Paper介绍

带接触的弹性形变的仿真一直是图形学中的一个重要课题，应用于外科手术训练、机器人技术、增强现实/虚拟现实（AR/VR）和布料和人体软组织的模拟等。然而，由于弹性项的非线性和非凸性，在不同外部条件下准确且稳健地仿真具有复杂自碰撞的弹性物体的动态行为是一个重大挑战。为了应对这一挑战，近年来提出了一种称为增量潜在接触（Incremental Potential Contact，IPC）的方法 [Li et al. 2020]。 IPC结合了内点法，使得弹性动力学和接触的仿真既稳健又准确。该方法利用对数障碍函数来近似由碰撞和接触引起的不等式约束，从而将问题转化为无约束优化问题，然后采用牛顿法来解决这一优化问题。IPC的稳健性归因于其线搜索机制。首先，采用连续碰撞检测（CCD）来确定最大步长，以确保物体保持无穿透状态。随后，实施回溯线搜索以实现目标函数的减少。然而，IPC的实际应用受到其相对较慢的仿真速度的限制。所以该文章的出发点就是想加速ipc算法的计算速度。

这篇文章中我们提出了PNCG算法用于模拟带碰撞的弹性形变。本文将介绍一下我们的方法以及我们文章idea的思路。

首先，简要介绍一下背景知识，从牛顿第二定律f=ma开始，

$$\frac{dx}{dt} = v$$
$$M\frac{dv}{dt} = f$$

其中x,v,f分别是点的位置，速度，受力， M是质量矩阵。

对时间做离散化，并使用向后欧拉公式可得

$$x^{t+1} = x^t + hv^{t+1},$$
$$v^{t+1} = v^t + hM^{-1}f^{t+1},$$

其中h 是时间步长, 。

即

$$M\left(x^{t+1} - \left(x^t + hv^t\right)\right) - h^2 f\left(x^{t+1}\right) = 0. \tag{1}$$

很自然地，这个问题的求解就等价于求解优化问题

$$x^{t+1} = \arg\min_{x} E(x)$$
$$\text{where } E(x) = \frac{1}{2}\|x - \tilde{x}\|_M^2 + h^2 P(x)$$

其中$\tilde{x}^t = x^t + hv^t$， $P(x)$ 表示力的势能，且$\frac{\partial P}{\partial x}(x) = -f(x)$. $E(x)$ 被称为增量势能（Incremental Potential）。E 的局部最小值，$\frac{\partial E}{\partial x}\left(x^{n+1}\right) = 0$ 对应的就是方程（1）的解。

+

再对方程做一些小修改，如下

$$\mathbf{x}^{t+1} = \arg\min_{\mathbf{x}} \frac{1}{2}(\mathbf{x} - \tilde{\mathbf{x}})^\top \mathbf{M}(\mathbf{x} - \tilde{\mathbf{x}}) + h^2\Psi(\mathbf{x}) \quad s.t.\ h_i(\mathbf{x}) \geq 0.$$

$\tilde{\mathbf{x}} = \mathbf{x}^t + h\mathbf{v}^t + h^2\mathbf{M}^{-1}\mathbf{f}_{ext}$ 也就是x 在受到外力f_{ext} （如重力等）的作用下预估的位置，$\frac{1}{2}(\mathbf{x} - \tilde{\mathbf{x}})^\top \mathbf{M}(\mathbf{x} - \tilde{\mathbf{x}})$ 也称为inertia potential。 此外超弹性能量$\Psi(x)$用于量化形变的大小。由$h_i(x) \geq 0$ 组成的不等式约束集C ，用于确保模拟过程中不会发生模型内和模型间的相交，即穿模现象的发生。所以问题现在变成了一个带约束的优化问题，优化问题的最小值即为问题的解。

带约束的优化问题的求解是一个很经典的问题，目前学术界通常使用IPC方法进行处理，即通过使用内点法(interior-point method) 引入barrier function将带约束的优化问题变为无约束的优化问题。

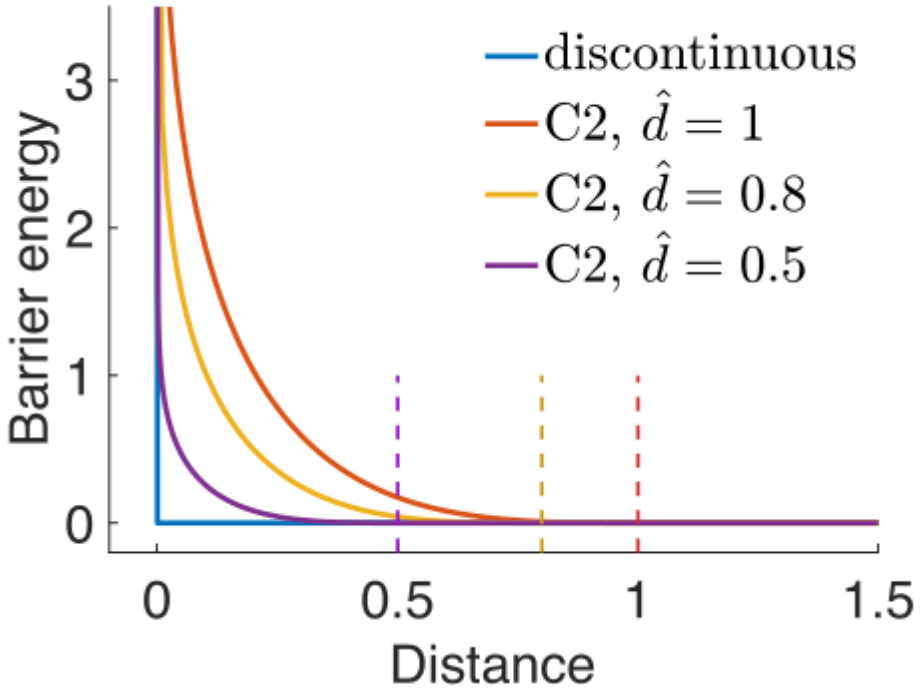
$$\mathbf{x}^{t+1} = \arg\min_{\mathbf{x}} E(\mathbf{x}), \tag{2}$$
$$E(\mathbf{x}) = \frac{1}{2}(\mathbf{x} - \tilde{\mathbf{x}})^\top \mathbf{M}(\mathbf{x} - \tilde{\mathbf{x}}) + h^2\Psi(\mathbf{x}) + \kappa \sum_{k \in C} b\left(d_k(\mathbf{x})\right). \tag{3}$$

其中barrier function 定义为

$$b\left(d_k(\mathbf{x})\right) = \begin{cases} -\left(d_k - \hat{d}\right)^2 \log\left(\frac{d_k}{\hat{d}}\right), & 0 < d_k < \hat{d}, \\ 0, & d_k \geq \hat{d}, \end{cases} \tag{4}$$

其中d 为距离。对于弹性形变的模拟，通常是使用有限元方法，将一个物理划分为多个小四面体，这里的距离计算就可以通过点-三角形 / 边-边 的碰撞对进行计算。$\hat{d}$ 控制碰撞发生的阈值，即距离小于$\hat{d}$ 时点之间才会受到碰撞力，κ 控制碰撞势能的大

小，是一个手动调整的参数。仔细看这个函数，当距离从 $\hat{d}$ 开始逐渐减小，能量也逐渐增大，在距离趋近于0的时候能量趋近于无穷大。这样就很好地将原本的约束项转化为了能量函数，同时也很好地定义了碰撞的能量，碰撞力的大小也可以由此得出。



这个函数设计的另一个好处是二阶连续，这个性质在优化问题中很关键，直接影响到收敛的效果。

至此简要地介绍了一下基本背景，接下来的重头戏就是怎么求解以上优化问题。

在IPC方法原文中使用的是经典的Newton法求解，算法流程如下

Algorithm 1 Barrier Aware Projected Newton

1: **procedure** BarrierAwareProjectedNewton(x^t, ϵ)
2: $x \leftarrow x^t$
3: $\hat{C} \leftarrow$ComputeConstraintSet$(x, \hat{d})$ ▷ Section 4.6, 6.1
4: $E_{\text{prev}} \leftarrow B_t(x, \hat{d}, \hat{C})$
5: $x_{\text{prev}} \leftarrow x$
6: **do**
7: $H \leftarrow \text{SPDProject}(\nabla_x^2 B_t(x, \hat{d}, \hat{C}))$ ▷ Section 4.3
8: $p \leftarrow -H^{-1} \nabla_x B_t(x, \hat{d}, \hat{C})$
9: // CCD line search: ▷ Section 4.4
10: $\alpha \leftarrow \min(1, \text{StepSizeUpperBound}(x, p, \hat{C}))$
11: **do**
12: $x \leftarrow x_{\text{prev}} + \alpha p$
13: $\hat{C} \leftarrow$ComputeConstraintSet$(x, \hat{d})$
14: $\alpha \leftarrow \alpha/2$
15: **while** $B_t(x, \hat{d}, \hat{C}) > E_{\text{prev}}$
16: $E_{\text{prev}} \leftarrow B_t(x, \hat{d}, \hat{C})$
17: $x_{\text{prev}} \leftarrow x$
18: Update κ, BCs and equality constraints ▷ Supplemental
19: **while** $\frac{1}{h}\|p\|_\infty > \epsilon_d$
20: return x

这带来一个问题，就是牛顿法的每次迭代都需要求解一个大型的线性系统，即第8行处，其中H是海森矩阵。这部分求解会占有大量的时间，所以原始的CPU版本的IPC方法速度是很慢的，因此学术界提出了许多不同的方法来加速求解。

本文的起初思路很简单，求解大型线性系统的方法中, PCG(Preconditioned conjugated gradient 预条件共轭梯度法）是很经典的一种算法。Newton + PCG 组合的求解方法在图形学的各种问题的求中也经常可以看到。那么一个很自然的想法，既然整个优化问题是非线性的(barrier 函数和弹性能量都是非线性的)， 那么与其先用Newton得到一个线性系统，然后用线性的PCG方法去求解，为什么不直接使用非线性的PCG去求解呢？ 而且PCG方法天然适合使用GPU 并行，非线性的PCG算法也是一个成熟的算法，针对不同的情景有许多成熟的优化技巧。

但是我们发现这个解法在图形学并不常见，在近年的许多加速IPC方法的文章还有相关的做弹性仿真的文章中也没有使用。这么直接的想法通常来说肯定有人想到了，很有可能已经有人试过然后发现存在某些问题。查找资料后发现原来在2015年王华民老师的文章Descent Methods for Elastic Body Simulation on the GPU 就提到了这个思路。以下是文章的原文

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.

这篇文章提到，非线性的共轭梯度法与文章中提出的方法相比，收敛性接近， 且不需要复杂的参数调整等模块。但是因为dot-product的计算消耗太大（15K 个点的模型，每次需要0.41ms ），所以当时没有采用nonlinear CG 方法。最后还提到了如果并行技术的发展使得dot-product 的计算加快了，会采用nonlinear CG 方法。

笔者当时第一反应就是“ 等等，dot-product ？ 向量之间的点乘？ 15K 的模型每次需要0.41ms？ 怎么可能？”

于是笔者当时转头就用Taichi 去测了一下，结果发现，同样大小的模型点乘只需要 0.012 ms, 速度提升了快40倍。那现在 nonlinear CG 的最大的瓶颈已经消失了。然后马上查了一波所有引用了这篇文章的其他文章，发现真没有人接着做 nonlinear CG方法。笔者大喜，马上开搞。

确定方向之后就开始算法的实现了。因为笔者对共轭梯度法并不熟悉，就翻出来经典的入门教程

《 An Introduction to the Conjugate Gradient Method Without the Agonizing Pain》, 看了足足一星期之后得出结论，这本书就是标题党，说好的 Without the Agonizing Pain 呢？ (╯°□°)╯ ┻━┻

总之最后还是成功地实现了无碰撞版本的nonlinear PCG了， 发现运行速度和收敛性都很好。这时候还是很开心的，主要的idea 已经验证正确了。

当然，nonlinear CG是一类很成熟的算法，这些年已经发展出了各种不同的变种，所以很自然地，我们对常用的几种类别的nonlinear CG测试了一下性能，发现Dai-Kou算法的收敛效果最佳，公式如下：

$$\mathbf{p}_{k+1} = -\mathbf{g}_{k+1} + \beta_k^{DK}\mathbf{p}_k, \tag{5}$$

$$\beta_k^{DK} = \frac{\mathbf{g}_{k+1}^T\mathbf{y}_k}{\mathbf{y}_k^T\mathbf{p}_k} - \frac{\mathbf{y}_k^T\mathbf{y}_k}{\mathbf{y}_k^T\mathbf{p}_k}\frac{\mathbf{p}_k^T\mathbf{g}_{k+1}}{\mathbf{y}_k^T\mathbf{p}_k}, \tag{6}$$

其中p 是每次迭代的更新的方向， g是梯度，$\mathbf{y}_k = \mathbf{g}_{k+1} - \mathbf{g}_k$. 可以看出$\beta$ 的计算需要4个向量间的点乘，而且这些点乘可以写在同一个循环中计算，很容易在gpu上进行并行计算，一个15 K 个点的模型计算一次只需要0.016ms， 几乎可以忽略。

另一个模块预处理算子 Preconditioner 是用于加速算法的收敛。这里我们用的是最简单的版本Jacobi preconditioning, 最后完整的公式如下

$$\mathbf{p}_{k+1} = -\mathbf{P}_{k+1}\mathbf{g}_{k+1} + \beta_k^{DK}\mathbf{p}_k \tag{7}$$

$$\beta_k^{DK} = \frac{\mathbf{g}_{k+1}^T\mathbf{P}_{k+1}\mathbf{y}_k}{\mathbf{y}_k^T\mathbf{p}_k} - \frac{\mathbf{y}_k^T\mathbf{P}_{k+1}\mathbf{y}_k}{\mathbf{y}_k^T\mathbf{p}_k}\frac{\mathbf{p}_k^T\mathbf{g}_{k+1}}{\mathbf{y}_k^T\mathbf{p}_k}, \tag{8}$$

其中$\mathbf{P} = diag(\mathbf{H})^{-1}$.为海森矩阵对角线矩阵的逆。

因为主要的idea 是要GPU加速版本的IPC, 所以具体实现就选择使用Taichi 语言实现， 以及使用MeshTaichi 库加速mesh的并行计算。

然后就是开始实现IPC版本了。 这时候就开始了疯狂的踩坑过程。 做带碰撞的模拟的时候一个关键的步骤就是碰撞检测。IPC 在使用牛顿法的时候，使用了CCD (连续碰撞检测) 对步长的最大值进行限制，从而保证结果的无穿透。 因为每一次的迭代中牛顿法得到的解是相对准确的，所以CCD的使用虽然有一定的耗时， 但是相比求解线性系统部分来说是占比很小。但是Nonlinear CG的收敛性是是没有牛顿法好的，所以需要

更多次的迭代，当然每次迭代的消耗也是远远小于牛顿法的，所以整体的速度会比牛顿法快很多。但是Nonlinear CG的最大的问题是对步长十分敏感，步长的选择对收敛性影响很大。在传统的共轭梯度法的文章中，通常会使用回溯法(backtracking) 或者插值的方法来确定一个合适步长，以保证算法的收敛。在无碰撞场景下，这些方法的耗时尚且能够接受。但是在有碰撞的场景，以回溯法为例，每次的回溯都需要更新点的位置(这里我们使用的是spatial hashing)，然后找出所有碰撞对，再计算所有的能量。前面也提到了向量点乘的耗时非常的小，这时候整个算法中耗时最多的就是上边这些步骤，那么为了找到一个合适的步长，一次回溯过程占用的时间跟计算出更新方向几乎差不多。所以一个好的步长选择的策略是至关重要的。

另外就是CCD的耗时了，CCD的作用是提供一个步长的上限，但是CCD在GPU上的耗时会占了整个计算过程的70%以上(这也跟我自己写的CCD确实有点烂有关)。总之，对于牛顿法，每次的更新方向都很准确，且迭代次数少，CCD的使用是合适的; 对于Nonlinear CG，每次计算出的更新方向其实并不是那么准确，所以需要更多的迭代次数。通过各种GPU计算上的优化(后边会提到)，每次迭代总耗时可以减少到1ms以内，但是只为了找到一个步长的上限，使用CCD却需要1ms以上的耗时 。所以现在关键就是怎么找到一个好的步长，最好是直接一次性得到一个合适的步长。

经过各种的测试，我们发现经典的牛顿法线搜索的效果最好。这里的牛顿法仅用于线搜索，跟前面的牛顿法解优化问题无关。其实就是通过Tayler展开进行估计

$$E(\mathbf{x}+\alpha\mathbf{p}) \approx E(\mathbf{x}) + \alpha\mathbf{g}^\top\mathbf{p} + \frac{\alpha^2}{2}\mathbf{p}^\top\mathbf{H}\mathbf{p} \tag{9}$$

其中g是梯度，p 是更新方向。计算能量E关于 步长alpha的梯度

$$\frac{\partial E(\mathbf{x}+\alpha\mathbf{p})}{\partial\alpha} \approx \mathbf{g}^\top\mathbf{p} + \alpha\mathbf{p}^\top\mathbf{H}\mathbf{p} \tag{10}$$

令其等于0可得

$$\bar{\alpha} = -\frac{\mathbf{g}_{k+1}^\top\mathbf{p}_{k+1}}{\mathbf{p}_{k+1}^\top\mathbf{H}_{k+1}\mathbf{p}_{k+1}}. \tag{11}$$

也就是根据当前的梯度和海森矩阵一次性估计出合适的步长，这样就避免了回溯的过程。这一步计算量主要在分母的矩阵-向量乘法上，虽然这一块的时间消耗会稍多一些，但是经过优化之后这一块的计算时间还是可以接受的，最大的好处是每次迭代仅需计算一次所有的碰撞对。而且因为使用了二阶导的信息，我们发现，大部分的时候这个步长是相对准确的。于是一个很自然的想法，既然已经预估出了一个合适的步长，那么是不是有可能可以直接去掉CCD?

实际上还是会出现某些情况下更新方向和步长计算会出现错误，因为参数alpha 和 beta 都是一个常数，针对所有点计算出的，即使计算出的整体优化的方向是对的，对于某个具体的点却并不是合适的。当步长的估计过大时，可能出现少数的点的穿透的问题。这个情况在复杂的碰撞场景和物体发生较大的形变是出现的频率更高。所以很简单的方法就是直接限制步长的上限，

$$\alpha = \min(\alpha_{\text{upper}}, \bar{\alpha}),\ \alpha_{\text{upper}} = \frac{\hat{d}}{2\|\mathbf{p}_{k+1}\|_\infty},$$

即每次迭代所有点的位移的上限是 $\frac{1}{2}\hat{d}$, 虽然限制alpha会影响到收敛的速度，使得需要更多的迭代次数，但因为每次的迭代的速度足够快，增加的耗时与耗时量大的CCD相比微乎其微。

总而言之，这个策略的核心想法就是不去计算出真正的步长的上限，而是直接用一个保守的步长上限，然后再根据已知的信息估计出一个合适的步长。在实践中我们发现在$\hat{d}, \kappa$ 合理设置的前提下，该策略确实可以在相对复杂的场景下不需要任何的碰撞检测仍然取得无穿透的效果。而且触发步长上限的频率并不高，仅有 1/20 左右，且都出现在前几次迭代中。

我们的另一个优化的策略就是用更大的$\hat{d}$, 这个想法是很直接的，比如假定barrier function 的最优解的值在区间[5e-6,1e-5]上。 $\hat{d}$ =1e-3, 时对应的是d 是[5e-8, 1e-5]， 而 $\hat{d}$ =2e-3时，对应的是d 的区间是是[1e-4, 3e-4]，显然使用更大的 $\hat{d}$ 对应b 的值的分布更均匀一些，不会全集中在接近于0的很小的区间内，更容易找到合适的最优解，算法也就更容易收敛。当前用太大的$\hat{d}$ 也会导致碰撞的时候出现“缝隙”，所以需要选择一个不大不小的$\hat{d}$ ，目前是手动设置的，个人经验是选择大概所有边的平均值的30% 左右。当然，更大的$\hat{d}$ 对应的碰撞对也会更多，不过因为可以并行，所以这其中增加的计算时间并不多。

这里我们还对终止条件做了一点修改

$$\Delta E = E(\mathbf{x}_k) - E(\mathbf{x}_{k+1}) \approx -\alpha \mathbf{g}_{k+1}^\top \mathbf{p}_{k+1} - \frac{\alpha^2}{2}\mathbf{p}_{k+1}^\top \mathbf{H}_{k+1}\mathbf{p}_{k+1}.$$

根据前边公式很容易得出能量变换的估计，因为这些值在前边计算步长的时候已经计算过了，所以完全不需要额外的计算。实践中发现这个估计值可以很好地判断模型的是否已经收敛。

所以整个算法的pipeline 如下:

ALGORITHM 1: Preconditioned Nonlinear Conjugate Gradient

$\mathbf{x}_0 \leftarrow x^t$
$\tilde{x} = x^t + hv^t + h^2 M^{-1} f_{ext}$
for $k = 0$ *to IterMax* **do**
  $C \leftarrow \text{ComputeConstraintSet}(x, \hat{d})$
  $\mathbf{g}_{k+1}, \mathbf{P}_{k+1} \leftarrow$
   $\text{ComputeGradientAndPreconditioning}(x, \tilde{x}, C)$
  if $k = 0$ **then**
    $\beta_k \leftarrow 0$
  else
    $\beta_k \leftarrow \text{ComputeBeta}(\mathbf{g}_{k+1}, \mathbf{g}_k, \mathbf{p}_{k+1}, \mathbf{p}_k)$
  end
  $\mathbf{p}_{k+1} \leftarrow -\mathbf{P}_{k+1}\mathbf{g}_{k+1} + \beta_k \mathbf{p}_k$
  $\alpha \leftarrow \min(\frac{\hat{d}}{2\|\mathbf{p}_{k+1}\|_\infty}, -\frac{\mathbf{g}_{k+1}^\top \mathbf{p}_{k+1}}{\mathbf{p}_{k+1}^\top \mathbf{H}_{k+1}\mathbf{p}_{k+1}})$.
  $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha \mathbf{p}_{k+1}$
  $\Delta E \leftarrow -\alpha \mathbf{g}_{k+1}^\top \mathbf{p}_{k+1} - \frac{\alpha^2}{2}\mathbf{p}_{k+1}^\top \mathbf{H}_{k+1}\mathbf{p}_{k+1}$
  if $i = 0$ **then**
    $\Delta E_0 \leftarrow \Delta E$
  end
  if $\Delta E < \epsilon \Delta E_0$ **then**
    break
  end
end

可以看出来我们的算法算是很简单朴素的。

我们在计算中发现Hessian矩阵相关的计算耗时很大，因此就对这部分也进行了优化。

首先是弹性项，这里我们参考了
https://www.tkim.graphics/DYNAMIC_DEFORMABLES/

Dynamic Deformables:Implementation and Production Practicalities

的记号，强烈安利这本书！写的太好了。

以Neo-Hookean 为例:

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

其中

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

每一项对应的梯度和hessian矩阵如下

$$\begin{aligned}
\mathbf{g}_1 &= \mathrm{vec}(\mathbf{R}), & \mathbf{H}_1 &= \sum_{i=0}^{2} \lambda_i \mathbf{q}_i \mathbf{q}_i^\top, \\
\mathbf{g}_2 &= 2\,\mathrm{vec}(\mathbf{F}), & \mathbf{H}_2 &= 2\mathbf{I}_{9\times 9}, \\
\mathbf{g}_3 &= \mathrm{vec}\left(\left[\mathbf{f}_1 \times \mathbf{f}_2, \mathbf{f}_2 \times \mathbf{f}_0, \mathbf{f}_0 \times \mathbf{f}_1\right]\right), & \mathbf{H}_3 &= \begin{bmatrix} \mathbf{0} & -\hat{\mathbf{f}}_2 & \hat{\mathbf{f}}_1 \\ \hat{\mathbf{f}}_2 & \mathbf{0} & -\hat{\mathbf{f}}_0 \\ -\hat{\mathbf{f}}_1 & \hat{\mathbf{f}}_0 & \mathbf{0} \end{bmatrix},
\end{aligned}$$

于是可以得到

$$\begin{aligned}
\frac{\partial^2 \Psi}{\partial \mathbf{x}^2} &= \mathrm{vec}\left(\frac{\partial \mathbf{F}}{\partial \mathbf{x}}\right)^\top \mathrm{vec}\left(\frac{\partial^2 \Psi}{\partial \mathbf{F}^2}\right) \mathrm{vec}\left(\frac{\partial \mathbf{F}}{\partial \mathbf{x}}\right) \\
&= \mathrm{vec}\left(\frac{\partial \mathbf{F}}{\partial \mathbf{x}}\right)^\top \left(\sum_{i=1}^{3} (\frac{\partial^2 \Psi}{\partial I_i^2} \mathbf{g}_i \mathbf{g}_i^\top + \frac{\partial \Psi}{\partial I_i} \mathbf{H}_i)\right) \mathrm{vec}\left(\frac{\partial \mathbf{F}}{\partial \mathbf{x}}\right),
\end{aligned}$$

这里我们要对Hessian的对角线元素和$p^\top Hp$ 进行计算上的优化。常用的方法就是先算出12x12的Hessian矩阵，再去算对角线和矩阵向量乘法。我们的思路有些不同。我们将Hessian矩阵分成六块

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

其中

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

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

于是

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

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

然后对这六块的计算都进行优化。其实就是一些基本的向量代数计算。因为有许多项都是$A^\top A$ 的形式的，所以其实很容易优化。我们甚至是直接用sympy把具体的公式推出来。具体的计算公式可以看我们的supplemental document和源码.

结果发现这个简单的操作其实能极大地减少计算量。与先算出整个12x12的$\frac{\partial^2 \Psi}{\partial \mathbf{x}^2}$ 然后再算$\overline{diag}\left(\frac{\partial^2 \Psi}{\partial \mathbf{x}^2}\right)$ 和 $\mathbf{p}^\top \frac{\partial^2 \Psi}{\partial \mathbf{x}^2}\mathbf{p}$ 相比， 计算量分别只有约1/3和1/6。

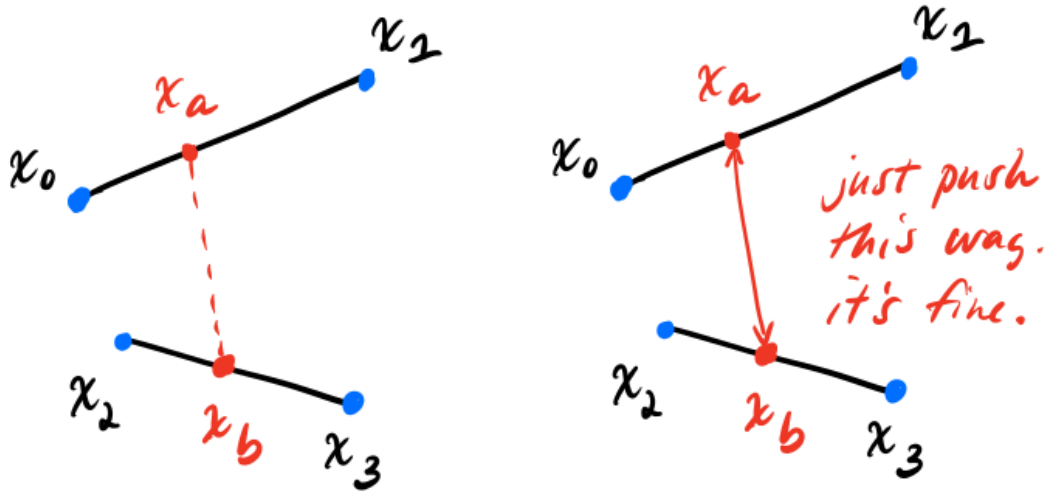
另外梯度和$\overline{diag}\left(\frac{\partial^2 \Psi}{\partial \mathbf{x}^2}\right)$ 的计算是可以放到同一个for-loop里，所以可以用很小的开销一块算出来。

另一个部分就是IPC的barrier function, 核心想法还是怎么不去计算12x12的矩阵。

IPC算法的barrier fucntion的距离d 是由Point-triangle , Edge-edge对 之间最小的距离定义的。我们这边做一点小改动

$$\begin{aligned} d = \|\mathbf{t}\| = \min_{c_0,c_1,c_2,c_3} \|c_0\mathbf{x}_0 + c_1\mathbf{x}_1 + c_2\mathbf{x}_2 + c_3\mathbf{x}_3\|, \qquad (12) \\ s.t.\ c_0 = 1,\ c_1, c_2, c_3 \le 0,\ c_1 + c_2 + c_3 = -1 \text{ for PT primitives}, \\ c_0, c_1 \ge 0,\ c_2, c_3 \le 0,\ c_0 + c_1 = 1,\ c_2 + c_3 = -1 \text{ for EE primitives}. \end{aligned}$$

以边-边对为例， x_a, x_b 是两条边上最近的点，且都可以由两个顶点线性组合得到。
$\mathbf{t} = |x_a - x_b|$



则t 关于x 的梯度和hessian矩阵就可以得到了

$$\frac{\partial \mathbf{t}}{\partial \mathbf{x}} = \begin{bmatrix} c_0\mathbf{I}_{3\times3} & c_1\mathbf{I}_{3\times3} & c_2\mathbf{I}_{3\times3} & c_3\mathbf{I}_{3\times3} \end{bmatrix}^\top \in \mathcal{R}^{12\times3}, \frac{\partial^2 \mathbf{t}}{\partial \mathbf{x}^2} = \mathbf{0},$$

这里做了一个简化，即认为$\frac{\partial c}{\partial x} = 0$.这样t 的海森矩阵就直接变成0了。这个操作还是从**Dynamic Deformables** 上学来的，这里就是做了一个整合，把PT,EE的计算公式合并起来，这样就能放到一起并行了。于是barrier function 的公式就可以得到了

$$\frac{\partial b}{\partial \mathbf{x}} = \frac{\partial b}{\partial d}\frac{\partial d}{\partial \mathbf{t}}\frac{\partial \mathbf{t}}{\partial \mathbf{x}} = \frac{\partial b}{\partial d}\frac{1}{d}\frac{\partial \mathbf{t}}{\partial \mathbf{x}}\mathbf{t}, \tag{13}$$

$$\frac{\partial^2 b}{\partial \mathbf{x}^2} = (\frac{\partial^2 b}{\partial d^2}\frac{1}{d^2} - \frac{\partial b}{\partial d}\frac{1}{d^3})(\frac{\partial \mathbf{t}}{\partial \mathbf{x}}\mathbf{t})(\frac{\partial \mathbf{t}}{\partial \mathbf{x}}\mathbf{t})^\top + \frac{\partial b}{\partial d}\frac{1}{d}(\frac{\partial \mathbf{t}}{\partial \mathbf{x}})(\frac{\partial \mathbf{t}}{\partial \mathbf{x}})^\top. \tag{14}$$

所以有

$$\overline{diag}((\frac{\partial \mathbf{t}}{\partial \mathbf{x}})(\frac{\partial \mathbf{t}}{\partial \mathbf{x}})^\top) = [c_0, c_0, c_0, c_1, c_1, c_1, c_2, c_2, c_2, c_3, c_3, c_3]^{\circ 2}, \tag{15}$$

$$\overline{diag}((\frac{\partial \mathbf{t}}{\partial \mathbf{x}}\mathbf{t})(\frac{\partial \mathbf{t}}{\partial \mathbf{x}}\mathbf{t})^\top) = [c_0\mathbf{t}^\top, c_1\mathbf{t}^\top, c_2\mathbf{t}^\top, c_3\mathbf{t}^\top]^{\circ 2}, \tag{16}$$

$$\mathbf{p}^\top(\frac{\partial \mathbf{t}}{\partial \mathbf{x}}t)(\frac{\partial \mathbf{t}}{\partial \mathbf{x}}t)^\top\mathbf{p} = (\mathbf{p}^\top\frac{\partial \mathbf{t}}{\partial \mathbf{x}}t)^2, \tag{17}$$

$$\mathbf{p}^\top(\frac{\partial \mathbf{t}}{\partial \mathbf{x}})(\frac{\partial t}{\partial \mathbf{x}})^\top\mathbf{p} = \|\mathbf{p}^\top\frac{\partial \mathbf{t}}{\partial \mathbf{x}}\|_2^2, \tag{18}$$

由此IPC部分的Hessian计算也可以简化了。

这个操作后来我们在今年另一篇SIGGRAPH文章 **GIPC: Fast and Stable Gauss-Newton Optimization of IPC Barrier Energy**上也看到了类似的操作，不过他们做了比较详细的分析，我们就是直接用了，这个操作具体的理论和实验的分析我们之后会借鉴这篇文章也做一个。

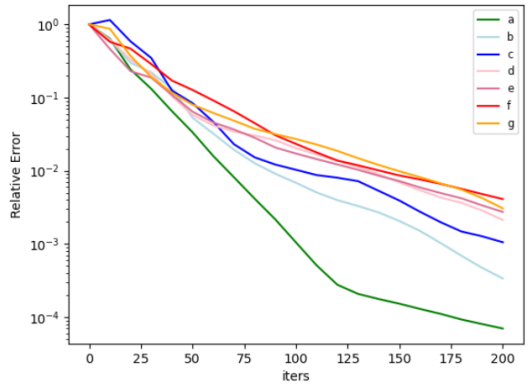
至此算法部分介绍就差不多了。具体实现我们全程使用的是Taichi, 也使用了MeshTaichi库加快mesh层面的并行。数据结构用的是spatial hashing, 因为容易写。不过Taichi有很多坑没填上，现在也不怎么更新了，很多东西都要自己写，我们已经在考虑使用Nvidia Warp 了。

代码已经在Git上开源了https://github.com/Xingbaji/PNCG_IPC/，中间有挺多优化地不是很好的地方，但是有点改不动了，之后会再整个Warp版的弄上去。(算法没有用CCD最大的原因其实是我自己写的CCD太烂了，最后一气之下干脆就不用了，结果发现好像不用也行(。﹏。))

下边是一些DEMO:

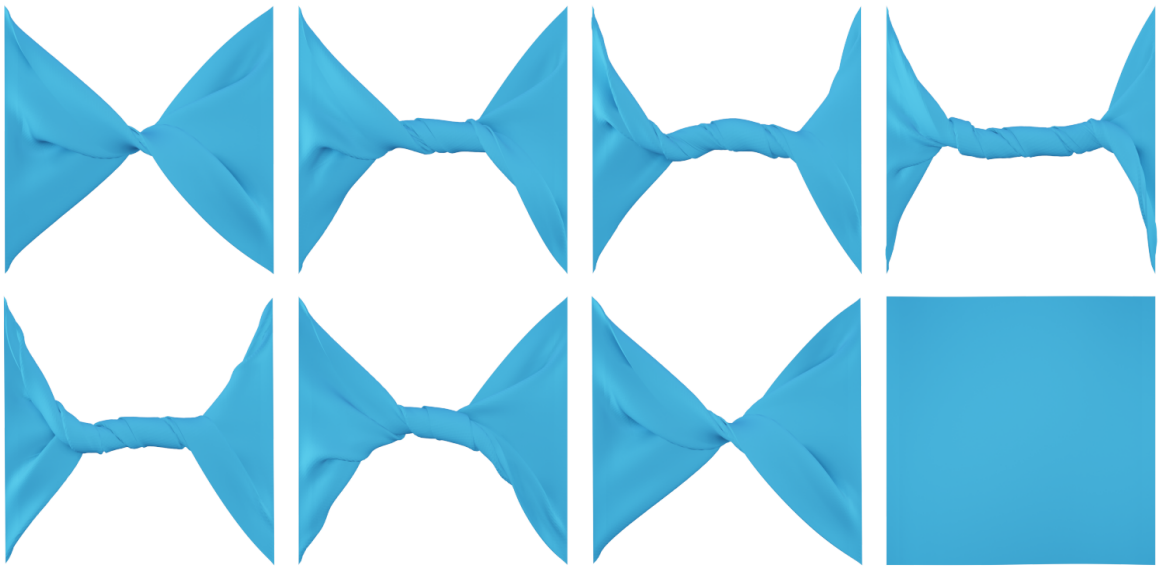
Scene	Material	# nodes, # tets # faces	# contact avg(max)	IterMax	Tolerance ϵ	avg Iters	FPS avg(min)
Drag armadillo (Fig. 3c)	SNH	13K, 55K, 22K	0K (0K)	150	5×10^{-5}	108.1	40.1 (25.0)
8 "E" falling (Fig. 8)	NH	8K, 27K, 13K	2K (8K)	50	3×10^{-4}	32.8	46.6 (24.3)
48 "E" falling (Fig. 9)	ARAP	50K, 164K, 78K	63K (99K)	25	1×10^{-3}	25.0	32.2 (27.9)
Four long noodles (Fig. 4)	ARAP	39K, 101K, 73K	13K (29K)	25	1×10^{-3}	24.9	27.6 (25.8)
Squeeze four armadillo (Fig. 6)	ARAP	53K, 168K, 91K	71K (322K)	25	1×10^{-3}	24.1	28.9 (25.6)
Twist mat (Fig. 7)	FCR	45K, 133K, 90K	87K (222K)	150	1×10^{-3}	138	7.4 (5.6)
Twist four rods (Fig. 5)	FCR	53K, 202K, 80K	39K (92K)	150	1×10^{-3}	107.5	10.2 (6.9)

第一个是测极端形变下的效果



即使是极端形变下算法150次迭代也收敛差不多, 因为每次的迭代足够快，所以这个DEMO也是能跑到实时的。

然后是经典Twist mat 的demo



也就是两端旋转两圈然后再转回去，可以看到最后是平整的转回去了，说明没有发生穿透。这里用的kappa 和$\hat{d}$ 全程都是固定值。但是这两个值是经过了手调的。不过这也说明了及时不用任何的碰撞检测，我们这个算法在这种场景下还是能不穿透，还是挺神奇的，这个效果我们一开始都没想到。当然如果继续旋转下去的话需要动态调整kappa的值才能继续保持不穿透。但是这时候因为中间部分弹性力会很大，对应的kappa的值很难调整。我们也没找到合适的自适应参数调整的机制，IPC原文中有提到类似的方法，但是好像不适用于我们这个情况。所以可以认为我们这一套方法能在相对复杂的场景下不穿透，但是极端的场景是没办法的。不过这个效果其实也能够满足大部分的模拟需求了。

这个demo速度是7FPS, 虽然没弄到实时，不过相比原始版本的33s每帧还是有差不多200倍的加速。

以及8 个“E”和48个E的demo

Figure 8: Free fall of 8 "E" models.

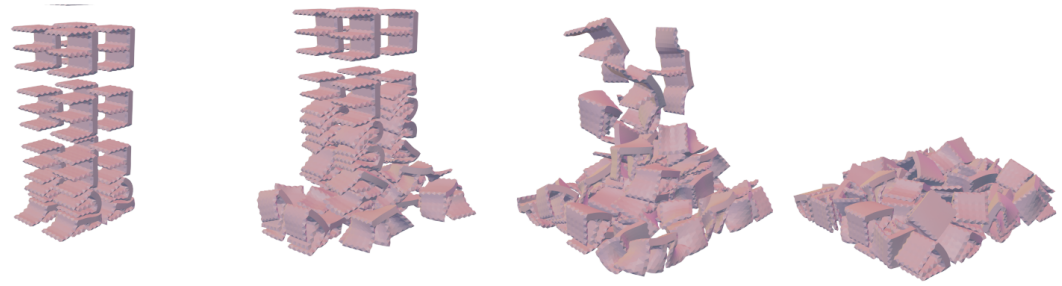


Figure 9: Free fall of stacked 48 "E" models.

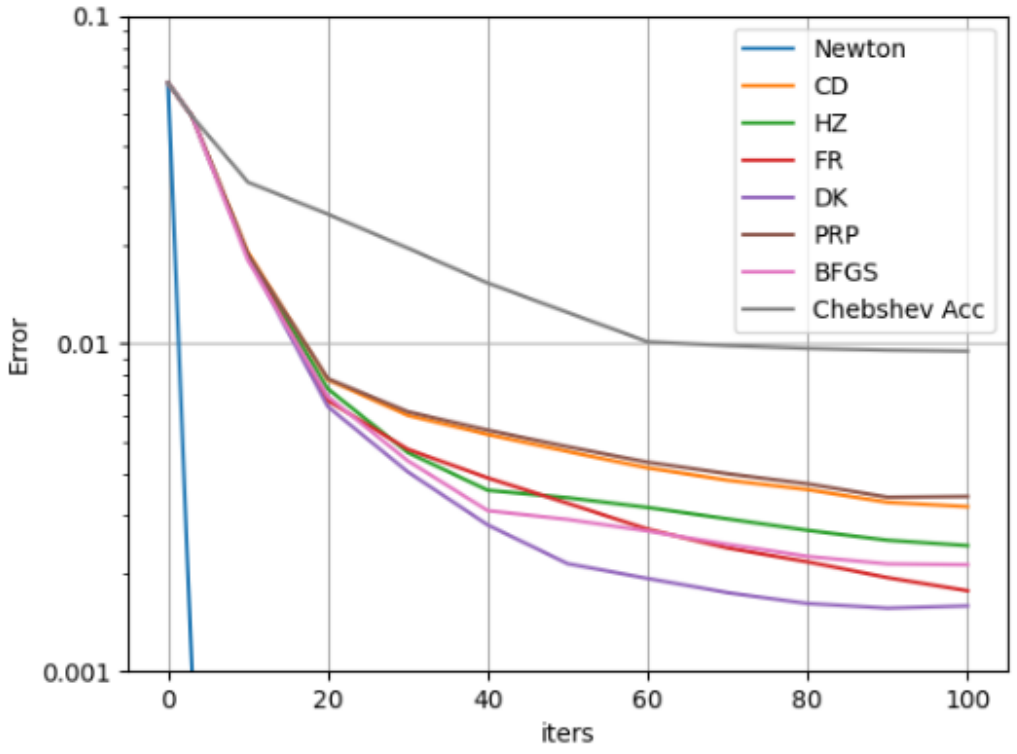
笔者个人很喜欢用字母“E”跑demo, 因为形变可以看得很清晰，有没有穿透也很容易看出来。

48 E的速度虽然号称实时，但是是有点tricky的，这边我们把重力调小了。 这是因为最上边的E掉下来的最终速度会很快，然后底下的E几乎没有惯性能量，上边的E惯性能量又很大，单凭一个固定值的kappa很难取得平衡。

然后是挤压Armadilli的demo. 为了刷到实时，我们没有把算法完全跑收敛，中间一些迭代提早结束了(Itermax设到了25，实际上要35左右)。

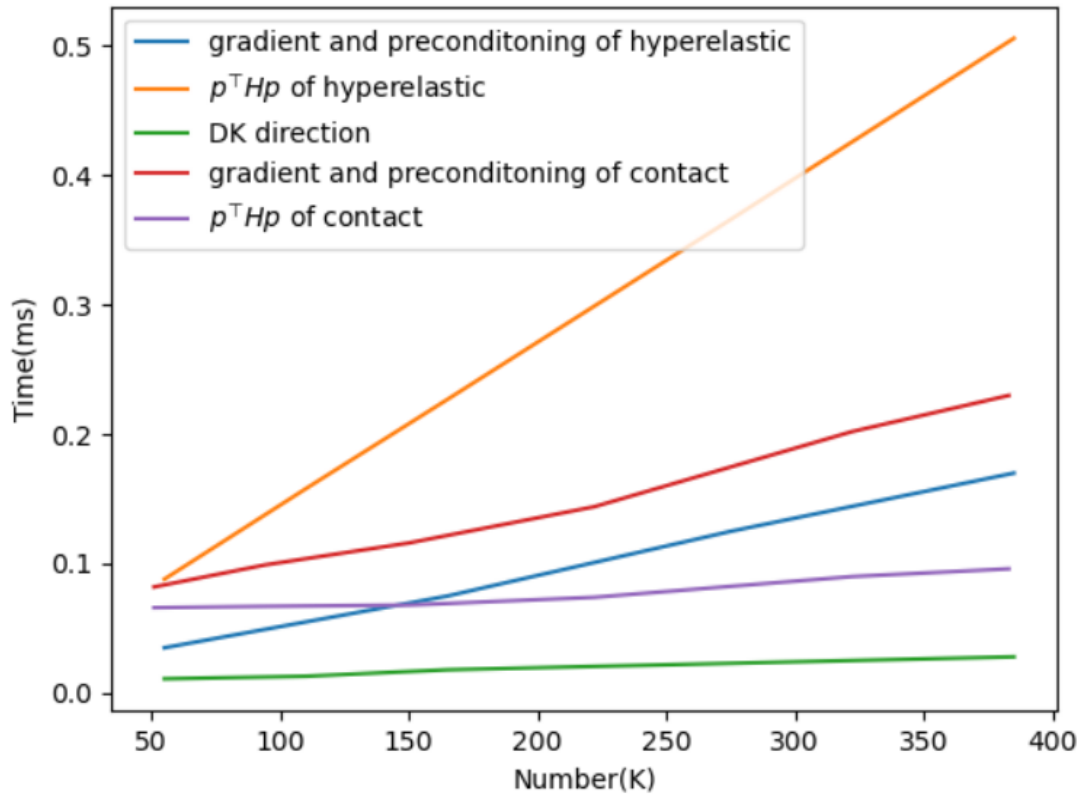
Figure 6: Squeeze and release of four armadillo models. We simultaneously compress the four small models from all six sides and then release them.

以及一些对照实验。收敛速度牛顿法还是有明显的优势的，毕竟PNCG还是一个一阶的优化算法。我们测了几个常见的NCG算法，最后发现DK算法效果最好。其实应该加上跟Projective Dynamics方法的比较的，但是当时来不及了。



(a) Convergence of different methods. DK algorithm demonstrates superior performance compared to other PNCG algorithms (CD,HZ,FR,PRP), and significantly outperforms the Chebyshev accelerated preconditioned descent method. Error is evaluated using the infinity norm with respect to the ground truth.

然后是每一块的计算速度



可以看出经过我们的优化，每一块的计算时间都很小。对于200K个四面体的模型每次迭代的计算总时间还是能控制在1ms以下的。所以即使牛顿法收敛速度比我们的好，但是速度上我们的还是遥遥领先的。

更多的demo可以参考原文和我们的补充材料
https://dl.acm.org/doi/10.1145/3641519.3657490。

总而言之，算法的速度和收敛效果都是很不错的，不过real-time的IPC还是有点overclaim了。写的时候确实有点慌，因为是纯图形学新人，所以当时想着把指标弄好看点是不是能更好中，结果被审稿人一眼看穿。后来也不好改，结果demo就这样子了。论文刚提交上去的时候写的非常混乱，对照实验也几乎没有。结果5个reviewer都是肯定了我们的idea，但是写作和实验被喷惨了，感谢审稿人和AC最后抬了一手.

当然这个论文还有很多不完善的地方，摩擦力也是因为时间来不及没有加上去。非线性的共轭梯度法这个坑还有很多可以填的，个人觉得这个算法前景还是很不错的，有很多继续提升的空间，现在这个感觉还是初级版本。以及ccd 其实完全可以加进去的。很多实现上的细节也没有写好。代码在https://github.com/Xingbaji/PNCG_IPC/上开源了。

我们现在已经有一些很不错的idea, PNCG V2已经在搞了。希望下一波能真的把IPC算法弄到实时。欢迎感兴趣的同学一起讨论交流。