Interactive Cutting and Tearing in Projective Dynamics with Progressive Cholesky Updates

JING LI, University of Utah, USA and AICFVE, Beijing Film Academy, China
TIANTIAN LIU, Microsoft Research Asia, China and Taichi Graphics, China
LADISLAV KAVAN, University of Utah, USA
BAOQUAN CHEN, CFCS, Peking University, China

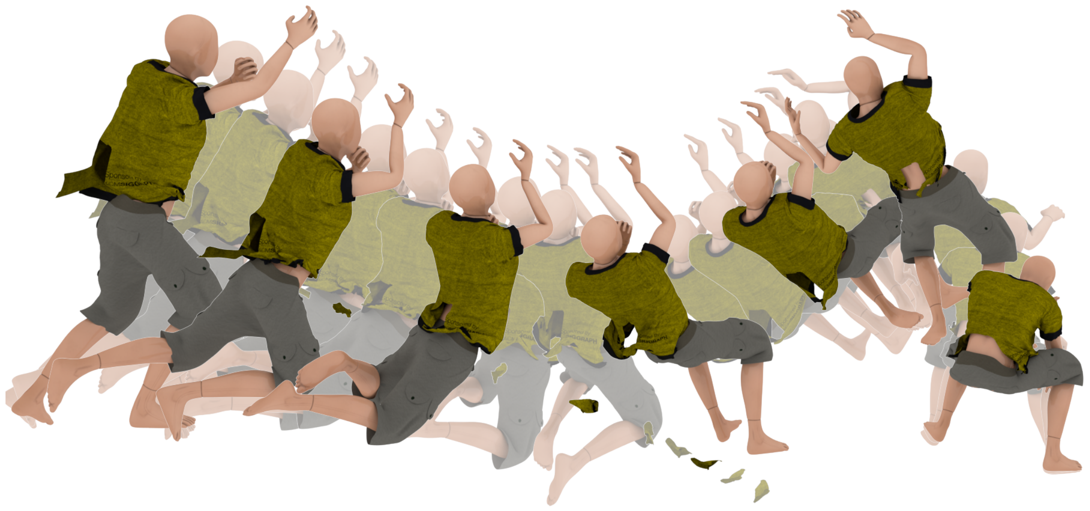

Fig. 1. A swinging person in a T-shirt is simulated using our system. One vertex of his T-shirt is being held and dragged, ripping apart the cloth. The entire simulation is performed using Projective Dynamics, with a constant frame-rate, even during the tearing process.

We propose a new algorithm for updating a Cholesky factorization which speeds up Projective Dynamics simulations with topological changes. Our approach addresses an important limitation of the original Projective Dynamics, i.e., that topological changes such as cutting, fracturing, or tearing require full refactorization which compromises computation speed, especially in real-time applications. Our method progressively modifies the Cholesky factor of the system matrix in the global step instead of computing it from scratch. Only a small amount of overhead is added since most of the topological changes in typical simulations are continuous and gradual. Our method is based on the update and downdate routine in CHOLMOD, but unlike recent related work, supports dynamic sizes of the system matrix and the addition of new vertices. Our approach allows us to introduce clean cuts and perform interactive remeshing. Our experiments show that our method works particularly well in simulation scenarios involving cutting, tearing, and local remeshing operations.

Authors' addresses: Jing Li, University of Utah, 50 Central Campus Dr, Salt Lake City, UT, 84112, USA, AICFVE, Beijing Film Academy, No.4, Xitucheng Rd, Haidian Qu, Beijing, China, 100088, jingli2070769@gmail.com; Tiantian Liu, Microsoft Research Asia, Beijing, China, Taichi Graphics, Beijing, China, ltt1598@gmail.com; Ladislav Kavan, University of Utah, 50 Central Campus Dr, Salt Lake City, Utah, 84112, USA, ladislav.kavan@gmail.com; Baoquan Chen, CFCS, Peking University, Jingyuan Courtyard 5, Peking University, 5 Yiheyuan Road, Haidian Qu, Beijing, China, baoquan@pku.edu.cn.

CCS Concepts: • **Computing methodologies** → **Computer graphics**; **Physical simulation**.

Additional Key Words and Phrases: physics-based animation, real-time simulation, Cholesky factorization, low-rank updates

ACM Reference Format:
Jing Li, Tiantian Liu, Ladislav Kavan, and Baoquan Chen. 2021. Interactive Cutting and Tearing in Projective Dynamics with Progressive Cholesky Updates. *ACM Trans. Graph.* 40, 6, Article 1 (December 2021), 12 pages. https://doi.org/10.1145/3478513.3480505

1 INTRODUCTION

Updating a Cholesky factorization after a small rank change in the matrix is a fundamental task which can be found in many computer graphics applications such as surface mesh parameterization and physics-based animation. In an interactive physics-based simulation, cutting and tearing of a deformable object are features commonly required by users. These operations would normally change the topology of the simulated object and require a corresponding update on the factors of the system matrix in Projective Dynamics. Especially in real-time simulation, it is important that the frame rate of the solver does not drop when the users start to interact with the scene. For example, when a surgeon starts to cut a tumor off a simulated organ, it would be highly undesirable if the simulation framerate dropped exactly when initiating the incision. A fast update of the Cholesky factorization ensures a smooth and realistic user experience even when topological changes of the simulated object are happening.

Projective Dynamics [Bouaziz et al. 2014] is a popular method to simulate deformable objects in real-time. The major computational advantage of Projective Dynamics comes from a local/global solver where all the nonlinearities of the system are grouped into the local step, leaving a constant linear system to solve in the global step. Since the system matrix in the global step only depends on state-independent quantities such as the topology of the mesh, the time-step size, and the material properties (e.g. mass density and elasticity, etc.), Projective Dynamics pre-computes and pre-factorizes its system matrix so it can reuse the Cholesky factorization during the simulation. However, when the topology of the mesh is changed, e.g., due to cutting or tearing, the computational benefits of Projective Dynamics may vanish. One possible alternative is represented by iterative methods such as multigrid [McAdams et al. 2011; Tamstorf et al. 2015; Xian et al. 2019]. However, these methods pose different types of tradeoffs, such as dependency on the topology of the simulated object in order to build their multi-level hierarchies.

Topological changes due to cutting and tearing are usually local. Therefore, re-factorizing the system matrix from scratch would be wasteful. We want to make use of the pre-factorized system matrix as much as possible while incorporating the topological changes during the simulation efficiently. Moreover, cutting and tearing operations normally operate in lower dimensions than the dimensionality of the simulated object, e.g., a 3D object (volumetric solid) is typically cut by a 2D plane, and a 2D object (thin shell) is usually cut by a 1D line. This means that we have far fewer vertices being affected by the topological changes, compared to the total number of vertices of the entire system. Especially in high-framerate simulations (needed e.g. in virtual reality), these topological changes typically happen gradually as the cutting progresses, leaving us with only a few degrees of freedom (DOFs) affected by the changes during *one frame*.

We propose a new method to reuse and update the factors of the system matrices of Projective Dynamics, building upon the body of work on updating sparse Cholesky factorizations [Davis and Hager 2001]. Classical Cholesky update methods provide us with a great point of departure. Specifically, they update the factor with additions of low-rank matrices, without changing the size of the matrix. However, in the process of cutting or tearing, new vertices will be inserted into the system to represent the cross-sections. Moreover, remeshing techniques might also be needed to improve the quality of the simulated elements, which can change the number of vertices of the system too. We observe that operations like adding and removing elements, or manipulating the mass of vertices in Projective Dynamics can be seen as low-rank updates without changing the size of the system matrix. Therefore, we propose to update the Cholesky factor in Projective Dynamics in four steps. First, we shift non-zeros in the factor to make room for new DOFs. Second, we run a low-rank update process to remove the elements which were affected by the topological changes. Third, we fill the diagonal of the Cholesky factor with the masses of the new vertices introduced by the topological changes, as if they are not connected to anything else (i.e., isolated mass points). Finally, we introduce new elements using another low-rank update pass, connecting the newly added vertices back to the simulated system. We show the effectiveness of our method in two types of simulations: 1) mass-spring systems and 2) finite elements. Our method introduces only minimal overhead upon the original Projective Dynamics, which ensures a constant frame-rate even during cutting and tearing operations.

Our main contributions are:

- We present an algorithm to efficiently update the Cholesky factorization which supports the addition of new vertices, topological changes, and remeshing operations.
- Our method is particularly effective when the updates are gradual and progressive, which is common in real-time physics-based simulation.
- We show that our factorization update method can be used to support various discretization models such as mass-spring systems and finite element methods simulated using Projective Dynamics, while avoiding fluctuations in computation time during the topological changes.

2 RELATED WORK

Gill et al. [1974] provide a technique for dense Cholesky factorization after rank-1 update. Based on the modification idea of [Gill et al. 1974] and the analysis of the underlying graph structure of the matrix to be factorized, Davis and Hager [1999] extend the rank-1 update to an arbitrary sparse symmetric positive definite matrix. Davis and Hager [2005] update the factorization after a symmetric modification of a row/column pair by using rank-1 update in [Davis and Hager 1999]. This method finds specific application in the dual approach of linear programming where the system is solved repeatedly with equality constraints being added or dropped before each solve. One prerequisite of using this method is to identify the total number of equality constraints beforehand and have them as empty rows/columns in the matrix. In applications such as surgical simulators, topological changes happen frequently and it is difficult or impossible to predict where they will occur, making the method of [Davis and Hager 2005] not suitable. Using the framework developed in [Davis and Hager 1999] on rank-1 modifications, Davis and Hager [2001] developed a multiple-rank update technique. The operation count of [Davis and Hager 1999] is optimal while the operation count [Davis and Hager 2001] is near-optimal. But [Davis and Hager 2001] enjoys better memory traffic and faster execution time since multiple-rank update makes one pass through the factors while a series of rank-1 updates requires multiple passes through the factors. Both [Davis and Hager 1999] and [Davis and Hager 2001] tested their methods on an optimization problem from airline scheduling, where the DOFs of the problem are fixed. However, in physics-based simulation of cutting and tearing, we need not only low-rank updates, but also adding new DOFs. For example, when a mesh is cut, new vertices need to be added to represent the newly generated cross-section. Our method builds on the idea of modifying the underlying graph structure, but also supports the addition of new DOFs, enabling fast physics-based simulation of cutting and tearing.

Enzenhöfer et al. [2019] implement a pivoting solver of LCP based on low-rank downdates to a Cholesky factorization of the system matrix in constrainted multibody simulations involving contacts. In [Cheshmi et al. 2020], SoMod is a new algorithm designed to enable efficient updating of the previously computed factors when constraints are added or removed from the proposed active set.

Hecht et al. [2012] present a fast method for nonlinear FEM simulation by approximating the system matrix with only partial changes. Those changes are performed through incremental updates that do work only in accordance with the modified entries of the matrix. The update cost of [Hecht et al. 2012] is at worst the same as the cost of the original factorization. However, Hecht et al. [2012] only update the numerical value of the Cholesky factor. Topological changes that would require changes of the nonzero pattern of the factor are not considered by [Hecht et al. 2012].

Herholz and Alexa [2018] propose to reuse factorization defined on the full mesh to solve linear problems on sub-meshes. For example, the user selects a set of handles and wants the handle to only affect a region of interest. Herholz and Alexa [2018] only care about the source state (the factor of the entire mesh) and their goal state (the factor of the submesh). All the factors of the submesh is computed from the factor of the entire mesh. No update from submesh to submesh is described. To overcome one of the shortcomings of [Herholz and Alexa 2018], specifically, the fact that the new update always starts from the original Cholesky factor, Herholz and Sorkine-Hornung [2020] present another algorithm to efficiently update Cholesky factors in-place to account for dynamic boundary conditions in the context of interactive surface parameterization. Both [Herholz and Sorkine-Hornung 2020] and our method incorporate subsequent updates on the previously modified factors, avoiding the need to start from the initial factor. Hecht et al. [2012] and Herholz and Sorkine-Hornung [2020] adopt the same strategy to update the matrix: only change the supernodes and their ancestors which need to be recomputed. Both are capable of handling a large number of changes in the factor. We want to compute the Cholesky factor of $\overline{\mathbf{C}}$ from the Cholesky factor of $\mathbf{C}$ where $\overline{\mathbf{C}} = \mathbf{C} + \sigma\mathbf{W}\mathbf{W}^{\mathsf{T}}$, $\sigma = \pm 1$, $\mathbf{W}$ is a matrix representing the update (not the information of the entire mesh). Instead of updating the factor of $\overline{\mathbf{C}}$ using the original factor and the information in $\mathbf{W}$, Herholz and Sorkine-Hornung [2020] compute the factor of $\overline{\mathbf{C}}$ directly from the modified matrix $\mathbf{C}+\sigma\mathbf{W}\mathbf{W}^{\mathsf{T}}$ itself. In order to update column k, all the columns that have a non-zero entry at row k are accessed. For factors stored in column-wise format or supernodal format, this kind of operation yields poor locality. Our method computes column k by only using the information of column k and $\mathbf{W}$. We first allocate a dense workspace that is of the size of $\mathbf{W}$, which all of the modification operates in. Our method thus has good memory locality. The speedup of [Herholz and Sorkine-Hornung 2020] is prominent especially when the underlying mesh is very large; examples shown by [Herholz and Sorkine-Hornung 2020] are on the order of millions of vertices. Real-time physics-based simulation typically requires a more modest resolution. In the typical scenarios where topology changes are localized and gradual (i.e., less than approximately 0.1% of the original DOFs, see Figure 13), our method outperforms [Herholz and Sorkine-Hornung 2020]. The symbolic addition algorithm of [Herholz and Sorkine-Hornung 2020] allows only copies (in terms of connectivity) of existing DOFs to be added; our method allows arbitrary new DOFs to be added. This feature is critical in the surgical simulation where conforming interactive cutting is often required.

In [Yeung et al. 2016, 2018], refactorization of the finite element stiffness matrix is avoided when a cut occurs by a different approach – the Schur complement technique. [Yeung et al. 2018] and our method both target scenarios when the percentage of mesh elements affected by topological changes is small. The authors of [Yeung et al. 2018] mentioned the multiple rank modification module of CHOLMOD, which our method is based on. However, they do not provide a comparison between their method and the CHOLMOD modification module for surgical simulation application because CHOLMOD does not provide the functionality to increase the number of DOFs of the modified system. Yeung et al. [2018] use linear elasticity, assuming that the deformations are small and only limited forces are applied. The algorithm of [Yeung et al. 2018] performs only one iteration at each timestep based on its linear material assumption. Nonlinear models are needed to simulate large deformations of elastic tissues or clothes. These models require multiple linearization iterations to achieve a reasonable solution. In a typical Projective Dynamics solver where 10 iterations are needed, the cost of the update is paid upon each iteration. The cost of one timestep (10 iterations) can be less than exactly 10 times the cost of 1 iteration because the cost can be amortized by memoization in [Yeung et al. 2018].

Projective Dynamics [Bouaziz et al. 2014] is a simple yet robust and fast method to simulate deformable objects. Narain et al. [2016] extended Projective Dynamics to support nonlinear constitutive models and hard constraints. Collisions with static obstacles can be handled as hard constraints in their method. Fratarcangeli et al. [2016] implemented Projective Dynamics on the GPU by using graph coloring. However, GPU compute resources are not always available e.g. in mobile/AR/VR platforms or might be used for other higher priority tasks such as rendering. Reusing matrix factorization remains a useful technique because it is computationally lighter than many iterative linear solvers. Furthermore, common strategies to accelerate the convergences of those iterative solvers take advantage of the topological information of the simulated meshes. For example, multigrid methods such as [Xian et al. 2019] set up the hierarchical linear problems using the rest-pose of the mesh; The Chebyshev semi-iterative method [Wang 2015] accelerates the convergence of the stationary iterative solvers using an estimated spectral radius ρ of the system matrix which also depends on the rest-pose of the mesh implicitly. Our method provides a valuable speedup for applications involving cutting and tearing. Overby et al. [2017] further extended [Narain et al. 2016] to support dynamically changing constraints, which is useful in situations of sliding and contact. Liu et al. [2017] added support for many different types of hyper-elastic materials without compromising the efficiency of Projective Dynamics. Brandt et al. [2018] combined Projective Dynamics with model reduction to achieve further speedups. Wang [2015] uses the Chebyshev method to accelerate the convergence rate of Projective

Dynamics. Li et al. [2019; 2020] couple rigid body dynamics and joint constraints with Projective Dynamics, which is useful in character animation. Ly et al. [2020] add support for contact and friction to Projective Dynamics at a small computational overhead. DiffPD [Du et al. 2021] speeds up back propagation (differentiable physics) using Projective Dynamics. Their contact algorithm is a reasonable trade-off between differentiability and physical plausibility while being compatible with PD. Komaritzan and Botsch [2018] present a method to compute character skinning based on Projective Dynamics. By storing all potential collisions, [Komaritzan and Botsch 2018] benefits from Cholesky prefactorization. The follow-up work [Komaritzan and Botsch 2019] improves the computation performance by a GPU implementation and high-quality up-sampling. A large number of existing methods exploited the prefactorized Cholesky decomposition of a constant system matrix. No efficient method to deal with topological change in Projective Dynamics has been proposed in prior art. To our knowledge, we propose the first method that supports updates of the system matrix including the addition of arbitrary new DOFs not limited to copies of existing ones.

3 BACKGROUND

3.1 Projective Dynamics

Projective Dynamics [Bouaziz et al. 2014] is a fast yet robust method to simulate deformable objects. It treats an implicit time integration step as a minimization problem:

$$g(\mathbf{x}) = \frac{1}{2h^2}\,||\mathbf{x}-\mathbf{y}||^2_{\mathbf{M}} + \sum_j w_j \left\|\mathbf{G}_j\mathbf{x} - \mathbf{S}_j\mathbf{p}\right\|^2_F \tag{1}$$

where the positions of the vertices in the next frame are determined by $\mathbf{x}_{k+1} = \operatorname{argmin}_{\mathbf{x}}\, g(\mathbf{x})$. In the previous equation, $k+1$ is the next frame number, j indexes the element and w_j is a positive weight, $\mathbf{G}_j$ is a discrete deformation gradient operator, $\mathbf{S}_j$ is a selector matrix, $\mathbf{p}$ is an auxiliary projection variable, $\mathbf{y}$ is a constant determined by the current position, velocity and external forces like the gravity, $\mathbf{M}$ is a mass matrix and $h \geq 0$ is the time-step. Projective Dynamics applies a local/global solver to minimize this objective function $g(\mathbf{x})$. The local step is processed in an element-wise fashion to compute the auxiliary projection variable $\mathbf{p}$, while the global step involves a linear system solve to compute the best fit positions $\mathbf{x}^*$ as follows:

$$\mathbf{x}^* = \left(\frac{\mathbf{M}}{h^2} + \mathbf{H}_L\right)^{-1}\left(\mathbf{J}\mathbf{p} + \frac{\mathbf{M}\mathbf{y}}{h^2}\right) \tag{2}$$

where both $\mathbf{H}_L = \Sigma_j w_j \mathbf{G}_j^\mathsf{T}\mathbf{G}_j$ and $\mathbf{J} = \Sigma_j \mathbf{w}_j \mathbf{G}_j^\mathsf{T}\mathbf{S}_j$ are state independent matrices. Liu et al. [2017] interpret $\mathbf{H}_L$ as the Laplacian matrix of the mesh which only depends on the rest-pose of the mesh and the material parameters. The solution of Eq. 2 only requires a linear system solve: $\mathbf{C}\mathbf{x} = \mathbf{b}$. Projective Dynamics assumes that the topology of the mesh is fixed during the simulation, and prefactorizes the system matrix using Cholesky factorization: $\mathbf{C} = \mathbf{L}\mathbf{L}^\mathsf{T}$. During the simulation, the global step only requires a forward and a backward substitution to compute the solution of $\mathbf{C}\mathbf{x} = \mathbf{b}$. Projective Dynamics gains much of its computational efficiency by reusing the factorization during its global solve – at the price of sacrificing the flexibility of changing the mesh topology on the fly, which is precisely what we address in this paper.

Note that the symmetric positive definite system matrix $\mathbf{C}$ comes from two parts, an elastic part, and a mass-matrix part. One of our key ideas is to rewrite $\mathbf{C}$ as $\mathbf{C} = \mathbf{A}\mathbf{A}^\mathsf{T}$, where $\mathbf{A} \in \mathbb{R}^{n\times(m+n)}$ is defined as:

$$\mathbf{A} = \left[\sqrt{w_1}\mathbf{G}_1^\mathsf{T} \quad \sqrt{w_2}\mathbf{G}_2^\mathsf{T} \quad \dots \quad \sqrt{w_m}\mathbf{G}_m^\mathsf{T} \quad \Big| \quad \sqrt{\frac{\mathbf{M}}{h^2}}\right] \tag{3}$$

where m is the number of elements and n is the number of vertices. This notation separates 1) the contribution of elasticity from each element and 2) the contribution of mass from each vertex. This formulation is important especially when we want to add or remove any element or vertex due to topological changes.

3.2 Sparse Cholesky Factorization

Sparse Cholesky factorization has two phases: a symbolic phase that determines the nonzero pattern of $\mathbf{L}$, and a numerical phase that produces the factorization itself. CHOLMOD [Chen et al. 2008] uses the compressed column storage format to store the factors in simplicial numeric mode, which stores entries of the matrix column by column (or row by row) instead of aggregated dense blocks as in the supernodal mode. In this data structure, all the elements in the column-wise list are stored in two contiguous arrays `Li` and `Lx`. `Li` contains the row indices of each element and `Lx` is the actual numerical values. A third array `Lp` points to the beginning of each column in `Li` and `Lx`. `Li` and `Lx` are of the same length, which is the number of non-zero elements in the factor. `Lp` is of length $n+1$ when $\mathbf{C} \in \mathbb{R}^{n\times n}$. For example, if the lower triangular Cholesky factor $\mathbf{L}$ is:

$$\begin{pmatrix} 1.73 & & & & & \\ 0 & 1.41 & & & & \\ 0 & 0 & 1 & & & \\ 0 & -0.70 & 0 & 1.58 & & \\ -0.57 & 0 & 0 & -0.63 & 1.50 & \\ -0.57 & 0 & 0 & 0 & -0.22 & 1.27 \end{pmatrix} \tag{4}$$

`Li` and `Lx` are as shown as below:

Li	0	4	5	1	3	2	3	4	4	5	5
Lx	1.73	-0.57	-0.57	1.41	-0.70	1	1.58	-0.63	1.50	-0.22	1.27

`Lp` points to the beginning of each column:

Lp	0	3	5	6	8	10	11

CHOLMOD uses a doubly linked list (`Lnext` and `Lprev`) to append modified columns at the end of `Li` and `Lx` instead of directly moving entries in `Li` and `Lx`. For example, `Lp[j]` is the starting index for j-th column, `Lp[Lnext[j]] - 1` is the index for the last space for the j-th column. The column indices of `Li` and `Lx` do not need to be monotonic. We refer interested readers to the textbook [Davis 2006] for further information on sparse matrix data structures.

3.2.1 *Symbolic Analysis.*

The first step in the symbolic analysis is to find a good fill-reducing permutation $\mathbf{P}$. Factorizing $\mathbf{P}\mathbf{C}\mathbf{P}^\mathsf{T}$ results in fewer nonzeros in $\mathbf{L}$ (known as "fill-in"). In our method, we use nested dissection ordering in METIS [Karypis and Kumar 2009] because continuous topological changes to the mesh result in limited, well-structured changes to the Cholesky factor. Other permutation methods can be used with our method as well. The non-zero structure of $\mathbf{C}$ can be modeled by a directed graph [Cuthill 1972; Tarjan 1976]. The process of symbolic analysis can be viewed as removing nodes and adding edges on the graph. We refer readers to algorithms 4.2 and 4.3 in a survey [Davis et al. 2016] for this process on elimination graph or quotient graph for more details. This graph-theoretic analysis illuminates the applicability of our method, which incorporates topological changes in simulation directly in the factorization.

3.2.2 Numerical Phase. The numerical phase follows the symbolic analysis by computing the actual numerical value of the non-zero entries. According to the order in which the factor is computed, we can differentiate between left-looking and up-looking Cholesky factorizations. According to whether dense submatrices are exploited, there are simplicial, supernodal and multifrontal methods. Our method uses the up-looking variant, corresponding to CHOLMOD's simplicial LDLT option. This choice has no impact on the run-time computation time since the numerical phase for the original factor is performed only once for the initial mesh before the simulation starts.

3.3 Sparse Cholesky Modification

Davis and Hager [2001] propose a method to update the factor of a symmetric positive definite matrix $\mathbf{C} = \mathbf{A}\mathbf{A}^\top$ when appending a new low-rank component: $\overline{\mathbf{C}} = \mathbf{A}\mathbf{A}^\top + \sigma\mathbf{W}\mathbf{W}^\top$, where $\sigma \in \{-1, +1\}$. This modification is called an *update* if σ is +1 and is called an *downdate* if σ is −1. The modification of the factor of $\overline{\mathbf{C}}$ can be very fast if the rank of $\mathbf{W}$ is significantly smaller than the rank of $\mathbf{A}$.

Here we explain how the factor $\mathbf{L}$ will change in the case when $\sigma = +1$, which corresponds to an update. A downdate is almost identical except for minor changes in the numerical phase of the update process. If the initial sparse symmetric positive definite matrix is in the form $\mathbf{A}\mathbf{A}^\top$, adding $\mathbf{W}\mathbf{W}^\top$ ($\mathbf{W} \in \mathbb{R}^{n\times r}$, $r << n$) can be written as:

$$\mathbf{A}\mathbf{A}^\top + \mathbf{W}\mathbf{W}^\top = [\mathbf{A}|\mathbf{W}]\,[\mathbf{A}|\mathbf{W}]^\top \tag{5}$$

Similarly to full factorization, the update of the factor $\mathbf{L}$ also requires a symbolic phase and a numerical one.

3.3.1 Symbolic Update. First, we introduce some notation in order to explain the symbolic update phase, following the notation from [Davis and Hager 2001]. The nonzero pattern of column j of the factor $\mathbf{L}$ is denoted as $\mathcal{L}_j$,

$$\mathcal{L}_j = \{i : l_{ij} \neq 0\} \tag{6}$$

Similarly, $\mathcal{A}_j$ denotes the nonzero pattern of column j of $\mathbf{A}$,

$$\mathcal{A}_j = \{i : a_{ij} \neq 0\} \tag{7}$$

The elimination tree can be defined in terms of a parent map π. For any node j, $\pi(j)$ is the row index of the first nonzero element in column j of $\mathbf{L}$ beneath the diagonal element,

$$\pi(j) = \min \mathcal{L}_j \setminus \{j\} \tag{8}$$

where $\min \mathcal{X}$ denotes the smallest element of $\mathcal{X}$. The children of node j are the nodes whose parent is j,

$$\{c : j = \pi(c)\} \tag{9}$$

The elimination trees (shown in the right of Figure 2) are constructed from π for all columns: $\pi(j)$ is the parent node of j in an elimination tree. For a matrix of the form $\mathbf{A}\mathbf{A}^\top$, the pattern $\mathcal{L}_j$ of column j is the union of the patterns of each column of $\mathbf{L}$ whose parent is j and each column of $\mathbf{A}$ whose smallest row index of its nonzero entries is j:

$$\mathcal{L}_j = \{j\} \cup \left(\bigcup_{\{c: j=\pi(c)\}} \mathcal{L}_c \setminus \{c\} \right) \cup \left(\bigcup_{\min \mathcal{A}_k = j} \mathcal{A}_k \right) \tag{10}$$

For the example illustrated in Figure 2, $\mathcal{L}_2 = \{2\} \cup \emptyset \cup \mathcal{A}_5$, where $\{j\}$ is $\{2\}$ on the diagonal; $\left(\bigcup_{\{c:j=\pi(c)\}} \mathcal{L}_c \setminus \{c\}\right)$ is $\emptyset$ because column 2 has no child; and k is 5 in $\left(\bigcup_{\min \mathcal{A}_k = j} \mathcal{A}_k\right)$ because $\mathbf{A}(2,5)$ is nonzero. According to Eq. 5 and Eq. 10, the new pattern $\overline{\mathcal{L}}_j$ of column j of $\overline{\mathbf{L}}$ after the update is:

$$\overline{\mathcal{L}}_j = \{j\} \cup \left(\bigcup_{\{c: j=\overline{\pi}(c)\}} \mathcal{L}_c \setminus \{c\} \right) \cup \left(\bigcup_{\min \mathcal{A}_k = j} \mathcal{A}_k \right) \cup \left(\bigcup_{\min \mathcal{W}_i = j} \mathcal{W}_i \right) \tag{11}$$

where $\mathcal{W}_i$ is the nonzero pattern of column i in $\mathbf{W}$. For the example in Figure 2, $\min \mathcal{W}_1 = 1$, then the first column of $\overline{\mathbf{L}}$ should incorporate the nonzero pattern of the first column of $\mathbf{W}$. Similarly, $\min \mathcal{W}_2 = 3$, the third column of $\overline{\mathbf{L}}$ should incorporate the nonzero pattern of second column of $\mathbf{W}$, which are shown as green dots in Figure 2.

The difference between $\overline{\mathcal{L}}_j$ and $\mathcal{L}_j$ comes from two sources: the difference in the second term of Eq. 10 and Eq. 11, which is due to the change of the elimination tree, and the fourth term of Eq. 11 which comes from the low-rank matrix $\mathbf{W}$. $\overline{\mathcal{L}}_j$ can be computed by adding nonzeros to and/or removing nonzeros from $\mathcal{L}_j$ based on these two sources of the difference. We refer to algorithm 3 in [Davis and Hager 2001] for further details.

3.3.2 Numerical Update. Once the symbolic update is finished, a numerical update starts by reordering the columns of $\mathbf{W}$ using a depth-first search. The reordering of $\mathbf{W}$ improves efficiency without affecting the result of $\mathbf{W}\mathbf{W}^\top$. This reordering is analogous to the permutation in the symbolic analysis phase. The algorithm to numerically update the factors operates in a dense workspace of size n by r. All of the intermediate values are stored and accumulated in this workspace. Note that if we want to perform a downdate where $\sigma = -1$, we only need to change the computation in the numerical update phase. In cases where a downdate cancels a nonzero entry in the factor, we will not change the nonzero pattern, but instead, note that entry as a numerical zero (as opposed to structural zeros). We refer to algorithm 5 in [Davis and Hager 2001] for more details.

4 METHOD

Our method is based on the low-rank update technique introduced by Davis and Hager [2001]. However, the original algorithm only supports low-rank updates with DOFs that are already present in $\mathbf{C}$. This is not ideal in applications like the simulation of tearing and cutting, where vertices need to be duplicated or created to represent the cross-sections, changing the size of the new matrix $\overline{\mathbf{C}}$.

Based on the observation that the system matrix of Projective Dynamics can be written as $\mathbf{C} = \mathbf{A}\mathbf{A}^\top$ where the $\mathbf{A}$ matrix is the aggregation of the contributions of elasticity and mass from each element and vertex (Eq. 3), it is straightforward to represent the change of an element or a vertex as low-rank updates. Assuming the

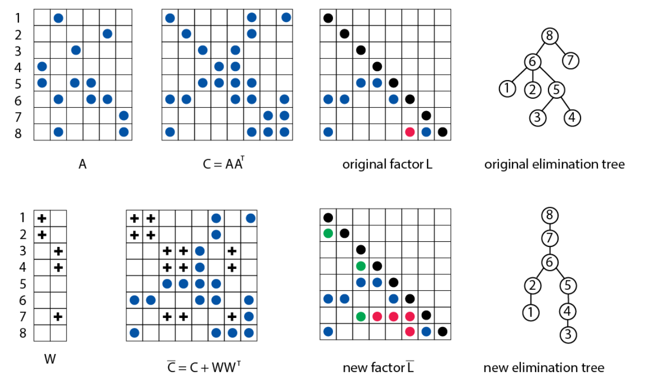

Fig. 2. Nonzero entries in $\mathbf{L}$ have 3 types of source according to Eq. 10. The black dots are from $\{j\}$. The blue dots are from $\left(\bigcup_{\min \mathcal{A}_k=j} \mathcal{A}_k\right)$. The red dots are from $\left(\bigcup_{\{c: j=\pi(c)\}} \mathcal{L}_c \setminus \{c\}\right)$. Nonzero entries in $\bar{\mathbf{L}}$ have 4 types of source. The black dots are from $\{j\}$. The blue dots are from $\left(\bigcup_{\min \mathcal{A}_k=j} \mathcal{A}_k\right)$. The red dots are from $\left(\bigcup_{\{c: j=\bar{\pi}(c)\}} \mathcal{L}_c \setminus \{c\}\right)$. The green dots are from $\left(\bigcup_{\min \mathcal{W}_i=j} \mathcal{W}_i\right)$. Considering that the four types of nonezero sources are not disjoint (multiple sources can contribute to the same nonzero entry), we color the dots with the priority that black > green > red > blue. For example, the first nonzero entry on the diagonal can be colored both black and green. We color it black because the priority of black is higher.

number of vertices is not changed, adding or removing an element e can be seen as a low-rank update $\sigma \mathbf{W}_e \mathbf{W}_e^\mathsf{T}$, where $\mathbf{W}_e = \sqrt{w_e}\mathbf{G}_e^\mathsf{T}$ is the contribution of elasticity of that element. Similarly, changing the mass of a particular vertex v can be seen as a low rank update $\sigma \mathbf{W}_v \mathbf{W}_v^\mathsf{T}$ as well. $\mathbf{W}_v = \sqrt{\frac{m_v}{h^2}}\delta_v$ is the change of mass of that vertex, where δ_v is an indicator column vector of vertex v. But what if the number of vertices needs to be changed? In that case, we can add the new vertices and the new elements in four separate passes. In the first pass, we make room for the incoming vertices by shifting the entries of the current vertices in the factor (Figure 3(b)). Details on how to accommodate the new vertices are shown in Figure 4. New vertices are inserted right before any old vertices connected to them by the element being cut. For example, in Figure 4, vertex 10 is inserted before 6 because 10 is connected with 6 in Figure 3(f). After the insertion, all the vertices are shifted to new positions and the new vertices (e.g. 10) are placed in the vacated column indices (e.g. 6). In this way, the elimination tree is not changed except for adding this new child to the old tree. In the second pass, we remove the corresponding elasticity and mass of the deleted elements (Figure 3(c)). This can be done via a low-rank downdate. In the third pass, we add the vertex masses as "island" vertices to the mesh, as if these new vertices were detached from the object. Because these island vertices are topologically independent of the rest of the mesh, they can only affect the diagonal entries of the expanded system matrix $\bar{\mathbf{C}}$. Therefore, they will only appear as diagonal entries in the corresponding factor $\bar{\mathbf{L}}$. In the fourth pass, we add the elements which can couple several vertices (e.g., springs or linear finite elements) that connect those island vertices to the original mesh. Since this pass does not change the number of vertices, it can be implemented via a low-rank update. To summarize, when a topological change takes place during the simulation, we first reorder the vertices; second, we remove the affected elements and vertices using a low-rank *downdate* without changing the system DOFs; third, we fill the diagonal of the system matrix, adding the mass of the newly added vertices; fourth, we add the new elements to the system matrix and its factor using a low-rank *update*. Figure 5 visualizes the nonzero pattern of the Cholesky factor using our method when cutting a piece of cloth nonconformally.

We explain this key idea of our approach in a didactic example of a simple mass-spring system, illustrated in Figure 3. When we cut a corner involving vertex 9, four steps are made to update the corresponding Cholesky factorization. First, we reorder the vertices. The old vertex 6 is shifted to 7, old vertex 8 is shifted to 10, old vertex 9 is shifted to 13. The new vertices 10, 11, 12 and 13 are inserted at indices 6, 11, 9 and 12. Second, we remove springs 7-13 and 10-13 and the associated masses of vertices 7, 10, and 13 which are all affected by the cut. This corresponds to a low-rank downdate, where the low-rank matrix $\mathbf{W}$ is:

$$\left[\sqrt{w_{7,13}}\mathbf{G}_{7,13}^\mathsf{T} \quad \sqrt{w_{10,13}}\mathbf{G}_{10,13}^\mathsf{T} \quad \middle| \quad \sqrt{\frac{\Delta m_7}{h^2}}\delta_7 \quad \sqrt{\frac{\Delta m_{10}}{h^2}}\delta_{10} \quad \sqrt{\frac{\Delta m_{13}}{h^2}}\delta_{13}\right]$$

Note that we did not remove the masses of vertices 7, 10, and 13 completely, but split them into two parts. We only remove parts of the masses which will compensate for masses of the newly created vertices from the cut. It would also be possible to remove the masses of vertices 7, 10, and 13 completely and add the correct fractions of them back afterward in the last step. Third, we add new vertices 6, 9, 11, and 12 to the system by adding their masses to the diagonal entries of the new factor $\bar{\mathbf{L}}$. Finally, we add the new springs colored in orange in Figure 3(e) to the factor using a low-rank update.

Our method can handle topological changes not only due to cutting and tearing but also due to local remeshing operations, which are useful to improve the quality of finite elements after cutting or large deformation. For example, if we need to swap an edge in a triangle mesh as shown in Figure 6, we only need to do a low-rank downdate to remove the unwanted elements and then perform a low-rank update to add the new ones.

We summarize our proposed workflow for Projective Dynamics supporting topological changes in Alg. 1. The red pseudo code in Alg. 1 highlights the modifications of the original version of Projective Dynamics [Bouaziz et al. 2014].

F_k is a matrix encoding the indices of all of the elements in frame k. In line 1, *TopologicalChange* is a function which tells us how the indices of the mesh elements are changed due to topological changes and mesh refinement. The implementation of this function is application specific and may depend on user interaction. Our method is able to handle addition of arbitrary DOFs. In lines 2 to 5, if any topological changes occur, we will update the relevant variables $\mathbf{y}$, $\mathbf{M}$, $\mathbf{J}$, $\mathbf{H}_L$ and the corresponding Cholesky factor. Line 4 is the core of our method, which is further detailed in Alg. 2. In line 9, the global solve is computed using the Cholesky factor updated by Alg. 2.

Our factorization update process is described in Alg. 2, where we starts with the reordering of the original matrix, corresponding to the rest pose. We found that the nested dissection reordering

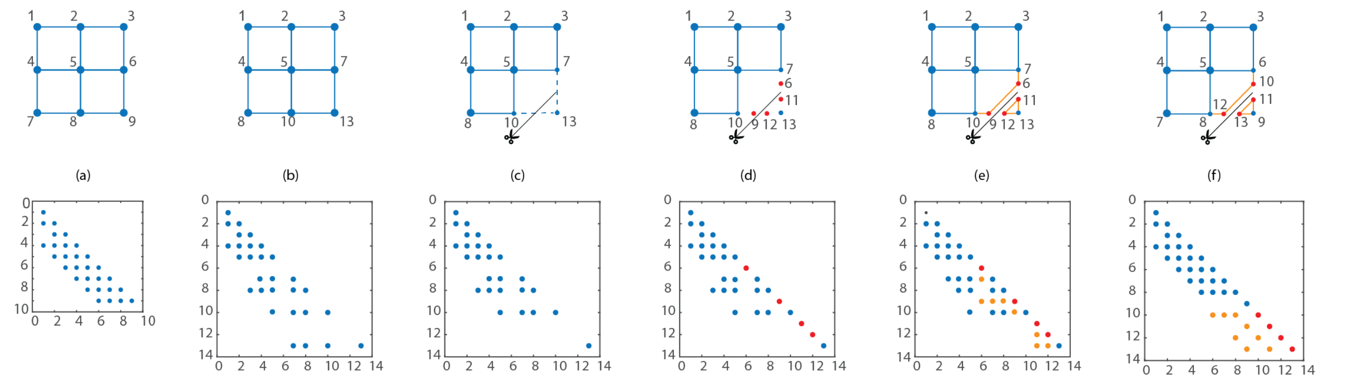

Fig. 3. A walkthrough of making a cut in a mass-spring system. A cut is being made along the springs 8-9 and 6-9. The bottom row shows the corresponding nonzero pattern of the factor. (a) the original mass-spring system. (b) make room for incoming vertices. (c) the system after removing springs 7-13 and 10-13, and also removing the partial masses of vertices 7, 10, and 13. (d) the system after adding the masses of vertices 6, 9, 11, and 12. (e) the system after adding springs 6-7, 6-9, 9-10, 11-12, 11-13 and 12-13 (f) vertices without our reordering strategy. The red dots indicate the new DOFs of the system, and the orange dots correspond to the nonzeros introduced by the new springs.

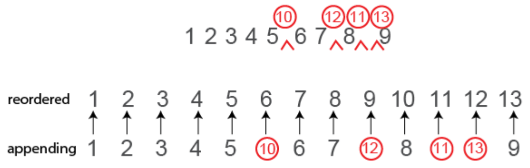
1 2 3 4 5 10 6 7 12 11 13 8 9
reordered 1 2 3 4 5 6 7 8 9 10 11 12 13
appending 1 2 3 4 5 10 6 7 12 8 11 13 9

Fig. 4. Top: inserting four new vertices 10, 11, 12 and 13 into the system described in Figure 3(f). Bottom: the map from the original naturally appending order to the reordered new vertices. For example, vertex 10(Figure 3(e)) is mapped to vertex 6 (Figure 3(f)).

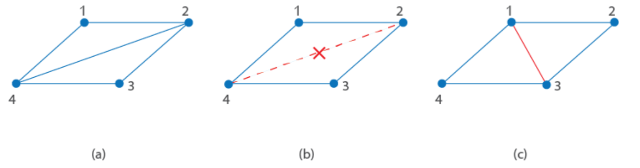

Fig. 6. An example of an edge swap operation [Hoppe et al. 1993]. (a) the original mesh. (b) the mesh after removing the unwanted edge 2-4. (c) the mesh after adding the edge 1-3.

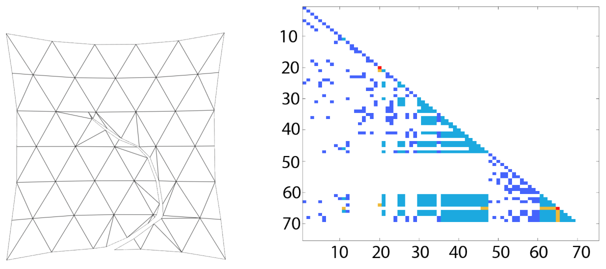

Fig. 5. Left: A piece of cloth (mass-spring) is being cut nonconformally. Right: the corresponding visualization of the Cholesky factor. Dark blue entries remain unchanged. Light blue entries are modified by the topological changes due to the cut. Red entries are the diagonal of the newly added DOFs. Orange entries correspond to new nonzeros added due to the cut.

ALGORITHM 1: Projective Dynamics with Topological Changes

```
1  $F_{k+1}$ = TopologicalChange($F_k$);
2  if $F_{k+1} \neq F_k$ then
3      update $\mathbf{y}, \mathbf{M}, \mathbf{J}$;
4      update $\frac{\mathbf{M}}{h^2} + \mathbf{H}_L$ with its Cholesky factor (Alg. 2);
5  end
6  init: $\mathbf{x}_{k+1}^0 = \mathbf{y}$;
7  for $i = 0, \ldots, max\_iter - 1$ do
8      local step: project $\mathbf{x}_{k+1}^i$ to obtain $\mathbf{p}^{i+1}$;
9      global step: solve $\mathbf{x}_{k+1}^{i+1} = \left[\frac{\mathbf{M}}{h^2} + \mathbf{H}_L\right]^{-1}\left(\mathbf{J}\mathbf{p}^{i+1} + \frac{\mathbf{M}\mathbf{y}}{h^2}\right)$;
10 end
11 $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_{k+1}^{\text{max\_iter}}$;
12 $\mathbf{v}_{k+1} \leftarrow \frac{1}{h}(\mathbf{x}_{k+1} - \mathbf{x}_k)$
```

computed in the rest-pose provides a good fill-in in our experiments. Based on the nested dissection reordering of the original topology, new nodes are inserted and the Cholesky factor is changed accordingly. In a low resolution example where a piece of cloth is cut into two by an S-shaped path, the visualization of one frame (Figure 5) shows that our method maintains moderately good sparsity.

Since our method allows the system matrix to expand during cutting or tearing, extra spaces may be needed. Spaces can be either pre-allocated or added on the fly. We use the doubly linked list in CHOLMOD to book-keep the spaces accordingly. Pre-allocating enough space is efficient for most small scaled cuttings, we applied this strategy in most of our implementations. However, our method

ALGORITHM 2: update $\frac{\mathrm{M}}{h^2} + \mathbf{H}_L$ with its Cholesky factor

```
1  reorder vertices
2  remove elements and partial mass from nodes:
3  downdate via Alg.4, and Alg.5 (dynamic supernodal version) in
     [Davis and Hager 2001]
4  add new nodes:
5  for $j \in$ new vertices do
6      need← $j - n_{\text{old}} + c$;
7      Lp[Lnext[j]] = Lp[j] + need;
8      insert j in the doubly linked list (Lprev, Lnext);
9      recompute Perm;
10     p = Lp[j];
11     Li[p] = j;
12     Lx[p] = $\frac{m}{h^2}$
13 end
14 add elements:
15 update via Alg.3, and Alg.5 (dynamic supernodal version) in [Davis
     and Hager 2001]
```

does not assume any limit of cuttings allowed since we can append discontinuous extra spaces during the simulation and manage these new spaces using the doubly linked list.

Specifically, in CHOLMOD, there are 3 parameters (`grow0`, `grow1` and `grow2`) to control the extra space in the factor if the option `final_pack` is set to False. `Li` and `Lx` are of length $MAX(1, \texttt{grow0})$ times the required space. The space for each column is of length `grow1` $*$ required space + `grow2`. In order to support added DOFs, we introduce a fourth parameter `grow3` to denote the growth of DOFs. `Lp` is of the length n + `grow3`. In Alg. 2, lines 2 and 3 use symbolic and numeric downdates to remove elements in the factor (Figure 3(c)). From line 4 to Line 13, the new DOFs are inserted in the factor to augment the original factor (Figure 3(d)). In line 6, we compute the space needed for the columns of the new DOFs. The constant c is a small integer determined according to the needs of a specific application, e.g., swing example shown in Figure 9 uses $c = 400$. Line 7 leaves room for column j. Line 8 inserts j in the doubly linked list described in Section 3.2. Lines 10-12 put the new nodes on the diagonal and fill them with the corresponding numeric values. Line 14 and 15 use symbolic and numeric updates to add the new elements into the system as illustrated in Figure 3(e).

5 IMPLEMENTATION DETAILS

5.1 LLT v.s. LDLT factorization

Both LLT and LDLT factorizations have almost identical algorithms in terms of symbolic and numerical updates. In CHOLMOD, the factor of LDLT is stored as one sparse lower triangular factor. Because all the diagonal entries of **L** are 1, there is no need to store the diagonal entries of **L**. Instead, the diagonal entries in **D** are stored on the diagonal of the factor. So, there is no difference between LDLT and LLT in terms of memory consumption. With regard to efficiency, the LDLT factorization uses a division operation for the diagonal terms while LLT applies a square root to the same terms which is slightly more expensive. Even though our method is applicable to both LLT and LDLT factorizations, in our implementation we use LDLT factorization to avoid the square root operations during the update and downdate.

5.2 Dynamic Supernodes

The supernodal Cholesky factorization method exploits dense matrix kernels by using supernodes [Liu et al. 1993], i.e., a contiguous set of columns of L with the identical nonzero pattern. The conventional supernode is suitable for the sparse Cholesky factorization of the initial matrix where the nonzero pattern of **L** does not change. After a low-rank change to **C**, because the nonzero pattern changes, the data structure of supernodes changes too. Two or more supernodes may qualify to form a bigger supernode, which is called the merge of supernodes. One supernode may no longer qualify to be a supernode and has to be split. The merge and split of supernodes disable the use of conventional supernodes. Davis and Hager [2009] introduced dynamic supernodes to achieve similar performance as the conventional supernodes while adding low-rank updates/downdates to the initial matrix. Instead of storing the factor in the supernodal form, they store **L** in a compressed column form as described in Section 3.2. This is because data movement is very costly when a supernode is split into two in conventional supernodes. Following the implementation of [Davis and Hager 2009], we also use the simplicial data structure upon which the dynamic supernodes can be detected and used on the fly. If a dynamic supernode is detected, the algorithm described in Section 3.3.2 is unrolled and blocked. We refer to algorithm 2 in [Davis and Hager 2009] for more details. The triangular solve phase can also benefit from the dynamic supernodal technique (algorithm 3 of [Davis and Hager 2009]).

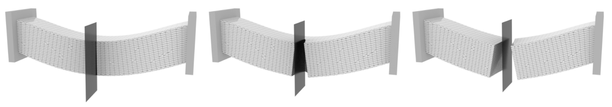

Fig. 7. A bar (tetrahedral mesh) is being cut by a plane.

5.3 Non-conforming Cut

There are various techniques to incorporate cuts into a tetrahedral mesh. Berndt et al. [2017] propose PBD-based method in surgery simulation. Wu et al. [2015] summarize and classify those techniques to six categories: element deletion, splitting along existing faces, element duplication, snapping of vertices, element refinement and snapping plus refinement. Our method can be used with any of the six methods. For example, the cut in Figure 7 splits tetrahedrons along the existing faces. Besides, the cuts can also be represented accurately by the element refinement method, which is used in the stomach example (Figure 10). We use the tetrahedral mesh intersection routine in [Wang et al. 2014], which provides provably-robust cutting with floating-point arithmetics.

6 RESULTS

Table 1 provides details about our example models and the compute time needed for their simulation. In Table 1, “refactorization” is the average time to factorize the system matrix from scratch using

Table 1. Results on our example models. †: the bending springs are not considered in the cloth cut example.

Example	model	#Verts	#Elems	local/global solve	factorization update		total time	
					refactorization	our method	PD + refactorization	PD + our method
cloth cut	cloth†	14,829	29,172	47ms	36ms	2ms	83ms	49ms
spinning cloth	cloth	14,829	29,172	52ms	77ms	15ms	129ms	67ms
swing	cloth	24,168	47,880	125ms	602ms	4ms	727ms	129ms
stomach	volumetric	36,055	13,7796	617ms	786ms	15ms	1403ms	632ms

Table 2. Scalability test on the cloth cut example of different resolutions.

Example	#Verts	#Elems	cut per frame						recomputation
			1	4	8	16	32	64	
cloth cut	3,785	7,326	1ms	1ms	2ms	5ms	9ms	20ms	22ms
cloth cut	14,829	29,172	2ms	5ms	8ms	19ms	26ms	37ms	36ms
cloth cut	58,697	116,424	5ms	13ms	17ms	32ms	56ms	251ms	416ms

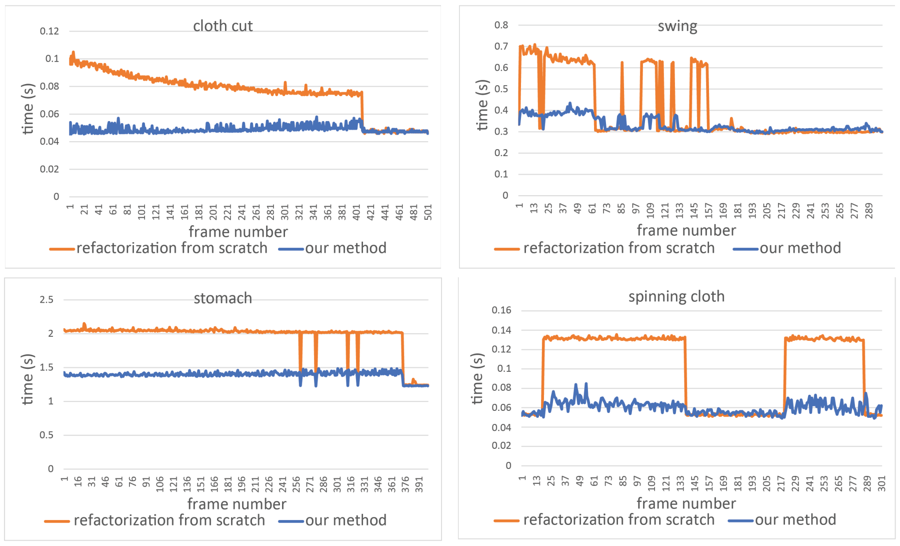

Fig. 8. Per frame compute time for our four examples. Top left: timings for the "cloth cut" animation (Figure 11). Top right: timings for the "swing" animation (Figure 9). The orange curve is jagged because a tearing event does not occur every frame. Bottom left: timings for "stomach" animation (Figure 10). Bottom right: timings for "spinning cloth" animation (Figure 14)

CHOLMOD. "local/global solve" is the solve time for one frame (lines 7-10 in Alg. 1) (i.e., the cost of Projective Dynamics with precomputed factorization). The column "our method" under "factorization update" is the average time to modify the factor using our method. "total time" means the total compute time for one frame (10 local/global iterations for all examples). "PD + refactorization" is the compute time for one frame where the system matrix is factorized from scratch and "PD + our method" reports the compute time for one frame using our factorization update method. All examples were executed on an Intel®Core^{TM} i7-8750H CPU. As we can see from the table, our method significantly reduces the time spent in factorization. Table 2 provides timings of the cloth cut example in Figure 11(b,c) of different resolutions and the number of cuts per frame. The timing is measured as the average of the first five frames. Our method is effective when the number of elements being cut per frame is under 0.1% of the number of total vertices.

The straightforward method of recomputing the factorization in Projective Dynamic when topological changes occur makes the frame-rate fluctuate significantly, as we can see from the orange curves in Figure 8. Compute time fluctuations are undesirable in real-time simulation because the simulation should be synchronized with display refresh rate. Our method solves this problem, adding little overhead even in frames where cutting or tearing occurs, as can be seen in the blue curves in Figure 8.

In Figure 1 and Figure 9, a person is swinging over a creek to escape from a zombie who is dragging his T-shirt. Pieces torn off the T-shirt are still in the system and are being simulated. The T-shirt is simulated using a mass-spring system, while the pants are animated

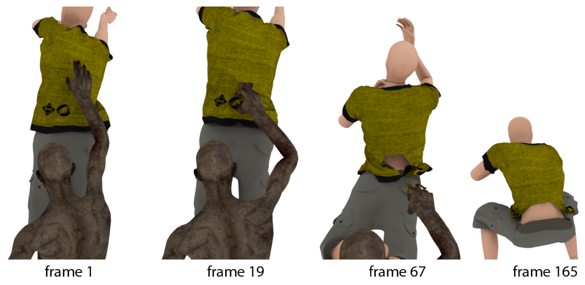

Fig. 9. “Swing” animation. Pieces of cloth from a swinging person are torn off their T-shirt by a zombie.

by skinning. We implemented a simple version of tearing: when the ratio of the length of the spring and the rest-length exceeds some threshold, a seam is created [Grimm 2005]. We prioritize the new seam near the existing seam by lowering the threshold for the neighboring springs since the cloth is prone to tearing near the material that has already been torn. We added contact and friction using the method in [Ly et al. 2020]. In interactive applications such as computer games or surgical simulators, the refactorization in frames needing mesh topology update would introduce lag, degrading the interactive user experience. Our method produces good results despite the fast swinging motion because we end up with the same factor as a full factorization would compute. The timings for each frame in the "swing" animation are shown in the top right graph in Figure 8.

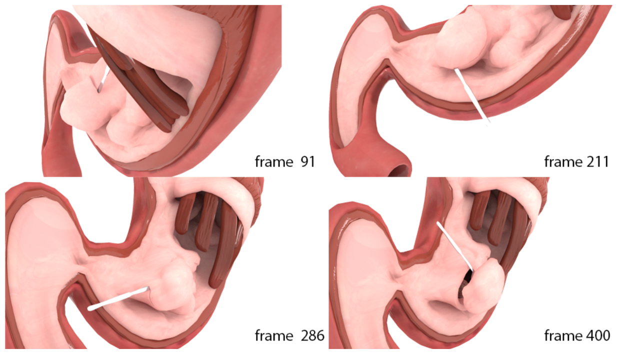

Fig. 10. “Stomach” simulation. Surgical simulation of cutting tissues in a stomach. The pink layer is a volumetric tetrahedron mesh discretized using linear finite elements. The dark red muscles and outer layers are not simulated.

Our method can be particularly useful in surgical simulators. In Figure 10, a scalpel is cutting through an inner layer of a stomach. The timings for each frame in the entire animation are shown in the bottom left graph in Figure 8.

Our reorder strategy (Figure 4) greatly reduces the nonzero items compared to appending all new vertices at the end of the matrix (natural ordering) shown in Figure 3(f). For instance, at the 300th frame of the cloth cut example in Figure 11(b,c), our method has 478,809 nonzeros while natural ordering produces 582,171 nonzeros. The number of nonzeros is 450,425 at the first frame. The solve time at the 300th frame before and after the reorder strategy are 52ms and 50ms.

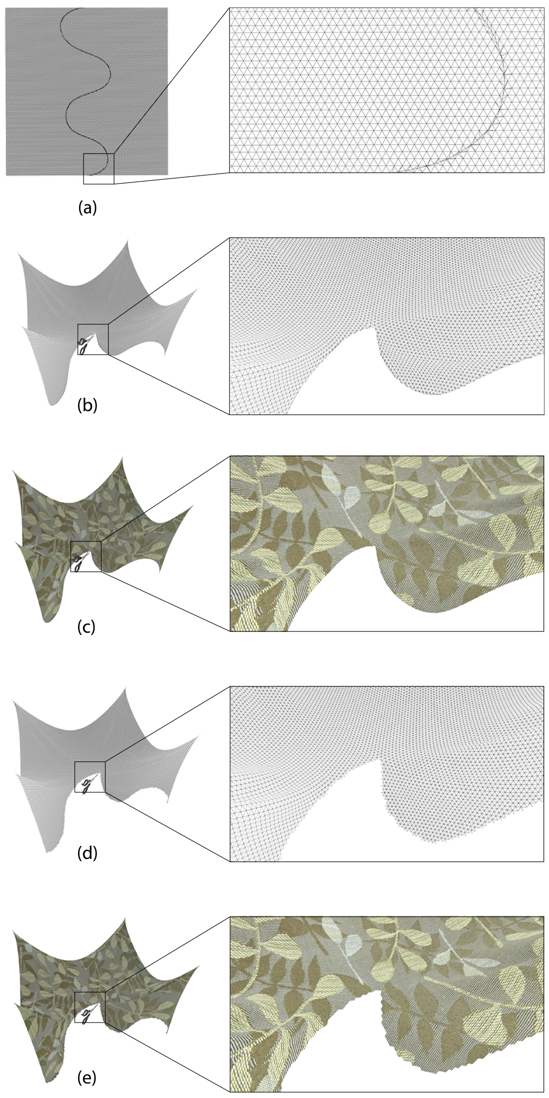

Fig. 11. “Cloth cut” simulation. A smooth S-shaped cut of a cloth. (a) the S-shaped path does not conform to the existing tessellation of the cloth. (b,c) wireframe/textured view of the cloth with a clean cut (with added vertices). (d,e) wireframe/textured view of the cloth using the original tessellation; note the jagged edges.

[Yeung et al. 2018] is based on Schur complement techniques that keep reusing the original system matrix before the topological changes. Therefore, the performance of [Yeung et al. 2018] is greatly impaired with the accumulated topological changes during the history. Our method, on the contrary, commits the factorization update during the simulation. Hence our method is hardly affected by the historical topological changes. We compare our method against [Yeung et al. 2018] using the cloth cut example in Figure 11(b,c) where the number of triangles being split each frame is one. Since

[Yeung et al. 2018] does not separate their linear solver into factorization phase and solving phase, we focus on the total time for one frame (with 10 PD iterations) in this comparison. The results can be seen in Figure 12. The computation time of our method starts from 0.0455s and increases to 0.0505s when the cloth is cut into two pieces. Whereas [Yeung et al. 2018] starts from comparable performance at 0.046s per frame, but quickly increases to 1.236s per frame as the cut accumulates.

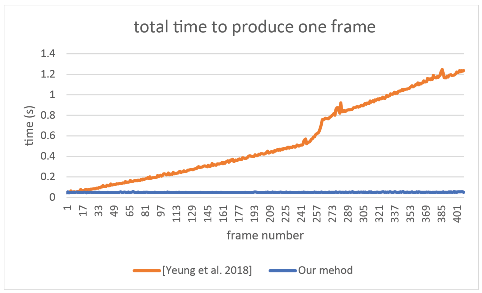

Fig. 12. Timings to produce each frame (with 10 PD iterations) of the example shown in Figure 11(b,c). Our method (blue curve) hardly grows its computational cost as the cut proceeds, while [Yeung et al. 2018] fails to maintain its high efficiency during the simulation.

In contrast to recent methods for sparse Cholesky updates such as [Herholz and Sorkine-Hornung 2020], our method does not assume fixed connectivity of the mesh and is thus able to handle a perfectly clean cut as shown in Figure 11(b, c). The corresponding compute time is shown in the top left graph in Figure 8. If sliver triangles are created during the cut, we can perform an edge swap operation [Hoppe et al. 1993]. As described in Figure 6, the changes in our Cholesky factors corresponding to edge swaps can be easily handled with our method. Our method is effective when the edge swaps are performed occasionally near the cut. We acknowledge that our method would create lots of structural fill-ins if swaps are performed in a large region (for example, in a remeshing application). We would like to point out that our method supports clean cuts for both volumetric meshes such as the stomach example (Figure 10) as well as triangle meshes as shown in Figure 11.

To make a fair quantitative comparison with [Herholz and Sorkine-Hornung 2020], we restrict our method to do a conforming cut as shown in Figure 11(d, e). In those two cases, our method outperforms [Herholz and Sorkine-Hornung 2020] when the update rank is small (Figure 13). We have obtained the implementation from the authors of [Herholz and Sorkine-Hornung 2020] and used the MKL libraries [Intel 2007] for acceleration. [Herholz and Sorkine-Hornung 2020] is less sensitive to the rank being updated, hence is more suitable for mesh parameterization scenarios where large numbers of vertices is added to the system at once. In applications involving interactions, low-rank updates are usually adequate. We repeat the same cutting scene with different numbers of triangles being cut (from 1 to 16) per frame in the accompanying video. The video shows that splitting 16 triangles per frame is already more than needed to perform an interactive cut. In real-life interactive applications under the assumption that the mesh is near-uniformly tessellated, the number of elements being cut is reasonably small (under 0.1% presumably), within the range where our method can be applied efficiently.

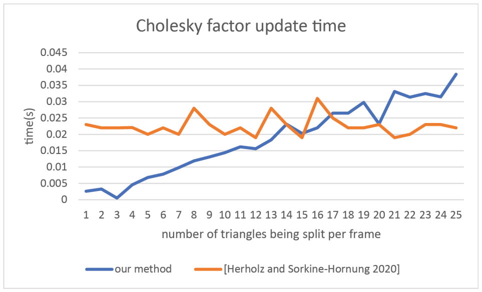

Fig. 13. Timing of Cholesky factor update time of a conforming cut (Figure 11(d,e)), measured as the average of the first 5 frames. Our method (blue) outperforms [Herholz and Sorkine-Hornung 2020] (orange) when the number of triangles being split per frame is small.

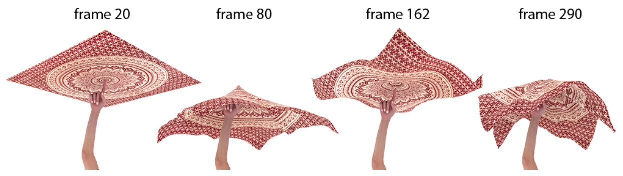

Fig. 14. "spinning cloth" animation. Collision springs are added when the cloth collides with the finger. Self collision is ignored. Bending springs are enabled in this example to generate wrinkles and prevent self-collision while the cloth is spinning.

Besides topological changes as tearing and cutting, our method can also be used to handle simple collisions where a small number of collision springs are instantiated. Figure 14 shows an example where our method is more effective than refactorization from scratch. The computation time of each frame is shown in the bottom right graph in Figure 8. Unlike cutting and tearing where the topological changes of the mesh are permanent, the topological changes during collisions are usually transient. Therefore, we update the Cholesky factor from the original factor instead of from the previous-frame factor in the collision examples. In this "spinning cloth" example, at most 16 collision springs are added in one frame.

7 LIMITATIONS AND FUTURE WORK

Similarly, as in [Davis and Hager 2001], our method requires **W** to be low-rank to ensure the low cost for the Cholesky factorization update. In the case if a user wants to cut through an object very quickly like cutting a watermelon within very few frames in Fruit Ninja [Halfbrick Studios 2010], the performance gain from our method will be compromised. We used dynamics supernodes in CHOLMOD to accelerate the update/downdate process. In the future, we would like to find out whether the arbitrary update/downdate can be done in the supernodal Cholesky factorization without destroying and recreating most of the supernodes.

8 CONCLUSIONS

We present a method to modify Cholesky factors with newly added DOFs, which is particularly suitable for physics-based simulations involving effects such as cutting or tearing. We extend the update and downdate routines in CHOLMOD with the ability to add new DOFs to a system matrix. This extended update and downdate routines can find application in cutting and tearing with Projective Dynamics. Our method works particularly well for low-rank updates, which can be brought into the system matrix by continuous and gradual interaction from the user. Our method supports dynamic remeshing methods such as element refinement and edge swap, therefore, produces clean cuts. The three examples show that our method saves a lot of time when topological changes occur while generating exactly the same results compared to refactorization from scratch.

ACKNOWLEDGMENTS

We thank Bin Wang for insightful discussions, Zhongxing Ma for the modeling of the stomach. This work was supported by National Key R&D Program of China (2018YFB1403900).

REFERENCES

Iago Berndt, Rafael Torchelsen, and Anderson Maciel. 2017. Efficient surgical cutting with position-based dynamics. *IEEE computer graphics and applications* 37, 3 (2017), 24–31.

Sofien Bouaziz, Sebastian Martin, Tiantian Liu, Ladislav Kavan, and Mark Pauly. 2014. Projective dynamics: Fusing constraint projections for fast simulation. *ACM transactions on graphics (TOG)* 33, 4 (2014), 1–11.

Christopher Brandt, Elmar Eisemann, and Klaus Hildebrandt. 2018. Hyper-reduced projective dynamics. *ACM Transactions on Graphics (TOG)* 37, 4 (2018), 1–13.

Yanqing Chen, Timothy A Davis, William W Hager, and Sivasankaran Rajamanickam. 2008. Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. *ACM Transactions on Mathematical Software (TOMS)* 35, 3 (2008), 1–14.

Kazem Cheshmi, Danny M Kaufman, Shoaib Kamil, and Maryam Mehri Dehnavi. 2020. NASOQ: numerically accurate sparsity-oriented QP solver. *ACM Transactions on Graphics (TOG)* 39, 4 (2020), 96–1.

Elizabeth Cuthill. 1972. Several strategies for reducing the bandwidth of matrices. In *Sparse matrices and their applications.* Springer, 157–166.

Timothy A Davis. 2006. *Direct methods for sparse linear systems.* SIAM.

Timothy A Davis and William W Hager. 1999. Modifying a sparse Cholesky factorization. *SIAM J. Matrix Anal. Appl.* 20, 3 (1999), 606–627.

Timothy A Davis and William W Hager. 2001. Multiple-rank modifications of a sparse Cholesky factorization. *SIAM J. Matrix Anal. Appl.* 22, 4 (2001), 997–1013.

Timothy A Davis and William W Hager. 2005. Row modifications of a sparse Cholesky factorization. *SIAM J. Matrix Anal. Appl.* 26, 3 (2005), 621–639.

Timothy A. Davis and William W. Hager. 2009. Dynamic Supernodes in Sparse Cholesky Update/Downdate and Triangular Solves. *ACM Trans. Math. Softw.* 35, 4, Article 27 (Feb. 2009), 23 pages. https://doi.org/10.1145/1462173.1462176

Timothy A. Davis, Sivasankaran Rajamanickam, and Wissam M. Sid-Lakhdar. 2016. A survey of direct methods for sparse linear systems. *Acta Numerica* 25 (2016), 383–566. https://doi.org/10.1017/S0962492916000076

Tao Du, Kui Wu, Pingchuan Ma, Sebastien Wah, Andrew Spielberg, Daniela Rus, and Wojciech Matusik. 2021. DiffPD: Differentiable Projective Dynamics with Contact. *arXiv preprint arXiv:2101.05917* (2021).

Andreas Enzenhöfer, Nicolas Lefebvre, and Sheldon Andrews. 2019. Efficient block pivoting for multibody simulations with contact. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games.* 1–9.

Marco Fratarcangeli, Valentina Tibaldo, and Fabio Pellacini. 2016. Vivace: A practical Gauss-Seidel method for stable soft body dynamics. *ACM Transactions on Graphics (TOG)* 35, 6 (2016), 1–9.

Philip E Gill, Gene H Golub, Walter Murray, and Michael A Saunders. 1974. Methods for modifying matrix factorizations. *Mathematics of computation* 28, 126 (1974), 505–535.

Johannes Grimm. 2005. Tearing of membranes for interactive real-time surgical training. *Studies in health technology and informatics* 111 (2005), 153–159.

Halfbrick Studios. 2010. *Fruit Ninja.* https://apps.apple.com/us/app/fruit-ninja/id403858572

Florian Hecht, Yeon Jin Lee, Jonathan R Shewchuk, and James F O'Brien. 2012. Updated sparse Cholesky factors for corotational elastodynamics. *ACM Transactions on Graphics (TOG)* 31, 5 (2012), 1–13.

Philipp Herholz and Marc Alexa. 2018. Factor once: reusing Cholesky factorizations on sub-meshes. *ACM Transactions on Graphics (TOG)* 37, 6 (2018), 1–9.

Philipp Herholz and Olga Sorkine-Hornung. 2020. Sparse Cholesky updates for interactive mesh parameterization. *ACM Transactions on Graphics (TOG)* 39, 6 (2020), 1–14.

Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. 1993. Mesh optimization. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques.* 19–26.

M Intel. 2007. Intel math kernel library. (2007).

George Karypis and Vipin Kumar. 2009. MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0. http://www.cs.umn.edu/~metis.

Martin Komaritzan and Mario Botsch. 2018. Projective skinning. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 1, 1 (2018), 1–19.

Martin Komaritzan and Mario Botsch. 2019. Fast Projective Skinning. In *Motion, Interaction and Games.* 1–10.

Jing Li, Tiantian Liu, and Ladislav Kavan. 2019. Fast simulation of deformable characters with articulated skeletons in projective dynamics. In *Proceedings of the 18th annual ACM SIGGRAPH/Eurographics Symposium on Computer Animation.* 1–10.

Jing Li, Tiantian Liu, and Ladislav Kavan. 2020. Soft Articulated Characters in Projective Dynamics. *IEEE Transactions on Visualization and Computer Graphics* (2020).

Joseph W. H. Liu, Esmond G. Ng, and Barry W. Peyton. 1993. On Finding Supernodes for Sparse Matrix Computations. *SIAM J. Matrix Anal. Appl.* 14, 1 (1993), 242–252. https://doi.org/10.1137/0614019 arXiv:https://doi.org/10.1137/0614019

Tiantian Liu, Sofien Bouaziz, and Ladislav Kavan. 2017. Quasi-Newton Methods for Real-Time Simulation of Hyperelastic Materials. *ACM Transactions on Graphics (TOG)* 36, 3 (2017), 23.

Mickaël Ly, Jean Jouve, Laurence Boissieux, and Florence Bertails-Descoubes. 2020. Projective Dynamics with Dry Frictional Contact. *ACM Transactions on Graphics* 1 (2020), 8.

Aleka McAdams, Yongning Zhu, Andrew Selle, Mark Empey, Rasmus Tamstorf, Joseph Teran, and Eftychios Sifakis. 2011. Efficient elasticity for character skinning with contact and collisions. In *ACM SIGGRAPH 2011 papers.* 1–12.

Rahul Narain, Matthew Overby, and George E Brown. 2016. ADMM ⊇ projective dynamics: fast simulation of general constitutive models. In *Symposium on Computer Animation*, Vol. 1.

Matthew Overby, George E Brown, Jie Li, and Rahul Narain. 2017. ADMM ⊇ Projective Dynamics: Fast Simulation of Hyperelastic Models with Dynamic Constraints. *IEEE Transactions on Visualization and Computer Graphics* 23, 10 (2017), 2222–2234.

Rasmus Tamstorf, Toby Jones, and Stephen F McCormick. 2015. Smoothed aggregation multigrid for cloth simulation. *ACM Transactions on Graphics (TOG)* 34, 6 (2015), 1–13.

Robert Endre Tarjan. 1976. Graph theory and Gaussian elimination. In *Sparse Matrix Computations.* Elsevier, 3–22.

Huamin Wang. 2015. A chebyshev semi-iterative approach for accelerating projective and position-based dynamics. *ACM Transactions on Graphics (TOG)* 34, 6 (2015), 1–9.

Yuting Wang, Chenfanfu Jiang, Craig A Schroeder, and Joseph Teran. 2014. An Adaptive Virtual Node Algorithm with Robust Mesh Cutting. In *Symposium on Computer Animation.* 77–85.

Jun Wu, Rüdiger Westermann, and Christian Dick. 2015. A survey of physically based simulation of cuts in deformable bodies. In *Computer Graphics Forum*, Vol. 34. Wiley Online Library, 161–187.

Zangyueyang Xian, Xin Tong, and Tiantian Liu. 2019. A Scalable Galerkin Multigrid Method for Real-time Simulation of Deformable Objects. *ACM Transactions on Graphics (TOG)* 38, 6 (2019).

Yu-Hong Yeung, Jessica Crouch, and Alex Pothen. 2016. Interactively cutting and constraining vertices in meshes using augmented matrices. *ACM Transactions on Graphics (TOG)* 35, 2 (2016), 1–17.

Yu-Hong Yeung, Alex Pothen, and Jessica Crouch. 2018. AMPS: A Real-time Mesh Cutting Algorithm for Surgical Simulations. *arXiv preprint arXiv:1811.00328* (2018).