ROBOMECH Journal
a SpringerOpen Journal

RESEARCH ARTICLE
Open Access

GPU-accelerated surgery simulation for opening a brain fissure

Kazuya Sase[1*], Akira Fukuhara[2], Teppei Tsujita[3] and Atsushi Konno[1]

Abstract

In neurosurgery, dissection and retraction are basic techniques for approaching the site of pathology. These techniques are carefully performed in order to avoid damage to nerve tissues or blood vessels. However, novice surgeons cannot train in such techniques using the haptic cues of existing training systems. This paper proposes a real-time simulation scheme for training in dissection and retraction when opening a brain fissure, which is a procedure for creating a working space before treating an affected area. In this procedure, spatulas are commonly used to perform blunt dissection and brain tissue retraction. In this study, the interaction between spatulas and soft tissues is modeled on the basis of a finite element method (FEM). The deformation of soft tissue is calculated according to a corotational FEM by considering geometrical nonlinearity and element inversion. A fracture is represented by removing tetrahedrons using a novel mesh modification algorithm in order to retain the manifold property of a tetrahedral mesh. Moreover, most parts of the FEM are implemented on a graphics processing unit (GPU). This paper focuses on parallel algorithms for matrix assembly and matrix rearrangement related to FEM procedures by considering a sparse-matrix storage format. Finally, two simulations are conducted. A blunt dissection simulation is conducted in real time (less than 20 ms for a time step) using a soft-tissue model having 4807 nodes and 19,600 elements. A brain retraction simulation is conducted using a brain hemisphere model having 8647 nodes and 32,639 elements with force feedback (less than 80 ms for a time step). These results show that the proposed method is effective in simulating dissection and retraction for opening a brain fissure.

Keywords: Surgery simulation, Finite element method, GPGPU

Background

Virtual reality (VR) surgery simulation is a safe and efficient approach to surgical training. In the last two decades, laparoscopic surgery simulators have been widely developed to support efficient training in surgical techniques. In contrast, neurosurgery simulators have not been investigated extensively. However, recent years have witnessed some significant advancements in the development of neurosurgery simulators [1, 2]. Neurosurgery requires surgeons to perform precise operations. Because surgeons rely on haptic cues in various contexts, the physics of soft-tissue deformation should be reliable not only for graphical rendering but also for haptic rendering.

One of the basic procedures in neurosurgery is the opening of a brain fissure, which is necessary to create a working space to access an affected area. Figure 1 shows a schematic of this procedure. In order to approach an affected area located at the bottom of the fissure, surgeons cut connective tissues such as the arachnoid membrane and arachnoid trabeculae using microscissors and push the brain tissues apart to keep the tissues open using spatulas [3]. Aspirators are used to apply tension to the membrane and remove blood. It is known that the position of the spatulas and the pushing force are related to the patient's prognosis [4]. An early simulator focusing on retraction in neurosurgery was the virtual retractor developed by Koyama et al. [5]. It modeled the deformations of intracranial vessels using geometrical theory, but physical consistency was not considered. Hansen et al. developed a real-time simulator for brain retraction [6]. They adopted a finite element

*Correspondence: sase@scc.ist.hokudai.ac.jp
[1] Graduate School of Information Science and Technology, Hokkaido University, Kita 14 Nishi 9, Kita-ku, Sapporo, Japan
Full list of author information is available at the end of the article

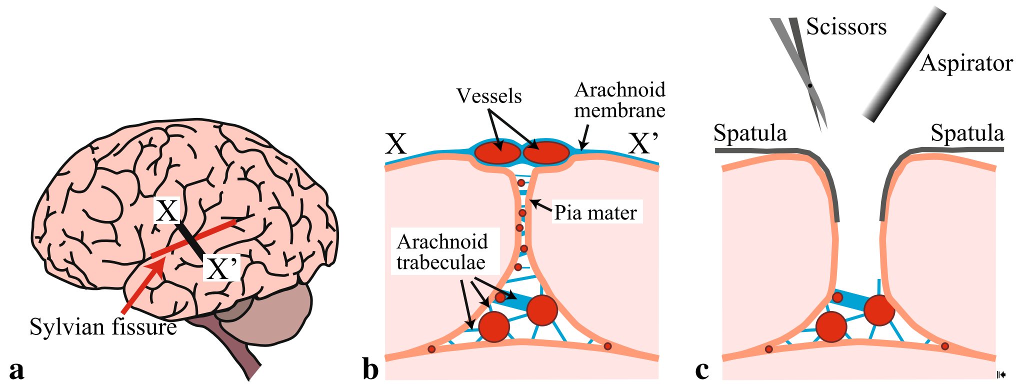


Fig. 1 Schematic of the opening of a Sylvian fissure. **a** Sylvian fissure, **b** cross-sectional view of *X-X'*, and **c** retraction and dissection using spatulas, scissors, and an aspirator

method (FEM) for calculating the deformation of brain tissues and the reaction force. However, the resolution of the mesh was limited to less than a thousand nodes because of its high computational cost. Hasegawa et al. conducted a cerebellar retraction simulation using a high-resolution model [7]. They considered the nonlinear viscoelastic behavior of soft tissues. However, their optimization method was inadequate for realizing real-time simulation.

To develop a haptic simulator for opening a brain fissure, the following system specifications are required.

- The refresh rate of the physics simulation must be greater than 30 Hz for smooth animation [8].
- The positions of the surgical instruments are input by haptic devices, and the reaction force must be returned immediately. Ideally, the refresh rate of the force feedback should be greater than 1 kHz for interaction with stiff materials [8].
- The number of nodes of target finite element model (brain hemisphere) should be approximately 10,000.
- The ability to perform connective-tissue dissection should be present [8].
- The soft tissues should have physically correct behavior. Brain tissues are known to have complex mechanical properties; for example, white matter is known to have anisotropic visco-hyperelasticity [9].

Generally, there is a compromise between the precision and the speed for the computation of soft-tissue physics. Therefore, we firstly simplify the mechanical properties of soft tissues and aim to develop a visually acceptable simulator with a refresh rate greater than 30 Hz for the physics simulation.

To fulfill the above-mentioned requirements, we firstly formulated the framework of the interactive simulation using a linear FEM and combined it with haptic devices [10]. In order to accelerate the calculation, we developed an algorithm for collision detection that is implemented on a graphics processing unit (GPU) [11]. However, the computational speed is not sufficient to run in real-time. In particular, the FEM solver was not optimized for calculation on the GPU, and it became the bottleneck of the simulation.

In this paper, an efficient implementation of an FEM solver is proposed to simulate the opening of a brain fissure. The contributions of this paper are the following:

- A GPU-accelerated method of a corotational linear FEM including a boundary-condition-based collision response is proposed.
- A stabilization algorithm for the fracture simulation based on element removal is incorporated into the proposed GPU-accelerated FEM framework.

In the conventional FEM implementation, the most time-consuming procedures are assembling the stiffness matrix and solving linear equations. Moreover, the calculation of the collision response requires additional procedures. In the FEM solver proposed in this paper, the collision response is calculated by applying geometrical boundary conditions. Because conventional FEM solvers do not consider the frequent changes in the geometrical boundary conditions, efficient matrix assembly and matrix rearrangement have not attracted much attention. In this paper, the three above-mentioned procedures, i.e., assembling the stiffness matrix, solving linear equations, and rearranging the stiffness matrix, are implemented by considering a sparse-matrix storage format ("Implementation and GPU parallelization"). All of these algorithms are implemented on a GPU. However, the proposed algorithms assume that the mesh topology of analysis area is constant during a simulation. Therefore, additions of new nodes and changes in the node connectivity are not permitted. In order to realize the ability to dissect under these limitations, we adopt a fracture representation on the basis of the element removal approach. In this approach, fracture behaviors are calculated by inactivating the physical contributions of removed elements. However, it has been reported that the dynamic behaviors become unstable, which may lead to diverse the numerical calculation when the tetrahedral mesh becomes nonmanifold because of element removals [12]. To solve the stability problem, we also propose an element removal algorithm to avoid topological singularities ("Modeling of dissection"). The concept of proposed algorithm is active element removal. Its implementation is considerably simpler and can be used as alternative to existing methods such as that in [13]. Finally, to evaluate the performance of these implementations, a blunt dissection and brain retraction are simulated, and the results of these simulations are presented ("Results").

Related work

Collision response

Recent cutting-edge studies on real-time haptic rendering for interacting deformable objects have focused on efficient contact handling including collisions between multiple deformable objects and self-collisions [14, 15]. They modeled contacts based on Signorini's law and Coulomb's law, and the linear or nonlinear complementarity problem needs to be solved. Although their algorithms are highly optimized for a GPU, the number of nodes is limited to several thousand for realizing real-time simulation because of their high computational costs.

The fastest implementation of collision response is the penalty method [16]. In this method, external forces are applied to contact nodes according to the penetration depth. To compute the magnitude of the external forces, scalar coefficients are required for multiplication with the depth. However, it is difficult to determine the scalar values for obtaining a stable response.

The other approach is the position constraint method. In this method, nodal displacements are directly input according to geometric relations. In an early study on a real-time surgery simulator, Cotin et al. introduced a position constraint formulation using the Lagrange multiplier method [17]. Hirota et al. adopted a boundary-condition-based constraint [18]. Although these method are essentially equivalent and both methods resulting in a large simultaneous linear equations, the method of Hirota et al. has the advantage in the term of the size of the linear equations (see "Collision response of soft tissues"). The limitation of this method is that accurate contact response subject to Signorini's law or Coulomb's law cannot be computed. However, because the computational cost is lower than that of the accurate methods, the position constraint method can be used for the analysis of a fine detailed mesh.

As mentioned above, the position constraint method based on a boundary condition involves a large number of simultaneous linear equations. Because the linear system is large and sparse, the sparse matrix format should be adopted for efficient execution of mathematical operations and to reduce memory consumption. However, its GPU-optimized implementations considering the sparse matrix format have not been discussed.

Fracture

In the field of computer graphics, several methods of fracture simulation have been discussed [19–21]. This section describes the details of fracture algorithms developed for real-time applications.

A notable approach is the extended FEM [22], which allows for representation of any crack without the limitations of a mesh topology by adding a shape function to the element displacement field interpolations. Thus, an additional degree of freedom (DOF) is provided to model crack discontinuities.

Mor and Kanade modeled the knife cutting of soft objects [23]. They proposed split patterns in which a tetrahedron is split into smaller tetrahedrons according to the knife path. In this approach, the tetrahedral mesh is explicitly modified by adding new nodes to it.

Delingette et al. proposed an element removal approach in the early years of surgery simulation studies [24]. Even though this approach suffers from the disadvantage of a loss of volume, it offers advantages such as a low computational cost and simple implementation. In particular, we focus on the fact that this algorithm does not require the addition of add new nodes. Thus, simulations can be performed at the same computational cost throughout. Because real-time characteristics are important for practical use of the surgery simulator, we adopt this element removal approach.

As shown by Forest, element removal may lead a tetrahedral mesh to become a nonmanifold geometry [12], which means that the tetrahedral mesh has vertices or edges where the thickness of the volumetric mesh cannot be defined. Vertices and edges are known as singular vertices and singular edges, respectively, and such a singularity is known as a topological singularity (Fig. 2). Because the dynamic behavior can be unstable when the FEM mesh becomes a nonmanifold geometry, topological singularities should be avoided during simulation.

Some topological-singularity avoidance algorithms have been proposed in the literature. Forest et al. proposed a node separation algorithm [12]. They separated

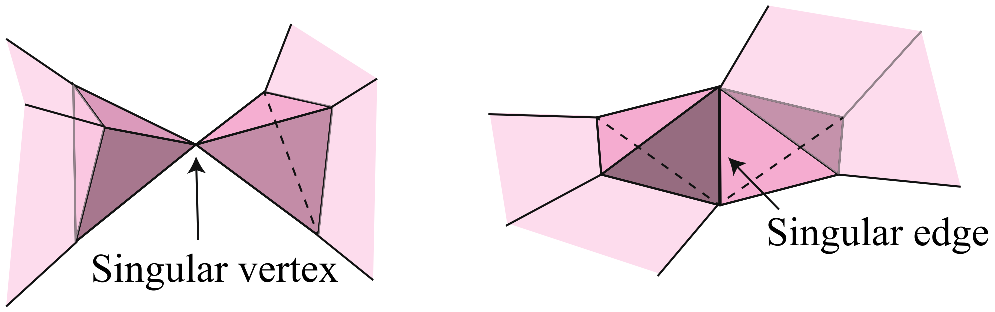


Fig. 2 Topological singularity

the singular vertices and singular edges by adding copies of such vertices and edges. However, this approach increases the computational cost because the DOFs of the system increase with the addition of nodes. Nakayama et al. proposed a delay algorithm that suspends the removal of elements that cause topological singularities [13]. However, the delay algorithm does not correctly simulate actual fracture phenomena. In the actual case, the stress will be concentrated at a singular vertex and singular edge. Thus, the two elements connected by a singular vertex or singular edge will be disconnected. Therefore, such elements should be removed immediately.

Finite element model of the brain

Corotational FEM

The corotational formulation is an approximate approach that considers the geometrical nonlinearity for small strain deformations. This formulation evaluates element strains with respect to the rotated element coordinates, which are referred to as corotational coordinates (see Fig. 3a). A corotational coordinate is a coordinate that is rotated using the rotation component of the current deformed configuration. A corotational FEM is a reasonable choice for achieving a trade-off between precision and computational cost [25]. In general, the element stiffness equation is defined as

$$\boldsymbol{f}^e_{\text{ext}} = \mathbf{K}^e \boldsymbol{u}^e, \tag{1}$$

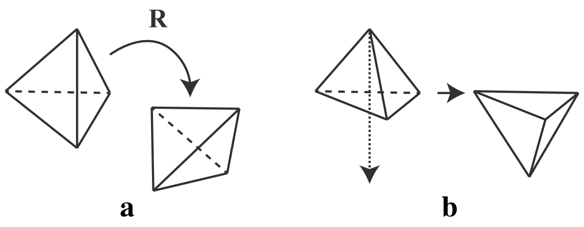


Fig. 3 Deformation of a tetrahedral element. **a** Geometrical nonlinearity. **b** Inversion of an element

where $\boldsymbol{f}^e_{\text{ext}}$ and $\boldsymbol{u}^e$ are the nodal force and displacement vectors of an element, respectively. The element displacement vector is defined as $\boldsymbol{u}^e = \boldsymbol{x}^e - \boldsymbol{x}^e_0$, where $\boldsymbol{x}^e$ and $\boldsymbol{x}^e_0$ are the displaced and initial nodal position vectors of the element, respectively. If the rotation of the element coordinates is represented by a rotation matrix $\mathbf{R} \in \mathbb{R}^{3\times 3}$, the displaced position and force vectors are transformed by $\mathbf{R}$ and the element stiffness equation becomes

$$\mathbf{R}^{e\mathrm{T}} \boldsymbol{f}^e_{\text{ext}} = \mathbf{K}^e_0 \left(\mathbf{R}^{e\mathrm{T}} \boldsymbol{x}^e - \boldsymbol{x}^e_0 \right), \tag{2}$$

where $\mathbf{K}^e_0$ is the element stiffness matrix of a linear FEM and $\mathbf{R}^e \triangleq \text{blockdiag}\,[\mathbf{R}, \mathbf{R}, \mathbf{R}, \mathbf{R}]$. The above equation is rewritten as

$$\boldsymbol{f}^e_{\text{ext}} = \mathbf{K}^e \boldsymbol{x}^e - \boldsymbol{f}^e_0, \tag{3}$$

where

$$\mathbf{K}^e = \mathbf{R}^e \mathbf{K}^e_0 \mathbf{R}^{e\mathrm{T}}, \tag{4}$$

$$\boldsymbol{f}^e_0 = \mathbf{R}^e \mathbf{K}^e_0 \boldsymbol{x}^e_0; \tag{5}$$

$\boldsymbol{f}^e_0$ is known as the force offset vector.

$\mathbf{R}$ is calculated by singular value decomposition (SVD) of deformation gradient. The details are described in "Appendix A: Rotation of a deformed tetrahedron".

Matrix/vector assembly

After $\mathbf{K}^e$ and $\boldsymbol{f}^e_0$ are obtained, the global stiffness matrix $\mathbf{K} \in \mathbb{R}^{3N_{\text{node}} \times 3N_{\text{node}}}$ and global force offset vector $\boldsymbol{f}_0 \in \mathbb{R}^{3N_{\text{node}}}$, where N_{node} is the number of nodes, are calculated by gathering all element contributions using connectivity information. This procedure is known as matrix/vector assembly, and it is expressed as

$$\mathbf{K} = \sum_e \mathbf{L}^{e\mathrm{T}} \mathbf{K}^e \mathbf{L}^e, \tag{6}$$

$$\boldsymbol{f}_0 = \sum_e \mathbf{L}^{e\mathrm{T}} \boldsymbol{f}^e_0, \tag{7}$$

where $\mathbf{L}^e \in \mathbb{R}^{12\times 3N_{\mathrm{node}}}$ is the gather matrix, which gathers element nodal data from global vectors. $\mathbf{L}^e$ is a Boolean matrix, which consists of zeros and ones. Note that Eqs. (6) and (7) are just mathematical formulations; the actual assembly process is implemented in a more efficient manner. The efficient implementations are described in "Efficient matrix/vector assembly in a sparse storage format".

The global stiffness equation is written as

$$\boldsymbol{f}_{\mathrm{ext}} = \mathbf{K}\boldsymbol{x} - \boldsymbol{f}_0, \tag{8}$$

where $\boldsymbol{f}_{\mathrm{ext}}$ and $\boldsymbol{x}$ are the global external force vector and global position vector, respectively. For simplicity, this equation can be written by analogy to the global stiffness equation of a linear FEM as

$$\boldsymbol{f} = \mathbf{K}\boldsymbol{x}, \tag{9}$$

where $\boldsymbol{f} = \boldsymbol{f}_{\mathrm{ext}} - \boldsymbol{f}_0$.

Collision response of soft tissues

When a surgical instrument contacts with the brain model, it is assumed that the contact nodes move together with the instrument (Fig. 4). Hence, the displacements of the contact nodes are known, but their contact forces are unknown. In contrast, the displacements of free nodes and internal nodes are unknown, but their external forces are known (they are zeros). Therefore, the nodes are rearranged into displacement-known nodes and force-known nodes, and Eq. (9) can be modified by rearranging the matrix and vectors as follows [18]:

$$\begin{bmatrix} \boldsymbol{f}_{\mathrm{f}} \\ \boldsymbol{f}_{\mathrm{d}} \end{bmatrix} = \begin{bmatrix} \mathbf{K}_{\mathrm{ff}} & \mathbf{K}_{\mathrm{fd}} \\ \mathbf{K}_{\mathrm{df}} & \mathbf{K}_{\mathrm{dd}} \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_{\mathrm{f}} \\ \boldsymbol{x}_{\mathrm{d}} \end{bmatrix}, \tag{10}$$

where the suffixes f and d denote the components of the force-known and displacement-known nodes, respectively. As shown in Fig. 4, the contact nodes are geometrically constrained on the surface of a rigid body (surgical instrument). Hence, the displacements of the contact nodes are known, whereas the forces are unknown. In contrast, the forces generated at unconstrained nodes are zero under the stationary condition, whereas the displacements are unknown. In Eq. (10), $\boldsymbol{f}_{\mathrm{d}}$ and $\boldsymbol{x}_{\mathrm{f}}$ are unknown, and $\boldsymbol{x}_{\mathrm{f}}$ is obtained by solving the following linear equation:

$$\mathbf{K}_{\mathrm{ff}}\boldsymbol{x}_{\mathrm{f}} = \boldsymbol{f}_{\mathrm{f}} - \mathbf{K}_{\mathrm{fd}}\boldsymbol{x}_{\mathrm{d}}. \tag{11}$$

After $\boldsymbol{x}_{\mathrm{f}}$ is obtained, the other unknown value $\boldsymbol{f}_{\mathrm{d}}$ is calculated as

$$\boldsymbol{f}_{\mathrm{d}} = \mathbf{K}_{\mathrm{df}}\boldsymbol{x}_{\mathrm{f}} + \mathbf{K}_{\mathrm{dd}}\boldsymbol{x}_{\mathrm{d}}. \tag{12}$$

Although only the formulation of the static FEM is described above, the adoption of the formulation of the dynamic FEM with implicit time integration introduces a mathematically similar equation to the formulation of static FEM. The details of the formulation of the dynamic FEM are presented in "Appendix B: Formulation of a dynamic FEM".

Implementation and GPU parallelization

This section describes the implementation of the FEM. To measure the computational time, we used a CIARA KRONOS S810R workstation, which employs an Intel Core i7-3960X (six cores, overclocked to 4.5 GHz) CPU with 64 GB of RAM and two GPUs, an NVIDIA K20c (2,496 CUDA cores) for general-purpose computing and an NVIDIA Quadro K5000 (1536 CUDA cores) for graphics processing. Parallel processing is implemented using OpenMP for multithread computing on a multi-core CPU and NVIDIA CUDA for general-purpose computing on GPUs (GPGPU).

Simulation procedures

The flowchart of our simulation scheme is shown in Fig. 5. Before the real-time simulation loop, the element stiffness matrices of the linear elasticity $\mathbf{K}_0^e$ and the reduction lists described in "Efficient matrix/vector assembly in a sparse storage format" are calculated. The real-time simulation procedures are as follows.

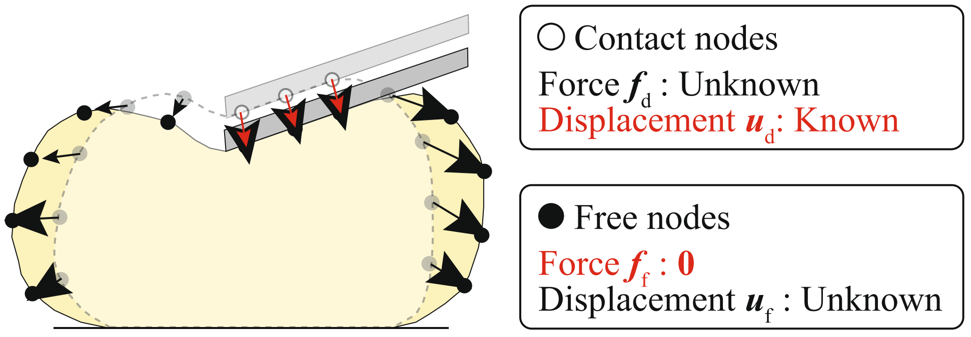

○ Contact nodes
Force $\boldsymbol{f}_{\mathrm{d}}$: Unknown
Displacement $\boldsymbol{u}_{\mathrm{d}}$: Known
● Free nodes
Force $\boldsymbol{f}_{\mathrm{f}}$: $\mathbf{0}$
Displacement $\boldsymbol{u}_{\mathrm{f}}$: Unknown

Fig. 4 Contact nodes and free nodes

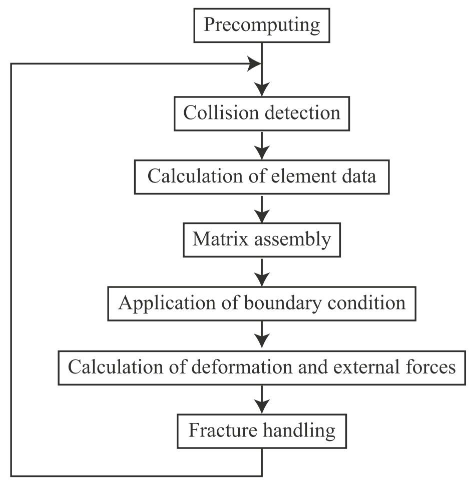

Precomputing
Collision detection
Calculation of element data
Matrix assembly
Application of boundary condition
Calculation of deformation and external forces
Fracture handling

Fig. 5 Flowchart of the simulation scheme

Calculation of element data: $\mathbf{R^e}$, $\mathbf{K^e}$, and $\boldsymbol{f}_0^e$ are calculated. The details of these parallel implementations are described in "Calculation of element data".

Matrix/vector assembly: $\mathbf{K}$ and $\boldsymbol{f}_0$ are assembled. The details of the parallelization of the assembly are described in "Efficient matrix/vector assembly in a sparse storage format".

Collision detection: Collision detection between a deformable object (brain) and rigid objects (brain spatulas) is executed. The contact nodes of the deformable object and the corresponding forced displacements are determined. The discrete collision detection approach reported in [11] is adopted. This method can deal with collisions between a nonconvex deformable object and a rigid object.

Application of boundary condition: On the basis of collision detection, a boundary condition is set. As mentioned in "Collision response of soft tissues", a large sparse matrix is rearranged according to the boundary condition. The implementation details are described in "Matrix rearrangement".

Calculation of the deformation and external forces: The calculation of the deformation is a problem involving a system of linear equations. The linear equations are solved by the conjugate gradient method. Sparse-matrix dense-vector multiplications are implemented by the sparse-matrix library CUSPARSE provided by NVIDIA Corp.

Calculation of element data

An element stiffness matrix $\mathbf{K}^e$ is calculated using Eq. (4). It is assumed that the materials are isotropic; hence, $\mathbf{K}^e$ is a symmetric matrix. Therefore, it is sufficient to store the elements of the upper triangular matrix of $\mathbf{K}^e$. Further, $\boldsymbol{f}_0^e$ is calculated in the same way as $\mathbf{K}^e$ (Eq. 5). Finally, all $\mathbf{K}^e$ ($\mathbf{K}^1, \mathbf{K}^2, \ldots, \mathbf{K}^{N_{\text{elem}}}$) and $\boldsymbol{f}_0^e$ ($\boldsymbol{f}_0^1, \boldsymbol{f}_0^2, \ldots, \boldsymbol{f}_0^{N_{\text{elem}}}$), where N_{elem} is the number of tetrahedral elements, are serialized and stored in the arrays *valuesKe* and *valuesF0e*, respectively. These procedures are implemented in parallel using one thread per element.

Efficient matrix/vector assembly in a sparse storage format

In the matrix/vector assembly procedure, the element stiffness matrices are assembled into the global stiffness matrix, as described in Eq. (6), and the element force offset vectors are assembled into the global force offset vector, as described in Eq. (7). First, the implementation of global stiffness matrix assembly is described, and then, that of global force offset vector assembly is described.

In most FEM problems, the global stiffness matrix is a large sparse matrix that is stored in a sparse-matrix storage format to reduce memory consumption. In this work, the global stiffness matrix is stored using the coordinate list (COO) sparse storage format during matrix assembly. This is because of the requirement of matrix rearrangement described in "Matrix rearrangement". The COO consists of three arrays: *values*, *rowIndices*, and *columnIndices*. In the COO format, only the nonzero elements of a sparse matrix are stored in the array *values*. The row and column indices of the nonzero elements are stored in the arrays *rowIndices* and *columnIndices*, respectively. These arrays are stored in row-major order. For example, the matrix

$$\begin{bmatrix} a & b & 0 \\ 0 & c & 0 \\ 0 & 0 & d \end{bmatrix}$$

is stored as the following three arrays.

$$\begin{aligned} values &= [a, b, c, d] \\ rowIndices &= [0, 0, 1, 2] \\ columnIndices &= [0, 1, 1, 2] \end{aligned}$$

In the remainder of this section, *values*, *rowIndices*, and *colmunIndices* represent the arrays of $\mathbf{K}$ in the COO format.

In order to implement a fast matrix assembly algorithm, we adopted the reduction list approach proposed in [26]. When the mesh topology does not change during simulation, *rowIndices* and *columnIndices* are constant matrices. Hence, only *values* should be updated at every time step. Matrix assembly involves a number of independent summations expressed as

$$values(i) = \sum_{j=1}^{N_{\text{src},i}} valuesKe(srcIndices_i(j)), \tag{13}$$

where $srcIndices_i$ is an array that stores the source indices pointing to the components of *valuesKe* for the summation of the i-th component of *values*, and $N_{\mathrm{src},i}$ is the number of components of $srcIndices_i$. The components of *srcIndices* are determined by the connectivity of a tetrahedral mesh. A reduction list stores the source indices and a negative-signed destination index as

$$reductionList_i = [srcIndices(1), srcIndices(2), \ldots, srcIndices(N_{\mathrm{src},i}), -i]. \tag{14}$$

As shown in Eq. 14, a reduction list has one destination index in general. On the other hand, because we assume that the material is isotropic and **K** is a symmetric matrix, the reduction list can store two destination indices as

$$reductionList_i = [srcIndices(1), srcIndices(2), \ldots, srcIndices(N_{\mathrm{src},i}), -i, -lower(i)], \tag{15}$$

where $lower(i)$ is an index pointing to the lower component of $values(i)$. An example of reduction-array-based summation is shown in Fig. 6. For a symmetric matrix, there is a storage format that stores only upper or lower triangular entries as in the case of $\mathbf{K}^e$. However, the adoption of such a storage format for **K** requires special treatment in the subsequent procedures, which might degrade the maintainability because of its complexity. Thus, all the nonzero components of **K** are stored using this reduction procedure. This approach avoids atomic operation because the output memories are independent of each other as in the case of the general reduction list approach.

The assembly of $\boldsymbol{f}_0$ is implemented in a similar manner. After the calculation of $\mathbf{R}^e$, all values of $\boldsymbol{f}_0^e$ are stored as an array. The reduction list for the assembly of $\boldsymbol{f}_0$ is constructed in advance. The reduction is performed in a thread per component of $\boldsymbol{f}_0$, which allows for the calculation of $\boldsymbol{f}_0$ without atomic operation. This reduction is independent of the assembly of **K**. Therefore, assemblies of **K** and $\boldsymbol{f}_0$ are performed concurrently, e.g., on two GPUs.

Matrix rearrangement

As described in "Collision response of soft tissues", collisions are represented by geometrical boundary conditions and the global stiffness matrix is rearranged by considering the boundary conditions. This rearrangement procedure involves permutation and separation processes.

Permutation is performed by referring to a permutation list that includes the source index i and destination index $list(i)$. The list is constructed according to the boundary conditions. Figure 7 shows an example of permutation in the case of one tetrahedron, in which nodes 0, 2, and 3 are constrained. The permutation procedure accumulates the variables of the contact nodes at the head of the array and those of the free nodes at the bottom. Permutation using a permutation list is represented as $m'(list(i), list(j)) = m(i,j)$, where m and m' represent the source matrix and permutated matrix, respectively, and $m(i, j)$ denotes the i, j component of matrix m. If a matrix is stored as a dense matrix, permutation is performed by simply copying the source components to the destinations. However, if the matrix is stored in a sparse storage format, the implementation of permutation differs according to the storage format.

In this work, the global stiffness matrix is stored in the COO format, as mentioned in "Efficient matrix/vector assembly in a sparse storage format". In the COO format, permutation is easily and efficiently performed as

$$\begin{aligned} rowIndices(i) &= list(rowIndices(i)), \\ columnIndices(i) &= list(columnIndices(i)). \end{aligned} \tag{16}$$

These operations do not conflict with each other, and all permutations are performed in parallel. Other sparse storage formats such as compressed sparse row (CSR) are also widely used. The CSR format can be constructed by compressing *rowIndices* used in the COO format. This approach can further reduce the memory consumption compared to the COO format, and it is suitable for parallelizing matrix–vector multiplication. However, the

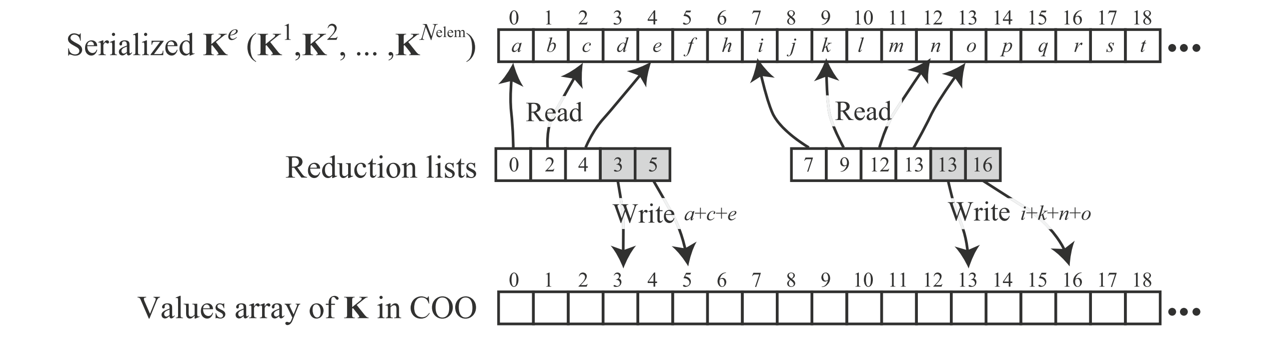


Fig. 6 Reduction array

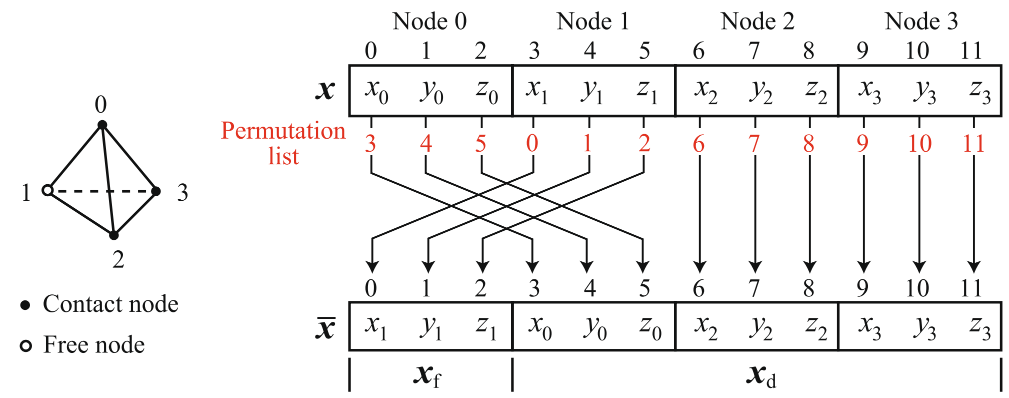

Node 0
Node 1
Node 2
Node 3
0 1 2 3 4 5 6 7 8 9 10 11
$\boldsymbol{x}$
x_0 y_0 z_0 x_1 y_1 z_1 x_2 y_2 z_2 x_3 y_3 z_3
Permutation list
3 4 5 0 1 2 6 7 8 9 10 11
$\bar{\boldsymbol{x}}$
x_1 y_1 z_1 x_0 y_0 z_0 x_2 y_2 z_2 x_3 y_3 z_3
$\boldsymbol{x}_\mathrm{f}$
$\boldsymbol{x}_\mathrm{d}$
● Contact node
○ Free node

Fig. 7 Permutation list

implementation of permutation is not easier than that of the COO format. Although it can be realized using the permutation matrix $\mathbf{P}$ as $\mathbf{M}' = \mathbf{P}^\mathrm{T}\mathbf{M}\mathbf{P}$, this implementation is not efficient because the memory traffic increases and additional arithmetic operations are required compared to the COO format. Therefore, the COO format was selected as the sparse storage format in this work.

Sorting should be performed to maintain the row and column index arrays in ascending order; however, the sorting process can be combined with a separation process. An overview of the entire procedure including permutation and separation is shown in Fig. 8 by taking a matrix $\mathbf{A}$ as an example. After Eq. (16) is executed, three arrays are sorted by *columnIndices*. Next, $\mathbf{A}$ is separated along its columns into $\mathbf{A}_\mathrm{f}$ and $\mathbf{A}_\mathrm{d}$. When the separation index is N_f, entries whose column index is less than N_f are copied to $\mathbf{A}_\mathrm{f}$. The other entries are copied to $\mathbf{A}_\mathrm{d}$. In order to fix *columnIndices* to zero-base indices, N_f is subtracted from all of the components of *columnIndices* in $\mathbf{A}_\mathrm{d}$. Subsequently, three arrays, namely *values*, *rowIndices*, and *columnIndices* of $\mathbf{A}_\mathrm{f}$ and $\mathbf{A}_\mathrm{d}$, are sorted by their *rowIndices*. Note that this sorting must be stable, which means that the original order is maintained when the compared values are equal to each other. This is because the ascending order of *columnIndices* might be disturbed if the sorting is not stable. After sorting, $\mathbf{A}_\mathrm{f}$ and $\mathbf{A}_\mathrm{d}$ are separated along their rows into $\mathbf{A}_\mathrm{ff}$, $\mathbf{A}_\mathrm{df}$, $\mathbf{A}_\mathrm{fd}$, and $\mathbf{A}_\mathrm{dd}$. Subtraction of *rowIndices* of $\mathbf{A}_\mathrm{df}$ and $\mathbf{A}_\mathrm{dd}$ is performed for the same reason as that for $\mathbf{A}_\mathrm{d}$. Finally, the separated matrices are obtained in the COO format.

These procedures require sorting of large arrays, which is computationally expensive. In order to accelerate the sorting process, they are implemented on a GPU using the NVIDIA CUDA thrust library.

For further optimization, the permutation list and the arrays (*values*, *rowIndices*, *columnIndices*) of $\mathbf{A}$ can be compressed by considering the series order of the arrays of $\mathbf{A}$. An example of the compression is shown in Fig. 9. Because three variables of each node are relocated together, the permutation list becomes a combination of three consecutive integers. Moreover, *rowIndices* and *columnIndices* consist of a combination of the same three integers, i.e., (0, 0, 0), and a combination of three consecutive integers, i.e., (0, 1, 2), respectively. Hence, sorting is performed according to these three-integer blocks. In order to sort per block, *values* is separated into three arrays, *values_x*, *values_y*, and *values_z*. Next, the permutation list, *rowIndices*, and *columnIndices* are compressed by storing only the first element of the consecutive-integer blocks. This compression reduces the size of the arrays to a third of their original size and the computational cost of sorting decreases. After $\mathbf{A}_\mathrm{ff}$, $\mathbf{A}_\mathrm{fd}$, $\mathbf{A}_\mathrm{df}$, and $\mathbf{A}_\mathrm{dd}$ are obtained, the compressed arrays are extracted in the original COO format.

Modeling of dissection

Topological-singularity avoidance algorithm for element removal

This section describes a simple and efficient topological-singularity avoidance algorithm for element removal. The basic concept of the approach is active element removal. Although the volume decreases as elements are removed, the approach is fast and easy to implement. The flow of the algorithm is summarized as follows.

1. *Fracture detection* Determine the tetrahedrons to be removed on the basis of a specified fracture criterion and list them in a set T_rm.
2. *Singularity verification* Check whether the vertices and edges that belong to T_rm are singular after removing the tetrahedrons listed in T_rm.
3. *Detection of additional tetrahedrons to be removed* If any vertices or edges are predicted to be singular, the

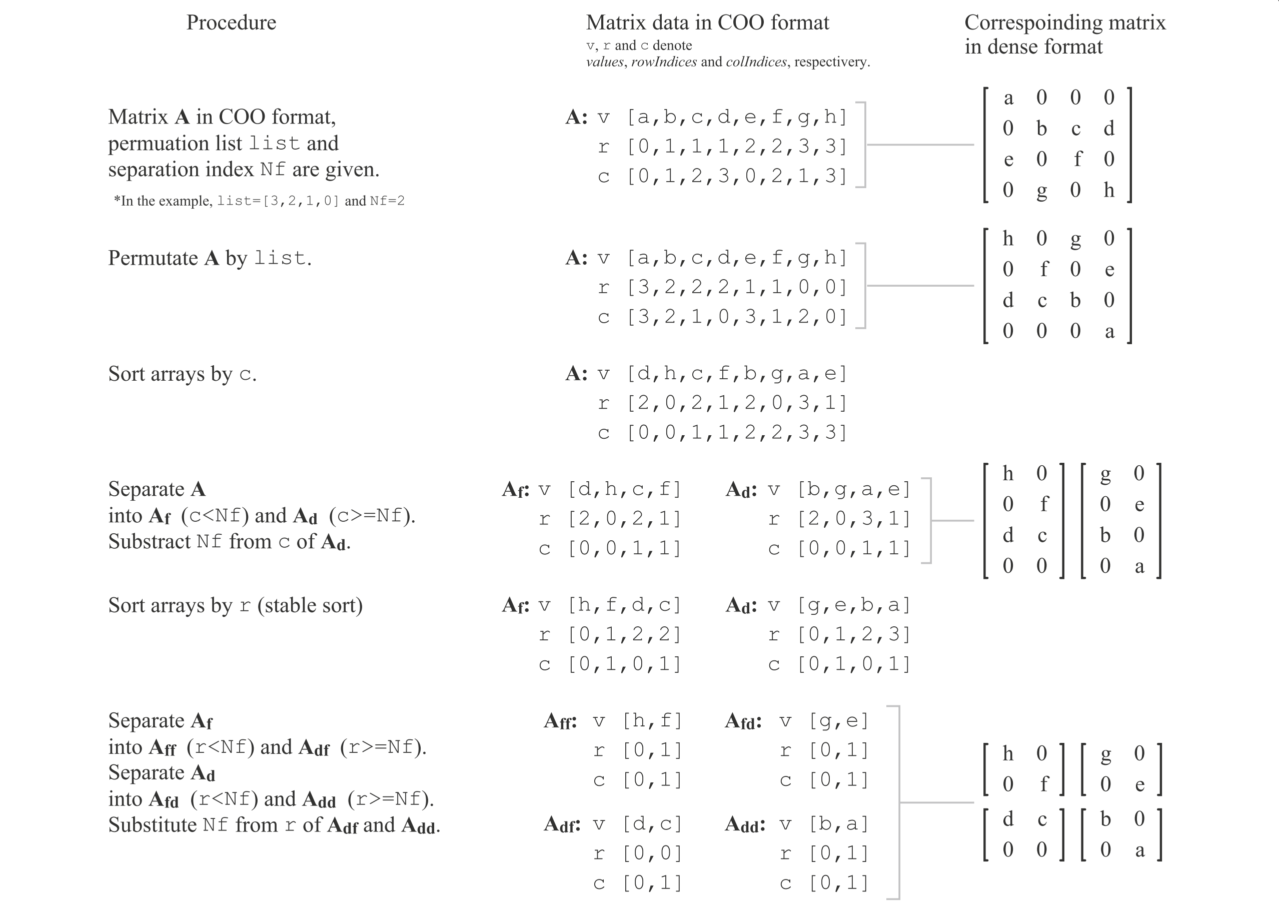

Procedure
Matrix data in COO format
v, r and c denote values, rowIndices and colIndices, respectivery.
Correspoinding matrix in dense format
Matrix A in COO format, permuation list list and separation index Nf are given.
*In the example, list=[3,2,1,0] and Nf=2
A: v [a,b,c,d,e,f,g,h]
r [0,1,1,1,2,2,3,3]
c [0,1,2,3,0,2,1,3]
a 0 0 0 / 0 b c d / e 0 f 0 / 0 g 0 h
Permutate A by list.
A: v [a,b,c,d,e,f,g,h]
r [3,2,2,2,1,1,0,0]
c [3,2,1,0,3,1,2,0]
h 0 g 0 / 0 f 0 e / d c b 0 / 0 0 0 a
Sort arrays by c.
A: v [d,h,c,f,b,g,a,e]
r [2,0,2,1,2,0,3,1]
c [0,0,1,1,2,2,3,3]
Separate A into Af (c<Nf) and Ad (c>=Nf). Substract Nf from c of Ad.
Af: v [d,h,c,f] r [2,0,2,1] c [0,0,1,1]
Ad: v [b,g,a,e] r [2,0,3,1] c [0,0,1,1]
h 0 / 0 f / d c / 0 0
g 0 / 0 e / b 0 / 0 a
Sort arrays by r (stable sort)
Af: v [h,f,d,c] r [0,1,2,2] c [0,1,0,1]
Ad: v [g,e,b,a] r [0,1,2,3] c [0,1,0,1]
Separate Af into Aff (r<Nf) and Adf (r>=Nf). Separate Ad into Afd (r<Nf) and Add (r>=Nf). Substitute Nf from r of Adf and Add.
Aff: v [h,f] r [0,1] c [0,1]
Afd: v [g,e] r [0,1] c [0,1]
Adf: v [d,c] r [0,0] c [0,1]
Add: v [b,a] r [0,1] c [0,1]
h 0 / 0 f
g 0 / 0 e
d c / 0 0
b 0 / 0 a

Fig. 8 Algorithm and example of matrix rearrangement

tetrahedrons that include the predicted singular vertices or edges are added to T_{rm}.

4. Repeat *Singularity verification* and *Detection of additional tetrahedrons to be removed* until T_{rm} becomes empty.

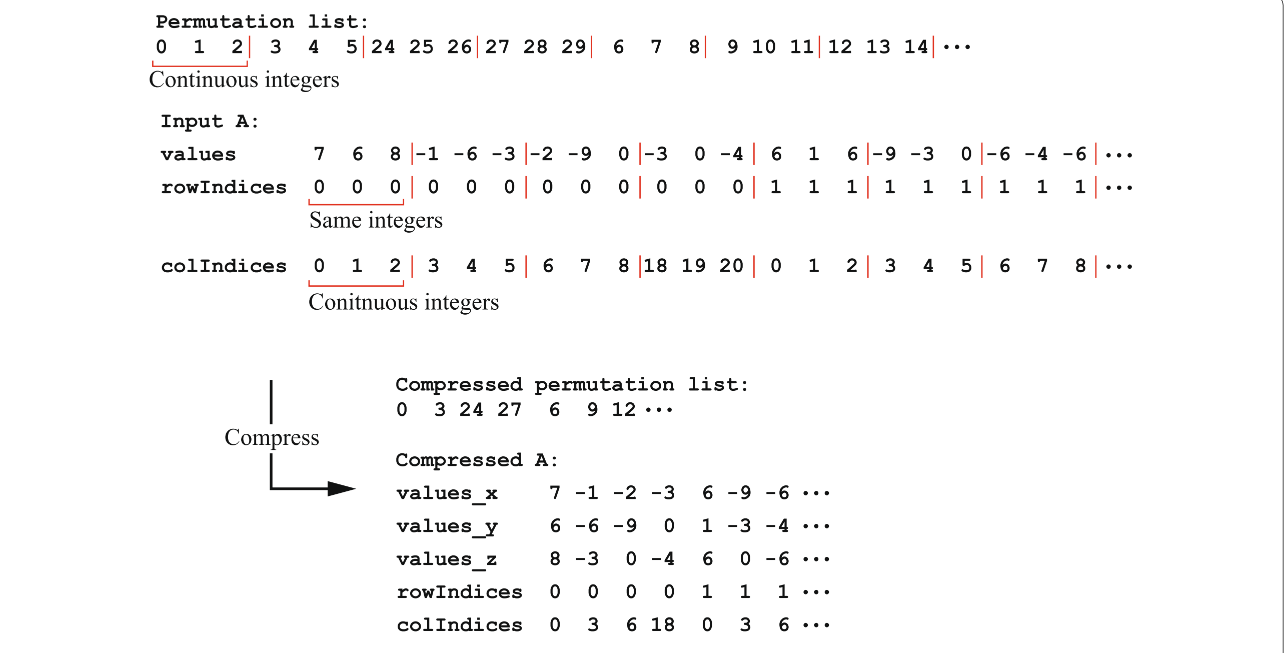
The maximum principal stress is selected as the criterion to determine the tetrahedrons to be removed. If the absolute value of the maximum principal stress of an element exceeds a previously specified threshold, the element is listed in T_{rm}, the set of tetrahedrons to be removed. Hence, the removal criterion is defined as

$$\max(|\sigma_1|, |\sigma_2|, |\sigma_3|) > \sigma_{\max}, \tag{17}$$

where σ_i $(i = 1, 2, 3)$ is the maximum principal stress of a tetrahedron, which is obtained as the eigenvalue of the stress tensor $\mathbf{S}^e$, and $\sigma_{\max}$ is the previously specified stress threshold. Note that we use a constant strain element; and thus, $\mathbf{S}^e$ becomes constant on an element. In the corotational FEM, $\mathbf{S}^e$ is calculated by considering the element rotation as $\mathbf{S}^e = \mathbf{D}^e\mathbf{B}^e\left(\mathbf{R}^e\boldsymbol{x}^e - \boldsymbol{x}_0^e\right)$, where $\mathbf{D}^e$ and $\mathbf{B}^e$ are the strain–stress matrix and displacement–strain matrix, respectively.

In the singularity detection phase, the sets of vertices V_{rm} and edges E_{rm} are constructed from the tetrahedrons listed in T_{rm}. Each vertex $v \in V_{\mathrm{rm}}$ and edge $e \in E_{\mathrm{rm}}$ is checked for a singularity. The algorithm of singularity detection of vertex v is summarized as follows. An example is shown in Fig. 10a.

1. Extract T^v, a set of tetrahedrons, that includes v as a vertex.
2. Select an arbitrary tetrahedron $t_0^v \in T^v$.
3. Construct T_{edge}^v, a set of tetrahedrons, that shares at least one edge with t_0^v.
4. Select a tetrahedron $t_x^v \in T_{\mathrm{edge}}^v$ and search for an edge-sharing tetrahedron as described above in steps 2 and 3. Add the new edge-sharing tetrahedron to T_{edge}^v and repeat until no entry is found.
5. If $n(T^v) \neq n(T_{\mathrm{edge}}^v)$, v is a singular vertex, where $n(\cdot)$ denotes the number of tetrahedrons.


```
Permutation list:
0  1  2| 3  4  5|24 25 26|27 28 29| 6  7  8| 9 10 11|12 13 14|···

Input A:
values       7  6  8|-1 -6 -3|-2 -9  0|-3  0 -4| 6  1  6|-9 -3  0|-6 -4 -6|···
rowIndices   0  0  0| 0  0  0| 0  0  0| 0  0  0| 1  1  1| 1  1  1| 1  1  1|···

colIndices   0  1  2| 3  4  5| 6  7  8|18 19 20| 0  1  2| 3  4  5| 6  7  8|···

                  Compressed permutation list:
                  0  3 24 27  6  9 12 ···

                  Compressed A:
                  values_x     7 -1 -2 -3  6 -9 -6 ···
                  values_y     6 -6 -9  0  1 -3 -4 ···
                  values_z     8 -3  0 -4  6  0 -6 ···
                  rowIndices   0  0  0  0  1  1  1 ···
                  colIndices   0  3  6 18  0  3  6 ···
```

Fig. 9 Example of actual input arrays and compressed arrays

The algorithm for singular edge detection is similar to that for singular vertex detection, as it is summarized as follows. An example is shown in Fig. 10b.

1. Extract T^e, a set of tetrahedrons, that include e as an edge.
2. Select an arbitrary tetrahedron $t_0^e \in T^e$.
3. Construct T_{edge}^e, a set of tetrahedrons, that shares at least one edge with t_0^e except edge e.
4. Select a tetrahedron $t_x^e \in T_{\mathrm{edge}}^e$ and search for an edge-sharing tetrahedron as described above in steps 2 and 3. Add the new edge-sharing tetrahedron to T_{edge}^e and repeat until no entry is found.
5. If $n(T^e) \neq n(T_{\mathrm{edge}}^e)$, e is a singular edge.

The detection of additional tetrahedrons to be removed phase determines a set of additional tetrahedrons to be removed, T_{add}, in order to avoid a topological singularity. When a singular vertex v is detected, T_{edge}^v and $\hat{T}_{\mathrm{edge}}^v \left(= T^v \cap \bar{T}_{\mathrm{edge}}^v\right)$ are defined. In order to prevent the loss of volume as much as possible, the number of tetrahedrons to be removed should be minimized. Therefore, the smaller set between T_{edge}^v and $\hat{T}_{\mathrm{edge}}^v$ is selected as T_{add} by comparing $n(T_{\mathrm{edge}}^v)$ and $n(\hat{T}_{\mathrm{edge}}^v)$. For the same reason, when a singular edge e is detected, the smaller set between T_{edge}^e and $\hat{T}_{\mathrm{edge}}^e = T^e \cap \bar{T}_{\mathrm{edge}}^e$ is selected as T_{add}. After T_{add} is determined, it is added to T_{rm} as mentioned at the beginning of this section.

Examples of fracture simulations are shown in Fig. 11. In the simulation without topological-singularity avoidance (Fig. 11b), tetrahedrons connected with only a singular vertex or edge exhibit unstable deformation. On the other hand, in the simulation with topological-singularity avoidance (Fig. 11a), the risk of instability is eliminated, and the simulation continues in any fracture situation.

Implementation

The calculation of the maximum principal stress on each element is computed in parallel by the GPU. To calculate the eigenvalues of the stress tensor, the Jacobi eigenvalue algorithm is adopted. The singularity avoidance algorithm is implemented on a six-core CPU because it requires numerous conditional branchings and complicated data structures for the mesh topology. However, it is not a time-consuming procedure and is rapidly computed, even on a CPU.

Results

Performance evaluation of GPU implementations

We compare three implementations of matrix/vector assembly and matrix rearrangement procedures: (1) a CPU with no parallelization, (2) a six-core CPU with multithread parallelization, and (3) a GPU implementation. Cube-shaped models discretized by various numbers of tetrahedrons were used for the comparison. The surface nodes of the two opposite sides of the cube are

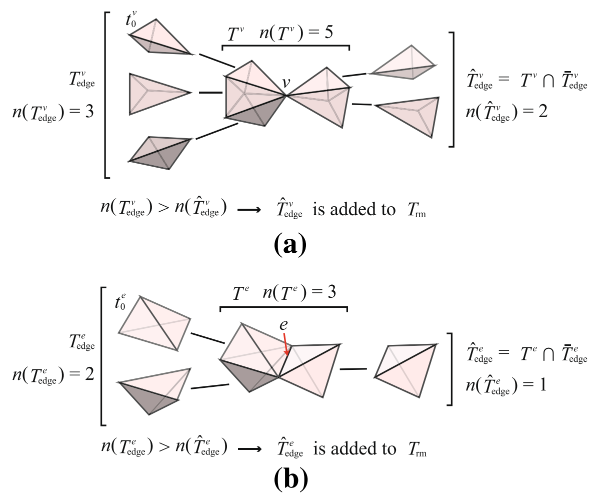

t_0^v
$T^v \quad n(T^v) = 5$
T_{edge}^v
$n(T_{\text{edge}}^v) = 3$
v
$\hat{T}_{\text{edge}}^v = T^v \cap \bar{T}_{\text{edge}}^v$
$n(\hat{T}_{\text{edge}}^v) = 2$
$n(T_{\text{edge}}^v) > n(\hat{T}_{\text{edge}}^v) \longrightarrow \hat{T}_{\text{edge}}^v$ is added to T_{rm}
(a)
t_0^e
$T^e \quad n(T^e) = 3$
T_{edge}^e
$n(T_{\text{edge}}^e) = 2$
e
$\hat{T}_{\text{edge}}^e = T^e \cap \bar{T}_{\text{edge}}^e$
$n(\hat{T}_{\text{edge}}^e) = 1$
$n(T_{\text{edge}}^e) > n(\hat{T}_{\text{edge}}^e) \longrightarrow \hat{T}_{\text{edge}}^e$ is added to T_{rm}
(b)

Fig. 10 Example of topological singularity detection. **a** Singular vertex detection. **b** Singular edge detection

constrained. The execution times of the three different implementations of the matrix/vector assembly and matrix rearrangement procedures are plotted in Figs. 12 and 13, respectively.

Blunt dissection simulation

Blunt dissection is an operation for separating tissues without cutting. It is generally performed along fissures by breaking connective tissues. In neurosurgery, surgeons perform cutting operations using scissors or blunt dissection depending on the context of the surgery.

An FE model of a cube with a fissure (4807 nodes and 19,600 tetrahedrons) was used in the simulation. It was assumed that the fissure was filled with connective tissues, the Young's modulus and Poisson's ratio of which were 100 and 0.4 Pa, respectively. The Young's modulus and Poisson's ratio of the main body were assumed to be 1000 and 0.4 Pa, respectively. The fracture threshold stresses were set to the same values as their Young's moduli. Note that these mechanical parameters are determined to distinguish the relative stiffness of the materials and not validated by experiments. Initially, the tips of two spatulas were inserted into the fissure, and then they were opened to dissect the connective tissues at a velocity of 5.0 mm/s. In order to compare different implementations, this simulation was executed by each implementation with a constant time step of 20 ms.

The simulation was conducted without oscillation or divergence. Figure 14a, b show the snapshots and its principal stress visualizations during the simulation. Figures 15 and 16 show the calculation time and the number of removed elements at each time step. An additional movie file shows the following simulations in greater detail (see Additional file 1).

Brain retraction simulation

Brain retraction is an operation performed by pushing soft tissues to create a working space. One of the important brain fissures, which are frequently performed retractions, is the the Sylvian fissure. The Sylvian fissure is filled with the arachnoid mater, which needs to be dissected by surgeons. This section shows the result of a brain retraction simulation conducted in real-time by user input using a Sensable Phantom Omni haptic device. The reaction force to the user-controlled instruments was fed back through the haptic device. The simulation was conducted under the assumption that the arachnoid mater was dissected beforehand. The task objective given to the user is to retract the brain tissues and expose the brain tumor existing at the bottom of the Sylvian fissure.

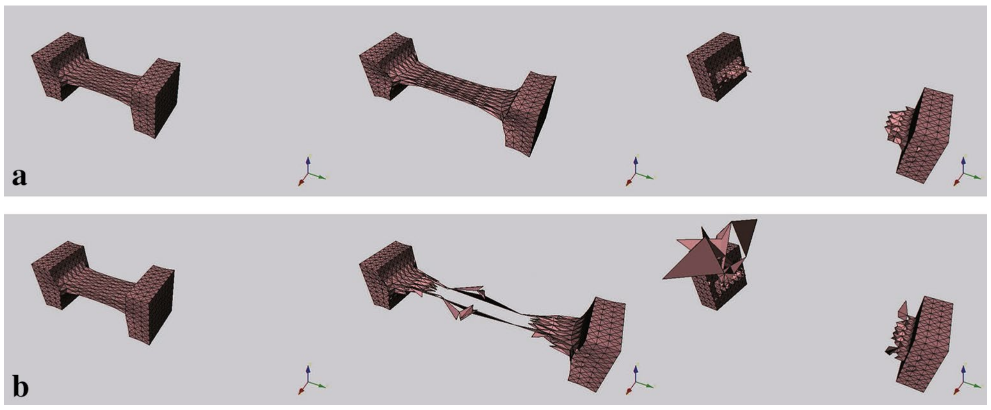


Fig. 11 Examples of fracture simulations [31]. These sequences show soft-tissue fracture simulations executed **a** with and **b** without topological-singularity avoidance

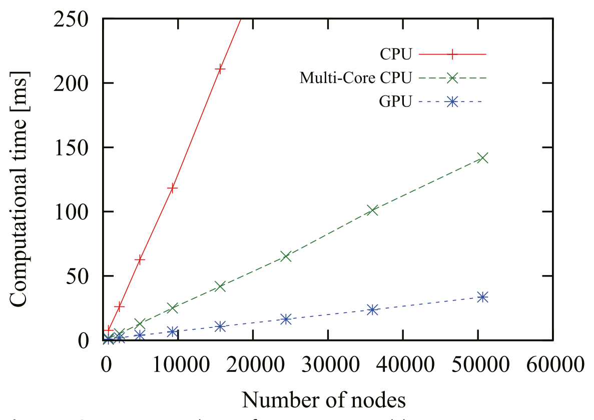


Fig. 12 Computational time for matrix assembly

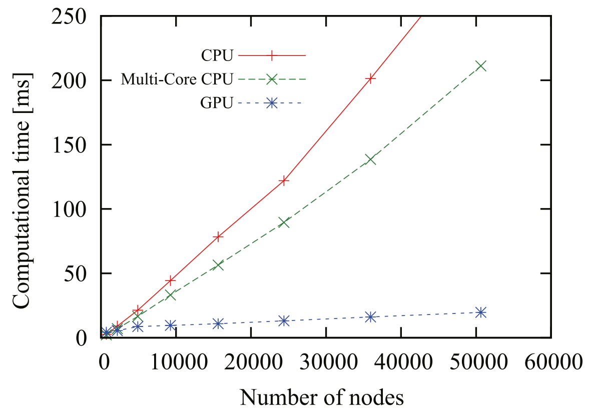


Fig. 13 Computational time for matrix rearrangement

A brain hemisphere mesh model (8647 nodes, 32,639 elements) was used in this simulation. This model was constructed by scanning an anatomical model of the human brain, Brain Model C20 (3B Scientific GmbH), and modifying it using 3D modeling software. The bottom nodes of the hemisphere mesh model were fixed; hence, the displacements of the bottom nodes were always set to zero.

Figure 17 shows the overview of the simulation. An operator moves a pointer displayed on the monitor by controlling a haptic device. Using the pointer, the operator can pick up and control the spatulas in the virtual space. Collision detection is performed between each spatula and the brain model. The reaction force applied to the spatula from the virtual brain is fed back by the haptic device. As seen in Fig. 17, the Sylvian fissure was opened by two spatulas and the tumor was exposed. Figure 18 shows the calculation time of the time steps. The calculation times for assembling a matrix, rearranging a matrix, and solving a linear system of equations are plotted. In this simulation, a stress analysis and the fracture procedure were not performed. Figure 19 shows the time history of the raw and filtered reference forces by a first-order low-pass filter (cut-off frequency 1.0 Hz). An additional movie file shows the following simulations in greater detail (see Additional file 1).

Discussions

Figures 12 and 13 show that the GPU implementations had the highest speed among the three implementations in both evaluations. In the comparison of the matrix/vector assembly, for the model with 15,625 nodes and 69,120 elements, the GPU implementation (10.7 ms) was 19.7 times faster than the single-CPU implementation (210.9 ms) and 3.9 times faster than the six-core CPU implementation (41.9 ms). In the comparison of the matrix rearrangement, for the same model, the GPU implementation (11.0 ms) was 7.1 times faster than the single-CPU implementation (78.4 ms) and 5.1 times faster than the six-core CPU implementation (56.3 ms).

The results of the blunt dissection simulation show that the combination of our GPU implementation and the fracture algorithm worked as expected. As seen in Fig. 14b, the connective tissue was easily deformed and removed owing to the stress concentration because it was specified to be softer than the main body. In Fig. 15, the calculation time jitter is shown. One of the causes is the difference in the convergence times of the conjugate gradient method. Another cause is the change in the boundary condition. When the boundary condition changed, the matrix rearrangement procedure is executed and takes additional calculation time. The average calculation times of the three implementations, a CPU with no parallelization, a six-core CPU with multithreaded parallelization, and a GPU implementation, were 103, 41, and 17 ms, respectively. The speed-up of the GPU versus the CPU was 6.1. Only the GPU realized smooth animation with a refresh rate greater than 30 Hz. As seen in Fig. 16, the fracture started at step 25, and the peak number of removed elements was 33 at step 60. It is shown that the number of removed elements did not affect the calculation time. This result shows that this approach is preferable for surgery simulation because the simulation can be continued at the same refresh rate throughout.

On the other hand, the results of the brain retraction simulation show that our implementation could not achieve the target calculation speed. The range of calculation time for a time step was 40–80 ms. This refresh rate is not sufficient for visually acceptable animations and reaction-force rendering. The reference force was discontinuous, and the force display could oscillate if we did not apply the low-pass filter. Although the low-pass filter reduced the discontinuous force feedback, this is not a fundamental solution for displaying realistic reaction forces. From these results, further acceleration is

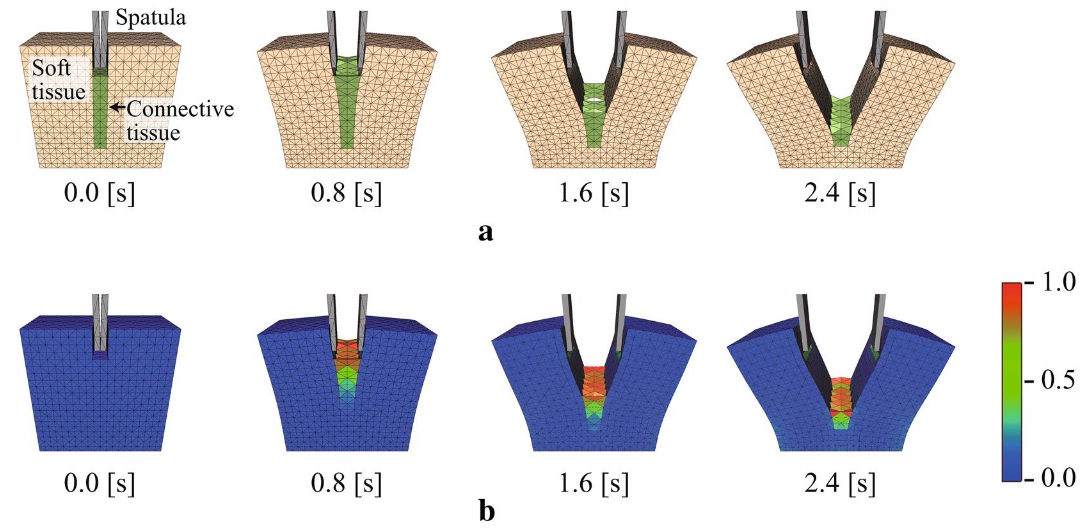


Fig. 14 Results of the blunt dissection simulation. **a** Snapshots. **b** Stress visualization. The *colors* of the tetrahedrons represent the regularized absolute values of the maximum principal stress $\max(|\sigma_1|, |\sigma_2|, |\sigma_3|)/\sigma_{\max}$, where $\sigma_i\,(i = 1, 2, 3)$ and $\sigma_{\max}$ are the principal stresses and the fracture threshold stress, respectively

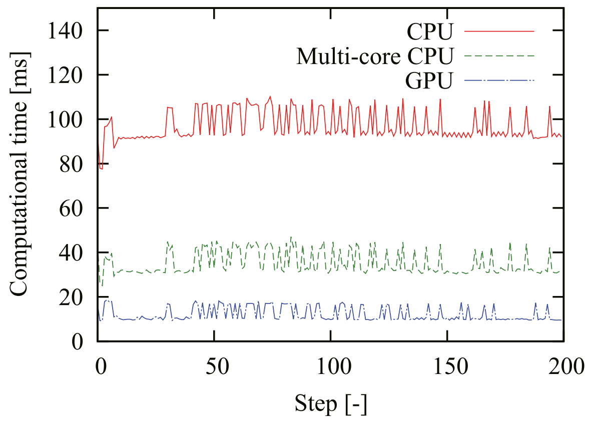


Fig. 15 Computational time of the blunt dissection simulation

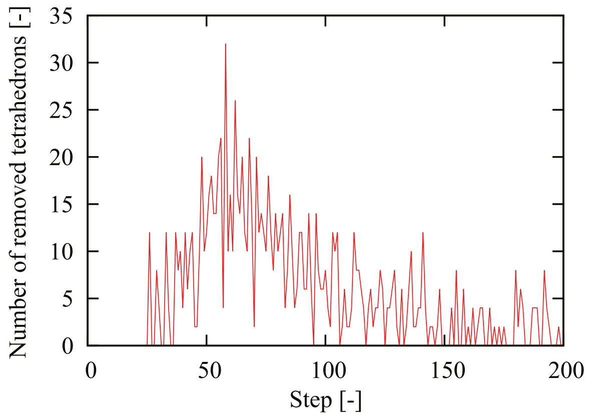


Fig. 16 Number of removed elements

needed to achieve stable and visually acceptable simulation. Moreover, the development of a method for displaying smooth and stable forces is a topic for future study.

Conclusion

In this paper, a real-time simulation scheme for soft-tissue deformation and fracture for brain retraction is proposed. GPU implementations for matrix/vector assembly and a matrix rearrangement procedure for accelerating a corotational FEM including boundary-condition-based collision response are proposed. A simple mesh modification method considering the avoidance of topological singularities is developed and combined with the proposed GPU-accelerated FEM framework. Finally, blunt dissection and brain retraction simulations are performed using the proposed implementation. Both simulations can be conducted in real time. Although the proposed method could not achieve a visually acceptable update rate for the brain retraction simulation using our target brain hemisphere model, it performs faster than the CPU implementations.

In this study, viscoelasticity and material nonlinearities were not considered. In order to obtain more realistic

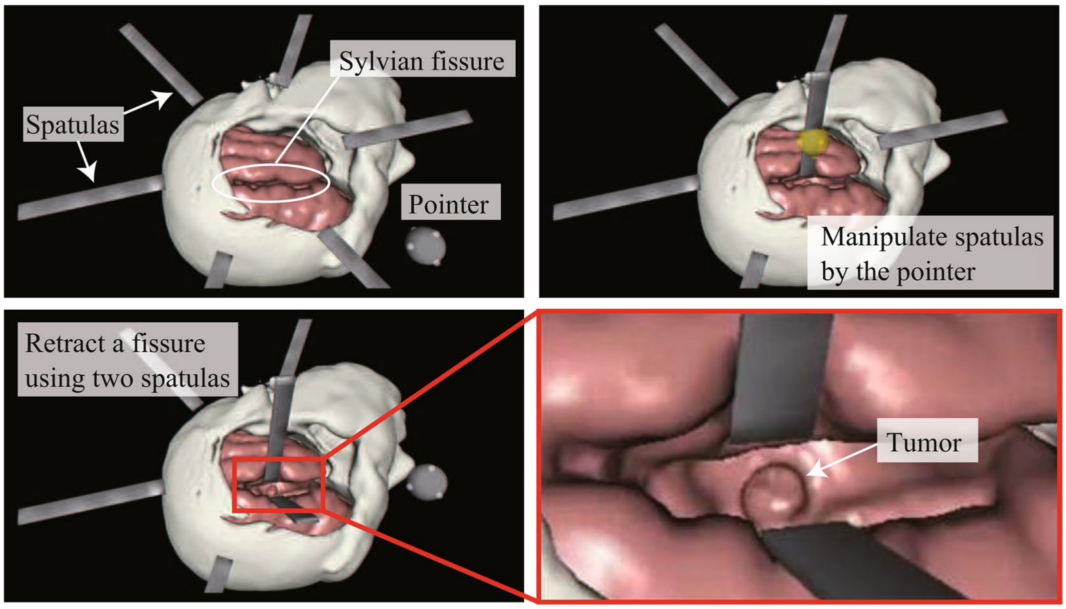

Sylvian fissure
Spatulas
Pointer
Manipulate spatulas by the pointer
Retract a fissure using two spatulas
Tumor

Fig. 17 Brain retraction simulation

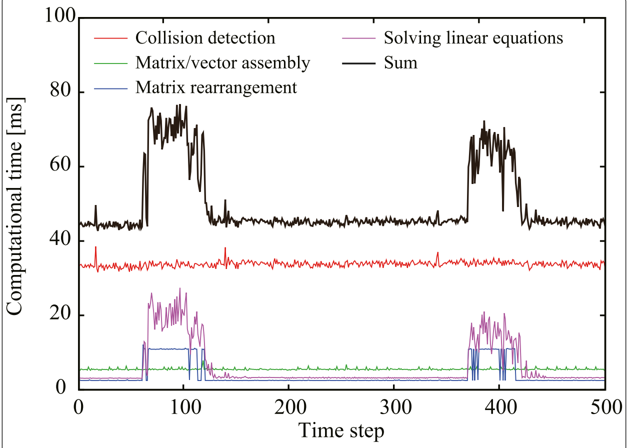


Fig. 18 Computational time of the brain retraction simulation

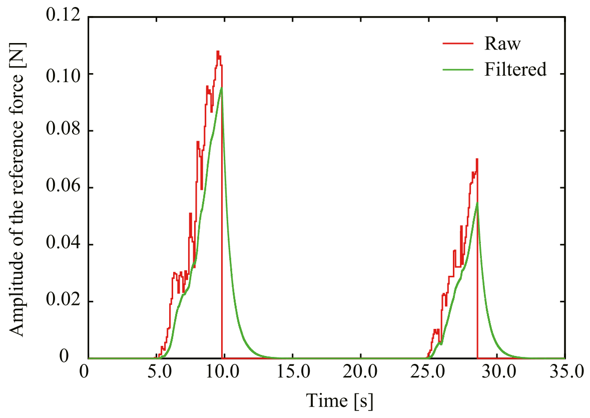


Fig. 19 Time history of reaction force. The raw data is the reference force value provided by the FEM solver and the filtered data is the actual reference value for the force feedback. In this simulation, a first-order low-pass filter (cut-off frequency, 1.0 Hz) was applied

material behavior, we plan to integrate material properties more precisely in our future implementation.

Additional file

Additional file 1. Blunt dissection simulation and brain retraction simulation using the proposed method.

Authors' contributions

AK led and directed the project. AK and TT showed the need for a fast and stable calculation method of deformation and fracture of soft tissues for neurosurgery. KS proposed the algorithms implemented them on a GPU, and drafted the manuscript. TT, AF, and AK participated in the discussion on the optimization of the algorithms. Furthermore, TT implemented the base simulation framework, and AF implemented the collision detection procedure. All authors read and approved the final manuscript.

Author details

[1] Graduate School of Information Science and Technology, Hokkaido University, Kita 14 Nishi 9, Kita-ku, Sapporo, Japan. [2] Graduate School of Engineering, Tohoku University, 2-1-1 Katahira, Aoba-ku, Sendai, Japan. [3] Graduate School of Science and Engineering, National Defense Academy, 1-10-20 Hashirimizu, Yokosuka, Japan.

Acknowledgements

This work was supported by JSPS through the Funding Program for Next Generation World-Leading Researchers (LR003), the Grant-in-Aid for Scientific Research (A) (15H01707), the Grant-in-Aid for Challenging Exploratory Research (24650288), and the Grant-in-Aid for JSPS Fellows (15J01452).

Competing interests
The authors declare that they have no competing interests.

Appendix 1: Rotation of a deformed tetrahedron

A rotation matrix of a tetrahedron element is obtained by SVD of the deformation gradient tensor $\mathbf{F}$ [27]. This formulation is stable even if the elements are inverted (see Fig. 3b). In the case of the first-order tetrahedral element, $\mathbf{F}$ transforms an edge vector of the initial shape $\boldsymbol{d}_{\mathrm{m}j}$ into an edge vector of the deformed shape $\boldsymbol{d}_{\mathrm{s}j}$ as $\boldsymbol{d}_{\mathrm{s}j} = \mathbf{F}\boldsymbol{d}_{\mathrm{m}j}$ $(j = 1, 2, 3)$. From this equation, $\mathbf{F}$ is calculated as $\mathbf{F} = \mathbf{D}_{\mathrm{s}}\mathbf{D}_{\mathrm{m}}^{-1}$, where $\mathbf{D}_{\mathrm{s}} = [\boldsymbol{d}_{\mathrm{s}1}\ \boldsymbol{d}_{\mathrm{s}2}\ \boldsymbol{d}_{\mathrm{s}3}]$ and $\mathbf{D}_{\mathrm{m}} = [\boldsymbol{d}_{\mathrm{m}1}\ \boldsymbol{d}_{\mathrm{m}2}\ \boldsymbol{d}_{\mathrm{m}3}]$. $\mathbf{F}$ can be represented as

$$\mathbf{F} = \mathbf{U}\Sigma\mathbf{V}^{\mathrm{T}}, \tag{18}$$

where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices, and Σ is a diagonal matrix. The rotation matrix is calculated as

$$\mathbf{R} = \mathbf{U}\mathbf{C}\mathbf{V}^{\mathrm{T}}, \tag{19}$$

where $\mathbf{C} \triangleq \mathrm{diag}\left[1, 1, \det(\mathbf{U}\mathbf{V}^{\mathrm{T}})\right]$ [28].

To obtain the rotation matrices of the tetrahedral elements, SVD of a large number of 3×3 matrices described in Eq. (18) must be performed. However, most existing parallel implementations of SVD are specialized for large matrices [29]. For SVD of a large number of small matrices, Bedkowski et al. introduced an algorithm for three-dimensional reconstruction using mobile robots [30]. In the present work, the algorithm introduced by Bedkowski et al. is modified. The modified algorithm is summarized as follows:

1. Diagonalize $\mathbf{F}^{\mathrm{T}}\mathbf{F}$ by the Jacobi eigenvalue algorithm as $\mathbf{F}^{\mathrm{T}}\mathbf{F} = \mathbf{V}^{\mathrm{T}}\mathbf{S}\mathbf{V}$, where $\mathbf{V}$ is an orthogonal matrix, and $\mathbf{S}$ is a diagonal matrix whose elements are the eigenvalues of $\mathbf{F}^{\mathrm{T}}\mathbf{F}$.
2. Construct a matrix Σ whose diagonal elements are the singular values of $\mathbf{F}^{\mathrm{T}}\mathbf{F}$. The singular values are obtained by calculating the square root of each diagonal element of $\mathbf{S}$.
3. Calculate $\mathbf{U} = \mathbf{F}\mathbf{V}\Sigma^{-1}$.
4. $\mathbf{U}$ and $\mathbf{V}$ are used in Eq. (19).

In this algorithm, the eigenvalue approach is different from that of Bedkowski et al. They calculated the eigenvalues by obtaining the roots of a cubic polynomial. On the other hand, we adopted the Jacobi eigenvalue algorithm to simplify the implementation.

Appendix 2: Formulation of a dynamic FEM

The equation of motion for a deformable object is written as

$$\mathbf{M}\ddot{\boldsymbol{x}} + \mathbf{C}\dot{\boldsymbol{x}} + (\mathbf{K}\boldsymbol{x} + \boldsymbol{f}_0) = \boldsymbol{f}_{\mathrm{ext}}, \tag{20}$$

where $\mathbf{M}$ and $\mathbf{C}$ are a mass matrix and a damping matrix, respectively. $\mathbf{M}$ is a diagonal matrix determined by gathering the equivalent masses of all nodes from the node-share tetrahedrons: $m_i = \sum_{T_i} m_{T_i}/4$, where m_i is the equivalent mass of node i, T_i is a tetrahedron that shares node i, and m_{T_i} is the mass of T_i. In general, $\mathbf{C}$ is determined on the basis of the material constitutive law. However, for simplicity, Rayleigh damping is adopted in this study:

$$\mathbf{C} = \alpha\mathbf{M} + \beta\mathbf{K}, \tag{21}$$

where α and β are scalar values representing the damping effect, which are selected heuristically for stabilizing the simulation.

Eq. (20) can be written in the same form as the linear FEM form as

$$\mathbf{M}\ddot{\boldsymbol{x}} + \mathbf{C}\dot{\boldsymbol{x}} + \mathbf{K}\boldsymbol{x} = \boldsymbol{f} \tag{22}$$

by defining a vector $\boldsymbol{f} = \boldsymbol{f}_{\mathrm{ext}} - \boldsymbol{f}_0$. When we substitute $\boldsymbol{v}$ for $\dot{\boldsymbol{x}}$, the time derivatives of the variables are defined as

$$\dot{\boldsymbol{x}} = \boldsymbol{v}, \tag{23}$$

$$\mathbf{M}\dot{\boldsymbol{v}} = -\mathbf{C}\boldsymbol{v} - \mathbf{K}\boldsymbol{x} + \boldsymbol{f}. \tag{24}$$

In order to avoid numerical instability in the dynamic simulation, we adopt implicit time integration because it has unconditionally stable characteristics. Implicit time integration is formulated as

$$\boldsymbol{x}^{i+1} = \boldsymbol{x}^i + \Delta t\, \boldsymbol{v}^{i+1}, \tag{25}$$

$$\mathbf{M}\boldsymbol{v}^{i+1} = \mathbf{M}\boldsymbol{v}^i + \Delta t\left(-\mathbf{C}\boldsymbol{v}^{i+1} - \mathbf{K}\boldsymbol{x}^{i+1} + \boldsymbol{f}^{i+1}\right). \tag{26}$$

By substituting Eqs. (21) and (25) into Eq. (26), $\boldsymbol{v}^{i+1}$ can be obtained by solving the following equation:

$$\begin{aligned}&\left((1 + \alpha\Delta t)\mathbf{M} + \left(\beta\Delta t + \Delta t^2\right)\mathbf{K}\right)\boldsymbol{v}^{i+1}\\ &\quad = \mathbf{M}\boldsymbol{v}^i + \Delta t\left(-\mathbf{K}\boldsymbol{x}^i + \boldsymbol{f}^{i+1}\right).\end{aligned} \tag{27}$$

As discussed in "Collision response of soft tissues", the contact nodes move together with the rigid body; hence, $\boldsymbol{x}_{\mathrm{d}}$ and $\boldsymbol{v}_{\mathrm{d}}$ are known, whereas $\boldsymbol{f}_{\mathrm{d}}$ is unknown. The forces applied to the unconstrained nodes are zero, i.e., $\boldsymbol{f}_{\mathrm{f}} = \mathbf{0}$. Therefore, Eq. (27) can be rewritten as

$$\begin{aligned}&\left((1 + \alpha\Delta t)\bar{\mathbf{M}} + \left(\beta\Delta t + \Delta t^2\right)\bar{\mathbf{K}}\right)\bar{\boldsymbol{v}}^{i+1}\\ &\quad = \bar{\mathbf{M}}\bar{\boldsymbol{v}}^i + \Delta t\left(-\bar{\mathbf{K}}\bar{\boldsymbol{x}}^i + \bar{\boldsymbol{f}}^{i+1}\right),\end{aligned} \tag{28}$$

where

$$\bar{\mathbf{M}} = \begin{bmatrix}\mathbf{M}_{\mathrm{f}} & \mathbf{0}\\ \mathbf{0} & \mathbf{M}_{\mathrm{d}}\end{bmatrix},\ \bar{\mathbf{K}} = \begin{bmatrix}\mathbf{K}_{\mathrm{ff}} & \mathbf{K}_{\mathrm{fd}}\\ \mathbf{K}_{\mathrm{df}} & \mathbf{K}_{\mathrm{dd}}\end{bmatrix},$$

$$\bar{\boldsymbol{v}} = \left[\boldsymbol{v}_{\mathrm{f}}\boldsymbol{v}_{\mathrm{d}}\right],\ \bar{\boldsymbol{x}} = \begin{bmatrix}\boldsymbol{x}_{\mathrm{f}}\\ \boldsymbol{x}_{\mathrm{d}}\end{bmatrix},\ \bar{\boldsymbol{f}} = \begin{bmatrix}\boldsymbol{f}_{\mathrm{f}}\\ \boldsymbol{f}_{\mathrm{d}}\end{bmatrix}.$$

Eq. (28) is rewritten as

$$\begin{bmatrix} \mathbf{A}_{\mathrm{ff}} & \mathbf{A}_{\mathrm{fd}} \\ \mathbf{A}_{\mathrm{df}} & \mathbf{A}_{\mathrm{dd}} \end{bmatrix} \begin{bmatrix} \boldsymbol{v}_{\mathrm{f}}^{i+1} \\ \boldsymbol{v}_{\mathrm{d}}^{i+1} \end{bmatrix} = \begin{bmatrix} \boldsymbol{b}_{\mathrm{f}}^{i+1} \\ \boldsymbol{b}_{\mathrm{d}}^{i+1} \end{bmatrix}. \qquad (29)$$

Received: 23 May 2015 Accepted: 13 December 2015
Published online: 29 December 2015

References

1. Delorme S, Laroche D, DiRaddo R, Del Maestro RF (2012) Neurotouch: a physics-based virtual simulator for cranial microneurosurgery training. Neurosurgery 71:32–42
2. Banerjee PP, Luciano CJ, Lemole GM, Charbel FT, Oh MY (2007) Accuracy of ventriculostomy catheter placement using a head- and hand-tracked high-resolution virtual reality simulator with haptic feedback. J Neurosurg 107:515–521
3. Yasargil MG (1995) Microneurosurgery. Thieme
4. Zhong J, Dujovny M, Perlin AR, Perez-Arjona E, Park HK, Diaz FG (2003) Brain retraction injury. Neurol Res 25:831–838
5. Koyama T, Okudera H, Kobayashi S (1999) Computer-generated surgical simulation of morphological changes in microstructures: concepts of "virtual retractor". J Neurosurg 90(4):780–785
6. Hansen KV, Brix L, Pedersen CF, Haase JP, Larsen OV (2004) Modelling of interaction between a spatula and a human brain. Med Image Anal 8(1):23–33
7. Hasegawa Y, Adachi K, Azuma Y, Fujita A, Kohmura E, Kanki H (2010) A study on cerebellar retraction simulation for developing neurosurgical training system. J Jpn Soc Comput Aided Surg 12(4):533–543
8. Spicer MA, van Velsen M, Caffrey JP, Apuzzo MLJ (2004) Virtual reality neurosurgery: a simulator blueprint. Neurosurgery 54:783–798
9. Sahoo D, Deck C, Willinger R (2014) Development and validation of an advanced anisotropic visco-hyperelastic human brain FE model. J Mech Behav Biomed Mater 33(1):24–42
10. Konno A, Nakayama M, Chen XS, Fukuhara A, Sase K, Tsujita T, Abiko S (2013) Development of a brain surgery simulator. In: Proceedings of the International Symposium on Interdisciplinary Research and Education on Medical Device Developments, pp 29–32
11. Fukuhara A, Tsujita T, Sase K, Konno A, Jiang X, Abiko S, Uchiyama M (2014) Proposition and evaluation of a collision detection method for real time surgery simulation of opening a brain fissure. ROBOMECH J 1(1):6
12. Forest C, Delingette H, Ayache N (2005) Removing tetrahedra from manifold tetrahedralisation: application to real-time surgical simulation. Med Image Anal 9(2):113–122
13. Nakayama M, Abiko S, Jiang X, Konno A, Uchiyama M (2011) Stable soft-tissue fracture simulation for surgery simulator. J Robot Mechatron 23(4):589–597
14. Courtecuisse H, Jung H, Allard J, Duriez C, Lee DY, Cotin S (2010) Gpu-based real-time soft tissue deformation with cutting and haptic feedback. Prog Biophys Mol Biol 103:159–168
15. Courtecuisse H, Allard J, Kerfriden P, Bordas SPA, Cotin S, Duriez C (2014) Real-time simulation of contact and cutting of heterogeneous soft-tissues. Med Image Anal 18(2):394–410
16. Galoppo N, Tekin S, Otaduy MA, Gross M, Lin MC (2007) Interactive haptic rendering of high-resolution deformable objects. In: Proceedings of the 2nd International Conference on Virtual Reality, pp 215–233
17. Cotin S, Delingette H, Ayache N (1999) Real-time elastic deformations of soft tissues for surgery simulation. IEEE Trans Visual Comput Graph 5(1):62–73
18. Hirota K, Kaneko T (2001) Haptic representation of elastic objects. Presence Teleoper Virtual Environ 10(5):525–536
19. O'Brien JF, Bargteil AW, Hodgins JK (2002) Graphical modeling and animation of ductile fracture. ACM Trans Graph 21(3):291–294
20. Wojtan C, Thürey N, Gross M, Turk G (2009) Deforming meshes that split and merge. ACM Trans Graph 28(3):76:1–76:10
21. Hegemann J, Jiang C, Schroeder C, Teran JM (2013) A level set method for ductile fracture. In: Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation, pp 193–201
22. Jeřábková L, Kuhlen T (2009) Stable cutting of deformable objects in virtual environments using xfem. IEEE Comput Graph Appl 29(2):61–71
23. Mor AB, Kanade T (2000) Modifying soft tissue models: progressive cutting with minimal new element creation. In: Proceedings of the Third International Conference on Medical Image Computing and Computer-Assisted Intervention, pp 598–607
24. Delingette H, Cotin S, Ayache N (1999) A hybrid elastic model allowing real-time cutting, deformations and force-feedback for surgery training and simulation. Proc Comput Animat 1999:70–81
25. Müller M, Dorsey J, McMillan L, Jagnow R, Cutler B (2002) Stable real-time deformations. In: Proceedings of the 2002 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, pp 49–55
26. Cecka C, Lew A, Darve E (2012) Application of assembly of finite element methods on graphics processors for real-time elastodynamics. In: Hwu WmW (ed) GPU Computing Gems Jade Edition, Boston, pp 187–205
27. Irving G, Teran J, Fedkiw R (2004) Invertible finite elements for robust simulation of large deformation. In: Proceedings of the 2004 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, pp 131–140
28. Myronenko A, Song X (2009) On the closed-form solution of the rotation matrix arising in computer vision problems. Tech Rep arXiv:09041613v1 [csCV]
29. Lahabar S, Narayanan P (2009) Singular value decomposition on gpu using cuda. In: Proceedings of 23rd IEEE International Parallel and Distributed Processing Symposium, pp 1–10
30. Bedkowski J, Maslowski A (2011) GPGPU computation in mobile robot applications. Int J Electr Eng Inform 4(1):15–26
31. Sase K, Konno A, Tsujita T, Fukuhara A, Chen X, Komizunai S (2014) Stable fracture model of soft materials for a simulation of brain tumor resection. In: Proceedings of the 2014 JSME Conference on Robotics and Mechatronics, pp 3A1–B03