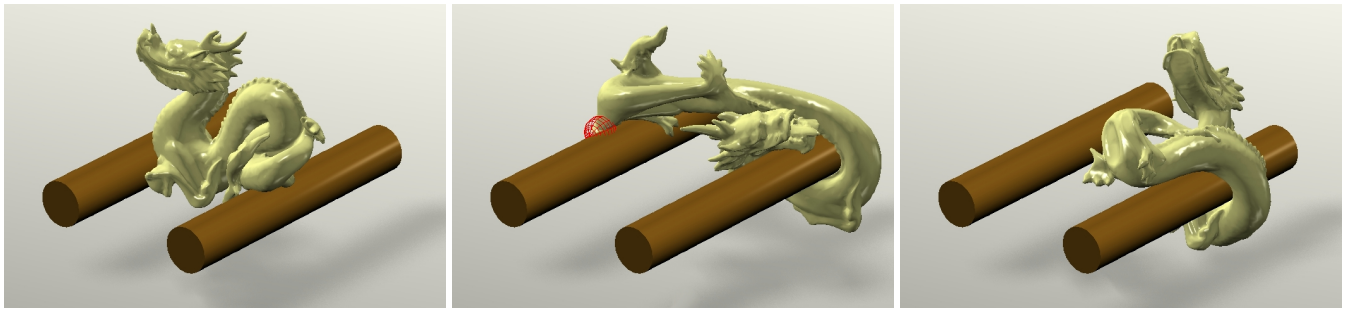
Descent Methods for Elastic Body Simulation on the GPU

Huamin Wang* The Ohio State University

Yin Yang* The University of New Mexico



(a) A resting dragon (b) A stretched dragon (c) A spiral dragon

Figure 1: *The dragon example. This model contains 16K vertices and 58K tetrahedra. Our elastic body simulator animates this example on the GPU at 30.5FPS, under the Mooney-Rivlin model. Thanks to a series of techniques we developed in this paper, the simulator can robustly handle very large time steps (such as $h = 1/30s$) and deformations.*

Abstract

We show that many existing elastic body simulation approaches can be interpreted as descent methods, under a nonlinear optimization framework derived from implicit time integration. The key question is how to find an effective descent direction with a low computational cost. Based on this concept, we propose a new gradient descent method using Jacobi preconditioning and Chebyshev acceleration. The convergence rate of this method is comparable to that of L-BFGS or nonlinear conjugate gradient. But unlike other methods, it requires no dot product operation, making it suitable for GPU implementation. To further improve its convergence and performance, we develop a series of step length adjustment, initialization, and invertible model conversion techniques, all of which are compatible with GPU acceleration. Our experiment shows that the resulting simulator is simple, fast, scalable, memory-efficient, and robust against very large time steps and deformations. It can correctly simulate the deformation behaviors of many elastic materials, as long as their energy functions are second-order differentiable and their Hessian matrices can be quickly evaluated. For additional speedups, the method can also serve as a complement to other techniques, such as multi-grid.

Keywords: Nonlinear optimization, Jacobi preconditioning, the Chebyshev method, GPU acceleration, hyperelasticity.

Concepts: •**Computing methodologies** → **Physical simulation;**

*e-mail: whmin@cse.ohio-state.edu; yangy@unm.edu

SA '16 Technical Papers,, December 05-08, 2016, , Macao
ISBN: 978-1-4503-4514-9/16/12
DOI: http://dx.doi.org/10.1145/2980179.2980236

ACM Reference Format
Wang, H., Yang, Y. 2016. Descent Methods for Elastic Body Simulation on the GPU. ACM Trans. Graph. 35, 6, Article 212 (November 2016), 10 pages. DOI = 10.1145/2980179.2980236
http://doi.acm.org/10.1145/2980179.2980236.

1 Introduction

Solid materials often exhibit complex elastic behaviors in the real world. While we have seen a variety of models being developed to describe these behaviors over the past few decades, our ability to simulate them computationally is rather limited. Early simulation techniques often use explicit time integration, which is known for its numerical instability problem. A typical solution is to use implicit time integration instead. Given the nonlinear force-displacement relationship of an elastic material, we can formulate implicit time integration into a nonlinear system. Baraff and Witkin [1998] proposed to linearize this system at the current shape and solve the resulting linear system at each time step. Their method is equivalent to running one iteration of Newton's method. Alternatively, we can linearize the system at the rest shape and solve a linear system with a constant matrix. While this method is fast thanks to matrix pre-factorization, the result becomes unrealistic under large deformation. To address this issue, Müller and Gross [2004] factored out the rotational component from the displacement and ended up with solving a new linear system at each time step again. Even if we accept formulating elastic simulation into a linear system, we still face the challenge of solving a large and sparse system. Unfortunately, most linear solvers are not fully compatible with parallelization and they cannot be easily accelerated by the GPU.

In recent years, graphics researchers studied the use of geometric constraints and developed a number of constraint-based simulation techniques, such as strain limiting [Provot 1996; Thomaszewski et al. 2009; Wang et al. 2010], position-based dynamics [Müller et al. 2007; Müller 2008; Kim et al. 2012], and shape matching [Müller et al. 2005; Rivers and James 2007]. While these techniques are easy to implement and compatible with GPU acceleration, they offer little control on their underlying elastic models. To solve this problem, Liu and collaborators [2013] and Bouaziz and colleagues [2014] described geometric constraints as elastic energies in a quadratic form. This implies that their technique, known as *projective dynamics*, is not suitable for simulating generic elastic models. The recent work by Tournier and colleagues [2015] proposed to incorporate both elastic forces and compliant constraints into a single linear system. This technique is designed

for highly stiff problems, where the condition number is more important than the problem size. Additionally, it must solve a new linear system in each time step.

We think 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.** Given sufficient computational resources, a good method should correctly simulate the behavior of a specified elastic model. 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.

Although existing elastic body simulation techniques can satisfy one or two of these requirements, none of them can satisfy all of the three, as far as we know. To develop a fast, flexible, and correct elastic body simulator, we made a series of technical contributions in this paper.

- **Insights.** We demonstrate that many recent simulation methods, including position-based dynamics, projective dynamics and its accelerated version, can be viewed as descent methods under an energy minimization framework. The main question here is how to find the descent direction, which differs in these methods.
- **Algorithm.** We propose to couple Jacobi preconditioning and Chebyshev acceleration with the gradient descent method. Our method offers a high convergence rate with a low computational cost. To further improve the performance of our method, we develop a number of techniques for step length adjustment, Chebyshev parameters, and initialization. The method is fully compatible with GPU acceleration.
- **Elastic model.** Many hyperelastic models were not designed for highly compressed or even inverted cases. To address this issue, we present a hybrid elastic model by mixing hyperelastic energy with projective dynamics energy. Our method can efficiently simulate this model, by interpolating forces and Hessian matrices on the fly.

In summary, our descent method handles 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 can be satisfied by many elastic models, including 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]. Given a sufficient number of iterations, our 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 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 [Zhu et al. 2010; McAdams et al. 2011b; Dick et al. 2011; Patterson et al. 2012] at this time.

2 Related Work

Early elastic body simulation techniques often use explicit time integration schemes, which are easy to implement but require small time steps to avoid numerical instability. To simulate cloth and thin shells using large time steps, Baraff and Witkin [1998] advocated the use of implicit time integration schemes. If we assume that elastic force is a linear function of vertex displacement, the implicit Euler scheme forms a linear system with a constant matrix, which can be pre-factorized for fast linear solve. Since linear elastic force is not rotation-invariant, it can cause unrealistic volume growth when an object is under large rotation. Müller and Gross [2004] alleviated this problem by factoring out the rotational component in their co-rotational method. For more accurate simulation of real-world elastic bodies, we must use nonlinear elastic force and form the implicit scheme into a nonlinear system. A typical solution to a nonlinear system is Newton's method, which needs a large computational cost to evaluate the Hessian matrix and solve a linearized system in every iteration. Teran and colleagues [2005] developed a technique to evaluate the Hessian matrix under a hyperelastic model, so they can use the implicit scheme to handle hyperelastic bodies. Although the implicit scheme is more numerically stable, it suffers from artificial damping. To overcome this issue, Kharevych and colleagues [2006] suggested to use symplectic integrators. Hybrid implicit-explicit integration is another technique for reducing artificial damping, as Bridson and collaborators [2003] and Stern and Grinspun [2009] demonstrated. For a mass-spring system, Su and colleagues [2013] investigated how to track and preserve the total system energy over time. Daviet and collaborators [2011] studied the development of a fast iterative solver for handling Coulomb friction in hair dynamics.

The force-displacement relationship of a real-world elastic material, such as human skin, is often highly nonlinear. This nonlinearity makes the material difficult and expensive to handle in physics-based simulation. A simple way to generate nonlinear effects without using an elastic model is to apply geometric constraints on springs [Provot 1996], or triangular and tetrahedral elements [Thomaszewski et al. 2009; Wang et al. 2010]. Müller and colleagues [2007; 2008; 2012] pushed this idea even further, by using geometric constraints to replace elastic forces in a mass-spring system. Later they extended this position-based method to simulate fluids [Macklin and Müller 2013; Macklin et al. 2014] and deformable bodies [Müller et al. 2014]. Similar to position-based method, shape matching [Müller et al. 2005; Rivers and James 2007] also uses the difference between deformed shapes and rest shapes to simulate elastic behaviors. Instead of using geometric constraints, Perez and collaborators [2013] applied energy constraints to produce nonlinear elastic effects.

An interesting question is whether there is a connection between an elastic model and a geometric constraint. Liu and collaborators [2013] found that the elastic spring energy can be treated as a compliant constraint. Based on this observation, they developed an implicit mass-spring simulator, which iteratively solves a local constraint enforcement step and a global linear system step. Bouaziz and colleagues [2014] formulated this method into projective dynamics, by defining the elastic energy of a triangular or tetrahedral element as a constraint. The main advantage of projective dynamics is that the matrix involved in the global step is constant, so it can be pre-factorized for fast solve. On the GPU, Wang [2015] proposed to solve projective dynamics by the Jacobi method and the Chebyshev semi-iterative method, both of which are suitable for parallelization. Recently, Tournier and colleagues [2015] presented a stable way to solve elastic forces and compliant constraints together using a single linear system. Their method reduces the condition number, but increases the system size.

Algorithm 1 `Descent_Optimization`

Initialize $\mathbf{q}^{(0)}$;
for $k = 0...K-1$ **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 $\mathbf{q}^{(K)}$;

3 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:

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

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:

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

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: $\mathbf{q}_{t+1} = \arg\min \epsilon(\mathbf{q})$,

$$\epsilon(\mathbf{q}) = \frac{1}{2h^2}(\mathbf{q} - \mathbf{q}_t - h\mathbf{v}_t)^{\mathsf{T}}\mathbf{M}(\mathbf{q} - \mathbf{q}_t - h\mathbf{v}_t) + E(\mathbf{q}), \tag{3}$$

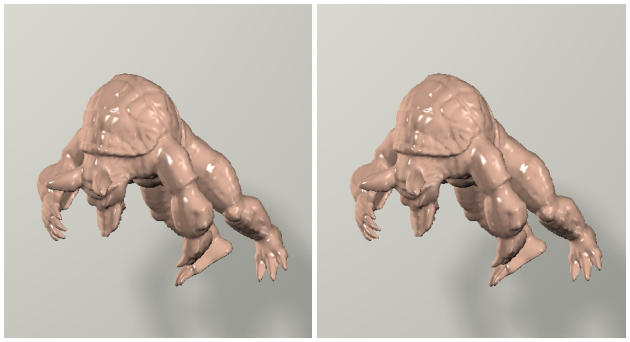
Nonlinear optimization is often solved by descent methods, which contain four steps in each iteration as Algorithm 1 shows. The main difference is in how to calculate the descent direction, which typically involves the use of the gradient: $\mathbf{g}^{(k)} = \nabla\epsilon(\mathbf{q}^{(k)})$.

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 2c. 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.

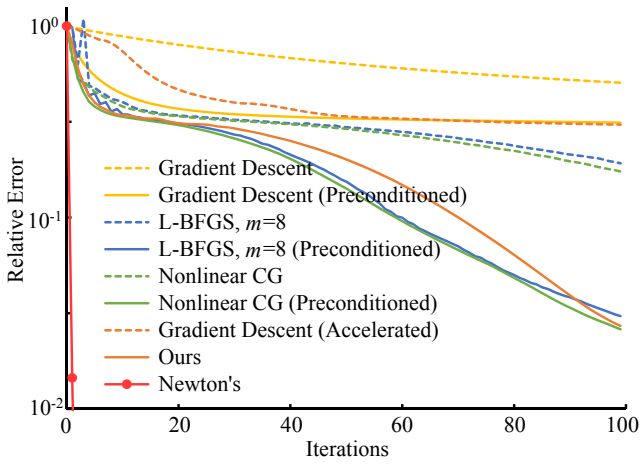
Newton's method. To achieve quadratic convergence, Newton's method approximates $\epsilon(\mathbf{q}^{(k)})$ by a quadratic function and it calculates the search direction[1] as: $\Delta\mathbf{q}^{(k)} = -(\mathbf{H}^{(k)})^{-1}\mathbf{g}^{(k)}$, where $\mathbf{H}^{(k)}$ is the Hessian matrix of $\epsilon(\mathbf{q})$ evaluated at $\mathbf{q}^{(k)}$. Figure 2c 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 2, 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].

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



(a) Ground truth (b) Our result



(c) The convergence plot

Figure 2: *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* $(\epsilon(\mathbf{q}^{(k)}) - \epsilon(\mathbf{q}^*))/(\epsilon(\mathbf{q}^{(k)}) - \epsilon(\mathbf{q}^{(0)}))$*, where* $\mathbf{q}^{(k)}$ *is the result in the k-th iteration and* $\mathbf{q}^*$ *is the ground truth.*

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:

$$\Delta\mathbf{q}^{(k)} = -\mathbf{g}^{(k)} + \frac{z^{(k)}}{z^{(k-1)}}\Delta\mathbf{q}^{(k-1)}, \quad z^{(k)} = \mathbf{g}^{(k)}\cdot\mathbf{g}^{(k)}. \tag{4}$$

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.

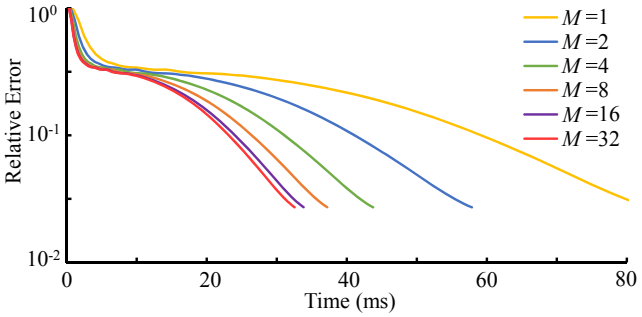


Figure 3: *The convergence of our method within 96 iterations, when using different M values to delay matrix evaluation. This plot shows that the method can reach the same residual error, regardless of M. Therefore we can use a larger M to reduce the matrix evaluation cost and improve the system performance.*

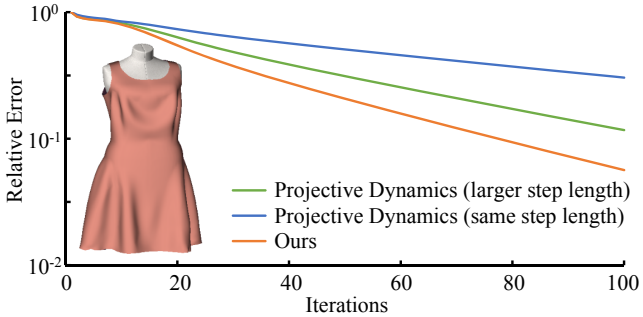


Figure 4: *The convergence of our method and projective dynamics. Although projective dynamics can use a large step length, it cannot converge as fast as our method.*

4 Our Descent Method

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

4.1 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:

$$\bar{\mathbf{q}} = \arg\min \epsilon(\mathbf{P}^{-1/2}\bar{\mathbf{q}}), \quad \text{for } \bar{\mathbf{q}} = \mathbf{P}^{1/2}\mathbf{q}, \tag{5}$$

where $\mathbf{P}$ is the preconditioner matrix. Mathematically, doing this is equivalent[2] to replacing $\mathbf{g}^{(k)}$ by $\mathbf{P}^{-1}\mathbf{g}^{(k)}$ in Equation 4. 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: $\mathbf{P} = \mathtt{diag}(\mathbf{H})$, 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 $\mathbf{P}(\mathbf{q}^{(k)}) = \mathtt{diag}(\mathbf{H}^{(k)})$ 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 2c shows.

This Jacobi preconditioner can be effectively applied to L-BFGS and gradient descent as well. Preconditioning in L-BFGS is essentially defining $\mathtt{diag}^{-1}(\mathbf{H}^{(k)})$ as the initial inverse Hessian estimate. Meanwhile, preconditioned gradient descent simply defines its new descent direction as: $\Delta\mathbf{q}^{(k)} = -\mathtt{diag}^{-1}(\mathbf{H}^{(k)})\mathbf{g}^{(k)}$. 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 2 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 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:

$$\epsilon(\mathbf{q}^{(k)} + \alpha^{(k)}\Delta\mathbf{q}^{(k)}) \le \epsilon(\mathbf{q}^{(k)}) + \alpha^{(k)}\Delta\mathbf{q}^{(k)} \cdot \mathbf{g}^{(k)} + \frac{B}{2}\left\|\alpha^{(k)}\Delta\mathbf{q}^{(k)}\right\|_2^2. \tag{6}$$

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 3 demonstrates that doing this has little effect on the convergence rate, but significantly reduces the computational cost per iteration, when $M \le 32$. 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 recent projective dynamics technique [Liu et al. 2013; Bouaziz et al. 2014] 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.

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

Projective dynamics was originally not suitable for GPU acceleration. Wang [2015] addressed this problem by removing off-diagonal entries of the preconditioner matrix. In this regard, that method is highly related to our method. Since both methods can handle mass-spring systems, we compare their convergence rates as shown in Figure 4. 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 2 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 2c 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.

4.2 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:

$$\epsilon(\mathbf{q}^{(k)} + \alpha^{(k)}\Delta\mathbf{q}^{(k)}) < \epsilon(\mathbf{q}^{(k)}) + c^{(k)}\alpha^{(k)}\Delta\mathbf{q}^{(k)} \cdot \mathbf{g}^{(k)}, \tag{7}$$

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: $\epsilon(\mathbf{q}^{(k)} + \alpha^{(k)}\Delta\mathbf{q}^{(k)}) < \epsilon(\mathbf{q}^{(k)})$. 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 condition still cannot be satisfied after the step length reaches a minimum value, we end the time step and start the next one.

4.3 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 5. 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[3] 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. When Wang [2015] adopted the Chebyshev method for accelerating projective dynamics, he defined the gradient decrease rate ρ as a constant:

$$\rho \approx \left\|\nabla\epsilon(\mathbf{q}^{k+1})\right\| \Big/ \left\|\nabla\epsilon(\mathbf{q}^{k})\right\|. \tag{8}$$

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 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

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

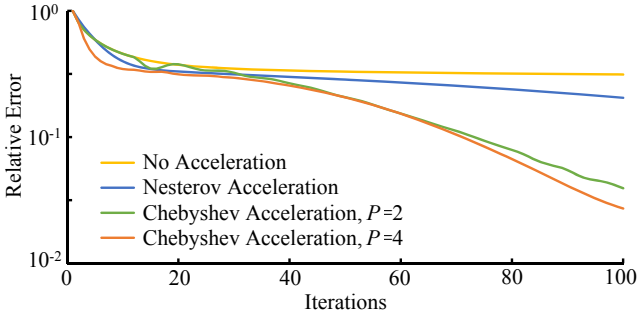


Figure 5: *The convergence of our method with different acceleration techniques. By using multiple phases, the Chebyshev method can more effectively accelerate the convergence process.*

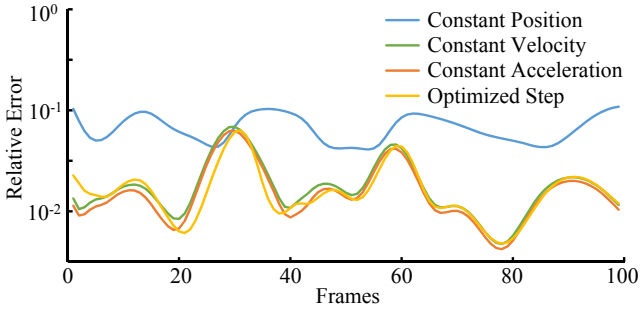


Figure 6: *The convergence of our method using different initialization approaches. This plot shows that the constant acceleration approach works the best in most cases.*

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:

$$\rho^{\text{new}} = 1 - (1 \pm \epsilon)(1 - \rho), \tag{9}$$

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 5 compares the convergence rates of our method, by using two and four Chebyshev phases respectively.

4.4 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 $\epsilon(\mathbf{q}_t + d\mathbf{v}_t)$:

$$d = \arg\min_d \left\{ \epsilon(\mathbf{q}_t) + (d\mathbf{v}_t) \cdot \nabla\epsilon(\mathbf{q}_t) + \frac{1}{2}(d\mathbf{v}_t) \cdot \mathbf{H}(\mathbf{q}_t)(d\mathbf{v}_t) \right\}, \tag{10}$$

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 6 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 6 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.

5 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.

5.1 Hyperelasticity

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 $\mathbf{C} = \mathbf{F}^{\mathrm{T}}\mathbf{F}$:

$$I = \mathtt{tr}(\mathbf{C}), \quad II = \mathtt{tr}(\mathbf{C}^2), \quad III = \mathtt{det}(\mathbf{C}). \tag{11}$$

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

$$W = \frac{s_0}{2}(I - 3)^2 + \frac{s_1}{4}(II - 2I + 3), \tag{12}$$

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:

$$W = s_0(III^{-1/3} \cdot I - 3) + s_1(III^{-1/2} - 1), \tag{13}$$

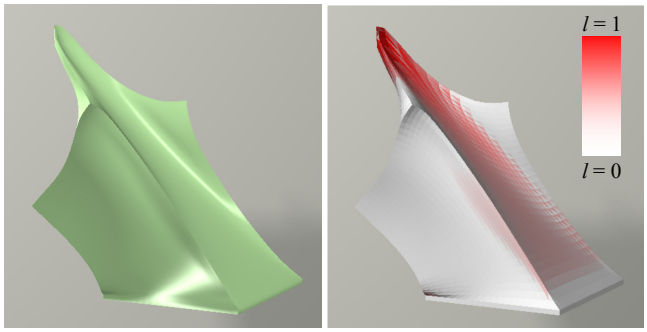
in which s_0 is the shear modulus and s_1 is the bulk modulus. Many hyperelastic models can be considered as extensions of the neo-Hookean model. For example, the compressible Mooney-Rivlin model for rubber-like materials uses the following strain energy density function [Macosko 1994]:

$$W = s_0(III^{-1/3} \cdot I - 3) + s_1(III^{-1/2} - 1) + s_2\left(\tfrac{1}{2}III^{-2/3}(I^2 - II) - 3\right). \tag{14}$$

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

$$W = s_0(III^{-1/3} \cdot I - 3) + s_1(III^{-1/2} - 1) + s_2\left(e^{s_3(III^{-1/3} \cdot I - 3)} - 1\right), \tag{15}$$

in which s_3 controls the speed of the exponential growth.



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

Figure 7: *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.*

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:

$$W^{\text{proj}} = \sum_{i=1}^{3} (\lambda_i - 1)^2, \tag{16}$$

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

$$l_t^{(k)} = \min\left(1, \max\left(0, l_t^{(k-1)} - L, \max_i (\lambda^+ - \lambda_i)/(\lambda^+ - \lambda^-)\right)\right), \tag{17}$$

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 17 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:

$$W_t^{\text{hybrid}} = \left(1 - l_t^{(k)}\right) W_t + l_t^{(k)} W_t^{\text{proj}}, \tag{18}$$

where W_t is the hyperelastic energy density of element t. According to Equation 18, we calculate the total contribution of element t to the Jacobi preconditioner as:

$$\mathbf{P}_t(\mathbf{q}^{(k)}) = \texttt{diag}\left((1 - l_t^{(k)})\mathbf{H}_t^{(k)} + l_t^{(k)} \mathbf{A}_t^{\mathsf{T}} \mathbf{A}_t\right), \tag{19}$$

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 19, thanks to the structural similarity between our algorithm and GPU-based projective dynamics developed by Wang [2015]. 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 7 shows.

Name	#vert	#ele	CPU Cost	GPU Cost	GPU FPS
Dragon (Fig. 1)	16K	58K	1.35s	32.8ms	30.5
Armadillo (Fig. 2)	15K	55K	1.28s	31.4ms	31.8
Box (Fig. 7)	14K	72K	1.47s	37.6ms	26.6
Dress (Fig. 4)	15K	44K	0.29s	26.6ms	37.6
Double helix (Fig. 9)	13K	41K	0.98s	27.5ms	36.4
Double helix (Fig. 9)	24K	82K	1.91s	38.5ms	26.0
Double helix (Fig. 9)	48K	158K	3.86s	65.4ms	15.3
Double helix (Fig. 9)	96K	316K	7.78s	122ms	8.2

Table 1: *Statistics and timings of our examples. By default, the CPU costs are evaluated with OpenMP enabled.*

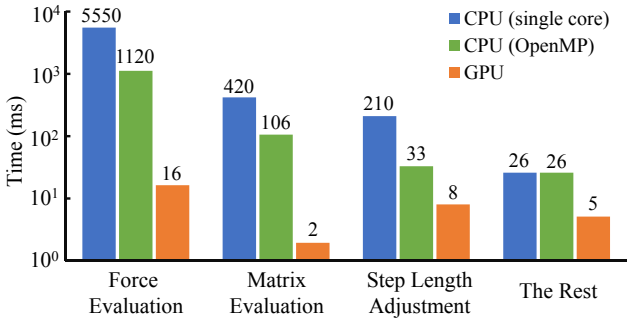
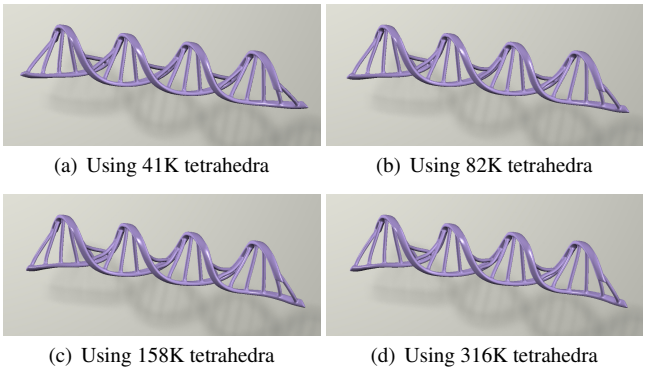


Figure 8: *The costs of the computational steps under three different implementations. This plot illustrates that the force evaluation step is the most expensive one.*

6 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 4.3. The statistics and the timings of our examples are provided in Table 1. 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.

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 matri-



(a) Using 41K tetrahedra (b) Using 82K tetrahedra

(c) Using 158K tetrahedra (d) Using 316K tetrahedra

Figure 9: *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.*

ces, 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.

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

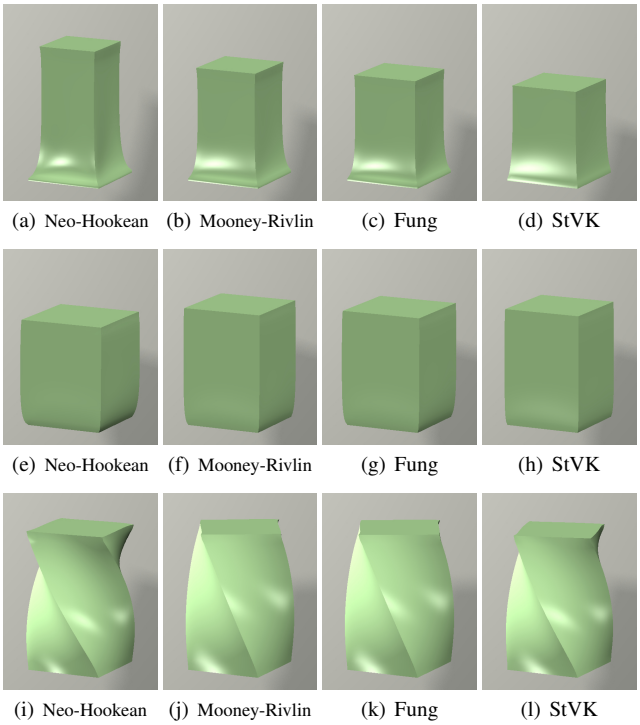
$$\mathbf{f}^{\text{air}}(\mathbf{q}) = -\frac{c}{h}(\mathbf{q} - \mathbf{q}_t), \tag{20}$$

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 8 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 4.1. 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 8.

To reveal the scalability of our algorithm, we simulate a double helix example at four resolutions. Table 1 shows that the computational cost is almost linearly proportional to the number of tetrahedra as expected. The high-resolution result in Figure 9d 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.



(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

Figure 10: *The box example. Our simulator can robustly and efficiently simulate the stretching, compression, and twisting behaviors of a box, under different hyperelastic models.*

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 10. 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 11 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.

Limitations. Our method can effectively handle high stiffness and high nonlinearity, at the expense of a lower convergence rate. If the method does not use enough iterations, it can cause various

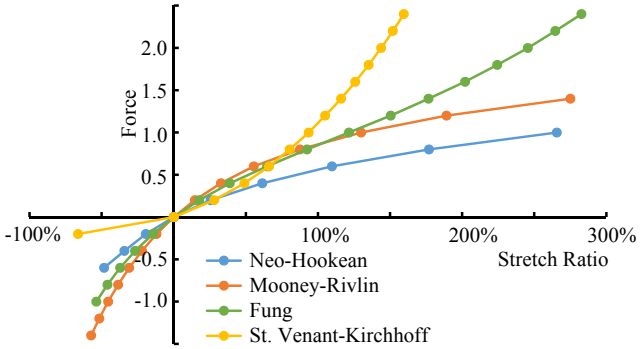


Figure 11: *The force-displacement curves generated by the box example. These curves are consistent with the stress-strain relationships of the underlying hyperelastic models.*

artifacts. For example, if bending elasticity is significantly stiffer than planar elasticity, it can cause cloth to be overly stretched. Meanwhile, if stiff elastic energy dominates gravitational energy, it can cause deformable bodies to fall slowly. Certain elastic models, such as the St. Venant-Kirchhoff model, do not offer sufficient resistance against compression. In that case, the method will have difficulty in avoiding inverted elements and oscillation artifacts at the same time. The initialization approach under the constant acceleration assumption can also cause small oscillation artifacts, if the parameter η is not sufficiently small. The whole idea behind our method is based on the implicit time integration scheme, so it suffers from the artificial damping issue. Finally, we still need additional mechanisms for self collision detection.

7 Conclusions and Future Work

In this paper, we show how to improve the gradient descent method by Jacobi preconditioning and Chebyshev acceleration, for solving the nonlinear optimization problem involved in elastic body simulation. While the convergence rate of our method is similar to that of nonlinear conjugate gradient, it requires zero dot product operation. This characteristics allows it to run efficiently and robustly on the GPU, after applying step length adjustment, initialization, model conversion techniques.

Since force evaluation is the bottleneck of our simulator, we will investigate possible ways to reduce its cost, especially the cost spent on singular value decomposition. We are also interested in finding better ways to handle step lengths and inverted elements. Potential solutions should have minimal impact on the simulation performance. Another interesting research direction we plan to explore is to couple our method with multi-grid techniques. The design of our method does not prevent it from using other parallelizable preconditioners. In particular, we would like to know whether the method can work with multi-color Gauss-Seidel preconditioners as well. Finally, we will study the use of our idea in solving other simulation problems, such as material and shape design.

Acknowledgments

This work was partly funded by NSF grants CHS-1524992, CHS-1464306, and CNS-1637092. The first author would also like to thank Adobe Research and NVIDIA Research for additional equipment and funding supports.

References

Baraff, D., and Witkin, A. 1998. Large steps in cloth simulation. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, SIGGRAPH '98, 43–54.

Bergou, M., Wardetzky, M., Harmon, D., Zorin, D., and Grinspun, E. 2006. A quadratic bending model for inextensible surfaces. In *Proceedings of SGP*, 227–230.

Bouaziz, S., Martin, S., Liu, T., Kavan, L., and Pauly, M. 2014. Projective dynamics: Fusing constraint projections for fast simulation. *ACM Trans. Graph. (SIGGRAPH) 33*, 4 (July), 154:1–154:11.

Bridson, R., Marino, S., and Fedkiw, R. 2003. Simulation of clothing with folds and wrinkles. In *Proceedings of SCA*, 28–36.

Cordero, A., Hueso, J. L., Martínez, E., and Torregrosa, J. R. 2010. New modifications of Potra-Pták's method with optimal fourth and eighth orders of convergence. *J. Comput. Appl. Math. 234*, 10 (Sept.), 2969–2976.

Daviet, G., Bertails-Descoubes, F., and Boissieux, L. 2011. A hybrid iterative solver for robustly capturing Coulomb friction in hair dynamics. *ACM Trans. Graph. (SIGGRAPH Asia) 30*, 6 (Dec.), 139:1–139:12.

Dick, C., Georgii, J., and Westermann, R. 2011. A real-time multigrid finite hexahedra method for elasticity simulation using CUDA. *Simulation Modelling Practice and Theory 19*, 2, 801–816.

Fei, Y., Rong, G., Wang, B., and Wang, W. 2014. Parallel L-BFGS-B algorithm on GPU. *Comput. Graph. 40* (May), 1–9.

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

Garg, A., Grinspun, E., Wardetzky, M., and Zorin, D. 2007. Cubic shells. In *Proceedings of SCA*, 91–98.

Golub, G. H., and Van Loan, C. F. 1996. *Matrix computations (3rd Ed.)*. Johns Hopkins University Press, Baltimore, MD, USA.

Gutknecht, M. H., and Röllin, S. 2002. The Chebyshev iteration revisited. *Parallel Computing*, 28, 263–283.

Irving, G., Teran, J., and Fedkiw, R. 2004. Invertible finite elements for robust simulation of large deformation. In *Proceedings of SCA*, 131–140.

Kharevych, L., Yang, W., Tong, Y., Kanso, E., Marsden, J. E., Schröder, P., and Desbrun, M. 2006. Geometric, variational integrators for computer animation. In *Proceedings of SCA*, 43–51.

Kim, T.-Y., Chentanez, N., and Müller-Fischer, M. 2012. Long range attachments - A method to simulate inextensible clothing in computer games. In *Proceedings of SCA*, 305–310.

Liu, T., Bargteil, A. W., O'Brien, J. F., and Kavan, L. 2013. Fast simulation of mass-spring systems. *ACM Trans. Graph. (SIGGRAPH Asia) 32*, 6 (Nov.), 214:1–214:7.

Macklin, M., and Müller, M. 2013. Position based fluids. *ACM Trans. Graph. (SIGGRAPH) 32*, 4 (July), 104:1–104:12.

Macklin, M., Müller, M., Chentanez, N., and Kim, T.-Y. 2014. Unified particle physics for real-time applications. *ACM Trans. Graph. (SIGGRAPH) 33*, 4 (July), 153:1–153:12.

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

McAdams, A., Selle, A., Tamstorf, R., Teran, J., and Sifakis, E. 2011. Computing the singular value decomposition of 3x3 matrices with minimal branching and elementary floating point operations. Technical report, University of Wisconsin - Madison.

McAdams, A., Zhu, Y., Selle, A., Empey, M., Tamstorf, R., Teran, J., and Sifakis, E. 2011. Efficient elasticity for character skinning with contact and collisions. *ACM Trans. Graph. (SIGGRAPH) 30*, 4 (July), 37:1–37:12.

Müller, M., and Gross, M. 2004. Interactive virtual materials. In *Proceedings of Graphics Interface*, 239–246.

Müller, M., Heidelberger, B., Teschner, M., and Gross, M. 2005. Meshless deformations based on shape matching. *ACM Trans. Graph. (SIGGRAPH) 24*, 3 (July), 471–478.

Müller, M., Heidelberger, B., Hennix, M., and Ratcliff, J. 2007. Position based dynamics. *J. Vis. Comun. Image Represent. 18*, 2 (Apr.), 109–118.

Müller, M., Chentanez, N., Kim, T., and Macklin, M. 2014. Strain based dynamics. In *Proceedings of SCA*, 21–23.

Müller, M. 2008. Hierarchical position based dynamics. In *Proceedings of VRIPHYS*, 1–10.

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

O'donoghue, B., and Candès, E. 2015. Adaptive restart for accelerated gradient schemes. *Found. Comput. Math. 15*, 3 (June), 715–732.

Ogden, R. W. 1997. *Non-linear elastic deformations*. Dover Civil and Mechanical Engineering. Dover Publications, Inc.

Patterson, T., Mitchell, N., and Sifakis, E. 2012. Simulation of complex nonlinear elastic bodies using lattice deformers. *ACM Trans. Graph. (SIGGRAPH Asia) 31*, 6 (Nov.), 197:1–197:10.

Perez, J., Perez, A. G., and Otaduy, M. A. 2013. Simulation of hyperelastic materials using energy constraints. In *Proceedings of the XXIII CEIG (Spanish Conference on Computer Graphics)*.

Provot, X. 1996. Deformation constraints in a mass-spring model to describe rigid cloth behavior. In *Proceedings of Graphics Interface*, 147–154.

Rivers, A. R., and James, D. L. 2007. FastLSM: Fast lattice shape matching for robust real-time deformation. *ACM Trans. Graph. (SIGGRAPH) 26*, 3 (July).

Stern, A., and Grinspun, E. 2009. Implicit-explicit variational integration of highly oscillatory problems. *Multiscale Model. Simul. 7*, 4, 1779–1794.

Su, J., Sheth, R., and Fedkiw, R. 2013. Energy conservation for the simulation of deformable bodies. *IEEE Transactions on Visualization and Computer Graphics 19*, 2 (Feb.), 189–200.

Teran, J., Sifakis, E., Irving, G., and Fedkiw, R. 2005. Robust quasistatic finite elements and flesh simulation. In *Proceedings of SCA*, 181–190.

Thomaszewski, B., Pabst, S., and Strasser, W. 2009. Continuum-based strain limiting. *Computer Graphics Forum (Eurographics) 28*, 2, 569–576.

Tournier, M., Nesme, M., Gilles, B., and Faure, F. 2015. Stable constrained dynamics. *ACM Trans. Graph. (SIGGRAPH) 34*, 4 (July), 132:1–132:10.

Wang, H., O'Brien, J., and Ramamoorthi, R. 2010. Multi-resolution isotropic strain limiting. *ACM Trans. Graph. (SIGGRAPH Asia) 29*, 6 (Dec.), 156:1–156:10.

Wang, H. 2015. A Chebyshev semi-iterative approach for accelerating projective and position-based dynamics. *ACM Trans. Graph. (SIGGRAPH Asia) 34*, 6 (Oct.), 246:1–246:9.

Xu, H., Sin, F., Zhu, Y., and Barbič, J. 2015. Nonlinear material design using principal stretches. *ACM Trans. Graph. (SIGGRAPH) 34*, 4 (July), 75:1–75:11.

Zhu, Y., Sifakis, E., Teran, J., and Brandt, A. 2010. An efficient multigrid method for the simulation of high-resolution elastic solids. *ACM Trans. Graph. 29*, 2 (Apr.), 16:1–16:18.