# Preconditioned Nonlinear Conjugate Gradient Method for Real-time Interior-point Hyperelasticity

Supplemental Document

Xing Shen
Fuxi AI Lab, NetEase Inc
shenxing03@corp.netease.com

Runyuan Cai
Fuxi AI Lab, NetEase Inc
cairunyuan@corp.netease.com

Mengxiao Bi
Fuxi AI Lab, NetEase Inc
bimengxiao@corp.netease.com

Tangjie Lv
Fuxi AI Lab, NetEase Inc
hzlvtangjie@corp.netease.com

**ACM Reference Format:**
Xing Shen, Runyuan Cai, Mengxiao Bi, and Tangjie Lv. 2024. Preconditioned Nonlinear Conjugate Gradient Method for Real-time Interior-point Hyperelasticity: Supplemental Document. In *Special Interest Group on Computer Graphics and Interactive Techniques Conference Conference Papers '24 (SIGGRAPH Conference Papers '24), July 27-August 1, 2024, Denver, CO, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3641519.3657490

## 1 SUPPLEMENTAL DETAILS OF SECTION 3.1

### 1.1 Supplemental Definition

Following is some supplemental definitions in Section 3.1 of main paper. In the main paper, we employ three invariants to formulate the energy density function for a hyperelastic material,

$$I_1 = \mathrm{tr}(\mathbf{S}), \quad I_2 = \mathrm{tr}\left(\mathbf{F}^T\mathbf{F}\right), \quad I_3 = \det(\mathbf{F}).$$

where $\mathbf{F}$ denotes the deformation gradient and $\mathbf{S}$ is derived from Polar decomposition $\mathbf{F} = \mathbf{RS}$.

The gradients and Hessians of each invariant with respect to $\mathbf{F}$ can be obtained as follows:

$$\mathbf{g}_1 = \mathrm{vec}(\mathbf{R}), \qquad \mathbf{H}_1 = \sum_{i=0}^{2} \lambda_i \mathbf{q}_i \mathbf{q}_i^\top,$$

$$\mathbf{g}_2 = 2\,\mathrm{vec}(\mathbf{F}), \qquad \mathbf{H}_2 = 2\mathbf{I}_{9\times 9},$$

$$\mathbf{g}_3 = \mathrm{vec}\left([\mathbf{f}_1\times\mathbf{f}_2, \mathbf{f}_2\times\mathbf{f}_0, \mathbf{f}_0\times\mathbf{f}_1]\right), \quad \mathbf{H}_3 = \begin{bmatrix} \mathbf{0} & -\hat{\mathbf{f}}_2 & \hat{\mathbf{f}}_1 \\ \hat{\mathbf{f}}_2 & \mathbf{0} & -\hat{\mathbf{f}}_0 \\ -\hat{\mathbf{f}}_1 & \hat{\mathbf{f}}_0 & \mathbf{0} \end{bmatrix},$$

where vectorization vec() is column-wise flattening of a matrix into a vector, as demonstrated below:

$$\mathbf{A} = \begin{bmatrix} a & c \\ b & d \end{bmatrix}, \quad \mathrm{vec}(\mathbf{A}) = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}.$$

Moreover $\mathbf{q}_i = \mathrm{vec}(\mathbf{Q}_i)$. $\lambda_i$ and $\mathbf{Q}_i$ are defined as :

$$\lambda_0 = \tfrac{2}{\sigma_x+\sigma_y}, \quad \mathbf{Q}_0 = \tfrac{1}{\sqrt{2}}\mathbf{U}\begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}\mathbf{V}^\top,$$

$$\lambda_1 = \tfrac{2}{\sigma_y+\sigma_z}, \quad \mathbf{Q}_1 = \tfrac{1}{\sqrt{2}}\mathbf{U}\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix}\mathbf{V}^\top,$$

$$\lambda_2 = \tfrac{2}{\sigma_x+\sigma_z}, \quad \mathbf{Q}_2 = \tfrac{1}{\sqrt{2}}\mathbf{U}\begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix}\mathbf{V}^\top.$$

where $\mathbf{U}, \mathbf{V}$ is derived from the SVD decomposition $\mathbf{F} = \mathbf{U}\Sigma\mathbf{V}$, and $\sigma_x, \sigma_y, \sigma_z$ is the diagonal elements of $\Sigma$.

$\mathbf{0}$ represents the zero matrix, and $\mathbf{I}$ represents the identity matrix. $\mathbf{f}_0$, $\mathbf{f}_1$ and $\mathbf{f}_2$ represent the columns of $\mathbf{F}$. The symbol $\hat{}$ denotes a cross-product matrix. For any vector $\mathbf{z} = [z_0, z_1, z_2]^\top$,

$$\hat{\mathbf{z}} = \begin{bmatrix} 0 & -z_2 & z_1 \\ z_2 & 0 & -z_0 \\ -z_1 & z_0 & 0 \end{bmatrix}.$$

### 1.2 Computaional Details of Hessian Matrix

The deformation gradient of a linear tetrahedral element is

$$\mathbf{F} = \mathbf{D}_s\mathbf{D}_m^{-1},$$

where the deformed shape matrix $\mathbf{D}_s$ and the reference shape matrix $\mathbf{D}_m$ are

$$\mathbf{D}_s = [\mathbf{x}_1 - \mathbf{x}_0 \,|\mathbf{x}_2 - \mathbf{x}_0|\, \mathbf{x}_3 - \mathbf{x}_0], \quad \mathbf{D}_m = [\bar{\mathbf{x}}_1 - \bar{\mathbf{x}}_0 \,|\bar{\mathbf{x}}_2 - \bar{\mathbf{x}}_0|\, \bar{\mathbf{x}}_3 - \bar{\mathbf{x}}_0].$$

Hence

$$\frac{\partial \mathbf{F}}{\partial \mathbf{x}} = \frac{\partial \mathbf{D}_s}{\partial \mathbf{x}}\mathbf{D}_m^{-1}, \tag{1}$$

and $\mathrm{vec}\left(\frac{\partial \mathbf{F}}{\partial \mathbf{x}}\right) \in \mathcal{R}^{9\times 12}$. For the sake of writing convenience, we omit the notation $\mathrm{vec}(\cdot)$ in the following. $\mathrm{vec}\left(\frac{\partial \mathbf{F}}{\partial \mathbf{x}}\right)$.

As we defined in the main paper,

$$\frac{\partial^2 \Psi}{\partial \mathbf{x}^2} = \sum_{i=1}^{3}\left(\frac{\partial^2 \Psi}{\partial I_i^2}\mathbf{h}_i + \frac{\partial \Psi}{\partial I_i}\mathbf{h}_{i+3}\right),$$

where

$$\mathbf{h}_i = \left(\frac{\partial \mathbf{F}}{\partial \mathbf{x}}\right)^\top \mathbf{g}_i\mathbf{g}_i^\top \left(\frac{\partial \mathbf{F}}{\partial \mathbf{x}}\right), \text{ for } i = 1, 2, 3$$

$$\mathbf{h}_i = \left(\frac{\partial \mathbf{F}}{\partial \mathbf{x}}\right)^\top \mathbf{H}_i \left(\frac{\partial \mathbf{F}}{\partial \mathbf{x}}\right) \text{ for } i = 4, 5, 6.$$

| Item | FLOPs | Item | FLOPs |
|---|---|---|---|
| $\frac{\partial \mathbf{F}}{\partial \mathbf{x}}^\top \mathbf{g}_3$ | 69 | $\mathbf{p}^\top \mathbf{h}_3 \mathbf{p}$ | 93 |
| $\frac{\partial \mathbf{F}}{\partial \mathbf{x}} \mathbf{p}$ | 72 | $\mathbf{p}^\top \mathbf{h}_5 \mathbf{p}$ | 90 |
| $\overline{diag}\left(\frac{\partial \mathbf{F}}{\partial \mathbf{x}}^\top \frac{\partial \mathbf{F}}{\partial \mathbf{x}}\right)$ | 26 | $\mathbf{p}^\top \mathbf{h}_6 \mathbf{p}$ | 126 |
| $\mathbf{g}_3 \mathbf{g}_3^\top$ | 45 | $\left(\frac{\partial^2 \Psi_{\mathrm{NH}}}{\partial \mathbf{F}^2}\right)$ | 405 |

**Table 1: FLOPs of some items.**

so we can derive

$$\overline{diag}(\mathbf{h}_1) = \left(\frac{\partial \mathbf{F}}{\partial \mathbf{x}}^\top \mathbf{g}_1\right)^{\circ 2}, \qquad \overline{diag}(\mathbf{h}_2) = \left(\frac{\partial \mathbf{F}}{\partial \mathbf{x}}^\top \mathbf{g}_2\right)^{\circ 2},$$

$$\overline{diag}(\mathbf{h}_3) = \left(\frac{\partial \mathbf{F}}{\partial \mathbf{x}}^\top \mathbf{g}_3\right)^{\circ 2}, \qquad \overline{diag}(\mathbf{h}_4) = \sum_{i=0}^{2} \lambda_i \left(\frac{\partial \mathbf{F}}{\partial \mathbf{x}}^\top \mathbf{q}_i\right)^{\circ 2},$$

$$\overline{diag}(\mathbf{h}_5) = 2\,\overline{diag}\left(\frac{\partial \mathbf{F}}{\partial \mathbf{x}}^\top \frac{\partial \mathbf{F}}{\partial \mathbf{x}}\right), \quad \overline{diag}(\mathbf{h}_6) = \mathbf{0},$$

and

$$\mathbf{p}^\top \mathbf{h}_1 \mathbf{p} = \left(\mathbf{p}^\top \frac{\partial \mathbf{F}}{\partial \mathbf{x}}^\top \mathbf{g}_1\right)^2, \quad \mathbf{p}^\top \mathbf{h}_2 \mathbf{p} = \left(\mathbf{p}^\top \frac{\partial \mathbf{F}}{\partial \mathbf{x}}^\top \mathbf{g}_2\right)^2,$$

$$\mathbf{p}^\top \mathbf{h}_3 \mathbf{p} = \left(\mathbf{p}^\top \frac{\partial \mathbf{F}}{\partial \mathbf{x}}^\top \mathbf{g}_3\right)^2, \quad \mathbf{p}^\top \mathbf{h}_4 \mathbf{p} = \sum_{i=0}^{2} \lambda_i \left(\mathbf{q}_i^\top \frac{\partial \mathbf{F}}{\partial \mathbf{x}} p\right)^2,$$

$$\mathbf{p}^\top \mathbf{h}_5 \mathbf{p} = 2\,\|\frac{\partial \mathbf{F}}{\partial \mathbf{x}} \mathbf{p}\|_2^2, \qquad \mathbf{p}^\top \mathbf{h}_6 \mathbf{p} = \left(\frac{\partial \mathbf{F}}{\partial \mathbf{x}} \mathbf{p}\right)^\top \mathbf{H}_3 \left(\frac{\partial \mathbf{F}}{\partial \mathbf{x}} \mathbf{p}\right).$$

So we have

$$\overline{diag}\left(\frac{\partial^2 \Psi}{\partial \mathbf{x}^2}\right) = \sum_{i=1}^{3} \left(\frac{\partial^2 \Psi}{\partial I_i^2} \overline{diag}(\mathbf{h}_i) + \frac{\partial \Psi}{\partial I_i} \overline{diag}(\mathbf{h}_{i+3})\right),$$

$$\mathbf{p}^\top \frac{\partial^2 \Psi}{\partial \mathbf{x}^2} \mathbf{p} = \sum_{i=1}^{3} \left(\frac{\partial^2 \Psi}{\partial I_i^2} \mathbf{p}^\top \mathbf{h}_i \mathbf{p} + \frac{\partial \Psi}{\partial I_i} \mathbf{p}^\top \mathbf{h}_{i+3} \mathbf{p}\right).$$

In Section 5, we present some codes about the computation relative to $\frac{\partial \mathbf{F}}{\partial \mathbf{x}}$.

## 1.3 Counting Details of FLOPs

The Neo-Hookean elasticity can be characterized in terms of $I_1, I_2, I_3$,

$$\Psi_{\mathrm{NH}} = \frac{\mu}{2}(I_2 - 3) - \mu \log(I_3) + \frac{\lambda}{2}(\log(I_3))^2.$$

The Hessian matrix of Neo-Hookean elasticity is

$$\left(\frac{\partial^2 \Psi_{\mathrm{NH}}}{\partial \mathbf{F}^2}\right) = \frac{\mu}{2}\mathbf{H}_2 + \frac{\lambda(1 - \log I_3) + \mu}{I_3^2} \mathbf{g}_3 \mathbf{g}_3^T + \frac{\lambda \log I_3 - \mu}{I_3} \mathbf{H}_3,$$

so

$$\overline{diag}\left(\frac{\partial^2 \Psi_{\mathrm{NH}}}{\partial \mathbf{x}^2}\right) = \frac{\mu}{2}\overline{diag}(\mathbf{h}_5) + \frac{\lambda(1 - \log I_3) + \mu}{I_3^2} \overline{diag}(\mathbf{h}_3),$$

and

$$\mathbf{p}^\top \frac{\partial^2 \Psi_{\mathrm{NH}}}{\partial \mathbf{x}^2} \mathbf{p} = \frac{\mu}{2}\mathbf{p}^\top \mathbf{h}_5 \mathbf{p} + \frac{\lambda(1 - \log I_3) + \mu}{I_3^2} \mathbf{p}^\top \mathbf{h}_3 \mathbf{p} + \frac{\lambda \log I_3 - \mu}{I_3} \mathbf{p} \mathbf{h}_6 \mathbf{p}.$$

Here, we assume that $\frac{\partial \Psi}{\partial I_i}$, and $\frac{\partial^2 \Psi}{\partial I_i^2}$ is precomputed. From the code in the Section 5, we derive the FLOPs of some items in Table 1.

With our approach, the FLOPs of computing $\overline{diag}\left(\frac{\partial^2 \Psi_{\mathrm{NH}}}{\partial \mathbf{x}^2}\right)$ is

$$26 + 69 + 12 \times 4 = 143.$$

The FLOPs of computing $\mathbf{p}^\top \frac{\partial^2 \Psi_{\mathrm{NH}}}{\partial \mathbf{x}^2} \mathbf{p}$ is

$$90 + 93 + 126 + 5 = 314.$$

In contrast, if we first compute $\left(\frac{\partial^2 \Psi_{\mathrm{NH}}}{\partial \mathbf{F}^2}\right)$, which needs 405 FLOPs, and then compute $\overline{diag}(\left(\frac{\partial \mathbf{F}}{\partial \mathbf{x}}\right)^\top \left(\frac{\partial^2 \Psi}{\partial \mathbf{F}^2}\right) \frac{\partial \mathbf{F}}{\partial \mathbf{x}})$, which requires 219 FLOPs. So FLOPs of computing $\overline{diag}\left(\frac{\partial^2 \Psi_{\mathrm{NH}}}{\partial \mathbf{x}^2}\right)$ is 624.

Similarly, if we first compute $\frac{\partial^2 \Psi_{\mathrm{NH}}}{\partial \mathbf{F}^2}$, and then compute $\frac{\partial^2 \Psi_{\mathrm{NH}}}{\partial \mathbf{x}^2}$, which requires $18 \times 12 \times 9 + 26 \times 3 \times 12 + 6$=2886 FLOPs. Finally computing $\mathbf{p}^\top \mathbf{H}_{12\times 12} \mathbf{p}$ requires 299 FLOPs. Hence the total number of FLOPs of required for computing $\mathbf{p}^\top \frac{\partial^2 \Psi_{\mathrm{NH}}}{\partial \mathbf{x}^2} \mathbf{p}$ is 3590 FLOPs.

It is evident from the above analysis that our approach requires significantly fewer FLOPs.

## 2 IMPLEMENTATION DETAILS

**Details of Computation of Hessian.** Computing of $\overline{\mathrm{diag}}(\mathbf{H})$ and $\mathbf{p}^\top \mathbf{H} \mathbf{p}$ may yield negative values for highly deformed elements or complex contacts due to the non-positive semi-definite (non-PSD) Hessian matrix. Instead of using a PSD-projected Hessian, we clamp negative values to zero. This effectively handles artifacts and penetrations from the non-PSD Hessian with negligible computational cost.

**The Gradient of Distance.** In Equation (8) in the main paper, we reformulate the distance as follows.

$$d = \|\mathbf{t}\| = \min_{c_0, c_1, c_2, c_3} \|c_0 \mathbf{x}_0 + c_1 \mathbf{x}_1 + c_2 \mathbf{x}_2 + c_3 \mathbf{x}_3\| \tag{2}$$

In the computation of $\frac{\partial \mathbf{t}}{\partial \mathbf{x}}$, we assume $\frac{\partial c_i}{\partial x} = 0$ by default. While this assumption may not be mathematically rigorous, as the value of $c_i$ could potentially be influenced by $\mathbf{x}$, this setting is inspired by Chapter 14 of [Kim and Eberle 2022]. In Section 14.2.4, the authors assert that $\frac{\partial^2 \mathbf{t}}{\partial \mathbf{x}} = 0$, and both Equations 14.34 and 14.45 rely on this result. Since these formulations have proven effective in practical applications, we follow the derivation process and apply them to describe the IPC energy. However, the detailed implications of this approach warrant further exploration.

**Unsquared Distances.** In IPC, squared distances are employed to avoid numerical errors and inefficiencies introduced by taking squared roots in gradient and Hessian computations. However, with our implementation in Section 3.2, the gradient and Hessian computations are simplfied, so the squared distances is not necessary. Moreover, the float32 precision may be insufficient for the application of squared distances. Conversely, the float64 precision, while offering higher precision, it would compromise the computational speed of our GPU-parallel implementation. Consequently, we opt for unsquared distances in our approach.

**Parallel Edge-Edge Conditions.** In IPC, a mollified edge-edge barrier is proposed to address the slow convergence of Newton iterations or convergence failures caused by parallel edge-edge conditions. However, since we employ float32 precision and our proposed PNCG method is less sensitive to the non-smoothness of the barrier function compared to Newton's method. The usage of

mollified edge-edge barriers, while potentially improving convergence and result accuracy, would incur additional computational costs. Considering these trade-offs, we opt not to incorporate mollified edge-edge barriers in our approach.

# 3 ADDITIONAL EXPERIMENTS

## 3.1 Twisting mat

In Figure 1, each side of the mat rotates at an angular speed of 25° per second for eight rounds, with a termination condition of $\epsilon = 1 \times 10^{-4}$ and a maximum number of iterations set to 500. Regarding the Dirichlet boundary condition setup, we update the positions of the boundary vertices at the start of each time step and enforce a zero gradient for these vertices during each iteration. Throughout the simulation, the IPC potential maintains a constant value of $\hat{d} = 0.0045$ and the value of $\kappa$ gradually increases from 0.5 to 5.0. As shown in Figure 1(d), the simulation remains penetration-free after 4.5 rounds of twisting; after that, finding a suitable $\kappa$ value to prevent penetration becomes challenging. Even when penetration occurs, the simulation can continue and exhibit more pronounced buckling effects, as illustrated in Figures 1(e-h). After penetration occurs, despite increasing the maximum number of iterations to 1000, the optimization fails to converge, resulting in a less substantial buckling effect at the center region compared to the sides. Additionally, the simulation speed gradually decreases from 4 FPS (Frame 720) to 0.7 FPS (Frame 2880).

# 4 SOME FORMULATIONS OF NONLINEAR CONJUGATE GRADIENT METHOD

Following are the precondition nonlinear gradient methods used in Section 6 of the main paper,

The preconditioned CD algorithm [Fletcher 2000] is :

$$\beta_k^{CD} = \frac{\mathbf{g}_{k+1}^\top \mathbf{P}_{k+1}\mathbf{g}_{k+1}}{-\mathbf{g}_k^\top \mathbf{p}_k}.$$

The preconditioned PRP algorithm [Polak and Ribiere 1969] is :

$$\beta_k^{PRP} = \frac{\mathbf{g_{k+1}}^\top \mathbf{P}_{k+1}\mathbf{y}_k}{-\mathbf{g}_k^\top \mathbf{p_k}},$$

where $\mathbf{y}_k = \mathbf{g_{k+1}} - \mathbf{g}_k$.

The preconditioned FR algorithm [Fletcher and Reeves 1964] is :

$$\beta_k^{FR} = \frac{\mathbf{g_{k+1}}^\top \mathbf{P}_{k+1}\mathbf{g_{k+1}}}{\mathbf{g}_k^\top \mathbf{g}_k}.$$

The preconditioned DK algorithm [Dai and Kou 2013] is:

$$\beta_k^{DK} = \frac{\mathbf{g}_{k+1}^T \mathbf{P}_{k+1}\mathbf{y}_k}{\mathbf{y}_k^T \mathbf{p}_k} - \frac{\mathbf{y}_k^T \mathbf{P}_{k+1}\mathbf{y}_k}{\mathbf{y}_k^T \mathbf{p}_k}\frac{\mathbf{p}_k^T \mathbf{g}_{k+1}}{\mathbf{y}_k^T \mathbf{p}_k}. \tag{3}$$

**SSML-BFGS method with Jacobi preconditioning.** The search direction of self-scaling memoryless BFGS method is computed as

$$\mathbf{P}_{k+1} = -\mathbf{H}_{k+1}\mathbf{g_{k+1}},$$

$$\mathbf{H}_{k+1} = \frac{1}{\tau_k}\left(I - \frac{\mathbf{s}_k\mathbf{y}_k^\mathrm{T} + \mathbf{y}_k\mathbf{s}_k^\mathrm{T}}{\mathbf{y}_k^\mathrm{T}\mathbf{s}_k}\right) + \left(1 + \frac{1}{\tau_k}\frac{\|\mathbf{y}_k\|^2}{\mathbf{y}_k^\mathrm{T}\mathbf{s}_k}\right)\frac{\mathbf{s}_k\mathbf{s}_k^\mathrm{T}}{\mathbf{y}_k^\mathrm{T}\mathbf{s}_k},$$

where $\mathbf{s}_k = x_{k+1} - x_k = \alpha\mathbf{p_k}$ ,and $\tau_k$ is the scaling parameter, known as the SSML-BFGS updating. Employing the Jacobi preconditioner can significantly increase the convergence rate.

$$\mathbf{H}_{k+1} = \mathbf{P}_{k+1} - \frac{\mathbf{s}_k\mathbf{y}_k^\top\mathbf{P}_{k+1} + \mathbf{P}_{k+1}\mathbf{y}_k\mathbf{s}_k^\top}{\mathbf{y}_k^\top\mathbf{s}_k} + (1 + \frac{\mathbf{y}_k^\top\mathbf{P}_{k+1}\mathbf{y}_k}{\mathbf{y}_k^\top\mathbf{s}_k})\frac{\mathbf{s}_k\mathbf{s}_k^\top}{\mathbf{y}_k^\top\mathbf{s}_k}.$$

So we have

$$\mathbf{P}_{k+1} = -\mathbf{H}_{k+1}\mathbf{g_{k+1}} = -\mathbf{P}_{k+1}\mathbf{g_{k+1}} + (\frac{\mathbf{g_{k+1}}^\top\mathbf{P}_{k+1}\mathbf{y}_k}{\mathbf{y}_k^\top\mathbf{p_k}} - (\alpha_k + \frac{\mathbf{y}_k^\top\mathbf{P}_{k+1}\mathbf{y}_k}{\mathbf{y}_k^\top\mathbf{p_k}})\frac{\mathbf{g_{k+1}}^\top\mathbf{p_k}}{\mathbf{y}_k^\top\mathbf{p_k}})\mathbf{p_k} + \frac{\mathbf{g_{k+1}}^\top\mathbf{p_k}}{\mathbf{y}_k^\top\mathbf{p_k}}\mathbf{y}_k.$$

Therefore we have the formulation of preconditioned SSML-BFGS method:

$$\mathbf{P}_{k+1} = -\mathbf{P}_{k+1}\mathbf{g_{k+1}} + \beta\mathbf{p_k} + \gamma\mathbf{P}_{k+1}\mathbf{y}_k \tag{4}$$

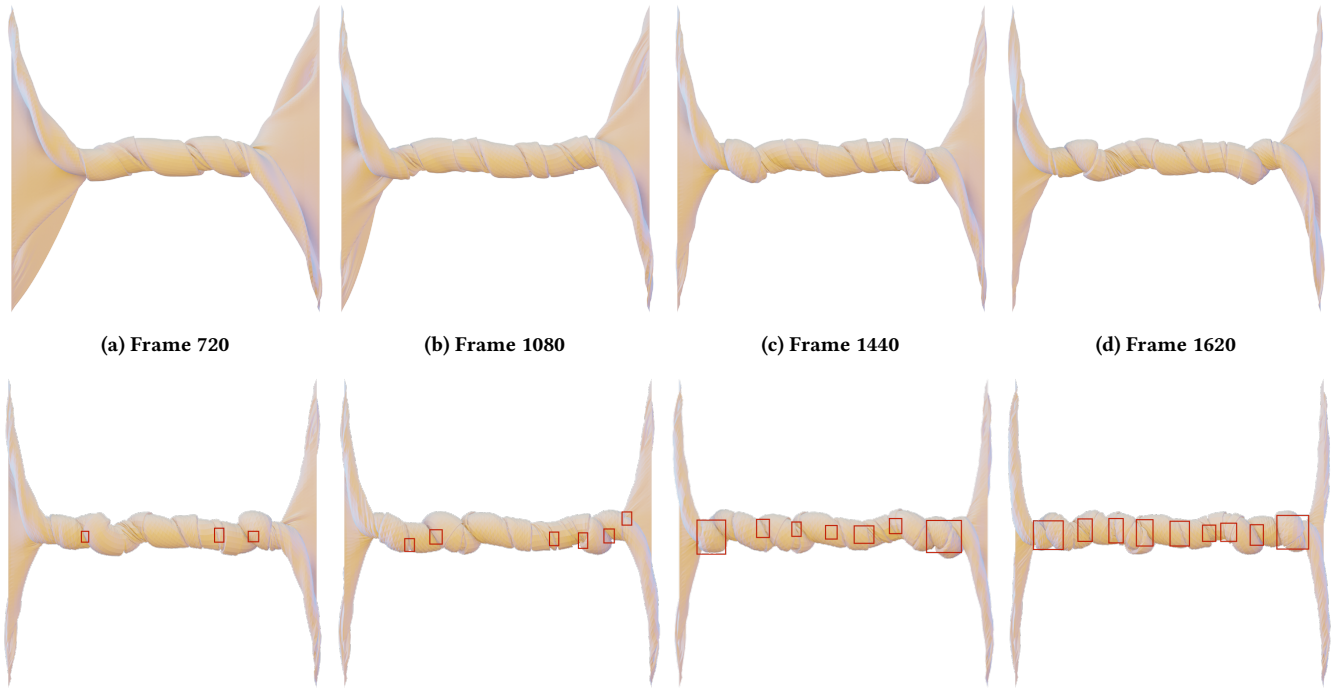
$$\beta = \frac{\mathbf{g_{k+1}}^\top\mathbf{P}_{k+1}\mathbf{y}_k}{\mathbf{y}_k^\top\mathbf{p_k}} - (\alpha_k + \frac{\mathbf{y}_k^\top\mathbf{P}_{k+1}\mathbf{y}_k}{\mathbf{y}_k^\top\mathbf{p_k}})\frac{\mathbf{g_{k+1}}^\top\mathbf{p_k}}{\mathbf{y}_k^\top\mathbf{p_k}}, \quad \gamma = \frac{\mathbf{g_{k+1}}^\top\mathbf{p_k}}{\mathbf{y}_k^\top\mathbf{p_k}}, \tag{5}$$

We can observe that the proposed method exhibits a similar form to the preconditioned DK method (Equation 3). It can also be viewed as a three-term conjugate gradient method. In our experiments, we find that two-term conjugate gradient methods, such as the DK method, demonstrate greater robustness compared to three-term conjugate gradient methods. We observe that the parameters $\beta$ or $\gamma$ sometimes become excessively large, leading to an incorrect search direction. Intuitively, the parameter $\beta$ can be interpreted as a modifier of the preconditioned gradient direction $\mathbf{P}_{k+1}g_{k+1}$, and one modifier is sufficient to find a conjugate gradient direction. Introducing an additional modifier $\gamma$, may increase accuracy, but it also makes the algorithm more sensitive.

# 5 SOME SOURCE CODES

Following is the some codes of the computing regrading $\frac{\partial \mathbf{F}}{\partial \mathbf{x}}$.

```
@ti.func
def compute_dFdx_p(B, p):
    # dFdx 9x12, p 12x1
    B00, B01, B02, B10, B11, B12, B20, B21,
        B22 = B[0, 0], B[0, 1], B[0, 2], B
        [1, 0], B[1, 1], B[1, 2], B[2, 0], B
        [2, 1], B[2, 2]
    p0, p1, p2, p3, p4, p5, p6, p7, p8, p9,
        p10, p11 = p[0], p[1], p[2], p[3], p
        [4], p[5], p[6], p[7], p[8], p[9], p
        [10], p[11]
    t1 = - B00 - B10 - B20
    t2 = - B01 - B11 - B21
    t3 = - B02 - B12 - B22
```



(a) Frame 720 (b) Frame 1080 (c) Frame 1440 (d) Frame 1620

(e) Frame 1800 (f) Frame 2160 (g) Frame 2520 (h) Frame 2880

**Figure 1: Twist mat for eight rounds. Penetration happens after Frame 1620 with the red boxes highlighting the penetrated regions.**

```
    return ti.Matrix([B00 * p3 + B10 * p6 +
        B20 * p9 + p0 * t1,B00 * p4 + B10 *
        p7 + B20 * p10 + p1 * t1,B00 * p5 +
        B10 * p8 + B20 * p11 + p2 * t1,B01 *
         p3 + B11 * p6 + B21 * p9 + p0 * t2,
        B01 * p4 + B11 * p7 + B21 * p10 + p1
         * t2,B01 * p5 + B11 * p8 + B21 *
        p11 + p2 * t2,B02 * p3 + B12 * p6 +
        B22 * p9 + p0 * t3,B02 * p4 + B12 *
        p7 + B22 * p10 + p1 * t3,B02 * p5 +
        B12 * p8 + B22 * p11 + p2 * t3],
        float)
```

```
@ti.func
def compute_dFdxTranspose_p(B, p):
    # dFdxTranspose 12x9, p 9x1
    B00, B01, B02, B10, B11, B12, B20, B21,
        B22 = B[0, 0], B[0, 1], B[0, 2], B
        [1, 0], B[1, 1], B[1, 2], B[2, 0], B
        [2, 1], B[2, 2]
    p0, p1, p2, p3, p4, p5, p6, p7, p8 = p
        [0], p[1], p[2], p[3], p[4], p[5], p
        [6], p[7], p[8]
    t1 = - B00 - B10 - B20
    t2 = - B01 - B11 - B21
    t3 = - B02 - B12 - B22
    return ti.Matrix([p0*t1 + p3*t2 + p6*t3,
        p1*t1 + p4*t2 + p7*t3,p2*t1 + p5*t2
        + p8*t3, B00*p0 + B01*p3 + B02*p6,
        B00*p1 + B01*p4 + B02*p7, B00*p2 +
        B01*p5 + B02*p8, B10*p0 + B11*p3 +
        B12*p6, B10*p1 + B11*p4 + B12*p7,
        B10*p2 + B11*p5 + B12*p8, B20*p0 +
        B21*p3 + B22*p6, B20*p1 + B21*p4 +
        B22*p7, B20*p2 + B21*p5 + B22*p8],
        float)
```

```
@ti.func
def compute_diag_dFdxTranspose_dFdx(B):
    B00, B01, B02,  B10, B11, B12, B20, B21,
         B22 = B[0,0], B[0,1],  B[0,2],  B
        [1,0],  B[1,1],  B[1,2],  B[2,0],  B
        [2,1],  B[2,2]
    t0 = (B00 + B10 + B20)**2 + (B01 + B11 +
         B21)**2 + (B02 + B12 + B22)**2
    t1 = B00**2 + B01**2 + B02**2
    t2 = B10**2 + B11**2 + B12**2
    t3 = B20**2 + B21**2 + B22**2
    return ti.Vector([t0, t0, t0, t1, t1, t1
        , t2, t2, t2, t3, t3, t3], float)
```

```
@ti.func
```

```
def compute_d_H3_d(F, d):
    # d: 9x1, H3 9x9,
    d0, d1, d2, d3, d4, d5, d6, d7, d8= d
        [0], d[1], d[2], d[3], d[4], d[5], d
        [6], d[7], d[8]
    F00, F01, F02, F10, F11, F12, F20, F21,
        F22 = F[0, 0], F[0, 1], F[0, 2], F
        [1, 0], F[1, 1], F[1, 2], F[2, 0], F
        [2, 1], F[2, 2]
    return 2 * (F00 * d4 * d8 - F00 * d5 *
        d7 - F01 * d1 * d8 + F01 * d2 * d7 +
         F02 * d1 * d5 - F02 * d2 * d4 - F10
         * d3 * d8 + F10 * d5 * d6 + F11 *
        d0 * d8 - F11 * d2 * d6 - F12 * d0 *
         d5 + F12 * d2 * d3 + F20 * d3 * d7
        - F20 * d4 * d6 - F21 * d0 * d7 +
        F21 * d1 * d6 + F22 * d0 * d4 - F22
        * d1 * d3)
```

```
@ti.func
def compute_dTranspose_H3_d(F, d):
    # d^T: 1x9, H3: 9x9, d: 9x1
    d0, d1, d2, d3, d4, d5, d6, d7, d8= d
        [0], d[1], d[2], d[3], d[4], d[5], d
        [6], d[7], d[8]
    F00, F01, F02, F10, F11, F12, F20, F21,
        F22 = F[0, 0], F[0, 1], F[0, 2], F
        [1, 0], F[1, 1], F[1, 2], F[2, 0], F
        [2, 1], F[2, 2]
    return 2 * (F00 * d4 * d8 - F00 * d5 *
        d7 - F01 * d1 * d8 + F01 * d2 * d7 +
         F02 * d1 * d5 - F02 * d2 * d4 - F10
         * d3 * d8 + F10 * d5 * d6 + F11 *
        d0 * d8 - F11 * d2 * d6 - F12 * d0 *
         d5 + F12 * d2 * d3 + F20 * d3 * d7
        - F20 * d4 * d6 - F21 * d0 * d7 +
        F21 * d1 * d6 + F22 * d0 * d4 - F22
        * d1 * d3)
```

## REFERENCES


Yu-Hong Dai and Cai-Xia Kou. 2013. A nonlinear conjugate gradient algorithm with an optimal property and an improved Wolfe line search. *SIAM Journal on Optimization* 23, 1 (2013), 296–320.

Roger Fletcher. 2000. *Practical methods of optimization.* John Wiley & Sons.

Reeves Fletcher and Colin M Reeves. 1964. Function minimization by conjugate gradients. *The computer journal* 7, 2 (1964), 149–154.

Theodore Kim and David Eberle. 2022. Dynamic deformables: implementation and production practicalities (now with code!). In *ACM SIGGRAPH 2022 Courses*. 1–259.

Elijah Polak and Gerard Ribiere. 1969. Note sur la convergence de méthodes de directions conjuguées. *Revue française d'informatique et de recherche opérationnelle. Série rouge* 3, 16 (1969), 35–43.