Localized solutions of sparse linear systems for geometry processing

PHILIPP HERHOLZ, TU Berlin
TIMOTHY A. DAVIS, Texas A&M University
MARC ALEXA, TU Berlin

Computing solutions to linear systems is a fundamental building block of many geometry processing algorithms. In many cases the Cholesky factorization of the system matrix is computed to subsequently solve the system, possibly for many right-hand sides, using forward and back substitution. We demonstrate how to exploit sparsity in both the right-hand side and the set of desired solution values to obtain significant speedups. The method is easy to implement and potentially useful in any scenarios where linear problems have to be solved locally. We show that this technique is useful for geometry processing operations, in particular we consider the solution of diffusion problems. All problems profit significantly from sparse computations in terms of runtime, which we demonstrate by providing timings for a set of numerical 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, Timothy A. Davis, and Marc Alexa. 2017. Localized solutions of sparse linear systems for geometry processing. *ACM Trans. Graph.* 36, 6, Article 183 (November 2017), 8 pages.
https://doi.org/10.1145/3130800.3130849

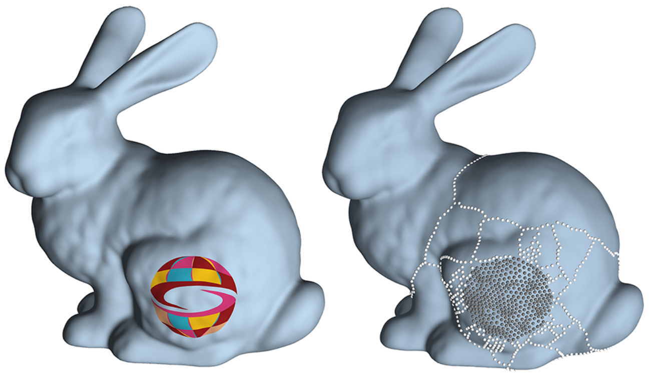


Fig. 1. To parameterize a small surface patch (left) a global factorization of the Laplacian can be utilized. For the computation of the exact solution for patch vertices (gray vertices on the right) not all columns of the Cholesky factor have to be considered. More specifically, only those associate with paths in the elimination tree connecting root and patch-vertices are required additionally (white vertices).

1 INTRODUCTION

The central operation in many geometry processing tasks is the (repeated) solution of a sparse linear system [Botsch and Sorkine 2008; Bouaziz et al. 2012; Chao et al. 2010; Desbrun et al. 2002, 1999; Lévy et al. 2002; Sorkine and Alexa 2007]. These problems can be efficiently solved by computing a (sparse) factorization of the system matrix and then performing a step of forward and back substitution. This is particularly attractive if several solves against varying right-hand sides are necessary.

Computing a solution vector given the factorization is quite efficient if the factors are sparse, but solving for a simple regular 2D mesh with n vertices still has a time complexity of at least $O(n \log n)$ [George 1973]. However, many geometry processing operations are local, i.e. a solution to the linear system is required only for a subset of the mesh (e.g. for generating a parameterization of a patch). Our main contribution lies in describing a method that uses a global factorization to compute solutions for subsets of mesh vertices in time that depends mostly on the size and structure of the subset and not the whole mesh. This approach is favorable to the common solution of setting up a new system for the subset of the mesh and factorizing it.

In order to present our approach we first provide some background on sparse Cholesky factorizations in Section 2. Then we explain how sparsity in the right-hand side as well as in the desired solution can be exploited. Sparsity in the right-hand side leads to sparse intermediate solutions – this is commonly exploited in libraries for numerical linear algebra. Our contribution lies in exploiting 'sparsity' in the desired solution. By sparsity we mean that only a subset of the solution vector is desired – while the solution may still be dense, i.e. nonzero in all its elements. We explain how this is possible without resorting to an approximation: the solution for the desired elements is exactly the same as if all elements of the solution were computed. An important point of our analysis and the resulting algorithm is that the sparsity in the solution can be exploited independently from sparsity in the right-hand side (if any). In addition, we explain how our implementation exploits vectorization and cache-coherence. The details of the analysis and resulting algorithms are described in Section 3.

We first evaluate the effectiveness of the main operations at the common example of generating a conformal parameterization with prescribed boundary. Our approach (see Section 4) follows the now classic idea to generate a harmonic embedding [Desbrun et al. 2002] and requires only a single linear solve. This allows us to compare the time required for numerical solutions based on the local patch or using the sparse localized solve based on the pre-factored global

We wish to thank the anonymous reviewers for their constructive comments and suggestions. This work was supported by the German Federal Ministry for Economic Affairs and Energy (BMWi) under Grant No. 01MT16004D..

0730-0301/2017/11-ART183 $15.00
https://doi.org/10.1145/3130800.3130849

system. We find that our approach provides a significant speed-up in all practical scenarios, suggesting that also iterative approaches to parameterization may benefit from localized sparse solves.

Next we analyze the behavior for extremely sparse vectors at the example of discrete heat diffusion. Here, we assume that the heat source is local (i.e. a single vertex). Then we demonstrate the gain in efficiency if only a single value is desired, either far away or in the vertex itself (the case of auto-diffusion, which is used for many signatures and matching [Gebal et al. 2009; Sun et al. 2009]). Both cases show significantly reduced computational cost compared to standard solutions; the scenario of different locations for source and desired heat value allow us to analyze the benefit of exploiting the sparsity in the solution independently from the sparsity in the right-hand side (see Section 5).

2 BACKGROUND

The central operation we are concerned with are solutions of linear systems that are the result of discretizing differential equations over the domain of a triangle mesh. Let the mesh be denoted $\mathcal{M} = (\mathbf{V}, \mathcal{F})$, with n being the number of vertices. Vertex positions are stored as a matrix $\mathbf{V} \in \mathbb{R}^{3\times n}$ and $\mathcal{F}$ as a matrix of integers $\mathbb{Z}^{3\times|\mathcal{F}|}$ indexing vertices in $\mathbf{V}$.

A large number of geometry processing operations involve discrete differential operators. There are many alternatives on how to derive the matrix representations of these operators [Alexa and Wardetzky 2011; Elcott and Schröder 2006; Meyer et al. 2003; Pinkall and Polthier 1993]. The important point is that in all constructions the operators are built from the adjacency structure of the mesh, and the sparsity is identical to a low order power of the adjacency matrix. Moreover, in most cases the system is, or can be made, symmetric. In other words, the common task is to solve a sparse symmetric system of the form $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{x}$ represents values attached to the mesh elements, most commonly the vertices, in which case $\mathbf{A} \in \mathbb{R}^{n\times n}$.

Sparse Cholesky factorization. The resulting sparse symmetric linear system can be solved using the Cholesky factorization

$$\mathbf{A} = \mathbf{LDL}^\mathsf{T} \quad \text{with } \mathbf{L}, \mathbf{D} \in \mathbb{R}^{n\times n}. \tag{1}$$

Here, $\mathbf{D}$ is a diagonal matrix, and $\mathbf{L}$ is a lower triangular matrix with unit diagonal commonly called the *Cholesky factor.* The Cholesky factorization has many favorable properties, for example it is unconditionally stable, allowing permutations for optimizing other properties. Important in our context, a permutation can be selected with the goal to reduce *fill-in.* A fill-in entry is a nonzero that appears in $\mathbf{L}$ but not in $\mathbf{A}$. In particular, for the adjacency structures resulting from triangle meshes it is possible in general to generate sparse Cholesky factors using an appropriate ordering.

Elimination tree. The sparsity of the Cholesky factor is important in our context because it leads to efficient solves. Solving a system for $\mathbf{x} \in \mathbb{R}^n$ given a right-hand side $\mathbf{b} \in \mathbb{R}^n$ requires a forward substitution (or forward solve) step to compute the auxiliary vector $\mathbf{y} \in \mathbb{R}^n$

$$\mathbf{Ax} = \mathbf{b} \Rightarrow \mathbf{LDL}^\mathsf{T}\mathbf{x} = \mathbf{b} \Rightarrow \mathbf{LDy} = \mathbf{b} \tag{2}$$

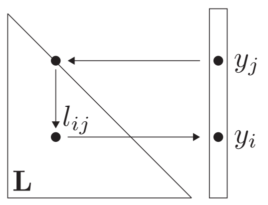
followed by the back substitution $\mathbf{L}^\mathsf{T}\mathbf{x} = \mathbf{y}$.

Now consider computing the forward solve. The elements of the solution $\mathbf{y}$ are computed in order of increasing index, i.e. $y_0 = b_0/d_{00}$ is computed first, then y_1, and so on.

For a sparse right-hand side $\mathbf{b}$ the vector $\mathbf{y}$ is usually also sparse and knowing the nonzero structure of $\mathbf{y}$ in advance would significantly improve performance — columns in $\mathbf{L}$ corresponding to a zero in $\mathbf{y}$ can simply be ignored.

In what follows we refer to elements as *nonzero* if they potentially take on values different from zero, because their computation depends on other nonzero elements. It is still possible that nonzero elements carry a zero value due to numerical cancellation. Zero values of this kind are commonly not exploited algorithmically.

The key observations for the nonzero structure of $\mathbf{y}$ are the following: (1) if b_i is nonzero then y_i is nonzero (because $\mathbf{D}$ has to be full-rank), and (2) if y_j is nonzero and there is a nonzero l_{ij}, then y_i will be nonzero, as illustrated in in the inset to the right. Consequently we can find the nonzero structure of $\mathbf{y}$ by traversing the graph of $\mathbf{L}$ that has an *oriented* edge (j, i) for each nonzero l_{ij}. Note that all edges in the graph are directed from a node with smaller index to a node with larger index, reflecting the computation of elements in increasing order of index. The traversal is started in the nodes corresponding to the nonzeros in $\mathbf{b}$, following observation (1). Then the graph is traversed following the oriented edges, reflecting observation (2). The nonzero structure of $\mathbf{y}$ is given by the nodes that are visited during this traversal [Gilbert and Peierls 1988].



The graph mentioned above is a useful tool for quickly identifying the columns in $\mathbf{L}$ that need to be treated during the solve. To store the graph without compromising on the possibility to quickly identify all dependencies a particular spanning tree of the graph is used: the *elimination tree.* This tree results from keeping only one *outgoing* edge for each node. In particular, in node i we pick the edge (i, j) with the smallest index j. By retaining only one edge per node and since all edges go from nodes with smaller index to larger index, it is clear that in the resulting graph there is at most one path from a node with smaller index to one with higher index. But why is it connected? It is a specific property of the Cholesky factor that whenever node i has outgoing edges (i, j) and (i, k) with $j < k$, then node j has the outgoing edge (j, k). We refer the reader to Liu [1990] and Davis [2006] for an explanation and proof of this property.

The elimination tree not just stores the connectivity of the graph, it also allows identifying the dependence among nonzero elements by starting the traversal in any of its nodes. By traversing only along the spanning tree we are guaranteed to reach all nodes we would find when traversing the full graph. A space and run-time efficient representation is to simply store for each index j its parent index $\pi(j)$ in the elimination tree.

By exploiting this representation of the elimination tree, computing the nonzero pattern of $\mathbf{y}$ takes an amount of time proportional to the number of nonzeros in $\mathbf{y}$.

Ordering: nested dissection. An important factor in the efficiency of computing the intermediate solution for sparse right-hand sides is

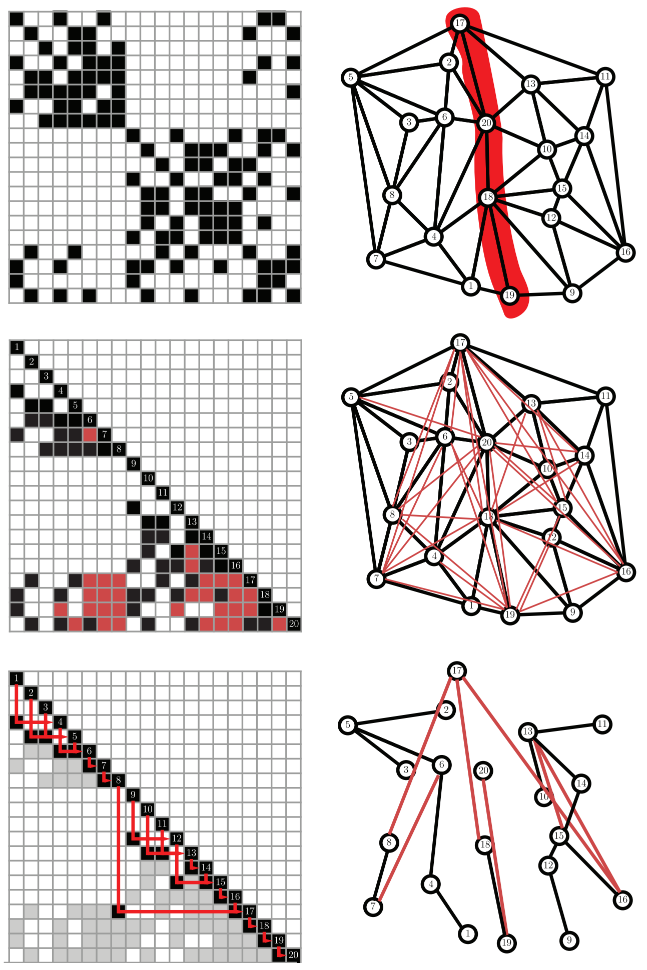


Fig. 2. The nonzero structure of a discrete Laplacian (upper left) based on the triangular mesh (upper right). The corresponding Cholesky factor contains elements that were zero in the Laplacian matrix (middle), however, because the elimination tree (lower right or inset) is nearly balanced, fill in can be limited. A balanced elimination tree will also facilitate fast solving times for sparse and localized sets of vertices.

the (average) length of the sequence $(j, \pi(j), \pi(\pi(j)), \ldots)$ or, in other words, the height of the elimination tree. Of much lesser importance (in our context) is the fill-in: note that zero fill-in structure might still have an elimination tree with worst case height. The sequence $\pi(j) = j + 1$, for example, corresponds to a dense subdiagonal and would imply that starting from column j all following columns are visited, which is similar to the situation for dense matrices, so the number of columns considererd are linear instead of logarithmic.

To limit the height of the elimination tree one can exploit a crucial property of Cholesky factors: If a_{ij} is the leftmost entry in row i of the matrix $\mathbf{A}$, all values l_{ik} with $k < j$ in the factor $\mathbf{L}$ are guaranteed to be zero. For the example in Figure 2 this implies that the block in rows $1 - 8$ will not be connected to block $8 - 16$ in the matrix graph of $\mathbf{L}$, therefore these blocks reside on separate subtrees in the elimination tree. This property can be brought to use by reordering the matrix columns and rows such that blocks are grouped together. In figure 2 (upper right) a *separator* consists of nodes 17, 20, 18, 19 wich are sorted last. All nodes left and right are grouped and sorted before the separator. Proceeding recursively produces a factorization with balanced elimination tree.

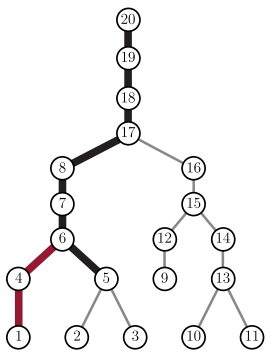
This process is called *nested dissection* [George and Liu 1978]. Note that finding the best ordering to minimize the tree height and/or to minimize fill-in is an NP-hard problem, so heuristics must be used. The main goal in nested dissection is to quickly find *good* dividers, i.e. consisting of a small number of edges while still generating similarly sized components, each of which can again be divided by good dividers. We note that the graphs of the meshes encountered in geometry processing naturally allow finding good dividers. The reason for this is that the triangle mesh is *embedded*. This means, adjacency along the surface, which is important for the differential operators, corresponds to adjacency in the graph. This makes it possible to find efficient dividers in the graph. Davis [2006] notes that for many discretization of two-dimensional problems this leads to an overall computational complexity of $O(n^{3/2})$ with $O(n \log n)$ nonzeros in $\mathbf{L}$. In Section 4 we indeed observe this to be the case for the particular problems we consider here.

3 LOCAL SOLVES

We have seen that it is possible to reorder the matrices commonly encountered in geometry processing such that the Cholesky factors have a small number of fill-edges (the total number of nonzero elements is on the order $O(n \log n)$ if the elimination tree is fairly balanced). The elimination tree provides an efficient implementation of the forward and back substitution for sparse right-hand sides.

Consider a right-hand side $\mathbf{b}$ with a single nonzero element for index i. The forward solve involves only visiting the elements on the path from i in the elimination tree to the root, which is available as the sequence $(i, \pi(i), \pi(\pi(i)), \ldots)$. Choosing $i = 5$ in the example (see Figure 2 and inset), one ends up with the nodes on the bold black path which represent the only columns that have to be considered. This is not only computationally efficient, it also means the number of nonzero elements in the intermediate solution $\mathbf{y}$ is bounded by the height of the elimination tree or, in other words, $\mathbf{y}$ is very sparse.

What happens if in addition to b_i the element b_j is nonzero? The additional elements in $\mathbf{y}$ are given by the path $(j, \pi(j), \pi(\pi(j)), \ldots)$.



If the additional node lies on the path of b_i, e.g. $j = 7$ in our example, there is no extra computational work involved. For a node not on the path the additional work depends on the position in the tree. If the paths for nodes i and j have many nodes in common, e.g. for $j = 1$, the amount of additional computation is very limited. This is a consequence of the nested dissection ordering, which is constructed such that vertices close in the triangle graph are likely to fall in the same subtree. More generally, the intermediate solution $\mathbf{y}$ is sparse for sparse right-hand sides $\mathbf{b}$ if the nonzero elements in $\mathbf{b}$ are close to each other. This property is also illustrated in Figure 1 which shows the subtree of nodes necessary to perform a forward solve for the gray vertices.

In other words, the particular geometry of many problems in geometry processing lends itself well to the sparse solve if nested dissection has been used for ordering.

While these connections between geometry processing and sparse linear solves might not be obvious, they have been routinely exploited by simply calling the right library functions. In the following we extend the possibilities for exploiting sparsity by also considering sparsity in the desired solution vector $\mathbf{x}$.

Subset solve. Even if the right-hand side $\mathbf{b}$ and the intermediate vector $\mathbf{y}$ are sparse, the final solution $\mathbf{x} = L^{-\mathsf{T}}\mathbf{y}$ is usually dense. For example in heat diffusion with a single vertex as source, $\mathbf{b}$ contains only a single nonzero entry but all vertices receive some nonzero quantity of heat, regardless of the time-step (c.f. Section 5). However, it turns out that any subset of the elements in the solution $\mathbf{x}$ can usually be computed *exactly*, without computing all of the solution.

Consider performing a back substitution with an upper triangular n-by-n system $\mathbf{L}^{\mathsf{T}}\mathbf{x} = \mathbf{y}$, one row at a time. The right-hand side $\mathbf{y}$ is sparse, and derives from the forward solve described in Section 2. The backsolve accesses $\mathbf{L}$ column-by-column (or $\mathbf{L}^{\mathsf{T}}$ row-by-row, the i-th step accessing the i-th column of $\mathbf{L}$):

```
for $i = n$ down to 1 do
    $x_i = y_i$
    for each $j > i$ for which $l_{ji} \neq 0$ do
        $x_i = x_i - l_{ji} x_j$
    $x_i = x_i / l_{ii}$
```

At first glance, it would seem that nothing can be gained by considering only a subset of the solution vector $\mathbf{x}$.

However, suppose we are interested in the solution only at a subset of the vertices. These vertices correspond to a particular subtree of the elimination tree (see Figure 1). By construction, all elements in this subtree are not influenced by the value in any other subtree. In other words, the solution for any subtree of the root node can be computed independently. This argument applies recursively for any subtree. In particular, suppose only a single entry x_i is desired. It suffices to compute all the components of $\mathbf{x}$ along the path from node i to the root, starting from the root and moving down towards node i. This gives a similar result as the forward solve: starting with the nodes of desired values of $\mathbf{x}$, find the reach of these nodes to determine the components of $\mathbf{x}$ that must be computed.

In a preprocessing symbolic analysis phase, we traverse the elimination tree from the nodes of all desired values and store the set of visited indices. This index set represents all nodes that could possibly influence the desired values. Visiting the necessary rows in order yields only a subset of the solution, including the desired nodes, and the values in this subset are correct. Note that the symbolic traversal goes up the tree to the root, starting from the desired nodes, but the numeric back substitution traverses the same nodes in opposite order.

Exploiting the elimination tree, the symbolic analysis time for determining which components of $\mathbf{x}$ are required for computing the desired subset is proportional to the size of the output set (the visited indices).

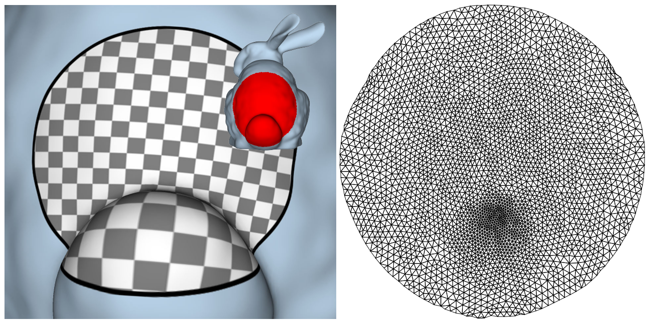


Fig. 3. A surface patch parameterized using localized harmonic parameterization.

It is important that the sparsity of $\mathbf{b}$ and the desired subset of $\mathbf{x}$ play different roles for the efficiency of the computation and can (and should) be set independently: the sparsity of $\mathbf{b}$ affects the efficiency of the forward substitution and the sparsity of the intermediate solution $\mathbf{y}$. The desired subset in the solution affects the subsets of rows that need to be evaluated during back substitution. It is possible, and we will exploit this feature, that the nonzeros in $\mathbf{b}$ are disjoint from the set of desired values in $\mathbf{x}$.

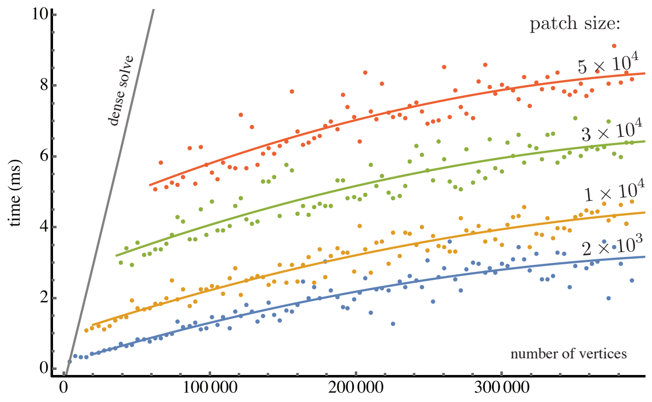
Solving for multiple right-hand sides. In geometry processing applications, geometry is often considered to be the signal. This means we commonly need to solve the same system for the different components of each coordinate, e.g. the x- and the y-vector in a parameterization task. In any case that requires solving for several right-hand sides with the same sparsity (but different numerical values) it is beneficial to perform the solves simultaneously. This has two reasons: First, cache coherence is improved for accessing the data structures that store the factorization. And, second, vectorization can be exploited for the numerical operations.

In our implementation we use templates to provide a convenient way to provide optimized solvers for different situations. Because of the limits of vectorization we limit the number of concurrent solves to 8. In this scenario we observe a 2.5-fold speed-up for 8 concurrent solves compared to 8 sequential ones.

4 POISSON PROBLEMS ON PATCHES

One of the most basic tasks in geometry processing is the parameterization of a surface patch. The classic technique is based on elementary results in complex analysis, namely that every topological disk admits a conformal mapping into a disk. The analogue for triangle meshes is the discrete harmonic map [Eck et al. 1995]. This method minimizes the Dirichlet energy subject to Dirichlet boundary conditions, i.e. fixing boundary vertices in the plane. To compute these mappings, the linear system $\mathcal{L}_b\mathbf{x} = \mathbf{b}$ with $\mathbf{x}, \mathbf{b} \in \mathbb{R}^{n\times 2}$ has to be solved, where $\mathcal{L}_b$ represents the cotan-Laplacian [Pinkall and Polthier 1993]. The boundary conditions require replacing rows that correspond to boundary vertices by canonical basis vectors and the right-hand side $\mathbf{b}$ is a sparse vector containing the x and y values of the desired positions, respectively.

We wish to use the factorization for the full mesh $\mathcal{M}$ to efficiently compute a harmonic parameterization of a surface patch



patch size:	2000	10000	30000	50000
refactorize & solve	6	42	180	386
conjugate gradients	18	227	1209	2580

Fig. 4. Parameterizing surface patches using sparse solves depends on the size of the full mesh. However, even for small patches on large meshes the method still significantly outperforms conjugate gradients and refactorization & solve techniques which do only depend on the patch size (see table, timings in ms).

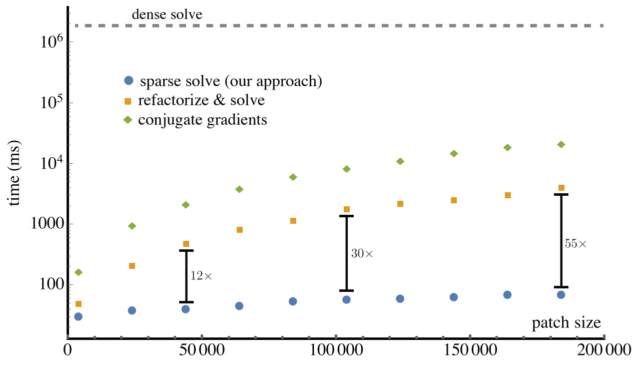


Fig. 5. For a mesh with 4×10^6 vertices patches of different size are parameterized. Despite the relatively large size of the full mesh compared to the surface patches, sparse solves are still orders of magnitude faster then local approaches. Because the number of columns involved in a sparse solve depend on the pattern of the right-hand side, we show the average of 100 runs for each patch size. The constant cost for a dense solve is depicted for reference as gray line.

$\mathcal{M}_p \subset \mathcal{M}$. The global factorization is unaware of the patch boundary $\partial\mathcal{M}_p$. This could be fixed by updating the system, which is possible but incurs extra cost. We rather opt for Neumann boundary conditions [Desbrun et al. 2002]. This allows us to use the Laplacian operator matrix $\mathcal{L}$ for the full mesh. We suggest generating the Neumann boundary data by laying out the boundary vertices in the plane and computing their Laplacian coordinates – other techniques would also work. The right-hand side value for a boundary vertex i is then computed by summing over each triangle containing the vertex i:

$$\mathbf{b}_i = \sum_{(i,j,k)\in\mathcal{F}_p} (\mathbf{v}_k - \mathbf{v}_j)^{\perp}. \tag{3}$$

We are now interested in the solution of $\mathcal{L}\mathbf{x} = \mathbf{b}$, evaluated at vertices $\mathcal{M}_p \subset \mathcal{M}$ of the surface patch. Note that this system also defines the positions of the vertices $\mathcal{M} \setminus \mathcal{M}_p$ outside of the surface patch. The beauty of our approach is that their positions do not have to be computed and have no effect on neither the solution inside the patch nor the computational complexity of the sparse subset solve applied to the interior of the patch.

Evaluation. We conduct two experiments to evaluate the computational efficiency of our approach relative to pertinent parameters. We compare the performance to the two solutions suggested in the literature, namely computing the solution inside the patch using conjugate gradients, or generating a new system matrix for the patch, factorizing the matrix, and computing the solution using the sparse factorization.

We base our implementation on Eigen. Sparsity is exploited whenever possible. In particular, we also exploit the sparsity of the right-hand side for solving of the locally factored system. For conjugate gradients we use diagonal preconditioning (the default in Eigen). For reference we also include timings for forward and back substitution without exploiting sparsity.

In the first experiment we consider patches with fixed number of vertices and vary the mesh size. The idea is that traditional approaches would be affected only by the size of the patch, but not by the size of the surrounding mesh. This is different in our approach, as the height of the elimination tree and, consequently, the local solve, depend on the size of the mesh.

We generate patches with vertex counts 2×10^3, 1×10^4, 3×10^4 and 5×10^4 using BFS on the vertex-edge graph of the triangle mesh. The table in Figure 4 shows the running times of the traditional solutions, which were indeed nearly constant across all experiments. The graph in Figure 4 shows the performance of our approach. As expected, the runtime increases with the size of the mesh, seemingly involving a logarithmic dependence. The local variation in runtime is due to the varying alignment of the patches with the nested dissection structure. In other words, if the patch happens to align nicely with the separators used in nested dissection, the sparsity in the right-hand side and the patch results in fewer nonzeros in the solution and reduced cost. If the patch intersects many such separators at a high level, the performance degrades. Importantly, the performance of our approach is significantly better then the traditional approaches, even in unfavorable constellations.

In the second experiment we take a fixed mesh with 4×10^6 vertices and vary the patch size. We take a relatively large mesh, as smaller meshes lead to smaller elimination trees and would be to the advantage of our approach. In this scenario we consider comparably small patches as, vice versa, larger patches are to the disadvantage of the traditional techniques. The graph in Figure 5 shows the result of this experiment: for all but the smallest patch sizes our approach outperforms the traditional techniques by one or more orders of magnitude.

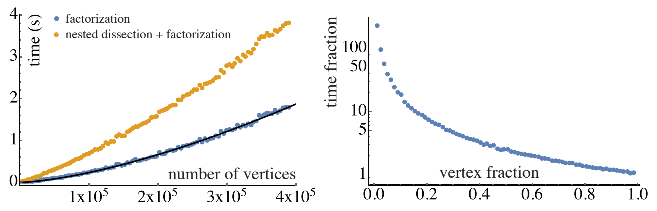


Fig. 6. Time required for Cholesky factorization and nested dissection (Metis) reordering (left). Time ratio of local patch factorization and factorization time for the full mesh over vertex fraction of the patch (right). Time fraction quantifies how many local factorizations can be computed until it becomes beneficial to compute and exploit a global factorization.

Unsurprisingly, the dense solve is the slowest in all experiments. And, consistent with the literature on the numerical solution of the 2D Poisson problems, the direct solvers based on sparse factorization outperform the iterative solver based on conjugate gradients.

Reusing the factorization of the mesh Laplacian obviously requires building this factorization in a preprocess. Figure 6 illustrates the time required for meshes of different size. For triangle meshes reordered by nested dissection the asymptotic runtime of $O(n^{3/2})$ [Golub and Van Loan 2013, 11.1.7] can be observed (see fitted curve). The second plot also includes time taken for nested dissection reordering (using Metis [Karypis and Kumar 1998]).

In many geometry processing applications the factorization of the full mesh is necessary anyway, and it could simply be reused for localized solves. If only few small patches are processed it may be faster to compute the factorizations for the subsets. The amortization of a full factorization depends on the size of the patch and the full mesh. We have found that the ratio of factorization times depends only on the ratio of vertex counts of the mesh and the patch and is independent from the absolute numbers of vertices. Figure 6 (right) illustrates the ratio of factorization time as a function of the relative size of the patch. For example, factorizing the Laplace matrix of a patch consisting of roughly 20% of the vertices is 10 times faster then factorizing the matrix for the whole mesh.

Compared to the times necessary for factorization, solving requires negligible time. So if indeed the overall processing time is of importance, one would have to estimate how many local solutions on patches of what size are required in order to gauge the benefit of local vs. global factorization. To keep with the previous example, if less then 10 local geometry processing problems have to be solved on patches consisting of 20% of the vertices in the mesh it is better to compute local factorizations. For more then 10 problems it is better to compute a global factorization.

Non-linear parameterization. More recent techniques for parameterizing patches are non-linear and typically involve iterative solutions of sparse linear systems. Some popular techniques are based on a fixed system and only modify the right-hand side throughout the iterations: As-rigid-as-possible parameterization [Liu et al. 2008] optimizes the per-triangle transformation to be rigid or a similarity transformation. They report up to 10 iterations for the minimization of their energy. Spectral-conformal parameterizations [Mullen et al. 2008] are based on extracting the Fiedler vector from a Laplacian system. Computing this eigenvector can be done based on inverse iterations quite efficiently using our approach, as the Cholesky factors allow quick computation of the iterates. Inverse iteration is known to converge extremely rapidly, with a good starting guess in well under 5 iterations [Ipsen 1997].

Given the performance data we have shown, even for these techniques it is beneficial to exploit a global pre-factorization and then use the local sparse subset solve we provide.

5 LOCAL SOLUTIONS TO DIFFUSION PROBLEMS

Many problems in geometry processing can be approached by solving a diffusion problem. The main idea is to simulate the diffusion process from a defined source over the geometry for a fixed time step t. We focus on heat diffusion, which is commonly appearing in the following discrete form in geometry processing. :

$$(\mathbf{M} - t\mathcal{L})\mathbf{x} = \mathbf{b} \tag{4}$$

Where $\mathbf{M}$ represents the mass matrix and $\mathcal{L}$ the cotan-Laplacian. In many cases heat is diffused from a single point, often a single vertex, in which case we have $\mathbf{b} = \mathbf{e}_i$, where $\mathbf{e}_i$ is the canonical basis vector with a single one in the entry with index i. So the right-hand side contains only a single non-zero and, as explained before, the complexity of the first step of the solve drops from at least linear to logarithmic in the number of nodes.

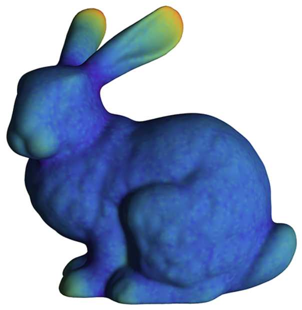
For small time steps, heat will have non-negligible values only in a small neighborhood around the source. In this case it might make sense to also restrict the set of values in the solution. Then the second step of the solve depends on the part of the elimination tree that contains the desired nodes. In our scenario they are close, and in most cases nested dissection makes sure all nodes are contained in a small subtree. Then, altogether, the operation of solving for the heat values reduces from at least quadratic in the size of the mesh to quasi-linear.

In the following we will analyze particular applications in more detail.

The autodiffusion function. The auto-diffusion function $ADF_t(x)$ [Gebal et al. 2009] measures the amount of heat that stays at a source after diffusing for time t. The more general heat kernel signature $HKS(x)$ [Sun et al. 2009] combines several values of the auto-diffusion function over time t. These functions are used to generate descriptors and have a variety of applications, particularly in shape matching.

The standard way to compute the auto-diffusion function or heat kernel signature is based on computing a set of eigenvectors of the discrete Laplace-Beltrami operator. However, it is clear that the value of $ADF_t(\mathbf{v}_i)$ could also be computed by solving heat diffusion for $\mathbf{b} = \mathbf{e}_i$ and considering the result in x_i. If a standard approach to solving the linear equation is used, the solution would provide all values in $\mathbf{x}$. Then computing the auto-diffusion amounts to effectively inverting the diffusion operator, which is more expensive than computing a few eigenvectors. Computing the HKS would mean to compute the inverses for a set of matrices of the form $\mathbf{M} - t\mathcal{L}$ with varying t, which appears not to be attractive.

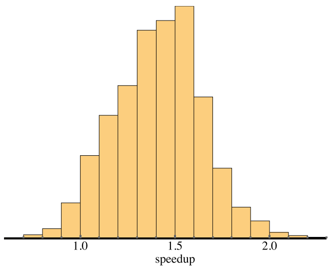
Using our approach, computing the value of the solution only for x_i is a best case scenario. What our approach effectively does is computing only the main diagonal of the inverse operator. This can be done at a fraction of the cost of computing the full inverse. For the HKS, we can exploit the common sparsity structure of all operators: Note that the elimination trees in the factorizations of $\mathbf{M} - t\mathcal{L}$ are independent of t. This means we can effectively solve for the values in one vertex with varying t in parallel, by traversing the elimination tree and computing the values based on different numerical values for the factorizations. This approach becomes particularly attractive if the auto-diffusion values are needed only in a sparse subset of the vertices, because the complexity is trivially linear in the number of desired values (whereas the computation of eigenvectors depends on the mesh size, even if the signatures are evaluated only on a subset of the vertices).



Computing the autodiffusion function using our method on a bunny mesh with 53000 vertices (see inset) took 11 seconds while solving the system each time using classical forward/back substitution would take 3 minutes and 45 seconds. Computing 300 eigenfunctions using ARPACK to compute the autodiffusion function as described in [Gebal et al. 2009] takes 39 seconds.

We wish to stress that computing the solution of the linear equation for only the source vertex is fundamentally different from restricting the computation to a submesh around the vertex i [Herholz et al. 2017]. For small time steps the results might be fairly similar, for larger t values, however, heat diffusion on a small disk like mesh around i will behave fundamentally different from the global solution. The sparse solve computes the exact results no matter what value of t is set. Moreover, the amount of computation is exactly the same for all time steps t.

Pairwise heat exchange. The case of autodiffusion is optimal for the sparse solve. All that is required is to walk down one path in the elimination tree for the forward solve and move back up for the back solve. In many cases we are interested in the distance or heat exchanged by a small set of points, for example when computing embeddings [Panozzo et al. 2013]. If we are interested in how much heat is exchanged by two disjoint sets of points or two single points, the obvious approach is to compute the subtree of the elimination tree with respect to the union of both sets. This can be quite efficient if both sets share most of the subtree. If this is not the case it is beneficial to use different subtrees for the forward and back solve respectively. We implemented this technique and conducted an experiment to assess its benefit. On a mesh we randomly select pairs of points. Depending on how many nodes in the paths traced to the root of the elimination are shared we expect different speedups. For 10^4 pairs on a mesh with 2×10^5 vertices we show a histogram of the achieved speedup (inset). In almost all cases runtime significantly profits from using two separate subtrees. In a small number of cases, where points are quite close on the mesh, setup cost for the second tree outweighs the benefit by a small margin.



Geodesics from diffusion. Crane et al. [2013] observed that heat diffusion can be used to obtain fairly accurate geodesic distances on meshes. The main insight is that the direction of the gradient field of the heat distribution induced by a point source after a small time-step matches the gradient of the geodesic distance function up to scale. Since the length of gradient vectors to geodesic distance is one, the function can be approximated by integrating the normalized gradient field.

Since the right-hand side of (4) is sparse, the forward substitution step can be optimized using the sparse solve. Back substitution and the solution of the second system, however, usually requires a full solve since the results will be generally dense. However, if distances are needed only in a small region around the source, we can compute these values at a cost that is dominated by the size of the neighborhood rather then the full mesh. This can be useful especially if one is interested in computing the vertex that is closest to a number of points, such as in constructing Voronoi diagrams.

6 DISCUSSION & CONCLUSIONS

Geometry processing applications commonly require solving symmetric, positive-definite linear systems $\mathbf{Ax} = \mathbf{b}$. In many situations either the right-hand side $\mathbf{b}$ and/or the set of solution values of interest in $\mathbf{x}$ corresponds to a small *and* localized set of vertices. We have shown how to systematically exploit the resulting sparsity of $\mathbf{x}$ and $\mathbf{b}$ when the Cholesky factorization of $\mathbf{A}$ is provided without compromising on numerical accuracy. As a result of the nested dissection reordering strategy, the speedup over a full solve can be significant when the sparse sets correspond to spatially close vertices. Even if this is not the case, improved solving times can usually be observed. Due to the small overhead, our strategy will exhibit nearly identical running times as the full solve not exploiting sparsity in the worst case. We have demonstrated the efficiency of local solutions on representative applications in geometry processing. For these we find that using sparse subset solves on prefactored system matrices can improve the runtimes by orders of magnitude.

Localization is not always straightforward. For example, fixing a subset of arbitrary vertices (such as in many modeling scenarios) requires updating the system matrix, which may not be very efficient [Davis et al. 2016]. We are currently investigating if the update or downdate of a matrix can be localized, thus also exploiting locality for these types of operations.

REFERENCES

Marc Alexa and Max Wardetzky. 2011. Discrete Laplacians on General Polygonal Meshes. In *ACM SIGGRAPH 2011 Papers (SIGGRAPH '11)*. ACM, New York, NY, USA, Article 102, 10 pages. https://doi.org/10.1145/1964921.1964997
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

Isaac Chao, Ulrich Pinkall, Patrick Sanan, and Peter Schröder. 2010. A Simple Geometric Model for Elastic Deformations. *ACM Trans. Graph.* 29, 4, Article 38 (July 2010), 6 pages. https://doi.org/10.1145/1778765.1778775

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

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.

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

Matthias Eck, Tony DeRose, Tom Duchamp, Hugues Hoppe, Michael Lounsbery, and Werner Stuetzle. 1995. Multiresolution Analysis of Arbitrary Meshes. In *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '95).* ACM, New York, NY, USA, 173–182. https://doi.org/10.1145/218380.218440

Sharif Elcott and Peter Schröder. 2006. Building Your Own DEC at Home. In *ACM SIGGRAPH 2006 Courses (SIGGRAPH '06).* ACM, New York, NY, USA, 55–59. https://doi.org/10.1145/1185657.1185666

K Gebal, Jakob Andreas Bærentzen, Henrik Aanæs, and Rasmus Larsen. 2009. Shape analysis using the auto diffusion function. In *Computer Graphics Forum*, Vol. 28. Wiley Online Library, 1405–1413.

Alan George. 1973. Nested Dissection of a Regular Finite Element Mesh. *SIAM J. Numer. Anal.* 10, 2 (1973), 345–363. https://doi.org/10.1137/0710032

A. George and J. W. H. Liu. 1978. An Automatic Nested Dissection Algorithm for Irregular Finite Element Problems. *SIAM J. Numer. Anal.* 15, 5 (1978), 1053–1069. http://dx.doi.org/10.1137/0715069

J. R. Gilbert and T. Peierls. 1988. Sparse Partial Pivoting in Time Proportional to Arithmetic Operations. *SISC* 9, 5 (1988), 862–874. http://dx.doi.org/10.1137/0909058

G.H. Golub and C.F. Van Loan. 2013. *Matrix Computations.* Johns Hopkins University Press.

Philipp Herholz, Felix Haase, and Marc Alexa. 2017. Diffusion Diagrams: Voronoi Cells and Centroids from Diffusion. *Computer Graphics Forum* 36, 2 (2017), 163–175. https://doi.org/10.1111/cgf.13116

Ilse C. F. Ipsen. 1997. Computing an Eigenvector with Inverse Iteration. *SIAM Rev.* 39, 2 (1997), 254–291. https://doi.org/10.1137/S0036144596300773

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.

Bruno Lévy, Sylvain Petitjean, Nicolas Ray, and Jérome Maillot. 2002. Least Squares Conformal Maps for Automatic Texture Atlas Generation. *ACM Trans. Graph.* 21, 3 (July 2002), 362–371. https://doi.org/10.1145/566654.566590

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

Ligang Liu, Lei Zhang, Yin Xu, Craig Gotsman, and Steven J. Gortler. 2008. A Local/Global Approach to Mesh Parameterization. *Computer Graphics Forum* 27, 5 (2008), 1495–1504. https://doi.org/10.1111/j.1467-8659.2008.01290.x

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

Daniele Panozzo, Ilya Baran, Olga Diamanti, and Olga Sorkine-Hornung. 2013. Weighted Averages on Surfaces. *ACM Trans. Graph.* 32, 4, Article 60 (July 2013), 12 pages. https://doi.org/10.1145/2461912.2461935

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.

Jian Sun, Maks Ovsjanikov, and Leonidas Guibas. 2009. A Concise and Provably Informative Multi-Scale Signature Based on Heat Diffusion. *Computer Graphics Forum* 28, 5 (2009), 1383–1392. https://doi.org/10.1111/j.1467-8659.2009.01515.x