## Sparse Cholesky Updates for Interactive Mesh Parameterization

## PHILIPP HERHOLZ, ETH Zurich, Switzerland OLGA SORKINE-HORNUNG, ETH Zurich, Switzerland

We present a novel linear solver for interactive parameterization tasks. Our method is based on the observation that quasi-conformal parameterizations of a triangle mesh are largely determined by boundary conditions. These boundary conditions are typically constructed interactively by users, who have to take several artistic and geometric constraints into account while introducing cuts on the geometry. Commonly, the main computational burden in these methods is solving a linear system every time new boundary conditions are imposed. The core of our solver is a novel approach to efficiently update the Cholesky factorization of the linear system to reflect new boundary conditions, thereby enabling a seamless and interactive workflow even for large meshes consisting of several millions of vertices.

# $\label{eq:ccs} \texttt{CCS Concepts:} \bullet \textbf{Computing methodologies} \to \textbf{Mesh models}; \bullet \textbf{Mathematics of computing};$

Additional Key Words and Phrases: Mesh parameterization, boundary conditions, sparse matrix factorization

#### **ACM Reference Format:**

Philipp Herholz and Olga Sorkine-Hornung. 2020. Sparse Cholesky Updates for Interactive Mesh Parameterization. *ACM Trans. Graph.* 39, 6, Article 202 (December 2020), 14 pages. https://doi.org/10.1145/3414685.3417828

## 1 INTRODUCTION

Mesh parameterization has been an active area of research for many years. During this time a large number of techniques for different purposes and applications have been proposed. Most of these techniques have not yet been adopted by practitioners using 3D modeling packages liker Blender or 3ds Max. One of the main reasons for this is scalability. For an artist working on a detailed mesh, a smooth, interactive and predictable workflow might be more important than geometric guarantees.

For a large class of algorithms, the scalability problem stems from the need to solve large sparse linear systems whenever the boundary conditions, interactively imposed by an artist, change. Since the linear systems usually change in a few coefficients only, one could ask weather it is possible to reuse information from a given matrix factorization in order to compute a new one. In fact, a number of methods exist to update Cholesky factors in such a way; however, we find most of them to exhibit unfavorable scaling behavior when applied to typical editing scenarios. Especially for larger meshes with more than 100k to several million vertices, these

Authors' addresses: Philipp Herholz, ETH Zurich, Switzerland, philipp.herholz@inf. ethz.ch; Olga Sorkine-Hornung, ETH Zurich, Switzerland, sorkine@inf.ethz.ch.

0730-0301/2020/12-ART202 \$15.00

https://doi.org/10.1145/3414685.3417828

methods commonly perform even worse than complete refactorization. This is because most of these existing methods are targeted at a very different scenario: They can be very efficient when frequently performing low-rank updates, e.g., symmetric changes to a row/column pair. The Schur complement based method of Yeung et al. [2018], for example, is able to efficiently simulate mesh cutting during physics based animation. The situation for interactive parameterization is quite different. Here, the user does not perform small changes to the boundary at every frame, but rather updates the boundary conditions much less frequently, affecting a larger number of vertices each time. The situation is similar for interactively imposing positional boundary conditions in Laplacian based shape editing to define a region of interest [Botsch and Sorkine 2008a]. Another limitation of update algorithms is their limited support for adding new variables representing new vertices in the mesh to the linear system. The algorithm presented in [Herholz and Alexa 2018] is not capable of changing the system dimension and has only limited support for iterative updates. The row addition algorithm implemented in Cholmod [Chen et al. 2008] is capable of adding a symmetric row/column pair, but the initial factorization needs to be aware of this future update during construction and store an empty row and column in its place. This is useful in the context of linear programming, where constraints have to be switched on and off during optimization, but prohibits the use of this algorithm for interactive mesh editing.

We propose a different method, building on partial refactorization [Hecht et al. 2012; Herholz and Alexa 2018]. The basic idea is to augment an optimized Cholesky factorization algorithm that typically constructs the sparse factor column by column. By analyzing dependencies between numerical values in the Cholesky factor, we can skip columns that are not affected by a change in the underlying system matrix and selectively update the columns that actually change. This procedure is less efficient when the change takes place in only a few columns of the system matrix (less than 10), but it outperforms competing methods for larger updates. This method therefore exhibits the performance characteristics we require for our applications.

We describe how this basic idea can be applied to interactively impose Dirichlet boundary conditions by removing degrees of freedom from the factorization (see Section 4.2). This enables us to interactively introduce new seams into the mesh. Cutting along a seam duplicates vertices along the cut, increasing the system dimension. In Section 5.1 we demonstrate that, in many cases, this does not require the addition of new degrees of freedom into the factorization. In other cases, however, additional vertices at the newly introduced seams have to be represented in the factorization. We describe how to handle this case by explicitly changing the structure of the Cholesky factor in Section 4.2. In Section 5.2 we present an example application that imposes linear constraints on seam vertices requiring such a structural update. For this application we

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org. © 2020 Association for Computing Machinery.

show an example of how to use general linear equality constraints in conjunction with our Cholesky updates.

This set of techniques allows us to perform very general updates to the factorization of linear systems, including the addition of degrees of freedom. We demonstrate that our algorithm improves upon related work in terms of performance for a large range of examples, especially if more than just a few degrees of freedom have to be updated at once. In Section 6 we specify this statement by comparing and evaluating our algorithms systematically with respect to previous work. We demonstrate the scalability of our algorithms using example meshes of up to 14 million vertices.

## 1.1 Contributions

- We present an algorithm to efficiently update Cholesky factors to account for changes in Dirichlet boundary conditions in the context of interactive surface parameterization. The method efficiently removes degrees of freedom from the system (Section 4.1).
- We extend the method to also support the addition of degrees of freedom to Cholesky factors. The required symbolic update takes advantage of the supernodal factor representation (Section 4.2).
- We demonstrate the effectiveness of our approach by comparing it to competing methods over a set of meshes at different scales.

#### 2 RELATED WORK

A number of works address updating sparse Cholesky factors, targeted at various applications. Here we review recent techniques that can be efficiently used to solve some of the problems we are considering in this paper.

*Low-Rank Updates.* Gill et al. [1972] present a set of techniques to update the Cholesky factorization of a dense matrix  $\mathbf{A} \in \mathbb{R}^{n \times n}$  after a rank-1 modification of the form

$$\mathbf{A}' = \mathbf{A} + \mathbf{v}\mathbf{v}^{\mathsf{T}},\tag{1}$$

with  $\mathbf{v} \in \mathbb{R}^n$ . The algorithms iterate over the columns of the Cholesky factor and update them consecutively. If A represents the system matrix of a least squares regression problem, the update can be interpreted as adding a data point to the problem. Davis and Hager [1999] present a variant of this technique applicable to sparse Cholesky factors. Translating the method to the sparse setting is complicated by the fact that the nonzero pattern of the factor might change which requires a sophisticated analysis of symbolic dependencies in the factor. Sorkine et al. [2005] present a specialized version of rank-1 updates in the context of shape approximation, where the update matrix contains only a single nonzero element on its diagonal. Instead of updating the Cholesky factor with the more general technique of Davis and Hager [1999], they apply Givens rotations on the matrix A and its Cholesky factor to eliminate the newly introduced value, which is faster in this special case.

General rank-*m* updates can be applied as a series of rank-1 updates. Davis and Hager [2000] show that multiple updates can be handled more efficiently. Unfortunately, this technique only provides a relatively small constant speedup compared to sequences of

ACM Trans. Graph., Vol. 39, No. 6, Article 202. Publication date: December 2020.

rank-1 updates. In contrast, our algorithm scales much better with the number of modified columns and is applicable in a more general setting than Equation (1). In subsequent work, the same authors describe an efficient implementation using dynamic supernodes [Davis and Hager 2009], which is implemented in CHOLMOD [Chen et al. 2008] and used for comparisons in Section 6.

A symmetric modification of a row and column of A can be interpreted as a rank-2 update. This special case allows for a dedicated approach optimized for this specific scenario [Davis and Hager 2005]. The case of row and column addition and removal is handled separately. Deleting a row/column pair replaces it by the corresponding row and column of the identity matrix which is also the case for our method in Section 4.1. A precondition for adding a row and column is that an empty row and column already exists in that position. This is useful in the context of linear programming, where there is a fixed set of constraints that have to be switched on and off dynamically. However, in our setting this renders the algorithm impractical, as we don't know a priori where and how many additional degrees of freedom are introduced. Row deletion (chomod\_rowdel) is implemented in CHOLMOD [Chen et al. 2008] and can be used to eliminate degrees of freedom from the factorization, therefore it constitutes a direct competitor of the algorithm we describe in Section 4.1. We also compare to row addition (cholmod\_rowadd), despite being not directly suited for mesh editing, by artificially anticipating the update during the initial factorization. We demonstrate in Section 6 that our method is significantly more efficient for the types of updates that occur in the application scenarios we consider. We also provide a theoretical justification for this drastically different scaling behaviour.

*Schur Complement Approaches.* Schur complement based methods are able to reuse the factorization of an initial matrix **A** to solve a linear system obtained by appending rows and columns to **A**. Yeung et al. [Yeung et al. 2016, 2018] demonstrate how this basic principle can be used to implement far more general updates. They apply their approach in the context of interactive mesh cutting and demonstrate the effectiveness of their incremental updates. As their algorithms can be used for the applications we are interested in, we provide a brief overview in Section 3.5 and compare to these methods in Section 6.

*Seletive Solving.* By exploiting the properties of the elimination tree, Herholz et al. [2017] desribe how to compute selected values of the solution to a linear problem. This allows to efficiently parameterize local patches of a shape as long as Neumann boundary conditions, or more generally, boundary conditions that can be formulated in terms of the right hand side, are imposed. The method can also be employed to efficiently compute smooth exponential maps [Herholz and Alexa 2019] for local parameterization.

Partial Refactorization. Herholz et al. [2018] present a method targeted at updating factorizations for mesh deformation. The idea is to start with the Cholesky factorization  $L^0$  of a linear system defined on the full mesh and use this information to compute the Cholesky factor of a sub mesh that is supposed to be deformed. This method performs best if regular and localized regions of the mesh (for example, an arm of a humanoid shape) are selected, with all

other vertices fixed in their position. Enlarging the region of interest by only one vertex requires a complete, new update starting from the original Cholesky factor. In the context of dynamically changing boundary conditions we have exactly the opposite situation: the constraints are located at vertices that do not form a local region but span many regions along an edge path; the region of interest is the full mesh except for a few constrained vertices. Moreover, we would like to make a large number of iterative changes to the constraints, which quickly becomes inefficient if we have to reconstruct the Cholesky factor every time starting at L<sup>0</sup>. In contrast, our proposed method modifies the Cholesky factor directly and in-place instead of constructing it based on initially available information, which would require copying and manipulating large amounts of data for every update. Our subsequent updates always start at a situation where all previously set constraints are already accounted for, which is especially beneficial for incremental modification of boundary conditions.

Approximate Methods. For the physically based simulation of deformable shapes, Newton's method is commonly employed, usually requiring the solution of a linear system defined by the energy Hessian for each simulation step. Linear methods assume a constant Hessian at the cost of severe artifacts for large deformations. Co-rotational methods [Müller and Gross 2004] reuse the initial factorization and employ local rotations to mitigate these artifacts. The method of Hecht et al. [2012] selectively updates the Cholesky factor for co-rotational methods whenever the current Hessian changes drastically in a certain region. Similar to our method, they exploit the structure of the elimination tree, defined in Section 3.2, to find all columns affected by an update. In contrast to our methods as well as low-rank updates and Schur based methods, approximate methods do not maintain a valid factorization at all times. This gives the method considerably more flexibility which can translate into performance benefits. However, this depends on the type of the numerical problem at hand, specifically the stiffness of the system matrix. Our algorithms maintain a correct representation of the current Cholesky factor without accumulating error over iterations due to exact partial refactorization which makes the method more predictable and arguably easier to use.

## 3 BACKGROUND

This section presents some key concepts and algorithms required to compute the sparse Cholesky factorization  $\mathbf{A} = \mathbf{L}\mathbf{L}^{\top}$  of a symmetric and positive definite matrix  $\mathbf{A} \in \mathbb{R}^{n \times n}$ . In our case **A** commonly represents the cotan Laplacian of a triangle mesh with *n* vertices. The *k*-th vertex corresponds to the *k*-th row and *k*-th column in the matrix. Since the cotan Laplacian is positive semi-definite, we add a small multiple ( $\epsilon = 10^{-8}$ ) of the identity matrix as regularizer. In this section we only highlight details that are relevant for our method and refer to textbooks on this topic (e.g., [Davis 2006]) for a more complete overview.

The Cholesky factor  $\mathbf{L} \in \mathbb{R}^{n \times n}$  is a lower triangular matrix that enables the fast solution of a linear system  $A\mathbf{x} = \mathbf{b}$  with  $\mathbf{x}, \mathbf{b} \in \mathbb{R}^n$  by means of forward- and backsubstitution. A crucial feature of the matrix  $\mathbf{L}$  is that the rows and columns of a sparse matrix  $\mathbf{A}$  can usually be reordered using a permutation, such that  $\mathbf{L}$  is also



Fig. 1. The different steps of the Cholesky algorithm, as detailed in Algorithm 1, with access patterns highlighted. The factorization of the input matrix (upper left) is built column by column. To compute the k-th column, information from all columns that have a nonzero in row k are accessed.

sparse. This reordering merely amounts to a reordering of the mesh vertices and can be done in a preprocess. There are several methods for finding a useful reordering; prominent examples are approximate minimum degree reordering or nested dissection. In this paper we assume that the mesh vertices have already been ordered in a useful way.

## 3.1 Sparse Cholesky Algorithm

The factorization algorithm is divided into two phases. During the symbolic phase, the nonzero structure of the sparse factor is determined and allocated. In the numeric phase, the actual values of the nonzero elements are computed. Here we focus on the latter. Figure 1 demonstrates the different steps of the algorithm: Given a symmetric and positive definite input matrix A (Figure 1a), the factor is constructed column by column starting from the left. To compute the k-th column (Figure 1b), the steps detailed in Algorithm 1 have to be performed. We use MATLAB notation for parts of a matrix, for example L(k : n, i) for rows k to n of column i of the sparse matrix L. The key point here is the access pattern visualized in Figure 1 (the sub figures are also referenced in Algorithm 1). To compute any column, only a small subset of previously computed columns is accessed. Moreover, the nonzero pattern of the new column is directly determined by nonzero positions in columns indexed by  $I_k$ (Algorithm 1, step (2)).

## 3.2 The Elimination Tree

The elimination tree encodes dependencies among columns in the Cholesky factor and can be easily constructed during the symbolic factorization phase. For details on the efficient construction of the elimination tree we refer the reader to [Davis et al. 2016]. Each node of this tree represents a vertex of the original mesh and a column of the sparse matrix **A**. The elimination tree helps answer an important question: which columns of **L** access information

(1) Copy entries from the input matrix (1c)  

$$\mathbf{L}(k:n, k) \leftarrow \mathbf{A}(k:n, k).$$

(2) Find the set I<sub>k</sub> of all column indices < k referencing a (structural) non zero element in row k. (1d)</li>

$$I_k \leftarrow \{i \in \mathbb{N} \mid i < k \text{ and } \mathbf{L}_{ki} \neq 0\}$$

(3) Update the *k*th column using all columns indexed in  $I_k$  (1e)

$$\mathbf{L}(k:n, k) \leftarrow \mathbf{L}(k:n, k) - \sum_{i \in I_k} \mathbf{L}(k, i) \mathbf{L}(k:n, i).$$

(4) Square root of diagonal element

 $L(k,k) \leftarrow \sqrt{L(k,k)}.$ (5) Divide by the diagonal element (1f)

 $\mathbf{L}((k+1):n, k) \leftarrow \mathbf{L}((k+1):n, k)/\mathbf{L}(k,k).$ 

ALGORITHM 2: Sparse Cholesky factorization
Name: Cholesky(A)
Input: System matrix A.
Output: Cholesky factor L.
$\overline{L \leftarrow 0}$ .
for $k = 0$ to $n - 1$ do
CholeskyColumn(A, k, L)
end
ALGORITHM 3: Sparse Cholesky update

Name: CholeskyUpdate(A, L A') Input: Matrix A, Cholesky factor L, modified matrix A'. Output: Cholesky factor L' of A'.

 $I_0 \leftarrow$  set of columns that differ between A and A'.

 $I_1 \leftarrow$  set of indices visited while traversing the elimination tree to the root for all  $i \in I_0$ .

for k = 0 to n - 1 do if  $k \in I_1$  then CholeskyColumn(A, k, L') end end

from a given column k of **L** – directly or indirectly – for their construction. In other words: if the numerical values in a specific column k change, we can identify the set of columns that have to be adapted in order to maintain a valid factorization. To this end

ACM Trans. Graph., Vol. 39, No. 6, Article 202. Publication date: December 2020.

we traverse the elimination tree up to its root starting at node k. The columns corresponding to all visited nodes have to be updated. It is also possible to determine column dependencies in the other direction and find the set of columns  $I_k$  that are accessed in order to compute a specific column k in the Cholesky factor. In this case the elimination tree is traversed downwards starting at a set of indices given by the original matrix **A**. More specifically, we start at nodes connected to k in the graph structure of the sparse matrix **A** that have an index smaller than k:

$$\{i \in \mathbb{N} \mid i < k \text{ and } \mathbf{A}_{ki} \neq 0\}.$$

In order to produce a sparse factor, it makes sense to strive for a balanced elimination tree by reordering columns and rows of the input matrix, which would limit the number of

accessed columns while constructing the k-th column (see Figure 1e). Because the nonzero pattern is determined by the accessed columns (Section 3.1), a well balanced elimination tree also fosters sparsity. However, a balanced elimination tree is not a prerequisite for sparsity. A diagonal matrix, for example, has a degenerate elimination tree but still features optimal sparsity without fill-in. For general sparse matrices, it has still proven effective to permute the input matrix such that a well balanced elimination tree is achieved. Nested dissection is a prominent re-  $\frac{1}{1}$ ordering strategy based on this observation. For this paper we use Metis [Karypis and Kumar 1998], a highly optimized nested dissection implementation.



#### 3.3 Updates

(1f)

Fast updates to sparse Cholesky factors are based on the properties of the elimination tree discussed above. We assume that we have a factorization  $\mathbf{A} = \mathbf{L}\mathbf{L}^{\mathsf{T}}$ . If the matrix  $\mathbf{A}'$  is identical to  $\mathbf{A}$  except for numerical values in a few columns without changes to the nonzero pattern, we can employ a sparse update algorithm on L to obtain the Cholesky factor L' of A'. To this end, we first build a list  $I_0$  of all columns that differ between A and A'. From Algorithm 1, step (1), we can conclude that all columns in  $I_0$  also differ between L and L'. Using the elimination tree we can now identify all columns that depend on columns in  $I_0$  for their construction. We have to traverse the elimination tree starting from every index  $i \in I_0$  and up to its root, collecting all discovered nodes into the set  $I_1$ . All columns in  $I_1$  potentially differ between L and L', while all other columns are identical. This observation suggests a simple algorithm to update L in order to construct L'. We can just run the numerical factorization algorithm (Algorithm 1) on L for each column that is not part of  $I_1$ and still end up with the correct factor L'. In Algorithm 3 we list the necessary steps.

An important observation in this context is that updating e.g. column 14 in our example requires exactly the same computational work as updating both 14 and 17, since the union of their paths to the root is the same as the path for node 14 alone. More generally, it is quite efficient to simultaneously update columns that are close in the elimination tree, compared to consecutive updates. Luckily,

reordering techniques implicitly organize vertices such that vertices that are close in terms of graph distance are in most cases also close in the elimination tree and reside in the same small subtree. For most meshes it is safe to assume that mesh graph distance between two vertices correlates with their geodesic distance. This makes updates to A affecting a local region on the mesh quite efficient (see [Herholz and Alexa 2018] for further details).

Another important property for updates computed this way is numerical stability. Sequentially updating numeric quantities always bears the possibility of building up numerical error. For partial refactorization we always recompute all values that change from scratch and reuse information from unaffected subtrees. As a consequence updating a factor or recomputing it completly will always lead to exactly the same floating point values.

### 3.4 Supernodal Implementation

Sparse matrices can be represented in compressed column format, which stores all nonzero entries of a column in a dense vector, together with the corresponding row indices. We show an example of a sparse matrix below, along with its compressed column representation (2a). Numbers to the left of the vertical bars represent row indices.

To obtain an efficient implementation of Cholesky factorization and updates, the concept of supernodes can be helpful. The idea is to treat neighboring columns that share an identical, or in some implementations just similar, nonzero pattern in the factor as one *supernode*, by storing dense matrices instead of vectors. This comes at the cost of explicitly representing some zeros, but the row information for each supernode is only stored once (2b).

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 6 & 7 & 0 & 4 & 0 \\ 0 & 0 & 8 & 9 & 5 \end{pmatrix} \nearrow \begin{bmatrix} \begin{pmatrix} 0 & 1 & 1 \\ 3 & 6 \end{pmatrix}, \begin{pmatrix} 1 & 2 \\ 3 & 7 \end{pmatrix}, \begin{pmatrix} 2 & 3 & 3 \\ 4 & 8 \end{pmatrix}, \begin{pmatrix} 4 & 5 \end{pmatrix} \end{bmatrix} (2a)$$
$$\gg \begin{bmatrix} \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 2 \\ 3 & 6 & 7 \end{pmatrix}, \begin{pmatrix} 2 & 3 & 0 & 0 \\ 4 & 8 & 9 & 5 \end{bmatrix}$$
(2b)

The efficiency of factorization algorithms can significantly improve when working with a supernodal representation. It allows considering several columns at once, which improves the applicability of highly optimized dense math libraries like Intel's MKL [Intel 2009] in different steps of the factorization. The supernodal implementation can be derived by replacing columns with supernodes in Algorithm 1. The mathematical operations of the algorithm have to be replaced by corresponding operations on matrices (supernodes). To give just one example, step (4) becomes

$$L(k0:k1, k0:k1) \leftarrow Cholesky(L(k0:k1, k0:k1)),$$

for a supernode spanning columns k0 to k1. Cholesky(.) can be implemented employing any dense Cholesky factorization. Details can be found in [Davis 2006]; the most important takeaway for our work is that columns can be grouped into supernodes that are processed together during factorization.

## 3.5 Schur Complement based Methods

The Schur complement is a powerful tool for progressively modifying linear systems [Yeung et al. 2016] and parallelization [Chu et al. 2017; Liu et al. 2016]. Suppose we have a positive definite and symmetric linear system with the following block form where  $\mathbf{A} \in \mathbb{R}^{n \times n}$  is a square matrix for which a factorization is already known and  $\mathbf{A}_1 \in \mathbb{R}^{n \times m}$ ,  $\mathbf{A}_2 \in \mathbb{R}^{m \times m}$ :

$$\begin{pmatrix} \mathbf{A} & \mathbf{A}_1 \\ \mathbf{A}_1^\top & \mathbf{A}_2 \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{pmatrix}.$$
 (3)

Instead of solving the system (3) by factorizing it from scratch, we can reuse the factorization of A by computing the solution in two steps using block-wise row elimination. Elimination of  $A_1^{\mathsf{T}}$  yields the equivalent system

$$\begin{pmatrix} \mathbf{A} & \mathbf{A}_1 \\ \mathbf{0} & \mathbf{A}_2 - \mathbf{A}_1^{\mathsf{T}} \mathbf{A}^{-1} \mathbf{A}_1 \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 - \mathbf{A}_1^{\mathsf{T}} \mathbf{A}^{-1} \mathbf{b}_1 \end{pmatrix}.$$
 (4)

In order to compute the solution, we first determine  $\mathbf{x}_2$  by constructing and solving the system

$$\underbrace{\left(\mathbf{A}_{2}-\mathbf{A}_{1}^{\mathsf{T}}\mathbf{A}^{-1}\mathbf{A}_{1}\right)}_{\mathbf{P}}\mathbf{x}_{2}=\mathbf{b}_{2}-\mathbf{A}_{1}^{\mathsf{T}}\mathbf{A}^{-1}\mathbf{b}_{1},$$
(5)

this requires *m* solves using the system A and the construction and solution of a dense  $m \times m$  system, which is quite efficient for  $m \ll n$ . Moreover, we can use the identity

$$\mathbf{A}_{1}^{\mathsf{T}}\mathbf{A}^{-1}\mathbf{A}_{1} = \mathbf{A}_{1}^{\mathsf{T}}\left(\mathbf{L}^{-1}\right)_{\mathsf{T}}^{\mathsf{T}}\mathbf{L}^{-1}\mathbf{A}_{1}$$
(6)

$$= \left(\mathbf{L}^{-1}\mathbf{A}_{1}\right)^{\mathsf{T}}\mathbf{L}^{-1}\mathbf{A}_{1}, \qquad (7)$$

which allows us to limit the computational work to a few backsubstitutions to compute  $L^{-1}A_1$ , followed by a matrix product to compute  $A_1^{\mathsf{T}}A^{-1}A_1$ . After  $\mathbf{x}_2$  has been determined, we can use the known factorization of A again to solve

$$\mathbf{A}\mathbf{x}_1 = \mathbf{b}_1 - \mathbf{A}_1\mathbf{x}_2.$$

The matrix **B** can be explicitly constructed and factorized; however, iterative approaches without the need to construct **B** explicitly are possible [Chu et al. 2017]. This basic technique can be used to incorporate updates to linear systems in several ways. Given a factorization of a linear system **A**, for example in the context of a linear physical simulation, Yeung et al. [2018] describe how to impose Dirichlet boundary constraints at a few vertices indexed in *B* with |B| = m

$$\min_{\mathbf{x}\in\mathbb{R}^n} \mathbf{x}^{\mathsf{T}} \mathbf{A} \mathbf{x}$$
  
s.t.  $\mathbf{x}_i = \mathbf{c}_i, \quad i \in B,$ 

without recomputing the factorization completely by solving the augmented system

$$\begin{pmatrix} \mathbf{A} & \mathbf{H}_B \\ \mathbf{H}_B^{\mathsf{T}} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \hat{\mathbf{x}} \\ \mathbf{y} \end{pmatrix} = \begin{pmatrix} \mathbf{b} \\ \mathbf{0} \end{pmatrix}. \tag{8}$$

This system can also be derived using Lagrange multipliers **y**. Here,  $\mathbf{H}_B \in \mathbb{R}^{n \times m}$  contains columns of the identity matrix corresponding to constrained vertices.  $\hat{\mathbf{x}} \in \mathbb{R}^n$  contains the correct values for unconstrained vertices, i.e.  $\mathbf{x}_i = \hat{\mathbf{x}}_i$  for all  $i \notin B$ . Applying the Schur complement technique, yields the equation

$$\mathbf{A}\hat{\mathbf{x}} = \mathbf{b} - (\mathbf{H}_B^{\mathsf{T}} \mathbf{A}^{-1} \mathbf{H}_B)^{-1} (\mathbf{H}_B^{\mathsf{T}} \mathbf{A}^{-1} \mathbf{b}).$$
(9)

ACM Trans. Graph., Vol. 39, No. 6, Article 202. Publication date: December 2020.

The main computational work to incorporate new boundary conditions is again the construction and factorization of a system of the type seen in (6) and solving a linear system; the factorization of **A** can be reused for all updates. More general symmetric updates to a linear system, for example when cutting a mesh during a linear FEM simulation, might require changes to the matrix itself apart from just constraining vertices and possibly the addition of new degrees of freedoms in the form of mesh vertices. Yeung et al. [2018] show how to employ Schur complements even for this more general case. The main computational steps are again the construction and factorization of a system of size  $m \times m$  where *m* refers to the number of modified and added column/row pairs followed by solving a linear system involving **A**.

For both cases, interactively imposing Dirichlet boundary conditions and more general structural changes, the main computational burden of the update is the construction of a linear system of the form

$$\mathbf{H}_{B}^{\mathsf{T}}\mathbf{A}^{-1}\mathbf{H}_{B} \tag{10}$$

where  $H_B$  contains unit vectors and the solution of two additional linear systems. For updates of medium size, i.e.  $20 < m \ll n$ , the construction of (10) dominates runtime in most of our experiments.

In Section 6 we demonstrate that our method outperforms these approaches in typical editing scenarios. However, when many small updates have to be made to a system, for example in the context of fracture in a physics based simulation, Schur complement based methods are more efficient. In that sense the methods complement each other.

#### 4 METHOD

Our central technical contribution consists of two techniques that are able to update Cholesky factors in order to reflect changes in the system matrices. The first approach is able to impose Dirichlet constraints and thereby removing degrees of freedom from the system. We detail this method in Section 4.1. As many applications we are targeting require only the removal of degrees of freedom, we offer this dedicated approach that is more efficient and easier to implement, at the cost of being less general compared to the second method. This method allows for the addition and removal of variables of a linear system, representing a change in the number of degrees of freedom in the underlying mesh. Section 4.2 describes the necessary symbolic (i.e. the nonzero pattern) and numeric updates. Both methods are based on the sparse Cholesky update algorithm 3. While removing degrees of freedom can be accomplished by changing numerical values only, we need to change the symbolic structure for adding degrees of freedom too.

#### 4.1 Removing Degrees of Freedom

The removal of degrees of freedom is a common operation when imposing Dirichlet boundary conditions for parameterization with a fixed boundary, or in interactive mesh deformation [Botsch and Sorkine 2008b]. Assuming the Cholesky factorization of a system matrix **A** corresponding to a mesh is available, we would like to solve the following constrained optimization problem:

where *B* represents a set of constrained vertices with associated fixed positions  $\mathbf{u}_i$  and  $\mathbf{U} \in \mathbb{R}^{n \times d}$  contains the sought vertex positions. If, for example, d = 2, the boundary conditions form a convex polygon and **A** represents the combinatorial discrete Laplacian, we obtain a Tutte embedding.

A common way of solving this problem is to consider the set of interior vertices

$$I = N \setminus B \quad \text{with} \quad N = \{0, \dots, n-1\} \tag{12}$$

and solve the linear system

$$\mathbf{A}_{II}\mathbf{U}_{I} = -\mathbf{A}_{IB}\mathbf{U}_{B}.$$
 (13)

Here, the notation  $A_{IB}$  selects rows and columns referenced in I and B, respectively, akin to Matlab's slicing operation A(I, B) where I and B are lists of integers.

While previous work [Herholz and Alexa 2018] tries to construct the factorization of  $A_{II}$  by extracting and updating components of the Cholesky factor L of A, we take a different approach that is more efficient for subsequent updates required during interactive parameterization. In contrast to Schur based methods, we modify the factorization directly, avoiding scalability issues due to the construction of (10).

The key idea is that in order to remove degrees of freedom, it is sufficient to obtain the factorization of the matrix  $A^B$ , which is obtained from A by setting all rows and columns that are indexed by *B* to zero and placing the value 1 on the respective diagonals:

$$\mathbf{A}_{ij}^{B} = \begin{cases} 0 & \text{if } (i \in B \lor j \in B) \land i \neq j, \\ 1 & \text{if } i \in B \land i = j, \\ \mathbf{A}_{ij} & \text{else.} \end{cases}$$
(14)

This way we effectively detach the constrained vertices from the rest of the mesh by canceling all interactions. Note that the matrix  $A^B$ inherits positive definiteness from A, because all principle minors of  $A^B$  must be non-negative if this is the case for A. Obtaining a factorization of  $A^B$  is also the strategy employed by the row deletion algorithm of Davis et al. [Davis and Hager 2005], however, they follow a very different update strategy to achieve that goal.

The Cholesky factorization

$$\mathbf{A}^{B} = \mathbf{L}^{B} \left( \mathbf{L}^{B} \right)^{\mathsf{T}} \tag{15}$$

can be efficiently computed by updating the given factorization  $\mathbf{A} = \mathbf{L}\mathbf{L}^{\top}$ . To this end, we apply Algorithm 3, where the set  $I_0$  of columns that differ between  $\mathbf{A}$  and  $\mathbf{A}^B$  can be deduced from Equation (14). All columns that depend on these columns in the Cholesky factor are identified using the elimination tree and updated subsequently. As we show in Appendix A, rows and columns of  $\mathbf{L}^B$  corresponding to nodes in *B* will be identical to the rows and columns of the identity matrix in the same way as for  $\mathbf{A}^B$ . This allows for an even more efficient update procedure as these columns can be skipped during numerical factorization and replaced by the

ACM Trans. Graph., Vol. 39, No. 6, Article 202. Publication date: December 2020.



Fig. 2. Splitting vertex 2 in the input mesh (left) into two new vertices (right) results in a new Laplacian (second row) with a column and row added.

corresponding unit vector. Because of this property we can conclude that

$$\mathbf{A}_{II} = \mathbf{L}_{II}^{B} \left( \mathbf{L}_{II}^{B} \right)^{\mathsf{T}},\tag{16}$$

holds which means that we have access to the Cholesky factorization of  $A_{II}$  as soon as  $L^B$  has been constructed using the update algorithm. Using this Cholesky factor we can solve (13), which provides us with a solution for (11).

It should be noted that the Cholesky factor of  $L_{II}$  obtained from refactoring  $A_{II}$  is numerically identical to  $L_{II}^B$ , but the latter might contain additional explicit zeros. This is a consequence of our method to remove degrees of freedom by setting coefficients to zero explicitly. We find this effect to be negligible in our experiments (Section 6).

## 4.2 Adding Degrees of Freedom

Defining new seams in interactive parameterization or simulating fracture requires adding degrees of freedom. Suppose a mesh is cut along a vertex path. We have to duplicate all vertices and edges on that path except for the start and end vertices. The resulting system has a larger dimension compared to the original matrix, but most coefficient remain equivalent. Yeung et al. [2018] show that a solution of the modified system can be obtained using Schur complements (Section 3.5). Our approach follows a different idea. Instead of reusing the original factorization in order to compute a solution to the modified system, we want to update the existing factorization to become the new factorization. This approach leads to an algorithm with very different performance characteristics compared to previous work.

The idea is to actually add rows and columns to the factorization of A to account for the additional vertices. The key to an efficient implementation of this idea is the supernodal approach used in fast Cholesky factorization algorithms (Section 3.4). The update is divided into a symbolic and a numeric phase.

To illustrate the technique, we assume that vertex k of the mesh is cut into two, as in Figure 2. The question is how the structure of the corresponding Cholesky factorization changes and how we can

ALGORITHM 4: Symbolic DOF Addition
Name: AddDegreeOfFredomSymbolic(L, k)
Input: Cholesky factor L.
Output: Symbolic updated factor L.
foreach supernode S do
ShiftIndizesGreaterK( $\mathbf{S}, k$ )
<pre>if ContainsRow(S, k) then</pre>
InsertRow(S, $k + 1$ )
end
<pre>if ContainsColumn(S, k) then</pre>
AppendColumnToSupernode(S)
end
end

compute the new factorization by updating the old one. The generalization to more than one new boundary vertex is straightforward.

Symbolic Update. Computing the Laplacian of both meshes (second row of Figure 2) shows a similar structure, which comes as no surprise, since splitting a vertex only affects its immediate neighbours. Structurally we can construct the new Laplacian from the old one by duplicating the kth row and column, which amounts to labeling the new vertex as k + 1 and shifting all following vertex labels by one. This way we are introducing coefficients representing connections between vertices that are not connected in the new mesh (gray lines and crosses). These coefficients are assigned the value zero despite being explicitly represented in the sparse matrix. By following this recipe for all duplicated vertices we guarantee that the structure of the elimination tree is also changed in a very predictable way: the new vertex is just inserted after the original one (top row in Figure 2). As a consequence, the overall structure of the Cholesky factor changes in a systematic way: again, just a row and a column are duplicated. In particular, the structure of all other supernodes not directly affected by the update does not, change apart from the duplicated rows. From an algorithmic point of view, the new structure can be created following the steps detailed in Algorithm 4.

For applications in parameterization as well as for cutting or fracture simulation of triangle or tetrahedral meshes, it is sufficient to consider the case of duplicating vertices, as described above, or merging previously duplicated vertices. Removing vertices by merging follows the same steps as duplication, by replacing insertion of a row and column by their removal. Computationally, both operations require virtually the same amount of work. Our implementation is capable of duplicating and removing a large amount of degrees of freedom at once: the required modifications to Algorithm 4 are straightforward. Instead of shifting row indizes by 1 they are now shifted according to the number of added and removed rows preceding it.

More general structural updates beyond duplicating and merging vertices are possible, albeit more complex, as the structure of the elimination tree might change in different ways. As a consequence, structural updates of supernodes not directly affected by the update become necessary. As we are focusing on the application of interactive mesh editing for parameterization, we did not explore these more general updates.

Numerical Update. After performing the symbolic update, we have a factor that has the correct nonzero structure to store the factorization of the modified system matrix. However, the values of modified or added rows and columns are not correct and might represent some structural zeros explicitly. At this point the setting is exactly the same as for the basic update algorithm described in Section 3.3: some columns contain invalid values and we have to traverse the elimination tree starting at these nodes and identify all nodes that are affected by the update. Running Algorithm 2 only on these supernodes results in the updated factorization. This makes the implementation of our dimension modifying update particularly simple. We can reuse the basic update procedure (Alg. 3), which is just a slight modification of the basic numerical factorization algorithm 2, for which highly optimized implementations exist.

*Example.* Here we illustrate the difference between the original factor (17) and the updated one (18) using the example from Figure 2. Both matrices are also given in their supernodal form (17/18 b). The update introduces a copy 2' of vertex 2, which is reflected by a new row and column in the system matrix and the factor. The update of column 2 and 2' propagates through the factor to nodes 3 and 4, as can be seen from the elimination tree of the new factor (Figure 2, top right). Looking at the supernodal form of the new factor (18b), we can see that the numerical values in the first supernode do not change, as expected, only row indices are shifted to account for the newly introduced row. The second supernode is expanded by one column and row and updated numerically. It is important that we merely split nodes, which has a very predictable and local effect on the factor. Introducing a vertex connecting different parts of the mesh would affect it in a significantly more complicated way.

FACTOR

$$\begin{pmatrix} 1.7 & 0 & 0 & 0 & 0 \\ 0 & 1.7 & 0 & 0 & 0 \\ -0.58 & -0.58 & 1.5 & 0 & 0 \\ -0.58 & -0.58 & -0.44 & 1.5 & 0 \\ -0.58 & -0.58 & -1.1 & -1.5 & 0.022 \end{pmatrix}$$
(17a)

SUPERNODAL STRUCTURE:

$$\begin{bmatrix} 0 & 1.7 \\ 2 & -0.58 \\ 3 & -0.58 \\ 4 & -0.58 \end{bmatrix}, \begin{pmatrix} 1 & 1.7 & 0 \\ 2 & -0.58 & 1.5 \\ 3 & -0.58 & -0.44 \\ 4 & -0.58 & -1.1 \end{bmatrix}, \begin{pmatrix} 3 & 1.5 & 0 \\ 4 & -1.5 & 0.022 \end{pmatrix}$$
(17b)

ACM Trans. Graph., Vol. 39, No. 6, Article 202. Publication date: December 2020.

MODIFIED FACTOR:

( 1.7	0	0	0	0	0
0	1.7	0	0	0	0
-0.58	0	1.3	0	0	0
0	-0.58	0	1.3	0	0
-0.58	-0.58	-0.26	-0.26	1.4	0
-0.58	-0.58	-1.0	-1.0	-1.4	0.0077

MODIFIED SUPERNODAL STRUCTURE:

$$\begin{bmatrix} 0 & 1.7 \\ 2 & -0.58 \\ 4 & -0.58 \\ 5 & -0.58 \end{bmatrix}, \begin{pmatrix} 1 & 1.7 & 0 & 0 \\ 2 & 0 & 1.3 & 0 \\ 3 & -0.58 & 0 & 1.3 \\ 4 & -0.58 & -0.26 & -0.26 \\ 5 & -0.58 & -1.0 & -1.0 \end{bmatrix}, \begin{pmatrix} 4 & 1.4 & 0 \\ 5 & -1.4 & 0.0077 \end{pmatrix}$$
(18b)

*Implementation.* A key to an efficient implementation of the symbolic update is the supernodal structure (18b). We store each supernode, consisting of row indizes and a dense value matrix, as a separate instance in contrast to the traditional representation that uses arrays storing all row indizes and values sequentially. The benefit of this storage scheme is that rows and columns can be inserted efficiently per supernode without shifting the position of all following supernodes in the sparse factor. Moreover, the structure update can be easily parallelized on a per supernode basis when using this representation. The structural transformation from (17b) to (18b) illustrates this property. We have found that our storage scheme has a negligible performance impact during the numerical phase even for the full initial factorization despite of possible worse cache coherency.

#### 5 APPLICATIONS

#### 5.1 Interactive Conformal Parameterization

A natural application for updating Dirichlet boundary conditions for parameterization are Tutte embeddings, described in Section 4.1. While this is already a practical application, we are interested in further artistic freedom regarding the mapping of the boundary. The recent boundary first flattening (BFF) method of [Sawhney and Crane 2017] can significantly benefit from our update algorithm. The key idea of this method is to express the solution of several parameterization design objectives entirely with respect to the boundary, and fill in the interior vertices by solving the constrained optimization problem (11). Design objectives can be fixed cone angles, a target boundary shape or minimal area distortion. Here we focus on parameterizations with minimal area distortion, which can be characterized as having conformal scaling factors of zero at the boundary. Shawney and Crane [2017] describe how to compute corresponding boundary angles by solving another instance of (11) and integrate them to produce the final boundary. Combined with our update algorithm, this makes for a very useful interactive parameterization method. Whenever the user decides to modify the boundary, our method updates the system matrix that is used to solve two instances of Dirichlet-constrained quadratic optimization to compute the new parameterization. Since refactorization is by



Fig. 3. These images from our interactive application (based on code provided with [Sawhney and Crane 2017]) show the initial situation and the updated one after cutting the mesh ( $\approx$ 100k vertices) and optimizing for the conformal parameterization with minimal area distortion. Constructing and solving the system from scratch takes 410 milliseconds versus 24 milliseconds with our approach.



Fig. 4. By selecting two vertices our application prototype computes a seamless orbifold Tutte embedding. Direct solution of the KKT system (20) requires 3.4 seconds, our algorithm just 320 milliseconds.

far the most expensive operation in BFF, the speedups of about an order of magnitude or more for reasonable seam lengths reported in Section 6.1 apply to BFF as well. Other parameterization objectives for BFF might require the computation of conformal scaling factors from boundary curvature. In these cases the factorization of the full Laplacian is required, which calls for an update that explicitly introduces new boundary vertices into the factorization, as described in Section 4.2. We integrated our update algorithm into the code provided by the authors of [Sawhney and Crane 2017] and show an interactive editing session in the accompanying video, highlighting the achieved speedup in a typical session.

#### 5.2 Orbifold Tutte Embeddings

The second application we consider constructs tileable parameterizations of genus zero meshes. The work of Aigerman et al. [2017] generalizes the concept of Tutte embeddings in the plane to Euclidean orbifolds, which are flat surfaces except for a discrete set of cone points. Orbifolds can be used to tile the plane  $\mathbb{R}^2$ , which makes them particularly interesting for seamless mesh parameterization. The resulting parameterization depends on the choice of a seam and cone vertices along which the surface is cut open (see Figure 4). Since the quality of the resulting map can be vastly different depending on the specific choice of seam, an interactive tool allowing the user to modify the seam without refactoring a system matrix can be useful. Moreover, orbifold Tutte embeddings have also been employed in the context of convolutional neural networks [Maron et al. 2017], where it is necessary to compute many different embeddings of the same object. In contrast to the previous application, we now need to represent vertices along the seam explicitly as two separate vertices and cannot just eliminate them from the system.

Figure 4 shows an input mesh that has been cut along a seam (in red) by splitting each edge along an edge path into a pair of edges. The orbifold Tutte embedding (right) maps the highlighted points onto the corners of a square and enforces tileability by ensuring that edge pairs are related by a rotation of  $\pi/2$  about one of the fixed points. Additionally, smoothness across tile boundaries is explicitly enforced. This makes for three types of linear constraints:

- (1) positional constraints on four vertices,
- (2) rotational constraints on edge pairs,
- (3) smoothness across tile boundaries.

The rotational constraints couple *x* and *y* coordinates in the parameterization space and prevent us from solving separate problems for both dimensions, which is possible for regular Tutte embeddings. All constraints on the coordinates  $\mathbf{X} \in \mathbb{R}^{n \times 2}$  in parameterization space can be expressed in terms of a matrix  $\mathbf{B} \in \mathbb{R}^{m \times 2n}$  and a vector  $\mathbf{b} \in \mathbb{R}^m$ , where *m* is the number of linear constraints. We obtain a Tutte orbifold embedding by minimizing the Dirichlet energy subject to these linear constraints, which amounts to solving the following optimization problem:

where vec(·) stacks the columns of a matrix into a vector; the system matrix A represents the cotan Laplacian. This optimization problem can be solved by introducing a vector of Lagrange multipliers  $\lambda \in \mathbb{R}^m$  and forming the KKT system

$$\begin{pmatrix} \mathbf{A} & \mathbf{0} & \mathbf{B}^{\mathsf{T}} \\ \mathbf{0} & \mathbf{A} & \mathbf{B}^{\mathsf{T}} \\ \hline \mathbf{B} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \operatorname{vec}(\mathbf{X}) \\ \lambda \end{pmatrix} = \begin{pmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{b} \end{pmatrix}.$$
 (20)

For details on this transformation we refer to [Nocedal and Wright 2006]. The authors of the original paper [Aigerman et al. 2017] directly solve the system (20) using a sparse LU solver every time boundary conditions change. Unfortunately we cannot immediately apply our update algorithms, as they are limited to positive definite matrices; however, we can apply the Schur complement technique

(Section 3.5) to eliminate the matrix B in the last row, leading to

$$\begin{pmatrix} \mathbf{A} & \mathbf{0} & \mathbf{B}^{\mathsf{T}} \\ \mathbf{0} & \mathbf{A} & \mathbf{B}^{\mathsf{T}} \\ \hline \mathbf{0} & | & \mathbf{-C} \end{pmatrix} \begin{pmatrix} \operatorname{vec}(\mathbf{X}) \\ \lambda \end{pmatrix} = \begin{pmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{b} \end{pmatrix}, \quad \mathbf{C} = \mathbf{B} \begin{pmatrix} \mathbf{A}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}^{-1} \end{pmatrix} \mathbf{B}^{\mathsf{T}}$$
(21)

With this formulation it is possible to solve the system in three steps. First, the matrix C is constructed and used to solve for the values  $\lambda$ . Using these values the actual positions X can be computed.

$$\lambda = -\mathbf{C}^{-1}\mathbf{b} \tag{22}$$

$$\operatorname{vec}(\mathbf{X}) = -\begin{pmatrix} \mathbf{A}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}^{-1} \end{pmatrix} \mathbf{B}^{\mathsf{T}} \lambda.$$
(23)

A key benefit here is that we can solve this system based on a Cholesky factorization of **A**, which is significantly more efficient to compute than the LU factorization of the full system. More importantly for our setting, however, is the fact that we can now employ structural updates to **A** in order to account for the definition of new seams. We implement orbifold Tutte embeddings using the formulation (21) in combination with our update algorithm, described in Section 4.2. Figure 4 and the accompanying video show our interactive application, demonstrating the practical effectiveness of our update formulation.

## 5.3 Mesh Deformation

While we developed our solver specifically for the task of mesh parameterization, it can also be applied in more general settings. We demonstrate how our techniques can be used for as-rigid-as-possible mesh deformation [Sorkine and Alexa 2007]. One of the problems in these types of algorithms is the need to refactor the system whenever the handle or the region of interest is changed by introducing or removing Dirichlet boundary constraints. This is similar to the change in boundary conditions during interactive parameterization. The main difference is that now a larger number of vertices may be constrained to specific positions and that refactorization is only sporadically necessary during an editing session. Consequently our method can only help to improve performance for this particular part of the algorithm, especially if many incremental changes to boundary conditions take place. To adapt our algorithm to this new situation, we exploit the information about which columns have been replaced by a unit vector. It is easy to see that the corresponding column in the factor is also a unit vector (see Appendix A for details), such that we can just replace columns of the factor by a unit vector instead of running Algorithm 1. Moreover, while computing the values for a column that needs to be updated via Algorithm 2, we do not have to take these unit columns into account, since they do not contribute any values to the update (we remove these columns from the set  $I_k$  in the second step of Algorithm 2).

We have integrated our Cholesky updates into a Blender plugin demonstrating the use of our algorithm in a production environment. The accompanying video features an editing session posing the armadillo model.

## 6 EVALUATION

We evaluate our solver on a set of meshes of varying sizes and randomly generated cuts to systematically asses the performance of our algorithms. We start with a closed manifold mesh, randomly



Fig. 5. Meshes used for our experiments. The value of *n* refers to the number of mesh vertices.

select two vertices and compute the shortest edge path between them using Dijkstra's algorithm. Since the performance of our updates depends on how many paths in the elimination tree have to be traversed, we expect the performance to degrade in a nonlinear manner with respect to path length. Tracing shortest paths between vertices is therefore a worst case scenario, as these paths tend to cut through nodes belonging to many different subtrees compared to a set of vertices that is concentrated in a small region on the mesh. We still see significant speedup factors compared to previous methods: about one order of magnitude in most situations.

The path lengths cover a wide range of sensible editing scenarios. Note that the number of vertices forming a seam on a mesh grows with  $O(\sqrt{n})$ , where *n* is the number of vertices, when uniformly increasing mesh resolution. As an example, a regular mesh of a halfsphere with 90 vertices has 25% boundary vertices, while a mesh of the same shape with 200k vertices has 0.5% boundary vertices. For the buddha mesh with 1M vertices, for example, we consider path lengths of up to 6,000 vertices. Vertically cutting the mesh through the center along one of its main axes always results in seam lengths below 2,500 vertices. Even though cutting a mesh for parameterization might require many individual seams, we expect users to introduce them one by one, and therefore believe that our evaluation even goes beyond typical interaction scenarios. In the context of mesh deformation (Section 5.3), the position of large parts of the mesh often has to be constrained, however, in all cases it suffices to constrain vertices at the interface between the current region of interest and the rest of the mesh.

For the experiments we sample paths of specific lengths l in the following way: We select a vertex at random, compute the shortest graph distance to all others and pick a second vertex with the desired distance. If all vertices are closer than l, we restart the procedure with a different seed vertex. If this fails several times, we split the length and look for k individual random paths of length l/k. We use this experimental setup to perform various updates, as described in the following sections.

We compare with different competing methods. For the algorithms described in [Yeung et al. 2018] and [Herholz and Alexa 2018], we use code provided by the authors. The row/column addition and deletion algorithm presented in [Davis and Hager 2005] is available in CHOLMOD [Chen et al. 2008]. As performance critical code is using BLAS and LAPACK routines across all implementations, we always link with Intel's BLAS and LAPACK implementation provided as



Removing degrees of freedom

Fig. 6. Performance comparisons for Cholesky updates and refactorization. Our algorithm scales very well with mesh and update size while competing methods are better for relatively small updates.

part of MKL [Intel 2009] for a fair comparison. We use the ICC compiler with the same settings for all results and comparisons.

### 6.1 Updating Parameterization Seams

Figure 6 illustrates that our update algorithm compares favorably to competitively over a large range of mesh sizes and experiments. We do not include any setup cost for computing the cut mesh or setting up the Laplacian, which would benefit our method. We use code provided with [Sawhney and Crane 2017] to introduce cuts and construct the new mesh. Figure 6 shows results for four different meshes ranging from 170 thousand to 14 million vertices. For very small updates, the row deletion algorithm of Davis and Hager [2005] and Schur complement based updates [Yeung et al. 2018] are usually more efficient; however, when introducing a new seam with more than 20 vertices, our method is faster, in many cases even by orders of magnitude.

As expected, our method is slightly slower for introducing new degrees of freedom. This is caused by two factors. First, the symbolic update described in Section 4.2 has to be computed for these types of updates and secondly, the numeric phase now has to update a system that is slightly larger. Overall the performance of adding and removing degrees of freedom is comparable. We observe that our updates are always faster than full refactorization. This is due to the fact that in a worst case scenario, our method updates all columns and therefore degrades to refactorization. While the performance is not realtime for extremely large meshes and long cuts, we see updates for the Lucy model with 14 million vertices dropping from approximately 35 seconds for a full refactorization to 1-3 seconds, depending on cut length. For a mesh with 3 million vertices we see a drop from 7 seconds to 100-600 milliseconds. We believe this is a situation where our method actually enables interactive editing.

*Comparison to [Herholz and Alexa 2018].* We compare our method for incorporating Dirichlet boundary conditions (Section 4.1) with the update algorithm presented in [Herholz and Alexa 2018]. We are able to achieve a significant speedup over this method because our method does not need to copy large portions of the original factorization. However, both methods are based on partial refactorization and have therefore a similar scaling behaviour with respect to the number of modified degrees of freedom. A key advantage of our method is that we can attractively add new seams in contrast to [Herholz and Alexa 2018], where each update has to start from the original factorization. Moreover, this method is limited to Dirichlet boundary conditions and is not capable of adding degrees of freedom to a factorization.

Comparison to [Davis and Hager 2005]. Cholmod has a dedicated implementation for 2-rank updates representing a symmetric addition or deletion of a row/column pair. Deleting such a pair replaces the row and column by unit vectors, similar to our algorithm. For very small updates this methods can outperform our approach, however, this is usually not in the realm of editing scenarios we are interested in. In contrast to our approach, this method scales linearly with the number of updates. Unfortunately the method cannot be easily parallelized, but a cache optimized implementation similar to [Davis and Hager 2000] could be possible, leading to a small performance boost. Our approach on the other hand can in many situations update several vertices at the cost of one, as they share a common path in the elimination tree. For the purpose of this comparison experiment, when adding degrees of freedom, we add empty rows and columns during the initial factorization. This is not useful in practice, as the position of these rows and columns will be generally unknown.

*Comparison to [Yeung et al. 2018].* Updates using this methods are targeted at mesh cutting in simulations featuring frequent, small updates. Therefore it comes as no surprise that this algorithm only performs well for small seam lengths. Because we do not sample many paths in this realm, the plots seem to indicate that we are always outperforming Schur based updates, which is only true for paths with more than about 20 vertices. Similar to Cholmod's 2-rank updates, the performance of this method scales at least linearly with the number of constraints *m* because solving the system requires the construction of the matrix (10), involving *m* backsubstitutions.

#### 6.2 Comparison to Iterative Methods

Iterative methods for the solution of linear systems are a viable alternative to direct methods like Cholesky factorization. Their main



Updating orbifold Tutte embeddings





Fig. 8. Comparison of two preconditioned conjugate gradient implementations to the solution of the linear system using our updates and refactorization, followed by forward and backward substitution. We remove degrees of freedom along a path for parameterization like in Section 6.1. For the iterative methods the reduced system  $A_{II}$  is solved explaining the slight slope. Updates and refactorization perform at least an order of magnitude faster.

advantage over direct methods is their scalability thanks to the low memory requirements; the system matrix does not even have to be represented explicitly. In the context of system updates there are two potential benefits: we can warm-start the solution process with a previous solution, and there is no factorization to be updated. For symmetric and positive definite problems (possibly after a small perturbation) as considered here, the preconditioned conjugate gradient method is one of the most popular and versatile approaches. However, the convergence rate depends on the specific problem and the preconditioner. We compare our update method and refactorization, both including forward and backward substitution, with two implementations of preconditioned conjugate gradients on the example of removing degrees of freedom, see Figure 8. The method implemented in Eigen [Guennebaud et al. 2010] uses the very basic Jacobi preconditioner, which is about an order of magnitude slower than Hypre [Falgout et al. 2006], a highly optimized state-of-the-art implementation employing an algebraic multigrid preconditioner. However, both methods are slower than Cholesky factorization for the problems we consider. We use a threshold of  $10^{-8}$  as the convergence criterion for both conjugate gradient implementations. We



Fig. 9. Posing ARMADILLO in an editing session requires imposing new boundary conditions whenever the handle region changes. Our solver does not have to update for all current boundary conditions every time but can add and remove them on the fly, resulting in better performance.

do not observe any substantial performance benefit from warmstarting the algorithms, which is most likely attributed to the fact that introducing cuts can result in vastly different parameterizations. However, there might be scenarios where the parameterization does not change substantially due to new constraints, which could benefit from warm starts.

## 6.3 Orbifold Tutte Embeddings

We evaluate the introduction of more general boundary conditions at the example of orbifold Tutte embeddings, as detailed in Section 5.2. We compare to the direct solution of (20) using the sparse LU decomposition implemented in UMFPACK [Davis 2004] in Figure 7. Our implementation uses the formulation as a Schur complement (21) in combination with an existing factorization of the system matrix **A**. For each new cut we update the factorization by introducing new vertices as described in Section 4.2. For the construction of **C** (21) and solution of the system (22) we employ dense linear algebra procedures from the Intel MKL library. With growing boundary length we see the benefit of our implementation decrease, which is due to the dense linear algebra operations; however, in all our experiments our formulation is commonly orders of magnitude faster or at least as efficient as the LU decomposition based approach, even on relatively large meshes.

#### 6.4 Mesh Deformation

In Figure 9 we show a short editing session posing the ARMADILLO model using as-rigid-as-possible mesh deformation [Sorkine and Alexa 2007]. This is the setting for which the update algorithm in [Herholz and Alexa 2018] is designed for. Our algorithm is still able to be faster in this example because we can iteratively add constraints for each step while the competing approach has to extract and update a Cholesky factor with respect to all active constraints in



Randomly removing degrees of freedom

Fig. 10. Removing random degrees of freedom can result in faster refactorization compared to updates, because the effective system dimension decreases.

every step. For larger meshes and distributed boundary conditions the difference is expected to be even more pronounced; however, our method can only provide substantial improvements over [Herholz and Alexa 2018] for frequent, incremental changes of boundary conditions. The total performance improvement for mesh deformation, especially for ARAP based methods, depends on the number of iterations and the amount of computation to be performed besides the factorization of a system matrix, specifically the computation of the local rotation matrices.

#### 6.5 Scalability

While our methods are able to efficiently update a substantial amount of vertices, they become less efficient when a very large number of degrees of freedom are modified. This effect is more prominent if nodes that are far apart in terms of the elimination tree are updated. When removing a large fraction of all degrees of freedom, refactorization becomes more efficient, because the reduced system  $A_{II}$  to be factorized is much smaller than the full system we have to update. To evaluate the behaviour of our method with respect to update size, we perform a stress test. For a mesh with n vertices we randomly choose a fraction of the vertices covering the range from 10% to 90% (see Figure 10). We observe that our method is competitive when up to 25% degrees of freedom are removed, even though the vertices are chosen at random. This is a very artificial scenario, as it is unlikely that such a large amount of degrees of freedom is ever removed all at once. However, if many degrees of freedom in a connected mesh region have to be removed, for example in mesh deformation, we can make use of the fact that the Cholesky factor for the affected vertices contains unit vectors, as described in Section 5.3. If an application indeed requires updating many random degrees of freedom, our method does not yield a benefit.

#### 7 CONCLUSION

We presented different variants of Cholesky factor updates specifically tailored to the demands of interactive mesh parameterization. We demonstrated that our updates scale with mesh size and outperform competing approaches. The three presented example applications are just a small sample; every algorithm requiring updates to linear boundary conditions can potentially benefit from our approach. While our method scales to relatively large updates, it does not provide a benefit over refactorization if more than about 25% of degrees of freedom are removed, as demonstrated in Section 6.5. Adding degrees of freedom, however, is always competitive with refactorization because the effective system dimension is not decreased.

The main difference of our algorithms to most other Cholesky updating approaches lies in the fact that we are not processing the update of each degree of freedom individually, but resort to partial refactorization, leading to drastically different scaling behaviour. Updating a small seam that cuts the mesh in a local region takes the same amount of time as cutting just a single vertex, because the set of columns of the Cholesky factor that have to be modified is usually almost the same for both cases.

*Future Work.* In the future we would like to explore more complex mesh modifications and their corresponding Cholesky updates. Interactively cutting meshes could also be an interesting application scenario. Approximate Cholesky update methods [Hecht et al. 2012] for this kind of tasks have to resort to refactorization if the mesh modifications become too complex. Our methods could replace these full refactorizations and form new algorithms combining approximate and exact Cholesky updates.

## ACKNOWLEDGMENTS

We thank Rohan Shawhney and Noam Aigerman for inspiring discussions. Jacques Lucke helped us tremendously with integrating our solver into Blender. We are grateful to the anonymous reviewers for their constructive comments. This work was partially supported by the Personalized Health and Related Technologies (PHRT) SwissHeart grant.

#### REFERENCES

- Noam Aigerman, Shahar Z. Kovalsky, and Yaron Lipman. 2017. Spherical Orbifold Tutte Embeddings. ACM Trans. Graph. 36, 4, Article 90 (July 2017), 13 pages. https: //doi.org/10.1145/3072959.3073615
- Mario Botsch and Olga Sorkine. 2008a. On Linear Variational Surface Deformation Methods. *IEEE Transactions on Visualization and Computer Graphics* 14, 1 (Jan 2008), 213–230. https://doi.org/10.1109/TVCG.2007.1054
- Mario Botsch and Olga Sorkine. 2008b. On linear variational surface deformation methods. IEEE Transactions on Visualization and Computer Graphics 14, 1 (2008), 213–230.
- Yanqing Chen, Timothy A. Davis, William W. Hager, and Sivasankaran Rajamanickam. 2008. Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate. ACM Trans. Math. Softw. 35, 3, Article 22 (Oct. 2008), 14 pages. https://doi.org/10.1145/1391989.1391995
- Jieyu Chu, Nafees Bin Zafar, and Xubo Yang. 2017. A Schur Complement Preconditioner for Scalable Parallel Fluid Simulation. ACM Trans. Graph. 36, 5, Article 163 (July 2017), 11 pages. https://doi.org/10.1145/3092818
- Timothy A. Davis. 2004. Algorithm 832: UMFPACK V4.3—an Unsymmetric-Pattern Multifrontal Method. ACM Trans. Math. Softw. 30, 2 (June 2004), 196–199. https: //doi.org/10.1145/992200.992206
- Timothy A. Davis. 2006. Direct Methods for Sparse Linear Systems. SIAM, Philadelphia, PA.
- Timothy A. Davis and W. Hager. 1999. Modifying a Sparse Cholesky Factorization. SIAM J. Matrix Anal. Appl. 20, 3 (1999), 606–627. https://doi.org/10.1137/ S0895479897321076
- Timothy A. Davis and William W. Hager. 2000. Multiple-Rank Modifications of a Sparse Cholesky Factorization. SIAM J. Matrix Anal. Appl. 22, 4 (July 2000), 997–1013. https://doi.org/10.1137/S0895479899357346
- 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. https://doi.org/10. 1137/S089547980343641X
- 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

ACM Trans. Graph., Vol. 39, No. 6, Article 202. Publication date: December 2020.

#### 202:14 • Philipp Herholz and Olga Sorkine-Hornung

- 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.
- Robert D. Falgout, Jim E. Jones, and Ulrike Meier Yang. 2006. The Design and Implementation of hypre, a Library of Parallel High Performance Preconditioners. In *Numerical Solution of Partial Differential Equations on Parallel Computers*, Are Magnus Bruaset and Aslak Tveito (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 267–294.
- Phillip E. Gill, Gene H. Golub, Walter A. Murray, and Michael A. Saunders. 1972. Methods for Modifying Matrix Factorizations. Technical Report. Stanford, CA, USA.
- Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. http://eigen.tuxfamily.org.
- 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 31, 5 (oct 2012), 123:1–13. https://doi.org/10.1145/2231816.2231821
- Philipp Herholz and Marc Alexa. 2018. Factor Once: Reusing Cholesky Factorizations on Sub-Meshes. ACM Transaction on Graphics (Proc. of Siggraph Asia) 37, 6 (2018), 9. https://doi.org/10.1145/3272127.3275107
- Philipp Herholz and Marc Alexa. 2019. Efficient Computation of Smoothed Exponential Maps. Computer Graphics Forum 38, 6 (2019), 79–90. https://doi.org/10.1111/cgf. 13607
- Philipp Herholz, Timothy A. Davis, and Marc Alexa. 2017. Localized Solutions of Sparse Linear Systems for Geometry Processing. ACM Trans. Graph. 36, 6, Article 183 (Nov. 2017), 8 pages. https://doi.org/10.1145/3130800.3130849
- Intel. 2009. Intel Math Kernel Library. Reference Manual.
- George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. SIAM J. Sci. Comput. 20, 1 (1998), 359–392. https: //doi.org/10.1137/S1064827595287997
- Haixiang Liu, Nathan Mitchell, Mridul Aanjaneya, and Eftychios Sifakis. 2016. A Scalable Schur-Complement Fluids Solver for Heterogeneous Compute Platforms. ACM Trans. Graph. 35, 6, Article 201 (Nov. 2016), 12 pages. https://doi.org/10.1145/ 2980179.2982430
- Haggai Maron, Meirav Galun, Noam Aigerman, Miri Trope, Nadav Dym, Ersin Yumer, Vladimir G. Kim, and Yaron Lipman. 2017. Convolutional Neural Networks on Surfaces via Seamless Toric Covers. ACM Trans. Graph. 36, 4, Article 71 (July 2017), 10 pages. https://doi.org/10.1145/3072959.3073616
- Matthias Müller and Markus Gross. 2004. Interactive Virtual Materials. In Proceedings of Graphics Interface 2004 (London, Ontario, Canada) (GI '04). Canadian Human-Computer Communications Society, Waterloo, CAN, 239–246.
- Jorge Nocedal and Stephen Wright. 2006. Numerical optimization. Springer Science & Business Media.
- Rohan Sawhney and Keenan Crane. 2017. Boundary First Flattening. ACM Trans. Graph. 37, 1, Article 5 (Dec. 2017), 14 pages. https://doi.org/10.1145/3132705
- Olga Sorkine and Marc Alexa. 2007. As-rigid-as-possible Surface Modeling. In Proceedings of the Fifth Eurographics Symposium on Geometry Processing (Barcelona, Spain) (SGP '07). Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 109–116.
- Olga Sorkine, Daniel Cohen-Or, Dror Irony, and Sivan Toledo. 2005. Geometry-Aware Bases for Shape Approximation. *IEEE Transactions on Visualization and Computer Graphics* 11, 2 (2005), 171–180.
- Yu-Hong Yeung, Jessica Crouch, and Alex Pothen. 2016. Interactively Cutting and Constraining Vertices in Meshes Using Augmented Matrices. ACM Trans. Graph. 35, 2, Article 18 (Feb. 2016), 17 pages. https://doi.org/10.1145/2856317
- Yu-Hong Yeung, Alex Pothen, and Jessica Crouch. 2018. AMPS: A Real-time Mesh Cutting Algorithm for Surgical Simulations. arXiv:1811.00328 [cs.CE]

#### A EXTRACTING L<sub>II</sub>

In Section 4.1 we make use of the fact that the factorization of

$$\mathbf{A}_{II} = \begin{pmatrix} \mathbf{A}_1 & \mathbf{A}_2 \\ \mathbf{A}_3 & \mathbf{A}_4 \end{pmatrix} \tag{24}$$

can be easily extracted from the factorization of

$$\mathbf{A}^{B} = \begin{pmatrix} \mathbf{A}_{1} & \mathbf{0} & \mathbf{A}_{2} \\ \hline \mathbf{0} & 1 & \mathbf{0} \\ \hline \mathbf{A}_{3} & \mathbf{0} & \mathbf{A}_{4} \end{pmatrix}.$$
 (25)

Here we demonstrate that this can be easily achieved by extracting an appropriate submatrix in the special case of removing the degree of freedom k, i.e.,  $B = \{k\}$ .

ACM Trans. Graph., Vol. 39, No. 6, Article 202. Publication date: December 2020.

By construction the factorization of  $A^B$  has the form

$$\mathbf{L}^{B} = \begin{pmatrix} \mathbf{L}_{1}^{B} & \mathbf{0} & \mathbf{0} \\ \hline \mathbf{0} & 1 & \mathbf{0} \\ \hline \mathbf{L}_{3}^{B} & \mathbf{0} & \mathbf{L}_{4}^{B} \end{pmatrix},$$
(26)

as can be verified by explicitly forming the product  $\mathbf{L}^{B} \left( \mathbf{L}^{B} \right)^{\top}$ . By forming this product it also becomes apparent that

$$\mathbf{A}_{II} = \begin{pmatrix} \mathbf{A}_1 & \mathbf{A}_2 \\ \mathbf{A}_3 & \mathbf{A}_4 \end{pmatrix} = \begin{pmatrix} \mathbf{L}_1^B & \mathbf{0} \\ \mathbf{L}_3^B & \mathbf{L}_4^B \end{pmatrix} \begin{pmatrix} \mathbf{L}_1^B & \mathbf{0} \\ \mathbf{L}_3^B & \mathbf{L}_4^B \end{pmatrix}^{\top}$$
(27)
$$= \mathbf{L}_{II}^B \begin{pmatrix} \mathbf{L}_{II}^B \end{pmatrix}^{\top}$$
(28)

holds, which is the statement in Equation (16). The construction extends to general sets *B* by applying this argument recursively.