this is because the concerned 'for' loop is "embarrassingly parallel", and also because manual parallelization is as easy as just replacing 'for' with 'parfor'.

The time taken to execute the loop after just replacing 'for' with 'parfor' (without initializing 'matlabpool') is 0.101835 s, 0.101571 s, and 0.101935 s, for the three trials considered; here, the code runs on the 'client' only, not on the 'MATLAB workers'.

Next, 'matlabpool' is used to initialize 'matlabpool', and 'matlabpool close' is used to close 'matlabpool'. Again, 'for' is replaced with 'parfor'. The time taken to execute the concerned 'for' loop is 13.680187 s, 13.666539 s, and 11.532791 s, for the three trials considered; times include the time taken to initialize and close 'matlabpool'. If one ignores the time taken to initialize and close 'matlabpool', the time taken to execute the concerned 'for' loop is 0.323947 s, 0.322048 s, and 0.329585 s, for the three trials. Hence one can see that initializing and closing 'matlabpool' takes a lot of time. All the simulations mentioned in this paragraph use

the default 8 'workers' (since there are 8 cores in the desktop); 'workers' are also known as 'labs'.

Now, simulations exactly similar to the ones in the last paragraph but only with 1 'worker' are carried out. The time taken to execute the same 'for' loop, excluding the time taken for executing 'matlabpool' and 'matlabpool close', is 0.262614 s, 0.264331 s, and 0.265059 s, for three trials.

Now, the time taken for the same simulation as the one in the last paragraph but with 2 'workers' is found to be 0.276764 s, 0.273726 s, and 0.275193 s, for three trials. The time taken for the same simulation as the one in the last paragraph but with 4 'workers' is found to be 0.289515 s, 0.289589 s, and 0.289226 s, for three trials.

One can observe that as the number of 'labs' increase, simulation becomes slower in this case. Also, one can observe that none of the simulations mentioned in the present subsection could be completed in realtime. One can also observe that if one does not substitute 'parfor' for 'for', and uses the default automatic parallelization of MATLAB, the time taken to execute the 'for' loop is just 0.032205 s, 0.031991 s, and 0.032115 s, for the three trials, which makes the simulation a realtime one; the default option of using all the available 8 cores is used here. Instead of using the default option of using all the available cores, if 'maxNumCompThread' is used to restrict the number of cores to be used, following is the time needed to execute the portion of the MATLAB code within and including the concerned 'for' loop, without substituting 'parfor' for 'for': 0.031925 s, 0.031949 s, and 0.031941 s for the three trials if only one core is used; 0.032054 s, 0.032067 s, and 0.032008 s for the three trials if two cores are used; 0.033016 s, 0.032037 s, and 0.032364 s for the three trials if four cores are used; 0.031911 s, 0.032015 s, and 0.031862 s for the three trials if eight cores are used.

From the results presented in this subsection, at least for the problem considered in this subsection, there is no use in manually parallelizing the MATLAB code using 'parfor'. One should also note that not all statements that can be put inside a 'for' loop can be put inside a 'parfor' loop; for example, the fourth 'for' loop (considering only the outer 'for' loops) in 'bemconst.m' cannot be parallelized by replacing 'for' with 'parfor' (replacing 'for' with 'parfor' here generates error, and the code cannot run). Of course, one cannot rule out the possibility of a future version of MATLAB providing a better implementation of 'parfor'.

3.3 Parallelizing the MATLAB code to run on a GPU

Since the previous two subsections are not successful in obtaining realtime performance, there is a need to run the MATLAB code on the GPU to see whether one can achieve realtime performance.

GPU computing features available in MATLAB depend on the features available in CUDA and GPU drivers, which in turn depend on the features supported by the GPU hardware. But one has to note that only a small subset of MATLAB functions can be run on GPUs using the MATLAB Parallel Computing Toolbox. One can observe that newer versions of MATLAB (and the Parallel Computing Toolbox) have better support for GPUs, and one can definitely hope to see the future versions of MATLAB enabling more and more MATLAB functions to be run on GPUs using Parallel Computing Toolbox. But, as of now, a lot more needs to be done from the software developers to make many MATLAB functions to readily run on GPUs. Not only functions, but some MATLAB scripts cannot readily be ported to GPUs. Of course, programs written in lower level languages like Fortran and C may also be modified to run on GPUs; however, this task may not be easy always, and sometimes, porting a code to a GPU could itself be a research problem. Also, whenever a GPU is used, one needs to transfer the variables from CPU to GPU, and after performing computations on the GPU, results have to be transferred from GPU to CPU; and these data transfers are time consuming. Although author is aware of these limitations of GPU computing, present subsection makes an attempt to run the MATLAB code on a GPU with the intention of obtaining realtime performance, using the MATLAB Parallel Computing Toolbox.

Just like in the last subsection (which tried to parallelize the MATLAB code using 'parfor'), only a portion of the MATLAB code 'bemconst.m' is parallelized in this subsection first; idea is that if realtime performance cannot be obtained for even a portion (or a part) of the code, there is no need to attempt to parallelize the whole MATLAB code. Hence the parallelization is attempted only for the second 'for' loop in the code 'bemconst.m' now. Whenever the parallelized code is run on the GPU, there is a need to transfer the variables from CPU to GPU; the variables 'dispbcelements', 'fbcelements', and 'xyzofelements' are transferred from CPU to GPU in this case.

With prior initialization of the GPU arrays 'nx', 'ny', 'nz', 'xbar', 'ybar', 'zbar' and 'J' using 'parallel.gpu.GPUArray.zeros', time needed to execute the MATLAB program from the beginning of the program to the end of the concerned 'for' loop is found to be 1.249487 s, 1.247235 s, and 1.251234 s, for the three trials. But if GPU arrays are not initialized using 'parallel.gpu.GPUArray.zeros', time needed to execute the MATLAB program from the

beginning of the program to the end of the concerned 'for' loop is found to be 1.788709 s, 1.807157 s, and 1.804754 s for three trials.

The simulation that is exactly same as the one carried out in the last paragraph but carried out on the CPU alone (without using the GPU at all) takes 0.172460 s, 0.173135 s, and 0.172616 s (for three trials) to complete, if run on a single core of the desktop; but if all the 8 cores of the desktop are utilized, the same simulation takes 0.173307 s, 0.173084 s, and 0.173164 s, for three trials.

Now, with prior initialization of the GPU arrays using 'parallel.gpu.GPUArray.zeros', time needed to execute on the GPU the concerned 'for' loop alone is 1.251801 s, 1.251078 s, and 1.252707 s for three trials. Now, if the time needed to execute 'parallel.gpu.GPUArray.zeros' is also taken into account, time needed to execute on the GPU the concerned 'for' loop alone plus the time needed to execute 'parallel.gpu.GPUArray.zeros' is 1.270507 s, 1.257633 s, and 1.265864 s for three trials. But if GPU arrays are not initialized using 'parallel.gpu.GPUArray.zeros', time needed to execute on the GPU the concerned 'for' loop alone is 1.728226 s, 1.708492 s, and 1.729318 s for three trials.

One can see that none of the simulations that are tried out in the present subsection turned out to be realtime. Also, as far as the problems considered in this subsection are concerned, there is no advantage in using a GPU instead of a CPU. And since it is found that not even a portion of the MATLAB code 'bemconst.m' could be executed in realtime on a GPU, there is no point in trying to run the whole MATLAB code on a GPU in realtime.

3.4 Using the GPU to solve the system of equations only

Results presented in the previous subsections of the present section show that it is not possible to solve the sample problem in realtime by making use of a desktop computer loaded with MATLAB, even if a GPU together with the MATLAB Parallel Computing Toolbox is made use of. One can note that the sample problem is a small-sized problem and if it is not possible to solve this small-sized problem in realtime, it is not possible to solve a larger size problem in realtime. But it is of use sometimes, even if one manages to solve only a part of the sample problem in realtime. For example, if one is happy with linear elastostatics and if there is no change in the geometry during a simulation, the 'characteristic matrix' and its inverse can be precomputed, and the problem then reduces to just a matrix multiplication; and in this case, if one can manage to complete the matrix multiplication in realtime, it could be as good as solving the whole problem in realtime. Hence there is a need to see whether particular portions of the MATLAB code can be executed in realtime, either by making use of a GPU or not.

Hence, in this subsection, attempt is made to obtain the realtime performance while solving a system of linear simultaneous algebraic equations; also, in the next subsection, attempt is made to obtain the realtime performance while multiplying a matrix by a vector. One can note that solving a system of equations is a part of the MATLAB code 'bemconst.m'. The task that is carried out in the present subsection is taken up just out of curiosity (or academic interest) since, unlike the task that is concerned with just multiplying a matrix by a vector, present author cannot think of any use in achieving the realtime performance just for the portion of the MATLAB code that just solves the system of simultaneous equations.

One can see that the time needed to multiply a matrix by a vector depends on the size of the matrix, not on the actual values of the elements of the matrix. Similarly, at least when not using an iterative solver, the time needed to solve a system of equations is mainly dependent on the size of the system of equations only. Hence, in the present work, instead of solving the actual system of equations obtained through BEM, a dummy system of equations is generated and solved. Using dummy system of equations is useful here because, while the sample problem always generates a system of equations that has 288 simultaneous equations, different problem sizes can easily be tried out if dummy system of equations are utilized. In fact, the supplementary file 'GPU_time_solve_trial1.txt' gives the time needed to solve the system of equations when the size of the system varies from 500 to 9000, for every 500 increment in the size; similarly, the supplementary file 'GPU_time_solve_trial2.txt' lists the time for the same problem but for the second trial. In these files, times are listed for these cases: (i) when only the time needed to solve the system of equations on the GPU is taken into account (ii) time needed to transfer the right hand side from the CPU to the GPU plus the time needed to solve the system of equations on the GPU (iii) time needed to solve the system of equations on the GPU plus the time needed to transfer the result from the GPU to the CPU (iv) time needed to transfer the right hand side from the CPU to the GPU plus the time needed to solve the system of equations on the GPU plus the time needed to transfer the result from the GPU to the CPU (v) time needed to transfer the 'characteristic matrix' from the CPU to the GPU plus the time needed to transfer the right hand side

from the CPU to the GPU plus the time needed to solve the system of equations on the GPU plus the time needed to transfer the result from the GPU to the CPU (vi) time needed to solve the system of equations on the CPU.

From the results given in the supplementary files 'GPU_time_solve_trial1.txt' and 'GPU_time_solve_trial2.txt', there is not much deviation between the results from the two trials. Also, one can see that when the size of the system of equations is equal to 500, the system of equations can be solved in realtime, either by making use of the GPU or otherwise; here, solving the system of equations on the GPU, and solving the system of equations on the CPU, both take almost the same amount of time. But when the size of the system of equations is equal to around 1000, the system of equations can be solved in realtime only if the GPU is made use of. And when the size of the system of equations is equal to 1500, the system of equations cannot be solved in realtime whether a GPU is used or not.

3.5 Using the GPU to perform the matrix multiplication only

Motivation for the present attempt to obtain the realtime performance while multiplying a matrix by a vector is explained in the previous subsection above. The same arguments used in the last subsection to make use of dummy problems are applicable to the present subsection too. Here, the supplementary file 'GPU_time_matmul_trial1.txt' gives the time needed to multiply a matrix by a vector when the size of the vector varies from 100 to 16000, for every 100 increment in the size; similarly, the supplementary file 'GPU_time_matmul_trial2.txt' lists the time for the same problem but for the second trial. In these files, times are listed for these cases: (i) when only the time needed to multiply a matrix with a vector, on the GPU, is taken into account (ii) time needed to transfer the vector from the CPU to the GPU plus the time needed to multiply the matrix with the vector on the GPU (iii) time needed to multiply the matrix with the vector on the GPU plus the time needed to transfer the result from the GPU to the CPU (iv) time needed to transfer the vector from the CPU to the GPU plus the time needed to multiply the matrix with the vector on the GPU plus the time needed to transfer the result from the GPU to the CPU (v) time needed to transfer the vector from the CPU to the GPU plus the time needed to transfer the matrix from the CPU to the GPU plus the time needed to multiply the matrix with the vector on the GPU plus the time needed to transfer the result from the GPU to the CPU (vi) time needed to multiply the matrix with the vector, on the CPU.

Out of the (i) to (vi) cases mentioned in the last paragraph above, only the case (iv) has the practical significance. When the 'characteristic matrix' does not change over time, the precomputed inverse of the 'characteristic matrix' can be transferred to the GPU beforehand and the "realtime simulation" in this case reduces just to the case (iv) in the last paragraph. Hence only the result pertaining to the said case (i.e., case (iv) in the last paragraph) is given a close look in the remainder of this subsection.

A careful observation of 'GPU_time_matmul_trial1.txt' and 'GPU_time_matmul_trial2.txt' reveals that there is quite a bit of deviation between the results obtained through the two trials sometimes. There can be many reasons (e.g., CPU and/or GPU simultaneously carrying out some other task (e.g., rendering the display) while carrying out the assigned task) for this deviation, but one has to note that once it is observed that a certain simulation can be performed within certain time duration, it should be possible to perform the same simulation within the same time duration again. These points are applicable for the simulations to be mentioned in the next subsection (i.e., the simulations to be carried out on a computer cluster) also.

Now, upon studying the supplementary file 'GPU_time_matmul_trial2.txt', one can observe that when the size of the vector is 16000, the simulation on the CPU takes 0.418 s while the simulation that uses the GPU takes just 0.030 s (i.e., 14x speed up). The thirty computations per second desired by realtime graphics amounts to an allowable time of up to 0.033 s per computation, and one can note than this targeted speed for this simulation cannot be achieved if the GPU is not made use of here.

3.6 Running the Fortran code on a computer cluster

This subsection is a very important part of the present paper. Motivation for trying to obtain the realtime performance while running the Fortran translation of the entire MATLAB code has already been explained in the first section (i.e., 'Introduction') of the present paper; also, the last section (i.e., 'Concluding remarks') mentions some points related to the present subsection.

In this subsection, the Fortran translation of the whole of the MATLAB code 'bemconst.m' is run on a cluster. The Fortran code is run on 1, 4, 16, 64, and 256 processors, and the time needed to execute the Fortran code is noted down for each of the cases. One can note that the code first calculates the

'characteristic matrix' and the 'right hand side', and then solves the system of equations; the 'time' needed to execute the Fortran code (as noted down in the tables included in this subsection) always includes both the time needed to calculate the 'characteristic matrix' and the 'right hand side' and the time needed to solve the system of equations. Part of the code that calculates the 'characteristic matrix' and the 'right hand side' is separated from the part of the code that solves the system of equations by using the BLACS routine 'blacs_barrier'; hence, in the code, the task of solving the system of equations begins only just after the whole of both the 'characteristic matrix' and the 'right hand side' are assembled (i.e., solution of the system of equations can begin only after each of the processes in the process grid complete their part of the work in calculating the 'characteristic matrix' and the 'right hand side'). In the parallel version of the Fortran code, parallelization of the part of the code that calculates the 'characteristic matrix' and the 'right hand side' uses the 'Block Distribution' whereas the part of the code that solves the system of equations uses the 'Block-Cyclic Distribution' for distributing the data among processors. When 4 processors are used, a 2 by 2 process grid is used; when 16 processors are used, a 4 by 4 process grid is used; when 64 processors are used, a 8 by 8 process grid is used; and when 256 processors are used, a 16 by 16 process grid is used. Parallel versions of the Fortran code use ScaLAPACK to solve the system of equations while the sequential version of Fortran code uses LAPACK while solving the system of equations. All the parallel versions of Fortran code use BLAS and MPI.

One can see that the present simulation uses a 'characteristic matrix' of 288 by 288 size. One can note that when the parallelized Fortran code is run on a single processor, the 'characteristic matrix' is Block-Cyclically distributed by using a block size of 288 by 288. But here, when the parallelized Fortran code is run on 4 processors, block sizes of 144, 128, 64, 32, and 1 are tried out. Similarly, when the code is run on 16 processors, block sizes of 64, 32, and 1 are tried out; when the code is run on 64 processors, block sizes of 32 and 1 are tried out; and when the code is run on 256 processors, a block size of 16 is used. Idea behind choosing these block sizes is that, from literature, codes execute slower if the block sizes are too small (like 1), and again, if the block sizes are too large, some of the processors may not get any data to process and hence the very use of higher number of processors to achieve better parallelism may lose its purpose; also, some references recommend using a block size of 32, 64, or even 128, for good performance. Hence the block sizes for different cases that use different number of processors are chosen such that all the processors get some data to process, and different block sizes are tried out for the same cases to find out how block sizes affect the speed.

Now, the time taken for the code to execute on different number of processors, for different block sizes, is listed in Table 1; results are presented for four trials, and the averages of the four trials are also listed. Now, Table 2 gives the average time taken by the code to execute itself on different total number of processors, taking into account all the block sizes considered for the case, and taking into account all the trials of a particular simulation also.

Table 1: Solution time in seconds

	First run	Second run	Third run	Fourth run	Average
Serial (with '-mkl=sequential')	0.169622	0.245911	0.164885	0.133891	0.178577
Threaded (with '-mkl')	0.364344	0.335435	0.329739	0.246264	0.318946
Parallel (1 process, block size=288)	0.340316	0.230240	0.148864	0.181890	0.225328
Parallel (4 processes, block size=144)	0.225918	0.069416	0.135531	0.052501	0.120842
Parallel (4 processes, block size=128)	0.219353	0.116207	0.099092	0.106350	0.135251
Parallel (4 processes, block size=64)	0.405905	0.497169	0.212202	0.460067	0.393836
Parallel (4 processes, block size=32)	0.220805	0.186688	0.255070	0.238785	0.225337
Parallel (4 processes, block size=1)	0.531692	0.441317	0.344186	0.527060	0.461064
Parallel (16 processes, block size=64)	0.055054	0.066835	0.063014	0.069986	0.063722
Parallel (16 processes, block size=32)	0.065156	0.057134	0.050330	0.042243	0.053716
Parallel (16 processes, block size=1)	0.052755	0.049116	0.057509	0.064338	0.055930
Parallel (64 processes, block size=32)	1.007848	1.229159	0.750016	0.077925	0.766237
Parallel (64 processes, block size=1)	1.740406	2.041077	2.400125	1.170568	1.838044
Parallel (256 processes, block size=16)	2.067212	1.167335	1.320818	1.460196	1.503890

From the results presented in Table 1 and Table 2, for the sample problem (i.e., the problem considered here), block sizes do not play any significant and meaningful role in the overall sense. Also, on an average, the speediest performance is obtained when 16 processors together with a block size of 32 are used; for this

Table 2: Average solution time in seconds, considering all the runs and all the block sizes that are considered

Serial (with '-mkl=sequential')	0.178577
Threaded (with '-mkl')	0.318946
Parallel (1 process)	0.225328
Parallel (4 processes)	0.267266
Parallel (16 processes)	0.057789
Parallel (64 processes)	1.302141
Parallel (256 processes)	1.503890

case, the simulation took 0.053716 s to complete (this corresponds to about 19 computations per second). One can also observe that the fastest performance recorded in the tables corresponds to the 'Fourth run' corresponding to the case when 16 processors together with a block size of 32 are used; for this instance, the simulation took 0.042243 s to complete (and this corresponds to about 24 computations per second). And one can observe that if the average is taken for all the runs and all the block sizes together, on an average, speediest performance is obtained when 16 processors are used (solution time = 0.057789 s; this corresponds to about 17 computations per second).

One might need to keep a few points in mind while studying the results presented in this subsection. A cluster used here is made up of heterogeneous nodes. The nodes with 32 cores have 2.4 GHz processors whereas the nodes with 64 cores have 2.2 GHz processors. Also, internode communications could be slower when compared to intranode communications. In the present subsection, simulations using 1, 4, and 16 processors are usually run on a node having 32 cores, and the simulation that uses 64 processors is run on a node having 64 cores; the simulation requiring 256 processors is run using four nodes with 64 cores each. Of course, for each and every simulation mentioned in this subsection, each of the processors run one and only one process.

One can see that one is not likely to achieve faster performance by going for higher number of processors in this cluster.

One can note that because the whole of the MATLAB code 'bemconst.m' (not a part of 'bemconst.m') is translated and made to run on a cluster here, results presented here are useful when one wishes to learn about the performance of BEM when the 'characteristic matrix' changes during simulations (e.g., during the simulation of cutting, during the simulations that use nonlinear BEM). One can also note that the BEM used here is the 'standard' BEM, not any specialized version of BEM (e.g., the Fast Multipole Boundary Element Method which uses the fast multipole method to accelerate the solution of the system of equations).

3.7 Possibility of simulating nonlinear behaviour in realtime using BEM

Many a times, realistic description of nonlinear behaviour of biological organs (like liver) requires the use of hyperelastic material models (e.g., Mooney-Rivlin model, Neo-Hookean model). Solving one hyperelastic problem is equivalent to solving many linear problems. Although the total number of iterations (i.e., linear solutions or Newton iterations) required to solve a nonlinear problem within the specified tolerance (for the error) cannot be known beforehand, one can get an idea of the total number of iterations needed to solve a nonlinear problem by referring to the literature that deals with the solution of similar type of problems. For example, [43] includes the task of solving a hyperelastic problem, and by going through [43], the total number of Newton iterations needed to solve the problem is always between 62 to 70. Also, by making use of software like ANSYS which have in-built hyperelastic material models, one can solve similar hyperelastic problems to get an idea of the total number of iterations needed to solve such problems. Thus by solving dummy hyperelastic problems on ANSYS, and also by referring to the literature dealing with the solution of hyperelastic problems, one can see that solving a hyperelastic problem takes about 5, 10, 20, 50 or even 80 Newton iterations, while solving different nonlinear problems; it is also observed by the present author that more than 100 linear solutions are rarely needed to solve a hyperelastic problem, although this conclusion is reached just by observing a limited number of examples. Hence it is reasonable to assume that a hyperelastic problem can be solved in realtime if the corresponding linear problem can be solved in realtime 100 times.

But looking at the results presented in the previous subsections of this section, it may be difficult to obtain the realtime performance with a hyperelastic material model.

4 Accuracy

Constant boundary elements are around since a long time. Text books on BEM (e.g., [27]) inform that solutions obtained using constant boundary elements always converge as the total number of elements is increased. Hence there is no need to doubt the performance of constant boundary elements in fact. Still, there is a need to demonstrate that the author's code gives accurate results, especially when used for the simulation of biological organs.

First, author's code was used to solve the same problem as the one taken up in the last section (i.e., to demonstrate the speed). It was observed that the result obtained was in very good agreement with the result obtained using the formula from [44]. The same problem was also solved using the well known commercial software package ANSYS, and it was observed that the result from ANSYS was in very good agreement with the result obtained using the author's code as well as the result obtained using the formula from [44]. Similarly, author's code was used to solve many other problems where a bar was loaded with axial tensile or compressive loads, and where a beam was bent in different directions; the same problems were solved using different mesh resolutions also. Although results from these simulations are not presented here, it was found that the results from the code were always in good agreement with the results from ANSYS and the results obtained using the formulae from [44] both, thus validating the code.

Now, the code is used for the simulation of a biological organ. To perform any simulation on a biological organ, one needs the geometry of the organ considered. Here, author's code is used to do a simulation on a pig liver. The geometry of the pig liver is obtained from a stack of CT-scan images downloaded from [45]; the CT-scan images were downloaded a few years back when they were available for download from [45], but the images are not available for download now. The pig liver is reconstructed from the stack of CT-scan images using the procedure explained and illustrated in [46-49]; in fact, the geometry used here is the same as the pig liver which is described by 200 surface triangles (or 102 vertices) and shown in [46] and [49]. Also, as explained in [46-49], different filters are used to reduce the total number of surface triangles describing the geometry of the liver and also to improve the quality of the triangles describing the geometry. The final (processed using filters) geometry is described by about 200 surface triangles; here, all the triangles are of good quality. It is observed that the filters alter the geometry a bit, and it is also observed that the processed geometry is no longer to the correct scale (i.e., 1:1 scale). Hence, the processed geometry is scaled to the correct scale next. The final geometry is available from the supplementary file 'liver.db'.

Now, coming to material properties of the liver, Young's modulus is assumed to be equal to 150 N/mm^2 and Poisson's ratio is assumed to be equal to 0.4. One can note that values close to these values are widely used in the literature whenever a linear elastic material model is used for describing livers. One can also note that different sources in literature report different values for these material parameters, and sometimes the values of a particular material constant reported by different sources could be so different that the simulation results obtained by making use of these different values can deviate so much from one another that the results may lose their practical significance.

Now, coming to boundary conditions, some of the surface triangles that describe the liver geometry are fixed (in all x, y, and z direction) and some other surface triangles are prescribed a fixed value of displacement in a specified direction. To elaborate, referring to the supplementary file 'liver.db' that can be opened in the commercial (FEM based) software package ANSYS, the five surface triangles that share the 'keypoint' number 77 (xyz coordinates (rounded to one decimal place): -19.4, -1.59, 45.5) are fixed while the three surface triangles that share the 'keypoint' number 31 (xyz coordinates (rounded to one decimal place): -34.1, -48.1, 5.68) are prescribed a displacement value of 5 mm in the y direction. To make it clear, the location of keypoint 77 and keypoint 31 are shown in Figure 1 and Figure 2 respectively. Also, one can note that one needs to fix in all directions at least three boundary elements if constant boundary elements are used, and in the present case five elements (i.e., more than three elements) are fixed in all directions.

Now, coming to meshing in ANSYS, elements of tetrahedral shape are used. A fine mesh is created and the liver geometry is discretized into 9140 finite elements; the element used is 'Solid187' (also called 'Tet 10node 187').

Now the problem is solved using ANSYS. One can verify that there are 19 nodes in total, which are located on the three surface triangles that share the keypoint 31. The reaction forces at these 19 nodes in the y direction, after performing the 'solution' step in ANSYS, are found to be: for node 2723, 3416.8 N; for node 3943, 150.10 N; for node 3945, 111.98 N; for node 3947, 333.06 N; for node 3949, 511.75 N; for node 4004, -34.060 N; for node 4006, -7.6839 N; for node 4009, -72.606 N; for node 4011, -67.722 N; for node 4019, 820.62 N; for node 4021, 720.91 N; for node 4023, 854.16 N; for node 4025, 477.53 N; for node 4028, -40.859

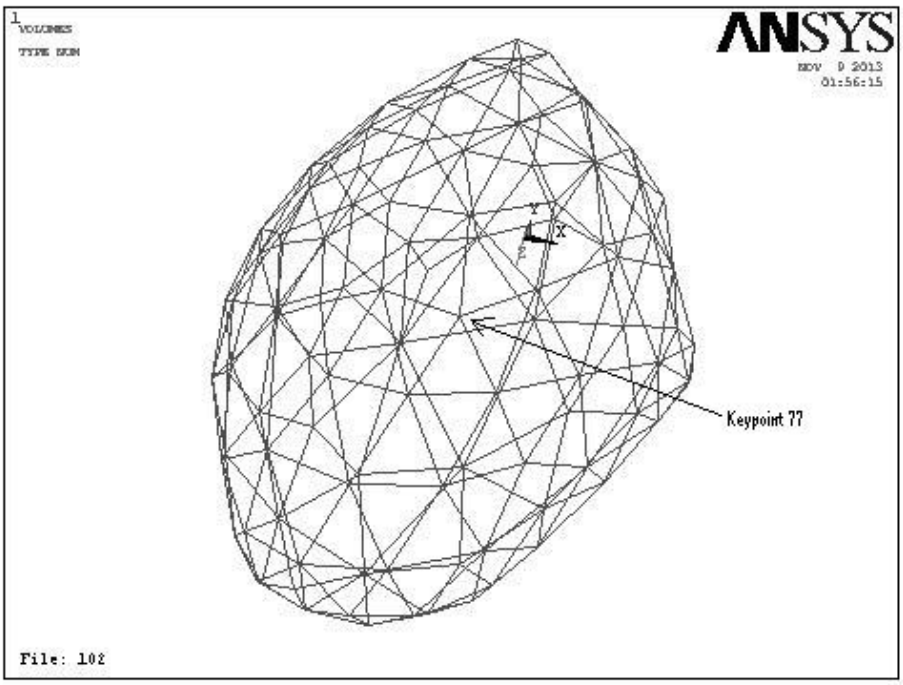

Figure 1: Location of keypoint 77

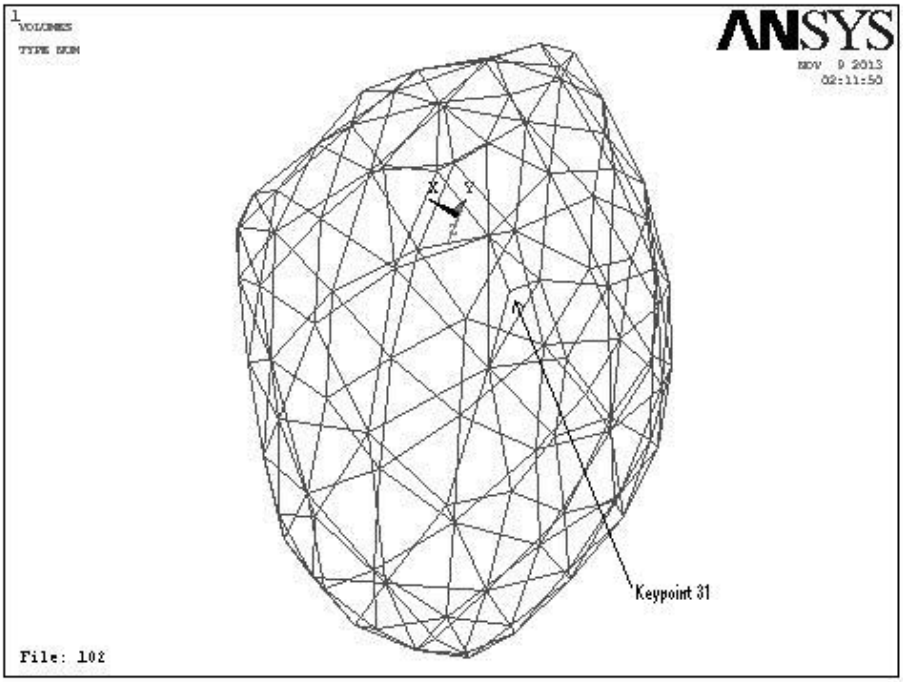

Figure 2: Location of keypoint 31

N; for node 4030, -34.094 N; for node 4316, 83.663 N; for node 4318, 124.30 N; for node 4331, -1.4593 N; and for node 4332, 9.3620 N. One can see that the sum of all these reaction forces turns out to be 7355.8 N; one can consider this value to be the solution given by ANSYS (i.e., the reaction force corresponding to the imposed displacement of 5 mm).

Now the same problem is solved using BEM. The MATLAB code 'bemconst.m' is modified to take into account the liver geometry and the associated boundary conditions as well as to take into account the proper values of material constants; the modified MATLAB code is available from the supplementary file 'liver.m'. Here, each of the 200 surface triangles that describe the liver geometry act as boundary elements also; hence no separate step of discretization is required, and the 200 surface triangles that describe the liver geometry act as the 200 boundary elements that together represent the discretization needed to solve the BEM problem. One can note that element numbers 103, 113, 126, 149, 153 ('keypoint' number 77 is the common 'keypoint' for these elements) are fixed while the element numbers 23, 26, 36 ('keypoint' number 31 is the common 'keypoint' for these elements) are subjected to a displacement of 5 mm in the y direction. Upon solving the BEM problem, one obtains the solutions (traction in y direction) at the elements 23, 26, 36. The traction in the y direction is found to be 55.396109 N/mm^2 for the boundary element number 23. Similarly, traction in y direction is found to be 19.902604 N/mm^2 for the element number 26, and the traction in y direction is found to be 23.103882 N/mm^2 for element number 36. To calculate the reaction force corresponding to the imposed displacement of 5 mm, one needs to calculate the surface area of the elements that are subjected to an imposed displacement of 5 mm. Using the well known Heron's formula to calculate the area of triangles, area of the element 23 is found to be equal to 238.2615 mm^2 whereas the area of the element 26 is found to be 198.5379 mm^2 and the area of the element 36 is found to be 297.4421 mm^2. The total reaction force corresponding to the imposed displacement of 5 mm is obtained by multiplying the traction with the area of the corresponding element, for each of the three elements, and adding the three products. The total reaction force obtained by following this procedure is found to be equal to 24022 N; one can consider this value to be the solution given by BEM (i.e., the reaction force corresponding to the imposed displacement of 5 mm).

By comparing the solution obtained from ANSYS with the solution obtained from BEM, solutions may seem to be not in good agreement. But this this is not true in fact. One can see that if the same problem is solved using ANSYS but with a different mesh resolution, the result is observed to be considerably different from the one obtained for the mesh used earlier in this section. Hence, as far as the reaction forces of the type that arise out of the problems similar to the one taken up in this section are concerned (i.e., when reaction forces that are spread over the surface of a few elements need to be calculated), from the results obtained in this section, the solution given by BEM can be thought to be in good agreement with the solution given by ANSYS. Further, one has to note that the BEM uses 200 surface elements here whereas ANSYS uses 9140 volume elements, and hence a direct comparison of the result obtained from BEM with the result obtained from ANSYS is not of much significance. Of course, it would have been a cause for concern if the results obtained from BEM and ANSYS differed from one another to such an extent that one is an order (or several orders) of magnitude different from the other, or if one result happened to be positive while the other result happened to be negative. Of course, if a very fine volume meshing is used while solving the problem using ANSYS, and if a very fine surface meshing is used while solving the problem using BEM, the solutions obtained are bound to be in very good agreement with one another.

Now a few observations. One can see that unknowns can either be displacements or tractions in BEM while the unknowns in FEM typically are displacements only; hence while solving the present problem using BEM, reaction forces at the keypoint 31 are obtained directly by solving the system of equations, without the need to calculate the forces from displacements (as carried out by ANSYS during postprocessing). During surgical simulations, often the goal is to obtain reaction forces corresponding to prescribed values of displacements (i.e., goal is not to obtain displacements that correspond to known values of forces), and one can observe that BEM could be better than FEM for these cases.

Results from this section indicate that BEM can be used for simulations involving complex geometries (like a liver) also. Although only one complicated geometry (i.e., a liver which is a biological organ) is considered here, there is no need to try out simulations on different biological organs since there is no reason a simulation that can be carried out on a particular complicated geometry cannot be carried out on a different complicated geometry (proper meshing may be necessary of course).

It may be possible to obtain better convergence rates if other types of elements (e.g., linear elements, quadratic elements) are used instead of constant elements. But use of constant elements has certain advantages. Although convergence is slow, constant elements are fast in the sense that they do not need complicated shape functions that are computationally expensive. Constant elements are easy to use and program. No

connectivity information is needed if constant elements are used. This makes it easy to parallelize the code, and the resulting code is highly scalable. Constant elements are suitable for handling multiple regions since nodes are located completely within the elements (not on the edges or corners). Of course, accuracy may be poor for a given number of elements, when compared to linear and quadratic elements. But there are applications (like the realtime simulation of biological organs) where accuracy is not too important when compared to speed.

There are advantages in using boundary elements of triangular shape. Any complicated geometry can easily be represented by surface triangles. Also, one can construct a geometry using any standard CAD software package and then use the 'stl export' option to obtain the surface mesh (exporting a mesh to a VRML file may also serve the purpose). Mesh processing tools and remeshing tools (e.g., MeshLab, ReMESH) are widely available for surface meshes made up of triangles. Tools that can modify, edit, heal, and improve the quality of surface meshes that are made up of triangles (i.e., tools that can convert a mesh made up of ill shaped triangles to a mesh made up of well shaped triangles) are also widely available. One can note that it is easy to obtain 3D models described by surface triangles from 3D scan. One can also note that many of the software packages that can do 3D reconstruction of biological organs from 2D image sequences have the 'stl export' option. Literature tells that it is easier to perform rendering and collision detection with meshes made up of surface triangles.

Now, a few of the advantages of using BEM for simulations is mentioned here. The BEM needs a meshing of only the boundary of any geometry (at least for linear problems). Hence lesser number of elements can describe geometry. Also, literature mentions that the BEM is "embarrassingly parallel"; hence parallelizing BEM would help in reducing solution times. BEM is generally thought to be an efficient numerical technique. In BEM, some of the unknowns can be displacements while at the same time the remaining unknowns can be tractions; hence computation of tractions from displacements is not needed. Often, one needs to know the solutions only on the boundary of an organ (or only at the nodes of boundary elements), and with BEM, there is no need to calculate solutions at internal points to get the solutions at the boundary, while FEM unnecessarily calculates solutions at internal nodes also. Also, it is widely mentioned in the literature that collision detection and rendering are easy with the type of object representation used in BEM. The system of equations resulting from BEM is dense while FEM produces sparse 'characteristic' matrices. Since the algorithms and routines that take advantage of the sparseness of characteristic matrices are rare, BEM has an upper hand in this case (also since BEM needs meshing of only the boundary of the geometry and hence lesser number of elements can describe a geometry).

Coming to the question of the validity of the linear elastic material models for large deformations, there is no clear line that separates linear analysis from nonlinear analysis. Some sources in literature mention that linear elastic analysis is valid up to an 'elongation' of 0.2%, while some other sources utilize linear elastic formulations when the 'elongations' are 1%, 2%, 5%, or even 10%. Clearly, whether a particular simulation must make use of nonlinear formulations depends on the application, and linear elastic formulations are all right as long as the results obtained are useful and satisfactory. Again, same type of argument is applicable when it comes to the allowable error in a particular simulation, even when the simulation employs a linear analysis only. Although nonlinear formulations can give more realistic results when it comes to the simulation of biological organs, as already mentioned in the first section (i.e., 'Introduction') of this paper, present work concerns itself mostly about linear elastic analysis only.

5 Concluding remarks

In this work, boundary element codes that have been developed by the author are presented first. Codes are supplied as supplementary files. The codes may be used to solve any linear elastostatic problem without accounting for body forces (not just the simulation of biological organs) either in realtime or otherwise.

Next, a try is given to parallelize the entire MATLAB code with the intention of running it on a desktop computer equipped with a GPU. Here the goal is to compute the 'characteristic matrix' and the 'right hand side', and also to solve the system of equations, all in realtime. Author has not been successful in achieving this goal using MATLAB and the Parallel Computing Toolbox.

Next, thought was given to obtain the realtime performance, again by utilizing just an ordinary desktop computer and a GPU, by resorting to precomputations. In this case, one has to precompute the 'characteristic matrix' and its inverse offline. Again, MATLAB and Parallel Computing Toolbox are used for this purpose. Results show that it is possible to get realtime performance (realtime graphics, not realtime haptics) when the size of the 'characteristic matrix' is up to about 16000 by 16000; one can also see that one can get a speed

up of about 14x when the 'characteristic matrix' is of about 16000 by 16000 size. This means that one can perform realtime simulations using boundary elements if the geometry is described by about 5300 constant boundary elements. Of course, the usual limitations with approaches that use precomputations apply in this case too (like one may not be able to compute the 'characteristic matrix' beforehand if there is change in the geometry during simulations, e.g., during simulation of cutting). But one can note that it is not possible to perform the same simulation (with the same precomputions of course) in realtime using just an ordinary desktop if a GPU is also not made use of.

Next, a computer cluster is used to carry out BEM simulations. The fully parallelized version of the code (i.e., the parallel Fortran code) is used for this purpose. In this case, the goal is to compute the 'characteristic matrix' and the 'right hand side', and also to solve the system of equations, all in realtime. Simulations are carried out on 1, 4, 16, 64, and 256 processors. Also, simulations are carried out for different block sizes while Block-Cyclically distributing matrices among processors. Results indicate that one can barely obtain a good realtime performance as far as realtime graphics is concerned; the fastest simulation could complete 24 computations per second (as against the required 30 computations per second for high quality realtime graphics).

Results from the simulations carried out on the computer cluster demonstrate that it is possible to perform realtime graphical simulations using BEM on a computer cluster. Here, carrying out a simulation means not just performing a matrix multiplication, or not just solving a system of equations, but it includes computing the 'characteristic matrix' and the 'right hand side' from scratch also. One has to note that the present study has used only standard methods of parallelizing a program; no custom methods and communication protocols have been used.

Results also indicate that BEM is useful for the realtime linear elastostatic simulation of not only biological organs but anything that attempts to simulate just the linear elastostatic response in realtime.

One can also see that it may be difficult to obtain realtime performance (even realtime graphics) if nonlinear material behaviour is to be incorporated into simulations, if custom methods and communication protocols are not used while parallelizing simulations. Results also imply that it may be difficult to obtain realtime haptic feedback with nonlinear material models, even if custom methods and communication protocols are used while parallelizing simulations. However, results also indicate that BEM could be a very useful numerical technique for the realistic and realtime simulation of biological organs (including the highly nonlinear soft biological organs) if one is happy with nearly realtime performance, i.e., one computation taking just a few seconds.

Of course, simulations may not need to be strictly realtime sometimes (i.e., a 'hard' realtime system may not be needed, but 'firm' or 'soft' realtime systems may also be suitable). For example, [18] mentions that a neurosurgeon takes about 0.25 seconds to perform each incremental stage of surgical cutting in practice; hence it may be all right for the simulations to take 0.25 seconds to complete in this case. Also, [50] mentions that the allowable lag time is 0.1 seconds for inexperienced users and it is 1 second for experienced users; hence the simulations can take 0.1 or 1 second in this case. References [51-53] mention that the just noticeable difference of the human sensory system for force perception at human fingertips is about 10%. But these references cannot serve as excuses for not obtaining realtime performance, since there would not have been a need for the various attempts in the literature to obtain a strictly realtime performance if one could always justify that a realtime performance is not a necessity when it comes to the realistic and realtime simulation of biological organs.

Present work also serves to make some record of the performance (like speed) that can be obtained by present day typical computing hardware (like a desktop computer, a graphics processing unit (GPU), a computer cluster) together with contemporary software (like MATLAB, MATLAB Parallel Computing Toolbox, Fortran, MPI, BLACS, ScaLAPACK).

As to the limitations of the present work, only 'standard' methods of parallelization are tried out; only standard and readily available tools like BLACS, ScaLAPACK, MPI are used. But this in fact is done purposefully, with the intention of knowing whether one can achieve the targeted realtime performance without using custom methods and communication protocols; the task of parallelizing should not itself be a research problem. The second limitation could be that the third section of this paper (i.e., 'Speed') uses a simple geometry (i.e., a cantilever beam) to demonstrate the speed of BEM, and does not mention anything about the speed of BEM when simulations are carried out on complicated geometries like biological organs. But one can note that the speed of a simulation here depends solely on the total number of elements and boundary conditions, and the speed does not have any relevance to the actual geometry; hence there is no need to carry out simulations on different geometries just to measure how fast (or slow) the simulations are; in

fact, trying out simulations on different geometries could be of help when one is concerned about accuracy but one can also note that the fourth section of the present paper named 'Accuracy' uses a complicated geometry (i.e., a liver), not just a cantilever beam, to demonstrate the accuracy of BEM. The third limitation of the present study could be that only one problem is taken up here for studying the performance of the parallel Fortran code on the computer cluster, and different problem sizes have not been tried out on the cluster to understand how the speed varies with problem size. But since there is no reason why a larger size problem can be solved faster when compared to a smaller size problem, and since the problem that has been taken up and solved can be solved only for a maximum of 24 times a second, there really is not a need to go for a higher problem size as far as the realtime performance is concerned although going for a higher problem size may offer better performance in the sense that there may be only a small increase in the solution time when the problem size increases by a significant amount.

Future work could be to develop an exhaustive BEM library that could be used to solve highly nonlinear problems on parallel computers. The library could then be utilized to perform either realtime or nearly realtime simulation of biological organs. Finally the simulations could be integrated into a surgical simulator. However, these tasks would require significant amount of time, expertise and resources.

Future work could be in a different direction also, i.e., using 'soft computing' techniques (e.g., using artificial neural networks (ANN), using genetic algorithms and genetic programming, using fuzzy logic systems etc.) to achieve both realistic and realtime simulation of biological organs simultaneously. But it is not clear how one can use fuzzy logic systems to simulate biological organs. And one can note that genetic algorithms are inherently sequential and hence difficult to parallelize; also in this case, one cannot know beforehand the total number of iterations needed (and hence the total time) to solve a particular problem, i.e., one cannot know beforehand the total 'generations' needed to obtain an accurate solution; also, the methods need to use a few parameters the values of which may not be known beforehand. But from literature, one can see that ANN has already bean used to achieve realistic simulations of biological organs in realtime ([54]) by making use of just an ordinary desktop computer; use of hardware like neurocomputers (an overview of neurocomputers may be found in [55]) may speed up the ANN implementations further. Also, present author has done some preliminary work on using Support Vector Machines (SVM) to achieve realistic simulations in realtime [56], and the future work could be to continue the work.

Although it is possible to simulate nonlinear material behaviour in realtime (including realtime haptics) by utilizing just an ordinary desktop computer if one makes use of ANN or SVM, there are serious limitations for this approach. The Support Vector Machines or artificial neural networks need to be trained before using them for the simulations, and the training requires a huge amount of training data. And whenever there is a change in the geometry (e.g., because of cutting), the SVM or ANN has to be trained again, using the training data generated for the changed geometry. Hence it is not possible to use SVM or ANN in these cases to perform realtime simulations. One can note that continuum mechanics based numerical techniques (like BEM or FEM), although computationally intensive and may require high end computer clusters, can handle this type of problems and hence can be used to obtain realistic simulations for this type of cases also, at least with nearly realtime performance if not with realtime performance.

Disclaimer

Codes are provided without any guarantee or warranty. Also, author is not responsible for any loss or damage that may arise out of any type of use of any of the supplementary files that are provided with this paper.

Acknowledgments

Author is grateful to the Robotics Lab, Department of Mechanical Engineering & Centre for Product Design and Manufacturing, Indian Institute of Science, Bangalore, INDIA, for providing access to a Graphics Processing Unit (GPU).

Author acknowledges Supercomputer Education and Research Centre (SERC), Indian Institute of Science, Bangalore, INDIA, for providing access to computer clusters.

References

[1] U. Meier, O. Lopez, C. Monserrat, M. C. Juan, M. Alcaniz, 2005, Real-time deformable models for surgery simulation: a survey, Computer Methods and Programs in Biomedicine, 77(2005), pp. 183-197, doi:10.1016/j.cmpb.2004.11.002

[2] Herve Delingette, 1998, Toward Realistic Soft-Tissue Modeling in Medical Simulation, Proceedings of the IEEE, Vol. 86, No. 3, March 1998, pp. 512-523.

[3] Herve Delingette, Nicholas Ayache, Soft Tissue Modeling for Surgery Simulation, Online resource from http://www.inria.fr/centre/sophia/ (accessed October 25, 2013)

[4] Yongmin Zhong, Bijan Shirinzadeh, Julian Smith, Chengfan Gu, 2009, An electromechanical based deformable model for soft tissue simulation, Artificial Intelligence in Medicine, 47(2009), pp. 275-288, doi:10.1016/j.artmed.2009.08.003

[5] Sarthak Misra, K. T. Ramesh, Allison M. Okamura, 2008, Modeling of Tool-Tissue Interactions for Computer-Based Surgical Simulation: A Literature Review, Presence, October 2008, Vol. 17, No. 5, pp. 463-491.

[6] Jung Kim, Mandayam A. Srinivasan, 2005, Characterization of viscoelastic soft tissue properties from in vivo animal experiments and inverse FE parameter estimation, Medical Image Computing and Computer-Assisted Intervention - MICCAI 2005, Lecture Notes in Computer Science, Volume 3750, pp. 599-606.

[7] Jung Kim, Bummo Ahn, Suvranu De, Mandayam A. Srinivasan, 2008, An efficient soft tissue characterization algorithm from in vivo indentation experiments for medical simulation, The International Journal of Medical Robotics and Computer Assisted Surgery, Volume 4, Issue 3, pp. 277-285, September 2008, DOI: 10.1002/rcs.209

[8] Basdogan C., 2001, Real-time simulation of dynamically deformable finite element models using modal analysis and spectral Lanczos decomposition methods, Stud Health Technol Inform. 2001;81:46-52.

[9] Ahn B., Kim J., 2010, Measurement and characterization of soft tissue behavior with surface deformation and force response under large deformations, Medical Image Analysis, 14, pp. 138-148.

[10] Firat Dogan, M. Serdar Celebi, 2011, Real-time deformation simulation of non-linear viscoelastic soft tissues, Simulation, Volume 87, Issue 3, March 2011, pp. 179-187, doi:10.1177/0037549710364532

[11] http://www.scholarshipnet.info/postgraduate/uk-msc-and-phd-studentships-in-brain-surgery-simulatio (accessed October 24, 2013)

[12] http://www.stanford.edu/group/sailsbury_robotx/cgi-bin/salisbury_lab/?page_id=343 (accessed October 24, 2013)

[13] http://www.ansys.com/staticassets/ANSYS/staticassets/resourcelibrary/article/AA-V5-I2-Cut-to-the-Bo pdf (accessed October 24, 2013)

[14] Kirana Kumara P, Ashitava Ghosal, 2012, Real-time Computer Simulation of Three Dimensional Elastostatics using the Finite Point Method, Applied Mechanics and Materials, 110-16, pp. 2740-2745.

[15] Suvranu De, Jung Kim, Yi-Je Lim, Mandayam A. Srinivasan, The point collocation-based method of nite spheres (PCMFS) for real time surgery simulation, Computers and Structures, 83(2005), pp. 1515-1525.

[16] Ullrich Meier, Carlos Monserrat, Nils-Christian Parr, Francisco Javier Garcia, Jose Antonio Gil, 2001, Real-Time Simulation of Minimally-Invasive Surgery with Cutting Based on Boundary Element Methods, Medical Image Computing and Computer-Assisted Intervention - MICCAI 2001, Lecture Notes in Computer Science, Volume 2208, pp. 1263-1264.

[17] C. Monserrat, U. Meier, M. Alcaniz, F. Chinesta, M.C. Juan, A new approach for the real-time simulation of tissue deformations in surgery simulation, Computer Methods and Programs in Biomedicine, 64(2001), pp. 77-85.

[18] P. Wang, A.A. Becker, I.A. Jones, A.T. Glover, S.D. Benford, C.M. Greenhalgh, M. Vloeberghs, Virtual reality simulation of surgery with haptic feedback based on the boundary element method, Computers and Structures, 85(2007), pp. 331-339.

[19] Foster, Timothy,Mark, 2013, Real-time stress analysis of three-dimensional boundary element problems with continuously updating geometry, Doctoral thesis, Durham University.

[20] Robert G Aykroyd, Brian A Cattle, Robert M West, A boundary element approach for real-time monitoring and control from electrical resistance tomographic data, 4th World Congress on Industrial Process Tomography, Aizu, Japan.

[21] P Wang, A A Becker, I A Jones, A T Glover, S D Benford, M Vloeberghs, 2009, Real-time surgical simulation for deformable soft-tissue objects with a tumour using Boundary Element techniques, Journal of Physics: Conference Series, Volume 181, Number 1, doi:10.1088/1742-6596/181/1/012016

[22] Bo Zhu, Lixu Gu, 2012, A hybrid deformable model for real-time surgical simulation, Computerized

Medical Imaging and Graphics, 36(2012), pp. 356-365.
[23] Martin Bayliss, 2003, The Numerical Modelling of Elastomers, PhD Thesis, School of Engineering, Cranfield University.
[24] Alex Rhomberg, 2001, Real-time finite elements: A parallel computer application, Doctoral Thesis, Swiss Federal Institue of Technology, Zurich, Publisher: Shaker, http://dx.doi.org/10.3929/ethz-a-004089534
[25] M. Kreienmeyer, E. Stein, 1995, Parallel implementation of the boundary element method for linear elastic problems on a MIMD parallel computer, Computational Mechanics, 15(1995), pp. 342-349.
[26] Kirana Kumara P, 2012, A MATLAB Code for Three Dimensional Linear Elastostatics using Constant Boundary Elements, International Journal of Advances in Engineering Sciences(IJAES), 2(3), pp. 9-20.
[27] W. T. Ang, 2007, A Beginner's Course in Boundary Element Methods, Universal Publishers, Boca Raton, USA.
[28] Gernot Beer, Ian Moffat Smith, Christian Duenser, 2008, The Boundary Element Method with Programming: For Engineers and Scientists, Springer
[29] C.A. Brebbia, J. Dominguez, Boundary Elements - An Introductory Course, Second Edition, WIT Press
[30] Brebbia CA, 1978, The boundary element method for engineers, London/New York, Pentech Press/Halstead Press
[31] Youssef F. Rashed, Available from: http://www.boundaryelements.com/index.php?option=com_content&view=category&layout=blog&id=13&Itemid=44 (accessed October 22, 2013)
[32] M. Guiggiani, A. Gigante, 1990, A General Algorithm for Multidimensional Cauchy Principal Value Integrals in the Boundary Element Method, Journal of Applied Mechanics, Transactions of the ASME, Vol. 57, December 1990, pp. 906-915.
[33] M. Guiggiani, Formulation and numerical treatment of boundary integral equations with hypersingular kernels, Contained in: "Singular Integrals in B. E. Methods, V. Sladek and J. Sladek, eds., 1998"
[34] www.efunda.com (accessed October 22, 2013)
[35] Errata, http://www.ntu.edu.sg/home/mwtang/bem2011.html#errata (accessed October 22, 2013)
[36] Alastair McKinstry, Alin Elena, Ruairi Nestor, An Introduction to Fortran, Available from: www.ichec.ie/support/tutorials/fortran.pdf (accessed October 22, 2013)
[37] Blaise Barney, Message Passing Interface (MPI), Available from: https://computing.llnl.gov/tutorials/mpi/ (accessed October 22, 2013)
[38] Intel Math Kernel Library Reference Manual, Available from: http://software.intel.com/sites/products/documentation/hpc/mkl/mklman/index.htm (accessed October 22, 2013)
[39] ScaLAPACK Example Programs, Available from: http://www.netlib.org/scalapack/examples/ (accessed October 22, 2013)
[40] ScaLAPACK - Scalable Linear Algebra PACKage, Available from: http://www.netlib.org/scalapack/ (accessed October 22, 2013)
[41] Parallel ESSL Guide and Reference, Available from: http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/topic/com.ibm.cluster.pessl.v4r2.pssl100.doc/am6gr_pguide.htm (accessed October 22, 2013)
[42] Simulation Acceleration using System Objects, MATLAB Coder and Parallel Computing Toolbox, Documentation Center, MathWorks, Available from: http://www.mathworks.in/help/comm/examples/simulation-acceleration-using-system-objects-matlab-coder-and-parallel-computing-toolbox.html (accessed October 30, 2013)
[43] Mark Adams, James W. Demmel, 1999, Parallel Multigrid Solver for 3D Unstructured Finite Element Problems, ACM 1-58113-091-8/99/0011, Available from: www.cs.illinois.edu/~snir/PPP/mesh/multigrid3.pdf (accessed October 30, 2013)
[44] Warren C. Young, Richard G. Budynas, 2002, Roark's Formulas for Stress and Strain, Seventh Edition, McGraw-Hill, Available from: http://www.google.co.in/url?q=http://materiales.azc.uam.mx/gjl/Clases/MA10_I/Roark's%2520formulas%2520for%2520stress%2520and%2520strain.pdf&sa=U&ei=3JFwUvu0J8fQrQfCpY0CC0QFjAF&usg=AFQjCNETpC7wyT5V60kDsDdATx-Of_VC2Q (accessed October 30, 2013)
[45] http://biorobotics.harvard.edu/ (accessed October 30, 2013)
[46] Kirana Kumara P, Ashitava Ghosal, 2010, A Procedure for the 3D Reconstruction of Biological Organs from 2D Image Sequences, International Conference on Biomedical Engineering and Assistive Technologies (BEATs-2010), 17-19 December, 2010, Dr. B R Ambedkar National Institute of Technology, Jalandhar, India.
[47] Kirana Kumara P, 2012, Extracting Three Dimensional Surface Model of Human Kidney from the Visible Human Data Set using Free Software, Leonardo Electronic Journal of Practices and Technologies,

11(20), pp. 115-126.
[48] Kirana Kumara P, 2012, Demonstrating the Usefulness of CAELinux for Computer Aided Engineering using an Example of the Three Dimensional Reconstruction of a Pig Liver, International Journal of Advancements in Technology, 3(4), pp. 301-309.
[49] Kirana Kumara P, 2011, Reconstructing Solid Model from 2D Scanned Images of Biological Organs for Finite Element Simulation, Google Knol, Available from: http://knol.google.com/k/kirana-kumara-p/reconstructing-solid-model-from-2d/3rc2kfwq179j2/5#
[50] Valerie E. Taylor, Milana Huang, Thomas Canfield, Rick Stevens, Daniel Reed, Stephen Lamm, Performance Modeling of Interactive, Immersive Virtual Environments for Finite Element Simulations, International Journal of High Performance Computing Applications, June 1996, vol. 10, no. 2-3, pp. 145-156, doi: 10.1177/109434209601000203
[51] Sonya Allin, Yoky Matsuoka, Roberta Klatzky, 2002, Measuring Just Noticeable Differences For Haptic Force Feedback: Implications for Rehabilitation, Proceedings of the 10th Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems (HAPTICS '02), IEEE Computer Society
[52] H. Pongrac, B. Farber, P. Hinterseer, J. Kammerl, E. Steinbach, 2006, Limitations of human 3D-force discrimination, Human-Centered Robotics Systems
[53] Hong Z. Tan, Brian Eberman, Mandayam A. Srinivasan, Belinda Cheng, 1994, Human Factors for the Design of Force-reflecting Haptic Interfaces, DSC-Vol.55-1, Dynamic Systems and Control, Editor: Clark J. Radcliffe, Book No. G0909A-1994 The American Society of Mechanical Engineers
[54] Suvranu De, Dhannanjay Deo, Ganesh Sankaranarayanan, Venkata S. Arikatla, 2011, A Physics-Driven Neural Networks-based Simulation System (PhyNNeSS) for Multimodal Interactive Virtual Environments Involving Nonlinear Deformable Objects, Presence, Vol. 20, No. 4, August 2011, 289-308.
[55] L. M. Patnaik, 1995, Overview of neurocomputers, Current Science, Vol 68, No 2, 25 January 1995, pp. 156-162.
[56] Kirana Kumara P, 2013, A study on the usefulness of Support Vector Machines for the realtime computational simulation of soft biological organs, Seventh M.I.T. Conference on Computational Fluid and Solid Mechanics – Focus: Multiphysics & Multiscale, June 12-14, 2013, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, Available from: http://eprints.iisc.ernet.in/46772/