Factor Once: Reusing Cholesky Factorizations on Sub-Meshes

PHILIPP HERHOLZ, TU Berlin, Germany
MARC ALEXA, TU Berlin, Germany

A common operation in geometry processing is solving symmetric and positive semi-definite systems on a subset of a mesh, with conditions for the vertices at the boundary of the region. This is commonly done by setting up the linear system for the sub-mesh, factorizing the system (potentially applying preordering to improve sparseness of the factors), and then solving by back-substitution. This approach suffers from a comparably high setup cost for each local operation. We propose to reuse factorizations defined on the full mesh to solve linear problems on sub-meshes. We show how an update on sparse matrices can be performed in a particularly efficient way to obtain the factorization of the operator on a sun-mesh significantly outperforming general factor updates and complete refactorization. We analyze the resulting speedup for a variety of situations and demonstrate that our method outperforms factorization of a new matrix by a factor of up to 10 while never being slower in our experiments.

CCS Concepts: • **Computing methodologies** → **Mesh geometry models**; • **Mathematics of computing** → *Computations on matrices*; *Solvers*;

Additional Key Words and Phrases: geometry processing, matrix factorizations

ACM Reference Format:
Philipp Herholz and Marc Alexa. 2018. Factor Once: Reusing Cholesky Factorizations on Sub-Meshes. *ACM Trans. Graph.* 37, 6, Article 230 (November 2018), 9 pages. https://doi.org/10.1145/3272127.3275107

1 INTRODUCTION

Many interactive modeling operations require solving a linear system on a subset of a mesh. As a guiding example for this type of operation we take *mesh deformation*: the user marks a region of interest and a set of handle vertices; moving the handle vertices will then affect the vertex positions in the region of interest, while the vertices outside the region of interest stay fixed. Many approaches for this operation are based on minimizing a quadratic (or nonlinear) deformation energy subject to position constraints of all fixed vertices [Bouaziz et al. 2012; Sorkine and Alexa 2007; Sorkine et al. 2004]:

$$\begin{aligned} \underset{\mathbf{x}\in\mathbb{R}^{n\times 3}}{\operatorname{argmin}} \quad & \operatorname{Tr}\left(\mathbf{x}^\mathsf{T}\mathbf{A}\mathbf{x}\right) \\ \text{s.t. } & \mathbf{x}_i = \mathbf{x}_i^c \text{ for } i \in \mathcal{B} \end{aligned} \tag{1}$$

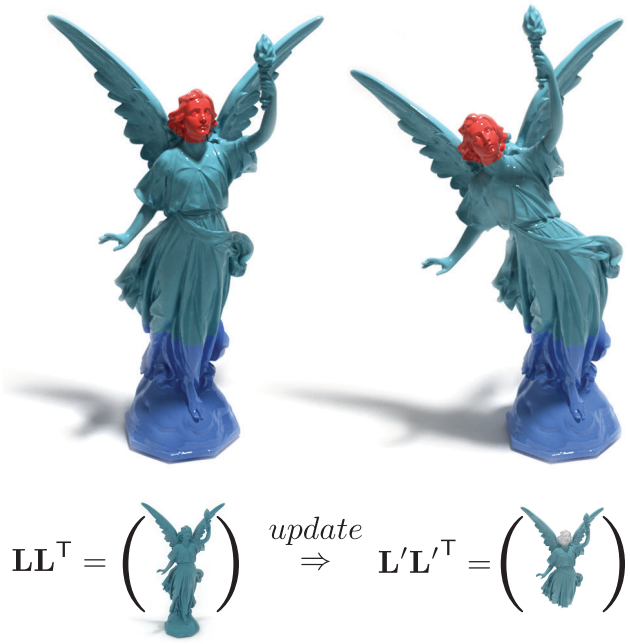


Fig. 1. Deforming a mesh based on selecting a region of interest (green) and a handle (red) commonly requires solving a sparse linear system. This is typically done using the Cholesky factorization. The computation time for the factorization depends on the number of vertices in the region of interest, and may prohibit interactive use for large meshes. We propose a method that exploits the factorization of the operator $\mathbf{L}$ on the whole mesh for computing the new factorization of $\mathbf{L}'$ on the sub-mesh, instead of computing it from scratch. For the depicted mesh with one million vertices our update algorithm computes a new factorization in 0.47 seconds whereas the factorization of the desired system matrix for the region of interest using Cholmod takes 3.4 seconds.

Note that these are actually three separate optimization problems, one for each dimension. Here the index set $\mathcal{B}$ contains the constrained vertices. The matrix $\mathbf{A} \in \mathbb{R}^{\mathbf{n}\times\mathbf{n}}$ is symmetric and positive semi-definite, in many cases the mesh Laplacian [Botsch and Sorkine 2008; Meyer et al. 2003; Pinkall and Polthier 1993]. The order of vertices in $\mathbf{x}$ is arranged so that the set of unconstrained vertices $\mathcal{I}$ and the fixed vertices $\mathcal{B}$ form blocks:

$$\mathbf{x} = \begin{pmatrix}\mathbf{x}_\mathcal{I}\\ \mathbf{x}_\mathcal{B}\end{pmatrix}, \qquad \mathbf{A} = \begin{pmatrix}\mathbf{A}_{\mathcal{I}\mathcal{I}} & \mathbf{A}_{\mathcal{I}\mathcal{B}}\\ \mathbf{A}_{\mathcal{B}\mathcal{I}} & \mathbf{A}_{\mathcal{B}\mathcal{B}}\end{pmatrix}. \tag{2}$$

The minimization can be performed by substituting $\mathbf{x}_\mathcal{B}$ by the constrained vertex positions $\mathbf{x}_\mathcal{B}^c$ and solving the resulting system

$$\begin{pmatrix}\mathbf{A}_{\mathcal{I}\mathcal{I}} & \mathbf{A}_{\mathcal{I}\mathcal{B}}\end{pmatrix}\begin{pmatrix}\mathbf{x}_\mathcal{I}\\ \mathbf{x}_\mathcal{B}^c\end{pmatrix} = \mathbf{0} \quad \Rightarrow \quad \mathbf{A}_{\mathcal{I}\mathcal{I}}\mathbf{x}_\mathcal{I} = -\mathbf{A}_{\mathcal{I}\mathcal{B}}\mathbf{x}_\mathcal{B}^c. \tag{3}$$

Authors' addresses: Philipp Herholz, philipp.herholz@tu-berlin.de, TU Berlin, Germany; Marc Alexa, marc.alexa@tu-berlin.de, TU Berlin, Electrical Engineering & Computer Science, Marchstr. 23, Sekretariat MAR 6-6, Berlin, 10587, Germany.

0730-0301/2018/11-ART230
https://doi.org/10.1145/3272127.3275107

Because $\mathbf{A}$ is SPD, so is $\mathbf{A}_{II}$, and the system is commonly solved by computing the *Cholesky* factorization $\mathbf{A}_{II} = \mathbf{L}\mathbf{L}^\mathsf{T}$ followed by performing back- and forward substitution. This is particularly effective if the system is sparse. The sparsity of the resulting factor can be further improved by reordering. We provide background on sparse Cholesky factorization including pre-ordering in Section 3.

While solving a sparse system given its Cholesky factorization is very fast, the necessary pre-ordering and factorization steps are expensive computations. This can lead to problems for operations that are intended for real-time interaction. *Our central idea is to re-use the ordering and factorization of a linear system defined on the complete mesh to construct the factorization of a linear operator on a sub-mesh.* The first idea is straightforward: The reordering that is necessary for the computation of Cholesky factors can be reused for linear systems on sub-meshes by just keeping the vertex indices in the same relative order. This avoids the time-consuming ordering step for the local problem.

More significantly, we observe that the factors for the local problem can be derived from the factors of the global problem by a series of rank-one updates (Section 4). This updates can be efficiently performed by copying data from the global factor and running a Cholesky factorization algorithm only on a subset of columns.

In Section 5, we first show that the non-zero structure of the desired local Cholesky factor can be easily deferred from the global factor. This observation is non-trivial as it is not generally true for arbitrary sparse factorizations. Based on this observation we develop an update strategy that is significantly faster than general purpose rank-one updates [Davis and Hager 2000].

We compare the efficiency of our approach with complete refactorization for a range of different meshes and differently sized sub-meshes. Reusing the ordering already provides a speedup for solving local problems. Our update strategy results in significant further progress. Compared to the common call to a matrix library, which would perform pre-ordering and factorization, our update strategy is significantly faster. We present detailed results in Section 7. In general, every algorithm that requires solving several constrained local problems on a mesh can benefit from our approach. This includes parameterization [Desbrun et al. 2002; Mullen et al. 2008], local mesh filtering [Desbrun et al. 1999] and constrained heat diffusion [Crane et al. 2013].

2 RELATED WORK

Hecht at al. [2012] suggest a method which is similar in spirit to ours. They apply partial refactorization to Cholesky factors in a nonlinear finite element simulation of elastic objects. During the simulation the problem is repeatedly linearized, leading to sparse linear systems. The main observation is that in many cases the simulated object and therefore the linear system only changes locally. These changes in turn only affect certain parts of the Cholesky factor, which can be partially refactored. This leads to an approximation that is good enough for small time steps. The system is completely factored only sporadically. Like our approach, this method leverages the block structure of the Cholesky factor of a matrix that has been rearranged using nested dissection reordering. The authors report a two to three fold speedup compared to full refactorization at every frame.

In a similar vein, Yeung et al. [2016] consider updates to a linear FEM system when the structure of a simulated mesh is changing, e.g. during a surgical simulation. They augment the matrix with additional rows and columns to account for new constraints effectively modifying columns of the original matrix. The new system can be solved by using the original factorized matrix in conjunction with an iterative scheme. This method is very well suited for continuous changes in mesh topology. After a certain number of iterations, however, a new factorization has to be computed. Our approach on the other hand is targeted towards incorporating many constraints at once and would not be competitive if changes have to be made one at a time. In that sense this method is complementary to ours.

Herholz et al. [2017] demonstrate how to exploit the structure of the Cholesky factorization to efficiently solve linear systems defined on a mesh when only a subset of the solution is required. The Cholesky factor itself remains unchanged. With this technique it is possible to solve local linear problems as long as they can be formulated in terms of modifications of the right hand side only. This includes Neumann boundary conditions when working with mesh Laplacians. Dirichlet boundary conditions, i.e. constraining certain vertex values, requires changes in the system matrix and, consequently, the factorization.

Modifying a given Cholesky factorization has been an active research area for many years. The central observation is that the factorization of a given matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ can be modified to become the factorization of a matrix $\mathbf{A}' \in \mathbb{R}^{n\times n}$ with much less effort than computing the new factorization as long as $\mathbf{A}'$ has been obtained from $\mathbf{A}$ by certain simple modifications. Not all modifications are allowed since $\mathbf{A}'$ has to stay symmetric and SPD. One class of modifications that necessarily respects these properties are rank-one updates:

$$\mathbf{A}' = \mathbf{L}'\mathbf{L}'^\mathsf{T} = \mathbf{A} + \mathbf{v}\mathbf{v}^\mathsf{T} = \mathbf{L}\mathbf{L}^\mathsf{T} + \mathbf{v}\mathbf{v}^\mathsf{T}. \tag{4}$$

with $\mathbf{v} \in \mathbb{R}^n$. This type of modification appears quite naturally, e.g. when adding an observation in a regression model that has to be solved using least squares. Davis et al. [2000] provide an efficient implementation of this method for sparse matrices. Special care has to be taken when the non-zero structure of the factor changes, which can generally happen. The method is very efficient if updates are performed one at a time. If several updates are necessary at once, i.e. $\mathbf{A} + \mathbf{V}\mathbf{V}^\mathsf{T}$ with $\mathbf{V} \in \mathbb{R}^{n\times k}$, the update can be performed slightly faster than consecutive rank-one updates, however, the amount of speedup is limited to about a factor of two because updates can only be efficiently processed in small batches [Davis and Hager 2009]. If the update is of very low rank (i.e. k is very small) this method might actually be faster than our approach. However, this assumption is not true for almost all use cases considered here. In Section 7 we compare the performance of our approach to multi-rank updates as implemented in Cholmod.

It is one of our central observations that in our particular scenario, where only very specific updates have to take place, the non-zero structure is unaffected by the updates. We show how this simplifies the operation and allows for much faster multi-rank updates.

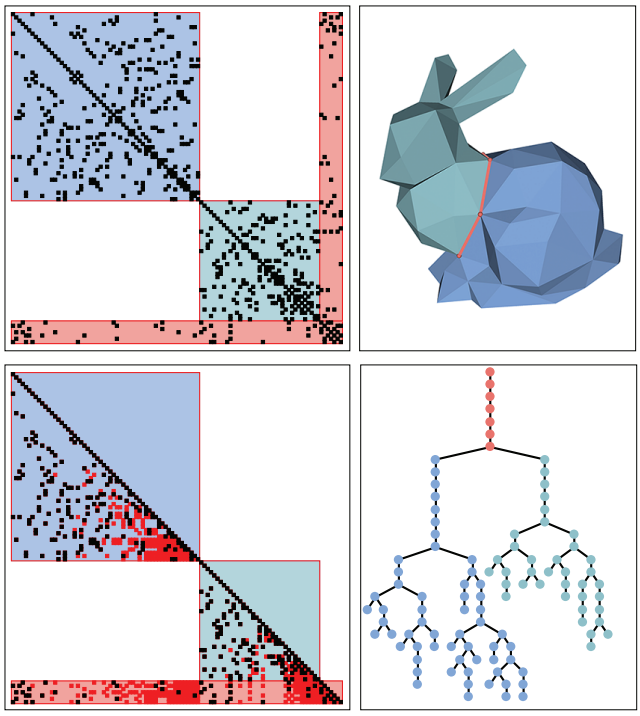


Fig. 2. Nested dissection reordering hierarchically divides the mesh into two balanced parts separated by a small divider (top right). The mesh Laplacian will then exhibit a block structure (top left). These blocks are preserved in the Cholesky factor (bottom left) which has additional non-zero entries only inside these blocks (red pixels). The block structure yields a fairly balanced elimination tree (bottom right).

3 BACKGROUND: SPARSE CHOLESKY FACTORIZATION

Our approach is based on the Cholesky factorization of the linear system. We make use of several properties of sparse Cholesky decompositions, its common data structures, and the basic factorization algorithm. While this material is covered in the literature, we provide the necessary basics as far as they are required for understanding our algorithm so that the paper is self-contained. The reader familiar with the details of sparse Cholesky factorizations may skip this section.

The factorization represents a symmetric and positive semi-definite matrix $\mathbf{A}$ as the product of a lower triangular matrix and its transpose

$$\mathbf{A} = \mathbf{L}\mathbf{L}^\mathsf{T} \quad \text{with } \mathbf{L} \in \mathbb{R}^{n \times n}. \tag{5}$$

The factorization can be used to efficiently solve a linear system of the form $\mathbf{A}\mathbf{x} = \mathbf{b}$. To this end the substitute vector

$$\mathbf{y} = \mathbf{L}^\mathsf{T}\mathbf{x} \tag{6}$$

is defined which yields

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{7}$$

$$\Leftrightarrow \quad \mathbf{L}\mathbf{L}^\mathsf{T}\mathbf{x} = \mathbf{b} \tag{8}$$

$$\Leftrightarrow \quad \mathbf{L}\mathbf{y} = \mathbf{b} \tag{9}$$

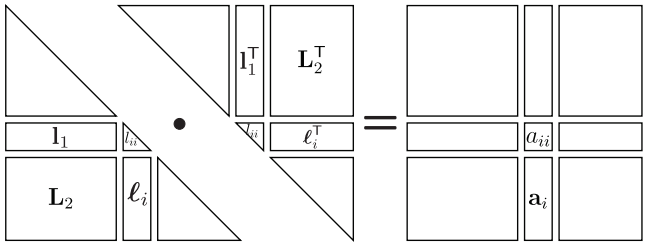


Fig. 3. Schematic illustration of the left-looking Cholesky algorithm. The algorithm subsequently computes the columns $\boldsymbol{\ell}_i$ by using already computed values to its left in the factor.

The vector $\mathbf{y}$ can then be determined by solving the triangular system (9) (forward substitution). To obtain the final solution vector, the triangular system (6) is solved (back substitution).

In our setting we are concerned with sparse matrices since it would be infeasible, both in terms of memory and computation, to work with dense factors. It is one of the favorable properties of Cholesky factorizations that a sparse system matrix $\mathbf{A}$ leads to a sparse Cholesky factorization in the sense that block structures are preserved. The system matrix $\mathbf{A}$ in Figure 2 (upper left) is structured into three blocks. The Cholesky factor of $\mathbf{A}$ (lower left) will have additional entries (red pixels) but they will always fall within these blocks. More specifically, if the first non-zero element in the i-th row of $\mathbf{A}$ is the element a_{ij} (this means $a_{ik} = 0$ for all $k < j$) it is guaranteed that the same is true for the Cholesky factor i.e. $l_{ik} = 0$ for all $k < j$. This property can only be helpful if such a block structure exists, however, this can usually be achieved by reordering $\mathbf{A}$.

Nested dissection reordering. The idea of nested dissection reordering is to recursively subdivide the mesh into two parts of roughly the same number of vertices, separated only by a small set of vertices (divider). Figure 2 illustrates one step of this procedure. By sorting the vertices such that vertices in the two parts are sorted subsequently followed by the dividing vertices one achieves the desired block structure. The blocks can then recursively be subdivided in the same manner. Since meshes in geometry processing are usually embedded and edges connect spatially close vertices, it is usually possible to find good dividers on Laplacian-type systems. This makes nested dissection reordering very effective for problems in geometry processing.

Left-looking Cholesky factorization. In principle there are many ways to compute Cholesky factorizations. A popular method is the so called left-looking algorithm that computes the factor $\mathbf{L}$ column by column using the already computed columns to the left – hence the name. Assume the first $i-1$ columns of the factor have already been computed and one wants to determine the i-th column $(l_{ii}, \boldsymbol{\ell}_i)$ (see Figure 3). For the diagonal value l_{ii} we get

$$\mathbf{l}_1 * \mathbf{l}_1^T + l_{ii}^2 = a_{ii} \tag{10}$$

$$\Rightarrow \quad l_{ii} = \sqrt{a_{ii} - \mathbf{l}_1 * \mathbf{l}_1^T}. \tag{11}$$

Expressing $\mathbf{a}_i$ with the matrix product on the left in Figure 3 provides a relationship for $\boldsymbol{\ell}_i$:

$$\mathbf{L}_2 * \mathbf{l}_1^T + l_{ii} * \boldsymbol{\ell}_i = \mathbf{a}_i \tag{12}$$

$$\Rightarrow \quad \boldsymbol{\ell}_i = \frac{1}{l_{ii}} \left(\mathbf{a}_i - \mathbf{L}_2 * \mathbf{l}_1^T \right). \tag{13}$$

If the matrix is positive semi-definite, the square root in the expression for l_{ii} is real-valued. In many computer graphics application the matrix might actually not be positive definite because of a rank deficit. Due to numeric errors the square root might produce complex values. This can be prevented by regularizing the system matrix effectively adding a small multiple of the identity matrix to the operator. Most importantly for our setting, this algorithm can be efficiently implemented for sparse matrices. Equation (13) also demonstrates why the block structure of $\mathbf{A}$ is preserved. If the k-th row of $\mathbf{L}$ does not contain a non-zero until column i and the same is true for $\mathbf{A}$, the value of $l_{k,i}$ will also be zero.

Sparse matrix representation. Sparse matrices can be represented in several ways with advantages for different access patterns or modifications. For Cholesky factors and system matrices we use the compressed column format which maintains three arrays: one that holds all non-zero values in column major order, a second one of the same size that stores the corresponding row indices and a smaller array that contains pointers to the start of each column in the row and value array respectively. It is generally very fast to visit all non-zeros per column. To traverse one matrix row is much less efficient. Alternatively the matrix can be represented in a compressed row major order by interchanging the role of rows and columns. All operations performed during the sparse Cholesky factorization, as described in [Davis 2006], are carefully designed with the compressed column format in mind.

Elimination Tree. Considering the left-looking Cholesky factorization algorithm one can see that the values of the i-th column of the Cholesky factor depend only on a sparse set of columns to the left because $\boldsymbol{\ell}_i$ is sparse (see equation 13). Knowing what columns depend on each other is the key to our efficient factor update: Changing a set of columns in $\mathbf{A}$ only affects columns that depend on them in the Cholesky factor.

The elimination tree [Liu 1990] (Figure 2, lower right) provides a very efficient way of identifying the column dependencies. Each node in this tree represents a column of the Cholesky factor. To find the parent of the i-th column, the first off-diagonal non-zero in this column is found. The row index of this value determines the parent of i. If no such value exists the i-th node represents the root of the tree. For meshes with more than one connected component multiple roots exist and we actually have an elimination forest. We do not consider this case here although it is a straight forward extension.

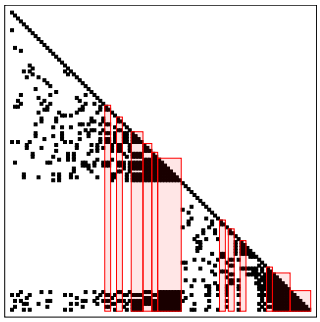
The crucial property of the elimination tree is that one can identify all columns that depend on a specific column i by traversing the tree up to its root starting at node i and collecting all visited nodes. A proof of this fact can be found in [Liu 1990].

This makes an update quite efficient and also illustrates why changing a column usually does not affect too many others: when modifying a column in the blue block of Figure 2 (lower left) all columns in the green block remain unaffected because they reside on a separate sub-tree.

Sparse Cholesky. The left-looking Cholesky factorization can be efficiently implemented for column oriented sparse matrices. The computation proceeds in two phases. The symbolic phase builds the elimination tree and uses it to analyze the non-zero pattern of the Cholesky factor. During the numeric phase the actual values are computed and stored in the prepared sparse matrix. Running the left-looking Cholesky factorization proceeds by constructing $\mathbf{L}$ column by column. The computation is dominated by the evaluation of equation (13) which involves the multiplication of the block $\mathbf{L}_2$ with the row vector $\boldsymbol{\ell}_1$. Due to the column oriented sparse matrix format we do not have an efficient way to access the non-zero pattern of a row. In our setting, however, we scan the rows $\boldsymbol{\ell}_i$ of the matrix one after another which enables us to use a simple pointer per column that is incremented whenever a value in the current row is accessed. With this pointer one has access to the value of a column in that row. However, this technique is only beneficial if we know the non-zero pattern of $\boldsymbol{\ell}_i$ because every column pointer needs to be checked for every row otherwise. This non-zero pattern can be found quite efficiently by traversing the elimination tree, starting from all non-zeros in $\mathbf{a}_i$ (see [Davis 2006]).

In our implementation we take a slightly different route. During the symbolic phase we not only compute the combinatorial structure of the matrix in column oriented form but also in row oriented form. This gives us easy access to each column in a row and makes row access more efficient – at the cost of storage. Moreover, this row information is needed during our partial update procedure.

Supernodal Cholesky. When inspecting the elimination tree one can identify strings of nodes that have only one child. Since traversing the tree determines the non-zero structure of each column it can be shown that columns of the Cholesky factor in such a string have two properties: They are immediate neighbors and they have, apart from diagonal entries, the same non-zero pattern. It is therefore beneficial to aggregate these columns into so called *supernodes* to store the common row information only once and keep the values of such a group of columns as a dense matrix. The inset highlights supernodes of a Cholesky factor in red. Supernodes are not only convenient for storage, they can also be exploited during computation, leveraging fast dense matrix kernels (BLAS / LAPACK). Our implementation based on Cholmod makes extensive use of this fact. However, for clarity of exposition we describe our algorithm in terms of simple columns and hide the implementation details of the supernodal version of the algorithm. For more detail see the book by Davis [2006].



4 APPROACH

Our goal is to use the factorization $\mathbf{A} = \mathbf{L}\mathbf{L}^\mathsf{T}$ of the (global) matrix for a local problem on the same mesh. Note that $\mathbf{A}$ has been reordered to optimize the sparsity of the factor $\mathbf{L}$; changing the order of $\mathbf{A}$ would require recomputing the factorization. This means the common approach of ordering the system such that the unconstrained and constrained vertices form blocks (as described in the introduction) is infeasible.

Nonetheless, $\mathbf{A}_{\mathcal{I}\mathcal{I}}$ and $\mathbf{A}_{\mathcal{I}\mathcal{B}}$ can still be *expressed* in terms of sub-blocks of $\mathbf{L}$. Let a_{ij} be an arbitrary matrix entry of $\mathbf{A}$. This entry can be represented by the inner product $a_{ij} = \mathbf{l}_i^\mathsf{T}\mathbf{l}_j$ of two rows of the Cholesky factor $\mathbf{L}$. These two rows can be split into columns for $\mathcal{B}$ and $\mathcal{I}$, which can be interpreted as computing the inner product in a different order:

$$a_{ij} = \mathbf{l}_i^\mathsf{T}\mathbf{l}_j = \sum_k l_{ik}l_{jk} = \sum_{k\in\mathcal{I}} l_{ik}l_{jk} + \sum_{k\in\mathcal{B}} l_{ik}l_{jk} = \mathbf{l}_{i\mathcal{I}}^\mathsf{T}\mathbf{l}_{j\mathcal{I}} + \mathbf{l}_{i\mathcal{B}}^\mathsf{T}\mathbf{l}_{j\mathcal{B}}. \tag{14}$$

Putting this together for all entries a_{ij} of $\mathbf{A}_{\mathcal{I}\mathcal{I}}$ and $\mathbf{A}_{\mathcal{I}\mathcal{B}}$ yields:

$$\mathbf{A}_{\mathcal{I}\mathcal{I}} = \mathbf{L}_{\mathcal{I}\mathcal{I}}\mathbf{L}_{\mathcal{I}\mathcal{I}}^\mathsf{T} + \mathbf{L}_{\mathcal{I}\mathcal{B}}\mathbf{L}_{\mathcal{I}\mathcal{B}}^\mathsf{T} \tag{15}$$

$$\mathbf{A}_{\mathcal{I}\mathcal{B}} = \mathbf{L}_{\mathcal{I}\mathcal{I}}\mathbf{L}_{\mathcal{B}\mathcal{I}}^\mathsf{T} + \mathbf{L}_{\mathcal{I}\mathcal{B}}\mathbf{L}_{\mathcal{B}\mathcal{B}}^\mathsf{T}. \tag{16}$$

Note that if the system was ordered such that the indices in $\mathcal{I}$ came first, the zero structure of $\mathbf{L}$ would imply $\mathbf{L}_{\mathcal{I}\mathcal{B}} = \mathbf{0}$ and $\mathbf{L}_{\mathcal{I}\mathcal{I}}$ would be sufficient to solve the local problem. In general, however, this is not the case and Eq. (15) is the key to generate the desired factor: we are updating the Cholesky factor $\mathbf{L}_{\mathcal{I}\mathcal{I}}$ to become the factorization of $\mathbf{A}_{\mathcal{I}\mathcal{I}}$.

It has been observed [Davis and Hager 1999] that a low-rank update will only affect a small set of columns. Instead of using the general purpose rank-one update procedure for sparse Cholesky factorizations, which is prohibitively slow when changing a larger set of columns, we identify the set of columns that will change and rerun the factorization algorithm only on these columns, while simply copying columns that remain unchanged from the matrix $\mathbf{L}_{\mathcal{I}\mathcal{I}}$.

5 SPARSE UPDATES

Our central observation is that in order to compute the Cholesky factor of a submatrix $\mathbf{A}_{\mathcal{I}\mathcal{I}}$ the submatrix of the factor $\mathbf{L}_{\mathcal{I}\mathcal{I}}$ has to be modified by means of a low-rank update, affecting only a small number of columns in most cases. Moreover, the set of these columns can be determined very efficiently. Furthermore the update leaves the sparsity structure of the sub factor unaffected, allowing us to just copy the data and refactor only those columns that need to be updated.

Sparsity structure. Changing the non-zero structure of a sparse matrix can be computationally challenging since inserting values in the continuous row index and value arrays potentially involves reallocation and shifting of a lot of entries. It is therefore beneficial in our situation that the non-zero structure of the factor matrix $\mathbf{L}_{\mathcal{I}\mathcal{I}}$ does not change after it has been extracted from $\mathbf{L}$. Note that a non-zero in a sparse matrix always refers to a structural non-zero, that is, a value that is explicitly represented in the matrix – it may contain the numerical value 0. To see that the structure of the matrix $\mathbf{L}_{\mathcal{I}\mathcal{I}}$ does not have to be changed when performing the update we have to show that the same structure can represent the Cholesky factorization of $\mathbf{A}_{\mathcal{I}\mathcal{I}}$.

Proof: Consider an element a_{ij} of $\mathbf{A}$ with $i, j \in \mathcal{I}$. In order to represent this element the value

$$\left(\mathbf{L}_{\mathcal{I}\mathcal{I}}\mathbf{L}_{\mathcal{I}\mathcal{I}}^\mathsf{T}\right)_{ij} = \mathbf{l}_{i\mathcal{I}}\mathbf{l}_{j\mathcal{I}}^\mathsf{T} \tag{17}$$

can not be structurally zero where $\mathbf{l}_{i\mathcal{I}}$ and $\mathbf{l}_{j\mathcal{I}}$ are rows in $\mathbf{L}_{\mathcal{I}\mathcal{I}}$. Assume $i \geq j$ without loss of generality. Since the non-zero structure of Cholesky factors only contain additional non-zeros as compared to the lower triangular part of the system matrix $\mathbf{A}$, we know that all non-zeros in $\mathbf{A}_{\mathcal{I}\mathcal{I}}$ are also non-zero in $\mathbf{L}_{\mathcal{I}\mathcal{I}}$. This means that $l_{i,j}$ and $l_{j,j}$ are both non-zero and it follows that the inner product in (17) does not vanish in general. □

Due to this fact, $\mathbf{A}_{\mathcal{I}\mathcal{I}}$ can indeed be represented by a Cholesky factor with the same sparsity structure as $\mathbf{L}_{\mathcal{I}\mathcal{I}}$. This saves us the need to analyze the sparsity structure of $\mathbf{L}'$ based on its elimination tree. However, it is possible that the updated Cholesky factors explicitly store elements that are structurally zero as an explicit zero. In section 7 we show that these values are actually quite rare.

Update Algorithm. Our algorithm proceeds as depicted in Figure 4. Once the sub-mesh consisting of vertices indexed by $\mathcal{I}$ is selected, the submatrix consisting of rows and columns in $\mathcal{I}$ is extracted from $\mathbf{L}$. The update

$$\mathbf{A}_{\mathcal{I}\mathcal{I}} = \mathbf{L}'\mathbf{L}'^\mathsf{T} = \mathbf{L}_{\mathcal{I}\mathcal{I}}\mathbf{L}_{\mathcal{I}\mathcal{I}}^\mathsf{T} + \mathbf{L}_{\mathcal{I}\mathcal{B}}\mathbf{L}_{\mathcal{I}\mathcal{B}}^\mathsf{T} \tag{18}$$

can be thought of as a series of rank one updates of the form

$$\mathbf{L}'_k\mathbf{L}'^\mathsf{T}_k = \mathbf{L}_{k-1}\mathbf{L}'^\mathsf{T}_{k-1} + \boldsymbol{\ell}_{\mathcal{I}b_k}\boldsymbol{\ell}_{\mathcal{I}b_k}^\mathsf{T} \tag{19}$$

where $k > 0$ and $\boldsymbol{\ell}_{\mathcal{I}b_k}$ is a column in $\mathbf{L}_{\mathcal{I}\mathcal{B}}$ and $\mathbf{L}'_0 = \mathbf{L}_{\mathcal{I}\mathcal{I}}$. The columns in which $\mathbf{L}'_k$ and $\mathbf{L}'_{k-1}$ differ can be determined efficiently by finding the first off-diagonal non-zero in $\boldsymbol{\ell}_{\mathcal{I}b_k}$. The row index of this value is then used to traverse the elimination tree of $\mathbf{L}_{\mathcal{I}\mathcal{I}}$ upwards to the root. The set of nodes that are discovered along the way represent the set of columns that will change due to the update. The columns that are modified across all rank one updates are found by traversing the tree starting from the first entries in every column of $\mathbf{L}_{\mathcal{I}\mathcal{B}}$ (depicted red in Figure 4c,d). What contributes to the efficiency of the traversal is that it is stopped as soon as a node is found that has been discovered and treated previously. At this point it is guaranteed that all nodes up to the root have already been discovered while traversing from a different start node before.

The performance of the algorithm depends critically on the number of columns we have to update. Luckily the nested dissection structure ensures that in most practical cases this number is low compared to the recomputation of all columns – as would be needed for the full refactorization of the sub-mesh.

To see this, consider the selected surface patch (turquoise region in Figure 4a). As the surface patch only covers a very localized part of the mesh, the matrix $\mathbf{L}_{\mathcal{I}\mathcal{B}}$ will be very sparse with most of its columns containing only zeros. This is despite its large dimensions, covering all columns of vertices not contained in $\mathcal{I}$. Moreover, considering the position of the first non-zero per column in the elimination tree of $\mathbf{L}_{\mathcal{I}\mathcal{I}}$ (red in Figure 4c,d), it becomes clear that they will only induce a change in a small subset of columns. Note that

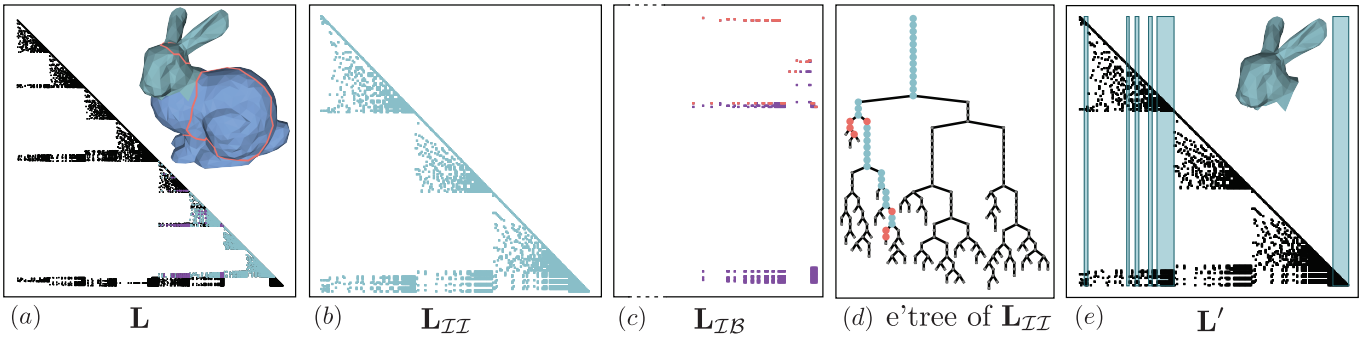


Fig. 4. (a) The turquoise section of the bunny model, consisting of vertices indexed by $\mathcal{I}$, is selected and a Cholesky factor $\mathbf{L}'$ for this sub-mesh is to be computed. The Cholesky factor of the full mesh $\mathbf{L}$ induces an elimination tree, the first three levels of which are highlighted on the mesh. (b) The submatrix $\mathbf{L}_{\mathcal{I}\mathcal{I}}$, highlighted in (a), is extracted. (c) The matrix $\mathbf{L}_{\mathcal{I}\mathcal{B}}$ can be used to determine the columns in $\mathbf{L}_{\mathcal{I}\mathcal{I}}$ that need to be updated. To do this, the index of the first entry in each column (highlighted in red) is identified and the elimination tree is traversed upwards, collecting the indices of columns that need to be updated (highlighted in (e)). To obtain the numerical values for the updated Cholesky factor $\mathbf{L}'$ the left-looking Cholesky factorization algorithm is run only on the highlighted columns while copying data from $\mathbf{L}_{\mathcal{I}\mathcal{I}}$ for all other columns.

there are only seven distinct row indices highlighted in Figure 4c. The columns corresponding to discovered nodes are highlighted in Figure 4e. Values in not highlighted columns are directly copied from $\mathbf{L}$ and will not be touched by the update procedure. All sub-trees that are completely contained in $\mathcal{I}$ are skipped when updating $\mathbf{L}_{\mathcal{I}\mathcal{I}}$ since $\mathbf{L}_{\mathcal{I}\mathcal{B}}$ cannot have a row index contained in these sub-trees. Again the block structure given by the nested dissection ordering limits the amount of computation. After the columns that are affected by the update are determined, the left-looking Cholesky algorithm is run on the sub-factor, skipping all columns that remain unchanged. Our algorithm can be summarized as follows. Given the vertex indices of the sub-mesh vertices $\mathcal{I}$

(1) Compute the elimination tree of $\mathbf{L}'$.
(2) Find the first non-zero in all columns of $\mathbf{L}_{\mathcal{I}\mathcal{B}}$.
(3) Traverse the elimination tree to its root starting at these indices and mark all columns corresponding to discovered nodes.
(4) Initialize the Cholesky factor $\mathbf{L}'$ of $\mathbf{A}_{\mathcal{I}\mathcal{I}}$ based on its elimination tree.
(5) Fill all unmarked columns of $\mathbf{L}'$ with corresponding values from $\mathbf{L}_{\mathcal{I}\mathcal{I}}$ and set all marked columns to zero.
(6) Run the left-looking Cholesky factorization of $\mathbf{A}_{\mathcal{I}\mathcal{I}}$ using the initialized factor while skipping unmarked columns.

Row indices. As discussed in Section 3, we store information about all columns in a row explicitly. As we are skipping most columns during factorization we cannot use techniques that rely on the fact that rows are accessed one after the other in order to access all values in a row. The row information we maintain has to be recomputed when extracting the sub-factor $\mathbf{L}_{\mathcal{I}\mathcal{I}}$, producing only a small overhead, however, it still doubles the amount of information necessary to store the non-zero structure of $\mathbf{L}_{\mathcal{I}\mathcal{I}}$. Since the algorithm is implemented in terms of supernodes instead of columns, the amount of structural information necessary is reduced compared to a regular sparse format. In Section 7 we analyze this overhead which amounts to below 15% of the total memory used to store the factorization in all our experiments.

6 IMPLEMENTATION

We implemented our algorithm from scratch in C++ inspired by Cholmod [Chen et al. 2008]. Since supernodal Cholesky factorization algorithms can leverage fast BLAS kernels we use the Intel MKL [int 2009] with openMP support. To ensure a fair comparison, we link Cholmod with the same library for our performance evaluation. Besides the need for a BLAS/LAPACK compatible library, the code we provide is self contained, however, if desired, it can be used in conjunction with Eigen [Guennebaud et al. 2010]. For nested dissection reordering we employ Metis [Karypis and Kumar 1998]. All timings have been conducted on a computer with a 3.49 GHz Intel Core i7 processor (four physical cores) and 32 GB of RAM.

7 EVALUATION

To evaluate performance we compare our approach to Cholmod [Chen et al. 2008]. For a set of meshes of different sizes (shown in Figure 5), patches are generated by randomly selecting a vertex and then collecting 10%, 25% or 50% of all vertices closest to that vertex. For each such patch, the sub-factor is extracted and updated according to our algorithm.

The traditional approach consists in building the operator for the surface patch and refactoring it from scratch using Cholmod. For a fair comparison, we do not include setup operations like forming a sub-mesh and setting up the operator matrix. Instead the submatrix $\mathbf{A}_{\mathcal{I}\mathcal{I}}$ is extracted directly from $\mathbf{A}$, which is much faster. We observe some variance in performance – our approach depends on how the patch appears in the elimination tree. When selecting a well separated feature, like in Figure 4, it is very likely that the sub-factor will contain large sub-trees of the elimination tree. Any sub-tree that is completely contained in the patch can be copied without modification. Selecting patches at random is therefore a slight disadvantage for our method.

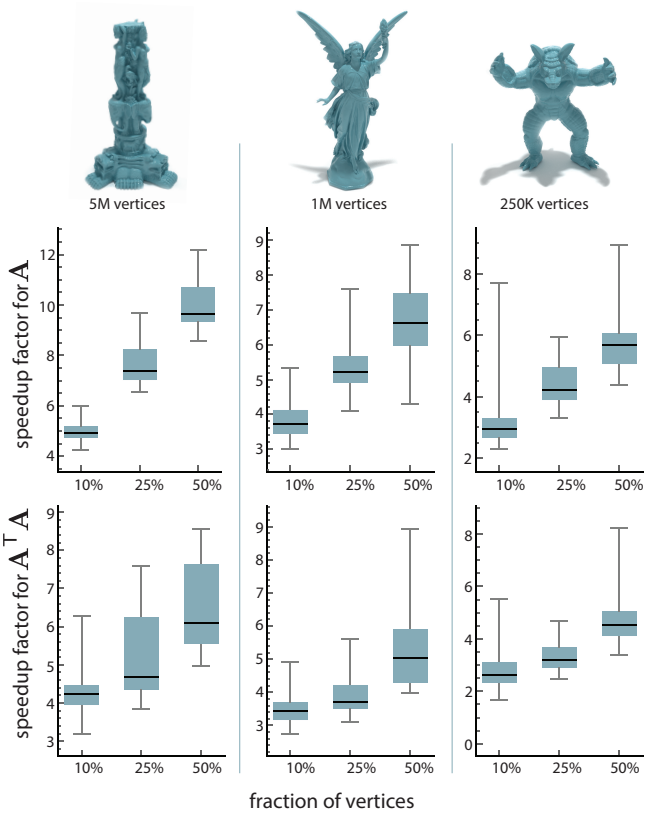


Fig. 5. Speedup factor of our update method compared to refactorization using Cholmod. The plots show the total range and quartiles for updates consisting of 10%, 25% and 50% of all vertices on three different meshes. Each experiment has been repeated 50 times for the mesh Laplacian $\mathbf{A}$ (upper row) and $\mathbf{A}^T\mathbf{A}$ (lower row).

Because the nested dissection ordering is spatially adaptive, it makes sense to extract sub-factors and submatrices in ascending order. In other words, the set of patch indices $\mathcal{I}$ is sorted. For a compact, localized patch, a restriction of the ordering to that patch will yield a good block structure which in turn guarantees a sparse factorization and fast updates. This is illustrated in Figure 4b. Selecting the submatrix from $\mathbf{L}$ while keeping the relative ordering of columns and rows produces a well ordered matrix $\mathbf{L}_{\mathcal{I}\mathcal{I}}$.

However, it might be possible to improve sparsity by reordering the matrix $\mathbf{A}_{\mathcal{I}\mathcal{I}}$, which is not an option for our update procedure. To test if Cholmod can profit from sorting the matrix, we ran our experiments with Cholmod twice, turning resorting on and off respectively. Across all instances, Cholmod could not amortize resorting times during the factorization phase. It appears that the matrices are already sorted in a way that allows efficient computation, i.e. induce a well balanced elimination tree so the benefit of resorting is negligible compared to the necessary computation.

Across all experiments, our method outperformed the costly refactorization step by a factor of 4 to 6 on average. Figure 5 highlights

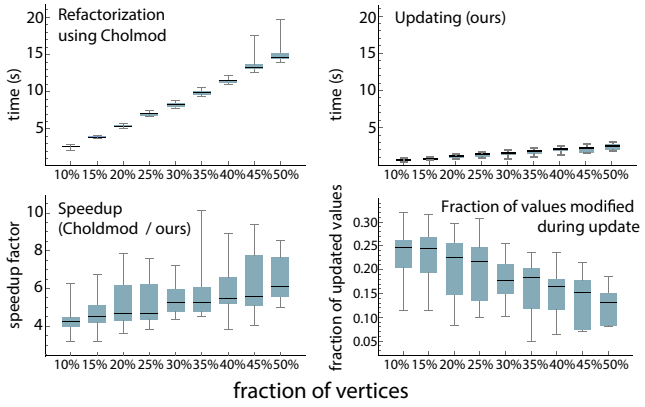


Fig. 6. Detailed statistics for the Cholesky factor of $\mathbf{A}^T\mathbf{A}$ on a mesh with 5 million vertices. Absolute updating and factorization times across 50 experiments are illustrated (top row) along with the distribution of speedup factors and the fraction of values that need to be updated in $\mathbf{L}_{\mathcal{I}\mathcal{I}}$.

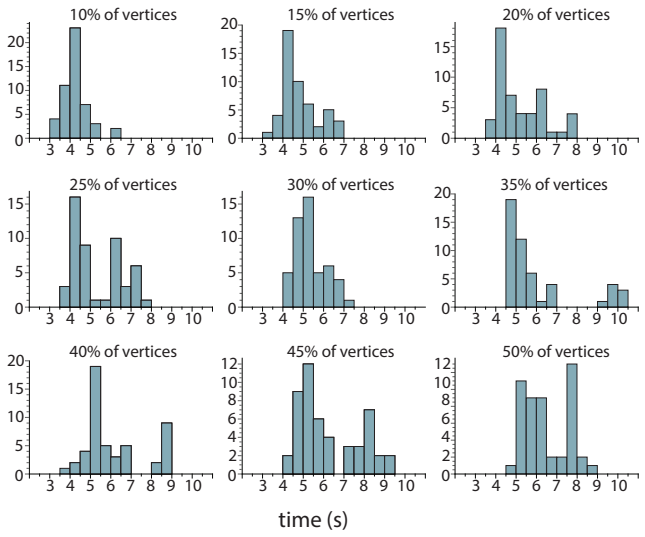


Fig. 7. Histograms of speedup for updates of the Cholesky factor of $\mathbf{A}^T\mathbf{A}$ on a mesh with 5 million vertices. The diagrams show the frequency of occurrence in 50 random updates producing Cholesky factorizations for sub-meshes of the given size.

how total mesh size affects our algorithm. The plots show the mean (black), quartiles (blue) and the range (gray) of speedup factors across 50 experiments. We present experiments on three meshes of different sizes updating the factorization of the mesh Laplacian $\mathbf{A}$ and $\mathbf{A}^T\mathbf{A}$ used in least squares systems.

Our method performs better for large patches on large meshes as in these cases a lot of data can be reused, which would otherwise have to recomputed. Best performance is to be expected if a majority of vertices is selected forming a locally separated patch as shown in Figure 1. This also explains the high variance towards larger

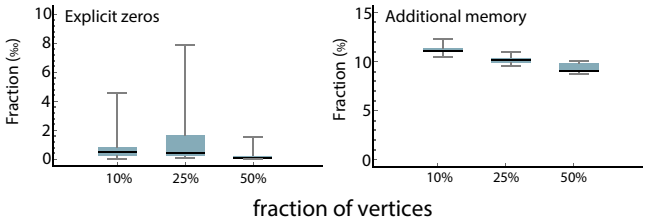


Fig. 8. Left: The fraction of matrix values that are explicitly stored in our updated factorization although they are actually zero. Right: The amount of extra memory used by our factorization to store additional row information.

mem (MB)		sub-mesh vertices		
		10%	25%	50%
vertices	250k	19 (17)	45 (40)	111 (100)
	1M	114 (113)	324 (294)	697 (637)
	5M	599 (539)	1626 (1476)	3395 (3096)

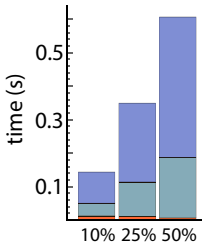
Table 1. Memory consumptions (in MB) for factors computed using our updating algorithm and using refactorization (in brackets).

speedups whenever the randomly chosen patch happened to select a well separated region.

The performance of Cholmod is measured separately and shown in Figure 6. The data shows almost linear growth in time with system size. The performance of our method depends critically on the fraction of values that have to be updated in the Cholesky factor (Figure 6, right). We can see, again, that updating becomes more beneficial for larger patch sizes.

Figure 7 shows histograms of performance data for a mesh with 5 million vertices across 50 random experiments for each sub-mesh size (Figure 5, left). The bimodal distribution for larger neighborhoods can be explained by patches that cover the complete upper or lower half of the mesh, and patches that cut the mesh in half, therefore having a larger interface region between vertices of the patch and the rest of the mesh.

Performance breakdown. To get a better understanding of how much each step of our algorithm contributes to the overall runtime, we break down the total runtime by setup costs such as determining the columns that need to be updated (red), copying data from the global Cholesky factor (green) and running the refactorization (blue). Again, we averaged values for 50 experiments on a mesh of one million vertices, selecting patches of 10^5, 2.5×10^5 and 5×10^5 vertices respectively. Generally 30% of the time is used to copy the unchanged values to the new factor. The refactorization accounts for around 68% and the rest of the time is spend for graph traversal to find the columns that need to be updated. These numbers are consistent across mesh and sub-mesh sizes.



Memory cost. Compared to refactoring from scratch our algorithm needs to maintain the matrix structure in row-major form

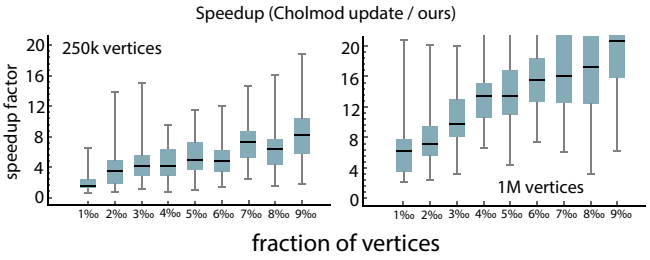


Fig. 9. Performance comparison of our algorithm and the update procedure implemented in Cholmod. Even for very small meshes and sub-meshes our method outperforms Cholmod significantly.

as well. This amounts to some extra memory that we measure as percentage of the memory for this data with respect to the total memory occupied by the Cholesky factor (Figure 8, right). Again we used a mesh with one million vertices and different sub-mesh sizes. The fraction of additional memory decreases with sub-mesh size and stays below 15% in all our experiments. The results are also consistent across mesh sizes. Table 1 shows absolute memory consumption in MB for factors computed by updating a global factorization compared to factorizations computed from scratch. In Section 5 we have shown that the non-zero structure of the extracted sub-factor $\mathbf{L}_{II}$ is compatible with the updated factor. However, there are possibly non-zeros present in $\mathbf{L}_{II}$ that are zero in $\mathbf{L}'$. This might be another source for additional memory. In Figure 8 (left) we show that these structural non-zeros that have the numerical value zero amount for less than 0.1% of all values and therefore occupy only a negligible amount of extra memory.

Comparison to low-rank updates. Cholmod provides a method to update Cholesky factors when the original matrix has been updated by a low rank modification of the form $\mathbf{A} + \mathbf{V}\mathbf{V}^\mathsf{T}$ with $\mathbf{V} \in \mathbb{R}^{n \times k}$ [Davis and Hager 2009]. This method is quite efficient when the update has low rank (i.e. k is very small). We could use this method to perform the update in Equation (18), which is the central step in our algorithm. Even for very small sub-meshes with less than 0.1% of the vertices our algorithm outperforms Cholmod's update procedure (Figure 9). The speedup of our method compared to Cholmod becomes more extreme for larger meshes and we conclude that our algorithm is to be preferred for almost all practical usage scenarios where sub-meshes of reasonable size are considered. However, when updating only a few rows of a least squares system to constrain single vertices, which might for example be necessary when constructing Least Squares Meshes [Sorkine and Cohen-Or 2004], Cholmod's algorithm will be faster.

8 DISCUSSION & CONCLUSION

We proposed a method to quickly build a sparse Cholesky factorization for a linear operator defined on a localized sub-mesh using a factorization of an operator defined on the full mesh. This operation is more efficient than constructing a new factorization for that sub-mesh from scratch. Depending on the shape and size of the sub-mesh we observe speedup factors of 4 to 6 on average, however, this factor gets higher for sub-meshes covering a significant

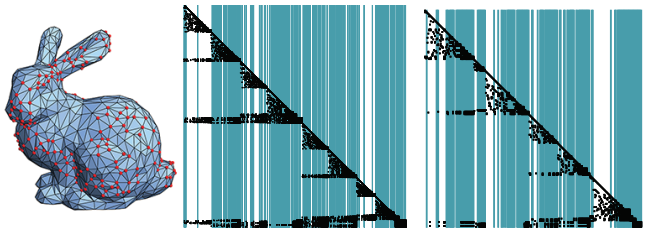


Fig. 10. Non local patches (left) will induce sub-factors $\mathbf{L}_{\mathcal{I}\mathcal{I}}$ collecting data from many different sub-trees of the elimination tree of $\mathbf{L}$ (center, columns in $\mathcal{I}$ colored) and consequently require costly updates of $\mathbf{L}_{\mathcal{I}\mathcal{I}}$ (right, columns requiring an update are highlighted).

number of vertices on large meshes. These are the situations where performance is most critical and applications will benefit from our algorithm the most. Moreover, our approach was never slower than a complete refactorization in our experiments because it just performs a partial refactorization, reducing to a full refactorization in the worst case, which is very unlikely for sensible regions of interest. An example of an update that is not very efficient is depicted in Figure 10. Selected vertices (in red) are not localized and the columns and rows that are selected by $\mathbf{L}_{\mathcal{I}\mathcal{I}}$ are scattered all over $\mathbf{L}$. This also means that many sub-trees of the elimination tree are covered and a large number of columns have to be updated as depicted in the right image. However even in this extreme example not all columns of the factor need to be updated and we can slightly outperform a complete refactorization.

Compared to other methods for fast factorization updates, like [Yeung et al. 2016], our method is not approximate. We construct the unique Cholesky factor of the constrained matrix, moreover, since we are just reusing information that has been already computed and compute updated columns from scratch, our algorithm is numerically as stable as a complete refactorization. This also enables cascading updates: After computing the Cholesky factor with respect to a sub-mesh by our update procedure it can be used to compute a Cholesky factor on a new sub-mesh contained in the first one. This is a reasonable scenario when interactively editing a mesh.

As the solution of linear systems is a fundamental building block in geometry processing and simulation we expect a variety of methods to benefit from our algorithm. We provide ready to use C++ code that only depends on a BLAS/LAPACK implementation to facilitate further applications. The code can be conveniently used with Eigen [Guennebaud et al. 2010] sparse matrices.

ACKNOWLEDGMENTS

We thank David Lindlbauer for helping us create the figures in this paper and Timothy A. Davis for providing Cholmod. This work was supported by the German Federal Ministry for Economic Affairs and Energy (BMWi) under Grant No. 01MT16004D.

REFERENCES

2009. *Intel Math Kernel Library. Reference Manual.* Intel Corporation.

Mario Botsch and Olga Sorkine. 2008. 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

Sofien Bouaziz, Mario Deuss, Yuliy Schwartzburg, Thibaut Weise, and Mark Pauly. 2012. Shape-Up: Shaping Discrete Geometry with Projections. *Computer Graphics Forum* 31, 5 (2012), 1657–1667. https://doi.org/10.1111/j.1467-8659.2012.03171.x

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), 22.

Keenan Crane, Clarisse Weischedel, and Max Wardetzky. 2013. Geodesics in Heat: A New Approach to Computing Distance Based on Heat Flow. *ACM Trans. Graph.* 32, 5, Article 152 (Oct. 2013), 11 pages. https://doi.org/10.1145/2516971.2516977

T. 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

T. A. Davis. 2006. *Direct Methods for Sparse Linear Systems.* SIAM, Philadelphia, PA.

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. 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

Mathieu Desbrun, Mark Meyer, and Pierre Alliez. 2002. Intrinsic Parameterizations of Surface Meshes. *Computer Graphics Forum* 21, 3 (2002), 209–218. https://doi.org/10.1111/1467-8659.00580

Mathieu Desbrun, Mark Meyer, Peter Schröder, and Alan H. Barr. 1999. Implicit Fairing of Irregular Meshes Using Diffusion and Curvature Flow. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '99).* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 317–324. https://doi.org/10.1145/311535.311576

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, 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

George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing* 20, 1 (1998), 359–392.

Joseph W.H. Liu. 1990. The Role of Elimination Trees in Sparse Factorization. *SIAM J. Matrix Anal. Appl.* 11, 1 (1990), 134–172. https://doi.org/10.1137/0611010

Mark Meyer, Mathieu Desbrun, Peter Schröder, and Alan H. Barr. 2003. *Discrete Differential-Geometry Operators for Triangulated 2-Manifolds.* Springer Berlin Heidelberg, Berlin, Heidelberg, 35–57. https://doi.org/10.1007/978-3-662-05105-4_2

Patrick Mullen, Yiying Tong, Pierre Alliez, and Mathieu Desbrun. 2008. Spectral Conformal Parameterization. In *Proceedings of the Symposium on Geometry Processing (SGP '08).* Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 1487–1494. http://dl.acm.org/citation.cfm?id=1731309.1731335

Ulrich Pinkall and Konrad Polthier. 1993. Computing discrete minimal surfaces and their conjugates. *Experim. Math.* 2 (1993), 15–36.

Olga Sorkine and Marc Alexa. 2007. As-rigid-as-possible Surface Modeling. In *Proceedings of the Fifth Eurographics Symposium on Geometry Processing (SGP '07).* Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 109–116.

Olga Sorkine and Daniel Cohen-Or. 2004. Least-Squares Meshes. In *Proceedings of the Shape Modeling International 2004 (SMI '04).* IEEE Computer Society, Washington, DC, USA, 191–199. https://doi.org/10.1109/SMI.2004.38

Olga Sorkine, Daniel Cohen-Or, Yaron Lipman, Marc Alexa, Christian Rössl, and Hans-Peter Seidel. 2004. Laplacian Surface Editing. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing (SGP '04).* ACM, New York, NY, USA, 175–184. https://doi.org/10.1145/1057432.1057456

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