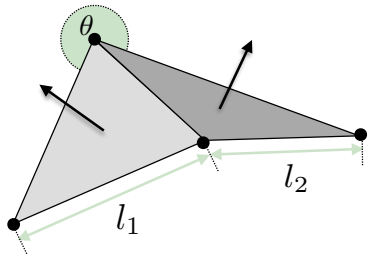
XPBD: POSITION-BASED SIMULATION OF COMPLIANT CONSTRAINED DYNAMICS

Miles Macklin, Matthias Müller, Nuttapong Chentanez

PROBLEM

- Position-Based Dynamics (PBD) stiffness depends on the number of iterations and time step size
- Converges to an infinitely stiff solution given enough iterations
- Makes setting relative stiffness difficult
- Stiffness does not have any physical basis
- No concept of a constraint force magnitude



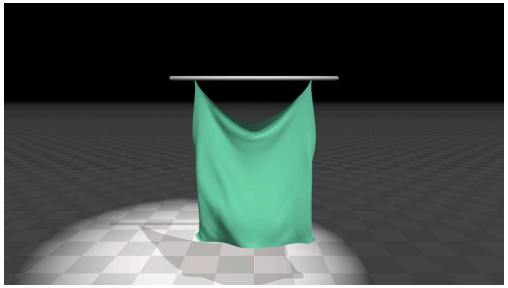
The long standing problem of PBD is that the stiffness depends on iteration count. Regardless of how you set the stiffness coefficients in PBD, given enough iterations it will converge to an infinitely stiff solution, effectively overriding the stiffness values.

For example in a cloth model, we have bending and stretch constraints. If we need to increase iterations to reduce stretching, we inadvertently increase resistance to bending. This makes it difficult to set up relative stiffness in simulation assets, and difficult to create re-usable assets.

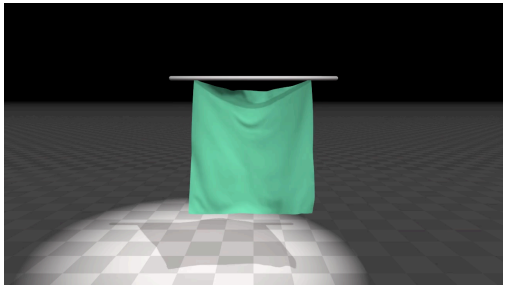
In addition, stiffness does not have a physical basis in PBD. It does not correspond to a traditional engineering stiffnesss. A connection to real-world physics would be nice in order to match real-world material parameters.

Finally, PBD does not have a concept of constraint force magnitude. Although constraints may be solved effectively, we have no idea how much work they have done. We need this information to drive force dependent effects such as haptic feedback, particle system spawning, motors and motor limits.

PBD ITERATION DEPENDENT STIFFNESS



20 ITERATIONS



160 ITERATIONS

This example shows how PBD's behavior changes with iteration count. Ideally these two pieces of cloth should behave exactly the same, but we have much stiffer cloth at high iteration counts despite having used a low stiffness coefficient.

CONSTRAINED DYNAMICS

- Start with Newton's equations of motion given an energy potential U(q)
- Define potential function U(x) in terms of constraint vector C(x)
- Compliance $\boldsymbol{\alpha} = \mathbf{S}^{-1}$

Equations of motion (Newton's 2nd Law)

$$\mathbf{M}\ddot{\mathbf{x}} = -\nabla U^T(\mathbf{x})$$

Elastic potential

$$U(\mathbf{x}) = \frac{1}{2}\mathbf{C}(\mathbf{x})^T\boldsymbol{\alpha}^{-1}\mathbf{C}(\mathbf{x})$$

NVIDIA

I want to start by deriving PBD from a more principled basis. We start here with our equations of motion, Newton's second law, F=Ma. Here F is derived from an energy potential U, which we define as a quadratic form in terms of the constraint functions C.

We prefer to work in terms of compliance which is defined as inverse stiffness. Here alpha is a diagonal, or block diagonal compliance matrix.

CONSTRAINED DYNAMICS

- Introduce Lagrange multiplier
- Combine with our equations of motion to obtain compliant constrained equations of motion
- [Servin et al 2006]

Lagrange multiplier

$$\boldsymbol{\lambda} = -\boldsymbol{\alpha}^{-1}\mathbf{C}(\mathbf{x})$$

Compliant constrained equations of motion

$$\mathbf{M}\ddot{\mathbf{x}} - \nabla\mathbf{C}^{T}(\mathbf{x})\boldsymbol{\lambda} = \mathbf{0}$$
$$\mathbf{C}(\mathbf{x}) + \boldsymbol{\alpha}\boldsymbol{\lambda} = \mathbf{0}$$

NVIDIA

We can further re-write the equations of motion by introducing a new variable lambda. This is our Lagrange multiplier. Plugging it into the equations of motion we obtain the familiar constrained equations of motion, with the addition of this compliant alpha*lambda term in the second equation. This can be thought of as a regularization term that makes our constraints correspond to the energy potential we defined initially.

TIME DISCRETIZATION

- Implicit time discretization
- Introduce predicted position

$$\tilde{\mathbf{x}} = \mathbf{x}^n + \Delta t \mathbf{v}^n$$

- Introduce time-scaled compliance:

$$\tilde{\boldsymbol{\alpha}} = \frac{\boldsymbol{\alpha}}{\Delta t^2}$$

Continuous Constrained Equations of Motion

$$\mathbf{M}\ddot{\mathbf{x}} - \nabla \mathbf{C}^T(\mathbf{x})\boldsymbol{\lambda} = \mathbf{0}$$
$$\mathbf{C}(\mathbf{x}) + \boldsymbol{\alpha}\boldsymbol{\lambda} = \mathbf{0}$$

$$\mathbf{M}(\mathbf{x}^{n+1} - \tilde{\mathbf{x}}) - \nabla \mathbf{C}(\mathbf{x}^{n+1})^T \boldsymbol{\lambda}^{n+1} = \mathbf{0}$$
$$\mathbf{C}(\mathbf{x}^{n+1}) + \tilde{\boldsymbol{\alpha}}\boldsymbol{\lambda}^{n+1} = \mathbf{0}$$

Discrete Constrained Equations of Motion

NVIDIA

Now we have our equations of motion we need to solve them. This is a differential algebraic equation of order three (DAE3), a notoriously difficult problem to solve well. We take a simple approach of performing an implicit time-discretization on positions.

Here we have introduced two new variables, tilde(x) which is sometimes called the inertial, predicted, or unconstrained position. This is the state the system would obtain if we integrated it forward in time without considering the constraints.

tilde(alpha) is simply a time-step scaled version of compliance that we use to make the equations more compact.

NEWTON'S METHOD

- Implicit time discretization produces a non-linear system of equations
- How do we solve such a system?
- Newton's method

Implicit Discrete Equations of Motion

$$\mathbf{M}(\mathbf{x}^{n+1} - \tilde{\mathbf{x}}) - \nabla\mathbf{C}(\mathbf{x}^{n+1})^T\boldsymbol{\lambda}^{n+1} = \mathbf{0}$$
$$\mathbf{C}(\mathbf{x}^{n+1}) + \tilde{\boldsymbol{\alpha}}\boldsymbol{\lambda}^{n+1} = \mathbf{0}$$

$$\mathbf{g}(\mathbf{x}^{n+1}, \boldsymbol{\lambda}^{n+1}) = \mathbf{0}$$
$$\mathbf{h}(\mathbf{x}^{n+1}, \boldsymbol{\lambda}^{n+1}) = \mathbf{0}$$

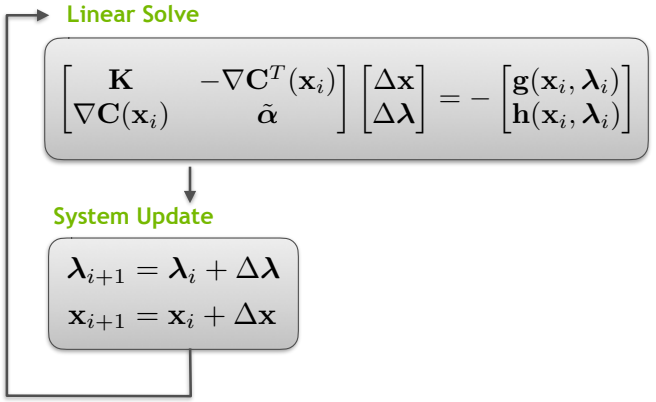
Non-Linear System

NVIDIA

Now we have our discretized equations we need to solve them. These are non-linear equations, and the standard way to solve non-linear equations is Newton's method. We will apply Newton's method, but first we re-label the equations as g() and h() respectively.

NEWTONS METHOD

- Linearize g,h, around current iterate x_i, lambda_i
- Update system with delta
- Works well, but may need line-search / trust-region to be robust



Newton's method involves first linearizing the equations to obtain a linear system that we solve for deltaX and deltaLambda. These are then applied directly to the system, and the process repeated until the deltas fall below some tolerance value.

This method works well, although it can be difficult to implement. Computing K involves the second derivatives (Hessians) of the constraint functions, and it may require a line-search or trust region method to converge. We now seek a simpler approach based on PBD, but to do that we need to make some approximations..

APPROXIMATE NEWTON STEP

First approximation:

- M = K + O(dt^2)
- Common Quasi-Newton simplification

Second approximation:

- g = 0
- True for first iteration
- Typically remains small

Full Newton System

$$\begin{bmatrix} \mathbf{K} & -\nabla\mathbf{C}^T(\mathbf{x}_i) \\ \nabla\mathbf{C}(\mathbf{x}_i) & \tilde{\boldsymbol{\alpha}} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{x} \\ \Delta\boldsymbol{\lambda} \end{bmatrix} = - \begin{bmatrix} \mathbf{g}(\mathbf{x}_i, \boldsymbol{\lambda}_i) \\ \mathbf{h}(\mathbf{x}_i, \boldsymbol{\lambda}_i) \end{bmatrix}$$

Approximate System

$$\begin{bmatrix} \mathbf{M} & -\nabla\mathbf{C}^T(\mathbf{x}_i) \\ \nabla\mathbf{C}(\mathbf{x}_i) & \tilde{\boldsymbol{\alpha}} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{x} \\ \Delta\boldsymbol{\lambda} \end{bmatrix} = - \begin{bmatrix} \mathbf{0} \\ \mathbf{h}(\mathbf{x}_i, \boldsymbol{\lambda}_i) \end{bmatrix}$$

NVIDIA

The first approximation is to replace K with M. This drops some terms on the order of dt^2, it may affect converge rate, but it doesn't change the solution of our Newton iteration. Sometimes this is called a quasi-Newton method.

The next approximation is to treat g(x, lambda) as zero for all iterations. We make two observations to justify this approximation. First, if we initialise the Newton iteration with x0 = tilde(x), and lambda0=0, this means g(x0, lambda0) is automatically zero, so the first Newton iteration is unchanged. In addition, g() typically remains small because it is related to the change in the constraint gradient direction. If constraints are linear, then g() is always zero. In practice g() is close to linear for small time-steps.

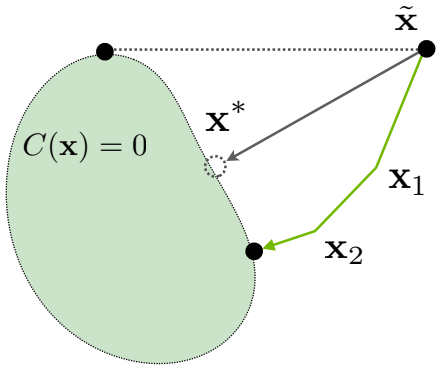
VARIATIONAL VIEWPOINT

Implicit Euler

$$\begin{aligned} &\text{minimize} && \frac{1}{2}(\mathbf{x}-\tilde{\mathbf{x}})^T\mathbf{M}(\mathbf{x}-\tilde{\mathbf{x}}) \\ &\text{subject to} && \mathbf{C}(\mathbf{x})=0 \end{aligned}$$

PBD

$$\begin{aligned} &\text{minimize} && \frac{1}{2}(\mathbf{x}-\mathbf{x}_i)^T\mathbf{M}(\mathbf{x}-\mathbf{x}_i) \\ &\text{subject to} && \mathbf{C}(\mathbf{x})=0 \end{aligned}$$



NVIDIA

Some further intuition can be obtained by looking at this from a variational point of view. We can view implicit Euler as trying to find the closest point on the constraint manifold to the predicted or internal position, tilde(x). In contrast, PBD tries to find a close, but not necessarily the closest point on the manifold. Each iteration solves a slightly different minimization, starting from the current iterate, this leads to this ‘curved path’ back to the manifold. As soon as PBD finds a point on the manifold it stops, however there is no guarantee that this is the closest point. In practice the difference is small, but this is the consequence of our approximation that g() = 0.

CONNECTION TO PBD

- Take Schur complement of approximate system with respect to M
- Obtain PBD or Fast Projection form
- [Goldenthal et al 2007]

Approximate System

$$\begin{bmatrix} \mathbf{M} & -\nabla\mathbf{C}^T(\mathbf{x}_i) \\ \nabla\mathbf{C}(\mathbf{x}_i) & \tilde{\boldsymbol{\alpha}} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{x} \\ \Delta\boldsymbol{\lambda} \end{bmatrix} = - \begin{bmatrix} \mathbf{0} \\ \mathbf{h}(\mathbf{x}_i, \boldsymbol{\lambda}_i) \end{bmatrix}$$

Schur complement

$$\left[\nabla\mathbf{C}(\mathbf{x}_i)\mathbf{M}^{-1}\nabla\mathbf{C}(\mathbf{x}_i)^T + \tilde{\boldsymbol{\alpha}}\right] \Delta\boldsymbol{\lambda} = -\mathbf{C}(\mathbf{x}_i) - \tilde{\boldsymbol{\alpha}}\boldsymbol{\lambda}_i$$

NVIDIA

We can now make a connection back to PBD by taking the Schur complement (just plug the expression for deltaX into the second equation). This gives a familiar constrained system, with the addition of the compliance terms. This is equivalent to a compliant version of Goldenthal et al's Fast Projection algorithm.

So far we have written things out in terms of the system as a whole, but one of the benefits of PBD is that it lets us work in terms of single constraints. So let's see what that looks like..

A GAUSS-SEIDEL UPDATE

- View PBD “scaling fator” *s* as incremental Lagrange multiplier
- Additional compliance terms
- Must store Lagrange multiplier for each constraint
- PBD solves the infinite stiffness case

PBD

$$s_j = \frac{-C_j(\mathbf{x}_i)}{\nabla C_j \mathbf{M}^{-1} \nabla C_j^T}$$

XPBD

$$\Delta\lambda_j = \frac{-C_j(\mathbf{x}_i) - \tilde{\alpha}_j \lambda_{ij}}{\nabla C_j \mathbf{M}^{-1} \nabla C_j^T + \tilde{\alpha}_j}$$

NVIDIA

Here we have the original “scaling factor” from PBD, and the new deltaLambda term from XPBD. We can see that when alpha == 0 (infinite stiffness, or zero compliance) that the two equations are the same. From this we can see that s in PBD is actually an incremental Lagrange multiplier change for the infinitely stiff case. This makes sense, as it’s the behavior we observe in PBD as we increase iteration count.

When alpha > 0 we have some additional terms in the numerator and denominator which act to regularise the constraint.

XPBD ALGORITHM

- Only two differences from PBD:
- Lagrange multiplier calculation (include compliance terms)
- Lagrange multiplier update (store instead of discard)

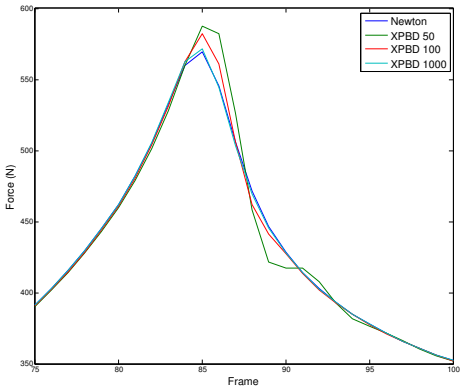
1: predict position $\tilde{\mathbf{x}} \Leftarrow \mathbf{x}^n + \Delta t \mathbf{v}^n + \Delta t^2 \mathbf{M}^{-1} \mathbf{f}_{ext}(\mathbf{x}^n)$
2:
3: initialize solve $\mathbf{x}_0 \Leftarrow \tilde{\mathbf{x}}$
4: initialize multipliers $\boldsymbol{\lambda}_0 \Leftarrow \mathbf{0}$
5: **while** $i < solverIterations$ **do**
6: **for all** constraints **do**
7: compute $\Delta\lambda$
8: compute $\Delta\mathbf{x}$
9: update $\lambda_{i+1} \Leftarrow \lambda_i + \Delta\lambda$
10: update $\mathbf{x}_{i+1} \Leftarrow \mathbf{x}_i + \Delta\mathbf{x}$
11: **end for**
12: $i \Leftarrow i + 1$
13: **end while**
14:
15: update positions $\mathbf{x}^{n+1} \Leftarrow \mathbf{x}_i$
16: update velocities $\mathbf{v}^{n+1} \Leftarrow \frac{1}{\Delta t}\left(\mathbf{x}^{n+1} - \mathbf{x}^n\right)$

NVIDIA

So here is the final XPBD algorithm. It is identical to the original PBD method with the addition of these two changes to the deltaLamba update, and then rather than throw away that value, we accumulate it in a per-constraint auxiliary variable.

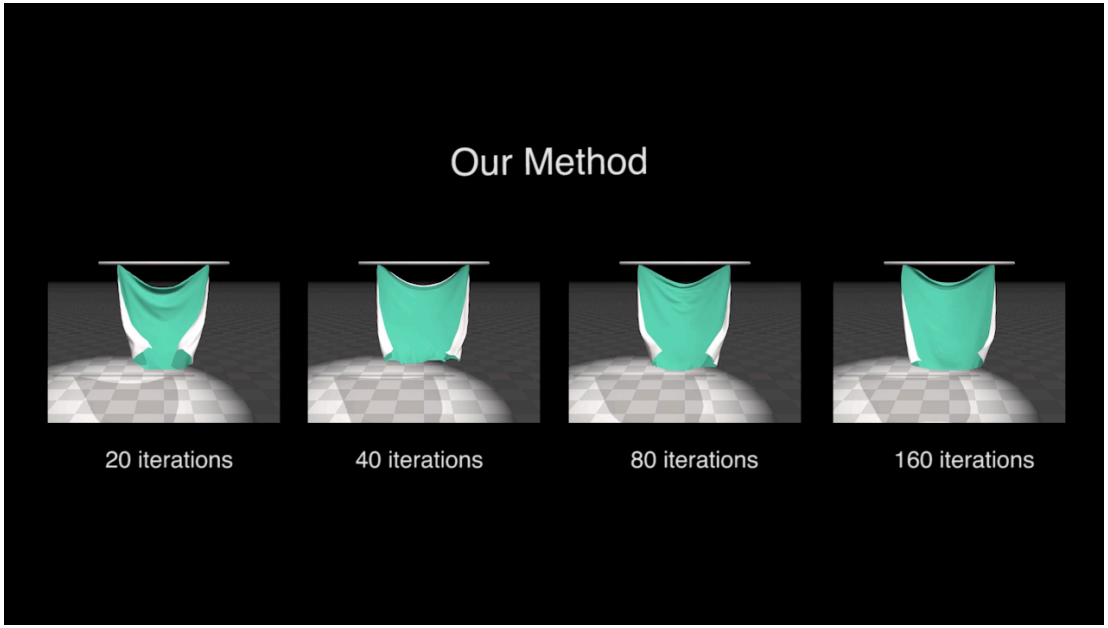
RESULTS

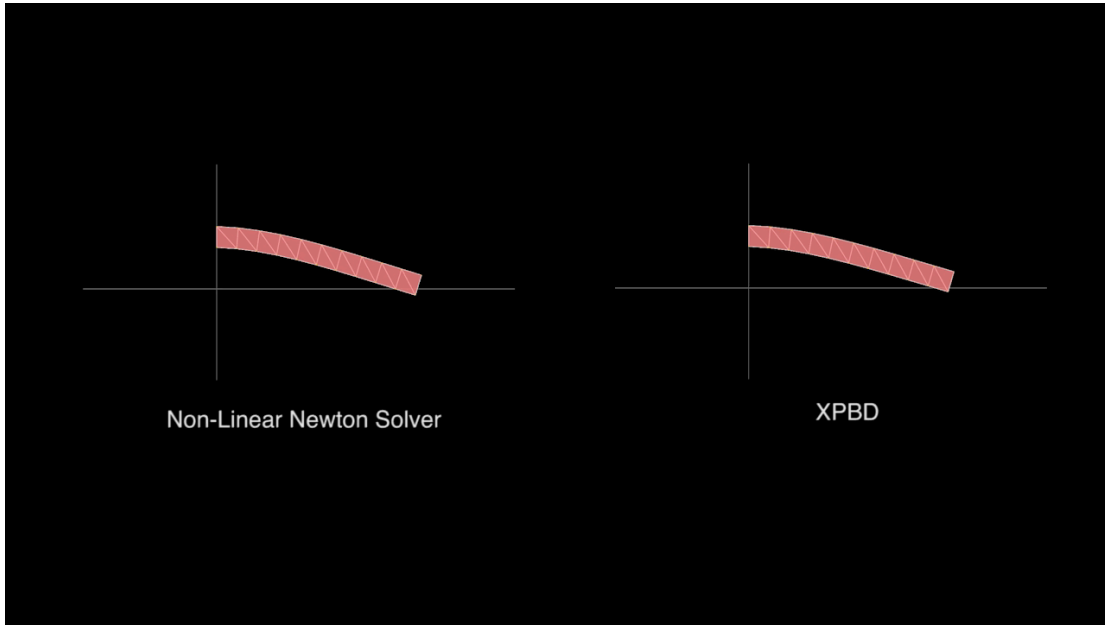
- Compare solver output to a non-linear Newton method
- Close agreement for primal and dual variables
- Extend PBD to handle non-linear FEM

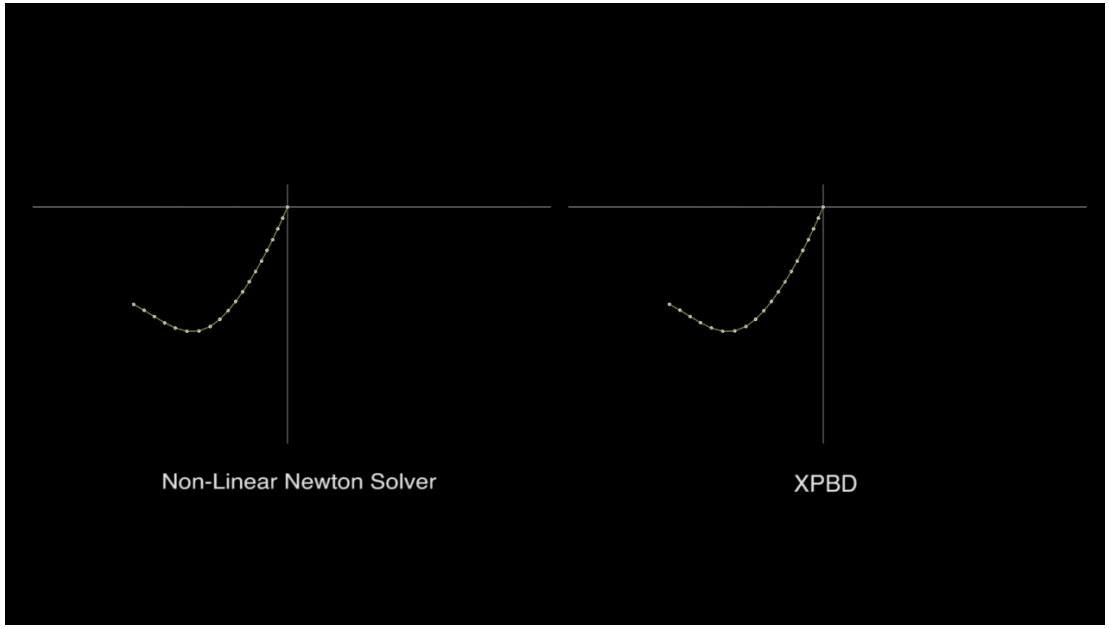


NVIDIA

We compare our method to a non-linear Newton method applied directly to the implicit equations of motion and find that it produces very close results. This graph shows the restoring force at the attachment for a chain swinging under gravity. XPBD approachs the reference solution given enough iterations. Here our Newton solver has used an LDLT decomposition as its linear solver backend.


Our Method
20 iterations
40 iterations
80 iterations
160 iterations


Non-Linear Newton Solver
XPBD


Non-Linear Newton Solver
XPBD

SUMMARY

- Extend PBD to simulate well defined elastic and dissipative potentials
- Maintains simplicity and robustness of original method
- Requires storing a a single scalar auxiliary variable representing a Lagrange multiplier
- Can be used to compute total constraint force magnitude for e.g.: motor limits, haptic devices, etc
- Reduces to PBD when alpha=0, too few iterations means artificial compliance
- Contact, assume zero compliance

THANK YOU

REFERENCES

- Servin, Martin, Claude Lacoursiere, and Niklas Melin. "Interactive simulation of elastic deformable materials." *SIGRAD 2006. The Annual SIGRAD Conference; Special Theme: Computer Games*. No. 019. Linköping University Electronic Press, 2006.
- Goldenthal, Rony, et al. "Efficient simulation of inextensible cloth." ACM Transactions on Graphics (TOG) 26.3 (2007): 49.
- Müller, Matthias, et al. "Position based dynamics." Journal of Visual Communication and Image Representation 18.2 (2007): 109-118.

DAMPING

- Extend the same method to handle implicit damping
- Define a Rayleigh dissipation potential D(x,v)
- Implicit damping just a few additional terms in Lagrange multiplier update

$$D(\mathbf{x},\mathbf{v}) = \frac{1}{2}\dot{\mathbf{C}}(\mathbf{x})^T\boldsymbol{\beta}\dot{\mathbf{C}}(\mathbf{x})$$
$$= \frac{1}{2}\mathbf{v}^T\nabla\mathbf{C}^T\boldsymbol{\beta}\nabla\mathbf{C}\mathbf{v},$$