Computer Graphics Forum, 32(1), 2013, p. 38-50
The definite version is available
at www.interscience.wiley.com.

Volume 0 (1981), Number 0 pp. 1–13 COMPUTER GRAPHICS *forum*

Vega: Nonlinear FEM Deformable Object Simulator

F. S. Sin[1], D. Schroeder[1,2], J. Barbič[1]

[1]University of Southern California, USA
[2]Carleton College, USA
sinfunshing@gmail.com, dan5schroeder@gmail.com, jnb@usc.edu

Abstract

This practice and experience paper describes a robust C++ implementation of several nonlinear solid 3D deformable object strategies commonly employed in computer graphics, named the Vega FEM simulation library. Deformable models supported include co-rotational linear FEM elasticity, Saint-Venant Kirchhoff FEM model, mass-spring system, and invertible FEM models: neo-Hookean, Saint-Venant Kirchhoff, and Mooney-Rivlin. We provide several timestepping schemes, including implicit Newmark and backward Euler integrators, and explicit central differences. The implementation of material models is separated from integration, which makes it possible to employ our code not only for simulation, but also for deformable object control and shape modeling. We extensively compare the different material models and timestepping schemes. We provide practical experience and insight gained while using our code in several computer animation and simulation research projects.

Categories and Subject Descriptors (according to ACM CCS): I.6.8 [Simulation and Modeling]: Types of Simulation—Animation I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Physically based modeling

Keywords: deformable objects, finite element method, nonlinear, practice and experience, open source

1. Introduction

Researchers in physically-based modeling have presented many 3D nonlinear deformable models and numerical timestepping schemes. A Google search, however, reveals that few of these models have publicly available implementations. Even when implementations are available, different authors tend to use different code conventions and organization, making it difficult to evaluate and reuse the many FEM deformable strategies employed in computer graphics. In this practice and experience paper, we describe a C++ implementation of the most common FEM deformable models and timestepping schemes employed in computer graphics. We extensively compare the different deformable models and timestepping schemes to each other, and provide observations and insight for practical use. The code can simulate the dynamics of 3D solid deformable objects undergoing large deformations (see Figure 1), and optionally supports static deformations where the mass inertia is neglected. It is suitable both for offline applications in film, and real-time applications in games and virtual medicine. Our models are solid 3D volumetric meshes; we do not support shells or strands. Vega supports linear FEM elasticity, corotational linear FEM elasticity, the Saint-Venant Kirchhoff FEM model (isotropic geometrically-nonlinear elasticity), invertible FEM models (neo-Hookean, Saint-Venant Kirchhoff, Mooney-Rivlin), as well as 3D mass-spring systems. We provide several timestepping schemes, including implicit Newmark and backward Euler integrators, and explicit central differences. We support tetrahedral meshes (with linear elements) and cube hexahedral meshes (trilinear elements), both of which are very common in computer graphics.

Our simulator is a result of several years of research in computer animation and simulation, and we hope that it will find usage in future research projects in these fields. The code is optimized for speed, well-commented and easy to modify. It supports CPU multicore computation of internal forces and stiffness matrices. Also supported are multicore sparse linear system solves for implicit integration. Vega is released under the BSD open-source license, permitting free reuse by academia and industry. The code units depend min-

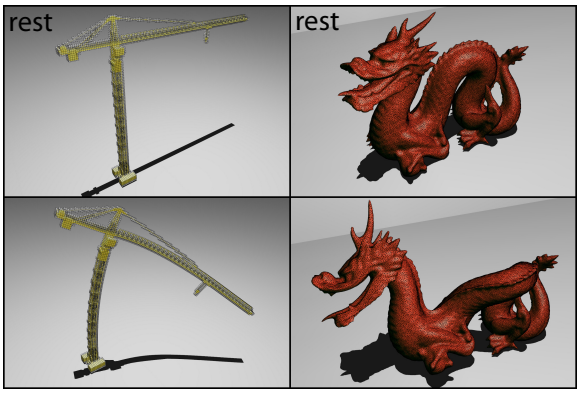

Figure 1: ***Large FEM deformations:*** *The crane voxel mesh is shown blended on top of the embedded triangle mesh, whereas the dragon uses the external surface of the tetrahedral mesh for rendering.*

imally on each other and are independently reusable. For example, we separate internal elastic force computation from integration, so that, e.g., our mass-spring system library can be used with any of our integrators, or the user can provide her own integrator. Similarly, any of our integrator libraries can timestep any of our deformable models, or any physical system provided by the user. We provide documentation, compilation instructions, example meshes, and a complete OpenGL application example. Vega can be downloaded at `http://www.jernejbarbic.com/vega`.

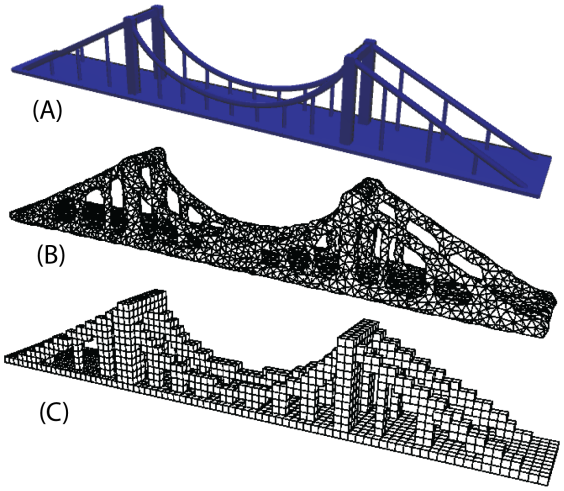

Figure 2: ***Our simulator supports tetrahedral (B) and voxel meshes (C).*** *The embedded triangle rendering mesh is shown in (A).*

2. Related Work

Deformable object simulation is a well-studied problem in computer graphics. We review the FEM approaches relevant to our paper; please see [NMK*06] for a general survey. In engineering, many papers discuss linear methods, which are limited to small deformations. One common approach to model large deformations is to employ quadratic Green-Lagrange strain (geometrically nonlinear models). Its isotropic version, the St.Venant-Kirchhoff (StVK) material model, has been employed in several papers [OH99, DDCB01, PDA01, CGC*02]. Specialized edge data structures have been presented to decrease the computation time of evaluating StVK internal forces [KTY09]. Co-rotational linear FEM was introduced to computer graphics in [MG04]. It can handle large deformations by extracting local material rotations using polar decomposition, and is a popular approach [PO09, CPSS10, MZS*11]. In simulations of soft tissue, common in computer graphics, mesh elements may invert during the simulation. Irving [ITF04] introduced material models that can restore from inversion, and Teran [TSIF05] showed how to compute the tangent stiffness matrix for such material models. Many papers then improved upon various aspects of these methods, e.g. employing multigrid [MZS*11], efficient material coarsening [NKJF09, KMOD09], or performing the simulation in Eulerian coordinates [LLJ*11].

Co-rotational, StVK and invertible FEM approaches are commonly used in computer animation practice. The OpenTissue [Ope] (ZLib license) and SOFA [SOF] libraries (LGPL) both offer linear and co-rotational FEM. OpenTissue also provides invertible co-rotational FEM internal forces (without stiffness matrices), finite differences and shape matching, whereas SOFA includes GPU computation, composite elements and multi-scale simulation. Whereas these libraries offer comprehensive solutions for deformations, plasticity, collision detection and rendering, our code is less complex and focuses on FEM deformable simulation and CPU multithreading. Most models and integrators are implemented in a single {.h,.cpp} file pair, simplifying code reuse. To the best of our knowledge, Vega (BSD) is the only free library offering both internal forces and tangent stiffness matrices for linear, co-rotational, StVK, and invertible FEM models. The Cubica [Kim] library (GPL) provides StVK and invertible FEM, as well as meshing and model reduction. Popular physics engines such as Bullet Physics [Cou], Havok [Hav] and Nvidia Phsyx [Phyb] do not support FEM, and typically model deformations using geometric shape matching [MHTG05]. The Physbam library [Phya] (BSD), while versatile and supporting many physical systems, currently does not support FEM solid deformable objects in its public release. Hyper-sim [Erl] is a Matlab library offering StVK, linear, and co-rotational FEM. Commercial engineering (Abaqus [ABA]) and open-source (CalculiX [DW]) deformable object simulation tools can simulate large-deformation FEM models, but they do

not support co-rotational or invertible models. Also, such engineering software was designed for scripted, offline use, where the functionality to compute internal forces and stiffness matrices is built-in and not exposed to the end user.

3. Overview

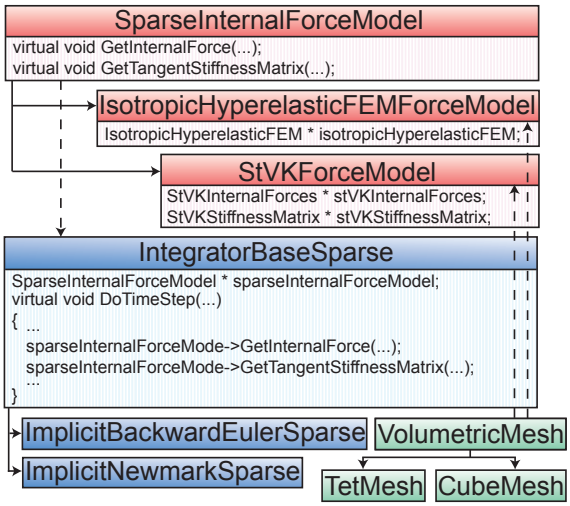

Figure 3: ***Implementation overview:*** *Solid lines denote class derivation, dashed lines denote that a class serves as input to another class.*

For a detailed introduction to FEM deformable objects, we refer the reader to [Sha90, Bar07]. In solid mechanics, three-dimensional deformable objects are modeled by the nonlinear partial differential equations of elasticity. After applying FEM, one obtains an ordinary differential equation,

$$M\ddot{u} + D\dot{u} + f_{\text{int}}(u) = f_{\text{ext}}(t), \tag{1}$$

where $u \in \mathbb{R}^{3n}$ contains the displacements of the n mesh vertices away from the rest configuration, $M \in \mathbb{R}^{3n\times 3n}$ is the *mass matrix*, $D = \alpha M + \beta K(u) + \overline{D}$ is the *damping matrix*, $f_{\text{int}}(u) \in \mathbb{R}^{3n}$ are the *internal elastic forces*, and $f_{\text{ext}}(t) \in \mathbb{R}^{3n}$ are *external forces* (e.g., gravity, wind, contact, user forces). The gradient of $f_{\text{int}}(u)$ with respect to u, $K(u) = \partial f_{\text{int}}(u)/\partial u \in \mathbb{R}^{3n\times 3n}$, is called the *tangent stiffness matrix*. Equation 1 covers many deformable models used in computer graphics, including FEM deformable models, cloth and mass-spring systems. Scalar parameters α and β and matrix $\overline{D} \in \mathbb{R}^{3n\times 3n}$ control damping: α sets the level of "mass" damping (slows down deformations globally, as in underwater damping), β sets the "stiffness" damping (damps primarily any relative deformation velocity differences; useful to remove temporal high-frequency instabilities), and $\overline{D}$ is an additional damping matrix that can optionally be set by the user. For $\overline{D} = 0$, we obtain the familiar Rayleigh damping [Sha90].

A deformable object simulator must timestep Equation 1 forward in time, under some user-provided initial and boundary conditions. Initial conditions include the initial position and velocity of the deformable object. For boundary conditions, the user typically selects an arbitrary set of fixed vertices, i.e., vertices whose deformation is zero at all times. The simulator must incorporate the ability to store 3D volumetric meshes, including their (potentially spatially-varying) material properties. It must also provide the ability to calculate matrices $M, D, K(u)$ and internal forces $f_{\text{int}}(u)$. There is a certain degree of independence in these tasks. Calculation of internal forces does not depend on the specific numerical integration scheme to timestep Equation 1, and vice versa. Integrators do not depend on the type of the volumetric mesh. These observations motivated the design of our Vega library, which provides all of these components, and exposes and exploits independence. Our C++ classes can be divided into three groups (Figure 3): (1) volumetric mesh containers (green), (2) internal forces calculators (red) and (3) integrators (blue). Each of the classes is a stand-alone unit that can be reused independently of the rest of the code.

Volumetric mesh containers store the 3D locations of the mesh vertices in the undeformed configuration, element connectivity, and the material properties of each element. Our classes support operations such as loading and saving a mesh to a file, calculating mesh volume, center of mass, inertia tensor and mesh mass matrix, identifying mesh elements neighboring a vertex, and interpolating vertex deformations to a higher-resolution embedded rendering mesh. Such interpolation is very useful in practice as it makes it possible to drive a detailed triangle mesh with a coarse simulation [JBT04, MTG04]. We interpolate using the CPU, but a GPU implementation would also be possible. The material properties (Young's modulus, Poisson's ratio, and mass density) may be defined per-element. Alternatively, subsets of the elements can be defined, and then the properties are applied commonly to all the elements in a subset. In order to specify the 3D geometry and support such material properties, Vega introduces an ASCII file format `.veg`, extended from the familiar free 3D mesh geometry format of Jonathan Shewchuk and used by the TetGen 3D mesher [Han11]. We also provide a routine to compute the (consistent, non-lumped) mass matrix M, which can be lumped to a diagonal matrix if so desired (more on lumping in §7).

Internal force and stiffness matrix calculators evaluate the internal forces $f_{\text{int}}(u)$ and tangent stiffness matrices $K(u)$ for all the material models stated in the Introduction, for any deformation u. Our classes are:

```
CorotationalLinearFEM
StVKInternalForces    StVKStiffnessMatrix
IsotropicHyperelasticFEM    MassSpringSystem
```

Each class is initialized by providing a volumetric mesh and the parameters needed for the material model. Some mate-

rials (mass-spring system, Saint-Venant Kirchhoff) support both tet and cube meshes, while others were only designed for tet meshes [MG04, ITF04, TSIF05]. The resulting internal forces and stiffness matrices can be timestepped with any numerical scheme, either one provided by our code, or any other timestepping scheme of choice. In order to establish a common interface to our integrators, we standardize access to the different material models through the "SparseInternalForceModel" abstract base class. This "black-box" class merely defines the following two abstract **virtual** functions:

```
virtual void GetInternalForce(double * u,
   double * internalForces) = 0;
virtual void GetTangentStiffnessMatrix(
   double * u, SparseMatrix *
   tangentStiffnessMatrix) = 0;
```

For each of our material classes, we then provide a short wrapper class that derives from SparseInternalForceModel, and that implements the above functions by calling the functions in the material class. For example, MassSpringForceModel derives from SparseInternalForceModel, and is implemented by calling functions from the MassSpringSystem class. Such a construction is very useful in practice because it decouples the integrators from the material models. Our integrators work simply with a pointer to the abstract base class "SparseInternalForceModel", exploiting C++ virtual derivation polymorphism, and are completely agnostic of the specific material model implementation. Similarly, material models need not be aware of integrators and are implemented free of any integration-related code, making it possible to use the internal forces and stiffness matrices for any general purpose, e.g., inverse kinematics or optimization.

Integrators timestep a deformable object (or any other physical system) forward in time. They obtain internal forces and stiffness matrices from an instance of a SparseInternalForceModel class. Our integrator classes are:

```
ImplicitNewmarkSparse
ImplicitBackwardEulerSparse
CentralDifferencesSparse   EulerSparse
```

Support classes: In addition to the "core" classes described above, Vega also includes classes to store and perform arithmetics on sparse matrices, load and store obj meshes, render deformed obj meshes using OpenGL, measure C++ code execution time, and parse custom-defined configuration files. In order to facilitate code re-use, we provide a demo application, which allows the user to pull on a mesh with the mouse and see the resulting dynamic deformations in real-time.

Free-flying objects: Without any fixed vertices, the object possesses free rigid degrees of freedom in addition to the deformations, which causes the stiffness matrix K to be singular. The dimension of the nullspace of K equals the number of free rigid degrees of freedom, which is six if no vertices are constrained, and less depending on what vertices are constrained. For example, if only one vertex is constrained, the object will be able to freely rotate around that vertex (three free degrees of freedom; nullspace of K will consist of all infinitesimal rotations). By constraining at least three vertices that do not lie on the same line, all rigid degrees of freedom can be removed. In practice, however, we usually constrain more than three vertices if the intent is to remove all rigid degrees of freedom. We note that our dynamic simulations remain stable even for deformable objects that simultaneously undergo both free-flying rigid motion and deformations. Namely, the linear system that must be solved at each timestep under implicit integration, is a weighted sum of the mass matrix, damping matrix and stiffness matrix, with integrator-dependent, positive weights. The presence of the mass and damping matrix terms shifts the system eigenvalues in a positive direction, which regularizes the system.

Overview simulation times are given in Table 1 (single-processor six-core 3.33 GHz Intel i7 CPU with 9GB RAM running Ubuntu Linux 11.04). More detail is provided in Tables 2, 3 and 4. All our simulations use semi-implicit integration, i.e., performing a single step of the Newton-Raphson relaxation procedure at every timestep [BW98].

4. Elastic Materials

We now describe our supported deformable models: co-rotational linear elasticity, standard Saint-Venant Kirchhoff, invertible neo-Hookean and invertible Saint-Venant Kirchhoff models, and comment on their strengths, weaknesses and practical performance. Ultimately, each of these materials results in an implementation of the "GetInternalForce" and "GetTangentStiffnessMatrix" routines (Section 3), and can then be used with any of our integrators. Table 2 gives a speed comparison of the supported deformable models.

4.1. Co-Rotational Linear Elasticity

The co-rotational linear elasticity FEM model [MG04] is a very popular solid 3D nonlinear deformable model in computer graphics. Purely linear models result in inflated volume artifacts under large deformations. The co-rotational model removes linearization artifacts, by assuming that the deformation, at every element in the mesh, consists of a rotation plus a small amount of "pure" deformation. The rotation is determined via polar decomposition of the deformation gradient F. Once the rotation is identified, the forces on an element are computed based on only the "pure" deformation, and properly rotated to the world frame of reference. Our implementation supports tet meshes, and includes a polar decomposition library. Optionally, the caller can disable "warping", which yields a standard linear FEM model, with a constant stiffness matrix K, and internal forces $f^{\text{lin}} = Ku$.

Specifically, our code implements the following calculation. In the standard (i.e., non-warped) linear model, the 12-dimensional vector $f_e^{\text{lin}} = K_e u_e$ of vertex forces of an element e is a linear function of the element stiffness matrix

	#ver	#el	total [msec]	N	#Rver	#Rtri	interp [msec]	normals [msec]	fps
dinosaur (T)	344	1,031	39.7	2	28,098	56,192	0.8	21.7	13.2
bridge (V)	3,923	1,736	480	8	16,085	16,616	0.4	8.1	1.9
bridge (T)	4,000	12,827	361	8	16,085	16,616	0.3	8.3	2.4
crane (V)	9,875	5,571	1,574	2	149,797	260,187	8.1	99.1	0.5
dragon (T)	46,736	160,553	6,783	1	46,736	77,250	0	23.1	0.1

Table 1: ***Simulation statistics:*** *T=tet mesh, V=voxel mesh, #vertices (#ver), #elements (#el), total dynamics time (total; includes internal forces and stiffness matrix evaluation and integrator time, as separately reported in Tables 2 and 3), #simulation steps per rendered frame (N), #vertices in rendering mesh (#Rver), #triangles in rendering mesh (#Rtri), time to interpolate deformations from volumetric to triangle mesh for rendering (interp), time to dynamically recompute the mesh normals for rendering (normals), and the overall achieved frame rate (fps), using OpenGL, with dynamically recomputed normals. The dragon interpolation time is zero, because that model uses the outer surface of the volumetric mesh as the rendering mesh. All timings are in msec per time step, using a single core, for the StVK material model integrated with implicit Newmark.*

	co-rotational linear FEM	StVK		invertible StVK		invertible Neo-Hookean		mass-spring system	
		INT	STM	INT	STM	INT	STM	INT	STM
dinosaur	2.1	14.5	18.5	0.1	3.5	0.1	3.2	0.1	0.3
bridge (V)	–	136	256	–	–	–	–	1.2	5.3
bridge (T)	32.1	99.8	191	0.8	40.1	1.6	41.3	0.8	4.1
crane	–	443	828	–	–	–	–	3.1	20.1
dragon	484	1,219	2,737	15.5	582	21.2	599	17.1	70.8

Table 2: ***Internal force (INT) and stiffness matrix (STM) evaluation times,*** *in milliseconds per time step, using a single core. Co-rotational FEM timings are reported total for INT+STM, because they cannot easily be separated in the implementation.*

$K_e \in \mathbb{R}^{12\times 12}$ and the displacement $u_e = x_e - x_e^0 \in \mathbb{R}^{12}$ of its vertices from their rest position $x_e^0 \in \mathbb{R}^{12}$. After calculating the rotational component R_e of the element deformation gradient using polar decomposition, one assembles a 12×12 block-diagonal matrix $\hat{R}_e$, with four 3×3 blocks R_e. One then obtains *co-rotational vertex forces* by applying K_e to the rotation-canceled deformations, followed by a rotation,

$$f_e = \hat{R}_e K_e(\hat{R}_e^T x_e - x_e^0) = \hat{R}_e K_e \hat{R}_e^T u_e + \hat{R}_e K_e(\hat{R}_e^T - I)x_e^0. \quad (2)$$

Note that for $u_e = 0$, we have $R_e = I$, and $f_e = 0$, as expected. For implicit integration, we use a stiffness matrix assembled from the warped stiffness matrices $\hat{R}_e K_e \hat{R}_e^T$ of each tetrahedron, which is a very common choice in computer graphics [MG04, PO09]. Although this choice in practice gives stable simulations (except in cases of extreme velocities), we note that this matrix is *not* the exact gradient of the internal forces ($df_e/du_e \neq \hat{R}_e K_e \hat{R}_e^T$), because the gradients of R_e with respect to u_e are ignored. This limitation has been recently addressed by [CPSS10] and [Bar12]; an implementation is available in Vega. All experiments in this paper refer to the approximate stiffness matrix $\hat{R}_e K_e \hat{R}_e^T$ [MG04].

4.2. Saint-Venant Kirchhoff Elasticity

The Saint-Venant Kirchhoff material model is another commonly employed deformable model in computer graphics. It uses nonlinear Green-Lagrange strain $E = 1/2(F^T F - I)$, which ensures that the model is free of large rotation artifacts. StVK is perhaps the simplest nonlinear model, because it models the stress-strain relationship with a linear function. Unlike general linear materials which can be anisotropic, StVK has the additional property that it is isotropic, and is therefore sometimes referred to as the isotropic geometrically nonlinear material model. Because of all of these assumptions, StVK can be parameterized by only specifying two scalar values. Two representations are commonly employed and supported by our code: the Lamé coefficients λ, μ, and the Young's modulus and Poisson's ratio E, ν. StVK is given by the energy density function (see [BW08])

$$\Psi = \frac{1}{2}\lambda(\mathrm{tr}(E))^2 + \mu E : E. \quad (3)$$

It can be shown that the StVK elastic energy is a quartic (4th order) polynomial in the deformations of the mesh vertices (see, e.g., [CGC*02, Bar07]). Therefore, the internal forces and the tangent stiffness matrix are cubic and quadratic polynomials, respectively. The coefficients of these polynomials can be derived analytically, using integration of FEM shape functions over each element. They are given in [CGC*02, Bar07]. With general deformable models, such integration can only be performed numerically. One advantage of StVK is that the integration can be performed exactly during precomputation, with a negligible computational overhead. We performed the integration analytically in Mathematica for both tets (of arbitrary shapes) and cubes, and transferred the formulas to efficient C code. Our StVK implementation is then immediately loadable and requires only a small memory footprint to store the cubic polynomial coefficients. Be-

cause the StVK internal forces are cubic polynomials, it is very easy to compute, analytically, the exact StVK tangent stiffness matrix, and even further derivatives. Our code provides the StVK stiffness matrix and also its derivative (the Hessian of the internal forces). Such high-order derivatives are useful in applications involving optimization and control of deformable models, as they make it possible to use higher-order optimization schemes. Furthermore, the cubic polynomial formula is useful in model reduction [BJ05], because linear projections to low-dimensional spaces commute with any polynomial expression.

We observe two disadvantages of StVK. First, under large compression, the material collapses, and may even permanently invert. This problem generally applies to soft tissue; stiffer models are less prone to collapse. To address invertibility, we provide an invertible StVK implementation (Section 4.3). The second disadvantage of a cubic-polynomial-based StVK implementation is that, although exact, the number of cubic polynomial terms is quartic in the degrees of freedom of the element (see Table 2). Therefore, the evaluation of internal forces is slower with StVK than, say, with co-rotated linear models. Note that for tetrahedral elements, one may use the invertible StVK method of Section 4.3, which is faster (see Table 2) and gives identical results to standard StVK when the correction of inverted elements is disabled. The invertible FEM approach, however, cannot easily give stiffness matrix derivatives (Hessians). We note that if invertibility is not needed, it would be possible to gain further speed in StVK (for tet meshes) by computing the first-Piola Kirchoff stress tensor P (Section 4.3), and analytically differentiate it with respect to F (avoiding SVD); we leave this extension for future work. We compare the stretching behavior of co-rotational linear FEM to StVK in Figure 4. It can be seen that the two materials behave quite differently under large deformations. Under small deformations, however, the two materials are visually similar (see Figure 9, bottom).

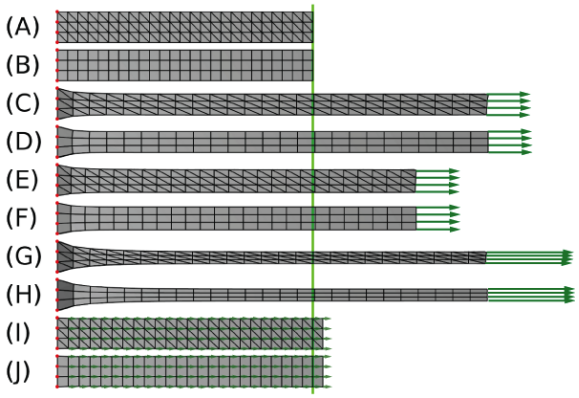

Figure 4: ***Material behavior under stretching:*** *(A) (B) cantilever beam at rest, with tets and cubes. Pulling end of beam: (C) co-rotational linear FEM with tets, (D) linear FEM with cubes, under the same forces as C, (E) (F) StVK, under the same forces as C, (G) (H) StVK, stretched to the same length as C. Applying gravity: (I) (J) StVK, under same total force as C. Poisson's ratio was 0.45. Because there are almost no local rotations, (C) and (D) coincide, and internal forces are linear in displacement. StVK, however, stiffens nonlinearly. Volumetrically distributed loads (gravity) produce smaller displacements than concentrated loads.*

4.3. Invertible Element Methods

The material models above share the limitation that if a tet becomes inverted due to extreme deformation, the internal forces do not act to restore the tet to an uninverted state. The invertible model [ITF04] addresses this weakness by calculating material stress using the singular value decomposition (SVD) of the deformation gradient F, where inversion is easily detected. The vertex forces on a deformed tet rotate with the tet if the tet is rotated in world space. For isotropic materials, the forces are similarly invariant with respect to rotations in material space, so the first Piola-Kirchhoff stress $P(F)$, viewed as a function of F, satisfies

$$P\left(U\hat{F}V^T\right) = UP\left(\hat{F}\right)V^T, \tag{4}$$

where $F = U\hat{F}V^T$, and U and V^T are rotations such that $\hat{F}$ is diagonal. One can now determine that the tet is near-inverted if one of the diagonal entries $\lambda_1, \lambda_2, \lambda_3$ of $\hat{F}$ is below a small positive threshold, and clamp such entries to that threshold to produce forces that restore the tet. We found such clamping to work well in practice. For stress defined by an energy function Ψ, it can be shown that $\hat{F}$ yields diagonal stress

$$P\left(\hat{F}\right) = \mathrm{diag}\left(\frac{\partial\Psi}{\partial\lambda_1}, \frac{\partial\Psi}{\partial\lambda_2}, \frac{\partial\Psi}{\partial\lambda_3}\right). \tag{5}$$

The values $\partial\Psi/\partial\lambda_i$ can be computed analytically by differentiating the energy function of any standard isotropic material model, or they could even be measured experimentally. Once we calculate $P(F)$ from $P(\hat{F})$, we obtain the force on a vertex of the tet by multiplying $P(F)$ with its area-weighted normal (see [ITF04] for details).

Implicit timestepping methods require the stiffness matrix for the invertible model. Such a stiffness matrix was derived in [TSIF05], and we implement this method in Vega. Using the diagonalization as in Equation 5, we calculate the entries of the gradient of the stress with respect to F as

$$\frac{\partial P}{\partial F_{ij}} = U\left(\left.\frac{\partial P}{\partial F}\right|_{U^TFV} : \left(U^T(e_i \otimes e_j)V\right)\right)V^T, \tag{6}$$

where e_i is the ith standard basis vector in $\mathbb{R}^3$. The gradient evaluated at U^TFV can be calculated as a function of the λ_i and the gradient and Hessian of Ψ with respect to the invariants of $C = F^TF$ (for more details, please refer to [TSIF05]). The invertible FEM models in Vega are able to recover very well from extremely inverted configurations (see Figure 5).

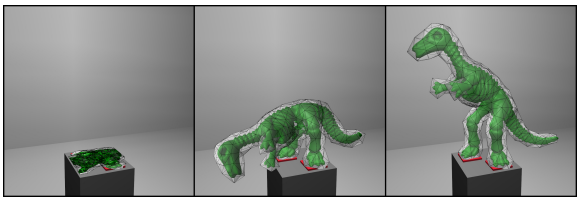

Figure 5: ***Our invertible simulator can recover from extreme deformations (invertible StVK):*** *(A) collapsed (initial condition), (B) partially recovered (intermediate state), (C) completely recovered (final state).*

Invertible Saint-Venant Kirchhoff material adds invertibility to the StVK material model given in Section 4.2. We first rewrite the energy density from Equation 3 in terms of the invariants of $C = F^T F$:

$$\Psi = \frac{1}{8}\lambda(I_C - 3)^2 + \frac{1}{4}\mu(II_C - 2I_C + 3). \tag{7}$$

Invariants are defined as follows [BW08]:

$$I_C = \mathrm{tr}(C) = \lambda_1^2 + \lambda_2^2 + \lambda_3^2, \tag{8}$$

$$II_C = \mathrm{tr}(C^2) = \lambda_1^4 + \lambda_2^4 + \lambda_3^4, \tag{9}$$

$$III_C = \det(C) = \lambda_1^2\lambda_2^2\lambda_3^2. \tag{10}$$

We then analytically differentiate energy (Equation 7) with respect to the invariants (first and second derivative), and then apply these formulas to our implementation of [TSIF05] (Section 4.3).

Invertible Neo-Hookean material uses the energy density function (page 162 in [BW08])

$$\Psi = \frac{\mu}{2}(I_C - 3) - \mu \log J + \frac{\lambda}{2}(\log J)^2, \tag{11}$$

where μ and λ are the Lamé coefficients, and $J = \sqrt{III_C}$.

Invertible Mooney-Rivlin material uses the energy density function (see §3.5.5 in [Bow09])

$$\Psi = \frac{1}{2}\mu_{01}\left(\frac{I_C^2 - II_C}{J^{4/3}} - 6\right) + \mu_{10}\left(\frac{I_C}{J^{2/3}} - 3\right) + \nu_1(J-1)^2, \tag{12}$$

where $\mu_{01}, \mu_{10}, \nu_1$ are Mooney-Rivlin material parameters.

Arbitrary isotropic hyperelastic materials are supported by Vega, via C++ class polymorphism. One simply has to provide a class that derives from a standardized Vega abstract base class for isotropic materials, and that evaluates the first and second derivatives of the energy density Ψ with respect to I_C, II_C, III_C.

4.4. Mass-Spring System

The mass-spring system model simulates a set of particles connected by springs [BW01]. Each particle can have its own mass; similarly, different springs can have different stiffness and damping coefficients, as well as rest lengths. We support general mass-spring networks; there is no restriction on which particles are connected by springs. To facilitate the creation of a mass-spring network, we provide a routine which converts a tet mesh into a mass-spring system: vertices become particles, and each tet edge serves as a spring. Similarly, one can convert a cube mesh into a mass-spring network: all 12 cube edges become springs, along with the 12 face diagonals and the 4 main diagonals. We compute spring forces using the standard Hooke's law for springs. For a spring of rest length r and stiffness k_s between particles at positions a and $b = a + z$, the spring force is

$$f(z) = k_s(|z| - r)\frac{z}{|z|}, \tag{13}$$

such that particle a receives force $f(z)$ and particle b receives $-f(z)$. We compute damping forces in a similar way. To enable implicit integration, we calculate the gradient of f as

$$\frac{\partial f}{\partial z} = k_s\left[\left(1 - \frac{r}{|z|}\right)I + \frac{r}{(|z|)^3}zz^T\right], \tag{14}$$

from which we obtain the contribution of each spring to the global stiffness matrix. We also derive and implement the derivative of the stiffness matrix (Hessian).

5. Integrators

We support several explicit and implicit integration methods (see Table 3 for a speed comparison). All the methods timestep the ODE given in Equation 1. Except with soft objects, FEM deformable simulations are very stiff and require implicit integration with large timesteps. For explicit integration, the required timestep is imposed by the smallest element in the mesh (Courant condition).

	IBE	IN	ECD	EE
dinosaur	5.29	6.70	1.36	0.67
bridge (V)	90.3	87.0	20.1	9.54
bridge (T)	73.9	70.0	19.0	9.81
crane	339	303	53.8	25.0
dragon	2,856	2,827	239	110

Table 3: ***Integrator times:*** *IBE=implicit backward Euler, IN=implicit Newmark, ECD=explicit central differences, EE=explicit Euler. All timings are in msec per time step, using a single core. They include the entire time step computation, except internal force and stiffness matrix evaluation. For implicit integration and central differences, they include the Pardiso linear system solver time. Central differences were timed using a constant pre-factored system matrix.*

5.1. Implicit Backward Euler

Given deformations u_t and velocities v_t at time t, the implicit backward Euler method [BW98] attempts to find a future

deformation $u_{t+\Delta t}$ such that explicitly integrating at time t with forces $f_{\text{int}}(u_{t+\Delta t})$ evaluated at $t+\Delta t$ will produce the same deformation $u_{t+\Delta t}$. If we let $\Delta u = u_{t+\Delta t} - u_t$ and $\Delta v = v_{t+\Delta t} - v_t$ represent the changes in deformation and velocity from time t to $t+\Delta t$, then we want to set

$$\begin{bmatrix}\Delta u \\ \Delta v\end{bmatrix} = \Delta t \begin{bmatrix}\Delta v + v_t \\ M^{-1}(-D v_{t+\Delta t} - f_{\text{int}}(u_{t+\Delta t}) + f_{\text{ext}})\end{bmatrix}. \quad (15)$$

To find approximate solutions when f_{int} is nonlinear, we substitute a Taylor series approximation of $f_{\text{int}}(u_{t+\Delta t})$, computed using the tangent stiffness matrix $K = \partial f_{\text{int}}/\partial u$ (*semi-implicit* integration). This gives the linear equation

$$\left(M + \Delta t D + (\Delta t)^2 K\right)\Delta v = \Delta t\left(f_{\text{ext}} - f_{\text{int}}(u_t) - (\Delta t K + D)v_t\right). \quad (16)$$

After solving Equation 16 for Δv, we can calculate Δu from Equation 15. Such semi-implicit backward Euler is *not* unconditionally stable, unlike fully implicit backward Euler where the nonlinear Equation 15 is solved exactly. Such an exact solver is largely only a theoretical concept, however, as in practice, *finding* the required exact solution can only be done via a full Newton-Raphson relaxation at every timestep. This search may diverge if the initial guess is too far from the exact solution. Consequently, both semi-implicit and fully-implicit integration can become unstable under very large timesteps. In practice, however, semi-implicit backward Euler is a reasonable choice; it is both simple and very stable. Its main drawback is that it tends to introduce significant artificial damping with large timesteps.

5.2. Implicit Newmark

The implicit Newmark integrator is very popular in solid mechanics, mainly because of its quadratic-order timestep accuracy [Woo90]. In practice, it is slightly less stable than implicit backward Euler, but produces more lively animations with less artificial damping. Given current deformations u_t, velocities v_t, and accelerations a_t at time t, the implicit Newmark method [Wri02] assumes that the future deformations, velocities and accelerations $u_{t+\Delta t}, v_{t+\Delta t}, a_{t+\Delta t}$ satisfy

$$\begin{aligned} u_{t+\Delta t} &= u_t + \Delta t \cdot v_t + \frac{(\Delta t)^2}{2}\left((1-2\beta)a_t + 2\beta a_{t+\Delta t}\right) \\ v_{t+\Delta t} &= v_t + \Delta t\left((1-\gamma)a_t + \gamma a_{t+\Delta t}\right), \end{aligned} \quad (17)$$

where $0 \le \beta \le 0.5$ and $0 \le \gamma \le 1$ are user-chosen parameters. In practice, these values are often set to $\beta = 0.25$ and $\gamma = 0.5$, which gives second-order accuracy. To satisfy the above equations, we solve for $a_{t+\Delta t}$ and $v_{t+\Delta t}$ in terms of $u_{t+\Delta t}$ and the known values at time t. Inserting the results into Equation 1, we obtain a nonlinear equation for $u_{t+\Delta t}$:

$$\begin{aligned} G(u_{t+\Delta t}) = M\Big(\frac{1}{\beta(\Delta t)^2}(u_{t+\Delta t} - u_t) - \frac{1}{\beta \Delta t} v_t - \frac{1-2\beta}{2\beta} a_t\Big) + \\ + D\Big(\frac{\gamma}{\beta \Delta t}(u_{t+\Delta t} - u_t) + \Big(1 - \frac{\gamma}{\beta}\Big)v_t + \Big(1 - \frac{\gamma}{2\beta}\Big)\Delta t \cdot a_t\Big) + \\ + f_{\text{int}}\left(u_{t+\Delta t}\right) - (f_{\text{ext}})_{t+\Delta t} = 0. \end{aligned} \quad (18)$$

To solve for $u_{t+\Delta t}$, we start with $u^0_{t+\Delta t} = u_t$. We then perform a Newton-Raphson procedure, iteratively generating updated guesses $u^{i+1}_{t+\Delta t}$, by using the stiffness matrix and f_{int} at the current guess $u^i_{t+\Delta t}$:

$$\left(\frac{1}{\beta(\Delta t)^2}M + \frac{\gamma}{\beta \Delta t}D + \left.\frac{\partial f_{\text{int}}}{\partial u}\right|_{u=u^i_{t+\Delta t}}\right)\Delta u^{i+1}_{t+\Delta t} = -G(u^i_{t+\Delta t}), \quad (19)$$

where $u^{i+1}_{t+\Delta t} = u^i_{t+\Delta t} + \Delta u^{i+1}_{t+\Delta t}$. We halt the process when $G(u^i_{t+\Delta t})$ falls below a desired accuracy threshold, or after exceeding a maximum number of iterations.

Sparse linear solvers are required for implicit integration. Vega employs three solvers: Pardiso (commercial; direct; by the Pardiso Project and Intel [PAR]), SPOOLES (free; direct; by Boeing [SPO]), and our own Jacobi-preconditioned Conjugate Gradient (PCG) implementation (free; iterative; released with Vega), which we implemented by following [She94]. We give a timing comparison in Table 4. In most cases, direct solvers outperform PCG in computation time, but require more memory. Direct solver times do not depend on material stiffness (Young's modulus), whereas the number of required PCG iterations is proportional to stiffness (see also §7). Therefore, PCG is a better alternative to direct solvers with soft objects, and with large systems where the direct factorization may not fit into memory.

5.3. Central Differences

The central differences integrator [Wri02] is the explicit companion to the implicit Newmark integrator. It typically requires much smaller timesteps than implicit Newmark, and is useful in simulations that must take very small timesteps, e.g., simulations involving transient contact or sound simulations. The update equation is

$$\left(M + \frac{\Delta t}{2}D\right)\left(u_{t+\Delta t} - u_t\right) = \quad (20)$$

$$= (\Delta t)^2\left(f_{\text{ext}} - f_{\text{int}}(u_t)\right) + \frac{\Delta t}{2}D(u_{t-\Delta t} - u_t) + M(u_t - u_{t-\Delta t})$$

$$v_{t+\Delta t} = \frac{u_{t+\Delta t} - u_t}{\Delta t}. \quad (21)$$

A central differences timestep is faster than the implicit Newmark timestep because there is no need to construct the tangent stiffness matrix. Furthermore, if the damping matrix D does not vary through time, the matrix $M + \frac{\Delta t}{2}D$ can be prefactored. For Rayleigh damping, however, the matrix $D(u) = \alpha M + \beta K(u)$ varies in time. While it is tempting to ignore the dependency of K on u and simply use a constant stiffness matrix $K = K(0)$ for damping, we observed that this leads to very visible damping artifacts under large deformations. A reasonable alternative to using exact $K(u)$ is to periodically recompute $K(u)$ and refactor $M + \frac{\Delta t}{2}D(u)$.

Explicit Euler: We also provide explicit and symplectic Euler integrators. Symplectic Euler [SD06] preserves energy and is typically more stable than explicit Euler.

	time [msec]			memory [Mb]		
	Par	SPO	PCG	Par	SPO	PCG
dinosaur	5.1	16.4	4.6	1.3	1.1	0.03
crane	308	566	1,012	59.5	62.2	0.90
bridge (T)	72	136	9.1	16.5	16.2	0.36
bridge (V)	73	133	13.3	19.3	18.9	0.36
dragon	2,821	5,039	8,934	289	320	4.3

Table 4: ***Comparison of sparse solvers:*** *Two direct solvers (Par=Pardiso, SPO=SPOOLES) are compared to an iterative solver (PCG; Jacobi-preconditioned conjugate gradients). Timings are in msec per time step, using a single core. PCG uses zero initial guess and stops when the residual norm drops below $\varepsilon = 10^{-6}$ times the initial residual norm. Multicore scalability and the dependence of running time on material stiffness are investigated in Figure 11.*

6. CPU Multicore Implementation

We provide a CPU multicore implementation for all our material models, for both the internal forces and stiffness matrices. The implementation is available in classes with extension "MT" that derive from a single-core version of the class, e.g., "StVKInternalForcesMT" is derived from "StVKInternalForces". The user provides the desired number of computation threads T to the "MT" constructor. The routine to compute the internal forces then launches T threads, using the pthreads API (available on Linux, Mac OS X and Windows). The set of all mesh elements is divided into T "buckets", each of which is assigned to a thread. Each thread then processes its elements. For each element, it computes the internal forces, and then adds the forces into its own, separate, internal force buffer. This avoids write-conflicts, and removes the need for any thread synchronization. The buffers are of length $3n$, static, allocated once in the constructor, and cleared to zero by the thread at the beginning of each computation. After all threads are finished, the main thread sums the buffers into a total internal force. We use an equivalent procedure for stiffness matrices. The Pardiso and SPOOLES solvers support multithreading, and we wrote wrappers where we can conveniently set the number of solver threads. In practice, solver scalability is good for 2-3 cores, but diminishes with more cores. Figure 6 analyzes the performance of our multicore implementation.

In our implementation, we launch and kill the threads each time internal forces or stiffness matrices are computed. The advantage of this approach is that it leaves room for other (non-simulation) tasks to be mapped to the different cores. Because there is some OS overhead in launching the threads, our threading stops becoming useful with very small models, e.g., when the running time of each thread would be less than approximately 10 milliseconds. This is especially pronounced with mass spring systems where the internal force computation times are short. It would be easily possible to modify our code so that the threads are launched once and then persist, spinning idle on a mutex until needed. Note that the threads cannot be made to sleep, because the time to wake them up on a multitasking OS could easily be on the order of a few milliseconds.

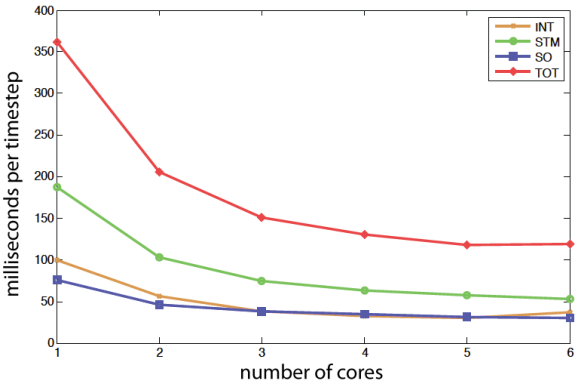

Figure 6: ***CPU multicore scalability:*** *INT=internal forces, STM=stiffness matrix, SO=solver time, TOT=total time. Bridge example with tets, StVK material, Pardiso solver, implicit backward Euler. Single-processor six-core 3.33 GHz Intel i7 CPU with 9GB RAM. Both INT+STM and solver can be seen to scale well to a few cores, with diminishing returns for many cores. INT+STM evaluations are more parallel and scale better than Cholesky decomposition and back-substitution inside direct sparse solvers. ITM+STM scalability is affected by contention to read the vertex displacements from memory, and the overhead of launching the treads.*

7. Experiments

Computation times were reported in Tables 2, 3, 4 and Figure 6. Performance under mesh and timestep refinement are analyzed in Figures 7 and 9, respectively. We also analyzed stability under mesh and timestep refinement, by applying an instantaneous force impulse to a vertex at the top of bridge's mast, in tetrahedral meshes L0, L1, L2 of Figure 9. We determined the largest stable timestep using bisection. Under implicit backward Euler, StVK and co-rotational linear FEM were both stable. Huge timesteps caused severe numerical viscosity, but simulations did not explode, for any mesh resolution. Under implicit Newmark with StVK, the largest stable timesteps at L1 and L2 meshes were 1.11x and 1.19x smaller than at L0, respectively. Co-rotational linear FEM stable timesteps were approximately 2x larger than StVK timesteps, at all mesh resolutions.

In Figures 8 and 10, we analyze volume preservation and locking of tetrahedral and voxel meshes. Unless the material is very soft, the standard FEM methods preserve volume well (but not highly accurately), across a wide range of Poisson ratios ν (Figure 10). Even rarely used values such as $\nu = -0.5$ or $\nu = 0$ produce visually plausible results.

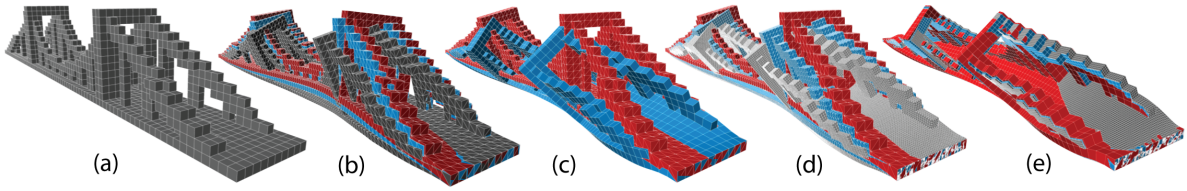

Figure 8: ***Locking*** *under a static load. Data also corresponds to Figure 10. (a) Rest pose, (b) tet mesh obtained by splitting voxels into tets, under* $\nu = 0.45$ *(gray),* 0.49 *(blue),* 0.499 *(red); severe locking can be observed, (c) tet mesh (red) vs voxel mesh (blue), for* $\nu = 0.49$*; the voxel mesh locks significantly less than the tet mesh, (d) (e) tet and voxel mesh, respectively, under progressive voxel subdivision, for* $\nu = 0.49$*, L0 (red), L1 (blue), L2 (gray); under subdivision, locking in the tet mesh becomes less severe; a similar (but much smaller) effect can be observed in the voxel mesh. Same static load in (b)-(e).*

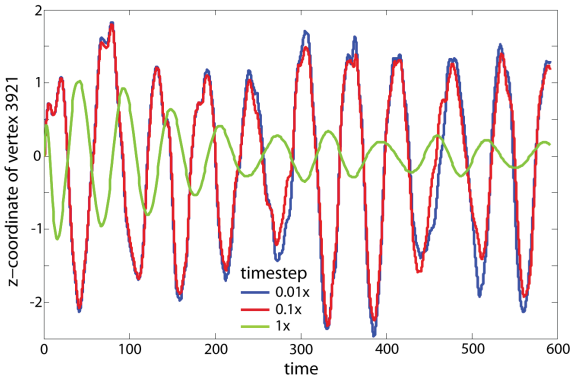

Figure 7: ***Timestep refinement:*** *We simulated the L0 bridge tet mesh (Figure 2 (b), and Table 1, StVK,* $\nu = 0.45$*, 4000 vertices; small user damping), under progressive timestep refinement. All simulations use the same initial velocity, followed by free vibration. We plot the trajectory of a vertex at the top of the bridge mast. Under a large timestep (denoted by 1x), numerical viscosity due to implicit integration is high, causing a rapid loss of energy. The rate of energy loss decreases as the deformations become smaller. Numerical viscosity is smaller under smaller timesteps. Smaller timesteps also produce greater dynamic detail. For scale: the height of the bridge is 3 units. Z-axis is perpendicular to the main bridge axis, and the mast.*

The deformable methods presented in this work, however, struggle with near-incompressible Poisson ratios close to 0.5. Although they produce an output that preserves volume well, they "lock": system condition numbers grow, animations lose energy rapidly, may become unstable, and visibly deform to smaller deformations than under equal loads with smaller values of ν (Figure 8, (b)). Locking becomes less severe with mesh refinement (Figure 8, (d,e)), and tetrahedral meshes lock significantly more than voxel meshes (Figure 8, (c)). In order to simultaneously avoid locking and achieve exact volume preservation, it is recommended to use FEM methods designed for this purpose, such as [ISF07].

We analyzed linear system solver multicore scalability in Figure 11. It can be seen that PCG parallelizes better than Pardiso. This is expected since matrix-vector multiplications of iterative solvers parallelize better than back-substitutions in direct solvers. The tradeoff between direct and iterative solvers is determined by the material stiffness, or, equivalently, timestep size. Running time of direct solvers is largely independent of stiffness, whereas for iterative solvers, it increases with stiffness, as well as timestep size (Figure 11). This is because the mass matrix M is typically much better conditioned than the stiffness matrix K. For low stiffness values, or small timesteps, the M term dominates the system matrix (Equations 16, 19), so PCG needs a small number of iterations for convergence. Under large stiffness or large timesteps, K is dominant, causing a higher system condition number and slower PCG convergence. In the extreme case, as timestep approaches infinity, all dynamics is removed and the solver becomes equivalent to a static solver: numerically the most difficult case.

Assembling the global stiffness matrix: Vega's sparse matrix library uses the compressed sparse row format to store sparse matrices. In order to correctly write each element's stiffness matrix into the global stiffness matrix, a naive implementation would need to sort the indices of non-zero entries in each row, at runtime. Vega avoids this overhead by pre-sorting the indices at startup. For every element, it precomputes V^2 integers, where V is the number of element vertices ($V = 4$ for tets and $V = 8$ for cubes), as follows. The $3V \times 3V$ element stiffness matrix consists of dense 3×3 blocks, each of which corresponds to a pair of element vertices. Let us consider the $n \times n$ matrix obtained from the global $3n \times 3n$ stiffness matrix, by shrinking each 3×3 block to a 1×1 matrix. Let v_i^e be the global integer index of vertex i of element e, $i = 1, \ldots, V$, in the shrunk $n \times n$ matrix. For

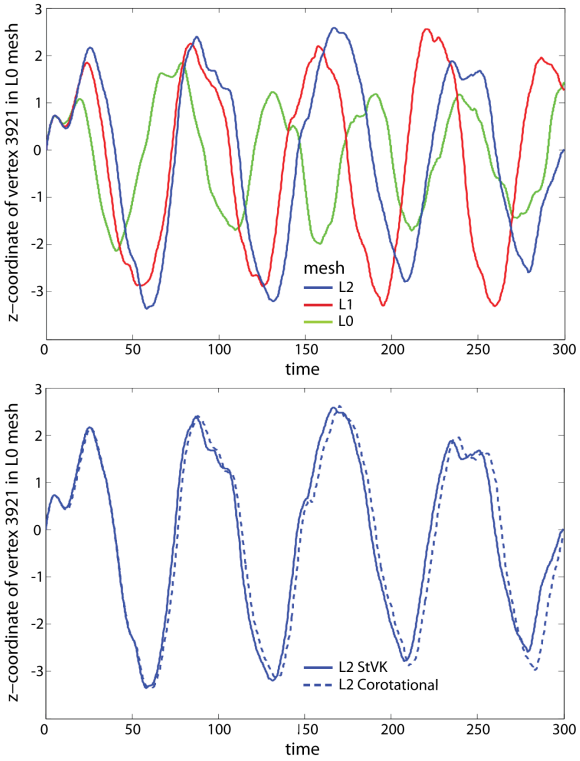

Figure 9: ***Mesh refinement:*** *Top: we simulate the same solid object (bridge), under progressive, non-nested, tet mesh refinement. The L0 mesh (4,000/12,827 vertices/tets) is also reported in Table 1 and Figure 7. L1 and L2 meshes have 20,921/83,566 and 106,827/477,589 vertices/tets respectively, and mesh approximately the same volume as the L0 mesh. Same timestep, initial velocity, material properties, damping, and plotted vertex as in Figure 7. It can be seen that finer simulations produce richer dynamic motion. Bottom: StVK vs co-rotational linear FEM.*

each element e and each integer pair $(i, j), i, j = 1, \ldots, V$, we pre-compute an integer that gives the location in the compressed array of row v_i^e of the 1×1 entry corresponding to the vertex pair (v_i^e, v_j^e) in e. Precomputing the V^2 integers for each element is fast and only needs to be done once at startup. At runtime, one can then write the element stiffness matrix entries into the global matrix in $O(1)$ time. Therefore, the global stiffness matrix assembly only requires a linear traversal of element stiffness matrix entries, with a negligible computational overhead. Note that direct solvers require all entries of the global stiffness matrix to be available for Cholesky factorization. For iterative solvers, in contrast to matrix-free approaches that need not form a global stiffness matrix and therefore reduce memory bandwidth [MZS*11], Vega computes the stiffness matrix once at each timestep, and re-uses it during the iterations.

Mass lumping refers to forming a diagonal matrix by summing the entries in each row of the mass matrix. Such a simplified matrix corresponds to pretending that the interiors of all mesh elements are massless, with all mass concentrated at the vertices. Therefore, the effect of mass lumping becomes less pronounced with mesh refinement. All experiments in this paper were performed using the consistent (non-lumped) mass matrix. The sparse matrix library in Vega exploits the fact that the non-zero indices of the consistent M are a subset of non-zero indices of K, to efficiently form the system matrix (Equations 16, 19). If the mass matrix is lumped to a diagonal, this does not yield a substantial speedup for implicit integration. It does, however, accelerate explicit integration when "stiffness" damping is not used ($\beta = 0$), because the system matrix (Equation 20) becomes diagonal.

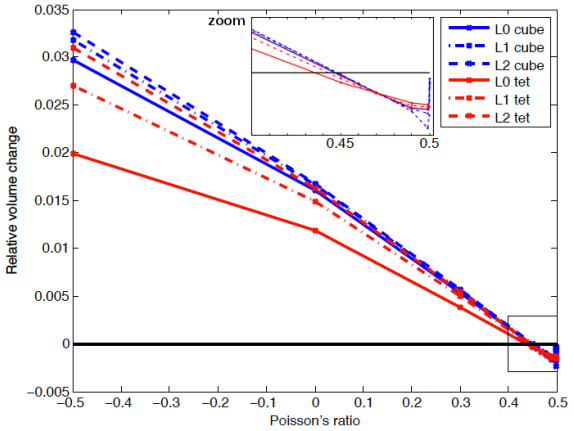

Figure 10: ***Volume preservation:*** *We measured the total mesh volume in the experiment depicted in Figure 8, as a function of* ν*, for tets and cubes, and under mesh refinement. Relative volume error is small (under* 3.5%*). We observed an interesting phenomenon: as* ν *is increased into the locking regime (*$\nu > 0.45$*), a switch occurs from overestimating to underestimating the volume. Beyond* $\nu > 0.499$*, all simulations are severely locked (see Figure 8, (b)) and very visibly under-deform. Tetrahedral mesh locks so severely that simulations become unstable. The voxel mesh, however, starts preserving the volume more closely as* $\nu \to 0.5$.

8. Discussion

We presented a simulator that implements several common FEM deformable simulation methods in computer graphics, as well as a mass-spring system, in a unified framework. We provided practical experience and extensively compared the implemented strategies. Our code is simple and easily extensible, and the different components depend minimally on each other. The simulator exposes internal forces and tangent stiffness matrices using a well-documented interface, which should facilitate code reuse in applications in computer graphics, animation, robotics and virtual reality.

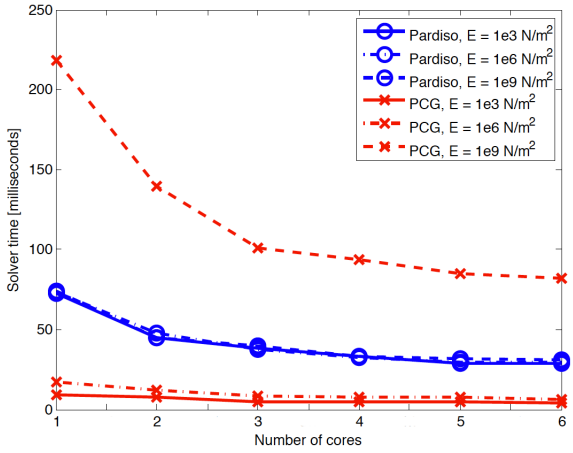

Figure 11: ***Multicore scalability of linear system solver:*** *Both Pardiso and PCG scale well with the number of cores. As material stiffness is increased, the direct solver (Pardiso) running times stay constant, whereas PCG times grow. L0 bridge tet mesh (4,000 vertices, 12,827 tets).*

Acceleration strategies: Our simulator can be readily extended to also include acceleration strategies such as model reduction [BJ05] and domain decomposition [HLB*06, BZ11]. In model reduction, one assumes that the deformations u lie in some (quality) low-dimensional space, $u = Uq$, where $U \in \mathbb{R}^{3n \times r}$ is the time-invariant basis of the subspace, and r is the basis size. Equations of motion are then projected to this low-dimensional space, yielding equations of motion for $q(t)$. A C/C++ implementation that timesteps q parallels the unreduced implementation, and can reuse several of its classes, such as the VolumetricMesh class. The key difference is that implicit solvers must solve *dense* linear systems of size $r \times r$. The basis U can be obtained by applying Principal Component Analysis (PCA) to some pre-existing unreduced simulation data (computed, say, using the unreduced Vega simulator), or using modal derivatives [BJ05]. A precomputation utility that performs the pre-process, as well as run-time code that shares some basic classes with Vega, are available at: `http://www.jernejbarbic.com/code`. For more details and a comparison to unreduced StVK, please refer to [BJ05]. For domain decomposition, the model reduction simulator can be extended, by using polar decomposition [MHTG05] to compute a frame for each domain. For details and experimental comparisons, please refer to [BZ11].

What approach is good for what application: We have found that co-rotational linear FEM and StVK behave very similarly under small / moderate deformations. With implicit integration, co-rotational linear FEM is faster ($\sim 20\%$) than invertible StVK. It is also more stable. For simulations that do not require the tangent stiffness matrix (explicit solvers), StVK is faster because one cannot easily separate stiffness matrix computation from internal force computation in the co-rotational linear FEM method. Under large compression, co-rotational linear FEM may permanently invert; therefore, for large compression, it is recommended to use QR decomposition as opposed to polar decomposition [PO09], or explicitly address invertibility [TSIF05]. Both co-rotational linear FEM and invertible StVK are good, general-purpose choices for computer graphics applications. The neo-Hookean material is typically used to simulate plastic and rubber-like materials in engineering. Mass-spring systems are faster and easier to implement than the FEM methods, but are poor at preserving volume, and can easily tangle (invert) permanently. Hexahedral meshes lock less than tet meshes, and thus are preferred in applications where volume preservation is important. Standard StVK that computes forces as cubic polynomials is significantly slower than invertible StVK that uses SVD. Therefore, it is recommended that practical StVK implementations use the invertible SVD algorithm. Such applications may even turn the invertibility threshold off, at which point invertible StVK exactly matches standard StVK, at a fraction of the computational cost. The cubic polynomial StVK algorithm, however, has the advantage that it commutes with a subspace projection operator, which is advantageous in model reduction applications. Cubic polynomials are also very easy to differentiate analytically, which is important in applications that need derivatives of the stiffness matrix, such as those involving optimization and control. Iterative solvers outperform direct solvers for large meshes where direct solvers struggle with fitting the factorization in memory, and in applications where the material is soft or the timestep is small.

Limitations and future work: Our simulator is currently limited to 3D solids, tetrahedral and cube meshes, and isotropic materials. It does not incorporate collision detection. One can, however, use any collision library, compute penalty forces, and apply them as external forces to our simulator. In the future, we plan to support anisotropic materials, higher-order mesh elements, and 3D shells (cloth) and rods. For example, we plan to include an implementation of the Baraff-Witkin cloth simulator [BW98]. We also plan to support deformable collision detection and contact resolution.

Acknowledgments: This research was sponsored in part by the National Science Foundation (CAREER-53-4509-6600) and by the James H. Zumberge Research and Innovation Fund at the University of Southern California.

References

[ABA] ABAQUS: ABAQUS Inc. www.hks.com.

[Bar07] BARBIČ J.: *Real-time Reduced Large-Deformation Models and Distributed Contact for Computer Graphics and Haptics*. PhD thesis, Carnegie Mellon University, Aug. 2007.

[Bar12] BARBIČ J.: *Exact Corotational Linear FEM Stiffness Matrix*. Tech. rep., University of Southern California, 2012.

[BJ05] BARBIČ J., JAMES D. L.: Real-time subspace integration for St. Venant-Kirchhoff deformable models. *ACM Trans. on Graphics 24*, 3 (2005), 982–990.

[Bow09] BOWER A.: *Applied Mechanics of Solids*. CRC Press, 2009.

[BW98] BARAFF D., WITKIN A. P.: Large Steps in Cloth Simulation. In *Proc. of ACM SIGGRAPH 98* (1998), pp. 43–54.

[BW01] BARAFF D., WITKIN A.: Physically based modelling, ACM SIGGRAPH Course Notes, 2001.

[BW08] BONET J., WOOD R. D.: *Nonlinear Continuum Mechanics for Finite Element Analysis, 2nd Ed.* Cambridge University Press, 2008.

[BZ11] BARBIČ J., ZHAO Y.: Real-time large-deformation substructuring. *ACM Trans. on Graphics (SIGGRAPH 2011) 30*, 4 (2011), 91:1–91:7.

[CGC*02] CAPELL S., GREEN S., CURLESS B., DUCHAMP T., POPOVIĆ Z.: Interactive skeleton-driven dynamic deformations. *ACM Trans. on Graphics 21*, 3 (2002), 586–593.

[Cou] COUMANS E.: Bullet physics. www.bulletphysics.com.

[CPSS10] CHAO I., PINKALL U., SANAN P., SCHRÖDER P.: A Simple Geometric Model for Elastic Deformations. *ACM Transactions on Graphics 29*, 3 (2010), 38:1–38:6.

[DDCB01] DEBUNNE G., DESBRUN M., CANI M.-P., BARR A. H.: Dynamic Real-Time Deformations Using Space & Time Adaptive Sampling. In *Proc. of ACM SIGGRAPH 2001* (2001), pp. 31–36.

[DW] DHONDT G., WITTIG K.: CALCULIX, A Free Software Three-Dimensional Structural Finite Element Program. www.calculix.de.

[Erl] ERLEBEN K.: Hyper-sim. http://code.google.com/p/hyper-sim.

[Han11] HANG SI: TetGen: A Quality Tetrahedral Mesh Generator and a 3D Delaunay Triangulator, 2011.

[Hav] HAVOK: Havok.com Inc. www.havok.com.

[HLB*06] HUANG J., LIU X., BAO H., GUO B., SHUM H.-Y.: An efficient large deformation method using domain decomposition. *Computers & Graphics 30*, 6 (2006), 927 – 935.

[ISF07] IRVING G., SCHROEDER C., FEDKIW R.: Volume conserving finite element simulations of deformable models. *ACM Trans. on Graphics (SIGGRAPH 2007) 26*, 3 (2007), 13:1–13:6.

[ITF04] IRVING G., TERAN J., FEDKIW R.: Invertible Finite Elements for Robust Simulation of Large Deformation. In *Symp. on Computer Animation (SCA)* (2004), pp. 131–140.

[JBT04] JAMES D. L., BARBIČ J., TWIGG C. D.: Squashing Cubes: Automating Deformable Model Construction for Graphics. In *Proc. of ACM SIGGRAPH Sketches and Applications* (2004).

[Kim] KIM T.: Cubica. http://www.mat.ucsb.edu/ kim/cubica/.

[KMOD09] KHAREVYCH L., MULLEN P., OWHADI H., DESBRUN M.: Numerical coarsening of inhomogeneous elastic materials. *ACM Trans. on Graphics 28*, 3 (2009), 51:1–51:8.

[KTY09] KIKUUWE R., TABUCHI H., YAMAMOTO M.: An edge-based computationally efficient formulation of saint venant-kirchhoff tetrahedral finite elements. *ACM Trans. on Graphics 28*, 1 (2009), 1–13.

[LLJ*11] LEVIN D., LITVEN J., JONES G., SUEDA S., PAI D.: Eulerian solid simulation with contact. *ACM Trans. on Graphics (SIGGRAPH 2011) 30*, 4 (2011).

[MG04] MÜLLER M., GROSS M.: Interactive Virtual Materials. In *Proc. of Graphics Interface 2004* (2004), pp. 239–246.

[MHTG05] MÜLLER M., HEIDELBERGER B., TESCHNER M., GROSS M.: Meshless Deformations Based on Shape Matching. In *Proc. of ACM SIGGRAPH 2005* (Aug 2005), pp. 471–478.

[MTG04] MÜLLER M., TESCHNER M., GROSS M.: Physically-Based Simulation of Objects Represented by Surface Meshes. In *Proc. of Comp. Graphics Int. (CGI)* (2004), pp. 26–33.

[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. on Graphics (SIGGRAPH 2011) 30*, 4 (2011).

[NKJF09] NESME M., KRY P. G., JEŘÁBKOVÁ L., FAURE F.: Preserving topology and elasticity for embedded deformable models. *ACM Trans. on Graphics 28*, 3 (2009), 52:1–52:9.

[NMK*06] NEALEN A., MÜLLER M., KEISER R., BOXERMAN E., CARLSON M.: Physically based deformable models in computer graphics. *Computer Graphics Forum 25*, 4 (2006), 809–836.

[OH99] O'BRIEN J., HODGINS J.: Graphical Modeling and Animation of Brittle Fracture. In *Proc. of ACM SIGGRAPH 99* (1999), pp. 111–120.

[Ope] OPENTISSUE:. Opensource Project, Generic algorithms and data structures for rapid development of interactive modeling and simulation. www.opentissue.org.

[PAR] PARDISO: PARALLEL DIRECT SPARSE SOLVER INTERFACE: Pardiso project, www.pardiso-project.org and Intel MKL, software.intel.com/en-us/articles/intel-mkl.

[PDA01] PICINBONO G., DELINGETTE H., AYACHE N.: Nonlinear and anisotropic elastic soft tissue models for medical simulation. In *IEEE Int. Conf. on Robotics and Automation* (2001).

[Phya] PHYSBAM: Stanford University. physbam.stanford.edu.

[Phyb] PHYSX: Nvidia. www.nvidia.com/object/physx_new.html.

[PO09] PARKER E. G., O'BRIEN J. F.: Real-time deformation and fracture in a game environment. In *Symp. on Computer Animation (SCA)* (2009), pp. 156–166.

[SD06] STERN A., DESBRUN M.: Discrete geometric mechanics for variational time integrators. In *ACM SIGGRAPH 2006 Courses* (2006), pp. 75–80.

[Sha90] SHABANA A. A.: *Theory of Vibration, Volume II: Discrete and Continuous Systems*. Springer–Verlag, 1990.

[She94] SHEWCHUK J. R.: An introduction to the conjugate gradient method without the agonizing pain, 1994.

[SOF] SOFA: SIMULATION OPEN FRAMEWORK ARCHITECTURE: INRIA. www.sofa-framework.com.

[SPO] SPOOLES: SPARSE OBJECT ORIENTED LINEAR EQUATIONS SOLVER: Boeing Phantom Works. www.netlib.org/linalg/spooles/spooles.2.2.html.

[TSIF05] TERAN J., SIFAKIS E., IRVING G., FEDKIW R.: Robust Quasistatic Finite Elements and Flesh Simulation. In *Symp. on Computer Animation (SCA)* (2005), pp. 181–190.

[Woo90] WOOD W. L.: *Practical time-stepping schemes*. Clarendon Press, 1990.

[Wri02] WRIGGERS P.: *Computational Contact Mechanics*. John Wiley & Sons, Ltd., 2002.