Eurographics/ ACM SIGGRAPH Symposium on Computer Animation (2016)
Ladislav Kavan and Chris Wojtan (Editors)

A macroblock optimization for grid-based nonlinear elasticity

Nathan Mitchell — University of Wisconsin-Madison
Michael Doescher — University of Wisconsin-Madison
Eftychios Sifakis — University of Wisconsin-Madison

Abstract

We introduce a new numerical approach for the solution of grid-based discretizations of nonlinear elastic models. Our method targets the linearized system of equations within each iteration of the Newton method, and combines elements of a direct factorization scheme with an iterative Conjugate Gradient method. The goal of our hybrid scheme is to inherit as many of the advantages of its constituent approaches, while curtailing several of their respective drawbacks. In particular, our algorithm converges in far fewer iterations than Conjugate Gradients, especially for systems with less-than-ideal conditioning. On the other hand, our approach largely avoids the storage footprint and memory-bound nature of direct methods, such as sparse Cholesky factorization, while offering very direct opportunities for both SIMD and thread-based parallelism. Conceptually, our method aggregates a rectangular neighborhood of grid cells (typically a $16\times8\times8$ subgrid) into a composite element that we refer to as a "macroblock". Similar to conventional tetrahedral or hexahedral elements, macroblocks receive nodal inputs (e.g., displacements) and compute nodal outputs (e.g., forces). However, this input/output interface now only includes nodes on the boundary of the $16\times8\times8$ macroblock; interior nodes are always solved exactly, by means of a direct, highly optimized solver. Models built from macroblocks are solved using Conjugate Gradients, which is accelerated due to the reduced number of degrees of freedom and improved robustness against poor conditioning thanks to the direct solver within each macroblock. We explain how we attain these benefits with just a small increase of the per-iteration cost over the simplest traditional solvers.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Computational Geometry and Object Modeling—Physically based modeling

1. Introduction

The Newton method has largely been the golden standard for the simulation of nonlinear elastic bodies, although a number of interesting deviations from this standard approach have garnered attention in the graphics literature (e.g., nonlinear multigrid cycles [ZSTB10], projective and position-based dynamics [MHHR07, BML*14, Wan15] and shape matching [RJ07]). In a typical Newton scheme, once a linear approximation to the governing equations is computed, most practitioners will either employ a direct method or select a technique from a spectrum of iterative methods in order to solve the resulting system.

Direct solvers are perhaps the safest and most straightforward way to solve the system that results from the linearization of the governing equations. These methods can be quite practical for relatively small problems when direct algebra is not very expensive. Additionally, these techniques are quite resilient to the conditioning of the underlying problem. Even for large models, high quality parallel implementations such as the Intel MKL PARDISO library are available. Despite such advantages, direct methods suffer from inherently superlinear computational complexity. Even with the benefit of parallelism, direct methods will typically be more expensive than several iterative schemes, especially if few number of iterations are performed. Additionally, direct methods are inherently memory bound; at the core of direct solvers are forward and backward substitution routines that carry out a very small number of arithmetic operations for each memory access required. This often results in grossly memory-bound execution profiles on modern hardware. This drawback is even more heavily felt for large models that do not fit in cache. Finally, each iteration of the Newton method is inherently inexact, providing only a step towards the converged solution. With direct methods we often find ourselves perfectly solving an inaccurate linearized approximation.

With iterative solvers, we can aim for an approximate solution to the linearized problem with the understanding that with each Newton iteration the problem itself will change. These methods include Krylov methods like Conjugate Gradient, Multigrid, and fixed-point iterations such as Jacobi, Gauss-Seidel and SOR. The primary benefit of iterative techniques is that each individual iteration is relatively cheap; this allows users the option to either iterate as much as they can afford, or alternatively truncate the iterative process when the approximate solution is acceptable. Also, many iterative methods are assembly-free, alleviating the need to construct or store the stiffness matrix. In fact, some of the most efficient techniques go to great lengths to minimize memory footprint [MZS*11] while leveraging SIMD and multithreading.

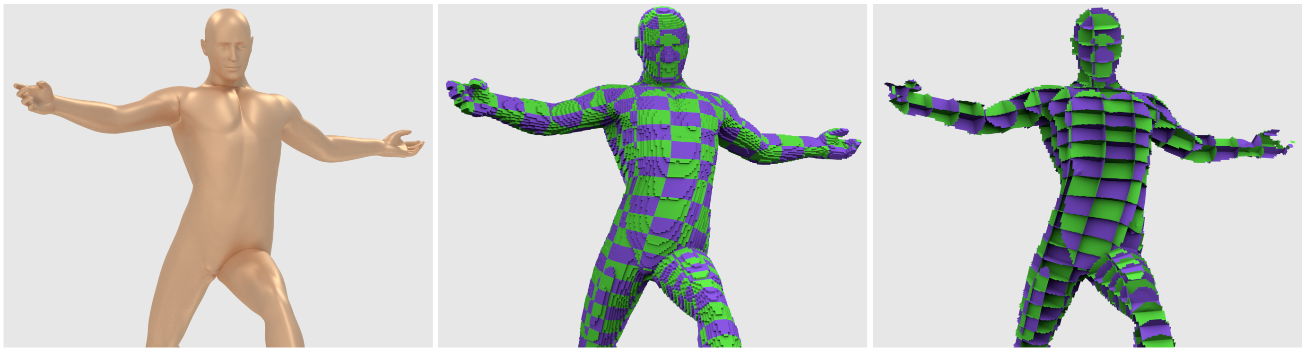

Figure 1: *(Left) High resolution human mesh posed quasistatically by a skeleton with soft spring constraints. (Center) Embedding lattice divided into macroblocks (shown as alternating regions of green and purple). (Right) Illustration of the degrees of freedom along the macroblock boundaries. Conjugate Gradients is applied to a system with the size of these interface nodes. The model has 286K grid cells.*

Iterative solvers often have to cope with challenges of their own. Local methods like Jacobi, GS, and SOR are slow to capture global effects, as they propagate information at a limited speed across the mesh. Krylov methods will typically prioritize the most important modes that contribute to a high residual; for example, consider a system with a few tangled elements that create large local forces. Elements suffering from small errors will be relatively neglected by a method like Conjugate Gradients, while the solver focuses on the highly tangled elements before turning its attention to the bigger picture. Multigrid is an interesting alternative that often emerges as the performance champion; however, it can often be tricky to get to work robustly, and might be less appropriate for thin elastic objects, such as a thin flesh layer on a simulated face. Preconditioning can accelerate the convergence of iterative solvers but, in contrast to certain fluids simulation scenarios, the accelerated convergence might not always justify the increased per-iteration cost. Preconditioners based on incomplete factorizations are memory bound as they require matrix assembly, and generally require an expensive re-factorization step at each Newton iteration. We note that the same factorization overhead would be incurred even when the Newton method is nearly converged, where just a handful of iterations would suffice to solve the linearized equations. Multigrid-based preconditioners might achieve more competitive performance, but such approaches have been primarily tested in the area of fluid simulation [FWD14] and not so much in nonlinear deformable solids.

We propose a hybrid method that balances certain advantages of both direct and iterative schemes. Specifically we endeavor to achieve a good compromise between memory and compute load, reduce the memory footprint whenever possible, while significantly reducing iteration count. We pursue these goals while being competitive with the per-iteration cost of unpreconditioned CG. We employ a grid-based discretization, and aggregate rectangular clusters of cells into "macroblocks" with a proposed size of $16 \times 8 \times 8$ cells. These clusters essentially act as composite elements the same way that a typical hexahedral element can be thought of as a black box that takes displacements as inputs and produces nodal forces as output. However, our composite elements only take in displacements on the nodes of their periphery and return forces on those same boundary nodes. Using this construct we obtain an equivalent linear system with degrees of freedom only on cluster boundaries.

Scope Our paper is an exploration of the performance potential offered by composite "macroblock" elements, initially focusing on the well-established simulation paradigm of a Newton-type scheme for solving a nonlinear system of governing equations. Thus, we only focus on grid-based discretizations of elasticity, and forgo the exploration of different simulation paradigms (e.g., multigrid, projective dynamics) where our formulation might still have a viable role (see brief discussion in section 7). Finally, we consciously restrict our investigation to grid-based models that do not exhibit non-local interactions, such as spring-based constraints or penalty-based self-collision resolution mechanisms (one-sided collisions between the elastic body and kinematic objects are supported).

2. Related Work

The need for efficient, ideally interactive simulation of deformable bodies has been catered to by several procedural techniques [JMD*07, KCvO08, VBG*13], although when fidelity and realism is the objective, physics-based methods are typically employed [TPBF87]. The Finite Element Method has been very popular in this aspect, and various authors have successfully used it to animate a diverse spectrum of behaviors [OH99, TBHF03, ITF04].

Grid-based, embedded elastic models [MTG04, NPF06, MZS*11, PMS12, MCS15] have been very popular due to their inherent potential for performance optimizations, and can also be used with shape-matching approaches [RJ07]. They form the foundation for a class of highly efficient, multigrid-based numerical solution techniques [ZSTB10, GW08, DGW11].

Authors have sought to accelerate simulation performance via a number of avenues, including the use of optimized direct solvers [SSB13] and delayed updates to factorization approaches [HLSO12]. Others have sought to leverage the Boundary Element Method [JP99] to approach real-time deformation and similar formulations that abstract away interior degrees of freedom to accelerate collision processing [GMS14]. Our method has significant ties to these approaches, as well as the general class of Schur complement methods [QV99]. In our present work, we leverage such a formulation to aggregate local neighborhoods of simulation elements into composite elements that interface with the simulation system exclusively via their boundary.

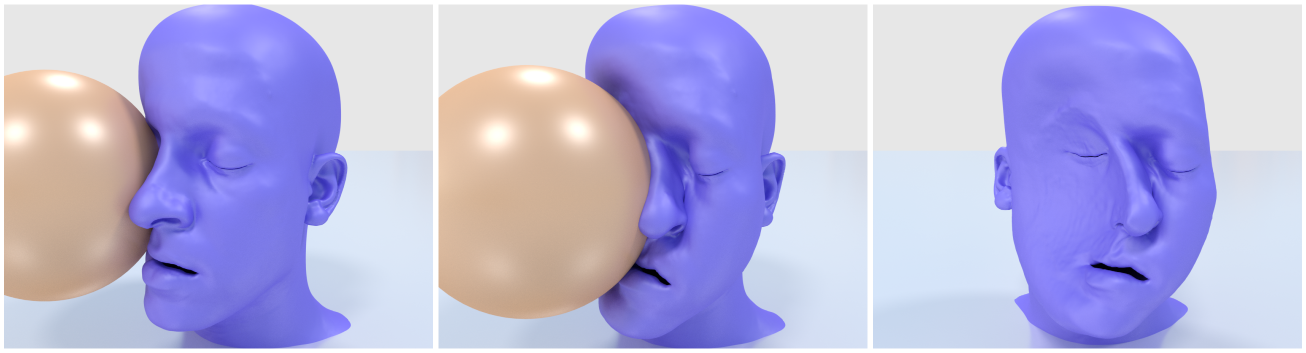

Figure 2: *A kinematic rigid sphere collides against a high-resolution embedded face model. The relatively small thickness of the elastic flesh, in addition to the topological features near the nose and mouth regions, would complicate the use of a typical multigrid solver [MZS*11].*

3. Macroblock-based discretization and numerical solution

We start by reviewing the equations that our method targets, and detailing how our proposed *macroblock* concept can reformulate them into an equivalent but more efficiently solvable form. This process will necessitate the exact solution of several smaller systems of equations, each in the order of a couple thousand of unknowns. In this section we will simply assume that a highly efficient *direct* solver for those systems is available. Section 4 will provide the implementation details of this highly optimized solver.

The governing equations describing the deformation of an elastic nonlinear solid depend on the time integration scheme employed. For example, in quasistatic simulation we have to solve the nonlinear equilibrium equation $\mathbf{f}(\mathbf{x};t) = 0$ at any time instance t. Using an initial guess $\mathbf{x}^{(k)}$ of the solution, Newton's method computes a correction $\delta\mathbf{x} = \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}$ by solving the linearized system:

$$\underbrace{-\left.\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right|_{\mathbf{x}^{(k)}}}_{\mathbf{K}(\mathbf{x}^{(k)})} \cdot\ \delta\mathbf{x} = \mathbf{f}(\mathbf{x}^{(k)}) \tag{1}$$

If an implicit Backward Euler scheme was used, a system with similar structure would form the core of Newton's method [SB12]:

$$\left[(1+\frac{\gamma}{\Delta t})\mathbf{K}(\mathbf{x}^{(k)})+\frac{1}{\Delta t^2}\mathbf{M}\right]\delta\mathbf{x} = \frac{1}{\Delta t}\mathbf{M}(\mathbf{v}^p - \mathbf{v}^{(k)}) + \mathbf{f}(\mathbf{x}^{(k)},\mathbf{v}^{(k)}) \tag{2}$$

where $\mathbf{M}$ is the mass matrix, γ is the Rayleigh coefficient, $\mathbf{v}^p$ the velocities at the previous time step, and $\mathbf{f}$ now includes both elastic and damping forces (see [SB12] for further details).

Despite the semantic differences, the linear systems in equations (1) and (2) are very similar from an algebraic standpoint:

- Their coefficient matrices are both symmetric positive definite.
- Their coefficient matrices have the same sparsity pattern.
- In a grid-based discretization, their coefficient matrices can be assembled from the contributions of individual grid cells.

We note that in order for this last property to hold true, we have assumed that our elastic model does not have any interactions between remote parts of its domain, such as penalty forces used to enforce self-collision (which we consciously excluded from our scope). Incidentally, penalty forces used to enforce collisions with external kinematic bodies are allowed, since their point of application on the elastic body can be embedded in a single grid cell. For brevity, we will write any linear system that shares the three properties above using the simplified notation $\mathbf{Kx} = \mathbf{f}$, without individual emphasis on whether the system originated from a quasistatic, or a dynamic implicit scheme as in equations (1) and (2), respectively.

The crucial next step in our proposed approach is a partitioning of the active grid cells into *macroblocks*, which are grid-aligned rectangular clusters of a predetermined size, as illustrated in figure 1. In our implementation we use macroblocks with dimensions of $16\times8\times8$ grid cells, although the formulations in this section are largely independent of the macroblock size. Section 5 provides the reasoning behind the choice of this particular size of a macroblock.

Each macroblock $\mathcal{B}_i$ consists of up to $16\times8\times8 = 1024$ grid cells $C_{i_1}, C_{i_2}, \ldots, C_{i_M}$; note that in some cases this maximum number of constituent cells will not be reached, if the macroblock overlaps with the boundary of the elastic object, or if "gaps" of empty grid cells are present within its extent. Similarly, up to $17\times9\times9$ nodal degrees of freedom will be present in the region spanned by $\mathcal{B}_i$. Up to $15\times7\times7$ of them will be on the *interior* of $\mathcal{B}_i$ and thus will not be touched by any other macroblock; we will denote this interior node set with I_i. The remaining nodes, located on the *boundary* of $\mathcal{B}_i$ are potentially shared by neighboring macroblocks; we will call these *interface nodes* (as they reside at the interface between macroblocks) and denote their set with Γ_i. All sets I_i are clearly disjoint, and we will denote their union by $\mathrm{I} = \cup I_i$. The interface sets Γ_i do overlap with one another, and we denote their union by $\Gamma = \cup\Gamma_i$. For large enough models, we expect around 72% of grid nodes to lie in some interior set, and approximately 28% on the interface set Γ, using the aforementioned macroblock size.

Our objective will be to replace the linear system $\mathbf{Kx} = \mathbf{f}$ with an equivalent system, which only includes the interface nodes in Γ as unknowns. To do so, we first write the system in block form, by separating interior and interface variables as follows:

$$\begin{pmatrix} \mathbf{K}_{\mathrm{II}} & \mathbf{K}_{\mathrm{I}\Gamma} \\ \mathbf{K}_{\Gamma\mathrm{I}} & \mathbf{K}_{\Gamma\Gamma} \end{pmatrix} \begin{pmatrix} \mathbf{x}_{\mathrm{I}} \\ \mathbf{x}_{\Gamma} \end{pmatrix} = \begin{pmatrix} \mathbf{f}_{\mathrm{I}} \\ \mathbf{f}_{\Gamma} \end{pmatrix}$$

Using block Gauss elimination, this system can be converted to the following equivalent block-triangular form:

$$\begin{pmatrix} \mathbf{K}_{\mathrm{II}} & \mathbf{K}_{\mathrm{I}\Gamma} \\ \mathbf{0} & \mathbf{K}_{\Gamma\Gamma} - \mathbf{K}_{\Gamma\mathrm{I}}\mathbf{K}_{\mathrm{II}}^{-1}\mathbf{K}_{\mathrm{I}\Gamma} \end{pmatrix} \begin{pmatrix} \mathbf{x}_{\mathrm{I}} \\ \mathbf{x}_{\Gamma} \end{pmatrix} = \begin{pmatrix} \mathbf{f}_{\mathrm{I}} \\ \mathbf{f}_{\Gamma} - \mathbf{K}_{\Gamma\mathrm{I}}\mathbf{K}_{\mathrm{II}}^{-1}\mathbf{f}_{\mathrm{I}} \end{pmatrix} \tag{3}$$

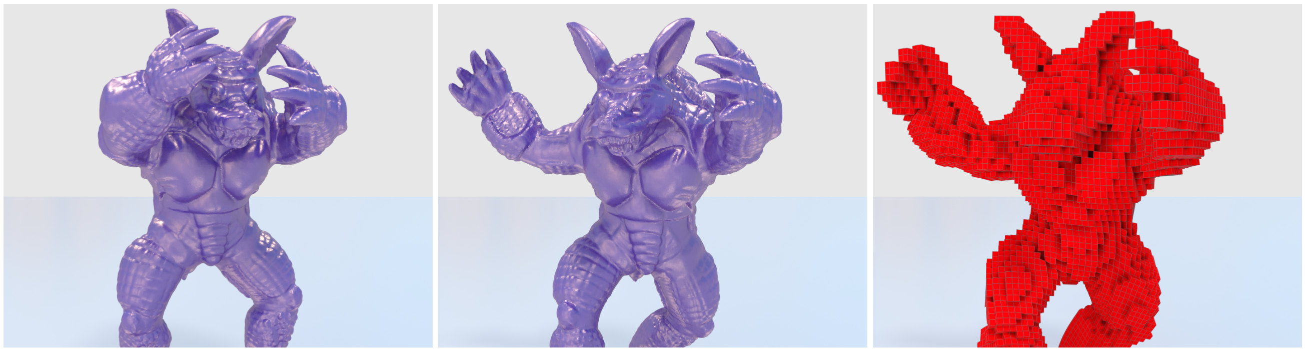

Figure 3: *Armadillo model deforming as a result of kinematically animated Dirichlet constraints. Embedding lattice shown on the right.*

Equation (3) suggests the following algebraically equivalent method for solving the system $\mathbf{K}\mathbf{x} = \mathbf{f}$:

Step 1 Compute an interface-specific right hand side, from the bottom block of the right hand side of system (3):

$$\hat{\mathbf{f}}_\Gamma = \mathbf{f}_\Gamma - \mathbf{K}_{\Gamma\mathrm{I}}\mathbf{K}_{\mathrm{II}}^{-1}\mathbf{f}_\mathrm{I} \tag{4}$$

Step 2 Solve the interface-specific system $\hat{\mathbf{K}}\mathbf{x}_\Gamma = \hat{\mathbf{f}}_\Gamma$ to compute the values $\mathbf{x}_\Gamma$ of all interface nodes. Note that the matrix of the system

$$\hat{\mathbf{K}} = \mathbf{K}_{\Gamma\Gamma} - \mathbf{K}_{\Gamma\mathrm{I}}\mathbf{K}_{\mathrm{II}}^{-1}\mathbf{K}_{\mathrm{I}\Gamma} \tag{5}$$

is the Schur complement of the symmetric positive definite original matrix $\mathbf{K}$, hence it is symmetric and positive definite in its own right. We will solve this system, which only involves interface degrees of freedom, using Conjugate Gradients.

Step 3 Conclude the computation by solving for the interior nodal variables from the top block of system (3) as:

$$\mathbf{x}_\mathrm{I} = \mathbf{K}_{\mathrm{II}}^{-1}\left(\mathbf{f}_\mathrm{I} - \mathbf{K}_{\mathrm{I}\Gamma}\mathbf{x}_\Gamma\right) \tag{6}$$

In order to reproduce the exact solution of $\mathbf{K}\mathbf{x} = \mathbf{f}$, we would need to solve the interface problem $\hat{\mathbf{K}}\mathbf{x}_\Gamma = \hat{\mathbf{f}}_\Gamma$ in Step 2 exactly. However, given that we only use this solution as part of an iterative Newton update, there is nothing preventing us from stopping the Conjugate Gradients solver for the interface system short of full convergence. However, as we discuss in sections 6 and 7, the interface problem requires far fewer CG iterations to produce good quality results than the same Krylov method applied to $\mathbf{K}\mathbf{x} = \mathbf{f}$. Furthermore, the optimizations of the following section allow us to make the per-iteration cost of CG on the interface problem be comparable to each CG iteration on the original problem, resulting in a significant net performance gain. When assessing the cost of Steps 1-3, it is important to observe the following:

Inversion of $\mathbf{K}_{\mathrm{II}}$ is the main performance challenge. The most performance-sensitive component of this process is the multiplication with the inverse $\mathbf{K}_{\mathrm{II}}^{-1}$ of the matrix block corresponding to variables interior to macroblocks. Nevertheless, since there is no direct coupling (in $\mathbf{K}$) between interior variables of neighboring macroblocks, $\mathbf{K}_{\mathrm{II}}$ is a block diagonal matrix, comprised of decoupled diagonal components for each set of interior variables of each macroblock. We thus use multithreading to invert the interior of each macroblock in a parallel and independent fashion. Within each macroblock, we use the aggressively SIMD-optimized *direct* solver detailed in section 4 to perform the inversion exactly and efficiently.

Multiplication with $\mathbf{K}_{\mathrm{I}\Gamma}, \mathbf{K}_{\Gamma\mathrm{I}}$ in Steps 1 & 3 is inexpensive. The off-diagonal blocks $\mathbf{K}_{\mathrm{I}\Gamma}$ and $\mathbf{K}_{\Gamma\mathrm{I}}$ appearing in Steps 1 and 3 are small and sparse sub-blocks of $\mathbf{K}$. In addition, they are only used in two matrix-vector multiplications across Steps 1 and 3 for an entire Newton iteration (we will address their role in Step 2, next). These matrices can be efficiently stored in sparse format, and their multiplication with vectors can be parallelized (in our implementation, via SIMD within macroblocks and multithreading across blocks). These matrices have minimal performance impact in our examples.

Conjugate Gradients does not need to construct $\hat{\mathbf{K}}$. The interface matrix $\hat{\mathbf{K}}$, being a Schur complement, is significantly denser than the original matrix $\mathbf{K}$; for example, any two nodal variables on the interface of the same macroblock would be coupled together. Fortunately, the Conjugate Gradients method does not need this matrix to be explicitly constructed. Instead, the only requirement is to be able to compute matrix-vector products of the form

$$\mathbf{s}_\Gamma = \hat{\mathbf{K}}\mathbf{p}_\Gamma = \left(\mathbf{K}_{\Gamma\Gamma} - \mathbf{K}_{\Gamma\mathrm{I}}\mathbf{K}_{\mathrm{II}}^{-1}\mathbf{K}_{\mathrm{I}\Gamma}\right)\mathbf{p}_\Gamma$$

for any given input vector $\mathbf{p}_\Gamma$. In fact, we can compute such products on a per-macroblock basis. We start by computing the restriction of $\mathbf{p}_\Gamma$ to the boundary Γ_i of each macroblock $\mathcal{B}_i$, which we denote by $\mathbf{p}_{\Gamma_i}$. Subsequently, we compute a partial contribution to the matrix-vector product as

$$\mathbf{s}_{\Gamma_i} = \hat{\mathbf{K}}_i\mathbf{p}_{\Gamma_i} = \left(\mathbf{K}_{\Gamma_i\Gamma_i} - \mathbf{K}_{\Gamma_i\mathrm{I}_i}\mathbf{K}_{\mathrm{I}_i\mathrm{I}_i}^{-1}\mathbf{K}_{\mathrm{I}_i\Gamma_i}\right)\mathbf{p}_{\Gamma_i} \tag{7}$$

The highly efficient evaluation of the expression in equation (7) is precisely the focus of section 4. We compute the contributions of all macroblocks $\mathbf{s}_{\Gamma_i}$ in parallel, via multithreading, and reduce them all together in a final summation to produce the global result $\mathbf{s}_\Gamma$.

Finally, we point out a significant intuition behind the nature of the macroblock-local Schur complement $\hat{\mathbf{K}}_i$, defined via equation (7). Similar to how an elemental stiffness matrix maps nodal displacements to nodal force differentials for a tetrahedral or hexahedral element, the *macroblock stiffness matrix* $\hat{\mathbf{K}}_i$ directly maps displacements on the boundary to forces on the same boundary nodes, under the assumption that all interior nodes are functionally constrained to their exact solution subject to the boundary displacement values. We note the similarity of this concept to the work of Gao et al [GMS14], although they used a Schur complement to abstract away the interior nodes of an entire model, rather than assembling an elastic solid from macroscopic cell blocks.

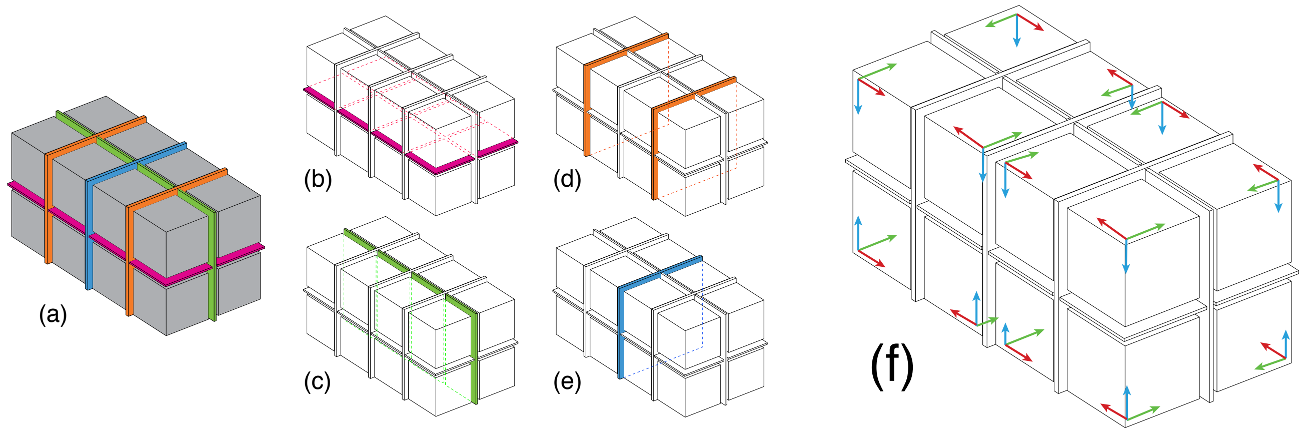

Figure 4: *The $15\times7\times7$ macroblock interior nodes are hierarchically subdivided, yielding (a) sixteen $3\times3\times3$ "subdomains" and (b,c,d,e) four "interface" layers. The first subdomain is reordered to maximize sparsity, and this ordering is mirrored (f) to the other 15 subdomains.*

4. An optimized direct solver for macroblocks

As outlined in section 3, inverting $\mathbf{K}_{\mathrm{I}_i\mathrm{I}_i}$ within each macroblock is the most performance-sensitive part of our numerical approach. In this section we explain how this operation can be performed with high efficiency, by reducing its memory footprint and aggressively leveraging instruction-level (SIMD) parallelism. We have designed a numerical data structure containing the appropriate metadata and computational routines to compute the matrix-vector product $\mathbf{s}_{\Gamma_i}$ of equation (7), given the boundary values $\mathbf{p}_{\Gamma_i}$ as input. This structure stores matrices $\mathbf{K}_{\Gamma_i\Gamma_i}$, $\mathbf{K}_{\Gamma_i\mathrm{I}_i}$ and $\mathbf{K}_{\mathrm{I}_i\Gamma_i}$ explicitly in compressed sparse format (with slight modifications to facilitate SIMD parallelism, as explained in section 4.3), as those are relatively compact and inexpensive to multiply with. In addition, we store just enough information to be able to multiply the interior inverse $\mathbf{K}^{-1}_{\mathrm{I}_i\mathrm{I}_i}$ with input vectors, without storing this matrix explicitly. As this section focuses on a single macroblock $\mathcal{B}_i$, we omit the macroblock index i, using the symbols I and Γ to denote its interior and interface nodes.

Given the sparsity and definiteness of $\mathbf{K}_{\mathrm{II}}$, one straightforward approach would be to compute its (exact) Cholesky factorization, under a sparsity optimizing variable reordering. This factorization would take place once per Newton iteration, while forward and backward substitution passes would be used to apply the inverse in every subsequent CG iteration based on equation (7). We do, in fact, compute exactly such a reordered Cholesky factorization; however, instead of forward/backward substitution, we leverage a hierarchical alternative (derived from the coefficients of the computed factorization) that achieves the same result in significantly less time, by reducing the required memory footprint.

4.1. Reordering

We utilize a custom reordering of the $15\times7\times7$ interior nodes of the macroblock, in order to optimize the sparsity of Cholesky factorization and expose repetitive regular patterns that can be matched with SIMD calculations. We define this reordering by means of a hierarchical subdivision, as illustrated in figure 4. First, we subdivide the $15\times7\times7$ interior region into two $7\times7\times7$ subregions, separated by a $1\times7\times7$ interface layer, illustrated in blue color in figure 4(e). Each of these two regions is further subdivided into two $3\times7\times7$ parts, separated by $1\times7\times7$ interface layers, shown in orange in figure 4(d). Those $3\times7\times7$ regions are then split into two $3\times3\times7$ parts, separated by $3\times1\times7$ interfaces, shown in green in figure 4(c). A last subdivision results in two $3\times3\times3$ subdomains, on either side of a $3\times3\times1$ connector, drawn in magenta in figure 4(b). We refer to the resulting $3\times3\times3$ blocks as *subdomains*, and the connective regions in figures 4(b) through 4(e) as Level-1 through Level-4 *interfaces*. We then proceed to compute a minimum-degree reordering for one of the 16 resulting $3\times3\times3$ subdomains, and *mirror* this reordering across their hierarchical interfaces to enumerate the nodes of all remaining subdomains. This mirroring is essential in creating repetitive patterns in the Cholesky factors, on which SIMD optimizations are crucially dependent. The final overall reordering is formed by assembling a tree of this hierarchical subdivision (with interfaces on parent nodes, and the regions they separate as their children), and computing a reverse breadth-first tree traversal.

We have found this reordering to be optimal; it matches or outperforms any heuristics (e.g., minimum-degree reordering in Matlab) in the sparsity of the Cholesky factors. The resulting sparsity pattern is illustrated in figure 5. Matrix entries colored red are a subset (but not all) of the entries that were filled-in during the Cholesky process. As expected, forward and backward substitution on this matrix is a pronouncedly memory-bound operation; hence we propose a further algorithmic modification that produces the same result with approximately one-seventh of the memory footprint. This alternative approach will only need to store the number of coefficients corresponding to the *black-colored* entries in figure 5. The metadata for this alternative approach, detailed next, will be harvested from the Cholesky factorization just computed.

4.2. Hierarchical factorization

Consider the first hierarchical subdivision, illustrated in 4(e), which separated the $15\times7\times7$ block of interior nodes into two $7\times7\times7$ subregions, which we denote by I_1 and I_2, along with a $7\times7\times1$ connective region, denoted I_c (drawn blue in the figure above). If we reorder the matrix $\mathbf{K}_{\mathrm{II}}$ to expose this partitioning, it assumes the following block form:

$$\begin{pmatrix} \mathbf{K}_{11} & & \mathbf{K}_{1c} \\ & \mathbf{K}_{22} & \mathbf{K}_{2c} \\ \mathbf{K}_{c1} & \mathbf{K}_{c2} & \mathbf{K}_{cc} \end{pmatrix}$$

It can be easily verified that the *inverse* of this matrix can be written in the following Block-LDL form:

$$\begin{pmatrix} \mathbf{I} & & -\mathbf{K}_{11}^{-1}\mathbf{K}_{1c} \\ & \mathbf{I} & -\mathbf{K}_{22}^{-1}\mathbf{K}_{2c} \\ & & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{K}_{11}^{-1} & & \\ & \mathbf{K}_{22}^{-1} & \\ & & \mathbf{C}^{-1} \end{pmatrix} \begin{pmatrix} \mathbf{I} & & \\ & \mathbf{I} & \\ -\mathbf{K}_{c1}\mathbf{K}_{11}^{-1} & -\mathbf{K}_{c2}\mathbf{K}_{22}^{-1} & \mathbf{I} \end{pmatrix}$$

where $\mathbf{C} = \mathbf{K}_{cc} - \mathbf{K}_{c1}\mathbf{K}_{11}^{-1}\mathbf{K}_{1c} - \mathbf{K}_{c2}\mathbf{K}_{22}^{-1}\mathbf{K}_{2c}$ is the Schur complement of $\mathbf{K}_{cc}$. With this formulation, solving a problem $\mathbf{K}_{\text{II}}\mathbf{x}_{\text{I}} = \mathbf{f}_{\text{I}}$ is equivalent to multiplying with the factorized version of $\mathbf{K}_{\text{II}}^{-1}$ in the equation above. We make the following significant observations:

- Other than the (seemingly elusive) inverses $\mathbf{K}_{11}^{-1}, \mathbf{K}_{22}^{-1}$ and $\mathbf{C}^{-1}$, the factorization above does not incur any fill-in; factors such as $\mathbf{K}_{1c}$, etc. have the original sparsity found in sub-blocks of $\mathbf{K}_{\text{II}}$.
- We can prove that the lower-triangular Cholesky factor of the Schur complement $\mathbf{C}$ is *exactly* the bottom-rightmost (dense) diagonal block of the matrix shown in figure 5 (also more prominently colored blue in figure 6). Thus, multiplication with $\mathbf{C}^{-1}$ can be performed simply via forward and backward substitution.
- The inverses of the two subregions, $\mathbf{K}_{11}^{-1}$ and $\mathbf{K}_{22}^{-1}$ can be applied recursively using the exact same decomposition and block-LDL factorization described here, by splitting each $7\times7\times7$ into two $7\times7\times3$ subregions and a $7\times7\times1$ connector as before. This recurrence can be unfolded until we arrive at the (sixteen) $3\times3\times3$ subdomains shown in figure 4. The Cholesky factors of those sixteen blocks are exactly the top-sixteen (sparse) diagonal blocks on the top-left of the Cholesky factorization in figure 5; thus those submatrices can be readily inverted without recursion.

We note that the Cholesky factors of the Schur complement matrices (**C**) that appear in deeper levels of this hierarchical solution scheme are similarly harvested from the (dense) diagonal blocks of the overall Cholesky factorization (highlighted in purple, green and orange color in figure 6, immediately above the blue block at the bottom-rightmost part which corresponds to the first hierarchical subdivision). At the final level of this hierarchical solution process, we need the inverses of the matrix blocks corresponding to the sixteen $3\times3\times3$ subdomains themselves. For those blocks, we employ directly their sparse Cholesky factorization, as seen in the top-sixteen (dark blue colored) diagonal blocks in figure 6, and solve using standard forward and backward substitution.

It would appear that the additional computation that this recursive solution entails would render it prohibitively expensive. However, the stock Cholesky forward and backward substitution are memory-bound by such a wide margin that our optimized recursive solution can afford to execute a significantly larger amount of arithmetic operations, while still being (barely, this time) bound by the time required to stream the requisite matrix coefficients from memory into cache. The not so obvious, but very significant, benefit is that the entire working set of this solver is less than 800KB per macroblock, allowing all subsequent memory accesses to occur exclusively in cache for every CPU core handling an individual macroblock. Note that, although the original reordered Cholesky factorization produces additional fill-in on the matrix entries colored red in figure 5, our recursive substitution process only touches a significantly sparser subset of entries (colored black), requiring about 27% of the entries and 15% of the storage footprint of the full, filled-in Cholesky (accounting for row/column indices of structurally sparse blocks). In section 6 we provide the effective memory bandwidth achieved by our macroblock solver, averaging between 13-18GB/s on a 10-core Haswell-EP Xeon processor.

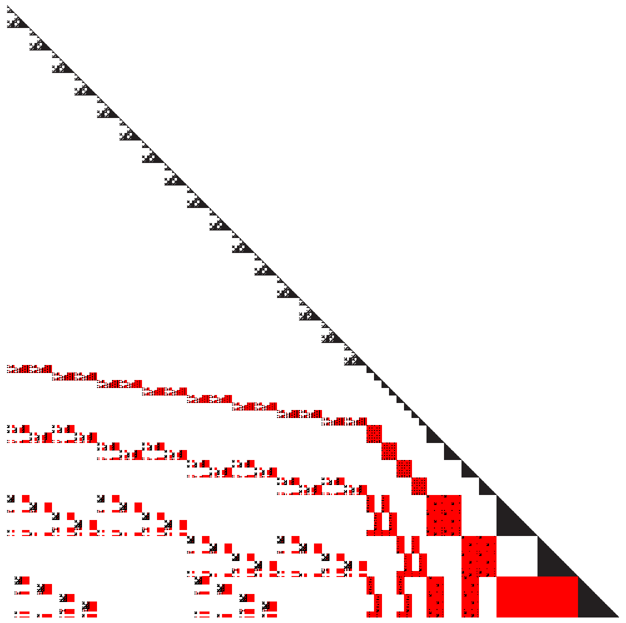
Figure 5: *Sparsity of Cholesky factorization (with our optimal reordering), shown with red **and** black colors. The memory footprint of our proposed solver only includes the black-colored coefficients.*

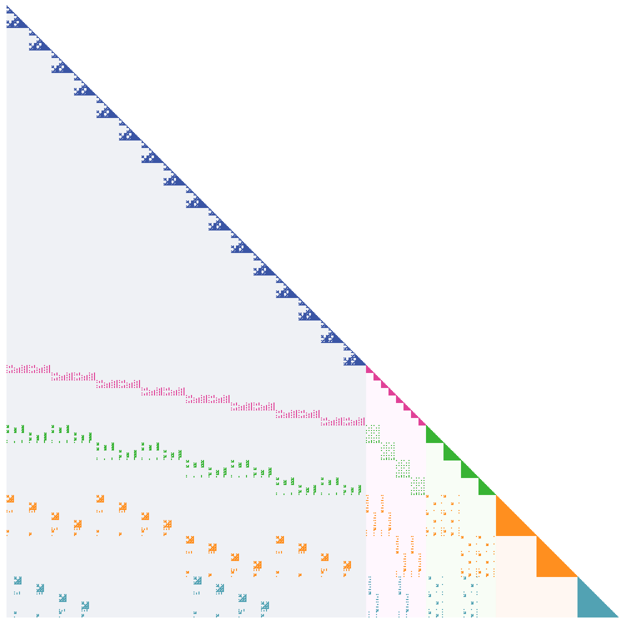
Figure 6: *Our method reveals regular structures in the matrix sparsity pattern, exploiting them for vectorization. Same-color entries in the off-diagonal blocks can be processed with SIMD instructions.*

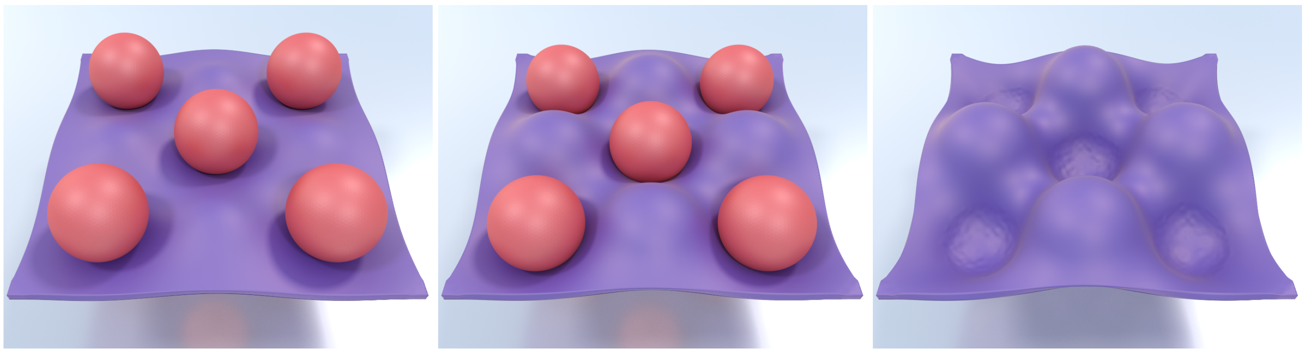

Figure 7: *An array of 9 kinematic spheres, arranged in an alternating pattern across a thin volumetric sheet, are pressed against it. The limited thickness of this model would hinder applicability of stock geometric multigrid, in the absence of nonstandard coarsening strategies.*

4.3. Vectorization

The sparse matrix data used in our method, as seen in figure 6, is characterized by extensive regular and repetitive sparsity patterns that can facilitate computation using SIMD instructions. We have used color coding to indicate data used within a level of our hierarchical solution scheme, and to highlight such patterns of regularity. Those include the sixteen sparse Cholesky factors corresponding to the interiors of the $3\times3\times3$ subdomains (colored as dark blue blocks, along the top-leftmost part of the matrix diagonal), the dense Cholesky factors of Schur complements at deeper levels (eight magenta, four green, two orange, and one cyan dense block, spanning the rest of the block-diagonal region of the matrix), and sparse submatrices on the block lower-triangular part of the matrix, corresponding to entries of the original stiffness matrix that touch an interface layer at a given level of the hierarchy and nodes on the two subregions that the interface layer separates.

Opportunities for aggressive vectorization directly emerge from such data regularities. For example, sparse forward and backward substitution on all sixteen $3\times3\times3$ subdomains can be done in tandem, with 16-way SIMD parallelism (e.g., using two 8-wide AVX instructions). Repetitive sparsity patterns in the lower-triangular part of the matrix of figure 6 are used in vectorized matrix-vector multiplication operations. The *dense* nature of the blocks along the lower part of the block-diagonal allows fine-grain vectorization via standard practices. Furthermore, even matrix operations that connect the $15\times7\times7$ interior node set with the *boundary* of the macroblock, as the multiplication with matrices $\mathbf{K}_{\Gamma_i\Gamma_i}$, $\mathbf{K}_{\Gamma_i\mathrm{I}_i}$ and $\mathbf{K}_{\mathrm{I}_i\Gamma_i}$ defined in the beginning of this section, can be vectorized by splitting up such matrices in parts that correspond to the sixteen $3\times3\times3$ macroblocks at the interior of the macroblock boundary. Ultimately, about 96% of the requisite computations can accommodate 16-wide SIMD parallelism, and the majority of the remaining operations offer at least 8-wide SIMD parallelism potential. We have extensively leveraged these vectorization opportunities in our optimized implementation based on AVX compiler intrinsics.

5. Justification of macroblock size choice

Our choice for utilizing macroblocks of dimension $16\times8\times8$ was motivated by a number of factors. First, we wanted to provide the opportunity for at least 16-way SIMD-based parallelism, which is a future-safe choice given the upcoming availability of CPUs with the AVX-512 instruction set. The working set size associated with macroblocks of that size is conveniently approximately 800KB, which allows the entire macroblock solver to fit entirely in cache, even if all cores of a typical modern Xeon processor are processing independent macroblocks, in parallel. Using an even larger macroblock size would allow the dimensionality of the interface to be further reduced, but the increment in the working set would be disproportionately large, due to the size of the next-level interface (would be $15\times1\times7$) which would, at that point, yield an unattractively large dense Schur complement matrix for that interface level.

6. Examples and performance evaluation

We visually demonstrate the applicability of our solver to a number of simulation scenarios including constraint-driven deformations, skinning animations and elastic models colliding with kinematic rigid objects. We used a hexahedral finite element discretization of corotated linear elasticity, with the standard adjustments for robust simulation in the presence of inverted elements [ITF04]. Given that our method uses a direct solver at the macroblock level, we opted to integrate the strain energy using the eight Gauss quadrature points for each hexahedron, as opposed to the one-point quadrature scheme that is often used [MZS*11, PMS12]. This more accurate quadrature scheme does not require explicit stabilization, and adds no extra algorithmic effort in our solver other than a modest increase in the matrix construction cost.

In figure 3, we demonstrate an armadillo model being deformed as a result of specific lattice nodes animated as kinematic Dirichlet boundary conditions. In order to incorporate Dirichlet boundary conditions in the interior of a macroblock, we replace the equation associated with any such node with an explicit Dirichlet condition $\delta\mathbf{x}_i=\mathbf{0}$ (the value can be set to zero without loss of generality, since equation (1) is solved for position corrections, which are zero for constraint nodes that have been already moved to their target locations). We restore symmetry of the overall matrix by zeroing out entries involving the Dirichlet node in the stencil of the elasticity operator of any neighboring node (again, a safe operation as the Dirichlet value is zero for the correction $\delta\mathbf{x}_i$). Similarly, any nodes in a macroblock that are exterior to the simulated model are treated as zero-Dirichlet conditions, to maintain a constant matrix structure for all macroblocks.

In figures 2 and 7, we demonstrate the compatibility of our method with penalty-based collisions with kinematic objects. We use an implicit representation for the colliding bodies to enable fast detection of collision events between such bodies and embedded collision proxies on the surface of our model. When such an event occurs, a zero rest length penalty spring constraint is instantiated connecting the offending point on the embedded surface to the nearest point on the surface of the collision object. Finally, figures 1 and 8 show two examples of a human character animated using embedded kinematic bone constraints. Skeletal motion data was drawn from the CMU motion capture database (http://mocap.cs.cmu.edu).

6.1. Performance benchmarks - Comparison to CG

Table 1 provides runtime details for individual solver components. The first two columns correspond to the models of figures 1 and 3, and have been processed with our proposed macroblock solver. In addition, we repeat the skinning simulation of figure 1 using this time a highly optimized and parallelized matrix-free implementation of unpreconditioned Conjugate Gradients, borrowed from the work of Mitchell et al [MCS15]. While using this matrix-free CG solver, we consider two discretization alternatives: (a) a one-point quadrature scheme, with explicit stabilization [MZS*11, PMS12], listed in the third column and (b) a more accurate 8-point quadrature scheme matching the one in our macroblock solver (fourth column). As mentioned, the quadrature scheme does not affect the solve times of our method, once the matrix has been constructed; the construction cost is included in the Newton iteration runtimes, and was less than 10% of the overall runtime in all our experiments. We observe that, in spite of the up-front factorization cost that our method incurs, it typically stays within a factor of 2-3x of the cost of the single quadrature point CG scheme, for the same number of iterations. However, in our supplemental video we demonstrate that the effect of as few as ten iterations of our macroblock scheme is commensurate with 5-10x more iterations of the stock CG method.

Table 1: *Runtime details on a 10-core Xeon E5-2687W CPU. The benchmark in the first column is repeated in the last two columns using stock CG, with one and eight quadrature points respectively. Interface-Multiply is the multiplication with the Schur complement.*

	Human	Armadillo	Human	Human
Solver	Macro-block	Macro-block	CG (1-QP)	CG (8-QP)
Active Cells	286K	24K	286K	286K
Macroblocks	642	95	*N/A*	*N/A*
Interface - Multiply	27.6 ms (17 GB/s)	4.36 ms (16 GB/s)	*N/A*	*N/A*
CG Iteration	33.3 ms	5.22 ms	18.8 ms	88.3 ms
Factorization	291 ms	88.0 ms	*N/A*	*N/A*
Newton Iteration				
10 CG	791 ms	166 ms	269 ms	958 ms
20 CG	1.29 s	244 ms	462 ms	1.84 s
50 CG	2.79 s	479 ms	1.07s	4.47 s

Note that if the more accurate quadrature scheme is employed, our method outperforms the CG option even on a per-iteration basis.

6.2. Additional solver comparisons

We report some additional comparisons with other established numerical algorithms or software packages. All our comparisons are relative to the skinning example in the first column of Table 1.

Macroblock inversion via Cholesky/PARDISO As an alternative to our optimized macroblock solver of section 4, one could choose to directly compute *and* apply a stock Cholesky factorization per macroblock. We tested this using the PARDISO library, which yielded a factorization cost of 748ms (ours: 291ms) and a solve time of 93ms via forward/backward substitution (ours: 20.9ms; part of the Interface-Multiply cost). Solve time savings are due to our reduced memory demands. Faster factorization time is attributed to intrinsic knowledge about the constant sparsity pattern of each block, allowing us to optimally vectorize over multiple blocks without duplicating the data that captures their sparsity patterns.

Different solvers for Newton Step Three options were investigated *(a) Full Cholesky* – We experimented with using a direct (complete) Cholesky solve at each Newton step, via PARDISO. The resulting Newton iteration cost was 31.8s, more than three times the cost our method would require for 250CG iterations (9.36s) and near-perfect convergence. However, our method hardly needs that many CG iterations to achieve excellent Newton convergence, and in the long run easily outperformed full Cholesky by more than an order of magnitude. *(b) Incomplete Cholesky PCG* – ICPCG performed very well in our examples, often requiring half (or less) of our CG iterations for comparable convergence. It is, however, in principle a serial algorithm. Our adequately optimized (albeit serial) implementation required 7.23s to factorize the preconditioner (ours: 291ms) and 422ms (ours: 33.3ms) for each CG iteration. *(c) Block Jacobi PCG* – A parallelism-friendly alternative to ICPCG was to compute a Block Jacobi Preconditioner, with block sizes comparable to our own macroblocks. Matrix entries that straddle blocks were discarded, and a standard Cholesky factorization of the resulting block-diagonal matrix computed via PARDISO. Convergence of this option was generally comparable, and at times slightly better than our solver. This parallel method required 1.24s for factorization (ours: 291ms) and yielded a CG iteration cost of 183ms (ours: 33.3ms). Visual comparisons of all three options to our technique are provided in the supplemental video.

7. Limitations and future work

The most important limitations of our present formulation are (a) the restriction of our scheme to Cartesian lattice-based discretizations of elasticity, and (b) the explicit lack of support for self collisions or other elastic interactions that would couple together disjoint parts of the mesh. We consciously limited our preliminary exploration to applications of macroblocks within a Newton-Raphson iterative solution scheme. In principle, there would have been an opportunity to also consider using macroblocks in the design of a highly efficient box smoother for multigrid, or as a replacement of the local optimization step in projective dynamics; we defer exploration of those interesting threads to future work.

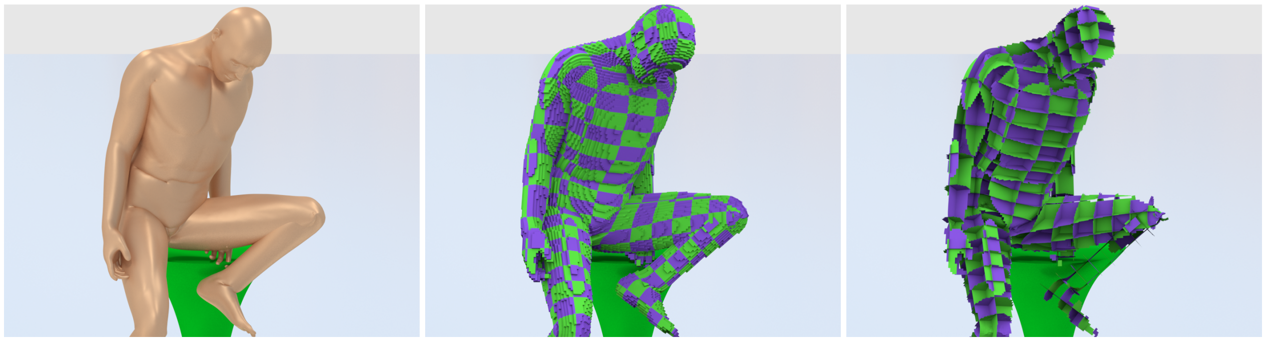

Figure 8: *An additional demonstration of a skinning simulation, driven by kinematic bones attached to the flesh via spring constraints.*

Acknowledgements This work was supported by NSF grants IIS-1253598, IIS-1407282, CCF-1423064, CCF-1533885.

References

[BML*14] BOUAZIZ S., MARTIN S., LIU T., KAVAN L., PAULY M.: Projective dynamics: Fusing constraint projections for fast simulation. *ACM Trans. Graph. 33*, 4 (July 2014), 154:1–154:11. doi:10.1145/2601097.2601116. 1

[DGW11] DICK C., GEORGII J., WESTERMANN R.: A hexahedral multigrid approach for simulating cuts in deformable objects. *IEEE Transactions on Visualization and Computer Graphics 17*, 11 (2011), 1663–1675. doi:10.1109/TVCG.2010.268. 2

[FWD14] FERSTL F., WESTERMANN R., DICK C.: Large-scale liquid simulation on adaptive hexahedral grids. *IEEE Trans. Visualization & Computer Graphics 20*, 10 (Oct 2014), 1405–1417. doi:10.1109/TVCG.2014.2307873. 2

[GMS14] GAO M., MITCHELL N., SIFAKIS E.: Steklov-Poincarè Skinning. In *Eurographics/ ACM SIGGRAPH Symposium on Computer Animation* (2014), Koltun V., Sifakis E., (Eds.), The Eurographics Association. doi:10.2312/sca.20141132. 2, 4

[GW08] GEORGII J., WESTERMANN R.: Corotated Finite Elements Made Fast and Stable. Faure F., Teschner M., (Eds.), VRIPHYS '08, The Eurographics Association. doi:10.2312/PE/vriphys/vriphys08/011-019. 2

[HLSO12] HECHT F., LEE Y. J., SHEWCHUK J. R., O'BRIEN J. F.: Updated sparse Cholesky factors for corotational elastodynamics. *ACM Trans.on Graph. 31*, 5 (2012), 123. doi:10.1145/2231816.2231821. 2

[ITF04] IRVING G., TERAN J., FEDKIW R.: Invertible finite elements for robust simulation of large deformation. SCA '04, Eurographics Association, pp. 131–140. doi:10.1145/1028523.1028541. 2, 7

[JMD*07] JOSHI P., MEYER M., DEROSE T., GREEN B., SANOCKI T.: Harmonic coordinates for character articulation. *ACM Trans. Graph. 26*, 3 (July 2007). doi:10.1145/1276377.1276466. 2

[JP99] JAMES D., PAI D.: ArtDefo: accurate real time deformable objects. In *Proceedings of SIGGRAPH 99* (1999), pp. 65–72. doi:10.1145/311535.311542. 2

[KCvO08] KAVAN L., COLLINS S., ŽÁRA J., O'SULLIVAN C.: Geometric skinning with approximate dual quaternion blending. *ACM Trans. Graph. 27*, 4 (Nov. 2008), 105:1–105:23. doi:10.1145/1409625.1409627. 2

[MCS15] MITCHELL N., CUTTING C., SIFAKIS E.: GRIDiron: An interactive authoring and cognitive training foundation for reconstructive plastic surgery procedures. *ACM Trans. Graph.* (2015). doi:10.1145/2766918. 2, 8

[MHHR07] MÜLLER M., HEIDELBERGER B., HENNIX M., RATCLIFF J.: Position based dynamics. *Journal of Visual Communication and Image Representation 18*, 2 (2007), 109–118. doi:10.1016/j.jvcir.2007.01.005. 1

[MTG04] MÜLLER M., TESCHNER M., GROSS M.: Physically-based simulation of objects represented by surface meshes. CGI '04, pp. 156–165. doi:10.1109/CGI.2004.1309189. 2

[MZS*11] MCADAMS A., ZHU Y., SELLE A., EMPEY M., TAMSTORF R., TERAN J., SIFAKIS E.: Efficient elasticity for character skinning with contact and collisions. *ACM Trans. Graph. 30*, 4 (July 2011), 37:1–37:12. doi:10.1145/2010324.1964932. 1, 2, 3, 7, 8

[NPF06] NESME M., PAYAN Y., FAURE F.: Animating shapes at arbitrary resolution with non-uniform stiffness. VRIPHYS '06, Eurographics. doi:10.2312/PE/vriphys/vriphys06/017-024. 2

[OH99] O'BRIEN J., HODGINS J.: Graphical modeling and animation of brittle fracture. In *Proc. of SIGGRAPH 1999* (1999), pp. 137–146. doi:10.1145/311535.311550. 2

[PMS12] PATTERSON T., MITCHELL N., SIFAKIS E.: Simulation of complex nonlinear elastic bodies using lattice deformers. *ACM Trans. Graph. 31*, 6 (Nov. 2012), 197:1–197:10. doi:10.1145/2366145.2366216. 2, 7, 8

[QV99] QUARTERONI A., VALLI A.: *Domain decomposition methods for partial differential equations*, vol. 10. Clarendon Press, 1999. 2

[RJ07] RIVERS A., JAMES D.: FastLSM: Fast lattice shape matching for robust real-time deformation. *ACM Trans. on Graphics (SIGGRAPH Proc.) 26*, 3 (2007). doi:10.1145/1275808.1276480. 1, 2

[SB12] SIFAKIS E., BARBIC J.: FEM simulation of 3D deformable solids: A practitioner's guide to theory, discretization and model reduction. In *ACM SIG. 2012 Courses* (2012), SIGGRAPH '12, ACM, pp. 20:1–20:50. doi:10.1145/2343483.2343501. 3

[SSB13] SIN F., SCHROEDER D., BARBIC J.: Vega: Non-linear fem deformable object simulator. *Comput. Graph. Forum 32*, 1 (2013), 36–48. doi:10.1111/j.1467-8659.2012.03230.x. 2

[TBHF03] TERAN J., BLEMKER S., HING V. N. T., FEDKIW R.: Finite volume methods for the simulation of skeletal muscle. SCA '03, pp. 68–74. URL: http://dl.acm.org/citation.cfm?id=846276.846285. 2

[TPBF87] TERZOPOULOS D., PLATT J., BARR A., FLEISCHER K.: Elastically deformable models. *SIGGRAPH Comput. Graph. 21*, 4 (Aug. 1987), 205–214. doi:10.1145/37402.37427. 2

[VBG*13] VAILLANT R., BARTHE L., GUENNEBAUD G., CANI M.-P., ROHMER D., WYVILL B., GOURMEL O., PAULIN M.: Implicit skinning: Real-time skin deformation with contact modeling. *ACM Trans. Graph. 32*, 4 (July 2013), 125:1–125:12. doi:10.1145/2461912.2461960. 2

[Wan15] WANG H.: A Chebyshev semi-iterative approach for accelerating projective and position-based dynamics. *ACM Trans. Graph. 34*, 6 (Oct. 2015), 246:1–246:9. doi:10.1145/2816795.2818063. 1

[ZSTB10] ZHU Y., SIFAKIS E., TERAN J., BRANDT A.: An efficient multigrid method for the simulation of high-resolution elastic solids. *ACM Trans. Graph. 29*, 2 (Apr. 2010), 16:1–16:18. doi:10.1145/1731047.1731054. 1, 2