Updated Sparse Cholesky Factors for Corotational Elastodynamics

Florian Hecht, Yeon Jin Lee, Jonathan R. Shewchuk, and James F. O'Brien
University of California, Berkeley

We present *warp-canceling corotation*, a nonlinear finite element formulation for elastodynamic simulation that achieves fast performance by making only partial or delayed changes to the simulation's linearized system matri ces. Coupled with an algorithm for incremental updates to a sparse Cholesky factorization, the method realizes the stability and scalability of a sparse direct method without the need for expensive refactorization at each time step. This finite element formulation combines the widely used corotational method with stiffness warping so that changes in the per-element rotations are initially approximated by inexpensive per-node rotations. When the errors of this approximation grow too large, the per-element rotations are selectively corrected by updating parts of the matrix chosen according to locally measured errors. These changes to the system matrix are propagated to its Cholesky factor by incremental updates that are much faster than refactoring the matrix from scratch. A nested dissection ordering of the system matrix gives rise to a hierarchical factorization in which changes to the system matrix cause limited, well-structured changes to the Cholesky factor. We show examples of simulations that demonstrate that the proposed formulation produces results that are visually comparable to those produced by a standard corotational formulation. Because our method requires computing only partial updates of the Cholesky factor, it is substantially faster than full refactorization and outperforms widely used iterative methods such as preconditioned conjugate gradients. Our method supports a controlled trade-off between accuracy and speed, and unlike iterative methods its performance does not slow for stiffer materials but rather it actually improves.

Categories and Subject Descriptors: G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*Sparse linear systems*; I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling—*Physically based modeling*; I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism—*Animation*; I.6.8 [**Simulation and Modeling**]: Types of Simulation—*Animation*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Sparse Cholesky factorization, corotational finite element method, elastodynamics, stiffness warping, physically-based animation

ACM Reference Format:

Hecht, F., Lee, Y. J., Shewchuk, J. R., and O'Brien, J. F. 2012. Updated sparse Cholesky factors for corotational elastodynamics. ACM Trans. Graph. 31, 5, Article 123 (August 2012), 13 pages.
DOI = 10.1145/2231816.2231821
http://doi.acm.org/10.1145/2231816.2231821

1. INTRODUCTION

The behavior of elastic objects and materials such as toys, rubber, buildings, clothing, and skin is governed by partial differential equations that are essentially linear for very small deformations, but become nonlinear for large deformations. The main source of nonlinearity is almost disappointingly mundane: objects rotate. In particular, one part of a deforming object can rotate relative to another part of the same object. An unfortunate consequence is that standard methods for visually realistic simulation of the dynamics of flexible objects are substantially slower than they would be if the behavior was purely linear. These methods are widespread in computer animation, special effects for films, secondary motion in games, and environmental components in training applications. The time available for computation varies wildly among applications, but the demand for faster simulation methods is universal.

The elastodynamic simulation methods most commonly used in computer graphics employ finite element methods on tetrahedral meshes with some form of implicit time integration scheme, such as Newmark or backward Euler integration, where the main bottleneck is to assemble and solve a large, sparse, positive-definite linear system. This system must be solved many times, but because the underlying physics are nonlinear, the work spent computing one solution generally does not benefit subsequent solutions. Instead, the nonlinear equations are linearized at the beginning of each time step, the resulting linear system is solved, and then the process starts from scratch at the next time step. The *corotational* formulation of the finite element method, discussed in Section 2, is a linearization method widely used for elastodynamics in graphics applications.

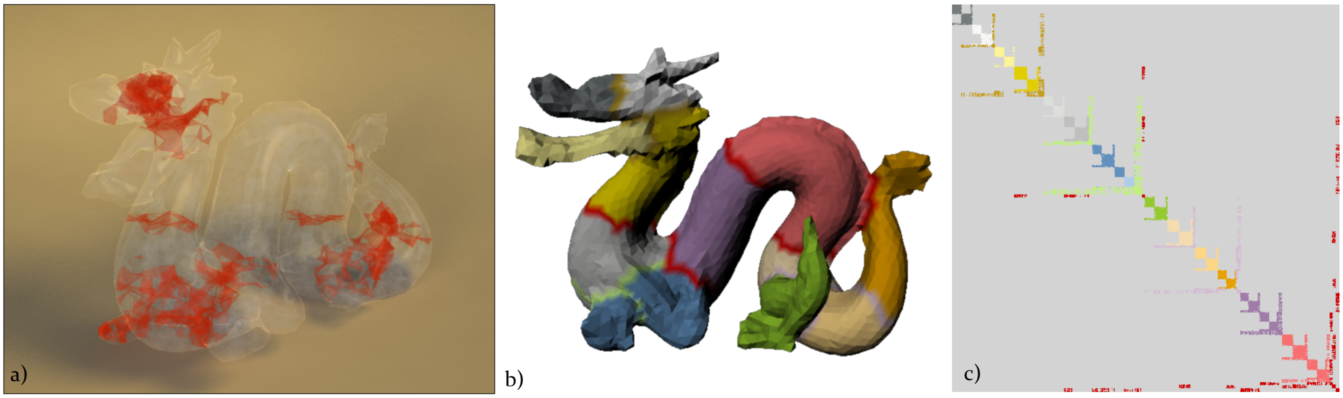
The predominant methods for solving these linear systems are iterative solvers, such as the conjugate gradient method [Hestenes and Stiefel 1952; Shewchuk 1994] with a diagonal or incomplete Cholesky preconditioner. These iterative methods have proven themselves efficient for systems of small to moderate sizes. They typically converge rapidly and, for applications such as video games where accuracy is not a main concern, the iterations may be halted early without unacceptable artifacts. However, as the size of the system increases, and in particular as its spectral span grows, these solvers require more iterations, and the total work scales superlinearly with the problem size. Moreover, slow convergence is common with stiff materials and meshes with large disparities in

This work was supported in part by NSF Awards CCF-0635381 and IIS-0915462, UC Lab Fees Research Program grant 09-LR-01-118889-OBRJ, Intel's Science and Technology Center for Visual Computing, and by gifts from Autodesk, NVIDIA, and Pixar.

Authors' addresses: F. Hecht, Y. J. Lee, J. R. Shewchuk, and J. F. O'Brien (correspondence author), University of California, Berkeley, CA; email: job@berkeley.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2012 ACM 0730-0301/2012/12-ART123 $10.00

DOI 0.1145/2231816.2231821
http://doi.acm.org/0.1145/2231816.2231821



Images copyright Hecht, Lee, Shewchuk, and O'Brien.

Fig. 1**:** A finite element simulation of an elastic object, sped up by incremental updates to the linear system's Cholesky factor. (a) An animation frame where red coloring indicates elements of the mesh whose entries in the Cholesky factor are updated during that frame. (b) Nested dissection of the mesh. (c) The nested dissection ordering of the system matrix, with nonzeros color-coded to match the mesh. This structure is preserved in the Cholesky factor.

element sizes, both of which tend to worsen the conditioning of the linear system. In these cases, efforts to achieve faster speeds by performing too few iterations can generate obviously unrealistic motion and even cause the time integration method to diverge.

Direct methods like Cholesky factorization followed by back substitution [Cholesky 1910; Golub and Van Loan 1996] produce accurate solutions and avoid many of the difficulties that plague iterative methods.[1] The Cholesky factor of a symmetric, positive-definite matrix $\mathbf{A}$ is a lower triangular matrix $\mathbf{L}$ such that $\mathbf{A} = \mathbf{L}\mathbf{L}^{\mathsf{T}}$. Unfortunately, even if $\mathbf{A}$ is sparse, the factor $\mathbf{L}$ is usually dense. The matrix positions that are zero in $\mathbf{A}$ but nonzero in $\mathbf{L}$ are collectively called *fill*. The pattern of nonzeros and fill in $\mathbf{L}$ depends (for all practical purposes) solely on the pattern of nonzeros in $\mathbf{A}$, and not on the numerical values of those nonzeros.

Cholesky factorization of sparse matrices consists of two phases, called *symbolic factorization* and *numerical factorization*. Symbolic factorization determines the positions, but not the numerical values, of the nonzero values in the Cholesky factor $\mathbf{L}$, and thus lays out the data structure that efficiently stores these values. Numerical factorization computes the values of $\mathbf{L}$ and records them in the data structure. Once the factorization has been computed, a linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$ can be solved for an unknown vector $\mathbf{x}$ given a known vector $\mathbf{b}$ through forward and backward substitution, which are simple and fast methods for computing $\mathbf{L}^{-1}\mathbf{b}$ and $\mathbf{L}^{-\mathsf{T}}\mathbf{L}^{-1}\mathbf{b}$ in succession. Sparse Cholesky solvers are robust against ill-conditioning, and unlike iterative solvers, their running times are unrelated to matrix conditioning.

Prior to factorization, *reordering methods* permute the rows and columns of $\mathbf{A}$ to reduce the amount of fill created. Off-diagonal nonzeros in $\mathbf{A}$ tend to generate fill to the right and downward in $\mathbf{L}$. Reordering strategies that move nonzero elements in $\mathbf{A}$ closer to the diagonal or break $\mathbf{A}$ into semi-independent blocks improve the sparsity of $\mathbf{L}$. Cholesky factors for large linear systems can require a great deal of memory, but good orderings can substantially mitigate both memory use and running times for factorization and back substitution.

The reason direct solvers are less often used to animate nonlinear elastic motion is that the cost of factorization cannot be amortized across multiple time steps if the linearized system's matrix changes. The ordering and symbolic factorization need to be recomputed only when the structure of the underlying mesh changes, but the dominant cost is that of numeric factorization, which must be repeated whenever matrix values change.

In this article we present *warp-canceling corotation*, a method that takes advantage of the facts that the matrix changes gradually and that we do not need an exact solution for visual applications, as long as there are no objectionable artifacts. We exploit the special nature of the changes to the matrix, namely that we can approximate small or global changes to the linear system without changing the Cholesky factors by wrapping orthonormal matrices around the factorization. These orthonormal matrices are functionally equivalent to a variation of the *stiffness warping* method of Müller et al.[2002].

The source of the nonlinearity in elastodynamics is the fact that elements can rotate and adjoining elements can rotate relative to each other. Stiffness warping uses per-node rotations to approximate per-element rotations. Because the per-node rotations can be expressed as orthonormal transformations to a fixed sparse matrix, we can update them at each time step without changing the core matrix. If the elements adjoining a node do not share the same rotation from their rest configurations, the error in the approximation is proportional to local differences among the element rotations. Unfortunately, the error manifests as unbalanced forces (so-called *ghost forces*), which can create odd visual artifacts and can cause the time integrator to lose stability and blow up.

We control this error by performing local, incremental updates of the sparse matrix and its Cholesky factors so that they better approximate the exact nonlinear corotational method. (See Fig. 1.) The updates are scheduled either by imposing a threshold on the maximum permissible error or by allotting a fixed computational budget per time step, thereby offering a range of choices about balancing speed with accuracy. The cost of these partial updates is typically a fraction of the cost of complete numerical refactorization. Back substitution is very fast, so these partial updates constitute most of the cost of a time step.

The greatest errors in stiffness warping arise where adjoining elements undergo large relative rotations, usually because of large forces or collisions. The computational budget is spent updating these regions first. Where smaller deformations occur, stiffness warping has acceptable artifacts, which we correct gradually over many time steps. The stiffer an object is, the less it deforms and the

[1] An interesting historical discussion of numerical algorithms including Cholesky factorization and its relation to other methods such as Gaussian elimination can be found in the article by Grcar [2011].

lower the error, so our warp-canceling method actually performs better for stiffer materials, contrary to most iterative methods.

2. BACKGROUND

Physically-based animation of elastic objects was introduced by Terzopoulos et al. [1987] and other contemporaneous work. A substantial amount of research has been done since, and we refer the reader to Gibson and Mirtich [1997] for a review of older methods, or Nealen et al. [2006] for a review of more recent results. Simulations of elastic systems have found many uses in animation, and researchers have extended the range of materials that can be simulated from solids to near-fluids exhibiting behaviors like incompressibility and plastic flow.

Our formulation of nonlinear finite element methods for elastodynamics is a *corotational method*, which is widely used in computer graphics. The nonlinearity of large material deformations arises from the fact that an element can deviate from its rest configuration not only by compression or expansion, but also by rotation, and different elements may undergo different rotations. Corotational methods explicitly account for these rotations by factoring them out of the element stiffness matrices. The method was introduced to the graphics community by Müller and Gross [2004] and Etzmuß et al. [2003]. Other researchers have extended the method to make simulations insensitive to element inversion [Irving et al. 2004], or modified it to work in a multigrid [Zhu et al. 2010] or modal [Choi and Ko 2005] setting.

Müller et al. [2002] describe a precursor of the full corotational method called *stiffness warping*, which uses per-node rotations instead of per-element rotations. As discussed in the previous section, the errors from this approximation induce ghost forces, and this flaw has limited the adoption of this method. Nevertheless, we build on this work by incrementally updating the matrix to make stiffness warping better approximate the corotational method, thus limiting the ghost forces so that they are not problematic. Courtecuisse et al. [2010] take an approach related to ours: they use stiffness warping with an incomplete Cholesky factorization to precondition a conjugate gradient solver. Bridson et al. [2006] use a conjugate gradient solver with a modified incomplete Cholesky preconditioner for fluid dynamics.

As Felippa [2007] notes, the notion of separating out rotations originates in the finite element literature outside graphics. Belytschko and Hsieh [1979] introduced the term *corotational* in 1979, with the goal of separating out a single rigid body motion from an otherwise linear deformation. Nour-Omid and Rankin [1991] describe an element-by-element method that acts as a wrapper around an existing finite element library, extending its domain of application from small, linear deformations to large, nonlinear deformations.

The corotational method requires reassembly of the system matrix every time step. This assembly can be a significant portion of the overall simulation cost, in part because of its irregular memory access patterns [Parker and O'Brien 2009]. Chentanez et al. [2009] reduce this assembly cost by using partial updates to skip the full reassembly in computations where the matrix changes locally because of local changes to the underlying mesh.

Any simulation of dynamics that uses an implicit time integration scheme requires a numerical solver for sparse linear systems. The structure of the linear system's sparsity is determined by the edges of the tetrahedral mesh used to discretize the physical domain. Botsch et al. [2005] provide an overview of sparse direct methods, iterative methods, and how they compare in computer graphics applications. Iterative methods such as conjugate gradient solvers are easy to implement. They require only a simple code template [Barrett et al. 1993], an implementation of a sparse matrix-vector multiplication operation, and optionally, an implementation of a preconditioning operator.

Sparse direct solvers are more complicated to implement and require optimized numerical kernels to achieve their potential for speed, so the use of a well-established library is recommended. We use Toledo's TAUCS software package [2003] and have modified it to support incremental updates to the Cholesky factors. Although we focus on elastodynamic simulation, sparse direct solvers have applications to other areas of graphics. For example, many geometry processing algorithms can benefit from sparse direct solvers [Botsch et al. 2005]. A detailed evaluation and comparison of the currently available software for solving large, sparse, symmetric linear systems can be found in the article by Gould et al. [2007].

Two key parts of our method are a formulation where only relatively small parts of the system matrix change at each time step and an incremental update scheme for the factorization that only does work proportional to these changes. The classic paper by Gill et al. [1974] describes several schemes for updating the inverse or factorization of a matrix. Most of these methods focus on small changes of a particular form. For example, the well-known Sherman-Morrison formula [Press et al. 2002] computes the inverse of a matrix $\mathbf{A}' = \mathbf{A} \pm \mathbf{v}\mathbf{v}^\mathsf{T}$ for a given vector $\mathbf{v}$ when $\mathbf{A}^{-1}$ is already known. This type of low-rank change can also be computed efficiently for sparse Cholesky factorizations using an algorithm based on Givens rotations developed by Davis and Hager [1999; 2009]. This algorithm is implemented in the publicly available library CHOLMOD [Chen et al. 2008]. Unfortunately our formulation requires higher-rank changes than can be efficiently performed with CHOLMOD. We discuss this issue in more detail and provide comparative running times in Section 5.2.

Sorkine et al. [2005] improve the speed of Davis and Hager's algorithm for the special case of inserting rows with a single nonzero entry, and it might be possible to devise a more efficient variation of the Davis-Hager algorithm for the specific kinds of changes we make to the matrix. However, we believe that any method based on Givens rotations will necessarily be slower than our update method for the high-rank modifications we require.

3. FINITE ELEMENT FORMULATION

We start with a standard discretization of Lagrangian mechanics

$$\mathbf{M}\ddot{\mathbf{x}} + \mathbf{C}\dot{\mathbf{x}} + \mathbf{K}(\mathbf{x} - \mathbf{m}) = \mathbf{f}_{\text{ext}}, \tag{1}$$

where $\mathbf{M}$ is the (lumped) mass matrix, $\mathbf{C}$ is the damping matrix, $\mathbf{K}$ is the stiffness matrix, $\mathbf{f}_{\text{ext}}$ is a vector of external forces, $\mathbf{x}$ is a vector specifying the nodes' coordinates in world space, and $\mathbf{m}$ specifies the coordinates in material space. We use implicit Euler time integration. Substituting $\ddot{\mathbf{x}}^{t+1} = (\dot{\mathbf{x}}^{t+1} - \dot{\mathbf{x}}^{t})/\Delta t$ into (1) and rearranging yields

$$(\mathbf{M} + \Delta t\mathbf{C} + \Delta t^2\mathbf{K})\dot{\mathbf{x}}^{t+1} = \mathbf{M}\dot{\mathbf{x}}^{t} + \Delta t(\mathbf{f}_{\text{ext}}^{t+1} + \mathbf{f}_{\text{els}}^{t}), \tag{2}$$

which we solve for the new velocities $\dot{\mathbf{x}}^{t+1}$. The internal elastic force $\mathbf{f}_{\text{els}}^{t}$ is a shorthand for $-\mathbf{K}(\mathbf{x}^t - \mathbf{m})$. Similar derivations can be done for other implicit integrators such as Newmark integration.

3.1 Corotational Finite Element Methods

In the corotational method described by Müller and Gross [2004], the right-hand side of Eq. (2) and the system matrix $\mathbf{A} = \mathbf{M} + \Delta t\mathbf{C} + \Delta t^2\mathbf{K}$ change at each time step. The system matrix changes

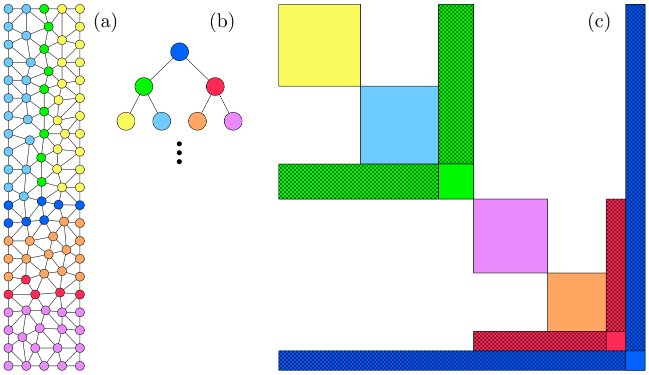


Fig. 2**:** Nested dissection partitions the simulation mesh (a) into a separator and two subdomains, which are bisected recursively, forming a hierarchy of separators called a dissection tree (b). Reordering the rows and columns of the system matrix according to a postorder traversal of the tree produces a hierarchical block structure (c). The elongated blocks shown with hatch marks contain the edges connecting a separator to its descendants in the tree.

because both $\mathbf{C}$ and $\mathbf{K}$ change as they are reassembled with the current element rotations. Both matrices have the same sparse structure and differ only in their scaling coefficients, as $\mathbf{C}$ maps the velocities to damping forces and $\mathbf{K}$ maps the displacements to internal elastic forces. Each tetrahedral element in the mesh contributes a 12×12 element stiffness matrix to the global stiffness matrix $\mathbf{K}$. The contribution of an element e is

$$\widehat{\mathbf{K}}_e = \begin{pmatrix} \mathbf{R}_e & & & \\ & \mathbf{R}_e & & \\ & & \mathbf{R}_e & \\ & & & \mathbf{R}_e \end{pmatrix} \mathbf{K}_e \begin{pmatrix} \mathbf{R}_e^\mathsf{T} & & & \\ & \mathbf{R}_e^\mathsf{T} & & \\ & & \mathbf{R}_e^\mathsf{T} & \\ & & & \mathbf{R}_e^\mathsf{T} \end{pmatrix}, \qquad (3)$$

where $\mathbf{K}_e$ is the unrotated 12×12 element stiffness matrix using the linear Cauchy strain and incorporating the two Lamé constants for the material parameters. $\mathbf{R}_e$ is the 3×3 rotation matrix that rotates from e's reference frame in material coordinates to the world coordinate frame. Likewise, e's contribution $\widehat{\mathbf{C}}_e$ to $\mathbf{C}$ is computed the same way from a material-frame element damping matrix $\mathbf{C}_e$. The rotations are computed from the element's deformation gradient using the modified 3×3 singular value decomposition described by Irving et al. [2004] that allows a simulation to continue even when some elements become inverted.

3.2 Cholesky Factorization

Section 1 describes three steps in computing the Cholesky factorization $\mathbf{A} = \mathbf{L}\mathbf{L}^\mathsf{T}$: reordering, symbolic factorization, and numeric factorization. The first two steps only need to be done at the beginning of the simulation or when the structure of the mesh changes.

Reordering permutes the rows and columns of $\mathbf{A}$ so that $\mathbf{L}$ will also be sparse. We find that the *nested dissection* strategy [George 1973; Lipton et al. 1979] produces excellent results. Nested dissection uses recursive graph bisection to partition the mesh into a hierarchy of small subdomains, and orders the rows and columns of $\mathbf{A}$ accordingly, as illustrated in Fig. 2. The graph in the figure, which represents the structure of a mesh, is subdivided into four subdomains (yellow, cyan, lavender, and orange) separated by subsets of nodes called *separators* (green, red, and blue). The colored blocks on the right illustrate the nonzero structure of $\mathbf{A}$. Some of the entries in these blocks are zero, but fast solvers typically treat the blocks with optimized dense matrix libraries such as the Basic Linear Algebra Subprograms (BLAS). The hierarchy of subdomains, illustrated in the center of the figure, is commonly called a *dissection tree*. It determines the structure of our incremental updates of the Cholesky factor $\mathbf{L}$.

The second step, symbolic factorization, computes $\mathbf{L}$'s sparsity pattern. For all but the simplest meshes, $\mathbf{L}$ has fill where nonzeros appear in positions that are zero in $\mathbf{A}$. This step determines where fill occurs and allocates the data structure that stores $\mathbf{L}$. The merit of the nested dissection ordering is that $\mathbf{L}$ retains the same sparse structure shown at the right of Fig. 2.

Changes in $\mathbf{A}$ trigger a cascade of changes in $\mathbf{L}$. Normally the whole numerical factorization would be redone at each time step, but the nested dissection ordering helps to limit the dependencies between entries of $\mathbf{A}$ and entries of $\mathbf{L}$. A change to $\mathbf{A}$ in one mesh subdomain affects $\mathbf{L}$ only in that subdomain and the separators above it in the hierarchy. We exploit this phenomenon in our incremental update method.

The efficiency of this type of sparse factorization depends heavily on implementation details such as the data structures used to store sparse matrices, the ordering of memory accesses, the graph partitioning algorithm, and the parallelization approach. The text by Davis [2006] provides an excellent discussion. We use the software TAUCS [Toledo 2003] for the initial factorization and METIS [Karypis and Kumar 1995] for graph partitioning. Note that TAUCS creates a supernode structure that is not always an exact match to METIS' hierarchy of separators, but the differences are generally minor and did not hurt the efficiency of our method.

3.3 Stiffness Warping

Stiffness warping approximates changes to the assembled system matrix $\mathbf{A}$ as the product of a constant core matrix and a pair of changing orthonormal rotation matrices. Because the core matrix does not change, there is no need to recompute its numerical factorization at every time step.

This approximate method does not apply the changing per-element rotations of the corotational method, but instead applies changing rotations to each node. Each per-node rotation is an average of the rotations of the surrounding elements. It is only an approximation because the elements adjoining a node do not have identical mappings from their rest frames to the world frame. However, it is a good approximation if the relative rotation differences between adjoining elements are small.

The approximate system has the form

$$\begin{pmatrix} \mathbf{R}_1 & & \\ & \ddots & \\ & & \mathbf{R}_n \end{pmatrix} \mathbf{A} \begin{pmatrix} \mathbf{R}_1^\mathsf{T} & & \\ & \ddots & \\ & & \mathbf{R}_n^\mathsf{T} \end{pmatrix} \dot{\mathbf{x}}^{t+1} = \mathbf{M}\dot{\mathbf{x}}^t + \Delta t(\mathbf{f}_{\text{ext}}^{t+1} + \mathbf{f}_{\text{els}}^t), \qquad (4)$$

where each $\mathbf{R}_i$ is a 3×3 nodal rotation matrix. This system is easily solved given the Cholesky factors of $\mathbf{A}$, because the rotation matrices are trivially inverted.

We compute the per-node rotations with Horn's algorithm [1987] on the node's one-ring neighbors. The algorithm finds the least-squares quaternion that best rotates the neighboring nodes from their material space positions to their world space positions. We find that Horn's algorithm yields a very smooth field of per-node rotations, which helps to keep the errors small.

This formulation differs slightly from the original stiffness warping formulation of Müller et al. [2002], wherein the rotations are also applied to the internal forces on the right-hand side. We use the correct forces from the corotational formulation on the right-hand side, and only incur error induced by the left-hand side's approximation of per-element rotations by per-node rotations.

A second difference is that the original method postmultiplies each 3×3 block of $\mathbf{A}$ with the same 3×3 rotation matrix used for premultiplication (albeit transposed), even for blocks not on the diagonal of $\mathbf{A}$. This treatment cannot be written in the form (4) or as a matrix multiplication by a block diagonal matrix. It requires the entries in the matrix to be changed at each time step, and it yields a nonsymmetric matrix.

The errors associated with our approximation are small for objects with low velocities or undergoing deformations with smoothly changing rotations. In those cases stiffness warping as we describe it here might be the preferred method, as it is very fast. In general, though, stiffness warping can introduce undesirable visual artifacts and exhibit poor stability.

4. WARP-CANCELING COROTATION

Section 3 describes the exact corotational method and the fast but flawed stiffness warping method. Here we fuse the two into a fast method that has limited error and fewer problems with stability than stiffness warping. Our goal is to exploit the fast solution times and robustness against ill-conditioning of a direct solver, without incurring the cost of having to recompute the numeric factorization at each time step. The matrix does not change dramatically at each time step, and we are willing to accept small errors that do not cause objectionable visible artifacts.

Warp-canceling corotation includes both per-element and per-node rotations in the system. When a simulation begins with an object in its rest configuration, all the per-element and per-node rotations are the identity, and our method mimics the corotational method with no error. As the elements begin to rotate relative to each other, we perform stiffness warping, updating the per-node rotations at each time step to account for the changes in orientation of the elements as best as possible. We also estimate the errors in intra-element forces incurred by the use of per-node rotations to approximate per-element rotations. For elements in which the error grows too large, we locally update the corresponding per-element rotations, thereby correcting the computation of the local velocities, and closing the gap between the behavior of stiffness warping and the behavior of the exact corotational method. Error accumulates again during subsequent time steps, so we continue to periodically correct some of the per-element rotations as the object continues to deform. Because only a subset of all the per-element rotations are updated at each time step, only the corresponding parts of $\mathbf{A}$ change. To take advantage of the fact that large parts of $\mathbf{A}$ do not change from one step to the next, we perform an incremental update of its Cholesky factor $\mathbf{L}$.

When the error threshold is zero, our method becomes exactly the corotational method, except that it is slightly slower due to the cost of computing per-node rotations. However, our method's strength appears when it permits some error to accumulate, thereby running substantially faster while giving the user a controllable trade-off between accuracy and speed.

The choice concerning which per-element rotations will be updated allows some flexibility in specializing the algorithm's behavior for particular applications. For example, an offline simulation may use a constant error threshold to ensure a level of quality even if the work per frame varies over the course of the simulation. Conversely, a real-time simulation may instead opt to update only as many elements as possible given the available time, thereby maintaining a constant frame rate with variable quality. When elements with large errors are not updated, we can determine which elements might cause stability problems and selectively damp them to preserve stability.

4.1 A Warp-Canceling Formulation

We combine per-node and per-element rotations into a single system where the system matrix $\mathbf{A}$ is bracketed by the orthonormal matrices of per-node rotations, as in (4), and we also apply a per-element rotation to the sub-matrices $\mathbf{K}_e$ and $\mathbf{C}_e$ that are assembled to form $\mathbf{A}$. However, we modify (3) so that the rotations applied to each element also cancel out the per-node rotations. The modified element stiffness matrix is

$$\widetilde{\mathbf{K}}_e = \begin{pmatrix} \mathbf{R}_1^\mathsf{T}\mathbf{R}_e & & & \\ & \mathbf{R}_2^\mathsf{T}\mathbf{R}_e & & \\ & & \mathbf{R}_3^\mathsf{T}\mathbf{R}_e & \\ & & & \mathbf{R}_4^\mathsf{T}\mathbf{R}_e \end{pmatrix} \mathbf{K}_e \begin{pmatrix} \mathbf{R}_e^\mathsf{T}\mathbf{R}_1 & & & \\ & \mathbf{R}_e^\mathsf{T}\mathbf{R}_2 & & \\ & & \mathbf{R}_e^\mathsf{T}\mathbf{R}_3 & \\ & & & \mathbf{R}_e^\mathsf{T}\mathbf{R}_4 \end{pmatrix}, \tag{5}$$

and the modified element damping matrix $\widetilde{\mathbf{C}}_e$ is defined likewise, by replacing $\mathbf{K}_e$ with $\mathbf{C}_e$. These matrices are assembled to form the system matrix $\mathbf{A}$ in (4).

When the system is initially assembled with the correct per-node and per-element rotations, all of the per-node rotations cancel and the system behaves identically to the standard corotational method. However, if the system subsequently deforms but parts of $\mathbf{A}$ are not updated, then the differences between the out-of-date matrix and the correct matrix are approximated by the per-node rotations. Note that the rest of (4), including each $\mathbf{R}_i$ and all terms on the right-hand side, are updated every time step. Only $\mathbf{A}$ is allowed to become stale.

For solid objects the per-element rotations typically vary slowly over the mesh, so the per-node rotations usually approximate them well. However, for thin structures we find that the per-node rotations may change abruptly near wrinkles and creases. When this circumstance occurs, the large differences between the per-node rotations for the nodes of a single element may cause excessive unbalanced forces that can be destabilizing. In these cases, forcing the per-node rotations to the identity, effectively disabling stiffness warping, creates larger linearization errors but improves stability.

4.2 Error Estimation

When parts of $\mathbf{A}$ have not been updated, we can estimate the expected time-integration error, which arises because the per-node rotations only imperfectly approximate the missing updates to the per-element rotations. With respect to the left-hand side of (4), this error is proportional to both the error in the rotations and the local nodal velocities. For each element e we compute $\mathbf{f}_e = \mathbf{R}_n(\mathbf{M}_e + \Delta t\widetilde{\mathbf{C}}_e + \Delta t^2\widetilde{\mathbf{K}}_e)\mathbf{R}_n^\mathsf{T}\dot{\mathbf{x}}_e^t$ with both the correct values of $\widetilde{\mathbf{C}}_e$ and $\widetilde{\mathbf{K}}_e$ and the values that were last updated in $\mathbf{A}$. This operation produces a correct force estimate $\mathbf{f}_{e,\mathrm{cor}}$, and an approximate one $\mathbf{f}_{e,\mathrm{apx}}$. The error is $||\mathbf{f}_{e,\mathrm{cor}} - \mathbf{f}_{e,\mathrm{apx}}||$.

For elements that have excessively large errors, we can perform local updates to $\mathbf{A}$. The rows and columns of $\mathbf{A}$ that depend on these elements' nodes are reassembled using the correct, updated values for $\widetilde{\mathbf{K}}_e$ and $\widetilde{\mathbf{C}}_e$. These adjustments are incorporated into the system matrix $\mathbf{A}$ in a manner that cancels out the corresponding per-node rotations in the linear system (4), and our method is locally restored to the exact corotational formulation. (See Section 4.4 for details about selecting which elements to update.)

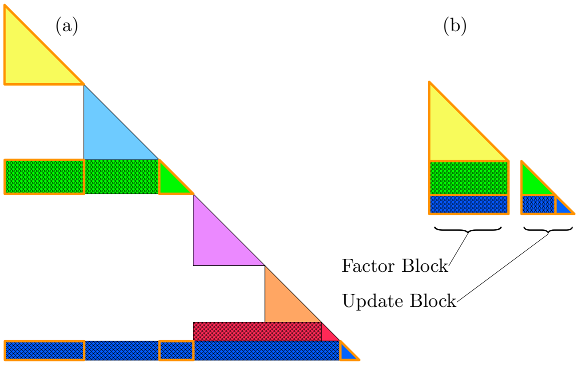


Fig. 3**:** The Cholesky factor $\mathbf{L}$ retains the same hierarchical block pattern (a) that was induced in the system matrix $\mathbf{A}$ by reordering. The supernode for a block in the matrix includes the dependent subblocks from its ancestors in the dissection tree. The relevant subblocks for the yellow leaf-block are outlined in orange. These subblocks are compacted (b) to form the factor and update blocks.

4.3 Incremental Cholesky Factor Updates

An update to the system matrix $\mathbf{A}$ requires a corresponding update to the Cholesky factor $\mathbf{L}$. Rather than recompute $\mathbf{L}$ from scratch, we instead perform an incremental update that does work in accordance with the amount that $\mathbf{A}$ has changed.

As part of nested dissection, recursive bisection finds a separator that disconnects the mesh into two subdomains of roughly equal sizes. These subdomains are bisected in turn, yielding a hierarchy of separators with small subdomains at the leaves of the dissection tree. (See Fig. 2.) Nested dissection orders the rows and columns of $\mathbf{A}$ and $\mathbf{L}$ according to a postorder traversal of the hierarchy, with each subdomain and separator preceding its ancestors and with the root separator coming last.

This reordering creates a sparse blocked, or "arrowhead," structure in $\mathbf{A}$. The Cholesky factor $\mathbf{L}$ encodes a process of Gaussian elimination that eliminates degrees of freedom according to this reordering. Consideration of this structure in the context of the standard Cholesky-Crout or Cholesky-Banachiewicz algorithms reveals that the lower-left half of this symmetric arrowhead structure is preserved in $\mathbf{L}$. (See Fig. 3.) Furthermore, the computation of each leaf block in $\mathbf{L}$ depends only on the corresponding leaf block in $\mathbf{A}$, and the computation of an interior separator block in $\mathbf{L}$ depends only on the corresponding block in $\mathbf{A}$ and its children in the dissection tree. If an update to $\mathbf{A}$ changes only a few blocks, then only the corresponding blocks and their ancestors in $\mathbf{L}$ need to be recomputed.

Our implementation of this incremental update strategy is built on the software TAUCS [Toledo 2003]. To gain efficiency by using optimized BLAS and LAPACK routines, TAUCS divides the columns of $\mathbf{L}$ into dense matrix subblocks, termed *supernodes*. Although not all of the entries in these subblocks are nonzero, there is a net gain in speed from using dense matrix libraries despite the loss from performing unnecessary computation on zeros. Typically, there is one supernode for each leaf subdomain and one for each separator, although sometimes a large subdomain or separator is represented by several supernodes to attain better cache behavior. The columns in a supernode are compacted into a *factor block* that omits all-zero rows and is stored in a dense matrix representation. The factor blocks are conjoined with *update blocks* that store the factor blocks' contribution to supernodes further up in the hierarchy. (See Fig. 3.)

The hierarchy of subdomains and separators implies a hierarchy of supernodes. An update to a supernode necessitates updates of all its ancestor supernodes, propagating up to the root. During TAUCS's standard supernodal Cholesky factorization, the update blocks are discarded after they are used, but we preserve them so that the factorization can be restarted at any supernode in the hierarchy. They consume some storage space, but allow us to update a Cholesky factor considerably faster than we could if we recomputed them. The update cost of our method is determined by the amount of the Cholesky factor that is modified, but at worst it is the same as the cost of the original factorization.

This procedure relies on the separators being small enough to make reasonably sized supernodes. A d-dimensional n-node mesh generally has a separator of size $\Theta(n^{(d-1)/d})$, which is sublinear and typically small for two- and three-dimensional structures. Moreover, many interesting physical structures contain natural separators that are substantially smaller than the theoretical worst case.

4.4 Scheduling Matrix Updates

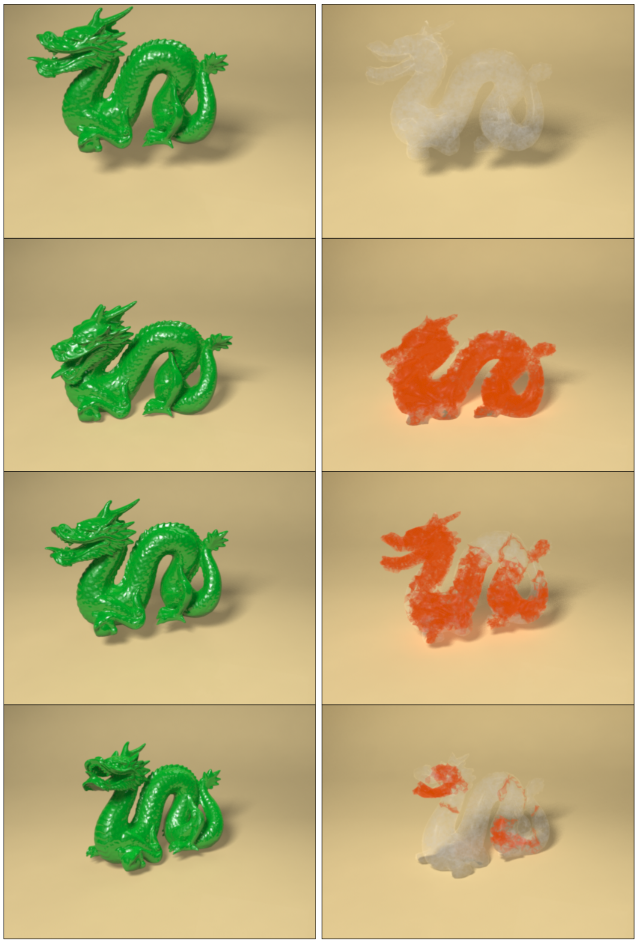
Initially $\mathbf{A}$ is exact, and our method exactly reproduces standard corotational behavior. However, as the simulation progresses, blocks in $\mathbf{A}$ become incorrect and introduce error. Updating $\mathbf{A}$ has a small cost, but we cannot update $\mathbf{A}$ without corresponding, potentially costly, updates to $\mathbf{L}$. We therefore need a criterion for deciding which elements to update during a time step, which entails choosing a trade-off between correctness and speed. A feature of our approach is that different criteria can be developed for the needs of different applications. Here we describe two criteria: *target-quality* and *target-time* update schedules.

Because the unit of recomputation we work with is a supernode, when we update a single element in a supernode we also update all the other elements in that supernode. The *badness* of a supernode is the maximum error over all the elements in that supernode.

The computations required to update a given supernode are fixed and do not depend on the particular numerical values of $\mathbf{A}$. We can therefore expect that the time required to update a given supernode will be roughly the same every time, with small variations due to cache occupancy and other machine-specific behavior. When we compute the initial factorization we record the time required to factor each supernode. This measurement provides an estimate of how long it would take to update that supernode. We refer to this time estimate as the supernode's *cost*. It is refined each time the supernode is updated using an exponentially weighted moving average.

For *target-quality* updates we examine each supernode and compare its badness score with a threshold τ. Every supernode above the threshold is marked for update, as are its ancestors in the dissection tree. Once the supernodes have been marked, we perform a postorder tree traversal and update the marked supernodes as they are visited.

For *target-time* updates we specify a total time budget for the update. We sort the supernodes by badness and compute the total update cost for each supernode, which is the sum of its update cost and the cost of all its ancestors that have not already been marked for update. Then the supernodes are examined in order of their badness, starting with the worst. If a supernode's total update cost is below the remaining budget, then it and all its ancestors are marked for update and the remaining budget is decreased by the supernode's total update cost. For each supernode freshly marked for update, the total update costs of its descendants are updated to reflect that the marked supernode and its ancestors have already been paid



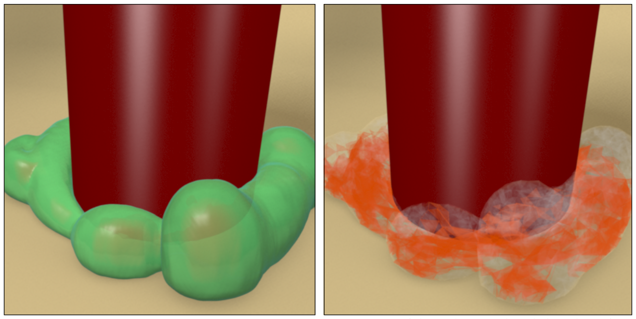
Images copyright Hecht, Lee, Shewchuk, and O'Brien.

Fig. 4**:** The images in the left column are taken from an animation of a deformable dragon model dropped onto a flat surface. The dragon compresses and then bounces. The images in the right column indicate in orange the locations of the elements that are being updated.

for. Supernode marking continues until no unmarked supernode fits in the remaining budget, at which point we perform a postorder tree traversal and update the marked supernodes as they are visited. To be effective, this update schedule requires that there is no supernode whose update cost exceeds the total budget.

A potential drawback of the target-time approach is that the worst uncorrected error could be arbitrarily bad, and extremely bad elements can cause stability problems. One solution is a hybrid approach where the target-quality algorithm is used with a very high badness threshold to mark supernodes that cannot be ignored. The target-time algorithm is then allowed to mark additional nodes using any remaining time budget.

A second solution is to apply selective damping to excessively bad elements that cannot be updated. This damping can be implemented by multiplying the nodal velocities of the offending elements by a scale factor less than one, which becomes smaller for larger errors. This type of damping is very inexpensive but creates strong visual artifacts. A slightly more expensive approach that tends to look better is to damp offending supernodes towards a local rigid-body motion chosen by shape matching [Müller et al. 2005].



Images copyright Hecht, Lee, Shewchuk, and O'Brien.

Fig. 5**:** The left image is taken from an animation in which a deformable bear model is compressed between the ground plane and a rigid cylinder. Many of the elements under the cylinder are forced into degenerate or inverted configurations, but the simulation remains stable and the object returns to its original configuration when the cylinder is removed. The right image indicates the locations of updates to the system matrix.

5. RESULTS AND DISCUSSION

We have implemented our algorithm for warp-canceling corotation with incremental Cholesky updates and tested it with several physical scenarios. We compare it with the most widely used alternative, the conjugate gradient method with a Jacobi preconditioner. Animations rendered from these simulations appear in the supplemental video.

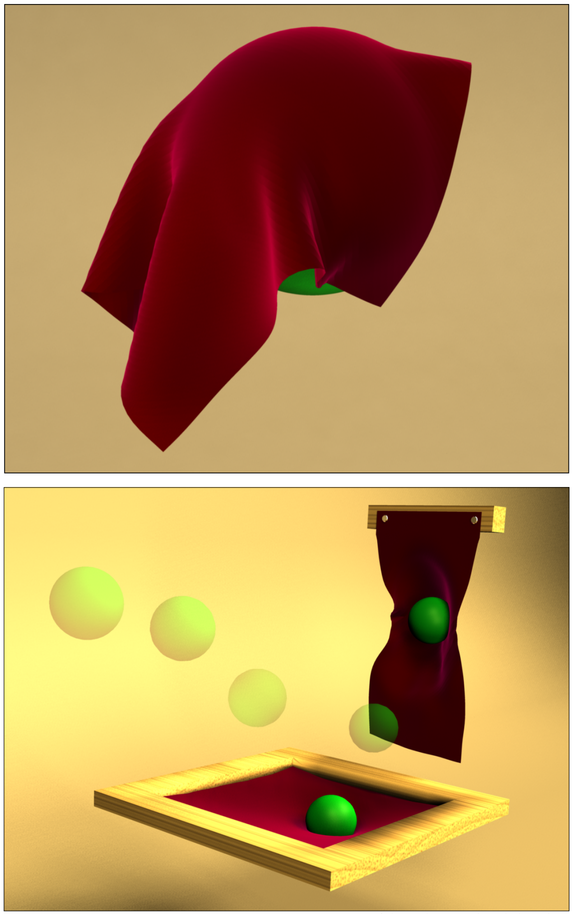
Fig. 4 shows frames from an animation of the Stanford Dragon being dropped onto a rigid ground plane. The nested dissection hierarchy for this model is shown in Fig. 1 with the sparse structure of the reordered system matrix. For this target-quality simulation, the number of updates varies widely from frame to frame, as the right column of Fig. 4 shows.

5.1 Numerical Robustness

A significant advantage of the corotational formulation is that when the per-element rotations are computed accurately the simulation becomes robust to element inversion and remains stable even for extreme deformations. In Fig. 5, a deformable bear is compressed between the ground plane and a cylindrical piston, and many of the trapped elements become degenerate or inverted. Nevertheless, the simulation remains well-behaved and the bear returns to its original shape when the piston rises.

Another way in which a direct solver is more robust than an iterative solver is that the solution time does not depend on the conditioning of the system matrix. Useful motion can be obtained even when the input mesh contains very poor quality elements, severe grading, or widely varying material properties, all of which can cause a poorly conditioned system matrix.

The $1\ \mathrm{m} \times 1\ \mathrm{m}$ sheet shown atop Fig. 6 is modeled with a tetrahedral mesh that is only $0.1\ \mathrm{mm}$ thick. The in-plane width of the elements averages roughly $2\ \mathrm{cm}$, which is 200 times their out-of-plane thickness. Although each element in the mesh is nearly degenerate, the system behaves well, with the sheet draping realistically around the sphere. Because the mesh has a small but nonzero thickness, it exhibits correct bending behavior and, unlike a triangulated surface, does not require special treatment of bending forces. The object's resistance to bending arises naturally because out-of-plane bending induces expansion and compression through the layers of the sheet. The bottom half of Fig. 6 shows tetrahedral meshes of similar design used to model a trampoline and a hanging sheet.



Images copyright Hecht, Lee, Shewchuk, and O'Brien.

Fig. 6: The top image shows a sheet draping over an immobile sphere. Although the sheet appears to be two-dimensional, it is actually three-dimensional and modeled by a mesh of extremely flat tetrahedra. Despite the elements' poor aspect ratios, the simulation remains stable and exhibits proper bending behavior. In the bottom image, tetrahedral meshes model a trampoline and a hanging sheet, both of which are struck by a moving projectile.

The trampoline is fixed along its edges, and the hanging sheet is fixed along its top edge. A rigid ball is launched onto the trampoline, bounces into the hanging sheet, and slides down it, causing the sheet to curl and swing.

The draping and swinging sheets are examples where the wrinkles can cause large differences in the per-node rotations across a single element. The consequent large force imbalance can be destabilizing, so as discussed in Section 4.1, we force the per-node rotations to the identity. Without the incremental updates, this solution would exhibit severe linearization artifacts. However, with our update scheme the linearization errors never grow to the point where they are noticeable.

The mesh for the spider example in Fig. 7 has very thin structures that model the legs and support thread. These thin structures contain poorly shaped tetrahedral elements that contribute to a poorly conditioned system matrix. The poor conditioning is further exacerbated because the legs and body are made of a material roughly 300 times stiffer than the materials for the joints and thread. Despite the poor conditioning, the solver produces correct behaviors.

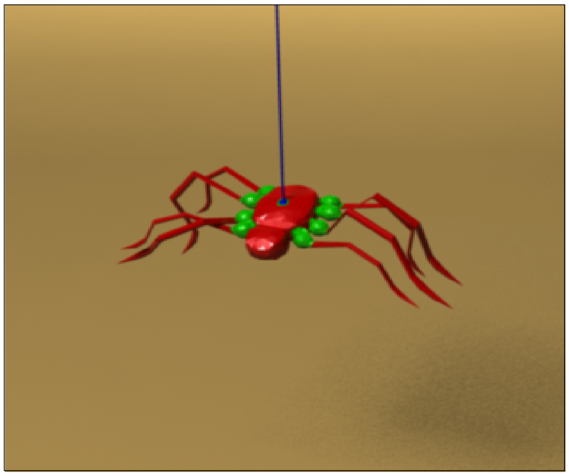


Image copyright Hecht, Lee, Shewchuk, and O'Brien.

Fig. 7: A frame from an animation of a toy spider on the end of an elastic line. The spider contains thin structures such as legs and a thread that are modeled with tetrahedral elements having very high aspect ratios. The parts of the spider are made of different materials, with the legs and the body being about 300 times stiffer than the joints and the thread.

For both the sheet and spider animations, the preconditioned conjugate gradient solver behaves poorly. It takes an exceedingly large number of iterations to converge to a reasonable error, and quite often it fails and the time integrator blows up. A better preconditioner might mitigate this problem, but a strong preconditioner would incur a substantially higher computational cost.

The capability to use meshes with poor elements provides two important advantages. First, tetrahedral meshes whose elements all have good quality are hard to generate; tolerance of a few bad elements eases the task substantially. Second, structures including solids, membranes, and thin threads can be modeled in a unified framework without explicit bending or torsion forces. For objects that can be obviously categorized as one- or two-dimensional structures, this generality comes at some cost compared to specialized methods [Bridson et al. 2003; Grinspun et al. 2003; Bergou et al. 2008]. However, as Martin et al. [2010] point out, in many contexts the flexibility to model all these structures in a unified framework may be worth a small loss in efficiency.

5.2 Running Times and Scalability

To assess the speed of our method, we ran several simulations with different choices of parameters and measured the running times. Our implementation uses double precision floating-point arithmetic for all computations. The measurements were performed on a Linux machine with 32 GB of RAM and two quad-core Intel Xeon X5450 processors running at 3.0 GHz. The timing numbers we present were measured with 8 execution threads enabled.

Fig. 8 shows a geometric sculpture discretized at three different resolutions. We simulated the sculpture dropping and bouncing on the ground plane several times, with 300 frames of animation per simulation. We ran this test with both a Jacobi preconditioned conjugate gradient (CG) solver and our warp-canceling method for each of the three resolutions and for materials of three different stiffnesses. For both CG and our method, we selected the error tolerance so that the resulting motion was visually indistinguishable from a reference motion computed with a full Cholesky factoriza-

Mesh		$E = 5 \times 10^6$ Pa		$E = 5 \times 10^7$ Pa		$E = 5 \times 10^8$ Pa	
Tets	Nodes	CG	WC	CG	WC	CG	WC
7K	2K	34 ms	39 ms	53 ms	25 ms	88 ms	25 ms
115K	23K	2.4 s	740 ms	**6.3 s**	**585 ms**	**11.7 s**	**421 ms**
329K	65K	6.9 s	2.9 s	18.0 s	2.4 s	**34.5 s**	**1.9 s**

Table I. **:** Comparison of the average per-frame running times for simulations of the sculpture from Fig. 8 dropping onto and bouncing off of a rigid ground plane. Simulations were run for each of the three different mesh resolutions with each of three different values for Young's modulus E. The sculpture's density is $\rho = 1{,}000\ \mathrm{kg/m^3}$, its radius is 0.8 m, and Poisson's ratio is $\nu = 0.4$. Times were measured for both a preconditioned conjugate gradient solver (CG) and our warp-canceling method (WC). Breakdowns of the bold entries appear in Table II.

tion at each time step. For CG this was a residual of less than 10^{-4}, and for our method the badness threshold was $\tau = 5 \times 10^{-3}$.

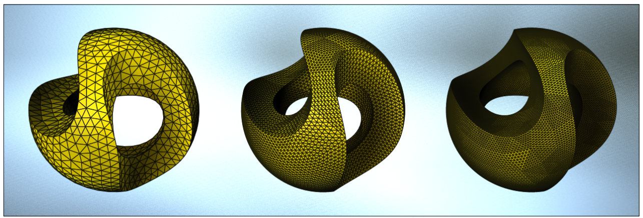
The data from these tests are summarized in Table I, which lists the average of total computation times of the solver. Table II provides a breakdown for selected examples to show how much time is used by each step of the algorithms.

Although our method is slightly slower than the preconditioned conjugate gradient method for very soft materials simulated at low resolutions, it is more than 15 times faster for larger meshes with stiffer materials. Higher-resolution meshes have a greater separation between their lowest-frequency vibrational modes and their highest-frequency modes, so a conjugate gradient solver requires more iterations to converge, whereas our warp-canceling method scales better to large linear systems.

As materials become stiffer, the conditioning of the stiffness matrix worsens and the conjugate gradient solver slows down, but our warp-canceling method actually becomes faster. This improvement occurs because updating the system matrix and its Cholesky factor becomes less expensive with increased stiffness—stiffer objects deform less and deform more smoothly, so fewer updates are required. The speed of the other steps of our algorithm is independent of the material stiffness. Table II quantifies these observations.

More timing information is provided in Tables IV and V. Table IV lists running times of tests where we systematically vary the update strategy; it is discussed in Section 5.4. Table V contains data from the other simulations discussed in this article.

We tested the suitability of a low-rank update method by creating a variant of our simulator in which our Cholesky update is replaced by the software CHOLMOD, discussed in Section 2. The term low-rank is relative, but even for moderately-sized meshes the changes in our formulation typically have rank 1,000 or greater. For our simulations, we find that updates by CHOLMOD are generally much slower than recomputing the factorization from scratch. For example, on the medium-sized sculpture mesh with 115,712 tetrahedra, our method completes the equivalent of a rank-614,844 modification (51,237 updated elements, 44% of the mesh) in 787 ms whereas CHOLMOD takes 19.9 min. This time is substantially slower than refactoring from scratch, which takes 1 s. Even updating 34 elements (a rank-408 modification) with CHOLMOD takes 1.1 s, longer than refactoring from scratch. We emphasize that these timings reflect no weakness of CHOLMOD, which excels when a global change to the matrix can be expressed as a true low-rank change. Furthermore, CHOLMOD works with any Cholesky factor and is not limited to those produced by nested dissection. However, CHOLMOD is not a good fit to our high-rank updates.



Images copyright Hecht, Lee, Shewchuk, and O'Brien.

Fig. 8**:** A geometric sculpture discretized with three meshes of different resolutions. From left to right, the meshes contain 6,681, 115,712, and 329,131 tetrahedra.

	Dragon		Sculpture Med Med		Sculpture Stiff Med		Sculpture Stiff Large	
	CG	WC	CG	WC	CG	WC	CG	WC
Elem. Rot.	2	2	10	12	10	13	37	38
Node Rot.		7		29		27		78
Error Est.		13		58		60		167
Mat. Upd.		3		26		12		36
Chol. Upd.		12		281		121		1,002
Assembly	29		141		140		420	
RHS	8	8	32	33	34	36	94	95
Solve	199	29	6,117	139	11,508	148	33,867	438
Integration	237	78	6,312	585	11,722	421	34,509	1,862
Total	270	110	6,312	585	11,722	421	34,509	1,862

Table II. **:** Breakdown of the running time (in milliseconds) spent in different stages of the time integrator for both a preconditioned conjugate gradient solver (CG) and our warp-canceling method (WC). For the CG solver, the stages are computing per-element rotations, assembling the system matrix, computing the right-hand side (RHS), and solving the system. For our WC solver, the stages are computing per-element rotations, computing per-node rotations, estimating the error for each element, updating the system matrix, updating the Cholesky factorization, computing the right-hand side, and solving the system. The last rows list the total time for the time integrator and the total time per frame including collision detection and other overhead. The examples in the table are the dragon drop from Fig. 4 and the sculpture drop from Fig. 8. The three sculpture examples are the 115K mesh with $E = 10^7$ Pa, the 115K mesh with $E = 10^8$ Pa, and the 329K mesh with $E = 10^8$ Pa, corresponding to the bold entries in Table I.

5.3 Parallel Scaling

Our implementations of both the preconditioned conjugate gradient solver and our warp-canceling method are parallelized with OpenMP and pthreads. Most components of the algorithms, such as per-element or per-node rotation computations and sparse matrix-vector multiplications, are easily parallelized with OpenMP. However, our incremental Cholesky factorization update poses difficulties because the tree structure of the supernodes creates dependencies among the computations. Our factorization implementation maintains a work queue of supernodes whose dependencies have been met. This queue is initially filled with the leaf supernodes. Work progresses up the hierarchy and ends with the root node being processed last. Every node has a dependency counter, which is initialized to its number of children. When a supernode's work is completed, the dependency counter for its parent is decreased; when the counter reaches zero, the parent node is placed in the work queue. The queue is serviced by a pool of worker threads that each process a single supernode at a time.

Table III summarizes the speedup obtained by our implementation as the number of allocated threads varies. Some operations, such as computing per-node and per-element rotations, scale fairly well. Others, such as the linear system solution, scale poorly; the

	Number of Threads						
	1	2		4		8	
Element Rotations	55	28	2.0×	14	3.9×	11	5.0×
Node Rotations	24	14	1.7×	7	3.4×	6	4.0×
Error Estimation	184	122	1.5×	69	2.7×	61	3.0×
Matrix Update ($\mathbf{A}$)	201	122	1.6×	72	2.8×	52	3.9×
Cholesky Update ($\mathbf{L}$)	1419	832	1.7×	520	2.7×	399	3.6×
Right-Hand Side	70	59	1.2×	34	2.1×	35	2.0×
Linear Solution	255	174	1.5×	149	1.7×	149	1.7×
Total	2257	1385	1.6×	885	2.6×	728	3.1×

Table III. **:** Running times in milliseconds for different stages of the time integrator with a varying number of execution threads for the dropped sculpture simulation with a 115K tetrahedron mesh and $E = 10^6$ Pa.

forward and back substitution threads run out of parallel tasks as they near the root of the supernode hierarchy. Away from the root, the amount of work per supernode is relatively small. The same problem occurs for the more expensive factor update, but the solver touches the entire tree while the the factor update only touches the parts of the tree that need to be recomputed. With eight threads, our implementation becomes more strongly impacted by memory latency. We believe that with additional effort, perhaps by starting from scratch instead of adapting TAUCS, better parallel scaling could be realized.

5.4 Effects of the Update Strategy

We ran a series of experiments where an object with the topology of a three-torus was dropped onto a fixed cylinder and ground plane (Fig. 9). The three-torus has 19,172 tetrahedral elements and 5,046 nodes. We ran the simulation with both a stiffer ($E = 10^7$ Pa) and a softer ($E = 10^6$ Pa) material, with several different solver schemes. The data from these simulations appear in Table IV.

For the stiffer material, the conjugate gradient solver is slower than both full Cholesky factorization and our warp-canceling method, with our method being about three times faster than conjugate gradients. For the softer material, the conjugate gradient solver is faster than full Cholesky, but our method outperforms both.

However, our method can be further accelerated by limiting the maximum budget allowed for updates. This cost reduction comes at a cost of greater error relative to the reference simulation produced by the full Cholesky solver, but the results are nonetheless plausible and our solver remains stable. When the maximum budget is reduced to 33% of that required for a full factorization, our method is roughly twice as fast as conjugate gradients even for this compliant material. If Cholesky updates are completely disabled, our method becomes a variant of stiffness warping, and damping is needed to preserve its stability. Specifically, we damp elements with badness greater than 5×10^{-1} toward a local rigid body motion. Although disabling updates improves the speed somewhat, the damping is visually apparent as sluggish motion.

The control that update scheduling affords our method enables a trade-off between accuracy and speed that was not previously possible with direct solvers. Iterative methods have always enabled some ability to trade accuracy for speed by limiting the number of iterations or increasing the residual tolerance, but if too much error is permitted the time integration method becomes unstable.

The simulation of a bowl of colliding bears in Fig. 10 uses a compliant material ($E = 3 \times 10^6$ Pa) and includes many collisions. There are 25 bears in the simulation, comprising a total of 244,625 tetrahedral elements and 55,400 nodes. Simulations like these are inherently chaotic because of the many collisions between the objects. Even small differences in the numerical solution of the linear system lead to very different final configurations. This sensitivity means that even with small error tolerances, all three methods—conjugate gradients, full Cholesky, and our warp-canceling method—produce simulations that have very different rest configurations but nevertheless look plausible.

Both conjugate gradients and our warp-canceling method can be sped up for this example. For conjugate gradients, we were able to increase the residual tolerance to 10^1. For our method, we increased the badness threshold to $\tau = 10^1$ and we limited the update budget to 20% of the cost of a complete factorization. As Table IV shows, these changes speed up both conjugate gradients and our method by more than a factor of two (compare the low-tolerance and high-tolerance timings). However, our method outperforms conjugate gradients by a substantial margin for both accuracy settings.

6. CONCLUSIONS, LIMITATIONS, FUTURE WORK

Direct solvers, such as Cholesky factorization, have strong advantages: they are robust against ill-conditioned matrices, and solving by using the factorization is much faster than an iterative solver like conjugate gradients. Their weakness, of course, is that computing the factorization is quite slow, and is therefore problematic for simulations in which the linear system matrix changes every time step, as it does with corotational finite element methods. Our method directly addresses this weakness by allowing an existing factorization to be incrementally updated as the simulation progresses.

The advantage of our warp-canceling method over the conjugate gradient method grows as the materials become stiffer and the meshes larger. Our solution time improves moderately with stiffer materials, whereas the conjugate gradient method slows down dramatically. Our method enables a trade-off between correctness and speed, and attains good results over a wide range of elastic materials and objects. Its errors are far less visible than those of stiffness warping.

Our method can simulate everything the exact corotational method can, usually considerably faster, but the exact method still has a place in simulation. We discuss here four limitations of our method: the fact that it is approximate, its memory use, the effort required to implement it, and the difficulty of incorporating implicit constraints or a varying time step.

The central premise of our method is to trade accuracy away for speed in the context of a direct solver, and it is inherently an approximate method. We have demonstrated substantial speedups compared to both full Cholesky factorization and the conjugate gradient method, at the cost of incurring errors. For a carefully chosen trade-off between speed and accuracy, these errors are not visible. Even when the errors are large enough to become visible, our method remains well behaved unlike conjugate gradient solvers, which can become unstable when run with a high error tolerance.

The memory occupied by a Cholesky factor is large, and we use additional storage space to save the update blocks, further reducing the size of mesh that can be treated in memory. For very large problems, over 300,000 tetrahedra, the memory overhead slows our implementation down enough that its advantage over a conjugate gradient solver lessens for very soft materials, though it remains superior for stiff materials. For example, the full system matrices for the small, medium, and large sculpture require 1.5 MB, 22 MB, and 62 MB, respectively. The sizes of the corresponding Cholesky factors are 7.8 MB, 268,MB, and 956 MB, respectively. As we retain the update blocks, we need to store additionally 14.8,MB, 884,MB, and 3,406,MB. A standard supernodal Cholesky factorization would only need to store a small subset of these update

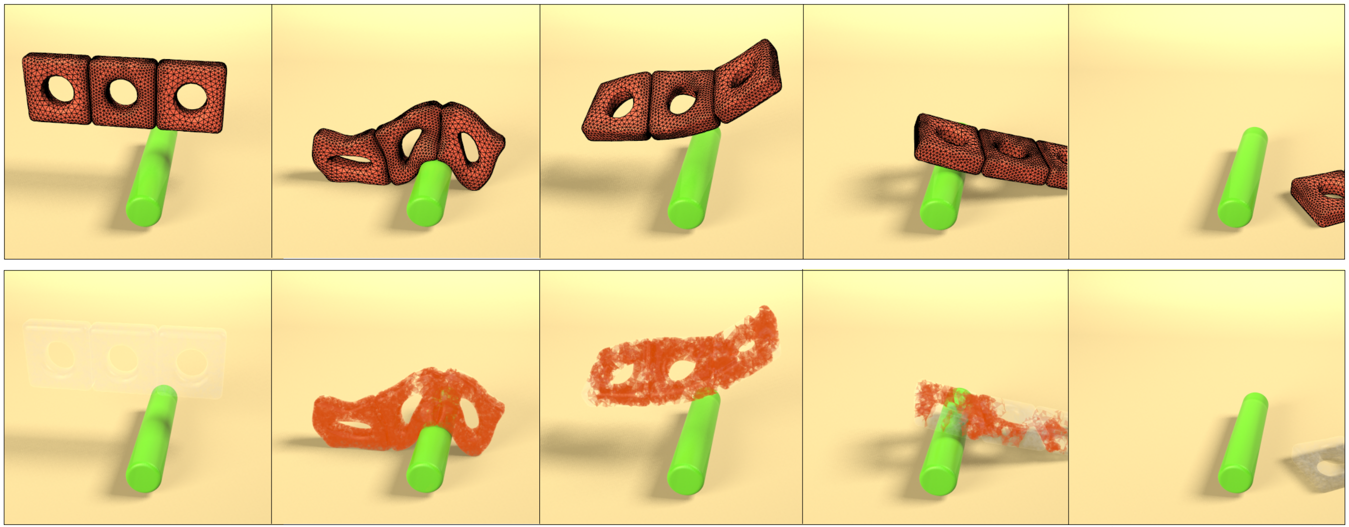

Images copyright Hecht, Lee, Shewchuk, and O'Brien.

Fig. 9: Frames from an animation of an elastic three-torus dropping onto an immobile cylinder. In the top row, the mesh is rendered as a wireframe overlay. In the bottom row, the object is rendered transparent with the elements being updated in each frame highlighted in red.

Object	Method	Minimum	Maximum	Average
3-Torus (stiff)	Full Cholesky	126 ms	237 ms	157 ms
3-Torus (stiff)	CG	50 ms	577 ms	307 ms
3-Torus (stiff)	WC (unlimited)	37 ms	207 ms	107 ms
3-Torus (soft)	Full Cholesky	123 ms	271 ms	146 ms
3-Torus (soft)	CG	43 ms	171 ms	121 ms
3-Torus (soft)	WC (unlimited)	36 ms	194 ms	87 ms
3-Torus (soft)	WC (66% limit)	35 ms	172 ms	72 ms
3-Torus (soft)	WC (33% limit)	33 ms	127 ms	65 ms
3-Torus (soft)	Damped SW	34 ms	111 ms	52 ms
Bowl of Bears	Full Cholesky	2,032 ms	2,214 ms	2,085 ms
Bowl of Bears	CG Low Tol	4,322 ms	5,536 ms	4,689 ms
Bowl of Bears	CG High Tol	1,586 ms	2,136 ms	1,895 ms
Bowl of Bears	WC Low Tol	556 ms	1,693 ms	1,594 ms
Bowl of Bears	WC High Tol	488 ms	861 ms	728 ms

Table IV. : Running times for two different simulations computed by different solvers and variations of update scheduling. Images from the simulations appear in Figs. 9 and 10. Times were measured for a full Cholesky solver, a conjugate gradient solver (CG), and our warp-canceling method (WC). The residual tolerance for CG was 10^1 for the high-tolerance Bowl of Bears and 10^{-4} for all other CG simulations. The badness threshold for WC was 10^1 for the high-tolerance Bowl of Bears and 5×10^{-3} for all other WC simulations. For the 3-torus with a stiffer material, the WC method had no budget limit. The 3-torus with a softer material was run multiple times with the WC method; the budget limit was a percentage of the time required for a full Cholesky factorization. When this budget is zero, our method is a modified version of stiffness warping (SW), which we damp for stability.

Object	Tets	Nodes	τ	Min	Max	Average
Bear Squash	30,490	7,361	10^{-2}	61 ms	355 ms	157 ms
Cloth Drop	15,000	5,202	10^{-4}	25 ms	209 ms	77 ms
Trampoline	22,500	7,854	10^{-5}	39 ms	219 ms	92 ms
Spider	12,683	3,991	10^{-4}	29 ms	183 ms	79 ms

Table V. : Running times for our simulations not covered by other tables (Figs. 5, 6, and 7). For each scenario, our warp-canceling method had no limit on the update budget. The badness threshold is τ.

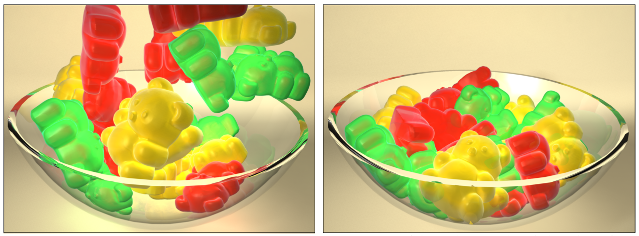

Images copyright Hecht, Lee, Shewchuk, and O'Brien.

Fig. 10: The left image shows deformable bears dropping into a glass bowl. Their final rest configuration appears at right.

blocks at any one time, but during the computation it would update the same amount of memory.

The supernodal direct solver and updates are substantially more complicated to understand and implement than the conjugate gradient method. However, with help from readily available optimized dense matrix subroutines, the implementation is less difficult than it might appear. Given the existing Cholesky solver in TAUCS, our greatest effort was to add new code to update the matrices.

The need to preserve the linearized system matrix from time step to time step prevents us from using time steps of varying lengths, which can improve the stability of some simulations. It also makes it difficult to incorporate either implicitly integrated penalties or Lagrangian constraints. The exact corotational method easily incorporates implicitly integrated forces, which robustly simulate large impulses, because the system matrix is computed from scratch at every time step. Lagrangian constraints typically produce an indefinite system which is incompatible with Cholesky factorization. We do not want to make large transient changes to the matrix and its Cholesky factors, so we instead use explicitly integrated collision forces, which are less stable and harder to choose parameters for.

In future work, we hope to address this constraint limitation. We believe that implicit constraints can be enforced with an additional block around the system matrix, which is included in the forward and back substitution and is updated and refactored whenever the constraints change.

Although we have realized substantial improvements with our warp-canceling solver, there is still a large design space that could be further explored. Our initial tests indicate that our approximate factorization makes an excellent preconditioner for the conjugate gradient method, but we find the combination to be slower than either method alone. Possibly some other variation, such as incremental update of an incomplete Cholesky factorization, would perform well. We also did not explore iterative refinement [Li and Demmel 1999]; it is possible that the time spent on iterative refinement might yield a net savings by necessitating fewer updates. We note that our badness metric simply measures expected error; perhaps some other quantity would allow more selective updates.

It is interesting to ask whether the method can be extended to support mesh modifications and the corresponding changes to the sparse structure of the system matrix and its symbolic factorization. These capabilities would support effects like fracture, cutting, extreme plastic flow, and the merging of viscous materials.

It is also interesting to ask whether there are other physical phenomena described by nonlinear partial differential equations whose structure admits a linearization that changes slowly enough to benefit from our methods, or can be made to change slowly with the adoption of an extra trick like the use of per-node rotations to approximate per-element rotations. For example, aerodynamics simulations can be notoriously nonlinear, but they sometimes reach quiescent states where the linearized system changes slowly. Looking more broadly, applications other than finite element simulations might also benefit from the same principles of incremental updates and partial refactoring.

ACKNOWLEDGMENTS

We thank Xiaoye Li and James Demmel for helpful discussions and commentary about sparse direct solvers. We thank Martin Wicke for his suggestion to simulate extremely thin objects with tetrahedra. The Stanford Dragon model was provided by the Stanford Computer Graphics Laboratory's 3D Scanning Repository.

REFERENCES

BARRETT, R., BERRY, M., CHAN, T., DEMMEL, J., DONATO, J., DONGARRA, J., EIJKHOUT, V., POZO, R., ROMINE, C., AND VAN DER VORST, H. 1993. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania.

BELYTSCHKO, T. AND HSIEH, B. 1979. Application of higher order corotational stretch theories to nonlinear finite element analysis. *Computers & Structures 11*, 175–182.

BERGOU, M., WARDETZKY, M., ROBINSON, S., AUDOLY, B., AND GRINSPUN, E. 2008. Discrete elastic rods. *ACM Transactions on Graphics 27,* 3 (Aug.), 63:1–63:12.

BOTSCH, M., BOMMES, D., AND KOBBELT, L. 2005. Efficient linear system solvers for mesh processing. In *IMA Conference on the Mathematics of Surfaces*. Springer, 62–83.

BRIDSON, R., FEDKIW, R., AND MULLER-FISCHER, M. 2006. Fluid simulation: SIGGRAPH 2006 course notes. In *ACM SIGGRAPH 2006 Courses*. 1–87.

BRIDSON, R., MARINO, S., AND FEDKIW, R. 2003. Simulation of clothing with folds and wrinkles. In *2003 Symposium on Computer Animation*. 28–36.

CHEN, Y., DAVIS, T. A., HAGER, W. W., AND RAJAMANICKAM, S. 2008. Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. *ACM Transactions on Mathematical Software 35*, 22:1–22:14.

CHENTANEZ, N., ALTEROVITZ, R., RITCHIE, D., CHO, L., HAUSER, K. K., GOLDBERG, K., SHEWCHUK, J. R., AND O'BRIEN, J. F. 2009. Interactive simulation of surgical needle insertion and steering. *ACM Transactions on Graphics 28,* 3 (Aug.), 88.1–88.10. Special issue on Proceedings of SIGGRAPH 2009.

CHOI, M. G. AND KO, H.-S. 2005. Modal warping: Real-time simulation of large rotational deformation and manipulation. *IEEE Transactions on Visualization and Computer Graphics 11,* 1, 91–101.

CHOLESKY, A.-L. 1910. Sur la résolution numérique des systèmes d'équations linéaires. Manuscript. Subsequently published in *Bulletin de la Sabix 39*, 81–95, 2005.

COURTECUISSE, H., ALLARD, J., DURIEZ, C., AND COTIN, S. 2010. Asynchronous preconditioners for efficient solving of non-linear deformations. In *Seventh Workshop on Virtual Reality Interaction and Physical Simulation*. 59–68.

DAVIS, T. A. 2006. *Direct Methods for Sparse Linear Systems*. Fundamentals of Algorithms, vol. 2. SIAM.

DAVIS, T. A. AND HAGER, W. W. 1999. Modifying a sparse Cholesky factorization. *SIAM Journal on Matrix Analysis and Applications 20,* 3, 606–627.

DAVIS, T. A. AND HAGER, W. W. 2009. Dynamic supernodes in sparse Cholesky update/downdate and triangular solves. *ACM Transactions on Mathematical Software 35,* 4.

ETZMUSS, O., KECKEISEN, M., AND STRASSER, W. 2003. A fast finite element solution for cloth modelling. In *Proceedings of the 11th Pacific Conference on Computer Graphics and Applications*. 244–251.

FELIPPA, C. 2007. Introduction to finite element methods. http://www.colorado.edu/engineering/cas/courses.d/NFEM.d. Course notes published as web pages.

GEORGE, A. 1973. Nested Dissection of a Regular Finite Element Mesh. *SIAM Journal on Numerical Analysis 10,* 2 (Apr.), 345–363.

GIBSON, S. F. F. AND MIRTICH, B. 1997. A survey of deformable modeling in computer graphics. Tech. Rep. TR97-19, Mitsubishi Electric Research Laboratory. Nov.

GILL, P. E., GOLUB, G. H., MURRAY, W., AND SAUNDERS, M. A. 1974. Methods for modifying matrix factorizations. *Mathematics of Computation 28,* 126 (Apr.), 505–535.

GOLUB, G. H. AND VAN LOAN, C. F. 1996. *Matrix Computations*, Third ed. The Johns Hopkins University Press.

GOULD, N. I. M., SCOTT, J. A., AND HU, Y. 2007. A numerical evaluation of sparse direct solvers for the solution of large sparse symmetric linear systems of equations. *ACM Transaction on Mathematical Software 33,* 2 (June), 1–32.

GRCAR, J. F. 2011. John von Neumann's analysis of Gaussian elimination and the origins of modern numerical analysis. *SIAM Review 53,* 4 (Nov.), 607–682.

GRINSPUN, E., HIRANI, A. N., DESBRUN, M., AND SCHRÖDER, P. 2003. Discrete shells. In *2003 Symposium on Computer Animation*. 62–67.

HESTENES, M. R. AND STIEFEL, E. 1952. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards 49*, 409–436.

HORN, B. K. P. 1987. Closed-form solution of absolute orientation using unit quaternions. *Journal of the Optical Society A 4,* 4 (Apr.), 629–642.

IRVING, G., TERAN, J., AND FEDKIW, R. 2004. Invertible finite elements for robust simulation of large deformation. In *Proceedings of the 2004 Symposium on Computer Animation*. 131–140.

KARYPIS, G. AND KUMAR, V. 1995. A fast and high quality multilevel scheme for partitioning irregular graphs. In *International Conference on Parallel Processing*. 113–122.

LI, X. S. AND DEMMEL, J. W. 1999. A scalable sparse direct solver using static pivoting. In *Proceedings of the 9th SIAM Conference on Parallel Processing for Scientic Computing*. San Antonio, Texas, 1–10.

LIPTON, R. J., ROSE, D. J., AND TARJAN, R. E. 1979. Generalized nested dissection. *SIAM Journal on Numerical Analysis 16,* 2 (Apr.), 346–358.

MARTIN, S., KAUFMANN, P., BOTSCH, M., GRINSPUN, E., AND GROSS, M. 2010. Unified simulation of elastic rods, shells, and solids. *ACM Transactions on Graphics 29,* 4 (July), 39:1–39:10.

MÜLLER, M., DORSEY, J., MCMILLAN, L., JAGNOW, R., AND CUTLER, B. 2002. Stable real-time deformations. In *Proceedings of the 2002 Symposium on Computer Animation*. 49–54.

MÜLLER, M. AND GROSS, M. 2004. Interactive virtual materials. In *Proceedings of Graphics Interface 2004*. 239–246.

MÜLLER, M., HEIDELBERGER, B., TESCHNER, M., AND GROSS, M. 2005. Meshless deformations based on shape matching. *ACM Transactions on Graphics 24,* 3 (July), 471–478.

NEALEN, A., MÜLLER, M., KEISER, R., BOXERMAN, E., AND CARLSON, M. 2006. Physically based deformable models in computer graphics. In *Computer Graphics Forum 25, 4*. 809–836.

NOUR-OMID, B. AND RANKIN, C. C. 1991. Finite rotation analysis and consistent linearization using projectors. *Computer Methods in Applied Mechanics and Engineering 93*, 353–384.

PARKER, E. G. AND O'BRIEN, J. F. 2009. Real-time deformation and fracture in a game environment. In *Proceedings of the 2009 Symposium on Computer Animation*. 156–166.

PRESS, W. H., TEUKOLSKY, S. A., VETTERLING, W. T., AND FLANNERY, B. P. 2002. *Numerical Recipes in C++: The Art of Scientific Computing*. Cambridge University Press.

SHEWCHUK, J. R. 1994. An introduction to the conjugate gradient method without the agonizing pain. Tech. Rep. CMU-CS-94-125, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania. Aug.

SORKINE, O., COHEN-OR, D., IRONY, D., AND TOLEDO, S. 2005. Geometry-aware bases for shape approximation. *IEEE Transactions on Visualization and Computer Graphics 11,* 2 (Mar.), 1–11.

TERZOPOULOS, D., PLATT, J., BARR, A., AND FLEISCHER, K. 1987. Elastically deformable models. In *Proceedings of SIGGRAPH '87*. 205–214.

TOLEDO, S. 2003. TAUCS: A library of sparse linear solvers. http://www.tau.ac.il/∼stoledo/taucs.

ZHU, Y., SIFAKIS, E., TERAN, J., AND BRANDT, A. 2010. An efficient multigrid method for the simulation of high-resolution elastic solids. *ACM Transactions on Graphics 29,* 2 (Apr.), 16:1–16:18.

Received September 2011; revised February 2012; accepted February 2012