Parallel Iterative Solvers for Real-time Elastic Deformations

Course Notes

MARCO FRATARCANGELI, Chalmers University of Technology, Sweden
HUAMIN WANG, The Ohio State University, USA
YIN YANG, University of New Mexico, USA

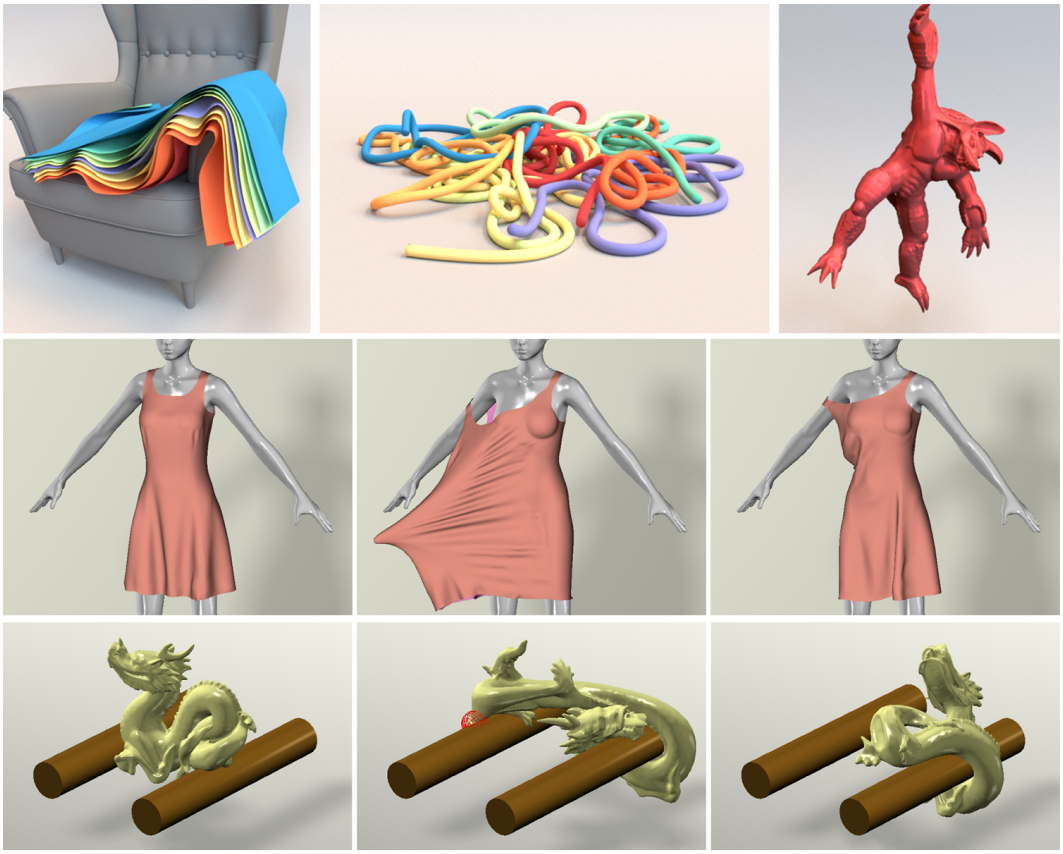


Fig. 1. Real-time animation of deformable bodies modeled using hundreds of thousands of constraints.

Authors' addresses: Marco Fratarcangeli, Chalmers University of Technology, Gothenburg, Sweden, marcof@chalmers.se; Huamin Wang, The Ohio State University, Columbus, USA, whmin@cse.ohio-state.edu; Yin Yang, University of New Mexico, Albuquerque, New Mexico, USA, yangy@unm.edu.

SA '18 Courses, December 04-07, 2018, Tokyo, Japan
ACM ISBN 978-1-4503-6026-5/18/12.
https://doi.org/10.1145/3277644.3277779

Abstract. Physics-based animation of elastic materials allows to simulate dynamic deformable objects such as fabrics, human tissue, hair, etc. Due to their complex inner mechanical behaviour, it is difficult to replicate their motions interactively and accurately at the same time. This course introduces students and practitioners to several parallel iterative techniques to tackle this problem and achieve elastic deformations in real-time. We focus on techniques for applications such as video games and interactive design, with *fixed and small hard time budgets* available for physically-based animation, and where responsiveness and stability are often more important than accuracy, as long as the results are believable. The course focuses on solvers able to fully exploit the computational capabilities of modern GPU architectures, effectively solving systems of hundreds of thousands of nonlinear equations in a matter of few milliseconds. The course introduces the basic concepts concerning physics-based elastic objects, and provide an overview of the different types of numerical solvers available in the literature. Then, we show how some variants of traditional solvers can address real-time animation and assess them in terms of accuracy, robustness and performance. Practical examples are provided throughout the course, in particular how to apply the depicted solvers to Projective Dynamics and Position-based Dynamics, two recent and popular physics models for elastic materials.

CCS Concepts: • **Computing methodologies → Massively parallel algorithms**; **Physical simulation**;

Additional Key Words and Phrases: Nonlinear Optimization, Hyperelasticity, Jacobi Method, Chebyshev Semi-iterative Method, Multi-color Gauss-Seidel Method, Projective Dynamics, Position Based Dynamics, Parallel Computing, GPU Acceleration

ACM Reference Format:
Marco Fratarcangeli, Huamin Wang, and Yin Yang. 2018. Parallel Iterative Solvers for Real-time Elastic Deformations: Course Notes. In *Proceedings of SA '18 Courses.* 1, 1 (December 2018), 45 pages. https://doi.org/10.1145/3277644.3277779

Contents

1 INTRODUCTION

1.1 Outline and Motivation

Many applications in Computer Graphics demand increasingly sophisticated models for animating deformable materials. While these topics have been studied extensively, the simulation of complex systems at interactive rates is still an open problem. This course presents some of the latest techniques in the literature to accelerate physics-based animation of soft bodies in the context of real-time applications, such as entertainment (videogames and visual effects) and interactive computational design. We believe that a good elastic body simulation method should satisfy at least the following three requirements:

- **Generality.** A good method should be flexible enough to handle most elastic models, if not all. In particular, it should be able to simulate hyperelastic models, which use energy density functions to describe highly nonlinear force-displacement relationships.
- **Correctness and Scalability.** Given sufficient computational resources, a good method should correctly simulate the behavior of a specified elastic model within a given time budget. In other words, the method is not just a temporary one for producing visually appealing animations. Instead, it can be more accurate for serious applications, once hardware becomes more powerful.
- **Efficiency.** A good method should be fast enough for real-time applications. It should also be compatible with parallelization, so that it can benefit significantly from the use of graphics hardware and computer clusters.

Concerning the latter point, in order to guarantee interactivity at 60 frames per second, the total amount of time available to update the scene is 16 ms, and only a fraction of this time can be devoted for advancing the dynamics of the objects. Often, the available time slice for physically-based animation within a software framework, such as a game engine, is as little as 5 ms.

In most of the simulators proposed by the Computer Graphics community, the main computational cost lies in the numerical solving of a big, usually sparse, linear system

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{1}$$

where $\mathbf{x}$ represents the unknown geometric configuration of the objects (e.g., the vertex coordinates), $\mathbf{A}$ embeds the dynamic properties of the system, and $\mathbf{b}$ represents some known boundary conditions. In general, $\mathbf{x}$ may involve hundreds of thousands of unknown variables, so the exact solution $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ is too costly to compute in real-time. Hence, *linear iterative solvers* have been employed effectively to provide fast, but approximated, solutions of the equations governing the system [Saad 2003]. The accuracy of the solution of linear iterative solvers depends on the number of iterations; still, only a small number of iterations can be accommodated in the available time budget. Some of the most popular iterative solvers are the *Jacobi*- and the *Gauss-Seidel*-based solvers. These do not need computationally expensive inner products unlike conjugate gradient and GMRES, or direct methods like Cholesky decompositions. Furthermore, the Jacobi-based approaches are simple to parallelize, and thus suitable for modern GPUs or multi-core processors. On the other hand, it is well-known that the convergence speed of the Gauss-Seidel method is much faster than Jacobi under an equal number of iterations. The Gauss-Seidel method is, however, an inherently serial algorithm, and thus it cannot be implemented easily on parallel architectures. There exists a significant body of literature on how to parallelize Gauss-Seidel, but in general these methods involve complex synchronization mechanisms and expensive communications between threads, which undermine their application in the context of interactive animation on commodity GPUs.

In the following, we focus on three different acceleration techniques to speed-up the performance of popular iterative solvers by several orders of magnitude, and achieve results qualitatively comparable with direct methods, which are more accurate but also more computational expensive.

In particular, we describe

- the fast computation of Chebishev polynomials to speed-up the convergence speed of parallel Jacobi solver on the GPU and serial Gauss-Seidel on the CPU (Sec. 2);
- how to use the accelerated Jacobi method to accelerate the computation of descent methods and solve non-linear problems interactively (Sec. 3);
- how to use a random coloring algorithm to partition the problem in independent chunks and solve it in parallel using the Gauss-Seidel method (Sec. 4).

Such techniques have been published in SIGGRAPH Asia in 2015 and 2016 [Fratarcangeli et al. 2016; Wang 2015; Wang and Yang 2016]. For each method, we provide both the theoretical framework and implementation details, in particular how to apply them to *Projective* and *Position-based dynamics*, two recently introduced models which are spreading in Computational Animation as well as many other scientific different fields related to numerical computing. For the sake of completeness, we describe such two models in Sec. 1.3, just after a brief primer about iterative linear systems in Sec. 1.2.

1.2 Iterative Linear Solvers

In this section, we will review some background knowledge about iterative solvers for linear systems, in particular the Jacobi and Gauss-Seidel methods. More advanced techniques will be depicted in the next sections, including the Chebyshev semi-iterative method (Sec. 2.1), and descent methods (Sec. 3.1). A linear system can be formulated as: $\mathbf{Ax} = \mathbf{b}$, in which $\mathbf{A} \in \mathbb{R}^{N \times N}$ is a matrix, $\mathbf{b} \in \mathbb{R}^N$ is a given vector, and $\mathbf{x} \in \mathbb{R}^N$ is the vector of N unknowns that needs to be found. When $\mathbf{A}$ is large and sparse, iterative methods are often favored over direct methods, to avoid matrices from being filled by new nonzeros during the solving process. Based on the splitting idea: $\mathbf{A} = \mathbf{B} - \mathbf{C}$, standard iterative methods, such as Jacobi and Gauss-Seidel, have the form:

$$\mathbf{x}^{(k+1)} = \mathbf{B}^{-1}\left(\mathbf{C}\mathbf{x}^{(k)} + \mathbf{b}\right). \tag{2}$$

It is straightforward to see that when these methods converge, they provide the solution to the linear system: $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} = \mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$, since $\mathbf{Bx} = \mathbf{Cx} + \mathbf{b}$. If we split $\mathbf{B}^{-1}\mathbf{b}$ into $\mathbf{B}^{-1}\mathbf{Ax} = \mathbf{x} - \mathbf{B}^{-1}\mathbf{Cx}$, we can convert Equation 2 into:

$$\mathbf{e}^{(k+1)} = \mathbf{x}^{(k+1)} - \mathbf{x} = \mathbf{B}^{-1}\mathbf{C}\left(\mathbf{x}^{(k)} - \mathbf{x}\right) = \mathbf{B}^{-1}\mathbf{C}\mathbf{e}^{(k)}, \tag{3}$$

in which $\mathbf{e}^{(k)}$ is the error vector at the k-th iteration. Let the eigenvalue decomposition of $\mathbf{B}^{-1}\mathbf{C}$ be $\mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^{-1}$, where $\mathbf{\Lambda}$ is the eigenvalue matrix. We can reformulate the error vector at the k-th iteration as:

$$\mathbf{e}^{(k)} = \left(\mathbf{B}^{-1}\mathbf{C}\right)^k \mathbf{e}^{(0)} = \mathbf{Q}\mathbf{\Lambda}^k\mathbf{Q}^{-1}\mathbf{e}^{(0)}. \tag{4}$$

This means that these iterative methods converge linearly and their convergence rates depend on the largest eigenvalue magnitude, known as the *spectral radius*: $\rho(\mathbf{B}^{-1}\mathbf{C})$. To ensure the convergence, we must have: $\rho(\mathbf{B}^{-1}\mathbf{C}) < 1$.

By setting $\mathbf{B} = \mathbf{D}$, with $\mathbf{D}$ formed by the diagonal entries of $\mathbf{A}$, and $\mathbf{C} = \mathbf{U} + \mathbf{L}$, with $\mathbf{U}$ and $\mathbf{L}$ two strictly upper and lower triangular parts respectively, then $\mathbf{A} = \mathbf{B} - \mathbf{C} = \mathbf{D} - \mathbf{U} - \mathbf{L}$, and we can define the following linear methods.

1.2.1 Jacobi Solver. Equation 2 can be rearranged as:

$$\mathbf{x}^{(k+1)} = \mathbf{D}^{-1}\left(\mathbf{U}+\mathbf{L}\right)\mathbf{x}^{(k)} + \mathbf{D}^{-1}\mathbf{b}. \tag{5}$$

This is equivalent to writing the equation corresponding to a single row i of $\mathbf{A}$ as:

$$x_i^{(k+1)} = \frac{1}{d_i}\left(b_i - \sum_{j=1}^{i-1} l_{ij}x_j^{(k)} - \sum_{j=i+1}^{N} u_{ij}x_j^{(k)}\right), \qquad i = 1, 2, ..., N \tag{6}$$

where d_i are the entries of the diagonal matrix $\mathbf{D}$, and l_{ij} and u_{ij} are the entries of $\mathbf{U}$ and $\mathbf{L}$, respectively. Note that for any i, all the values on the right side of the equation are known. This means that all the entries in $\mathbf{x}$ can be computed in one single parallel step, making the Jacobi solver particularly suited for highly-parallel computational frameworks such as the GPU. Unfortunately, in general this process must be repeated for many iterations in order to reach a solution with sufficiently low error. So, even though a single iteration of the Jacobi method can be computed very fast, the required number of iterations to reach a solution is too big, in particular when the number of unknowns is high, like in high-resolution deformable objects. Thus, the method becomes unsuitable for real-time applications. In Sec. 2, we introduce a technique to accelerate the convergence speed of the Jacobi method without compromising its low computational cost per iteration.

1.2.2 Gauss-Seidel Solver. Another way to rearrange Equation 2 is:

$$\mathbf{x}^{(k+1)} = \left(\mathbf{D}-\mathbf{L}\right)^{-1}\mathbf{U}\mathbf{x}^{(k)} + \left(\mathbf{D}-\mathbf{L}\right)^{-1}\mathbf{b}. \tag{7}$$

By using the same notation of the last section, this is equivalent to writing the equations corresponding to the single rows of $\mathbf{A}$ as:

$$x_i^{(k+1)} = \frac{1}{d_i}\left(b_i - \sum_{j=1}^{i-1} l_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{N} u_{ij}x_j^{(k)}\right), \qquad i = 1, 2, ..., N \tag{8}$$

The Gauss-Seidel method has a higher convergence speed w.r.t. Jacobi. In this case, however, the values of $\mathbf{x}_i^{(k+1)}$ cannot be computed in parallel, but they must be found sequentially using forward substitution. Thus, the Gauss-Seidel method can not be implemented on the GPU as it is. In Sec. 4, we show how to divide the set of unknowns $x_i^{(k+1)}$ in independent sets, and then solve effectively each one of these sets in parallel making the convergence speed of the Gauss-Seidel method *de-facto* available for parallel implementations on the GPUs.

1.3 Projective and Position-based Dynamics as a Nonlinear Optimization Problem

Position-based dynamics is a popular technique for simulating deformable objects, and it has been widely used in many high-end physics engines, such as PhysX, Havok Cloth, and Maya nCloth, thanks to its simplicity. The basic idea behind position-based dynamics is to update vertex positions by enforcing position-based constraints iteratively, rather than by using elastic forces. When a vertex is involved in multiple constraints, its position can be updated either sequentially, known as the *Gauss-Seidel* way, or simultaneously through averaging, known as the *Jacobi* way. To implement position-based dynamics on GPU, the Jacobi way is often preferred so that the constraints can be processed in parallel. Position-based dynamics uses the number of iterations to control how strictly the constraints are enforced and how stiff an object behaves, so it is free of numerical instability. It is difficult, however, to formulate the relationship between the mechanical properties of an object and the number of iterations. In fact, position-based dynamics can cause the stiffness behavior of an object to be affected the mesh resolution, since its convergence rate drops as the mesh

resolution increases. To speed up the convergence, researchers proposed to enforce the constraints in a multi-resolution fashion [Müller 2008; Wang et al. 2010]. But building a mesh hierarchy is not simple and the result can be even more mesh-dependent.

A new constraint-based simulation technique, known as *projective dynamics* [Bouaziz et al. 2014; Liu et al. 2013], emerged recently. Different from position-based dynamics, projective dynamics tries to solve implicit time integration of a dynamical system exactly, formulated under the variational form. Specifically, it iteratively runs two steps: a local step that projects each constraint into an acceptable state, and a global step that transfers these states to vertex positions. While projective dynamics can be considered as a generalized version of position-based dynamics, its converged result is physically plausible and controllable by stiffness variables. Since the linear system involved in the global step has a constant matrix, the original implementation of projective dynamics pre-factors the matrix and solve the global system directly by forward and backward substitutions. Previous research showed that doing this can achieve visually acceptable results even within a few iterations. The catch is that a direct solver cannot be easily accelerated by GPU. So the whole method becomes less efficient, when more iterations are needed to reduce errors and artifacts of a fast deforming object.

1.3.1 Projective Dynamics Formulation. Given a 3D dynamical system with N vertices, we can simulate its movement from the t-th time instant to the t+1-th time instant by implicit time integration [Baraff and Witkin 1998]:

$$\mathbf{q}_{t+1} = \mathbf{q}_t + h\mathbf{v}_{t+1}, \quad \mathbf{v}_{t+1} = \mathbf{v}_t + h\mathbf{M}^{-1}\mathbf{f}_{t+1}, \tag{9}$$

where $\mathbf{q} \in \mathbb{R}^{3N}$ and $\mathbf{v} \in \mathbb{R}^{3N}$ are stacked position and velocity vectors, $\mathbf{M} \in \mathbb{R}^{3N\times 3N}$ is the mass matrix, h is the time step, and $\mathbf{f} \in \mathbb{R}^{3N}$ is the total force. Assuming that $\mathbf{f}_{t+1} = \mathbf{f}_{\text{int}}(\mathbf{q}_{t+1}) + \mathbf{f}_{\text{ext}}$, in which the internal elastic force $\mathbf{f}_{\text{int}}$ is a function of $\mathbf{q}$ and the external force $\mathbf{f}_{\text{ext}}$ is constant, we reformulate Equation 9 into a nonlinear system:

$$\mathbf{M}\left(\mathbf{q}_{t+1} - \mathbf{q}_t - h\mathbf{v}_t\right) = h^2\left(\mathbf{f}_{\text{int}}(\mathbf{q}_{t+1}) + \mathbf{f}_{\text{ext}}\right). \tag{10}$$

One solution to Equation 10 is to use the Newton's method:

$$\mathbf{M}\left(\mathbf{q}^{(k+1)} - \mathbf{q}_t - h\mathbf{v}_t\right) = h^2\left(\mathbf{f}_{\text{int}}(\mathbf{q}^{(k)}) + \mathbf{K}(\mathbf{q}^{(k)})(\mathbf{q}^{(k+1)} - \mathbf{q}^{(k)}) + \mathbf{f}_{\text{ext}}\right), \tag{11}$$

in which $\mathbf{q}^{(0)} = \mathbf{q}_t$ and $\mathbf{K}(\mathbf{q}^{(k)}) = \partial\mathbf{f}_{\text{int}}/\partial\mathbf{x}$ is the stiffness matrix evaluated at $\mathbf{q}^{(k)}$. Although Newton's method converges fast, its iterations are computationally expensive and its overall performance is often far from satisfactory. If we just use a single iteration as proposed by Baraff and Witkin [1998], the result will not be accurate, but at least visually plausible for some applications.

A different way of handling implicit time integration is to convert it into an energy minimization problem:

$$\mathbf{q}_{t+1} = \arg\min_{\mathbf{q}} \epsilon(\mathbf{q}) = \arg\min_{\mathbf{q}} \frac{1}{2h^2}||\mathbf{M}^{\frac{1}{2}}(\mathbf{q} - \mathbf{s}_t)||^2 + E(\mathbf{q}), \tag{12}$$

in which $E(\mathbf{q})$ is the internal potential energy at $\mathbf{q}$, and $\mathbf{s}_t = \mathbf{q}_t + h\mathbf{v}_t + h^2\mathbf{M}^{-1}\mathbf{f}_{\text{ext}}$ is the expected position vector without internal forces. It is straightforward to see that the critical point of Equation 12 is identical to Equation 10. In other words, solving Equation 12 is equivalent to solving Equation 10. Projective dynamics [Bouaziz et al. 2014; Liu et al. 2013] assumes that the internal energy is the sum of the constraint energies and each constraint energy has a quadratic form:

$$\epsilon(\mathbf{q}) = \frac{1}{2h^2}||\mathbf{M}^{\frac{1}{2}}(\mathbf{q} - \mathbf{s}_t)||^2 + \sum\nolimits_c \frac{w_c}{2}\left\|\mathbf{A}_c\mathbf{q} - \hat{\mathbf{A}}_c\mathbf{p}_c\right\|^2, \tag{13}$$

where w_c is a positive weight of constraint c, $\mathbf{p}_c$ is the projection of $\mathbf{q}$ into the energy-free space defined by c, and $\mathbf{A}_c$ and $\hat{\mathbf{A}}_c$ are two constant matrices specifying the relationship between $\mathbf{q}$ and $\mathbf{p}_c$. For example, if c is a spring, $\mathbf{A}_c\mathbf{q}$ and $\hat{\mathbf{A}}_c\mathbf{p}_c$ give the displacement vectors of the two vertices, and the internal energy is equivalent to the spring potential energy. Based on this model, projective dynamics iteratively minimizes $\epsilon(\mathbf{q})$ in two steps: in a local step, it projects the current $\mathbf{q}$ into $\mathbf{p}_c$ for every constraint c; in a global step, it treats $\mathbf{p}_c$ as constant and finds the next $\mathbf{q}$ that minimizes $\epsilon(\mathbf{q})$. Liu and colleagues [2013] demonstrates that projective dynamics using spring constraints is equivalent to an implicit mass-spring system.

1.3.2 The Global Solve. An important question regarding projective dynamics is how to solve the global step. Liu and colleagues [2013] and Bouaziz and collaborators [2014] found that the linear system resulted from $\nabla\epsilon(\mathbf{q}) = 0$ has a constant matrix $\mathbf{A}$:

$$\mathbf{A}\mathbf{q} = \left(\frac{\mathbf{M}}{h^2} + \sum_c w_c \mathbf{A}_c^\mathsf{T}\mathbf{A}_c\right)\mathbf{q} = \frac{\mathbf{M}}{h^2}\mathbf{s}_t + \sum_c w_c \mathbf{A}_c^\mathsf{T}\hat{\mathbf{A}}_c\mathbf{p}_c, \tag{14}$$

which means $\mathbf{A}$ can be pre-factored for fast direct solve in every iteration. If each time step contains only a small number of iterations, we can solve projective dynamics efficiently in real time.

1.3.3 A Simple Example. Consider the mass-spring system in Fig. 2, composed by $m = 3$ particles connected by $s = 2$ springs.

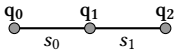


Fig. 2

We use Hook's law

$$E_c(\mathbf{q}_i, \mathbf{q}_j) = k_c\left(\|\mathbf{q}_i - \mathbf{q}_j\| - r_c\right),$$

to describe the energy of a spring connecting particles $\mathbf{q}_i$ and $\mathbf{q}_j$, with rest length r_c, and stiffness k_c. Following the notation in Sec. 1.3.2, each matrix $\mathbf{A}_c \in \mathbb{R}^{mx1}$ corresponds to a spring c, and an entry a_{i0} is set to 1 if the particle $\mathbf{q}_i$ is connected to the spring c, 0 otherwise. Each *selector* matrix $\hat{\mathbf{A}}_c \in \mathbb{R}^{sx1}$ also corresponds to a spring and they are composed by all 0s, except for the c-th entry which is set to 1:

$$\mathbf{A}_0 = \begin{pmatrix}1\\1\\0\end{pmatrix}, \qquad \mathbf{A}_1 = \begin{pmatrix}0\\1\\1\end{pmatrix}, \qquad \hat{\mathbf{A}}_0 = \begin{pmatrix}1\\0\end{pmatrix}, \qquad \hat{\mathbf{A}}_1 = \begin{pmatrix}0\\1\end{pmatrix}$$

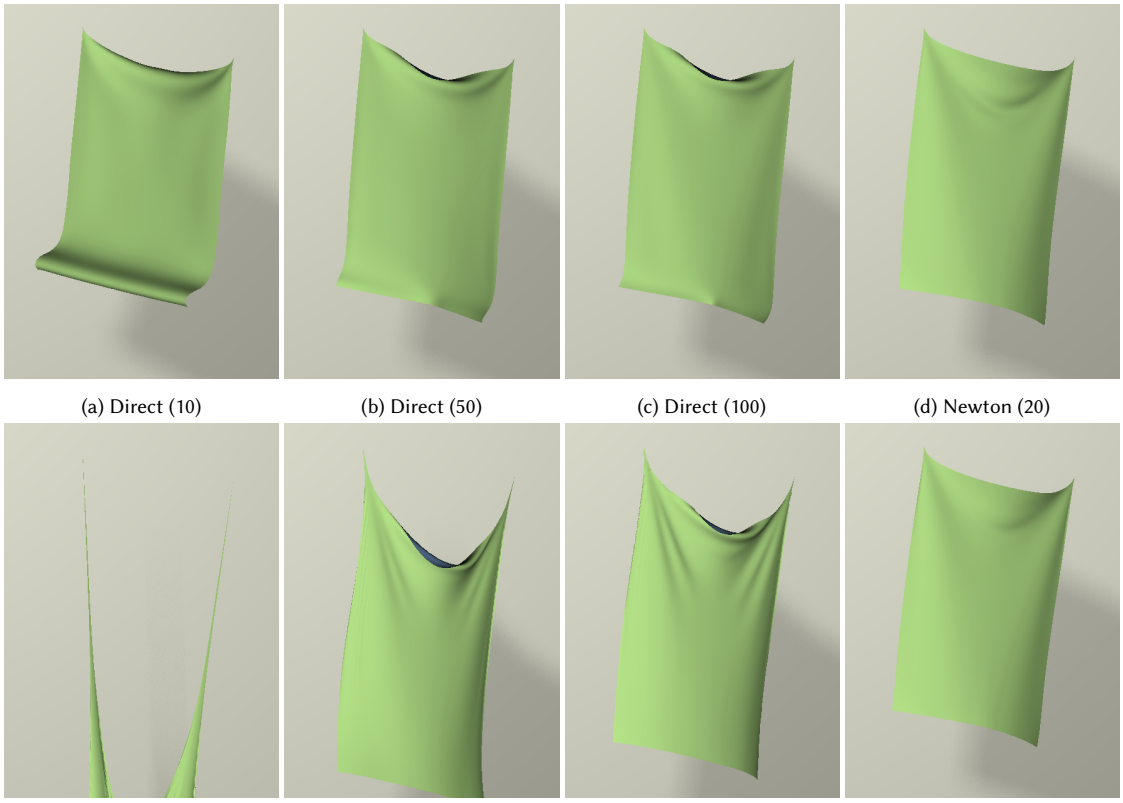
In this case, we can minimize the energy of the system (Equation 12) by first solving the local step *in parallel*:

$$\mathbf{p}_c = \frac{\mathbf{q}_i - \mathbf{q}_j}{\|\mathbf{q}_i - \mathbf{q}_j\|} r_c, \qquad c = 1, \ldots, s$$

and then solving the global step (Equation 14) by setting $w_c = k_c$. The linear system to be solved is:

$$\begin{pmatrix} \frac{m_0}{h^2} + k_0 & k_0 & 0 \\ k_0 & \frac{m_1}{h^2} + k_0 + k_1 & k_1 \\ 0 & k_1 & \frac{m_0}{h^2} + k_1 \end{pmatrix} \begin{pmatrix}\mathbf{q}_0\\\mathbf{q}_1\\\mathbf{q}_2\end{pmatrix} = \begin{pmatrix} k_0\mathbf{p_0} + \frac{m_0}{h^2}\mathbf{s}_0 \\ k_0\mathbf{p_0} + k_1\mathbf{p_1} + \frac{m_1}{h^2}\mathbf{s}_1 \\ k_1\mathbf{p_1} + \frac{m_2}{h^2}\mathbf{s}_2 \end{pmatrix}$$

where m_i is the mass of particle $\mathbf{q}_i$. By repeating this local/global step alternation process, $\mathbf{q}$ converges rapidly to a low-energy configuration, and it provides a solution to Equation 12. We want to stress once more the main advantage



(a) Direct (10) (b) Direct (50) (c) Direct (100) (d) Newton (20)

(e) Jacobi (400) (f) Jacobi (4000) (g) OR (4000) (h) Jacobi+Chebyshev (400)

Fig. 3. Projective dynamics results produced by using different solvers. The numbers in the brackets are the numbers of iterations. The direct solver can still cause curly edge artifacts after 100 iterations, as (c) shows. Meanwhile, the Jacobi solver converges slowly, even when it is accelerated by over-relaxations in (g). In contrast, the result generated by our Jacobi+Chebyshev method after 400 iterations in (h) is highly similar to the ground truth, which is simulated by the Newton's method shown in (d).

of projective dynamics: as opposed to the descent methods, *the matrix on the left side of the linear system above does not change over the local/global iterations*. This feature makes it possible to speed up the computation of the linear system and achieve real-time performance as depicted in Sections 2, 3 and 4.

1.3.4 Comments on Projective Dynamics. A few iterations of the local/global approach are often insufficient for projective dynamics to stabilize its results, when an object undergoes large and fast deformation. This issue is especially noticeable in mass-spring systems, where the displacement vectors change dramatically within one time step. For example, Figure 3a shows that 10 iterations are insufficient to remove the curly edge artifact near the bottom of a swinging tablecloth, which does not exist in the ground truth provided by the Newton's method in Figure 3d. This artifact becomes less noticeable, but still exists after we increase the number of iterations from 10 to 100, as Figure 3c shows. Unfortunately, we cannot afford running the direct solve too many times, since it is still relatively expensive and it cannot be easily accelerated by GPU.

From a different perspective, we may consider the cause of the aforementioned issue to be the unneeded accuracy of the direct solver. Without rounding errors, the direct solver finds the exact solution to the linear system in Equation 14. But such accuracy is unnecessary and the computational cost is wasted, once the local step modifies the linear system in the next iteration. One idea is to replace the direct solver by one Jacobi iteration, which is cheap and compatible with GPU acceleration. But projective dynamics will converge slowly as shown in Figure 3e and 3f. The over-relaxation method cannot make much difference, as Figure 3g shows. So to achieve the same accuracy, the Jacobi solver often ends up spending even more computational time. Figure 3h shows that, by accelerating Jacobi with the Chebyshev semi-iterative approach, as explained in Sec. 2, results have similar accuracy to Newton, but are computed much faster.

1.3.5 Position-based Dynamics. Liu and colleagues [2013] and Bouaziz and collaborators [2014] pointed out that the original implementation of position-based dynamics [Müller et al. 2007] can be considered as a simplified version of projective dynamics, by setting $\mathbf{A}_c^\mathsf{T}\mathbf{A}_c = \mathbf{A}_c^\mathsf{T}\hat{\mathbf{A}}_c = \mathbf{I}$, ignoring the first term in $\epsilon(\mathbf{q})$, and using one Gauss-Seidel iteration to solve the global step. If we replace the Gauss-Seidel iteration by a Jacobi iteration, the local step calculates the expected positions of a vertex according to the constraints, and the global step uses their weighted average to update the vertex position. This is basically equivalent to the Jacobi implementation of position-based dynamics, proposed by Macklin and colleagues [2014]. In general, position-based dynamics converges slower than projective dynamics, even when both of them use Jacobi iterations. This is because projective dynamics can use non-diagonal entries in $\mathbf{A}_c$ and $\hat{\mathbf{A}}_c$ to propagate the influence of the constraints faster.

2 ACCELERATING PROJECTIVE DYNAMICS WITH THE CHEBYSHEV SEMI-ITERATIVE METHOD

In this section, we demonstrate how the Chebyshev semi-iterative approach can be applied to accelerate both projective dynamics and position-based dynamics. The contents of this section include:

- **Analysis.** We noticed that the convergence of projective dynamics is highly similar to that of an iterative method solving a linear system, even when projective dynamics uses a direct solver. This is true to position-based dynamics as well, which can be considered as using a trivial global step. Based on these observations, we propose to estimate the "spectral radius" of projective or position-based dynamics from its convergence rate.
- **Implementation.** We show that the Chebyshev approach is easy to implement and compatible with GPU acceleration. Given an existing projective or position-based dynamics simulator, the approach can be implemented in less than five minutes! The Jacobi+Chebyshev combination further allows positional and contact constraints to be easily handled in each iteration. In contrast, the previous implementation [Bouaziz et al. 2014] requires updating the linear system.
- **Evaluation.** We tested our simulator using both triangular and tetrahedral meshes. Our experiment shows that the approach can accelerate projective dynamics by at least one order of magnitude, when the global step uses the direct solver, the Jacobi solver, or even the Gauss-Seidel solver. The Chebyshev approach can effectively accelerate position-based dynamics as well, especially for triangular meshes.

In summary, we present a simple, fast, and effective approach for accelerating projective and position-based dynamics, based on the Chebyshev semi-iterative method. This approach requires a small memory cost and it can handle large time steps and deformations. It is highly compatible with GPU acceleration and it can work together with other acceleration approaches as well, such as multi-resolution techniques.

This method was published in SIGGRAPH Asia 2015 [Wang 2015].

2.1 The Chebyshev Semi-Iterative Method

Given the results produced by the iterative formula in Equation 2: $\mathbf{x}^{(0)}$, $\mathbf{x}^{(1)}$, ..., $\mathbf{x}^{(k)}$, we would like to obtain a better result from their linear combinations, which has the following form:

$$\mathbf{y}^{(k)} = \sum_{j=0}^{k} v_{j,k}\mathbf{x}^{(j)}, \tag{15}$$

in which $v_{j,k}$ are the blending coefficients to be determined. If the results are good already: $\mathbf{x}^{(0)} = \mathbf{x}^{(1)} = \ldots = \mathbf{x}^{(k)} = \mathbf{x}$, we must have $\mathbf{y}^{(k)} = \mathbf{x}$. So we require the following constraint:

$$\sum_{j=0}^{k} v_{j,k} = 1. \tag{16}$$

The question is how to reduce the error of $\mathbf{y}^{(k)}$. Using Equation 16 and 3, we can formulate the error $\mathbf{y}^{(k)} - \mathbf{x}$ into:

$$\sum_{j=0}^{k} v_{j,k}\left(\mathbf{x}^{(j)} - \mathbf{x}\right) = \sum_{j=0}^{k} v_{j,k}\left(\mathbf{B}^{-1}\mathbf{C}\right)^{j}\mathbf{e}^{(0)} = p_k(\mathbf{B}^{-1}\mathbf{C})\mathbf{e}^{(0)}, \tag{17}$$

in which $p_k(x) = \sum_{j=0}^{k} v_{j,k}x^j$ is a polynomial function. So to reduce the error, we must reduce $\left\|p_k(\mathbf{B}^{-1}\mathbf{C})\right\|_2 = \max_{\lambda_i} |p_k(\lambda_i)|$, in which λ_i can be any eigenvalue of $\mathbf{B}^{-1}\mathbf{C}$. Suppose that all of the eigenvalues are real. If we know all of the eigenvalues

and if k is sufficiently large, we can construct the polynomial function in a way that $p_k(\lambda_i) = 0$ for any λ_i. Unfortunately, it is difficult to know the eigenvalues, when the linear system is large and varying. Instead, if we know the spectral radius ρ such that $-1 < -\rho \leq \lambda_n \leq \ldots \leq \lambda_1 \leq \rho < 1$, we can ask $p_k(x)$ to be minimized for all $x \in [-\rho, \rho]$:

$$p_k(x) = \arg\min \left\{ \max_{-\rho \leq x \leq \rho} |p_k(x)| \right\}. \tag{18}$$

The unique solution to Equation 18 is given by:

$$p_k(x) = \frac{C_k(x/\rho)}{C_k(1/\rho)}, \tag{19}$$

in which $C_k(x)$ is the Chebyshev polynomial with the recurrence relation: $C_{k+1}(x) = 2xC_k(x) - C_{k-1}(x)$, with $C_0(x) = 1$ and $C_1(x) = x$. It is trivial to see that $p_k(1) = 1$, satisfying Equation 16. For any $x \in [-1, 1]$, $|C_k(x)| \leq 1$, but for any $x \notin [-1, 1]$, $|C_k(x)|$ grows rapidly when $k \to \infty$. So $p_k(x)$ diminishes quickly for any $x \in [-\rho, \rho]$, when $k \to \infty$.

To reduce the computational and memory cost, we can avoid calculating $\mathbf{y}^{(k)}$ by its definition in Equation 15. Instead, we use Equation 19 to formulate the recurrence relation of $p_k(x)$ as:

$$\begin{aligned} p_{k+1}(x)C_{k+1}\left(\tfrac{1}{\rho}\right) &= \tfrac{2x}{\rho}C_k\left(\tfrac{x}{\rho}\right) - C_{k-1}\left(\tfrac{x}{\rho}\right) \\ &= \tfrac{2x}{\rho}p_k(x)C_k\left(\tfrac{1}{\rho}\right) - p_{k-1}(x)\left(\tfrac{2}{\rho}C_k\left(\tfrac{1}{\rho}\right) - C_{k+1}\left(\tfrac{1}{\rho}\right)\right), \end{aligned} \tag{20}$$

which can be reorganized into:

$$C_{k+1}\left(\tfrac{1}{\rho}\right)(p_{k+1}(x) - p_{k-1}(x)) = \tfrac{2C_k\left(\frac{1}{\rho}\right)}{\rho}(xp_k(x) - p_{k-1}(x)). \tag{21}$$

After replacing x by $\mathbf{B}^{-1}\mathbf{C}$ and multiplying both sides of Equation 21 by $\mathbf{e}^{(0)}$, we get:

$$C_{k+1}\left(\tfrac{1}{\rho}\right)\left(\mathbf{y}^{(k+1)} - \mathbf{y}^{(k-1)}\right) = \tfrac{2C_k\left(\frac{1}{\rho}\right)}{\rho}\left(\mathbf{B}^{-1}\mathbf{C}\left(\mathbf{y}^{(k)} - \mathbf{x}\right) - \mathbf{y}^{(k-1)} + \mathbf{x}\right). \tag{22}$$

Using the fact that $-\mathbf{B}^{-1}\mathbf{C}\mathbf{x} + \mathbf{x} = \mathbf{B}^{-1}(\mathbf{B} - \mathbf{C})\mathbf{x} = \mathbf{B}^{-1}\mathbf{b}$, we can now obtain the following update function:

$$\mathbf{y}^{(k+1)} = \omega_{k+1}\left(\mathbf{B}^{-1}\left(\mathbf{C}\mathbf{y}^{(k)} + \mathbf{b}\right) - \mathbf{y}^{(k-1)}\right) + \mathbf{y}^{(k-1)}, \tag{23}$$

where

$$\omega_{k+1} = \frac{2C_k\left(\frac{1}{\rho}\right)}{\rho C_{k+1}\left(\frac{1}{\rho}\right)}, \quad i \geq 1, \quad \omega_1 = 1. \tag{24}$$

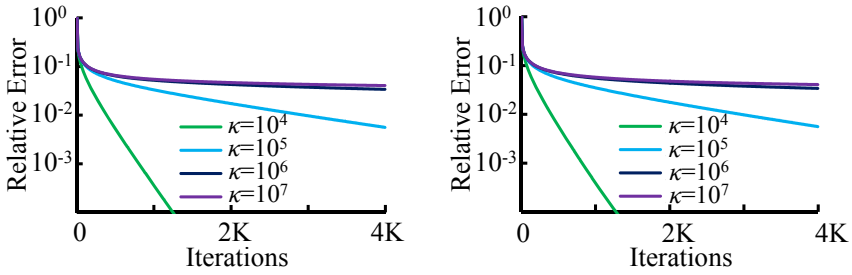
In Equation 23, $\mathbf{B}^{-1}\left(\mathbf{C}\mathbf{y}^{(k)} + \mathbf{b}\right)$ is essentially one iterative solve step described in Equation 2. By definition, we can further get:

$$\omega_{k+1} = \frac{2C_k}{\rho\left(\frac{2}{\rho}C_k - C_{k-1}\right)} = \frac{2C_k}{\rho\left(\frac{2}{\rho}C_k - \frac{\rho\omega_k C_k}{2}\right)} = \frac{4}{4 - \rho^2\omega_k}. \tag{25}$$

This allows us to more efficiently compute ω, given the initial conditions $\omega_1 = 1$ and $\omega_2 = \frac{2}{2-\rho^2}$.

Golub and Varga [1961] extensively analyzed the Chebyshev semi-iterative method. They pointed out that although the Chebyshev method looks similar to weighted Jacobi and successive over-relaxation (SOR), it converges much faster. This is because the Chebyshev method changes the factor ω in each iteration and it uses $\mathbf{y}^{(k-1)}$, not $\mathbf{y}^{(k)}$.

Real eigenvalues. The previous analysis is based on the assumption that all eigenvalues of $\mathbf{B}^{-1}\mathbf{C}$ are real. Although this is not true in general, if $\mathbf{A}$ is a symmetric matrix with positive diagonal entries and $\mathbf{B}^{-1}\mathbf{C}$ is created by Jacobi, the eigenvalues must be real. Let λ and $\mathbf{v}$ be an eigenvalue and its eigenvector of $\mathbf{B}^{-1}\mathbf{C}$: $\mathbf{B}^{-1}\mathbf{C}\mathbf{v} = \lambda\mathbf{v}$. We have



(a) The Jacobi method (b) Projective dynamics

Fig. 4. The convergence behaviors of the Jacobi method and projective dynamics, without Chebyshev acceleration. This example shows that the convergence of projective dynamics behaves similarly to that of the Jacobi method solving the linear system in one global step. Here κ is the stiffness variable controlling the weight w_c in Equation 14. The error in (a) is the error magnitude of the linear system, and the error in (b) is the error magnitude of the dynamical system in Equation 30.

$\mathbf{B}^{-\frac{1}{2}}\mathbf{C}\mathbf{B}^{-\frac{1}{2}}(\mathbf{B}^{\frac{1}{2}}\mathbf{v}) = \mathbf{B}^{\frac{1}{2}}\lambda\mathbf{B}^{-\frac{1}{2}}(\mathbf{B}^{\frac{1}{2}}\mathbf{v}) = \lambda(\mathbf{B}^{\frac{1}{2}}\mathbf{v})$. So λ is also the eigenvalue of $\mathbf{B}^{-\frac{1}{2}}\mathbf{C}\mathbf{B}^{-\frac{1}{2}}$. Since $\mathbf{B}$ is the diagonal matrix of $\mathbf{A}$ and its diagonal elements are all positive, $\mathbf{B}^{-\frac{1}{2}}$ must be real. So $\mathbf{B}^{-\frac{1}{2}}\mathbf{C}\mathbf{B}^{-\frac{1}{2}}$ is real symmetric and λ is real. Therefore, we can use the Chebyshev method to accelerate Jacobi iterations immediately.

Strengths and weaknesses. The biggest advantage of the Chebyshev semi-iterative method is its simplicity. Compared with Krylov subspace iterative methods, such as generalized minimum residual and conjugate gradient, the Chebyshev method has a short recurrence form and it does not use inner products, so it is ideal for parallel computing. Unfortunately, it is known that the Chebyshev method does not converge as fast as Krylov subspace methods do.

The Chebyshev method also requires[1] the spectral radius ρ, which is often estimated numerically. Let $\hat{\rho}$ be the estimated spectral radius. The Chebyshev method converges differently when $\hat{\rho}$ varies from 0 to 1. When $\hat{\rho} = 0$, the method is reduced to a standard iterative solver and it offers no acceleration. When $\hat{\rho}$ varies from 0 to ρ, the convergence rate is gradually improved. Once $\hat{\rho}$ grows above ρ, the method still converges but oscillation happens. Eventually, the method diverges as $\hat{\rho}$ becomes close to 1. We will explore this convergence property for estimating ρ later in Subsection 2.2.1.

2.2 The Chebyshev Approach

In this section, we will investigate the extension of the Chebyshev method from linear systems to projective dynamics. We will also discuss the use of the Chebyshev approach in position-based dynamics in Subsection 2.2.3.

2.2.1 Chebyshev for Projective Dynamics. Let us first reconsider the idea of replacing one direct solve by one Jacobi iteration in each global step of projective dynamics, as proposed in Subsection 1.3.2. In that case, each projective dynamics iteration contains a local step and a Jacobi iteration. Since the result of the local step is typically stabilized within a few iterations, the linear system in the global step is almost unchanged afterwards. So it is not surprising to see that the convergence of projective dynamics is similar to that of the Jacobi method solving the linear system in a single global step, as shown in Figure 4. Based on this observation, we propose a Chebyshev semi-iterative approach for projective dynamics, which simply replaces one Jacobi iteration in Equation 23 by one projective dynamics iteration.

[1]If we know an even tighter range of the eigenvalues: $\lambda_i \in [\alpha, \beta]$, the Chebyshev method can be adjusted to converge faster, as described in [Golub and Van Loan 1996; Gutknecht and Röllin 2002].

Figure 11a shows this approach effectively accelerates the convergence of projective dynamics, when the global step is solved by one Jacobi iteration.

Interestingly, Figure 11a shows the approach also works when the global step is solved by the direct method or the Gauss-Seidel method. We believe this is because projective dynamics converges in similar patterns, regardless of the methods used to solve the global step. Note that divergence can occur when using the Chebyshev method to immediately accelerate Gauss-Seidel iterations, because the iterative matrix $\mathbf{B}^{-1}\mathbf{C}$ may have complex eigenvalues as Golub and Van Loan [1996] pointed out. We found that the same issue happens to projective dynamics, when the global step is solved by one Gauss-Seidel iteration. Our solution is to solve the global step by two Gauss-Seidel iterations, once in the forward order and once in the backward order. By doing so, the joint iterative matrix has real eigenvalues and our Chebyshev approach is effective again.

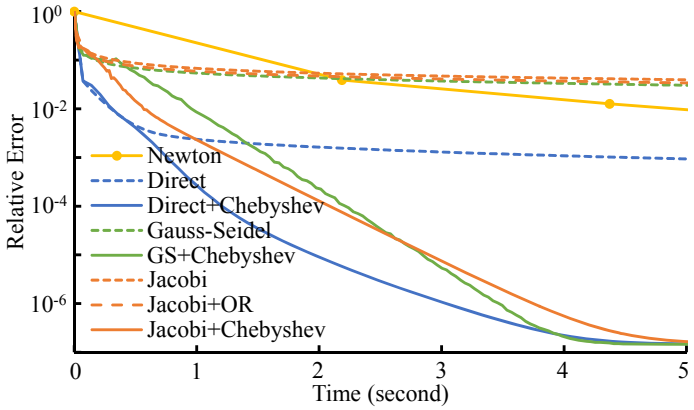
The choice of ρ. The coefficient ρ was originally defined in Section 2.1 as the spectral radius of the iterative matrix $\mathbf{B}^{-1}\mathbf{C}$, when we use the Chebyshev method to solve a linear system. But for projective dynamics, we cannot define ρ in this way, since the combination of the local step and the global step is fundamentally nonlinear. One may think that ρ is related to the iterative method solving the linear system in the global step. But that cannot explain why the Chebyshev approach still works when the global step is solved by the direct method.

Fortunately, due to the similarity between the convergence of projective dynamics and that of a linear solver, we treat ρ as constant for each simulation problem and we estimate ρ by pre-simulation in two steps. Let K be the total number of projective dynamics iterations. We first initialize ρ by $\left\|\mathbf{e}^{(K)}\right\|_2 \Big/ \left\|\mathbf{e}^{(K-1)}\right\|_2$, where the error is defined[2] as $\mathbf{e}^{(k)} = \nabla\epsilon(\mathbf{q}^{(k)})$. This is similar to how the power method estimates the spectral radius for a standard iterative method. After that, we manually adjust ρ and run the simulation by the Chebyshev approach multiple times, to find the optimal ρ that maximizes the convergence rate. Similar to the Chebyshev method for linear systems, our Chebyshev approach causes oscillation when ρ is over-estimated. So if oscillation occurs, we know ρ is too large and we reduce it accordingly.

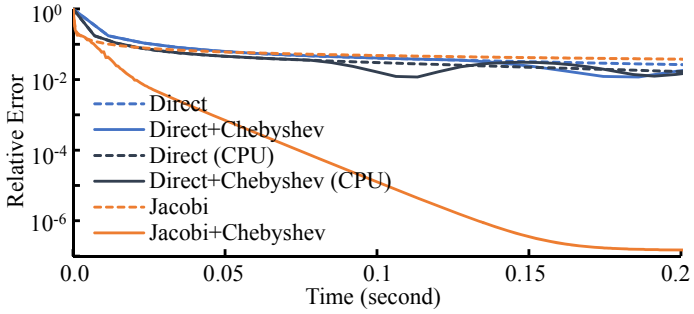
We can safely assume that ρ is constant through the whole simulation process, since the choice of ρ is insensitive to the simulation state $\mathbf{q}$ as shown in all of our simulation examples. In fact, ρ is relatively insensitive to small changes made to the system as well. These changes are typically needed by positional or collision constraints, or local remeshing processes. Therefore, we can make these changes immediately during simulation, without changing ρ. Our experiment does demonstrate a strong relationship between ρ and the stiffness of an object. When the stiffness increases, projective dynamics converges more slowly as shown in Figure 4b. So ρ needs to be larger to make the Chebyshev approach more effective. Our experiment also reveals that ρ depends on the total number of iterations K. Projective dynamics typically converges fast in the first few iterations. After that, the convergence rate gradually drops as Figure 11 shows. So we must make ρ larger, when K increases.

Performance evaluation. Figure 11 compares the performance of projective dynamics using different methods. The Newton's method converges fastest, but its iterations are too expensive to make its overall performance attractive, as Figure 11a shows. On CPU, projective dynamics using the direct method runs fastest. The use of Gauss-Seidel iterations allows projective dynamics to run slightly faster than the use of Jacobi iterations as expected, and over-relaxation (OR) has little effect on the performance. The Chebyshev approach accelerates projective dynamics that uses any of the three methods, among which the direct+Chebyshev method is the fastest.

[2]Our error definition is different from the one used in [Bouaziz et al. 2014], which measures the energy difference from the ground truth $\mathbf{q}^*$: $\epsilon(\mathbf{q}) - \epsilon(\mathbf{q}^*)$. We will use our definition in the rest of this paper.



(a) Performance on CPU



(b) Performance on GPU

Fig. 5. The performance of projective dynamics on the tablecloth example, using different methods. The Chebyshev approach can accelerate projective dynamics that uses any of the three methods: the direct method, the Gauss-Seidel method, or the Jacobi method. Among them, the Jacobi method benefits most from the Chebyshev approach, since it is more compatible with GPU acceleration as shown in (b).

On GPU, the Jacobi+Chebyshev method outperforms any other method as Figure 11b shows, since it is naturally compatible with parallel computing. In contrast, forward and backward substitutions involved in the direct method are more expensive to run on GPU than on CPU, as they are largely sequential. To implement the direct method, we use the eigen library on CPU and the cuSOLVER library on GPU. The pre-computed factorization is done by Cholesky decomposition with permutation. The GPU implementation of the direct method was tested on NVIDIA Tesla K40c.

2.2.2 Convergence and Robustness. It is uncommon to see divergence caused solely by the Chebyshev approach in practice, since we would notice oscillations first during the parameter tuning process and we can always lower ρ to fix the issue. However, there is one small problem due to the use of a constant ρ. Projective dynamics typically converges fast in the first few iterations, and its convergence rate drops after that. This means when the total number of iterations is large, ρ must be smaller than it should be, to avoid oscillation or even divergence in the first few iterations. To address this problem, we simply delay the start of the Chebyshev approach, by setting $\omega_1 = \omega_2 = \ldots = \omega_S = 1$. In our experiment, we typically use $S = 10$ for the Jacobi+Chebyshev method. A more suitable strategy is to use different ρ for different iterations, but doing so makes ρ difficult to estimate and we need to study this further in the future.

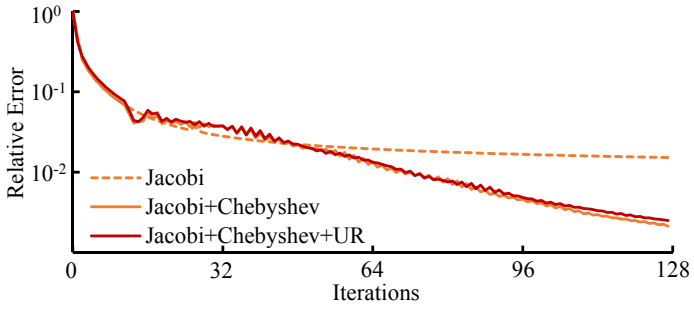
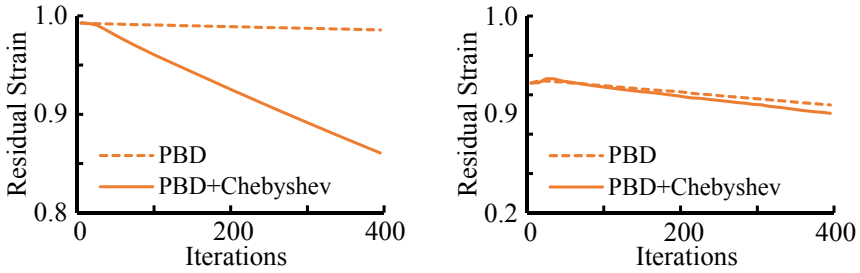


Fig. 6. The effect of under-relaxation (UR) on the armadillo example. This plot shows that under-relaxation has limited influence on the convergence of the Chebyshev approach, when $\gamma = 0.9$.



(a) Triangular mesh (b) Tetrahedral mesh

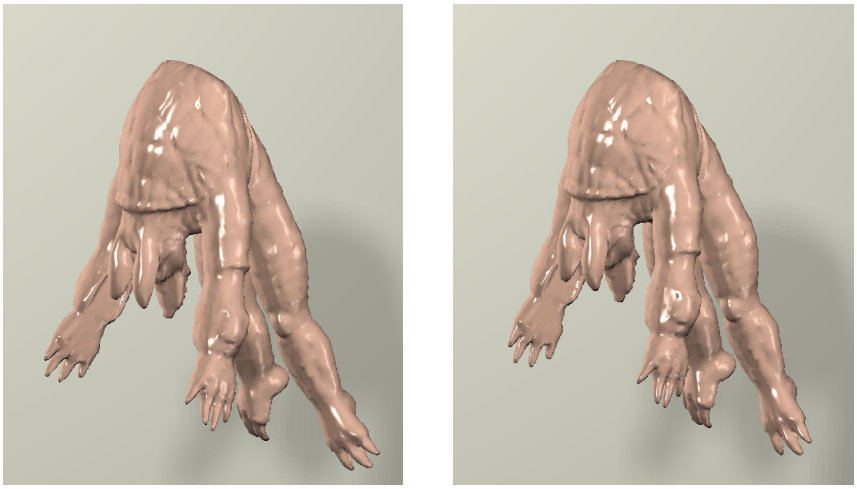
Fig. 7. The convergence of position-based dynamics, using Jacobi iterations. This example shows that the acceleration effect of the Chebyshev approach is more significant on triangular meshes as shown in (a) than on tetrahedral meshes as shown in (b).

To ensure the convergence of the Jacobi method, we must have: $\rho(\mathbf{B}^{-1}\mathbf{C}) < 1$. This is automatically satisfied by spring constraints, since their matrix $\mathbf{A}$ is diagonally dominant. Unfortunately, this may not be true for other constraints, and the Jacobi method will not be able to solve the linear system involved in the global step. Interestingly, this does not immediately lead to the failure of projective dynamics, when it solves the global step by only one Jacobi iteration. In fact, our experiment shows the divergence issue occurs, only when the mesh contains small elements or experiences very large stress. We also found that the divergence issue can be reduced by inserting an under-relaxation step before the Chebyshev update:

$$\mathbf{q}^{(k+1)} = \omega_{k+1}\left(\gamma\left(\hat{\mathbf{q}}^{(k+1)} - \mathbf{q}^{(k)}\right) + \mathbf{q}^{(k)} - \mathbf{q}^{(k-1)}\right) + \mathbf{q}^{(k-1)}, \tag{26}$$

in which $\hat{\mathbf{q}}^{(k+1)}$ is the result produced by one Jacobi iteration and $\gamma \in [0.6, 0.9]$ is the under-relaxation coefficient. This under-relaxation step has little influence on the convergence rate as shown in Figure 6. Note that under-relaxation cannot fully eliminate the divergence issue. An ultimate solution is to make the matrix diagonally dominant by using a smaller time step, but that increases the computational cost of the whole system. It is possible that strain limiting can address this issue, by preventing elements from being significantly deformed.

2.2.3 Chebyshev for Position-based Dynamics. Since position-based dynamics can be considered as a simplified version of projective dynamics as shown in Subsection 1.3.5, one may think that it can be naturally accelerated by the Chebyshev approach as well. Our research shows the actual situation is complicated.



(a) Before the Chebyshev approach (b) After the Chebyshev approach

Fig. 8. The armadillo example simulated by position-based dynamics. In this example, position-based dynamics is more unstable after it gets accelerated by the Chebyshev approach. To avoid divergence, a smaller under-relaxation coefficient must be used and the resulting acceleration becomes less noticeable.

Let i be a vertex and $\Delta\mathbf{q}_{i,c}$ be its position movement suggested by constraint c. We calculate its actual position update in each iteration by a relaxation coefficient α:

$$\mathbf{q}_i^{(k+1)} = \mathbf{q}_i^{(k)} + \alpha \sum_c \Delta\mathbf{q}_{i,c}. \tag{27}$$

We prefer using Equation 27 over the averaging scheme proposed by Macklin and colleagues [2014], since it is momentum preserving and slightly more robust against divergence. Given the same α, if no divergence occurs, using the Chebyshev approach always results in faster convergence than not using the Chebyshev approach, as our experiment shows. The problem is that the Chebyshev approach makes the algorithm more vulnerable to divergence, so smaller α is needed for better stability. When simulating triangular meshes, we found the difference in α is not significant: $\alpha = 0.3$ for not using the Chebyshev approach[3], and $\alpha = 0.25$ for using the Chebyshev approach. So the acceleration provided by the Chebyshev approach is still obvious, as Figure 7a shows. But when simulating tetrahedral meshes, α must be much smaller to avoid divergence: $\alpha = 0.045$ for not using the Chebyshev approach and $\alpha = 0.0025$ for using the Chebyshev approach. Overall, the simulation is still accelerated, but the effect is less evident as shown in Figure 7b and 8.

We believe these phenomena are largely due to the fact that the linear system in the global step is trivially solved in an iteration, when position-based dynamics assumes $\mathbf{A}_c^\mathsf{T}\mathbf{A}_c = \mathbf{A}_c^\mathsf{T}\hat{\mathbf{A}}_c = \mathbf{I}$. As a result, the convergence of position-based dynamics does not behave so closely to the convergence of an iterative method, and it cannot always benefit a lot from the Chebyshev approach.

2.3 Implementation

The pseudo code of our system is given in Algorithm 3.

[3]If we assume that a vertex has six neighbors, $\alpha = 0.3$ is equivalent to setting the over-relaxation coefficient to 1.8 in [Macklin et al. 2014].

Algorithm 1 `Chebyshev_PD_Solve`($\mathbf{q}_t$, $\mathbf{v}_t$, ρ, h, K)

$\mathbf{q}^{(0)} \leftarrow \mathbf{s}_t \leftarrow \mathbf{q}_t + h\mathbf{v}_t + h^2\mathbf{M}^{-1}\mathbf{f}_{\text{ext}}$;
for $k = 0 \ldots K-1$ **do**
  for each constraint c **do**
    $\mathbf{p}_c$ = `Local_Solve`$(c, \mathbf{q}^{(k)})$;
  $\hat{\mathbf{q}}^{(k+1)} \leftarrow$ `Jacobi_Solve`$(\mathbf{s}_t, \mathbf{q}^{(k)}, \mathbf{p}_1, \mathbf{p}_2, \ldots)$;
  if $k < S$ **then** $\omega_{k+1} \leftarrow 1$;
  if $k = S$ **then** $\omega_{k+1} \leftarrow 2/(2-\rho^2)$;
  if $k > S$ **then** $\omega_{k+1} \leftarrow 4/(4-\rho^2\omega_k)$;
  $\mathbf{q}^{(k+1)} = \omega_{k+1}\left(\gamma\left(\hat{\mathbf{q}}^{(k+1)} - \mathbf{q}^{(k)}\right) + \mathbf{q}^{(k)} - \mathbf{q}^{(k-1)}\right) + \mathbf{q}^{(k-1)}$;
$\mathbf{q}_{t+1} \leftarrow \mathbf{q}^{(K)}$;
$\mathbf{v}_{t+1} \leftarrow (\mathbf{q}_{t+1} - \mathbf{q}_t)/h$;

Name (#vert, #ele)	# constraints	Direct (CPU)			ρ	Jacobi (CPU)			Jacobi (GPU)		
		#iter	Cost	FPS		#iter	Cost	FPS	#iter	Cost	FPS
Tablecloth (10K, 20K)	30K+30K	10	60.9ms	16	0.9999	400	0.76s	1.3	400	26.3ms	38
Dress (16K, 29K)	43K+44K	10	151ms	7	0.9992	192	0.65s	1.5	192	27.0ms	37
Armadillo (15K, 55K)	55K	10	76.8ms	13	0.9992	64	1.34s	0.7	64	20.8ms	48
Fishnet (20K, 64K)	64K	10	92.9ms	11	0.9996	64	1.53s	0.7	64	26.7ms	38

Table 1. Statistics and timings of the examples. We did not test the use of the direct method on GPU for all of the examples, since the direct solve is typically slower on GPU than on CPU. To make our comparisons fair, the CPU timings of the direct method do not contain the costs of the local steps, which can be accelerated by GPU.

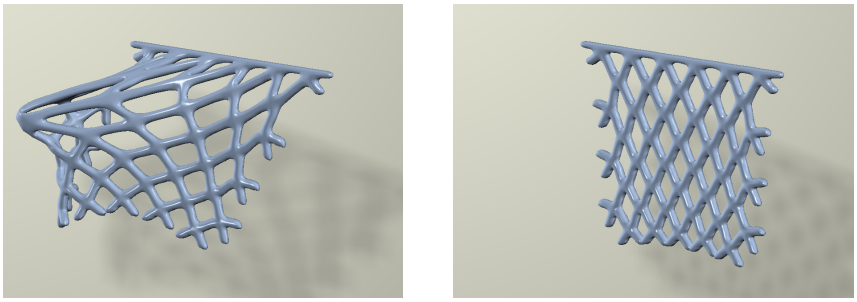
Constraints and their energies. We use two types of constraints to simulate triangular meshes: the spring constraint for every edge, and the hinge-edge constraint for every edge adjacent to two triangles. We define the energy of a spring constraint by the spring potential energy and the energy of a hinge-edge constraint by the quadratic bending energy [Bergou et al. 2006], assuming that the mesh is initially flat. Since the bending energy is a quadratic function of $\mathbf{q}$, its energy Hessian matrix can be directly inserted into the linear system and it needs no local step.

We use tetrahedral constraints to simulate tetrahedral meshes and we define the elastic energy of a tetrahedron as: $\kappa \text{vol}(c) \|\mathbf{F}_c - \mathbf{R}_c\|_{\text{F}}^2$, where κ is the stiffness, $\text{vol}(c)$ is the volume before deformation, $\mathbf{F}_c$ is the deformation gradient, and $\mathbf{R}_c$ is the rotational component of $\mathbf{F}_c$. To implement this energy in projective dynamics, we define $\mathbf{A}_c$ as the matrix that converts $\mathbf{q}$ into the deformation gradient $\mathbf{F}_c$ in the Voigt form. We directly formulate $\hat{\mathbf{A}}_c\mathbf{p}_c$ as the Voigt form of $\mathbf{R}_c$, so we do not define $\hat{\mathbf{A}}_c$ explicitly. Let $\mathbf{F}_c = \mathbf{R}_c\mathbf{Q}_c\mathbf{\Lambda}_c\mathbf{Q}_c^{\mathsf{T}}$ be the decomposition of $\mathbf{F}_c$, such that $\mathbf{Q}_c$ is an orthogonal matrix and $\mathbf{\Lambda}_c$ is a diagonal matrix containing the principal stretches λ_1, λ_2, and λ_3. The elastic energy density function is proportional to:

$$\left\|\mathbf{A}_c\mathbf{q} - \hat{\mathbf{A}}_c\mathbf{p}_c\right\|^2 = \|\mathbf{F}_c - \mathbf{R}_c\|_{\text{F}}^2 = \|\mathbf{\Lambda}_c - \mathbf{I}\|_{\text{F}}^2 = \sum_i (\lambda_i - 1)^2. \tag{28}$$

Our experiment shows that this hyperelastic model is sufficient to produce many interesting elastic behaviors.

Polar decomposition. To simulate tetrahedral meshes, we need polar decomposition to extract the rotational component $\mathbf{R}$ from the deformation gradient $\mathbf{F}$: $\mathbf{F} = \mathbf{RS}$, in which $\mathbf{S}$ is a symmetric matrix known as the *stretch tensor*. A typical way of obtaining $\mathbf{R}$ is to perform singular value decomposition (SVD) on $\mathbf{F} = \mathbf{U\Sigma V}^{\mathsf{T}}$ and calculate $\mathbf{R}$ as: $\mathbf{R} = \mathbf{UV}^{\mathsf{T}}$. Unfortunately, SVD is too computationally expensive and it can easily become the bottleneck of our algorithm. We also



(a) Deformed shape (b) Recovered shape

Fig. 9. The fishnet example. Our simulator can robustly recover the rest shape of an elastic fishnet in (b), after it got deformed by user interaction shown in (a).

tested several iterative methods [Bouby et al. 2005; Rivers and James 2007], which turned out to be either too expensive or too unstable. Our final implementation is based on the direct approach proposed by Franca [1989]. Their idea is to calculate the three principal invariants of $\mathbf{S}$ from $\mathbf{F}^T\mathbf{F}$ first, and then use them to derive $\mathbf{S}$. We note that their original method can be easily modified to handle inverted elements as well, by using the determinant of $\mathbf{F}$ to decide the sign of the third invariant.

Positional and contact constraints. Similar to [Bouaziz et al. 2014], we handle positional constraints by using stiff springs to connect vertices and their desired positions. Although we can introduce collision constraints into the system as Bouaziz and colleagues [2014] did, we found it is easier to simply enforce them at the end of each iteration. To efficiently detect cloth-body collisions, we model the female body by a signed distance field. We found that it is difficult to handle static friction correctly in each iteration, since the vertex position change does not have sufficient physical meaning. So in this example, we break each time step into eight sub-steps and handle collisions and frictions at the end of each sub-step.

2.4 Results and Discussions

(Please see the supplementary video for animation examples.) We integrated the Chebyshev approach into our simulators and tested their performances on both CPU and GPU. Our CPU test runs on an Intel i5-2500K processor using a single core. Our GPU test runs on an NVIDIA GeForce GTX 970 card, except for the timings in Figure 11. The computational cost of the system depends on the number of constraints and the number of vertices involved in each constraint, rather than the total number of vertices. For example, tetrahedral constraints are more expensive than spring constraints, as each tetrahedral constraint needs four vertices. In average, the local step that enforces all of the constraints consumes 20 to 50 percent of the dynamic simulation cost on GPU. Note that our GPU implementation does not use atomic operations to transfer results back to vertices in the local step, as did in [Macklin et al. 2014]. Instead, it collects the enforced results for every vertex in the global step. So the cost of the global step is dominated by memory access. Table 2 lists the statistics and the timings of our examples. All of the examples use $h = 1/30$s as the time step. Figure 9 shows a fishnet example simulated by our Chebyshev approach.

In Table 2, we use 10 iterations to solve projective dynamics that uses the direct method, as suggested in [Bouaziz et al. 2014]. But 10 iterations is typically not enough for projective dynamics to reach a small error, especially if an object

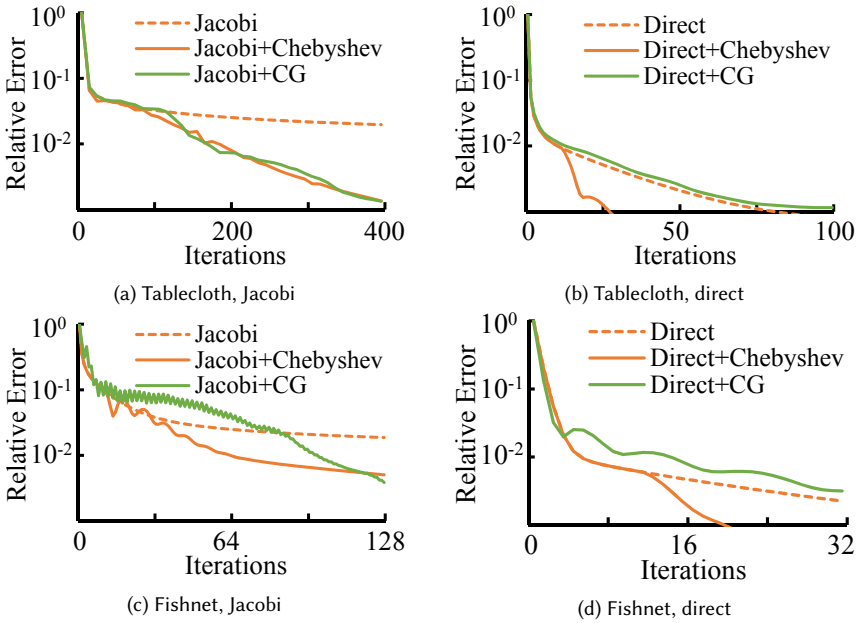


Fig. 10. The comparisons between the Chebyshev approach and the conjugate gradient approach. In both examples, the conjugate gradient approach using the Jacobi preconditioner cannot converge faster than the Chebyshev approach, as shown in (a) and (c). This is also true when the conjugate gradient approach uses the direct solver as the preconditioner, as shown in (b) and (d).

undergoes large and fast deformation. We do not recommend to couple the direct method with the Chebyshev approach in the first 10 iterations, for robustness reasons discussed in Subsection 2.2.2. Therefore, we need more iterations and computational time, if we want more accurate results from the use of the direct method.

2.4.1 Comparisons to Conjugate Gradient. An interesting question is whether projective dynamics can be accelerated by conjugate gradient as well. So we implement the nonlinear preconditioned conjugate gradient method as Algorithm 2 shows. Here we just use one sub-iteration to solve the linear search step, since the result often becomes worse when more sub-iterations are used. Figure 10 shows that regardless of the preconditioner, the convergence rate of the conjugate gradient approach cannot be higher than that of the Chebyshev approach in both examples. Since one conjugate gradient iteration is more computationally expensive than one Chebyshev iteration on both CPU and GPU, the overall performance of the conjugate gradient approach must be lower and we do not recommend its use in practice.

2.4.2 Strengths and Weaknesses. While the Chebyshev approach effectively accelerates projective dynamics that uses a variety of linear system solvers, we would like to advocate the combination of the Chebyshev approach and Jacobi iterations for several reasons. First, it is straightforward to implement and it does not require additional linear solver libraries. Second, it is compatible with GPU acceleration and it has a small memory cost. Finally, the Chebyshev approach is relatively insensitive to small system changes. In contrast, the direct method needs to re-factorize the matrix every time the system gets changed, which requires a large computational cost.

That being said, the convergence criterion of the Jacobi method compromises the robustness of projective dynamics for tetrahedral meshes. Since divergence happens only when the mesh is in low quality or under large stress as our

Algorithm 2 `PCG_PD_Solve`($\mathbf{q}_t$, $\mathbf{v}_t$, ρ, h, K)

$\mathbf{q}^{(0)} \leftarrow \mathbf{s}_t \leftarrow \mathbf{q}_t + h\mathbf{v}_t + h^2\mathbf{M}^{-1}\mathbf{f}_{\text{ext}}$;
for $k = 0 \ldots K-1$ **do**
  for each constraint c **do**
    $\mathbf{p}_c$ = `Local_Solve`($c, \mathbf{q}^{(k)}$);
  $\mathbf{r}^{(k+1)} \leftarrow -\nabla\epsilon(\mathbf{q}^{(k)}, \mathbf{p}_c)$;
  If $\left\|\mathbf{r}^{(k+1)}\right\|_2^2 < \varepsilon$ **then break**;
  $\mathbf{z}^{(k+1)} \leftarrow$ `Precondition_Solver`($\mathbf{r}^{(k+1)}$);
  if k=0 **then**
    $\mathbf{p}^{(k+1)} \leftarrow \mathbf{z}^{(k+1)}$;
  else
    $\beta \leftarrow (\mathbf{z}^{(k+1)} \cdot \mathbf{r}^{(k+1)})/(\mathbf{z}^{(k)} \cdot \mathbf{r}^{(k)})$;
    $\mathbf{p}^{(k+1)} \leftarrow \mathbf{z}^{(k+1)} + \beta\mathbf{p}^{(k)}$;
  $\alpha \leftarrow (\mathbf{p}^{(k+1)} \cdot \mathbf{r}^{(k+1)})/(\mathbf{p}^{(k+1)} \cdot \mathbf{A}\mathbf{p}^{(k+1)})$; ▷ Line search
  $\mathbf{x}^{(k+1)} \leftarrow \mathbf{x}^{(k)} + \alpha\mathbf{p}^{(k+1)}$;
$\mathbf{q}_{t+1} \leftarrow \mathbf{q}^{(K)}$;
$\mathbf{v}_{t+1} \leftarrow (\mathbf{q}_{t+1} - \mathbf{q}_t)/h$;

experiment shows, it would be interesting to know whether we can use failsafe methods (such as strain limiting) to avoid divergence in these cases accordingly. While we can now simulate the dynamics fast, we still cannot efficiently handle self-collisions on GPU yet. In our current implementation, the computational cost of self-collision handling is nearly twice of the dynamic simulation cost.

3 PARALLEL DESCENT METHODS

In this section, we discuss a generation of the Chebyshev method introduced in Sec. 2 to allow it to synergize with various material models such as linear models, spring models, hinge-edge bending models [Bergou et al. 2006; Garg et al. 2007], hyperelastic models, and spline-based models [Xu et al. 2015]. This method converges to exact implicit Euler integration under a given elastic model. It is robust against divergence, even when handling large time steps and deformations as shown in the bottom row of Figure 1. The whole method is fast, scalable and has a small memory footprint. For further speedups, it can also be combined with multi-grid methods, many of which were designed for hexahedral lattices [Dick et al. 2011; McAdams et al. 2011b; Patterson et al. 2012; Zhu et al. 2010] at this time.

This method was published in SIGGRAPH Asia 2016 [Wang and Yang 2016].

3.1 Descent Methods

Let $\mathbf{q} \in \mathbb{R}^{3N}$ and $\mathbf{v} \in \mathbb{R}^{3N}$ be the vertex position and velocity vectors of a nonlinear elastic body. We can use implicit time integration to simulate the deformation of the body from time t to $t+1$ as:

$$\mathbf{q}_{t+1} = \mathbf{q}_t + h\mathbf{v}_{t+1}, \quad \mathbf{v}_{t+1} = \mathbf{v}_t + h\mathbf{M}^{-1}\mathbf{f}(\mathbf{q}_{t+1}), \tag{29}$$

in which $\mathbf{M} \in \mathbb{R}^{3N\times 3N}$ is the mass matrix, h is the time step, and $\mathbf{f} \in \mathbb{R}^{3N}$ is the total force as a function of $\mathbf{q}$. By combining the two equations, we obtain a single nonlinear system:

$$\mathbf{M}(\mathbf{q}_{t+1} - \mathbf{q}_t - h\mathbf{v}_t) = h^2\mathbf{f}(\mathbf{q}_{t+1}). \tag{30}$$

Since $\mathbf{f}(\mathbf{q}) = -\partial E(\mathbf{q})/\partial \mathbf{q}$, where $E(\mathbf{q})$ is the total potential energy evaluated at $\mathbf{q}$, we can convert the nonlinear system into an unconstrained nonlinear optimization problem: $\mathbf{q}_{t+1} = \arg\min \epsilon(\mathbf{q})$,

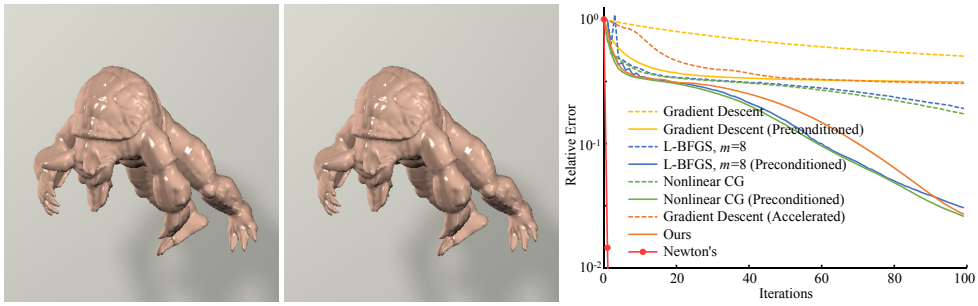
$$\epsilon(\mathbf{q}) = \frac{1}{2h^2}(\mathbf{q} - \mathbf{q}_t - h\mathbf{v}_t)^{\mathsf{T}}\mathbf{M}(\mathbf{q} - \mathbf{q}_t - h\mathbf{v}_t) + E(\mathbf{q}), \tag{31}$$

Nonlinear optimization is often solved by descent methods, which contain four steps in each iteration as Algorithm 3 shows. The main difference is in how to calculate the descent direction, which typically involves the use of the gradient: $\mathbf{g}^{(k)} = \nabla\epsilon(\mathbf{q}^{(k)})$.

Algorithm 3 `Descent_Optimization`

Initialize $\mathbf{q}^{(0)}$;
for $k = 0 \ldots K-1$ **do**
  Calculate the descent direction $\Delta\mathbf{q}^{(k)}$; *Step 1*
  Adjust the step length $\alpha^{(k)}$; *Step 2*
  $\bar{\mathbf{q}}^{(k+1)} \leftarrow \mathbf{q}^{(k)} + \alpha^{(k)}\Delta\mathbf{q}^{(k)}$; *Step 3*
  $\mathbf{q}^{(k+1)} \leftarrow$ `Acceleration`$(\bar{\mathbf{q}}^{(k+1)}, \bar{\mathbf{q}}^{(k)}, \mathbf{q}^{(k)}, \mathbf{q}^{(k-1)})$; *Step 4*
return $\mathbf{q}^{(K)}$;

Gradient descent. The gradient descent method simply sets the descent direction as: $\Delta\mathbf{q}^{(k)} = -\mathbf{g}^{(k)}$, using the fact that $\epsilon(\mathbf{q})$ decreases the fastest locally in the negative gradient direction. While gradient descent has a small computational cost per iteration, its convergence rate is only linear as shown in Figure 11c. Gradient descent can be viewed as updating $\mathbf{q}$ by the force, since the negative gradient of the potential energy is the force. This is fundamentally similar to explicit time integration. Therefore, the step length must be small to avoid the divergence issue.



(a) Ground truth (b) Our result (c) The convergence plot

Fig. 11. The outcomes of descent methods applied to the armadillo example. Thanks to preconditioning and momentum-based acceleration, our method converges as fast as nonlinear conjugate gradient and it needs a much smaller GPU cost. Our result in (b) is visually indistinguishable from the ground truth in (a) generated by Newton's method. In the plot, we define the relative error as $\big(\epsilon(\mathbf{q}^{(k)}) - \epsilon(\mathbf{q}^*)\big)/\big(\epsilon(\mathbf{q}^{(k)}) - \epsilon(\mathbf{q}^{(0)})\big)$, where $\mathbf{q}^{(k)}$ is the result in the k-th iteration and $\mathbf{q}^*$ is the ground truth.

Newton's method. To achieve quadratic convergence, Newton's method approximates $\epsilon(\mathbf{q}^{(k)})$ by a quadratic function and it calculates the search direction[4] as: $\Delta\mathbf{q}^{(k)} = -\big(\mathbf{H}^{(k)}\big)^{-1}\mathbf{g}^{(k)}$, where $\mathbf{H}^{(k)}$ is the Hessian matrix of $\epsilon(\mathbf{q})$ evaluated at $\mathbf{q}^{(k)}$. Figure 11c shows Newton's method converges the fastest. However, it is too computationally expensive to solve the linear system $\mathbf{H}^{(k)}\Delta\mathbf{q}^{(k)} = -\mathbf{g}^{(k)}$ involved in every iteration. For example, to solve a linear system in the armadillo example shown in Figure 11, the Eigen library needs 0.77 seconds by Cholesky factorization, or 2.82 seconds by preconditioned conjugate gradient with incomplete LU factorization. The use of the Pardiso module reduces the Cholesky factorization cost to 0.13 seconds, which is still not affordable by real-time applications. Most linear solvers cannot be easily parallelized on the GPU.

Quasi-Newton methods. Since it is too expensive to solve a linear system or even just evaluate the Hessian matrix, a natural idea is to approximate the Hessian matrix or its inverse. For example, quasi-Newton methods, such as BFGS, use previous gradient vectors to approximate the inverse Hessian matrix directly. To avoid storing a dense inverse matrix, L-BFGS defines the approximation by m gradient vectors, each of which provides rank-one updates to the inverse matrix sequentially. While L-BFGS converges slower than Newton's method, it has better performance thanks to its reduced cost per iteration. Unfortunately, the sequential nature of L-BFGS makes it difficult to run on the GPU, unless the problem is subject to box constraints [Fei et al. 2014].

Nonlinear conjugate gradient (CG).. The nonlinear conjugate gradient method generalizes the conjugate gradient method to nonlinear optimization problems. Based on the Fletcher–Reeves formula, it calculates the descent direction as:

$$\Delta\mathbf{q}^{(k)} = -\mathbf{g}^{(k)} + \frac{z^{(k)}}{z^{(k-1)}}\Delta\mathbf{q}^{(k-1)}, \quad z^{(k)} = \mathbf{g}^{(k)} \cdot \mathbf{g}^{(k)}. \tag{32}$$

Nonlinear CG is highly similar to L-BFGS with $m = 1$. The reason it converges slightly faster than L-BFGS in our experiment is because we use the exact Hessian matrix to estimate the step length. Intuitively, this is identical to conjugate gradient, except that the residual vector, i.e., the gradient, is recalculated in every iteration. Nonlinear CG is much more friendly with GPU acceleration than quasi-Newton methods. But it still requires multiple dot product operations, which restrict its performance on the GPU.

[4]The search direction of Newton's method is not guaranteed to be always descending, unless $\mathbf{H}^{(k)}$ is positive definite.

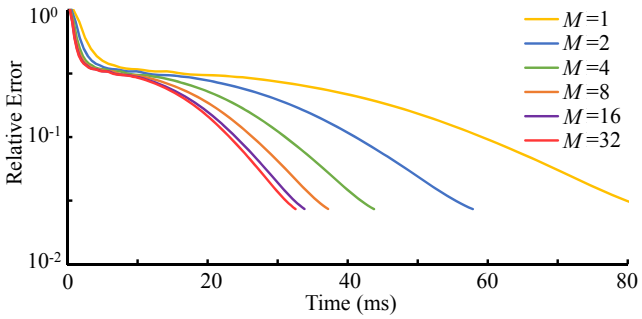


Fig. 12. The convergence of our method within 96 iterations, when using different M values to delay matrix evaluation. This plot shows that the method can reach the same residual error, regardless of M. Therefore we can use a larger M to reduce the matrix evaluation cost and improve the system performance.

3.2 Preconditioning Descent Method by Jacobi+Chebyshev

In this section, we describe the technique used in our descent method. We will also evaluate their performance and compare them with alternatives.

3.2.1 Descent Direction. The idea behind our method is inspired by preconditioned conjugate gradient. To achieve faster convergence, preconditioning converts the optimization problem into a well conditioned one:

$$\bar{\mathbf{q}} = \arg\min \epsilon\left(\mathbf{P}^{-1/2}\bar{\mathbf{q}}\right), \quad \text{for } \bar{\mathbf{q}} = \mathbf{P}^{1/2}\mathbf{q}, \tag{33}$$

where $\mathbf{P}$ is the preconditioner matrix. Mathematically, doing this is equivalent[5] to replacing $\mathbf{g}^{(k)}$ by $\mathbf{P}^{-1}\mathbf{g}^{(k)}$ in Equation 32. Among all of the preconditioners, we favor the Jacobi preconditioner the most, since it is easy to implement and friendly with GPU acceleration. When an optimization problem is quadratic, conjugate gradient defines the Jacobi preconditioner as a constant matrix: $\mathbf{P} = \texttt{diag}(\mathbf{H})$, where $\mathbf{H}$ is the constant Hessian matrix. To solve a general nonlinear optimization problem, if the Hessian matrix can be quickly evaluated in every iteration, we can treat $\mathbf{P}\left(\mathbf{q}^{(k)}\right) = \texttt{diag}(\mathbf{H}^{(k)})$ as the Jacobi preconditioner for nonlinear CG, which now varies from iteration to iteration. Such a Jacobi preconditioner significantly improves the convergence rate of nonlinear CG, as Figure 11c shows.

This Jacobi preconditioner can be effectively applied to L-BFGS and gradient descent as well. Preconditioning in L-BFGS is essentially defining $\texttt{diag}^{-1}(\mathbf{H}^{(k)})$ as the initial inverse Hessian estimate. Meanwhile, preconditioned gradient descent simply defines its new descent direction as: $\Delta\mathbf{q}^{(k)} = -\texttt{diag}^{-1}(\mathbf{H}^{(k)})\mathbf{g}^{(k)}$. While preconditioned gradient descent does not converge as fast as other preconditioned methods, it owns a unique property: its convergence rate can be well improved by momentum-based techniques. Based on this observation, we propose to formulate our basic method as accelerated, Jacobi preconditioned gradient descent. Figure 11 demonstrates that the convergence rate of our method is comparable to that of preconditioned nonlinear CG, and our result is visually similar to the ground truth after 96 iterations.

Why is our method special? While both Jacobi preconditioning and momentum-based acceleration are popular techniques, it is uncommon to see them working with gradient descent for solving general nonlinear optimization problems. There are reasons for this. The use of Jacobi preconditioning destroys the advantage of gradient descent in requiring zero Hessian matrix estimation. Meanwhile, Chebyshev acceleration is effective only when the problem is

[5]The calculation of $z^{(k)}$ should be updated as: $z^{(k)} = \mathbf{g}^{(k)} \cdot \mathbf{P}^{-1}\mathbf{g}^{(k)}$.

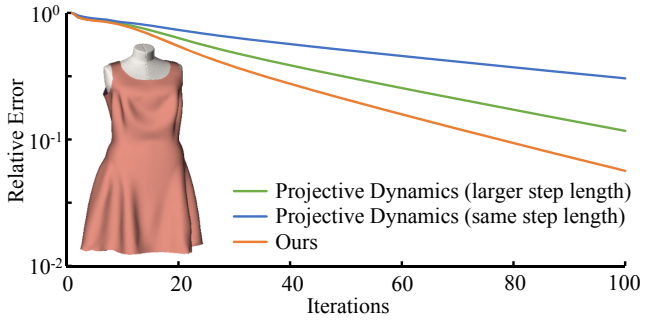


Fig. 13. The convergence of our method and projective dynamics. Although projective dynamics can use a large step length, it cannot converge as fast as our method.

mildly nonlinear [Gutknecht and Röllin 2002]. Fortunately, our method works well with elastic body simulation, both quasi-statically and dynamically.

Convergence and performance. The calculation of our descent direction has two obvious advantages. First, the diagonal entries of the Hessian matrix are typically positive. As a result, $\mathtt{diag}^{-1}(\mathbf{H}^{(k)})$ is positive definite and $\Delta\mathbf{q}^{(k)} \cdot \mathbf{g}^{(k)} < 0$. Within a bounded deformation space, the Hessian matrix of $\epsilon(\mathbf{q})$ is also bounded: $\mathbf{H} \preceq B\mathbf{I}$. We have:

$$\epsilon\left(\mathbf{q}^{(k)} + \alpha^{(k)}\Delta\mathbf{q}^{(k)}\right) \leq \epsilon\left(\mathbf{q}^{(k)}\right) + \alpha^{(k)}\Delta\mathbf{q}^{(k)} \cdot \mathbf{g}^{(k)} + \frac{B}{2}\left\|\alpha^{(k)}\Delta\mathbf{q}^{(k)}\right\|_2^2. \tag{34}$$

This means there must exist a sufficiently small step length $\alpha^{(k)}$ that ensures the energy decrease and eliminates the divergence issue. Second, both the Jacobi preconditioner and gradient descent are computationally inexpensive and suitable for parallelization. In particular, it requires zero reduction operation.

The use of the Jacobi preconditioner demands the evaluation of the Hessian matrix. This can become a computational bottleneck if it is done in every iteration. Fortunately, we found that it is acceptable to evaluate the Hessian matrix once every M iterations and use the last matrix for the preconditioner. Conceptually, this strategy is similar to high-order Newton-like methods that skip derivative evaluation [Cordero et al. 2010]. Figure 12 demonstrates that doing this has little effect on the convergence rate, but significantly reduces the computational cost per iteration, when $M \leq 32$. We choose not to make M even bigger, since it is unnecessary and it may slow down the convergence rate, especially if the object moves fast under large deformation.

Comparison to projective dynamics. The projective dynamics technique (Sec. 1.3) solves the optimization problem by interleaving a local constraint step and a global solve step. If we view the local step as calculating the gradient and the global step as calculating the descent direction, we can interpret projective dynamics as preconditioned gradient descent as well. Here the preconditioner matrix is constant, so it can be pre-factorized for fast solve in every iteration. But this is not the only advantage of projective dynamics. Bouaziz and collaborators [2014] pointed out that projective dynamics is guaranteed to converge by setting $\alpha^{(k)} \equiv 1$, if the elastic energy of every element has a quadratic form $\|\mathbf{Aq} - \mathbf{Bp}(\mathbf{q})\|^2$, where $\mathbf{A}$ and $\mathbf{B}$ are two constant matrices and $\mathbf{p}(\mathbf{q})$ is the geometric projection of $\mathbf{q}$ according to that element. Therefore, projective dynamics does not need to adjust the step length in every iteration.

By using the Chebyshev semi-iterative method (Sec. 2), projective dynamics can be implemented on the GPU by removing off-diagonal entries of the preconditioner matrix. In this regard, that method is highly related to the method presented here. Since both methods can handle mass-spring systems, we compare their convergence rates as shown in

Figure 13. When both methods use the same step length: $\alpha^{(k)} \equiv 0.5$, our method converges significantly faster. This is not a surprise, given the fact that our method uses the diagonal of the exact Hessian matrix and Newton's method converges faster than original projective dynamics. The strength of projective dynamics allows it to use $\alpha^{(k)} \equiv 1$. But even so, it is still not comparable to our method. Interestingly, we do not observe substantial difference in animation results of the two methods. We guess this is because the stiffness in this example is too large. As a result, small energy difference cannot cause noticeable difference in vertex positions.

Comparison to a single linear solve. Figure 11 may leave us an impression that it is always acceptable to solve just one Newton's iteration, as did in many existing simulators [Baraff and Witkin 1998; Dick et al. 2011]. Mathematically, it is equivalent to approximating the energy by a quadratic function and solving the resulting linear system. In that case, our method is simplified to the accelerated Jacobi method. Doing this has a clear advantage: the gradient does not need to be reevaluated in every iteration, which can be costly for tetrahedral elements. However, Newton's method may diverge, especially if the time step is large and the initialization is bad. This problem can be lessened by using a small step length. But then it becomes pointless to waste computational resources within one Newton's iteration. In contrast, gradient descent still converges reasonably well under the same situation. Because of that, we decide not to rely on quadratic approximation, i.e., one Newton's iteration.

Comparison to nonlinear CG. The biggest competitor of our method is actually nonlinear CG. Figure 11c shows that the two methods have similar convergence rates. The real difference in their performance is determined by the computational cost per iteration. While the two methods have similar performance on the CPU, our method runs three to four times faster than nonlinear CG on the GPU. This is because nonlinear CG must perform at least two dot product operations, each of which takes 0.41ms in the armadillo example using the CUDA thrust library. In contrast, the cost of our method is largely due to gradient evaluation, which takes 0.17ms per iteration and is also needed by nonlinear CG.

Similar to our method, nonlinear CG must use a smaller step length when the energy function becomes highly nonlinear. But unlike our method, it does not need momentum-based acceleration or parameter tuning. In the future, if parallel architecture can allow dot products to be quickly calculated, we may prefer to to use nonlinear CG instead.

3.2.2 Step Length Adjustment. Given the search direction $\Delta\mathbf{q}^{(k)}$, the next question is how to calculate a suitable step length $\alpha^{(k)}$. A simple yet effective approach, known as *backtracking line search*, gradually reduces the step length, until the first Wolfe condition gets satisfied:

$$\epsilon\left(\mathbf{q}^{(k)} + \alpha^{(k)}\Delta\mathbf{q}^{(k)}\right) < \epsilon\left(\mathbf{q}^{(k)}\right) + c^{(k)}\alpha^{(k)}\Delta\mathbf{q}^{(k)} \cdot \mathbf{g}^{(k)}, \tag{35}$$

in which c is a control parameter. The Wolfe condition is straightforward to evaluate on the CPU. However, it becomes problematic on the GPU, due to expensive energy summation and dot product operations. To reduce the computational cost, we propose to eliminate the dot product by setting $c = 0$. Intuitively, it means we just search for the largest $\alpha^{(k)}$ that ensures monotonic energy decrease: $\epsilon\left(\mathbf{q}^{(k)} + \alpha^{(k)}\Delta\mathbf{q}^{(k)}\right) < \epsilon\left(\mathbf{q}^{(k)}\right)$. We also choose to adjust the step length every eight iterations. Doing this reduces the total number of energy evaluations, although it causes more wasted iterations during the backtracking process.

Our simulator explores the continuity of α between two successive time steps. Specifically, it initializes the step length at time $t + 1$ as $\alpha = \alpha_t/\gamma$, in which α_t is the ending step length at time t. After that, the simulator gradually reduces α by $\alpha := \gamma\alpha$, until the Wolfe condition gets satisfied. In our experiment, we use $\gamma = 0.7$. When the step length is too small, our method converges slowly and it is not worthwhile to spend more iterations. Therefore, if the Wolfe

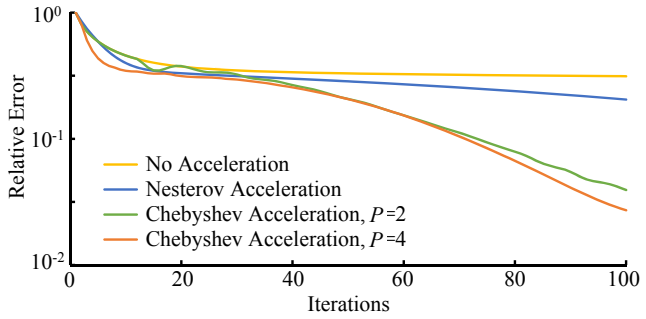


Fig. 14. The convergence of our method with different acceleration techniques. By using multiple phases, the Chebyshev method can more effectively accelerate the convergence process.

condition still cannot be satisfied after the step length reaches a minimum value, we end the time step and start the next one.

3.2.3 Momentum-based Acceleration. An important strength of our method is that it benefits from the use of momentum-based acceleration techniques, such as the Chebyshev semi-iterative method [Golub and Van Loan 1996] and the Nesterov's method [Nesterov 2004]. Here the term "momentum" refers to the result change between the last two iterations, not the actual physical momentum. Since the result change can be calculated independently for every vertex, momentum-based techniques are naturally compatible with parallel computing.

The two methods differ in how they define and weight the result change. The weight used by the Chebyshev method is calculated from the gradient decrease rate, which can be tuned for different problems as shown in [Wang 2015]. On the other hand, the control parameter used by the Nesterov's method is related to the strong convexity of the Hessian matrix. Since this parameter is not easy to find, it is often set to zero for simplicity. Because of such a difference, the Chebyshev method typically outperforms the Nesterov's method, as shown in Figure 14. Our experiment shows that the Chebyshev method is also more reliable, as long as the gradient decrease rate is underestimated. In contrast, the Nesterov's method may need multiple restarts to avoid the divergence issue [O'donoghue and Candès 2015].

We note that neither of the techniques was designed for general descent methods. The Chebyshev method was initially developed for linear solvers, while the Nesterov's method was proposed for speeding up the gradient descent method. Since our method is highly related to linear solvers[6] and gradient descent, it can be effectively accelerated by momentum-based acceleration techniques. Neither L-BFGS nor nonlinear CG can be accelerated by these techniques, as far as our experiment shows.

Adaptive parameters. By using the Chebytshev method for accelerating projective dynamics as in Sec. 2, the gradient decrease rate ρ is defined as a constant:

$$\rho \approx \left\|\nabla\epsilon(\mathbf{q}^{k+1})\right\| \Big/ \left\|\nabla\epsilon(\mathbf{q}^{k})\right\|. \tag{36}$$

This is a reasonable practice, since the rate is related to the spectral radius of the global matrix, which stays the same through the whole simulation process. The simulation of generic elastic materials, however, can exhibit more complex convergence behaviors. If a constant ρ is still used, it must be kept at the minimum level to avoid oscillation or even divergence issues, especially in the first few iterations. To make Chebyshev acceleration more effective, we propose to

[6]Our method can be viewed as solving the linear system in each Newton's iteration by only one iteration of the Jacobi method.

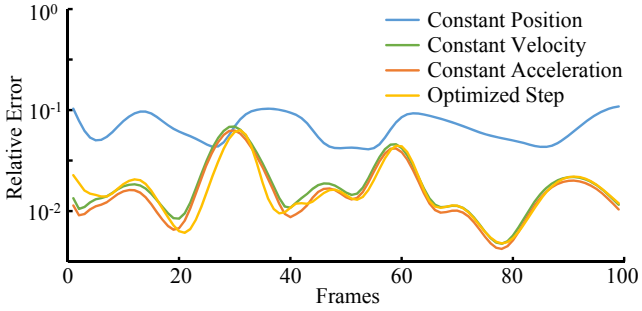


Fig. 15. The convergence of our method using different initialization approaches. This plot shows that the constant acceleration approach works the best in most cases.

use a varying ρ instead. Specifically, we divide the iterations into P phases and assign each phase with its own ρ. We can then perform the transition from one phase to another by simply restarting the Chebyshev method. The question is: *how can we tune the phases and their parameters*? Our idea is to change the length and the parameter of a phase each time, and then test whether that helps the algorithm reduce the residual error in pre-simulation. We slightly increase or decrease ρ each time by:

$$\rho^{\text{new}} = 1 - (1 \pm \epsilon)(1 - \rho), \tag{37}$$

where ϵ is typically set to 0.05. We accept the change that causes the most significant error decrease, and then start another tuning cycle. The tuning process terminates once the error cannot be reduced any further. Figure 14 compares the convergence rates of our method, by using two and four Chebyshev phases respectively.

3.2.4 Initialization. The initialization of $\mathbf{q}^{(0)}$ is also an important component in our algorithm. It helps the descent method reduce the total energy to a low level, after a fixed number of iterations. Intuitively, the initialization works as a prediction on the solution $\mathbf{q}_{t+1}$. Here we test four different prediction approaches. The first three assume that vertex positions, velocities, and accelerations are constant, respectively: $\mathbf{q}_{t+1} \approx \mathbf{q}^{(0)} = \mathbf{q}_t$; $\mathbf{q}_{t+1} \approx \mathbf{q}^{(0)} = \mathbf{q}_t + h\mathbf{v}_t$; $\mathbf{q}_{t+1} \approx \mathbf{q}^{(0)} = \mathbf{q}_t + h\mathbf{v}_t + \eta h(\mathbf{v}_t - \mathbf{v}_{t-1})$. We use the parameter η to damp the acceleration effect, which is typically set to 0.2. The fourth approach assumes that vertices move in the $\mathbf{v}_t$ direction with an unknown step distance d: $\mathbf{q}^{(0)} = \mathbf{q}_t + d\mathbf{v}_t$. We then optimize d by minimizing a quadratic approximation of $\epsilon(\mathbf{q}_t + d\mathbf{v}_t)$:

$$d = \arg\min_d \left\{ \epsilon(\mathbf{q}_t) + (d\mathbf{v}_t) \cdot \nabla\epsilon(\mathbf{q}_t) + \frac{1}{2}(d\mathbf{v}_t) \cdot \mathbf{H}(\mathbf{q}_t)(d\mathbf{v}_t) \right\}, \tag{38}$$

which can be solved as a simple linear equation. This is similar to how the conjugate gradient method determines the optimal step in a search direction.

Figure 15 compares the effects of the four approaches on the convergence of our method, over a precomputed sequence with 100 frames. It shows that the optimized step approach does not outperform the constant acceleration approach in most cases, even though it is the most sophisticated one. Because of this, our system chooses the constant acceleration approach to initialize $\mathbf{q}^{(0)}$ by default. We note that Figure 15 illustrates the errors during a single frame only. These errors can be accumulated over time, causing slightly larger differences in animation results. These differences are often manifested as small artificial damping artifacts, as shown in our experiment.

3.3 Nonlinear Elastic Models

Our new descent method can handle any elastic model, if: 1) its energy function is second-order differentiable; and 2) the Hessian matrix of its energy function can be quickly evaluated. These two conditions are satisfied by many elastic models, including spring model under Hooke's law, hinge-edge bending models [Bergou et al. 2006; Garg et al. 2007], hyperelastic models, and spline-based models [Xu et al. 2015]. In this section, we would like to specifically discuss hyperelastic models, some of which are not suitable for immediate use in simulation.

Hyperelastic models are developed by researchers in mechanical engineering and computational physics to model complex force-displacement relationships of real-world materials. The energy density function of an isotropic hyperelastic material is typically defined by the three invariants of the right Cauchy-Green deformation tensor $\mathbf{C} = \mathbf{F}^{\mathsf{T}}\mathbf{F}$:

$$I = \mathtt{tr}(\mathbf{C}), \quad II = \mathtt{tr}(\mathbf{C}^2), \quad III = \det(\mathbf{C}). \tag{39}$$

Here $\mathbf{F}$ is the deformation gradient. For example, the St. Venant-Kirchhoff model has the following strain energy density function:

$$W = \frac{s_0}{2}(I-3)^2 + \frac{s_1}{4}(II - 2I + 3), \tag{40}$$

where s_0 and s_1 are the two elastic moduli controlling the resistance to deformation, also known as the Lamé parameters. The compressible neo-Hookean model [Ogden 1997] defines its strain energy density function as:

$$W = s_0\left(III^{-1/3}\cdot I - 3\right) + s_1\left(III^{-1/2} - 1\right), \tag{41}$$

in which s_0 is the shear modulus and s_1 is the bulk modulus. Many hyperelastic models can be considered as extensions of the neo-Hookean model. For example, the compressible Mooney-Rivlin model for rubber-like materials uses the following strain energy density function [Macosko 1994]:

$$W = s_0\left(III^{-1/3}\cdot I - 3\right) + s_1\left(III^{-1/2} - 1\right) + s_2\left(\tfrac{1}{2}III^{-2/3}(I^2 - II) - 3\right). \tag{42}$$

To model the growing stiffness of soft tissues, the isotropic Fung model [Fung 1993] adds an exponential term:

$$W = s_0\left(III^{-1/3}\cdot I - 3\right) + s_1\left(III^{-1/2} - 1\right) + s_2\left(e^{s_3(III^{-1/3}\cdot I - 3)} - 1\right), \tag{43}$$

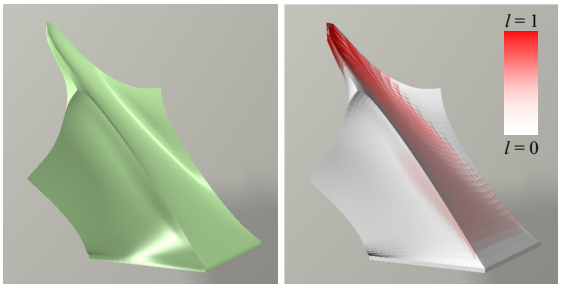
in which s_3 controls the speed of the exponential growth.

Invertible model conversion. A practical problem associated with the use of hyperelastic models is that they are not designed for highly compressed or inverted cases. As a result, a simulated hyperelastic body can become unnecessarily stiff, or even stuck in an inverted shape. A common solution to this problem is to set a limit on the compression rate or the stress, as described by Irving and colleagues [2004]. Since such a limit will cause C^2 discontinuity in the deformation energy, we choose not to do so in our system.

Our solution is to use projective dynamics instead. Bouaziz and colleagues [2014] proved that projective dynamics is numerically robust, even against inverted cases. Its basic form uses the following energy density function:

$$W^{\mathtt{proj}} = \sum\nolimits_{i=1}^{3}(\lambda_i - 1)^2, \tag{44}$$

in which λ_1, λ_2, and λ_3 are the three principal stretches, i.e., the singular values of the deformation gradient. Our basic idea is to gradually convert a hyperelastic model into projective dynamics, when an element gets highly compressed. Let $[\lambda^- = 0.05, \lambda^+ = 0.15]$ be the typical stretch interval for model conversion to happen in our experiment. For every



(a) A deformed box (b) Interpolants visualized in red

Fig. 16. A deformed box and its interpolants. For the St. Venant-Kirchhoff model, we set $\lambda^+ = 0.5$ to address its low resistance against compression. Even so, only a small number of elements need to use invertible model conversion.

element t in the k-th iteration, we define an interpolant $l_t^{(k)}$ as:

$$l_t^{(k)} = \min\left(1, \max\left(0, l_t^{(k-1)} - L, \max_i \left(\lambda^+ - \lambda_i\right)/\left(\lambda^+ - \lambda^-\right)\right)\right), \tag{45}$$

where $l_t^{(0)}$ is set to 0 and L is typically set to 0.05. The reason we use the $l_t^{(k-1)} - L$ term in Equation 45 is to prevent the interpolant from being rapidly changed between two time steps, which can cause oscillation artifacts in animation. We then formulate the hybrid elastic energy density of the element in the k-th iteration as:

$$W_t^{\texttt{hybrid}} = \left(1 - l_t^{(k)}\right) W_t + l_t^{(k)} W_t^{\texttt{proj}}, \tag{46}$$

where W_t is the hyperelastic energy density of element t. According to Equation 46, we calculate the total contribution of element t to the Jacobi preconditioner as:

$$\mathbf{P}_t\left(\mathbf{q}^{(k)}\right) = \texttt{diag}\left(\left(1 - l_t^{(k)}\right)\mathbf{H}_t^{(k)} + l_t^{(k)}\mathbf{A}_t^{\mathsf{T}}\mathbf{A}_t\right), \tag{47}$$

where $\mathbf{H}_t^{(k)}$ is the Hessian matrix of W_t and $\mathbf{A}_t^{\mathsf{T}}\mathbf{A}_t$ is the constant matrix of element t used by projective dynamics. It is straightforward to implement model conversion described in Equation 47, thanks to the structural similarity between our algorithm and GPU-based projective dynamics (Sec. 2). We note that the interpolant is defined for every element. This allows most elements to maintain the original hyperelastic model, even when we use a larger λ^+ as Figure 16 shows.

3.4 Implementation and Results

We implemented and tested our system on both the CPU and the GPU. Our CPU implementation used the Eigen library (eigen.tuxfamily.org) and OpenMP. The CPU tests ran on an Intel i7-4790K 4.0GHz quad-core processor. The GPU tests ran on an NVIDIA GeForce GTX TITAN X graphics card with 3,072 cores. Although many parameters are used in our system, most of them are related to the performance or the result quality, not the stability. The only exception is the Chebyshev parameters, which can be automatically tuned as described in Subsection 3.2.3. The statistics and the timings of our examples are provided in Table 2. All of our tetrahedral mesh examples use $h = 1/30$s as the time step and run 96 iterations per time step. The dress example also uses $h = 1/30$s as the time step, but it divides each time step into 8 substeps and executes 40 iterations per substep. It handles cloth-body collision at the end of each substep.

Name	#vert	#ele	CPU Cost	GPU Cost	GPU FPS
Dragon (Fig. 1)	16K	58K	1.35s	32.8ms	30.5
Armadillo (Fig. 11)	15K	55K	1.28s	31.4ms	31.8
Box (Fig. 16)	14K	72K	1.47s	37.6ms	26.6
Dress (Fig. 13)	15K	44K	0.29s	26.6ms	37.6
Double helix (Fig. 18)	13K	41K	0.98s	27.5ms	36.4
Double helix (Fig. 18)	24K	82K	1.91s	38.5ms	26.0
Double helix (Fig. 18)	48K	158K	3.86s	65.4ms	15.3
Double helix (Fig. 18)	96K	316K	7.78s	122ms	8.2

Table 2. Statistics and timings of our examples. By default, the CPU costs are evaluated with OpenMP enabled.

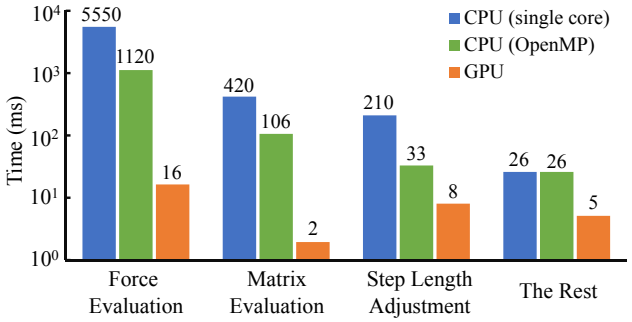
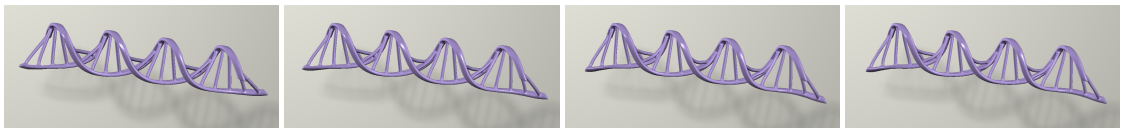


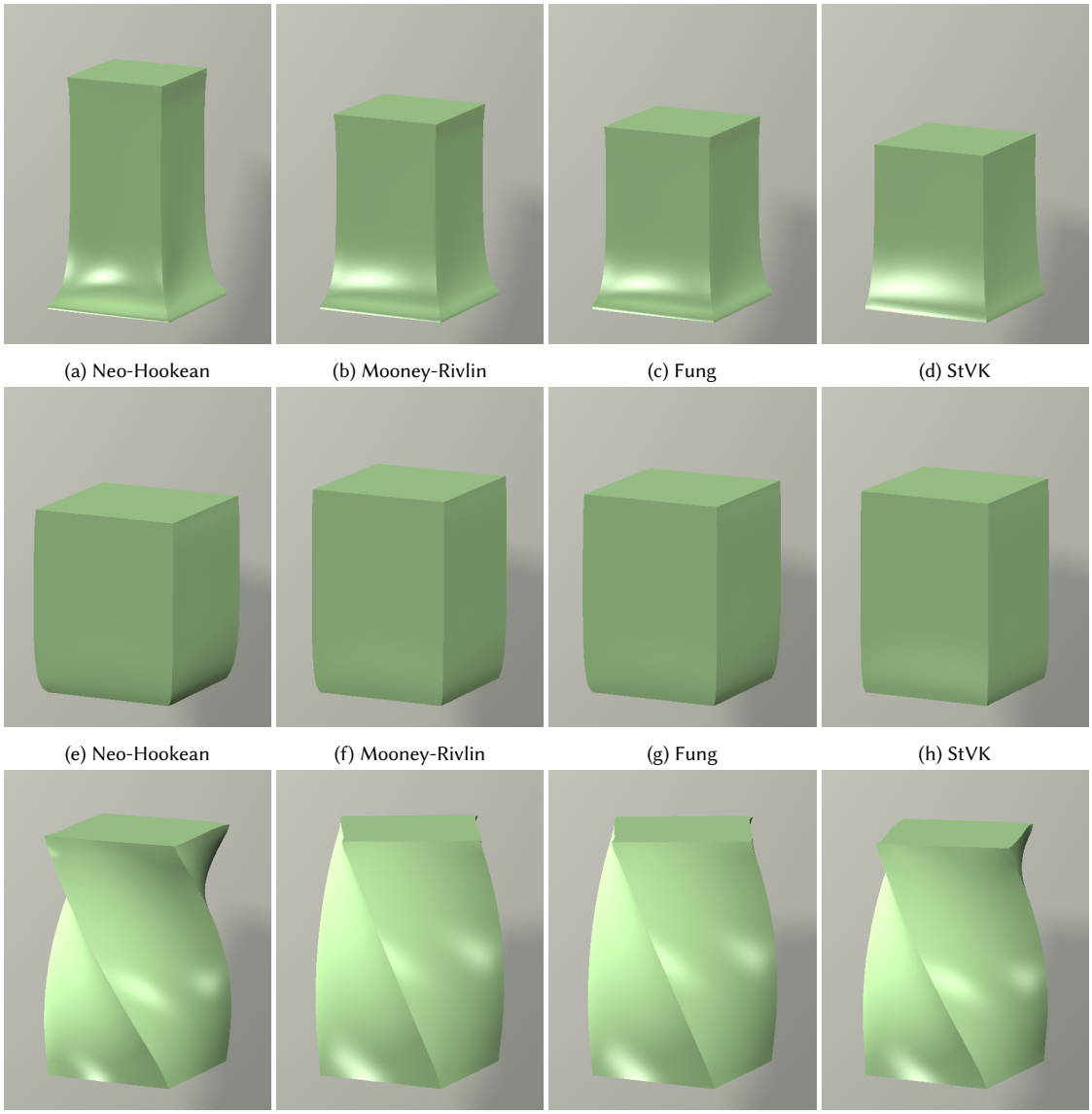
Fig. 17. The costs of the computational steps under three different implementations. This plot illustrates that the force evaluation step is the most expensive one.



(a) Using 41K tetrahedra (b) Using 82K tetrahedra (c) Using 158K tetrahedra (d) Using 316K tetrahedra

Fig. 18. The double helix example. This example indicates that our method can handle high-resolution meshes without overly stretching artifacts. All of the cases use 96 iterations per time step.

GPU implementation. In our GPU implementation, we handle each iteration in two steps. In the first step, we evaluate the forces and the matrices of every element. We apply the fast method proposed by McAdams and colleagues [2011a] for singular value decomposition. We provide two ways to evaluate the Hessian matrix of an elastic model: the co-rotational scheme using strain invariants [Teran et al. 2005] and the spline-based scheme using principal stretches [Xu et al. 2015], the latter of which is slightly more complex but flexibly handles generic orthotropic models. Since our algorithm needs only the diagonal entries of the Hessian matrix, we can avoid the evaluation of the whole matrix and reduce the computational cost. Once we obtain the forces and the matrices, we distribute them to the four vertices using atomic CUDA operations. In the second step, we calculate the descent direction, adjust the step length, and update vertex positions by Chebyshev acceleration. Our step length adjustment scheme needs the total energy, which is computed by a CUDA thrust reduction operation.



(a) Neo-Hookean (b) Mooney-Rivlin (c) Fung (d) StVK

(e) Neo-Hookean (f) Mooney-Rivlin (g) Fung (h) StVK

(i) Neo-Hookean (j) Mooney-Rivlin (k) Fung (l) StVK

Fig. 19. The box example. The simulator presented here can robustly and efficiently simulate the stretching, compression, and twisting behaviors of a box, under different hyperelastic models.

Both air damping and viscous damping can be easily integrated into our system. Let the air damping force be:

$$\mathbf{f}^{\mathtt{air}}(\mathbf{q}) = -\frac{c}{h}(\mathbf{q} - \mathbf{q}_t), \tag{48}$$

in which c is the air damping coefficient. The corresponding damping energy is $-\frac{c}{2h}\|\mathbf{q} - \mathbf{q}_t\|^2$ and its Hessian matrix is $-\frac{c}{h}\mathbf{I}$. Viscous damping can be implemented in a similar way, by taking the adjacency into consideration. Both air

damping and viscous damping can make the Hessian matrix more diagonally dominant and reduce the condition number of the optimization problem. To fully demonstrate the stability of our system, we turned damping off in our experiment. The observed energy loss is mainly caused by implicit time integration.

Our system can handle collisions in two ways. It can model collisions by repulsive potential energies and add them into the total energy. Alternatively, it can treat collisions as position constraints and enforce them at the end of each time step. Although the second approach requires smaller steps, it can simulate static frictions more appropriately. Therefore, we choose it to handle cloth-body collisions in the dress example.

Performance evaluation. Figure 17 shows that our algorithm is not attractive without parallelization. Its total computational cost is dominated by the force evaluation step, which is needed in every single iteration. In contrast, the matrix evaluation step is much less expensive, since it is performed once every $M = 32$ iterations as discussed in Subsection 3.2.1. Enabling OpenMP effectively reduces the computational costs on the CPU, but the algorithm is still not fast enough for real-time applications, even though our implementation has space for further optimization. Fortunately, the algorithm runs significantly faster on the GPU, thanks to the use of thousands of GPU threads as demonstrated in Figure 17.

To reveal the scalability of our algorithm, we simulate a double helix example at four resolutions. Table 2 shows that the computational cost is almost linearly proportional to the number of tetrahedra as expected. The high-resolution result in Figure 18d does not exhibit any overly stretching artifact, which is a common issue in position-based dynamics. Nevertheless, if computational resources permit, we still recommend the use of more iterations for high-resolution meshes, to reduce residual errors and artificial damping artifacts.

Model analysis. To evaluate the simulated behaviors of different hyperelastic models, we design a box example where the bottom face is fixed and the top face is loaded by stretching, compression, or twisting forces, as shown in Figure 19. Here we use the same s_0 and s_1 for the neo-Hookean model, the Mooney-Rivlin model, and the Fung model. As a result, the Mooney-Rivlin model and the Fung model behave stiffer than the neo-Hookean model, due to additional terms in their strain energy density functions. From our experiment, we found that the St. Venant-Kirchhoff model is the most difficult one to handle, because of its low resistance against compression. Although we can address this problem by using a larger λ^+ to perform invertible model conversion earlier, it is still difficult to tune the stiffness parameter of projective dynamics, since low stiffness cannot fix inverted elements while high stiffness can cause oscillation between the two models. An alternative solution is to use the isotropic strain limiting technique [Thomaszewski et al. 2009; Wang et al. 2010]. In that case, more iterations or smaller time steps are needed, as shown in our experiment.

Figure 20 demonstrates the relationship between the stretch ratio of a box and the uplifting force applied on the top face. The nature of our simulator guarantees that its quasistatic result is consistent with the stress-strain relationship specified by the underlying hyperelastic model. In particular, the stiffness of the Fung model grows more rapidly than that of the neo-Hookean model or the Mooney-Rivlin model. Meanwhile, the force is approximately a cubic function of the stretch ratio under the St. Venant-Kirchhoff model, as expected.

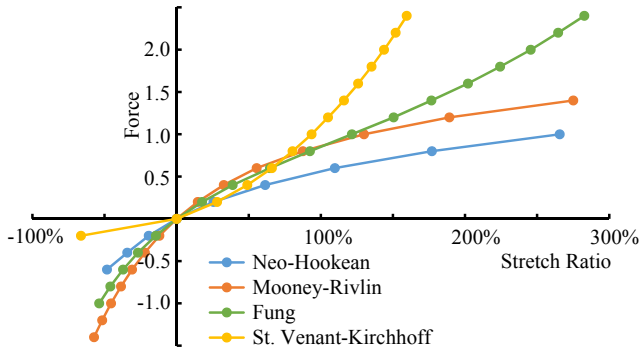


Fig. 20. The force-displacement curves generated by the box example. These curves are consistent with the stress-strain relationships of the underlying hyperelastic models.

4 MULTI-COLOR GAUSS-SEIDEL METHOD

As mentioned in Sec. 1.2, the Gauss-Seidel method has higher convergence speed compared with Jacobi. Its sequential nature, however, makes it unsuitable to run on the GPU. In this section, we present *Vivace*[7], a randomized parallel solver for sparse systems of linear constraints with the convergence speed of the Gauss-Seidel method. In *Vivace* we make use of a parallel randomized graph coloring algorithm [Grable and Panconesi 2000] for re-organizing the unknowns of the sparse linear system, such that no interdependent unknowns in a constraint share the same color. Then, unknowns that belong to the same color are solved in parallel with a single Gauss-Seidel step, leading to a significant acceleration of the convergence speed. The coloring algorithm itself is parallel and suitable for implementation on modern GPU architectures. While the use of graph coloring is known in our community, the main technical contribution of *Vivace* is the adoption of a randomized model and a careful study of its tradeoffs compared to existing models.

Vivace has three characteristics that make it particularly well suited for interactive soft-body dynamics. First, the randomized coloring algorithm produces partitions of similar size, which leads to balanced workloads, making sure that none of the computational units are overloaded or underutilized. Second, the coloring algorithm is fast enough to be rerun as needed, even for each frame. Thus the workload remains balanced even when the topology of the system changes, e.g. in case of constraints derived by collisions. Finally, the residual error in sequential Gauss-Seidel iterations depends on the order in which the equations are solved [Fratarcangeli and Pellacini 2015]. Randomizing this order, as is done in *Vivace*, removes this error faster leading to a more accurate solution.

The benefits of *Vivace* can be demonstrated for the parallel implementation of Projective Dynamics and Position Based Dynamics. As seen in Sec. 1.3, these two methods require to solve linear systems and as such can be accelerated efficiently with *Vivace*. Within these frameworks, we can comfortably solve hundreds of thousands of constraints in 5 ms including collisions on a modern GPU. Fig. 1, upper row, shows results from our solver. Compared to parallel Jacobi and Jacobi accelerated with Chebyshev polynomials Sec. 2, we found *Vivace* to converge faster and remain significantly more stable in case of small time budgets.

This method was originally published in SIGGRAPH Asia 2016 [Fratarcangeli et al. 2016].

4.1 Parallel Gauss-Seidel Coloring.

We focus our effort on improving the solution of the linear system $\mathbf{Aq} = \mathbf{b}$ since this is the dominant cost in the Projective Dynamics framework. We seek a parallel solution well-suited for GPU evaluation, achieved by an iterative stable scheme that works well even for small per-frame time budgets. We base our work on the parallel Gauss-Seidel literature given the high convergence property of the base method. While Gauss-Seidel is an inherently serial algorithm, for sparse matrices it can be parallelized by dividing the set of equations in partitions, such that any pair of equations belonging to the same partition do not share any unknown. The set of unknowns in each partition can then be safely solved in parallel.

Of the many methods for partitioning a sparse linear system, graph coloring is the one mostly used in practice, both in the numerical analysis literature and in the distributed computing one [Saad 2003]. A graph G is built from the matrix $\mathbf{A}$, where each vertex corresponds to an unknown, and two vertices are connected by an undirected edge if they belong to the same equation, i.e. they are related by a constraint. By coloring G with a distance-1 algorithm with q colors, the unknowns assigned to the same color belong to independent equations by definition. Then, the standard lexicographic Gauss-Seidel method can be applied and, instead of solving the equations one after the other, all the

[7] *Vivace* is a word used for musical movements performed in a lively and brisk manner. Our solver speeds-up the computation of physics-based animation making it more "lively", hence the name.

Algorithm 4 Simulation Step in *Vivace*

1: $\mathbf{q}^0 \leftarrow \mathbf{q}_t + h\mathbf{v}_t + h^2\mathbf{M}^{-1}\mathbf{f}_{ext}$
2: *Graph coloring:* $V = \{\mathbf{q}_i, i = 1, \ldots, N\}$ is partitioned into p colors $C_1, \ldots, C_p$, such that $\forall\ (\mathbf{q}_i, \mathbf{q}_j) \in C_i$, $\mathbf{q}_i$ and $\mathbf{q}_j$ are not shared by any constraint
3: **for** $k = 0 \ldots K-1$ **do**
4: **for** each partition $C_i \subset V$ **do**
5: **for** each $\mathbf{q}_i \in C_i$ **do in parallel**
6: $\hat{\mathbf{q}}_i^{k+1} \leftarrow solve\left(\mathbf{q}_i^0, \mathbf{q}_i^k\right)$
7: $\mathbf{q}_i^{k+1} \leftarrow \omega\left(\hat{\mathbf{q}}_i^{k+1} - \mathbf{q}_i^{k-1}\right) + \mathbf{q}_i^{k-1}$
8: $\mathbf{q}_{t+1} \leftarrow \mathbf{q}^K$
9: $\mathbf{v}_{t+1} \leftarrow (\mathbf{q}_{t+1} - \mathbf{q}_t)/h$

equations belonging to the same partition can be solved together in one parallel step, shifting the complexity from $O(n)$ where n is the number of vertices, to $O(c)$, where c is the number of colors. We refer the reader to [Saad 2003] for a more comprehensive treatment. Since graph coloring belongs to the NP-hard class [Garey and Johnson 1979], the outstanding problem is to find an approximate and efficient algorithm to partition the vertices.

Parallel Graph Coloring. Graph coloring introduces a significant overhead when run each frame. To reduce the per-frame time budget, we investigate parallelizable graph coloring methods, a problem well-studied in the literature [Garey and Johnson 1979; Saad 2003]. Most parallel graph coloring methods are techniques that follow the structure of the method in [Luby 1985], where a solution is formed by iteratively determining an independent set I of vertices (such that no two vertices share a common edge), and color them in parallel. The colored vertices are removed from the graph and process is iterated untill all the vertices have been colored. Different methods are characterized by the manner in which they choose an independent set I. Thus, vertices belonging to the same independent set can be colored in parallel.

Practical Desiderata. When the solver is run on the GPU, q kernels are run sequentially. Therefore it is desirable to have as small a number of colors as possible, to lower the number of kernel executions. It is also desirable to haven roughly an equal number of nodes per color to ensure a balanced workflow. Furthermore, we want to be able to color the graph at each frame, to allow the solver to adapt to topology changes in the graph induces by time-varying constraints, such as collisions. Given these desiderata and that graph coloring is an NP-hard problem, we set to investigate which methods perform well in practice.

4.2 Parallelizing Gauss-Seidel by Randomized Graph Coloring

Alg. 4 shows pseudocode for the *Vivace* solver. The core idea of *Vivace* is the parallelization of lexicographic Gauss-Seidel by using graph coloring. Whenever the topology of the graph induced by the constraints network changes, then the coloring algorithm divides the set of vertices into independent partitions; each one corresponding to a color. Vertices belonging to the same partition are solved in parallel.

The simulation step starts by advancing the dynamics of the system with an explicit Euler step (step 1). Then, the coloring algorithm divides the set of vertices in independent partitions; each one corresponding to a color (step 2). The solver iterates K times over all the constraints. Each particle $\mathbf{q}_i$ belonging to partition C_j is processed in parallel. All the corrections induced by all the constraints sharing the particle are computed and summed together (steps 3-6). Then, a Successive Over-Relaxation (SOR) is applied to further accelerate the convergence speed (step 7). In all our tests, we

have used $\omega = 1.9$. Higher values lead to spurious deformation modes or instabilities. Finally, in steps 8-9, the solution of the solver is assigned to the current position of the particles, and the velocity is updated accordingly.

4.2.1 Parallelization Strategy. In Alg. 5, we define a simple, randomized algorithm belonging to the class of Brooks-Vizing vertex coloring algorithms [Grable and Panconesi 2000]. These algorithms are appealing because they are simple to implement, fast, and use considerably fewer than Δ colors, where Δ is the maximal degree of the graph. As input, we consider graphs representing physically animated objects discretized as triangular or tetrahedral meshes. Such meshes are composed of hundreds of thousands of vertices connected by constraints (e.g., distance, bending and volume constraints), however Δ is low enough to enable real-time colorings.

The input of the algorithm is the undirected graph G corresponding to the matrix $\mathbf{A}$ in the linear system. Every vertex v is initially assigned a palette of available colors denoted as P_u. Colors are identified by consecutive natural numbers. V denotes the set of vertices, and U denotes the set of currently uncolored vertices.

During the *initialization* phase (steps 1-3), a list of Δ_v/s colors is given to the palette of each vertex v, where Δ_v is the degree of v and $s > 1$ is the palette *shrinking* factor, which is constant for the whole graph. Then, the actual coloring round procedure starts and is repeated until all the vertices have been colored. Each coloring round comprises three parallel steps. In the *tentative coloring* round (steps 5-6), a color $c(v)$ is randomly chosen among the available colors in the palette P_v, and assigned to v. Then, in *conflict resolution* (steps 7-12), each vertex checks that none of its neighbors has selected the same tentative color. If this occurs, the coloring of v is accepted and $c(v)$ is removed from the palette of the neighbors. In the *feed the hungry* phase (steps 14-16), a color is added to the palettes which have run out of colors. The maximal amount of colors allowed is $\Delta_v + 1$, but in our experiments we never reached this maximal threshold.

The effect of the shrinking factor is to reduce the number of colors; however increasing it too much leads to slower colorings without meaningful gains in terms of reducing the number of colors. In our case, we found that using the minimal degree of the graph as the value of the shrinking factor leads to the best colorings. To speed up the *conflict resolution* phase, we employed the *Hungarian heuristic* [Luby 1985]: in case of conflict, if the node has the higher index among its neighbors then the coloring is considered legitimate. We have found this strategy to greatly reduce the number of coloring rounds needed by the algorithm to color all the vertices in the graph.

4.2.2 Comparison with Other Graph Coloring. In this section, we compare the randomized coloring used in *Vivace* with other parallel algorithms used in literature. [Luby 1985] proposed a Monte Carlo method to find a *maximal independent set (MIS)* in parallel, that is the largest possible independent set of vertices in the graph. In this approach, a random weight is assigned to each vertex. The weights are a random permutation of the integers $1, 2, \ldots, |V|$. Then, each maximal independent set is constructed in parallel by choosing vertices which are local maxima i.e., that have a weight greater than any other neighbors in $|V|$. Once a maximal independent set is identified, all the vertices belonging to such set are assigned the same color and removed from the graph. The procedure is repeated, using different colors for different maximal independent sets, until all the vertices are colored.

[Jones and Plassmann 1993] improved on Luby's algorithm. Instead of using the same color for each vertex in the same independent set, each vertex is colored individually with the smallest available color not already assigned to a neighboring vertex. This apparently simple improvement reduces both the number of colors and the number of rounds.

The Largest-Degree-First (LDF) algorithm is similar to the Jones-Plassmann approach but, instead of using random weights per vertex, the weight is defined as the current degree of the vertex [Welsh and Powell 1967]. After an independent set is defined, the colored vertices are removed from the graph and the degree of the remaining nodes is updated. A vertex with i colored neighbors requires at most $i + 1$ colors. The LDF algorithm keeps i as small as possible

Algorithm 5 *Vivace* Graph Coloring Procedure [Grable and Panconesi 2000]

```
1:  U ← V                                                        ▷ Initialization
2:  for all vertex v ∈ U do
3:      P_v ← {0, ..., Δ_v/s}
4:  while |U| > 0 do
5:      for all vertices v ∈ U do                                ▷ Tentative coloring
6:          c(v) ← random color in P_v
7:      I ← ∅
8:      for all vertices v ∈ U do                                ▷ Conflict resolution
9:          S ← {colors of all the neighbors of v}
10:         if c(v) ∉ S then
11:             I ← I ∪ {v}
12:             remove c(v) from palette of neighbors of v
13:     U ← U − I
14:     for all vertices v ∈ U do                                ▷ Feed the hungry
15:         if |P_v| = 0 then
16:             P_v ← P_v ∪ {|P_v| + 1}
```

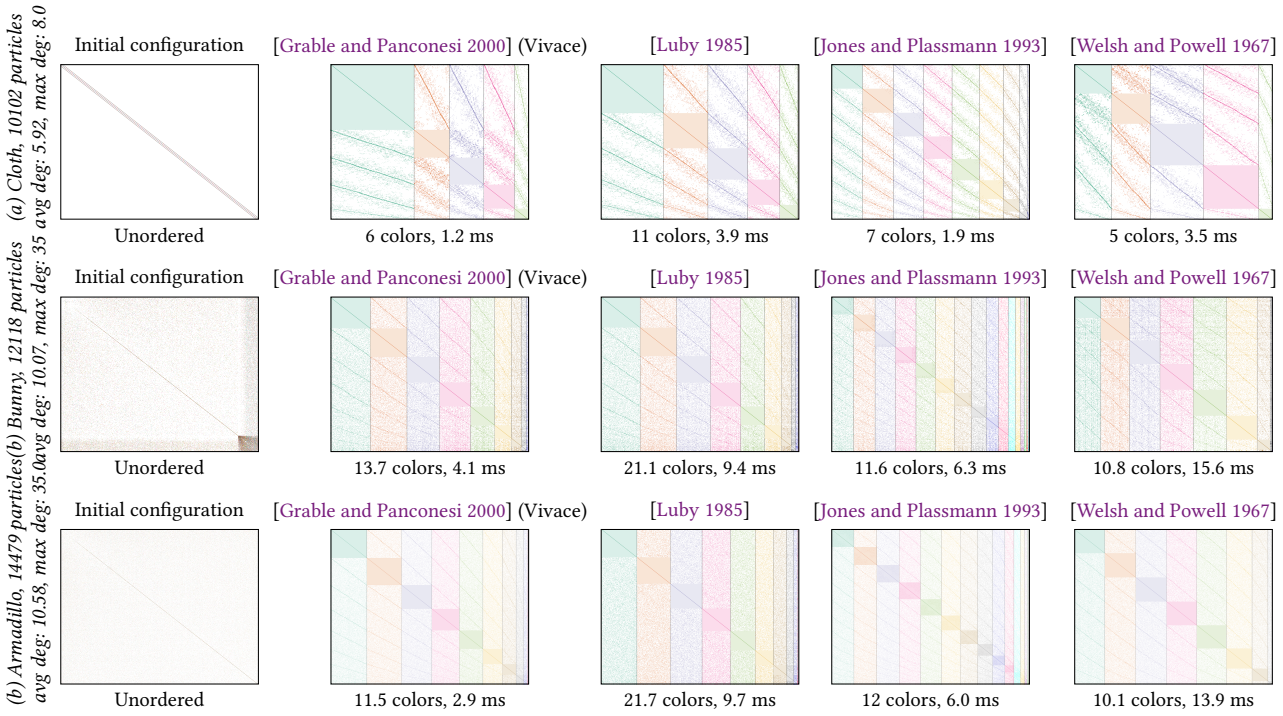


Fig. 21. Adjacency matrices corresponding to the constraint graphs. Leftmost column: initial configuration. Other columns: reordering induced by different coloring algorithms. Nodes belonging to the same color are independent from each other and can be solved in a single parallel step. The number of colors and computation time averaged over 30 frames is reported below each matrix.

for each round, so that there is a better chance of keeping low the number of used colors. In fact, LDF uses fewer

colors than the Jones-Plassmann algorithm; however the number of rounds to completely color the graph increases significantly.

In [Tonge et al. 2012], a greedy approach based on the Vizing's theorem [Vizing 1964] is used for reducing collision jittering of rigid bodies. Such coloring strategy leads to $2\Delta - 1$ colors, where Δ is the maximum degree of the graph, which in some cases can be high enough to compromise real-time performances (e.g., for the tetrahedral Armadillo used in our tests: $\Delta = 35$). In comparison, our approach extends the use of Gauss-Seidel not only to collision response but to the whole dynamics of deformable bodies, and it is faster because uses far less colors than Δ due to the shrinking factor s and the *feed the hungry* step (Alg. 5, steps 14-16). Graph coloring for collision handling has also been used in [Govindaraju et al. 2005], for meshes with rectangular connectivity, i.e. every vertex has four neighbors and every polygon is rectangular. Our approach can handle meshes without any restriction on connectivity.

Recently, [Naumov et al. 2015] presented a parallel coloring algorithm requiring a minimal amount of communication between the threads, making it particularly appealing for GPU implementations. This method performs better than the Jonas-Plassmann algorithm when incomplete colorings are acceptable (in this particular case if just 90% of the nodes need to be colored), making such an algorithm unsuitable for our purposes. Furthermore, this algorithm is based on custom hash functions which is not clear how to define in case of graphs derived from irregular meshes.

In [Fratarcangeli and Pellacini 2015], a method based on sequential coloring changes the topology of the graph in the initialization phase, in order to use a minimal number of colors. This method leads to balanced partitions but, being executed just at the beginning of the animation, it is not suitable for matrices varying over time. In contrast, our approach focuses on interactively changing sparse systems which require low overhead colorings.

Other approaches, such as the Smallest-Degree-First [Matula and Beck 1983], focus on how to provide better coloring rather than maximizing speed. And other good sequential algorithms, namely the Saturation-Degree-Ordering [Brélaz 1979] and Incidence-Degree-Ordering [Coleman and Moré 1983] algorithms, are not suitable for parallelization, and we do not consider them here.

Performance Comparison. We compared the randomized coloring algorithm employed in *Vivace* with other parallel coloring algorithms, namely Luby [Luby 1985], Jones-Plassmann (JP) [Jones and Plassmann 1993] and Largest-Degree-First (LDF) [Welsh and Powell 1967]. Unlike the existing literature, where the Brooks-Vizing colorings are tested on triangle- or square-free graphs, we applied the algorithms on graphs derived from irregular geometric models: 1) a triangulated cloth using a spring constraint for each edge, and 2) the Stanford Bunny and Armadillo using tetrahedral constraints for volume preservation. From each model, we generated different meshes with an increasing number of triangles and tetrahedrons, respectively.

The re-ordering induced by the coloring is depicted in the adjacency matrices in Fig. 21, which correspond to the matrix $\mathbf{A}$ in the linear system, and represent the topology of the constraints in the mesh before and after the coloring. The rows and columns represent the indices of the nodes. An element of the matrix m_{ij} is not empty if the corresponding nodes i and j are shared by a constraint. A single row of the matrix represents a constraint to be solved. The initial configuration is shown in the leftmost column, labeled as *unordered*.

By reordering the indices of the nodes in the graph according to their color, the elements are "pushed" away from the diagonal. Instead of solving for one unknown after the other as in the lexicographic Gauss-Seidel, all the unknowns belonging to the same color can be solved simultaneously in parallel.

We quantitatively compare algorithms with respect to the total time for coloring and number of colors (Fig. 22). We remind the reader that we seek a solution with the smallest number of colors, to increase parallelism, and the smallest

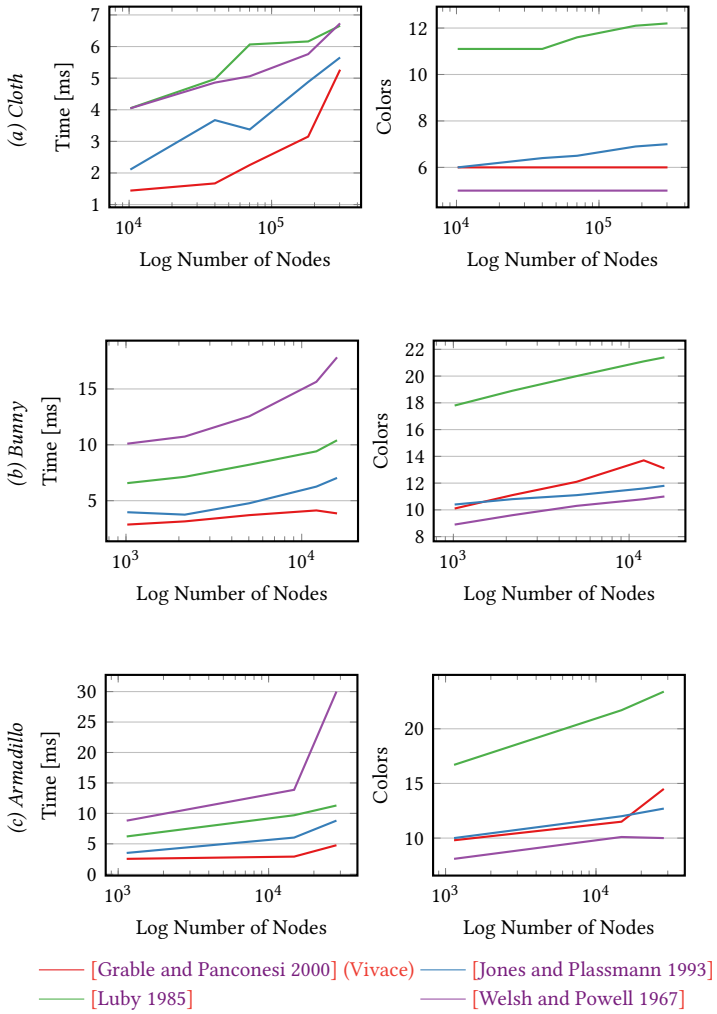


Fig. 22. Comparison of different coloring algorithms w.r.t. time to complete (*left column*), and number of used colors (*right column*).

computation time. As expected, colorings using [Welsh and Powell 1967] employ the smallest amount of colors but are 10-20 times slower than the other algorithms. *In general, our tests demonstrate that the random coloring employed in* Vivace *outperforms all the other algorithms in speed while using approximately the same number of colors.*

4.3 Results and Limitations

In this section, we provide a qualitative and quantitative assessment of *Vivace*. All algorithms have been fully implemented on the GPU using CUDA/c++, and run on an NVIDIA GeForce GTX 970.

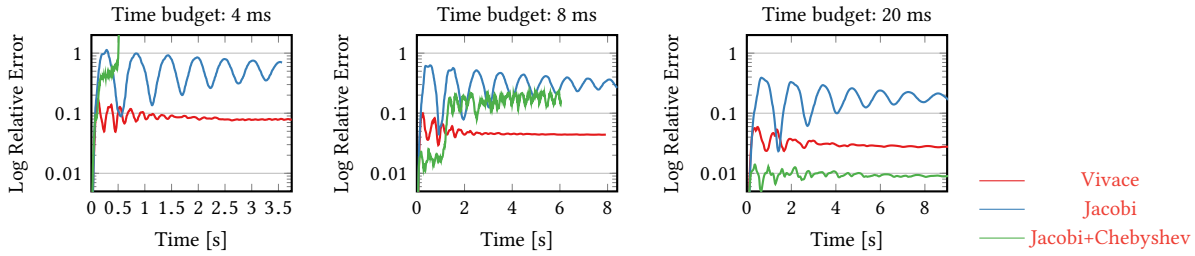


Fig. 23. Relative error vs. time in the *Tablecloth* animation in the accompanying video (triangulated model composed by 10K vertices, 20K triangles, 30K springs and 30K hinge-edge constraints; time step $h = 33$ms.). The convergence speed of our method (red curve) is compared with the Jacobi method (blue curve) and Jacobi+Chebyshev (green) using different time budgets.

Performance and stability. We have tested *Vivace* with respect to *stability, convergence speed and scalability*, and compared it with other two iterative solvers: the parallel Jacobi method [Macklin et al. 2014], and parallel Jacobi accelerated with the Chebyshev semi-iterative method, as presented in Sec. 2. For each experiment, we tested all the solvers with different time budgets, while the same conditions were used, i.e. same time step $h = 33$ms, same external forces, and same damping. The supplemental video shows the performed experiments.

The plots in Fig. 23 report the residual error over time for a triangulated cloth composed of 10K vertices and 20K triangles modeled with a spring constraint for each edge [Liu et al. 2013], and a hinge-edge constraint for each edge shared by two triangles [Bergou et al. 2006]. The accuracy of all the solvers increases with the number of iterations; however, the number of iterations must be relatively small in order to satisfy the time budgets. When the time budget is 4 or 8 ms, the residual error of the Jacobi solver is too big to be acceptable (blue curve), while Jacobi+Chebyshev diverges to an unstable state (green curve). One of the main practical benefits of our solver is its capability to provide stable solutions despite the low number of iterations, while being at least one order of magnitude more accurate than the other solvers (red curve). Fig. 25 compares the corresponding visual results.

The plots in Fig. 24 show the residual error over time for a volumetric armadillo composed of 55K tetrahedral constraints preserving volume and shape. Even in this case, the convergence speed of Jacobi and the stability of Chebyshev+Jacobi is not sufficient to provide reliable results in case of small time budgets (Fig. 26). However, interestingly, in case of larger time steps (25 ms) the convergence rate of *Vivace* is the same as that the Chebyshev solver presented in Sec. 2. In principle, the acceleration technique presented in Sec. 2 is orthogonal to our approach and can be used to speed-up the Gauss-Seidel solver.

Complex scenarios. The stability and the speed of *Vivace* enabled us to model more complex scenarios as depicted in Figures 27 and 28, where we demonstrate our solver's scalability by handling objects composed of a large number of interacting constraints. Here we modeled the system using Position-Based Dynamics. In Fig. 27 we animated a stack of deformable noodles, composed of 30K vertices, 52K triangles and 150K constraints falling on the floor and colliding with each other. In Fig. 28, we represent a heap of cloths falling on a static armchair. Each cloth is composed of 36K constraints, for a total number of 324K constraints. The armchair is modeled as a Signed Distance Field which is used for the collision tests. The scene is updated interactively at 30 fps, so that it is possible to pinch and drag the cloths. In these cases the topology of the graph changes due to collisions, so recoloring is necessary and *Vivace* handles this very well. We are not aware of any other existing solver able to handle such complex test cases within the considered time budgets.

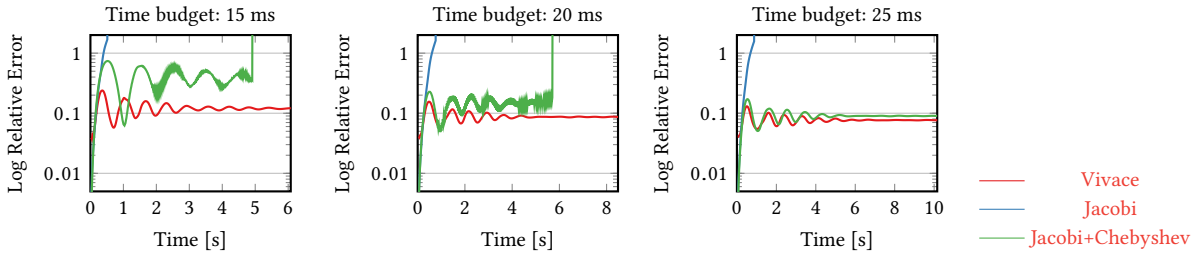


Fig. 24. Relative error vs. time in the *Armadillo* animation in the accompanying video (volumetric model composed by 10K vertices, 55K tetrahedral constraints; time step h = 33ms.). The convergence speed of our method (red curve) is compared with the Jacobi method (blue curve) and Jacobi+Chebyshev (green) using different time budgets.

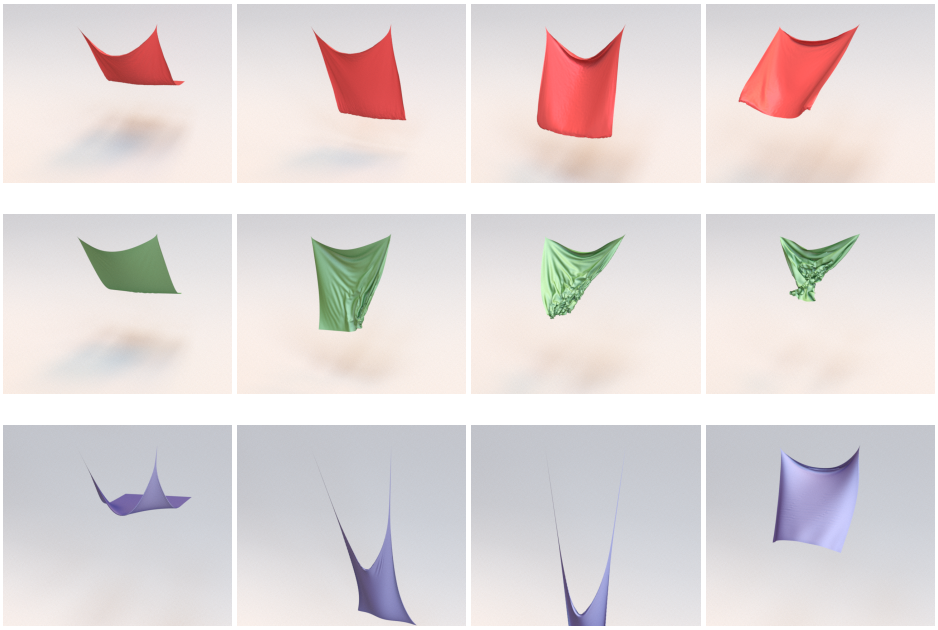


Fig. 25. Still frames from the animation corresponding to the plot in Fig. 23b. Solver time budget: 8 ms. Upper row: our method, middle row: Jacobi + Chebyshev, bottom row: Jacobi.

Limitations. *Vivace* has three main limitations. First, only a restricted number of iterations can be accommodated in the considered time budgets. Thus, *Vivace* can deliver only approximate solutions which, in some cases, may lead to artifacts; for example, in Fig. 25 the cloth flexes excessively near the anchor points due to the theoretical limits of the convergence speed of relaxation methods. The perceived animation is still visually acceptable in the domain of real-time applications (e.g., games). In case more accuracy is needed, it may be interesting to use our solver as a pre-conditioner in a parallel algebraic multigrid system (e.g., [Tamstorf et al. 2015]), for further accelerating the overall convergence speed. Second, while we have tested *Vivace* with deformable bodies, where the topology of the system changes due to collisions, we did not yet investigate its performance in handling fluids, where the constrained system

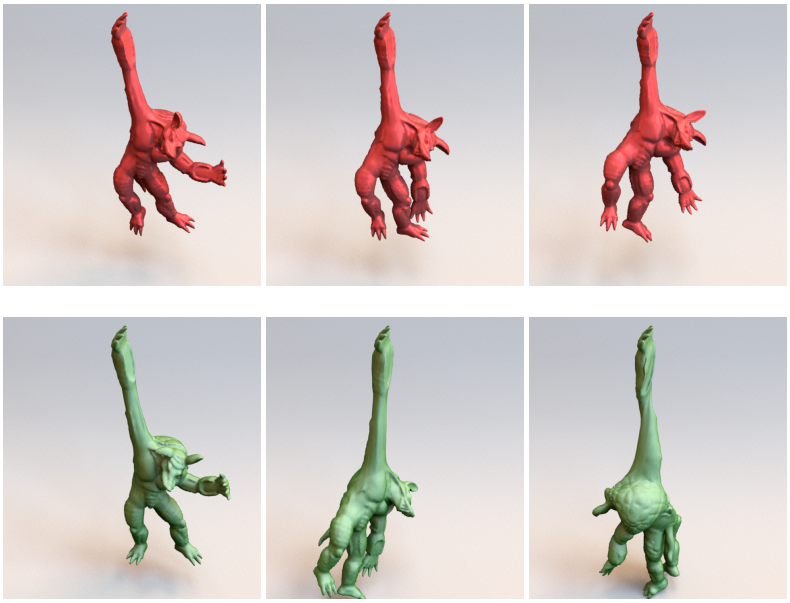


Fig. 26. Still frames from the animation corresponding to the plot in Fig. 24a. Solver time budget: 15 ms. Upper row: our method, bottom row: Jacobi + Chebyshev.

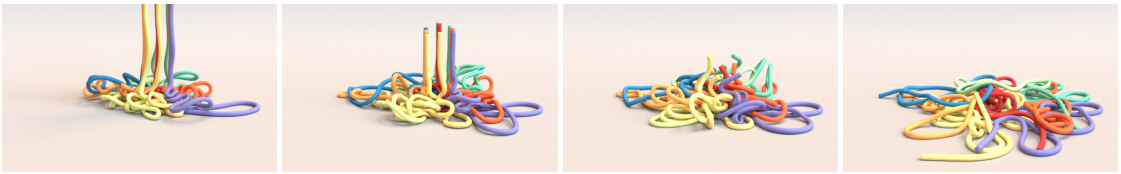


Fig. 27. Self-colliding noodles falling on the floor. 30K vertices, 52K elements, 150K constraints. Solver time budget including collision handling: 15ms per frame.

exhibits dramatic changes of topology for each frame. Third, the actual implementation of *Vivace* is more complex with respect to Jacobi-based solvers because it requires the graph coloring step. This additional complexity is mitigated by the simplicity of the randomized coloring algorithm.

REFERENCES

David Baraff and Andrew Witkin. 1998. Large steps in cloth simulation. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques (SIGGRAPH '98)*. ACM, New York, NY, USA, 43–54. https://doi.org/10.1145/280814.280821

Miklos Bergou, Max Wardetzky, David Harmon, Denis Zorin, and Eitan Grinspun. 2006. A quadratic bending model for inextensible surfaces. In *Proc. of SGP*. 227–230.

Sofien Bouaziz, Sebastian Martin, Tiantian Liu, Ladislav Kavan, and Mark Pauly. 2014. Projective Dynamics: Fusing Constraint Projections for Fast Simulation. *ACM Trans. Graph. (SIGGRAPH)* 33, 4, Article 154 (July 2014), 11 pages.

C. Bouby, D. Fortuné, W. Pietraszkiewicz, and C. Vallée. 2005. Direct determination of the rotation in the polar decomposition of the deformation gradient by maximizing a Rayleigh quotient. *Journal of Applied Mathematics and Mechanics* 85, 3 (March 2005), 155–162.

Daniel Brélaz. 1979. New Methods to Color the Vertices of a Graph. *Commun. ACM* 22, 4 (April 1979), 251–256. https://doi.org/10.1145/359094.359101

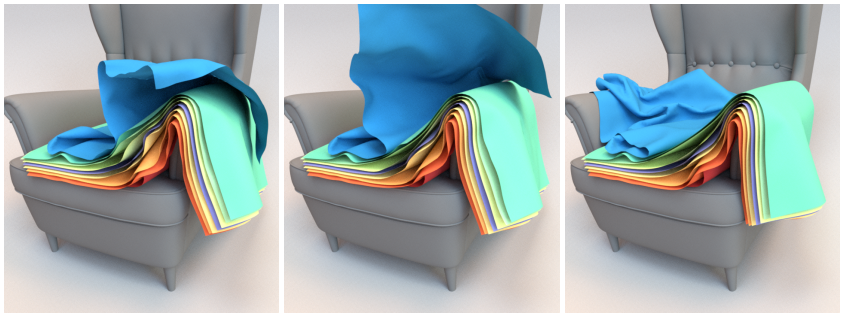


Fig. 28. Pinching and dragging a cloth on top of a heap. Solver time budget including collision handling: 15ms per frame.

Thomas F. Coleman and Jorge J. Moré. 1983. Estimation of Sparse Jacobian Matrices and Graph Coloring Problems. *SIAM J. Numer. Anal.* 20, 1 (1983), 187–209. https://doi.org/10.1137/0720013

Alicia Cordero, José L. Hueso, Eulalia Martínez, and Juan R. Torregrosa. 2010. New Modifications of Potra-PtáK's Method with Optimal Fourth and Eighth Orders of Convergence. *J. Comput. Appl. Math.* 234, 10 (Sept. 2010), 2969–2976.

Christian Dick, Joachim Georgii, and Rüdiger Westermann. 2011. A real-time multigrid finite hexahedra method for elasticity simulation using CUDA. *Simulation Modelling Practice and Theory* 19, 2 (2011), 801–816.

Yun Fei, Guodong Rong, Bin Wang, and Wenping Wang. 2014. Parallel L-BFGS-B algorithm on GPU. *Computers & Graphics* 40 (2014), 1 – 9. https://doi.org/10.1016/j.cag.2014.01.002

L.P. Franca. 1989. An algorithm to compute the square root of 3x3 positive definite matrix. *Computers. Math. Applic.* 18, 5 (1989), 459–466.

Marco Fratarcangeli and Fabio Pellacini. 2015. Scalable Partitioning for Parallel Position Based Dynamics. *Computer Graphics Forum (Eurographics)* 34, 2 (May 2015), 405–413. https://doi.org/10.1111/cgf.12570

Marco Fratarcangeli, Valentina Tibaldo, and Fabio Pellacini. 2016. Vivace: a Practical Gauss-Seidel Method for Stable Soft Body Dynamics. *ACM Trans. Graph. Article* 35, 214 (2016). https://doi.org/10.1145/2980179.2982437

Y.-C. Fung. 1993. *Biomechanics: Mechanical properties of living tissues.* Springer-Verlag.

Michael R. Garey and David S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA.

Akash Garg, Eitan Grinspun, Max Wardetzky, and Denis Zorin. 2007. Cubic shells. In *Proc. of SCA*. 91–98.

Gene H. Golub and Charles F. Van Loan. 1996. *Matrix Computations (3rd Ed.).* Johns Hopkins University Press, Baltimore, MD, USA.

Gene H. Golub and Richard S. Varga. 1961. Chebyshev semi-iterative methods, successive overrelaxation iterative methods, and second order Richardson iterative methods. *Numer. Math.* 3, 1 (1961), 157–168.

Naga K. Govindaraju, David Knott, Nitin Jain, Ilknur Kabul, Rasmus Tamstorf, Russell Gayle, Ming C. Lin, and Dinesh Manocha. 2005. Interactive Collision Detection Between Deformable Models Using Chromatic Decomposition. *ACM Trans. Graph.* 24, 3 (July 2005), 991–999. https://doi.org/10.1145/1073204.1073301

David A Grable and Alessandro Panconesi. 2000. Fast Distributed Algorithms for BrooksâĂŞVizing Colorings. *Journal of Algorithms* 37, 1 (2000), 85 – 120. https://doi.org/10.1006/jagm.2000.1097

Martin H. Gutknecht and Stefan Röllin. 2002. The Chebyshev Iteration Revisited. *Parallel Comput.* 28, 2 (Feb. 2002), 263–283.

G. Irving, J. Teran, and R. Fedkiw. 2004. Invertible Finite Elements for Robust Simulation of Large Deformation. In *Proceedings of SCA*. 131–140.

Mark T. Jones and Paul E. Plassmann. 1993. A Parallel Graph Coloring Heuristic. *SIAM J. Sci. Comput.* 14, 3 (May 1993), 654–669. https://doi.org/10.1137/0914041

Tiantian Liu, Adam W. Bargteil, James F. O'Brien, and Ladislav Kavan. 2013. Fast Simulation of Mass-spring Systems. *ACM Trans. Graph. (SIGGRAPH Asia)* 32, 6, Article 214 (Nov. 2013), 7 pages.

M Luby. 1985. A Simple Parallel Algorithm for the Maximal Independent Set Problem. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing (STOC '85)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/22145.22146

Miles Macklin, Matthias Müller, Nuttapong Chentanez, and Tae-Yong Kim. 2014. Unified Particle Physics for Real-time Applications. *ACM Trans. Graph. (SIGGRAPH)* 33, 4, Article 153 (July 2014), 12 pages.

C. W. Macosko. 1994. *Rheology: Principles, measurement and applications.* VCH Publishers.

David W. Matula and Leland L. Beck. 1983. Smallest-last Ordering and Clustering and Graph Coloring Algorithms. *J. ACM* 30, 3 (July 1983), 417–427. https://doi.org/10.1145/2402.322385

Aleka McAdams, Andrew Selle, Rasmus Tamstorf, Joseph Teran, and Eftychios Sifakis. 2011a. *Computing the Singular Value Decomposition of 3x3 matrices with minimal branching and elementary floating point operations.* Technical report. University of Wisconsin - Madison.

Aleka McAdams, Yongning Zhu, Andrew Selle, Mark Empey, Rasmus Tamstorf, Joseph Teran, and Eftychios Sifakis. 2011b. Efficient Elasticity for Character Skinning with Contact and Collisions. *ACM Trans. Graph. (SIGGRAPH)* 30, 4, Article 37 (July 2011), 12 pages.

Matthias Müller. 2008. Hierarchical Position Based Dynamics. In *Proceedings of VRIPHYS*. 1–10.

Matthias Müller, Bruno Heidelberger, Marcus Hennix, and John Ratcliff. 2007. Position Based Dynamics. *J. Vis. Comun. Image Represent.* 18, 2 (April 2007), 109–118.

Maxim Naumov, Patrice Castonguay, and Jonathan Cohen. 2015. *Parallel Graph Coloring with Applications to the Incomplete-LU Factorization on the GPU.* Tech Report. NVIDIA.

Yurii Nesterov. 2004. *Introductory lectures on convex optimization: A basic course.* Kluwer Academic Publ., Boston, Dordrecht, London.

Brendan O'donoghue and Emmanuel Candès. 2015. Adaptive Restart for Accelerated Gradient Schemes. *Found. Comput. Math.* 15, 3 (June 2015), 715–732.

R. W. Ogden. 1997. *Non-linear elastic deformations.* Dover Publications, Inc.

Taylor Patterson, Nathan Mitchell, and Eftychios Sifakis. 2012. Simulation of Complex Nonlinear Elastic Bodies Using Lattice Deformers. *ACM Trans. Graph. (SIGGRAPH Asia)* 31, 6, Article 197 (Nov. 2012), 10 pages.

Alec R. Rivers and Doug L. James. 2007. FastLSM: Fast Lattice Shape Matching for Robust Real-time Deformation. *ACM Trans. Graph. (SIGGRAPH)* 26, 3, Article 82 (July 2007).

Y. Saad. 2003. *Iterative Methods for Sparse Linear Systems* (2nd ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

Rasmus Tamstorf, Toby Jones, and Stephen F. McCormick. 2015. Smoothed Aggregation Multigrid for Cloth Simulation. *ACM Trans. Graph.* 34, 6, Article 245 (Oct. 2015), 13 pages. https://doi.org/10.1145/2816795.2818081

Joseph Teran, Eftychios Sifakis, Geoffrey Irving, and Ronald Fedkiw. 2005. Robust Quasistatic Finite Elements and Flesh Simulation. In *Proceedings of SCA*. 181–190.

Bernhard Thomaszewski, Simon Pabst, and Wolfgang Strasser. 2009. Continuum-based Strain Limiting. *Computer Graphics Forum (Eurographics)* 28, 2 (2009), 569–576.

Richard Tonge, Feodor Benevolenski, and Andrey Voroshilov. 2012. Mass Splitting for Jitter-free Parallel Rigid Body Simulation. *ACM Trans. Graph.* 31, 4, Article 105 (July 2012), 8 pages. https://doi.org/10.1145/2185520.2185601

V. G. Vizing. 1964. On an estimate of the chromatic class of a p-graph. (Russian). *Diskret. Analiz.* 3 (1964), 25–30.

Huamin Wang. 2015. A Chebyshev Semi-iterative Approach for Accelerating Projective and Position-based Dynamics. *ACM Trans. Graph. (SIGGRAPH Asia)* 34, 6, Article 246 (Oct. 2015), 9 pages.

Huamin Wang, James O'Brien, and Ravi Ramamoorthi. 2010. Multi-resolution isotropic strain limiting. *ACM Trans. Graph. (SIGGRAPH Asia)* 29, 6, Article 156 (Dec. 2010), 156:1–156:10 pages.

Huamin Wang and Yin Yang. 2016. Descent methods for elastic body simulation on the GPU. *ACM Transactions on Graphics (TOG)* 35, 6 (2016), 212.

D. J. A. Welsh and M. B. Powell. 1967. An upper bound for the chromatic number of a graph and its application to timetabling problems. *Comput. J.* 10, 1 (1967), 85–86. https://doi.org/10.1093/comjnl/10.1.85 arXiv:http://comjnl.oxfordjournals.org/content/10/1/85.full.pdf+html

Hongyi Xu, Funshing Sin, Yufeng Zhu, and Jernej Barbič. 2015. Nonlinear Material Design Using Principal Stretches. *ACM Trans. Graph. (SIGGRAPH)* 34, 4, Article 75 (July 2015), 11 pages.

Yongning Zhu, Eftychios Sifakis, Joseph Teran, and Achi Brandt. 2010. An Efficient Multigrid Method for the Simulation of High-resolution Elastic Solids. *ACM Trans. Graph.* 29, 2, Article 16 (April 2010), 16:1–16:18 pages.