Efficient Linear System Solvers for Mesh Processing

Mario Botsch, David Bommes, and Leif Kobbelt

Computer Graphics Group,
RWTH Aachen Technical University

Abstract. The use of polygonal mesh representations for freeform geometry enables the formulation of many important geometry processing tasks as the solution of one or several linear systems. As a consequence, the key ingredient for efficient algorithms is a fast procedure to solve linear systems. A large class of standard problems can further be shown to lead more specifically to sparse, symmetric, and positive definite systems, that allow for a numerically robust and efficient solution.
In this paper we discuss and evaluate the use of *sparse direct solvers* for such kind of systems in geometry processing applications, since in our experiments they turned out to be superior even to highly optimized multigrid methods, but at the same time were considerably easier to use and implement. Although the methods we present are well known in the field of high performance computing, we observed that they are in practice surprisingly rarely applied to geometry processing problems.

1 Introduction

In the field of geometry processing suitable data structures that enable the implementation of efficient algorithms are getting more and more important [23], especially since the complexity of the geometric models to be processed is growing much faster than the steadily increasing computational power and available memory of today's PC systems. Typical examples are higher order spline surfaces $\mathbf{f}(u,v) = \sum_i \mathbf{c}_i \Phi_i(u,v)$, represented as a weighted average of control points $\mathbf{c}_i$, or piecewise linear triangle meshes $\mathcal{M}$ obtained from sampling a surface at the mesh vertices $\mathbf{x}_i = \mathbf{f}(u_i, v_i)$.

Using finite differences or finite elements, many standard geometry processing problems, like for instance PDEs on or of surfaces, can be formulated as a set of (linear or non-linear) equations in either the control points $\mathbf{c}_i$ of a spline surface or the vertex positions $X = (\mathbf{x}_1, \ldots, \mathbf{x}_n)^T \in \mathbb{R}^{n\times 3}$ of an approximating triangle mesh.

A common technique to efficiently handle non-linear problems is their decomposition into a sequence of linear ones, like, e.g., the (semi-)implicit integration of non-linear geometric flows by solving a linear equation in each time step [13] or the Levenberg-Marquardt method for non-linear optimization [17]. Similarly, continuous energy functionals $E(\mathbf{f}) = \int_\Omega e(\mathbf{f}, \mathbf{x})\, d\mathbf{x}$ are approximated by quadratic forms $E(X) = X^T Q X$, such that their minimizer surfaces can

efficiently be derived by solving the linear systems $QX = B$, assuming proper boundary constraints B [22]. These examples motivate why a large class of geometric problems comes down to the solution of one or several linear systems. As a consequence, high performance linear system solvers are of major importance for the development of efficient algorithms.

Since differential surface properties are defined locally, the discretization of PDEs typically leads to *sparse* linear systems, in which the ith row contains non-zeros only in those entries corresponding to the geodesic or topological neighborhood of vertex $\mathbf{x}_i$. We are therefore interested in solvers that exploit this sparsity in order to minimize both memory consumption and computation times.

Within the class of sparse linear systems, we will further concentrate on symmetric positive definite (so-called *spd*) matrices, since exploiting their special structure allows for the most efficient and most robust implementations. Such systems frequently occur when minimizing energy functionals of the form $E(X) = X^T Q X$ with an spd matrix Q. A very popular source of spd systems is the discrete Laplace-Beltrami operator $\Delta_{\mathcal{S}}$ [30], which is closely related to frequencies of scalar fields defined on a two-manifold surface $\mathcal{S}$ [41]. This operator has various applications in surface smoothing [13, 41], surface parameterization [32, 12], variational surface modeling [25, 7, 38, 44], mesh morphing [3, 4, 43], and shape analysis [31]. Besides from surfaces, the standard Laplace operator is for instance also used in image editing [40] and fluid simulation [39]. Finally, all linear problems $A\mathbf{x} = \mathbf{b}$ that cannot be solved exactly and hence are approximated in the least squares sense by using the normal equations $A^T A\mathbf{x} = A^T\mathbf{b}$ also result in spd linear systems [26]. This large but still incomplete list of applications involving spd systems legitimates focusing on this special class of problems.

Another important point to be considered is whether the linear systems are solved just once or several times, e.g., for different right-hand sides. Since most geometric problems are separable w.r.t. the coordinate components, they can be solved component-wise for x, y, and z using the same system matrix. Multiple right-hand side problems also naturally occur in applications where the user interactively changes boundary constraints, e.g., in surface editing.

There is another situation for solving a sequence of similar systems: when decomposing a non-linear problem into a sequence of linear systems, the values of the matrix entries usually change in each iteration, but its structure, i.e., the pattern of non-zero elements $\{(i,j) \mid A_{ij} \neq 0\}$, stays the same, because it usually depends on the mesh connectivity, only which does not change. In both cases — solving for multiple right-hand sides or matrices of identical structure — this additional information should be exploited as much as possible, e.g., by investing pre-computation time in some kind of factorization or preconditioning.

In this paper we propose the use of direct solvers for the sparse spd systems, as they arise from typical computer graphics and geometry processing problems. We mainly focus on Laplacian or bi-Laplacian systems for triangle meshes, however, analogous results hold for systems of similar structure. The size of the linear systems corresponds to the number of vertices in the mesh, which, in our context, usually is of the order of 10^4 or 10^5. Due to the local definition of the Laplace

operator, the resulting matrices are highly sparse and in the average exhibit 7 or 19 non-zero entries per row for Laplacian or bi-Laplacian systems, respectively. Since in many applications these systems have to be solved for multiple right-hand sides, the sparse factorizations of direct solvers allow for highly efficient implementations. After reviewing the commonly known and widely used direct and iterative solvers, we introduce sparse direct solvers and point out their advantageous properties in Sect. 2. After comparing the different solvers in Sect. 3 we finally present a list of applications that greatly benefit from sparse direct solvers in Sect. 4.

2 Linear System Solvers

We describe and compare the following classes of solvers: dense direct solvers, iterative solvers, multigrid solvers, and finally sparse direct solvers. For the following discussion we restrict to sparse spd problems $A\mathbf{x} = \mathbf{b}$, with $A = A^T \in \mathbb{R}^{n\times n}$, $\mathbf{x}, \mathbf{b} \in \mathbb{R}^n$, and denote by $\mathbf{x}^*$ the exact solution $A^{-1}\mathbf{b}$. For completeness, the general case of a non-symmetric indefinite system is outlined in Sect. 2.5. More elaborate surveys on how to efficiently solve general large linear systems can be found in the books [10, 29].

2.1 Dense Direct Solvers

Direct linear system solvers are based on a factorization of the matrix A into matrices of simpler structure, e.g., triangular, diagonal, or orthogonal matrices. This structure allows for an efficient solution of the factorized system. As a consequence, once the factorization is computed, it can be used to solve the linear system for several different right-hand sides.

The most commonly used examples for *general* matrices A are, in the order of increasing numerical robustness and computational effort, the LU factorization, QR factorization, or the singular value decomposition. However, in the special case of a spd matrix the Cholesky factorization $A = LL^T$, with L denoting a lower triangular matrix, should be employed, since it exploits the symmetry of the matrix and can additionally be shown to be numerically very robust due to the positive definiteness of the matrix A [18].

On the downside, the asymptotic time complexity of all dense direct methods is $O(n^3)$ for the factorization and $O(n^2)$ for solving the system based on the precomputed factorization. Since for the problems we are targeting at, n can be of the order of 10^5, the total cubic complexity of dense direct methods is prohibitive. Even if the matrix A is highly sparse, the naïve direct methods enumerated here are not designed to exploit this structure, hence the factors are dense matrices in general (cf. Fig. 2, top row, on page 9).

2.2 Iterative Solvers

In contrast to dense direct solvers, iterative methods are able to exploit the sparsity of the matrix A. Since they additionally allow for a simple implementation

[33], iterative solvers are the de-facto standard method for solving sparse linear systems in the context of geometric problems. A detailed overview of iterative methods with precious implementation hints can be found in [5, 36].

Iterative methods compute a converging sequence $\mathbf{x}^{(0)}, \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(i)}$ of approximations to the solution $\mathbf{x}^*$ of the linear system, i.e., $\lim_{i\to\infty} \mathbf{x}^{(i)} = \mathbf{x}^*$. In practice, however, one has to find a suitable criterion to stop the iteration as soon as the current solution $\mathbf{x}^{(i)}$ is accurate enough, i.e., if the norm of the error $\mathbf{e}^{(i)} := \mathbf{x}^* - \mathbf{x}^{(i)}$ is less than some ε. Since the solution $\mathbf{x}^*$ is not known beforehand, the error has to be estimated by considering the residual $\mathbf{r}^{(i)} := A\mathbf{x}^{(i)} - \mathbf{b}$. These two are related by the *residual equations* $A\mathbf{e}^{(i)} = \mathbf{r}^{(i)}$, leading to an upper bound $\left\|\mathbf{e}^{(i)}\right\| \leq \left\|A^{-1}\right\| \cdot \left\|\mathbf{r}^{(i)}\right\|$, i.e., the norm of the inverse matrix has to be estimated or approximated in some way (see [5]).

The simplest examples for iterative solvers are the Jacobi and Gauss-Seidel methods. They belong to the class of static iterative methods, whose update steps can be written as $\mathbf{x}^{(i+1)} = M\mathbf{x}^{(i)} + \mathbf{c}$ with constant M and $\mathbf{c}$, such that the solution $\mathbf{x}^*$ is the fixed point of this iteration. An analysis of the eigenstructure of the update matrices M reveals that both methods rapidly remove the high frequencies of the error, but the iteration stalls if the error is a smooth function. By consequence, the convergence to the exact solution $\mathbf{x}^*$ is usually too slow in practice. As an additional drawback these methods only converge for a restricted set of matrices, e.g., for diagonally dominant ones.

Non-stationary iterative solvers are more powerful, and for spd matrices the method of conjugate gradients (CG) [20, 18] is suited best, since it provides guaranteed convergence with monotonically decreasing error. For a spd matrix A the solution of $A\mathbf{x} = \mathbf{b}$ is equivalent to the minimization of the quadratic form

$$\phi(\mathbf{x}) := \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{b}^T\mathbf{x} \ .$$

The CG method successively minimizes this functional along a set of linearly independent search directions $\mathbf{p}^{(i)}$, such that

$$\mathbf{x}^{(i)} = \operatorname{argmin}\left\{\phi(\mathbf{x}) \;\middle|\; \mathbf{x} \in \mathbf{x}_0 + \operatorname{span}\left\{\mathbf{p}^{(1)}, \ldots, \mathbf{p}^{(i)}\right\}\right\} \ .$$

Due to the nestedness of these spaces the error decreases monotonically, and the exact solution $\mathbf{x}^* \in \mathbb{R}^n$ is found after at most n steps (neglecting rounding errors). Minimizing ϕ by gradient descent results in inefficient zigzag paths in steep valleys of ϕ, which correspond to strongly differing eigenvalues of A. In order to cancel out the effect of A's eigenvalues on the search directions $\mathbf{p}_i$, those are chosen to be *A-conjugate*, i.e., orthogonal w.r.t. the scalar product induced by A: $\mathbf{p}_j^T A\mathbf{p}_i = 0$ for $i \neq j$ [37]. The computation of and minimization along these optimal search directions can be performed efficiently and with a constant memory consumption.

The complexity of each CG iteration is mainly determined by the matrix-vector product $A\mathbf{x}$, which is of the order $O(n)$ if the matrix is sparse. Given the maximum number of n iterations, the total complexity is $O(n^2)$ in the worst case, but it is usually better in practice.

As the convergence rate mainly depends on the spectral properties of the matrix A, a proper pre-conditioning scheme should be used to increase the efficiency and robustness of the iterative scheme. This means that a slightly different system $\tilde{A}\tilde{\mathbf{x}} = \tilde{\mathbf{b}}$ is solved instead, with $\tilde{A} = PAP^T$, $\tilde{\mathbf{x}} = P^{-T}\mathbf{x}$, $\tilde{\mathbf{b}} = P\mathbf{b}$, where the regular pre-conditioning matrix P is chosen such that $\tilde{A}$ is well conditioned [18, 5]. However, the matrix P is restricted to have a simple structure, since an additional linear system $P\mathbf{z} = \mathbf{r}$ has to be solved in each iteration of the solver.

The iterative conjugate gradients method manages to decrease the computational complexity from $O(n^3)$ to $O(n^2)$ for sparse matrices, but this is still too slow to compute exact (or sufficiently accurate) solutions of large linear systems, in particular if the systems are numerically ill-conditioned, like for instance the higher order Laplacian systems used in variational surface modeling [25, 7].

2.3 Multigrid Iterative Solvers

As mentioned in the last section, one characteristic problem of most iterative solvers is that they are *smoothers*: they attenuate the high frequencies of the error $\mathbf{e}^{(i)}$ very fast, but their convergence stalls if the error is a smooth function. This fact is exploited by multigrid methods, that build a fine-to-coarse hierarchy $\{\mathcal{M} = \mathcal{M}_0, \mathcal{M}_1, \ldots, \mathcal{M}_k\}$ of the computation domain $\mathcal{M}$ and solve the linear system hierarchically from coarse to fine [19, 8].

After a few (pre-)smoothing iterations on the finest level $\mathcal{M}_0$ the high frequencies of the error are removed and the solver becomes inefficient. However, the remaining low frequency error $\mathbf{e}_0 = \mathbf{x}^* - \mathbf{x}_0$ on $\mathcal{M}_0$ corresponds to higher frequencies when restricted to the coarser level $\mathcal{M}_1$ and therefore can be removed efficiently on $\mathcal{M}_1$. Hence the error is solved for using the residual equations $A\mathbf{e}_1 = \mathbf{r}_1$ on $\mathcal{M}_1$, where $\mathbf{r}_1 = R_{0\to1}\mathbf{r}_0$ is the residual on $\mathcal{M}_0$ transfered to $\mathcal{M}_1$ by a restriction operator $R_{0\to1}$. The result is prolongated back to $\mathcal{M}_0$ by $\mathbf{e}_0 \leftarrow P_{1\to0}\mathbf{e}_1$ and used to correct the current approximation: $\mathbf{x}_0 \leftarrow \mathbf{x}_0 + \mathbf{e}_0$. Small high-frequency errors due to the prolongation are finally removed by a few post-smoothing steps on $\mathcal{M}_0$. The recursive application of this two-level approach to the whole hierarchy can be written as

$$\Phi_i \;=\; S_\mu \, P_{i+1\to i} \, \Phi_{i+1} \, R_{i\to i+1} \, S_\lambda \;\;,$$

with λ and μ pre- and post-smoothing iterations, respectively. One recursive run is known as a *V-cycle* iteration.

Another concept is the method of *nested iterations*, that exploits the fact that iterative solvers are very efficient if the starting value is sufficiently close to the actual solution. One starts by computing the exact solution on the coarsest level $\mathcal{M}_k$, which can be done efficiently since the system $A_k\mathbf{x}_k = \mathbf{b}_k$ corresponding to the restriction to $\mathcal{M}_k$ is small. The prolongated solution $P_{k\to k-1}\mathbf{x}_k^*$ is then used as starting value for iterations on $\mathcal{M}_{k-1}$, and this process is repeated until the finest level $\mathcal{M}_0$ is reached and the solution $\mathbf{x}_0^* = \mathbf{x}^*$ is computed.

The remaining question is how to iteratively solve on each level. The standard method is to use one or two V-cycle iterations, leading to the so-called *full*

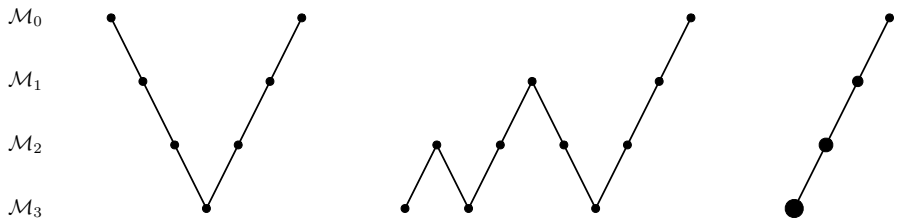


Fig. 1. A schematic comparison in terms of visited multigrid levels for V-cycle (*left*), full multigrid with one V-cycle per level (*center*), and cascading multigrid (*right*). The size of the dots represents the number of iterations on the respective level.

multigrid method. However, one can also use an iterative smoothing solver (e.g., Jacobi or CG) on each level and completely avoid V-cycles. In the latter case the number of iterations m_i on level i must not be constant, but instead has to be chosen as $m_i = m\,\gamma^i$ to decrease exponentially from coarse to fine [6]. Besides the easier implementation, the advantage of this *cascading multigrid* method is that once a level is computed, it is not involved in further computations and can be discarded. A comparison of the three methods in terms of visited multigrid levels is given in Fig. 1.

Due to the logarithmic number of hierarchy levels $k = O(\log n)$ the full multigrid method and the cascading multigrid method can both be shown to have linear asymptotic complexity, as opposed to quadratic for non-hierarchical iterative methods. However, they cannot exploit synergy for multiple right-hand sides, which is why factorization-based approaches are clearly preferable in such situations, as we will show in the next section.

Since in our case the discrete computational domain $\mathcal{M}$ is an irregular triangle mesh instead of a regular 2D or 3D grid, the coarsening operator for building the hierarchy is based on mesh decimation techniques [24, 14]. The shape of the resulting triangles is important for numerical robustness, and the edge lengths on the different levels should mimic the case of regular grids. Therefore the decimation usually removes edges in the order of increasing lengths, such that the hierarchy levels have uniform edge lengths and triangles of bounded aspect ratio.

The simplification from one hierarchy level $\mathcal{M}_i$ to the next coarser one $\mathcal{M}_{i+1}$ should additionally be restricted to remove a *maximally independent set* of vertices, i.e., no two removed vertices $v_j, v_l \in \mathcal{M}_i \setminus \mathcal{M}_{i+1}$ are connected by an edge $e_{jl} \in \mathcal{M}_i$. In [2] some more efficient alternatives to this standard Dobkin-Kirkpatric hierarchy are discussed. In order to achieve higher performance, we do not change the simple way the hierarchy is constructed, but instead solve the linear system on every second or third level only, and use the prolongation operator alone on all in-between levels.

The linear complexity of multi-grid methods allows for the highly efficient solution even of very complex systems. However, the main problem of these solvers is their quite involved implementation, since special care has to be taken for the

hierarchy building, for special multigrid pre-conditioners, and for the inter-level conversion by restriction and prolongation operators. A detailed overview of these techniques is given in [2].

Additionally, the number of iterations per hierarchy level have to be chosen: This includes the number of V-cycles and pre- and post-smoothing iterations per V-cycle for the full multigrid method, or m and γ for the cascading multigrid approach. These numbers have to be chosen either by heuristic or experience, since they not only depend on the problem (structure of A), but also on its specific instance (values of A). Nevertheless, if iterative solvers are to be used, multigrid methods are the only way to achieve acceptable computing times for solving large systems, as has been shown in [25, 34, 2].

2.4 Sparse Direct Solvers

The use of direct solvers for large sparse linear systems is often neglected, since naïve direct methods have complexity $O(n^3)$, as described above. The problem is that even when the matrix A is sparse, the factorization will not preserve this sparsity, such that the resulting Cholesky factor is a dense lower triangular matrix (cf. Fig. 2, top row).

However, an analysis of the factorization process reveals that a *band-limitation* of the matrix A will be preserved. Following [15], we define the bandwidth $\beta(A)$ in terms of the bandwidth of its ith row

$$\beta(A) := \max_{1 \le i \le n} \{\beta_i(A)\} \quad \text{with} \quad \beta_i(A) := i - \min_{1 \le j \le i} \{j \mid A_{ij} \neq 0\} \ .$$

If the matrix A has bandwidth $\beta(A)$ then so has its factor L. An even stricter bound is that also the so-called *envelope* or *profile*

$$\mathrm{Env}(A) := \{(i,j) \mid 0 < i - j \le \beta_i(A)\}$$

is preserved, i.e., no additional non-zeros (so-called *fill-in* elements) are generated outside the envelope.

This additional structure can be exploited in both the factorization and the solution process, such that their complexities reduce from $O(n^3)$ and $O(n^2)$ to linear complexity in the number of non-zeros $\mathrm{nz}(A)$ of A [15]. Since usually $\mathrm{nz}(A) = O(n)$, this is the same linear complexity as for multigrid solvers. However, in the graphics-related examples we will show in the following sections, sparse direct method turned out to be more efficient compared to multigrid methods, in particular for multiple right-hand side problems.

Since we assume the matrices to be sparse, but not band-limited or profile-optimized, the first step is to minimize the matrix envelope, which can be achieved by symmetric row and column permutations $A \mapsto P^T A P$ using a permutation matrix P, i.e., a re-ordering of the mesh vertices. Although this re-ordering problem is NP complete, several good heuristics exist, of which we will present the most commonly used in the following. All of these methods work on the undirected *adjacency graph* $\mathrm{Adj}(A)$ corresponding to the non-zeros of A, i.e., two nodes $i, j \in \{1, \ldots, n\}$ are connected by an edge if and only if $A_{ij} \neq 0$.

The standard method for envelope minimization is the *Cuthill-McKee* algorithm [9], that picks a start node and renumbers all its neighbors by traversing the adjacency graph in a breadth-first manner, using a greedy selection in the order of increasing valence. It has further been proven in [28] that reverting this permutation leads to better re-orderings, such that usually the *reverse Cuthill-McKee* method (RCMK) is employed. The result $P^T AP$ of this matrix re-ordering is depicted in the second row of Fig. 2.

Since no special pivoting is required for the Cholesky factorization, the non-zero structure of its matrix factor L can symbolically be derived from the non-zero structure of the matrix A alone, or, equivalently, from its adjacency graph. The graph interpretation of the Cholesky factorization is to successively eliminate the node with the lowest index from the graph and connect all its immediate neighbors mutually to each other. The additional edges e_{ij} generated in this so-called *elimination graph* correspond to the fill-in elements $L_{ij} \neq 0 = A_{ij}$.

In order to minimize fill-in the *minimum degree* algorithm (MD) and its variants [16, 27] remove the nodes with smallest valence first from the elimination graph, since this causes the least number of additional pairwise connections. Many efficiency optimizations of this basic method exist, the most prominent of which is the *super-nodal* approach: instead of removing eliminated nodes from the graph, neighboring eliminated nodes are clustered to so-called super-nodes, allowing for more efficient graph updates. The resulting minimum degree re-orderings do not lead to some kind of a band-structure (which implicitly limits fill-in), but instead directly minimize the fill-in of L (cf. Fig. 2, third row).

The last class of re-ordering approaches is based on graph partitioning. Consider a matrix A whose adjacency graph has m separate connected components. Such a matrix can be restructured to a block-diagonal matrix of m blocks, such that the factorization can be performed on each block individually. If the adjacency graph is connected, a small subset S of nodes, whose elimination would separate the graph into two components of roughly equal size, is found by one of several heuristics [21]. This graph-partitioning results in a matrix consisting of two large diagonal blocks (two connected components) and $|S|$ rows representing their connection (separator S). Recursively repeating this process leads to the method of *nested dissection* (ND), leading to matrices of the typical block structure shown in the bottom row of Fig. 2. Besides the obvious fill-in reduction, these systems also allow for easy parallelization of both the factorization and the solution.

For the comparison of the different matrix re-ordering strategies a rather small matrix was used in Fig. 2 to allow for clearer visualization. On an analogous $5k \times 5k$ matrix the number of non-zeros $\mathrm{nz}(L)$ decreases from 2.3M to 451k, 106k, and 104k by applying the RCMK, MD, and ND method, respectively. The timings to obtain those re-orderings are 17ms, 12ms, and 38ms. It can further be observed that for larger systems the nested dissection method [21] generally leads to the best results.

One important advantage of the Cholesky factorization is that the non-zero structure of the factor L can be determined from $\mathrm{Adj}(A)$ without any numerical

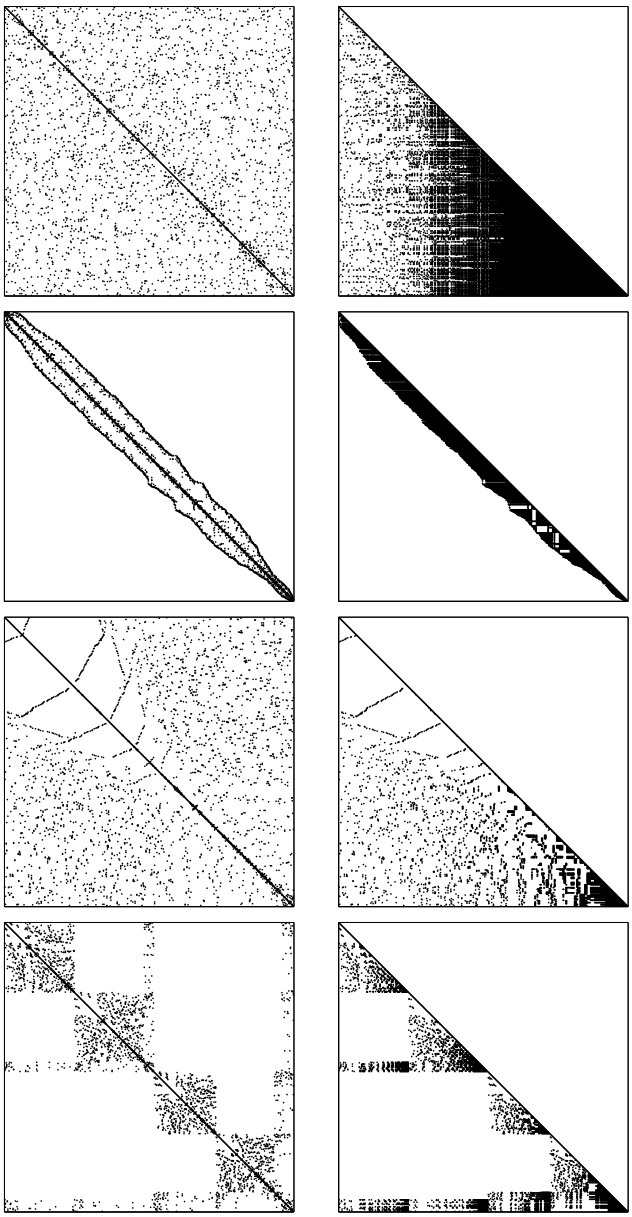


Fig. 2. The top row shows the non-zero pattern of a typical 500×500 matrix A and its Cholesky factor L, corresponding to a Laplacian system on a triangle mesh. Although A is highly sparse (3502 non-zeros), the factor L is dense (36k non-zeros). The reverse Cuthill-McKee algorithm minimizes the envelope of the matrix, resulting in 14k non-zeros of L (*2nd row*). The minimum degree ordering avoids fill-in during the factorization, which decreases the number of non-zeros to 6203 (*3rd row*). The last row shows the result of a nested dissection method (7142 non-zeros), that allows for parallelization due to its block structure.

computations. This allows us to setup an efficient *static* data structure for L *before* the actual *numerical factorization*, which is therefore called *symbolic factorization*. Since suitable data structures and proper memory layout are crucial for efficient numerical computations, this two-step factorization process allows for significant optimizations.

Analogously to the dense direct solvers, the factorization can be exploited to solve for different right-hand sides in a very efficient manner. In addition to this, whenever the matrix A is changed, such that its non-zero structure $\mathrm{Adj}(A)$ is preserved, then the matrix re-ordering as well as the symbolic factorization can obviously be re-used. Solving the modified system therefore only requires to re-compute the numerical factorization and performing the back-substitution, which typically saves about 50% of the total computation time for solving the modified system. As we will show in Sect. 4, this allows for an efficient implementation of a large class of algorithms that decompose a non-linear problem into a sequence of similar linear ones, like for instance the implicit fairing approach [13] or the Levenberg-Marquardt optimization for non-linear problems [33, 17].

Another advantage of sparse direct methods is that no additional parameters have to be chosen in a problem-dependent manner, as for instance the different numbers of iterations for the multigrid solvers. The only degree of freedom is the matrix re-ordering, but this only depends on the symbolic structure of the problem and therefore can be chosen quite easily. For more details and implementation notes the reader is referred to the book of George and Liu [15]; a highly efficient implementation is publicly available in the TAUCS library [42].

2.5 Non-Symmetric Indefinite Systems

When the assumptions about the symmetry and positive definiteness of the matrix A are not satisfied, optimal methods like the Cholesky factorization or conjugate gradients cannot be used. In this section we shortly outline which techniques are applicable instead.

From the class of iterative solvers the bi-conjugate gradients algorithm (BiCG) is typically used as a replacement of the conjugate gradients method [33]. Although working well in most cases, BiCG does not provide any theoretical convergence guarantees and has a very irregular non-monotonically decreasing residual error for ill-conditioned systems. On the other hand, the GMRES method converges monotonically with guarantees, but its computational cost and memory consumption increase in each iteration [18]. As a good trade-off, the stabilized Bi-CGSTAB [5] represents a mixture between the efficient BiCG and the smoothly converging GMRES; it provides a much smoother convergence and is reasonably efficient and easy to implement.

When considering dense direct solvers, the Cholesky factorization cannot be used for general matrices. Therefore the LU factorization is typically employed (instead of QR or SVD), since it is similarly efficient and also extends well to sparse direct methods. However, (partial) row and column pivoting is essential for the numerical robustness of the LU factorization, since this avoids zeros on the diagonal during the factorization process.

Similarly to the Cholesky factorization, it can be shown that the LU factorization also preserves the band-width and envelope of the matrix A. Techniques like the minimum degree algorithm generalize to non-symmetric matrices as well. But as for dense matrices, the banded LU factorization relies on partial pivoting in order to guarantee numerical stability. In this case, two competing types of permutations are involved: symbolic permutations for matrix re-ordering and pivoting permutations ensuring numerical robustness. As these permutations cannot be handled separately, a trade-off between stability and fill-in minimization has to be found, resulting in a significantly more complex factorization process.

As a consequence, the re-ordering depends on the numerical values of the matrix entries, such that an exact symbolic factorization like in the Cholesky case is not possible. In order to nevertheless be able to setup a static data structure, a more conservative envelope is typically used, such that pivoting within this structure is still possible. A highly efficient implementation of a sparse LU factorization is provided by the SuperLU library [11].

3 Laplace Systems

Most of the example applications shown in Sect. 4 require the solution of linear Laplacian systems, therefore we analyze these matrices and compare different solvers for their solution. Although we focus on Laplacian systems, we will see in Sect. 4 that analogous results hold for matrices of similar structure, like for instance sparse least squares systems.

The discrete Laplace-Beltrami operator $\Delta_{\mathcal{S}} f$ of a scalar-valued function f on the manifold $\mathcal{S}$ [13, 30, 32] is defined for a center vertex v_i as a linear combination with its one-ring neighbors $v_j \in N_1(v_i)$:

$$\Delta_{\mathcal{S}} f(v_i) \;=\; \frac{2}{A(v_i)} \sum_{v_j \in N_1(v_i)} \left(\cot\alpha_{ij} + \cot\beta_{ij}\right)\left(f(v_j) - f(v_i)\right) \quad ,$$

where $\alpha_{ij} = \angle(\mathbf{x}_i, \mathbf{x}_{j-1}, \mathbf{x}_j)$, $\beta_{ij} = \angle(\mathbf{x}_i, \mathbf{x}_{j+1}, \mathbf{x}_j)$, and $\mathbf{x}_i$ represents the 3D position of the mesh vertex v_i. The normalization factor $A(v_i)$ denotes the Voronoi area around the vertex v_i [30]. In matrix notation the vector of the Laplacians of $f(v_i)$ can be written as

$$\begin{pmatrix} \vdots \\ \Delta_{\mathcal{S}} f(v_i) \\ \vdots \end{pmatrix} \;=\; D \cdot M \cdot \begin{pmatrix} \vdots \\ f(v_i) \\ \vdots \end{pmatrix} \quad ,$$

where D is a diagonal matrix containing the normalization factors $D_{ii} = 2/A(v_i)$, and M is a symmetric matrix of cotangent weights with

$$M_{ij} \;=\; \begin{cases} 0 & i \neq j\,,\ v_j \notin N_1(v_i) \\ \cot\alpha_{ij} + \cot\beta_{ij}, & i \neq j\,,\ v_j \in N_1(v_i) \\ -\sum_{v_j \in N_1(v_i)} \left(\cot\alpha_{ij} + \cot\beta_{ij}\right) & i = j \end{cases} \quad .$$

Since the Laplacian of a vertex v_i is defined *locally* in terms of its one-ring neighbors, the matrix M is highly sparse and has non-zeros in the ith row only on the diagonal and in those columns corresponding to v_i's one-ring neighbors $N_1(v_i)$. Due to the Euler characteristic for triangle meshes, this results in about 7 non-zeros per row in average. Analogously, higher order Laplacian matrices $\Delta_{\mathcal{S}}^k$ have non-zeros for the k-ring neighbors $N_k(v_i)$, which are, e.g., about 19 for bi-Laplacian systems ($k = 2$).

For a closed mesh without boundaries, Laplacian systems $\Delta_{\mathcal{S}}^k \mathbf{x} = \mathbf{b}$ of any order k can be turned into symmetric ones by moving the first diagonal matrix factor D to the right-hand side:

$$M(DM)^{k-1}\mathbf{x} \;=\; D^{-1}\mathbf{b} \;.$$

Boundary constraints are typically employed by restricting the positions of certain vertices, which corresponds to eliminating their respective rows and columns of the left-hand side and hence keeps the matrix symmetric. The case of meshes with boundaries is equivalent to a patch bounded by constrained vertices and therefore also results in a symmetric matrix. Pinkall and Polthier [32] additionally showed that this system is positive definite, such that we can apply the efficient solvers presented in the last section.

In the following we compare the different kinds of linear system solvers for Laplacian as well as for bi-Laplacian systems. All timings we report in this and the next section were taken on a 3.0GHz Pentium4 running Linux. The iterative solver from the `gmm++` library [35] is based on the conjugate gradients method and uses an incomplete LDL^T factorization as preconditioner. Our cascading multigrid solver performs preconditioned conjugate gradient iterations on each hierarchy level and additionally exploits SSE instructions in order to solve for up to four right-hand sides simultaneously. The direct solver of the TAUCS library [42] employs nested dissection re-ordering and a sparse complete Cholesky factorization. Although our linear systems are symmetric, we also compare to the popular SuperLU solver [11], which is based on a sparse LU factorization, for the sake of completeness.

Iterative solvers have the advantage over direct ones that the computation can be stopped as soon as a sufficiently small error is reached, which — in typical computer graphics applications — does not have to be the highest possible precision. In contrast, direct methods always compute the exact solution up to numerical round-off errors, which in our application examples actually was more precise than required. The stopping criteria of the iterative methods have therefore been chosen to yield sufficient results, such that their quality is comparable to that achieved by direct solvers. The resulting residual errors were allowed to be about one order of magnitude larger than those of the direct solvers. While the latter achieved an average residual error of 10^{-7} and 10^{-5} for Laplacian and bi-Laplacian systems, respectively, the iterative solvers were stopped at an error of 10^{-6} and 10^{-4}.

Table 1 shows timings for the different solvers on Laplacian systems $\Delta_{\mathcal{S}} X = B$ of 10k to 50k and 100k to 500k unknowns, i.e., free vertices X. For each solver three columns of timings are given:

Setup: Computing the cotangent weights for the Laplace discretization and building the matrix structure (done per-level for the multigrid solver).

Precomputation: Preconditioning (*iterative*), building the hierarchy by mesh decimation (*multigrid*), matrix re-ordering and sparse factorization (*direct*).

Solution: Solving the linear system for three different right-hand sides corresponding to the x, y, and z components of the free vertices X.

Due to its effective preconditioner, which computes a sparse incomplete factorization, the iterative solver scales almost linearly with the system complexity. However, for large and thus ill-conditioned systems it breaks down. Notice that without preconditioning the solver would not converge for the larger systems.

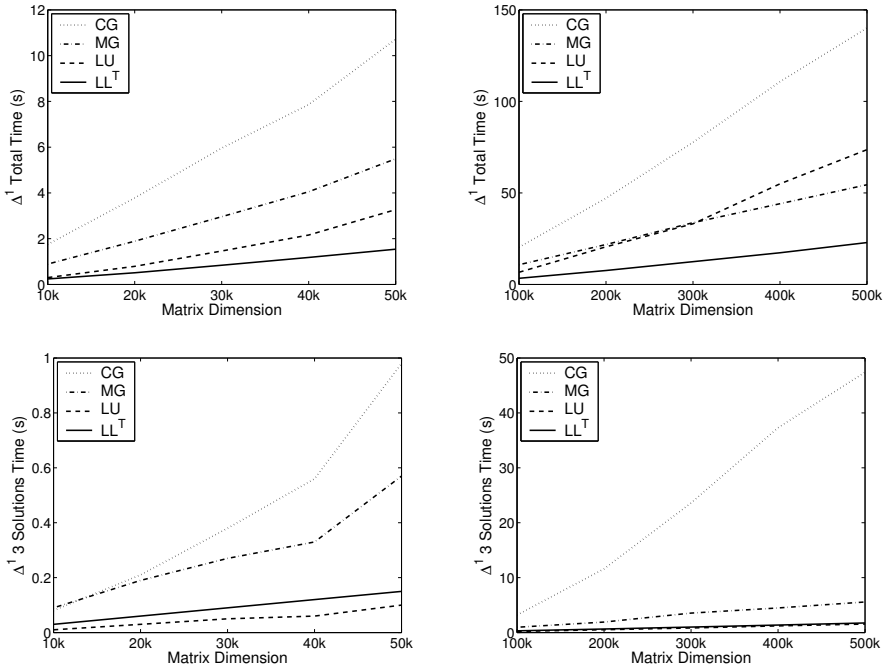
The experiments clearly verify the linear complexity of multigrid and sparse direct solvers. Once their sparse factorizations are pre-computed, the computational costs for actually solving the system are about the same for the LU and Cholesky solver. However, they differ significantly in the factorization performance, because the numerically more robust Cholesky factorization allows for more optimizations, whereas pivoting is required for the LU factorization to guarantee robustness. This is the reason for the break-down of the LU solver, such that the multigrid solver is more efficient in terms of total computation time for the larger systems.

Interactive applications often require to solve the same linear system for several right-hand sides (e.g. once per frame), which typically reflects the change of boundary constraints due to user interaction. For such problems the solution times, i.e., the third columns of the timings, are more relevant, as they correspond to the per-frame computational costs. Here the precomputation of a sparse factorization pays off and the direct solvers are clearly superior to the multigrid method.

Table 2 shows the same experiments for bi-Laplacian systems $\Delta_{\mathcal{S}}^2 X = B$ of the same complexity. In this case, the matrix setup is more complex, the matrix condition number is squared, and the sparsity decreases from 7 to 19 non-zeros per row.

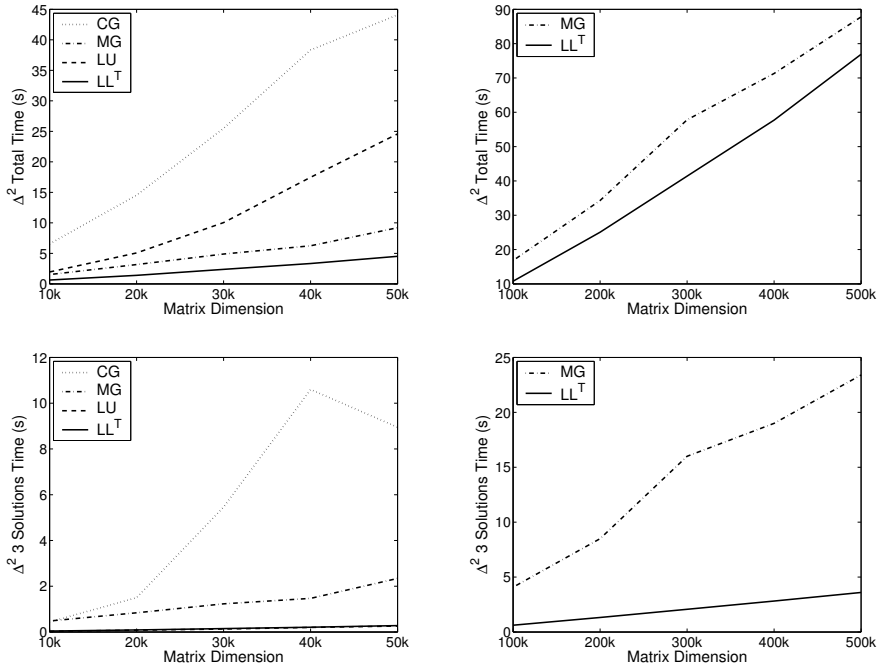
Due to the higher condition number the iterative solver takes much longer and even fails to converge on large systems. In contrast, the multigrid solver converges robustly without numerical problems; notice that constructing the multigrid hierarchy is almost the same as for the Laplacian system (up to one more ring of boundary constraints). The computational costs required for the sparse factorization are proportional to the increased number of non-zeros per row. The LU factorization additionally has to incorporate pivoting for numerical stability and failed for larger systems. In contrast, the Cholesky factorization worked robustly in all our experiments.

If we focus on the solution times for the bi-Laplacian systems and compare them to the Laplacian systems, we observe that the direct solver scales with the sparsity of the matrix, while the number of iterations required for the multigrid



Size	Iterative	Multigrid	LU	Cholesky
10k	0.11/1.56/0.08	0.15/0.65/0.09	0.07/0.22/0.01	0.07/0.14/0.03
20k	0.21/3.36/0.21	0.32/1.38/0.19	0.14/0.62/0.03	0.14/0.31/0.06
30k	0.32/5.26/0.38	0.49/2.20/0.27	0.22/1.19/0.05	0.22/0.53/0.09
40k	0.44/6.86/0.56	0.65/3.07/0.33	0.30/1.80/0.06	0.31/0.75/0.12
50k	0.56/9.18/0.98	0.92/4.00/0.57	0.38/2.79/0.10	0.39/1.00/0.15
100k	1.15/16.0/3.19	1.73/8.10/0.96	0.79/5.66/0.21	0.80/2.26/0.31
200k	2.27/33.2/11.6	3.50/16.4/1.91	1.56/18.5/0.52	1.59/5.38/0.65
300k	3.36/50.7/23.6	5.60/24.6/3.54	2.29/30.0/0.83	2.35/9.10/1.00
400k	4.35/69.1/37.3	7.13/32.5/4.48	2.97/50.8/1.21	3.02/12.9/1.37
500k	5.42/87.3/47.4	8.70/40.2/5.57	3.69/68.4/1.54	3.74/17.4/1.74

Table 1. Comparison of different solvers for Laplacian systems $\Delta_S X = B$ of 10k to 50k and 100k to 500k free vertices X. The three timings for each solver represent matrix setup, pre-computation, and three solutions for the x, y, and z components of X. The graphs in the upper row show the total computation times (sum of all three columns). The center row depicts the solution times only (3rd column), as those typically determine the per-frame cost in interactive applications.



Size	Iterative	Multigrid	LU	Cholesky
10k	0.33/5.78/0.44	0.40/0.65/0.48	0.24/1.68/0.03	0.24/0.35/0.04
20k	0.64/12.4/1.50	0.96/1.37/0.84	0.49/4.50/0.08	0.49/0.82/0.09
30k	1.04/19.0/5.46	1.40/2.26/1.23	0.77/9.15/0.13	0.78/1.45/0.15
40k	1.43/26.3/10.6	1.69/3.08/1.47	1.07/16.2/0.20	1.08/2.05/0.21
50k	1.84/33.3/8.95	2.82/4.05/2.34	1.42/22.9/0.26	1.42/2.82/0.28
100k	—	4.60/8.13/4.08	2.86/92.8/0.73	2.88/7.29/0.62
200k	—	9.19/16.6/8.50	—	5.54/18.2/1.32
300k	—	17.0/24.8/16.0	—	8.13/31.2/2.07
400k	—	19.7/32.6/19.0	—	10.4/44.5/2.82
500k	—	24.1/40.3/23.4	—	12.9/60.4/3.60

Table 2. Comparison of different solvers for bi-Laplacian systems $\Delta_S^2 X = B$ of 10k to 50k and 100k to 500k free vertices X. The three timings for each solver represent matrix setup, pre-computation, and three solutions for the x, y, and z components of X. The graphs in the upper row show the total computation times (sum of all three columns). The center row depicts the solution times only (3rd column), as those typically determine the per-frame cost in interactive applications. For the larger systems, both the iterative solver and the sparse LU factorization fail to compute a solution.

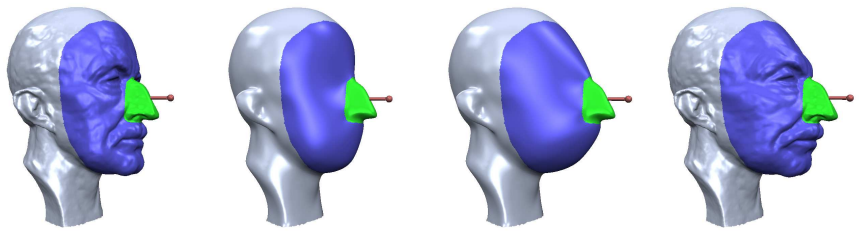


Fig. 3. Multiresolution modeling allows a low-frequency change of the global shape based on the change of a smooth base surface, that is computed by solving a bi-Laplacian system $\Delta_S^2 X = B$.

solver depends on the (squared) matrix condition. In our experiments it turned out that the performance gap between multigrid and direct methods is even larger for bi-Laplacian systems.

We also analyzed the memory consumption of the multigrid method and the sparse Cholesky solver, although both methods were optimized more for performance than for memory requirements. The memory consumption of the multigrid method is mainly determined by the meshes representing the different hierarchy levels. In contrast, the memory required for the Cholesky factorization depends significantly on the sparsity of the matrix, too. On the 500k example the multigrid method and the direct solver need about 1GB and 600MB for the Laplacian system, and about 1.1GB and 1.2GB for the bi-Laplacian system. Hence, the direct solver would not be capable of factorizing Laplacian systems of higher order on current PCs, while the multigrid method would succeed.

These comparisons show that direct solvers are a valuable and efficient alternative to multigrid methods even if the linear systems are highly complex. In all our experiments the sparse Cholesky solver was faster than the multigrid method, and if the system has to be solved for multiple right-hand sides, the precomputation of a sparse factorization is even more beneficial.

4 Applications

In this section we finally show several typical computer graphics and geometry processing applications that benefit from the use of sparse direct solvers. Most applications are based on solving Laplacian or bi-Laplacian systems, thus their characteristic behavior for different complexities or different solvers can be transferred from the experiments of the last section. Notice that it is difficult to compare to timings published in original papers on these approaches, since the computational costs depend on hardware factors (e.g., CPU, memory bandwidth), software factors (operating system, compiler), and on the datasets used. Although we tried to pick similar machines, these comparisons should be considered as a rough performance indication only.

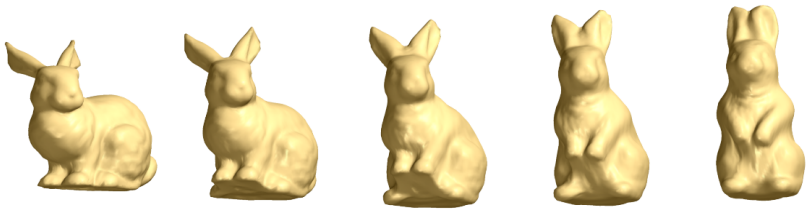


Fig. 4. Mesh morphing of two bunny models based on Poisson shape interpolation. Instead of absolute vertex positions, gradient fields (or Laplace coordinates) are interpolated as $D_t = (1-t)\,D_0 + t\,D_1$, and the vertex positions are derived by solving the Poisson system $\Delta X_t = D_t$ (Image courtesy of Xu et al. [43]).

Surface Modeling. The first application is freeform modeling or multiresolution modeling [25, 7], which requires to compute (the change of) a smooth base surface by solving bi-Laplacian systems $\Delta_S^2 X = B$ for the x, y, and z coordinates of the unconstrained (*dark/blue*) vertices X (cf. Fig. 3). Each time the designer drags some points on the surface, the boundary constraints change and the linear system has to be solved for another right-hand side in order to compute the deformed surface. As a consequence, these approaches greatly benefit from the sparse factorization solvers. The precomputation of basis functions for the deformation [7] also requires to solve the linear system for several right-hand sides, such that this precomputation gets more efficient, too.

Mesh Morphing. Given two meshes of identical connectivity, morphing between them corresponds to some linear interpolation of their geometry. But instead of using absolute vertex coordinates $\mathbf{x}_i$ for this task, Alexa [4] proposed to represent the meshes by *differential* Laplace coordinates $\mathbf{d}_i := \Delta\mathbf{x}_i$ and to linearly interpolate those instead. In a recent approach, Xu et al. [43] propose a non-linear interpolation of gradient fields, which avoids shrinkage of in-between models. In both cases each morphing step leads to a new set of Laplace vectors $D = (\mathbf{d}_0, \ldots, \mathbf{d}_n)$, from which the vertex positions can be derived by solving $\Delta X = D$. The resulting Laplacian multiple right-hand side problems can again be solved efficiently by sparse Cholesky factorizations.

Implicit Smoothing. In the implicit fairing approach [13] meshes are smoothed by an integration of the PDE $\partial\,\mathbf{x}_i/\partial t = \lambda\Delta_S\mathbf{x}_i$, leading to the so-called *mean curvature flow*. Using semi-implicit integration, this non-linear problem is decomposed into a sequence of linear ones, such that in each time-step the Laplace discretization $\Delta_{X^{(i)}}$ is updated and the Laplacian system $(I - \lambda\Delta_{X^{(i)}})\,X^{(i+1)} = X^{(i)}$ is solved. In this case the matrix re-ordering and the symbolic factorization can be kept and just the numerical factorization and the solution have to be computed. In our experiments this saved 40%-60% of the solver time per iteration.

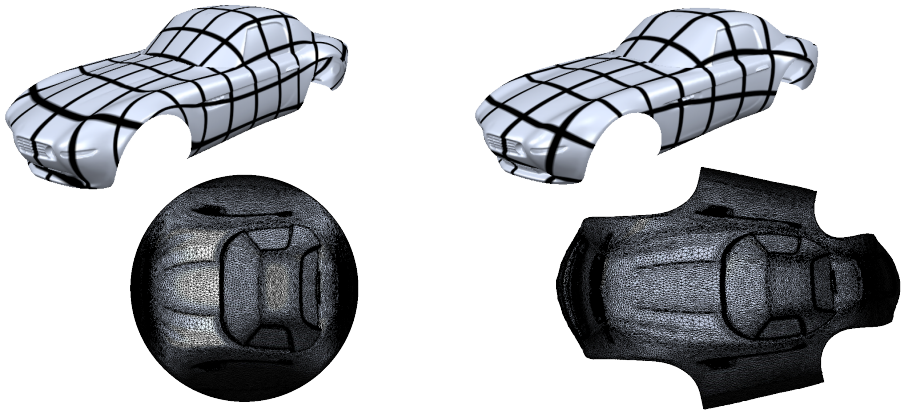


Fig. 5. Two different parameterizations of a car model: discrete conformal parameterization with fixed boundary (*left*), least squares conformal map with free boundary (*right*). Both parameterizations are computed by solving a sparse spd system for the free 2D parameter values associated to the mesh vertices.

Conformal Parameterization. Computing a conformal parameterization [32, 12] with fixed boundary vertices requires the solution of a Laplacian system $\Delta_{\mathcal{S}} X = B$ for x and y (cf. Fig. 5, left). In [2] a highly elaborate multigrid solver has been derived by evaluating different kinds of multigrid hierarchies and preconditioning strategies. This solver was then used for the parameterization of large meshes, where it takes only 37s for 580k DoFs on a 2.8GHz Pentium4. This time includes loading the system from disk, building the hierarchy, and solving the system for the x coordinate [1]. Our implementation based on the sparse Cholesky solver takes (on a 3.0GHz Pentium4) 28s for for the parameterization of 600k vertices, including matrix setup, re-ordering, factorization, and two solutions.

Least Squares Conformal Maps. In the approach of [26] a conformal parameterization is not computed by minimizing the discrete Dirichlet energy, but instead by solving a system of Cauchy-Riemann equations for each face (cf. Fig. 5, right). Since the number of faces F is about twice the number of vertices V, this system is overdetermined and hence solved in the least squares sense using the normal equations, leading to a spd matrix of dimension $2V \times 2V$, which is similar in structure to a Laplacian matrix. Since the iterative solver used in the original paper [26] was not capable of parameterizing large meshes, the use of multigrid methods was proposed in [34]. On an 1.2GHz Pentium4 their hierarchical approach takes 18s, 31s, and 704s for meshes of 18k, 36k, and 560k vertices, respectively. On a comparable machine (Athlon 1.2GHz) the direct sparse solver is about 4–5 times faster; on the 3.0GHz machine these parameterizations can be computed in 1.4s, 3.2s, and 95s, respectively.

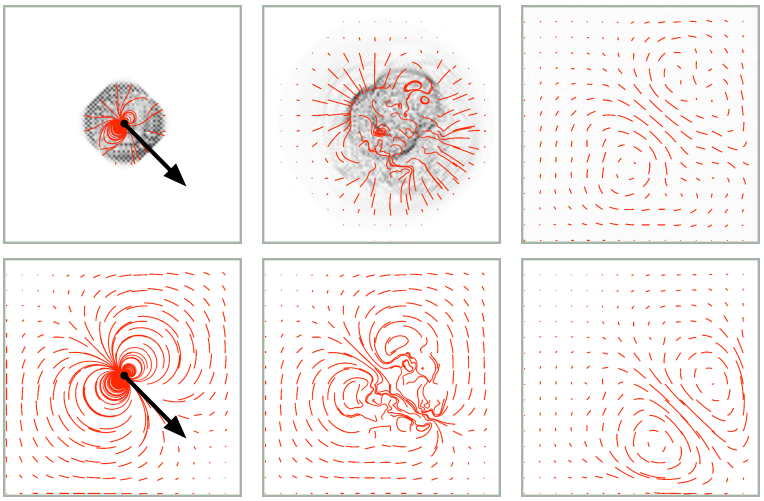


Fig. 6. This example shows a fluid's reaction to a high external force after 1, 3, and 20 time-steps (from left to right) on a 100×100 grid. The line segments visualize the velocity field, the background color shows the amount of divergence. A constant number of CG iterations per frame fails to sufficiently propagate the forces and to keep the field free of divergence (*top row*). The sparse Cholesky solver requires a constant time per frame, is significantly faster, and yields correct results independent from the external forces (*bottom row*).

Fluid Dynamics. In Stam's stable fluid approach [39] the Navier-Stokes equations are solved by a four-step procedure in each time step: after updating external forces and advecting the velocity field, a diffusion process considers the viscosity and a final projection yields a divergence-free velocity field. The last two steps both involve solving a Laplacian system. Since the field is assumed not to change too much from one time-step to the next, the current state yields good starting values, such that in most implementations a fixed small number of CG iterations is used for solving both systems. The break-down of this method in case of high external forces is shown in the top row of Fig. 6. In contrast, the sparse direct solver is twice as fast in this example and yields correct results also for rapidly changing fields (cf. Fig. 6, bottom row).

Poisson Matting. Laplacian systems are also used in image manipulation, like for instance the Poisson matting approach of [40]. A given image I is considered as a composition of a foreground object F and a background B using the *matting equation* $I = \alpha F + (1 - \alpha)B$, which is to be solved for the matte $\alpha\,(x, y)$ (cf. Fig. 7). We use a variant of the original approach, where taking the divergence of an approximate gradient of the matting equation leads to the Poisson system $\Delta\alpha = \operatorname{div}(\operatorname{sign}(F - B)\,\nabla I)$. Hence, the computation of the α-matte amounts to solving a spd Laplacian system and therefore benefits from the sparse direct solvers like the other examples.

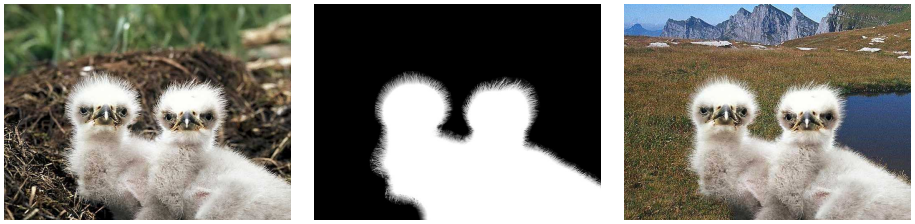


Fig. 7. In order to separate an image I into foreground F and background B, the Poisson matting approach derives an α-matte by solving a Poisson equation $\Delta\alpha = b$.

5 Conclusion

In this paper we discussed and compared different classes of linear system solvers for large sparse symmetric positive matrices, and pointed out that sparse direct solvers are a valuable alternative to the usually employed multigrid methods, since they turned out to be more efficient and easier to use in all our experiments.

Although the class of sparse spd matrices seems to be quite restricted, many frequently encountered geometry processing problems over polygonal meshes lead to exactly this kind of systems or can easily be reformulated in this form. As we demonstrated in our experiments, all these applications could benefit considerably from the use of sparse direct solvers.

References

[1] B. Aksoylu. personal communication.

[2] B. Aksoylu, A. Khodakovsky, and P. Schröder. Multilevel Solvers for Unstructured Surface Meshes. *SIAM Journal on Scientific Computing*, 26(4):1146–1165, 2005.

[3] M. Alexa. Local control for mesh morphing. In *Proc. of Shape Modeling International 01*, pages 209–215, 2001.

[4] M. Alexa. Differential coordinates for local mesh morphing and deformation. *The Visual Computer*, 19(2):105–114, 2003.

[5] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[6] F. A. Bornemann and P. Deuflhard. The cascading multigrid method for elliptic problems. *Num. Math.*, 75(2):135–152, 1996.

[7] M. Botsch and L. Kobbelt. An intuitive framework for real-time freeform modeling. In *Proc. of ACM SIGGRAPH 04*, pages 630–634, 2004.

[8] W. L. Briggs, V. E. Henson, and S. F. McCormick. *A Multigrid Tutorial*. SIAM, 2nd edition, 2000.

[9] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proc. of the 24th National Conference ACM*, pages 157–172, 1969.

[10] J. W. Demmel. *Applied numerical linear algebra*. SIAM, 1997.

[11] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20(3):720–755, 1999.

[12] M. Desbrun, M. Meyer, and P. Alliez. Intrinsic parameterizations of surface meshes. In *Proc. of Eurographics 02*, pages 209–218, 2002.

[13] M. Desbrun, M. Meyer, P. Schröder, and A. H. Barr. Implicit fairing of irregular meshes using diffusion and curvature flow. In *Proc. of ACM SIGGRAPH 99*, pages 317–324, 1999.

[14] M. Garland. Multiresolution modeling: Survey & future opportunities. In *Eurographics State of the Art Report 99*, 1999.

[15] A. George and J. W. H. Liu. *Computer Solution of Large Sparse Positive Definite Matrices*. Prentice Hall, 1981.

[16] A. George and J. W. H. Liu. The evolution of the minimum degree ordering algorithm. *SIAM Review*, 31(1):1–19, 1989.

[17] P. R. Gill, W. Murray, and M.H. Wright. *Practical Optimization*. Academic Press, 1981.

[18] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, 1989.

[19] W. Hackbusch. *Multi-Grid Methods and Applications*. Springer Verlag, 1986.

[20] M. Hestenes and E. Stiefel. Method of conjugate gradients for solving linear systems. *J. Res. Nat. Bur. Stand.*, 49:409–436, 1952.

[21] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal of Sci. Comput.*, 20(1):359–392, 1998.

[22] L. Kobbelt. Discrete fairing. In *Proc. on 7th IMA Conference on the Mathematics of Surfaces*, pages 101–131, 1997.

[23] L. Kobbelt and M. Botsch. Freeform shape representations for efficient geometry processing. In *Proc. of Shape Modeling International 03*, pages 111–118, 2003.

[24] L. Kobbelt, S. Campagna, and H.-P. Seidel. A general framework for mesh decimation. In *Proc. of Graphics Interface 98*, pages 43–50, 1998.

[25] L. Kobbelt, S. Campagna, J. Vorsatz, and H.-P. Seidel. Interactive multiresolution modeling on arbitrary meshes. In *Proc. of ACM SIGGRAPH 98*, pages 105–114, 1998.

[26] B. Lévy, S. Petitjean, N. Ray, and J. Maillot. Least squares conformal maps for automatic texture atlas generation. In *Proc. of ACM SIGGRAPH 02*, pages 362–371, 2002.

[27] J. W. H. Liu. Modification of the minimum-degree algorithm by multiple elimination. *ACM Trans. Math. Softw.*, 11(2):141–153, 1985.

[28] J. W. H. Liu and A. H. Sherman. Comparative analysis of the Cuthill-McKee and the reverse Cuthill-McKee ordering algorithms for sparse matrices. *SIAM J. Numerical Analysis*, 2(13):198–213, 1976.

[29] G. A. Meurant. *Computer solution of large linear systems*. Elsevier, 1999.

[30] M. Meyer, M. Desbrun, P. Schröder, and A. H. Barr. Discrete differential-geometry operators for triangulated 2-manifolds. In Hans-Christian Hege and Konrad Polthier, editors, *Visualization and Mathematics III*, pages 35–57. Springer-Verlag, Heidelberg, 2003.

[31] X. Ni, M. Garland, and J. C. Hart. Fair morse functions for extracting the topological structure of a surface mesh. In *Proc. of ACM SIGGRAPH 04*, pages 613–622, 2004.

[32] U. Pinkall and K. Polthier. Computing discrete minimal surfaces and their conjugates. *Experimental Mathematics*, 2(1):15–36, 1993.

[33] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 2nd edition, 1992.

[34] N. Ray and B. Levy. Hierarchical Least Squares Conformal Map. In *Proc. of Pacific Graphics 03*, pages 263–270, 2003.

[35] Y. Renard and J. Pommier. `Gmm++`: a generic template matrix C++ library. http://www-gmm.insa-toulouse.fr/getfem/gmm_intro.

[36] Y. Saad and H. A. van der Vorst. Iterative solution of linear systems in the 20th century. *J. Comput. Appl. Math.*, 123(1-2):1–33, 2000.

[37] J. R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, Carnegie Mellon University, 1994.

[38] O. Sorkine, D. Cohen-Or, Y. Lipman, M. Alexa, C. Rössl, and H.-P. Seidel. Laplacian surface editing. In *Proc. of Eurographics symposium on Geometry Processing 04*, pages 179–188, 2004.

[39] Jos Stam. Stable fluids. In *Proc. of ACM SIGGRAPH 99*, pages 121–128, 1999.

[40] J. Sun, J. Jia, C.-K. Tang, and H.-Y. Shum. Poisson matting. In *Proc. of ACM SIGGRAPH 04*, pages 315–321, 2004.

[41] G. Taubin. A signal processing approach to fair surface design. In *Proc. of ACM SIGGRAPH 95*, pages 351–358, 1995.

[42] S. Toledo, D. Chen, and V. Rotkin. Taucs: A library of sparse linear solvers. http://www.tau.ac.il/∼stoledo/taucs.

[43] D. Xu, H. Zhang, Q. Wang, and H. Bao. Poisson shape interpolation. In *Proc. of ACM symposium on Solid and Physical Modeling 05*, 2005.

[44] Yizhou Yu, Kun Zhou, Dong Xu, Xiaohan Shi, Hujun Bao, Baining Guo, and Heung-Yeung Shum. Mesh editing with Poisson-based gradient field manipulation. In *Proc. of ACM SIGGRAPH 04*, pages 644–651, 2004.