THE MODIFIED PRECONDITIONED CONJUGATE GRADIENT METHOD FOR CLOTH SIMULATION

A PREPRINT

Guanxiong Chen
Department of Computer Science
University of British Columbia
Vancouver, BC V6T 1Z4
gxchen@cs.ubc.ca

December 22, 2021

ABSTRACT

Following Shewchuk's extraordinary introductory text, we explore the Preconditioned Conjugate Gradient (PCG) Method and apply its modified version (MPCG) to time stepping for simulating cloths. We implement the MPCG method, use it to simulate a 4×4-piece of cloth, and show our simulation results. The entire project's source code (including referenced code) is on `https://github.com/ericchen321/math607e_project`.

Keywords First keyword · Second keyword · More

1 Introduction

1.1 Significance and Scope

Simulation of soft bodies is an active research area in computer graphics. Specifically, soft bodies refer mostly to cloths and human tissues; simulating how they deform under various internal forces such as stretch, shear and bend force, and external forces such as gravity and air drag has a multitude of applications in the movie, game and medical industry. These simulations are also relevant to this course in that we get to apply topics we have covered in class: PDEs, direct methods, iterative methods, eigenvalue problems. For this project we narrow down our scope to cloth simulation, and explore how the CG method is applied to simulating this particular kind of material.

1.2 The problem of Cloth Simulation: Why Solve DEs?

Simulating cloths' deformation requires us to solve systems of ordinary differential equations (ODEs). Following the convention by Baraff and Witkin from Baraff and Witkin [1998], we model a piece of cloth as a set of particles connected by springs Chen [2021]. For simplicity, we consider a square-shaped piece with five particles on each side; all particles have the same mass m. We define the number of faces on each side of the cloth as N, and we say the square cloth is of shape $N \times N$. So in our case, $N = 4$, and in total we have $(N + 1)^2 = 25$ particles. Following conventions from physics, we name the position of all particles $\mathbf{x}$ as the configuration space of of the system. Since we intend to simulate the system in a 3D space, each particle has its position $\mathbf{x}_i \in \mathbb{R}^3$. So if we make $\mathbf{x}$ a flattened column vector, then $\mathbf{x}$ has dimension $(3(N + 1)^2, 1)$. So to simulate the cloth's deformation under forces, we need to formulate an ODE for $\mathbf{x}$ and evolve it over time. We can formulate the system of ODEs linearly:

$$\dot{\mathbf{x}} = f(\mathbf{x}) = B\mathbf{x} \tag{1}$$

for some function f and matrix B. Of course the actual system of equations has a more complex formulation, but we would rather leave the details in Section 2 while presenting only the gist of the problem here. Baraff and Witkin selected

the Backward Euler method for stability under large time steps:

$$\begin{aligned}\dot{\mathbf{x}^{k+1}} &= \mathbf{x}^k + hf(\mathbf{x}^{k+1}) \\ &= \mathbf{x}^k + hB\mathbf{x}^{k+1},\end{aligned} \tag{2}$$

where k is the time stepping index and h is the time step size (following convention from Baraff and Witkin [1998]). So we have a system of equations

$$(I - B)\mathbf{x}^{k+1} = A\mathbf{x}^{k+1} = \mathbf{x}^k, \tag{3}$$

and we solve for $\mathbf{x}^{k+1}$ at every time step. Obviously we now have the "solving $Ax = b$ " problem and we need to devise a smart way of solving it.

1.3 The Preconditioned CG (PCG) method: Why Do We Use It?

To solve a system of the form "Ax=b" we usually either use a direct method (e.g. Gaussian elimination, LU factorization, etc.) or an iterative method (e.g. steepest descent, CG, etc.) While direct methods work well with systems where A is small and dense, for system with large, sparse A iterative methods offer better speed and lower storage complexity Ascher and Greif [2011]. The intuition is that, for direct methods such as Gaussian elimination, in intermediate steps elements of A that were initially zeros would be replaced with non-zero values, leading to more costly computation and storage.

Since the configuration space we have is usually large (when N is large), our matrix A would be large. Also Baraff and Witkin identified A as a sparse matrix. Therefore we pick an iterative method of direct methods.

Compared with the method of steepest descent (SD), the CG method converges faster, since intuitively it gets to the minimum of one search direction at each step and never needs to search again in the same direction Shewchuk et al. [1994] (more details in Section 3). Also, the CG method requires A being a symmetric positive definite matrix; for the purpose of cloth simulation Baraff and Witkin indeed claimed that they could make A from Equation 3 symmetric and positive definite.

Ascher and Greif [2011] shows that the CG method takes $\mathcal{O}(\sqrt{\kappa(A)})$ iterations to converge to a solution of sufficiently small error, where $\kappa(A)$ is the condition number of A. So if the condition number of A is too large, the CG method would take way too many steps to converge.

Preconditioning reduces the condition number of A: we set up a preconditioner P which approximates A; then $P^{-1}A$ would be approximately equal to an identity matrix with condition number equal to 1. So we are solving the system

$$P^{-1}A\mathbf{x} = P^{-1}\mathbf{b} \tag{4}$$

for which CG should offer faster convergence. We discuss more details on preconditioning in Section 3.

1.4 Modified PCG (MPCG) to Incorporate Constraints on Particles

While PCG works for solving $Ax = b$ systems, for the particular problem of cloth simulation we need to modify it a bit to incorporate kinematic constraints imposed on particles. We will cover the modification in detail in Section 4, but here we introduce the gist of the idea. To evolve a particle's position over time, we take three steps:

1. First of all, we relate the particle's motion with the net force applied on it through Newton's Second Law of Motion: $\mathbf{a} = \frac{1}{m}\mathbf{f}$;
2. Then, we integrate the acceleration $\mathbf{a}$ twice to get displacement $\Delta\mathbf{x}$.
3. At last we advance the position: $\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta\mathbf{x}$.

Now we look at Step 1: the particle should have three degrees of freedom and its movement should be entirely dependent on whatever forces applied on it. But let us say if the particle is constrained to move along a plane, a path or not move at all due to some factors not accounted for by the simulated forces, e.g. being pinned to a wall, then the particle would have fewer degrees of freedom. We augment such constraints into our system by modifying $\frac{1}{m}$, so that in the simple case where we constrain the particle to have constant velocity we have acceleration becoming zero along the constrained directions. For particles constrained to a fixed position, we can simply make $\frac{1}{m} = 0$, but for particles constrained to one or two arbitrary orthogonal directions the modification becomes more complex. In Section 4 we discuss exactly how the modification works and how Baraff and Witkin introduced `filter()` operation to incorporate it to the PCG method.

1.5 Simulation Result

In Chen [2021] we have simulated a 4-by-4 piece of cloth for 50 steps using time step of 0.01 seconds. The cloth starts off as a square at rest, and we apply only a force along the $-z$ direction on the particle at the centre as the external force. The implementation however did not leverage the sparsity of matrices by using the CG method; rather it used `np.linalg.inv()` which solves through LU factorization Harris et al. [2020].

For this work, we implement the CG-based solver and showcase the simulation results using a 4×4-piece of cloth.

2 Cloth Simulation and Its Relation to PCG

In this section we introduce cloth simulation and where numerical differential equations get applied in greater detail.

2.1 Formulate the Problem

Following our discussion from Section 1.2, we now introduce the complete formulation of the system. From Newton's Second Law of Motion we know

$$\begin{aligned} \mathbf{a} = \frac{d\mathbf{v}}{dt} = M^{-1} f(\mathbf{x}, \mathbf{v}); \\ \frac{d\mathbf{x}}{dt} = \mathbf{v}, \end{aligned} \tag{5}$$

where $\mathbf{v}$ is the $(3(N+1)^2, 1)$ vector of velocities of all particles, M is the $(3(N+1)^2, 3(N+1)^2)$ mass matrix; for each particle i, we have its mass encapsulated in a $(3, 3)$ diagonal block with each diagonal element being the mass of the particle m_i Baraff and Witkin [1998]. Also unlike in Section 1, we use f to represent the net force (internal and external) on particles instead of some arbitrary functions.

We then proceed to discretize the system from Equation 5 with a finite difference scheme. As mentioned in Ascher and Boxerman [2003], Baraff and Witkin employed the semi-implicit backward Euler method for computing the finite difference. Following conventions from Baraff and Witkin [1998], we first write per-step change in $\mathbf{x}$ and $\mathbf{v}$ as $\Delta\mathbf{x}$, $\Delta\mathbf{v}$ respectively; then

$$\begin{aligned} \Delta\mathbf{x} = \mathbf{x}^{k+1} - \mathbf{x}^k; \\ \Delta\mathbf{v} = \mathbf{v}^{k+1} - \mathbf{v}^k. \end{aligned} \tag{6}$$

Note that $\mathbf{x}$ is the numerical solution as opposed to exact solution to the system in Equation 5. Then by the backward Euler method, the discretized system becomes

$$\begin{bmatrix} \Delta\mathbf{v} \\ \Delta\mathbf{x} \end{bmatrix} = h \begin{bmatrix} M^{-1} f(\mathbf{x}^{k+1}, \mathbf{v}^{k+1}) \\ \mathbf{v}^{k+1} \end{bmatrix} = h \begin{bmatrix} M^{-1} f(\mathbf{x}^k + \Delta\mathbf{x}, \mathbf{v}^k + \Delta\mathbf{v}) \\ \mathbf{v}^k + \Delta\mathbf{v} \end{bmatrix}. \tag{7}$$

However we see that System 7 becomes non-linear and is thus difficult to solve. So Baraff and Witkin Taylor-expanded f about $\mathbf{x}^k$ to linearize the system:

$$f(\mathbf{x}^k + \Delta\mathbf{x}, \mathbf{v}^k + \Delta\mathbf{v}) = f(\mathbf{x}^k, \mathbf{v}^k) + \Delta\mathbf{v}\frac{\partial f}{\partial \mathbf{v}}^k + \Delta\mathbf{x}\frac{\partial f}{\partial \mathbf{x}}^k + \mathcal{O}(\Delta\mathbf{v}^2) + \mathcal{O}(\Delta\mathbf{x}^2). \tag{8}$$

Ignoring the error terms, Equation 8 gives us a first-order Taylor approximation to f at $(\mathbf{x}^{k+1}, \mathbf{v}^{k+1})$. Also notice that since for the Jacobians we are using the derivatives at k instead of $k+1$, we say the method is semi-implicit as opposed to "fully" implicit. Plugging Equation 8 into System 7, we have

$$\begin{bmatrix} \Delta\mathbf{v} \\ \Delta\mathbf{x} \end{bmatrix} = h \begin{bmatrix} M^{-1}(f(\mathbf{x}^k, \mathbf{v}^k) + \Delta\mathbf{v}\frac{\partial f}{\partial \mathbf{v}}^k + \Delta\mathbf{x}\frac{\partial f}{\partial \mathbf{x}}^k) \\ \mathbf{v}^k + \Delta\mathbf{v} \end{bmatrix}. \tag{9}$$

We see that to solve System 9, we can substitute $\Delta\mathbf{x}$ by $h(\mathbf{v}^k + \Delta\mathbf{v})$ and plug it into the first equation, then move $\Delta\mathbf{v}$ to the left-hand side. Thus we get

$$\Big(I - hM^{-1}\frac{\partial f}{\partial \mathbf{v}}^k - h^2 M^{-1}\frac{\partial f}{\partial \mathbf{x}}^k\Big)\Delta\mathbf{v} = hM^{-1}\Big(f(\mathbf{x}^k, \mathbf{v}^k) + h\mathbf{v}^k\frac{\partial f}{\partial \mathbf{x}}^k\Big), \tag{10}$$

and at each step $k \geq 0$, we

1. solve for $\Delta\mathbf{v}$,
2. compute $\Delta\mathbf{x}$ (second equation from System 9),
3. compute $\mathbf{v}^{k+1}$ and $\mathbf{x}^{k+1}$.

at every iteration.

We also define the initial conditions $\mathbf{x}^{k=0}$ and $\mathbf{v}^{k=0}$. For this project, we set $\mathbf{x}^{k=0}$ as grid positions lying flat on the $x-y$ plane, and we set $\mathbf{v}^{k=0}$ as a zero vector.

2.2 Apply PCG

Why apply PCG? Apparently we want to apply a method, direct or indirect to solve for $\Delta\mathbf{v}$ in Equation 10. We prefer the PCG method as over a direct method such as LU factorization (which was adopted by `np.linalg.inv()` Harris et al. [2020]) because the system we are solving for

- contains many particles and is therefore large;
- is sparse. Using the cloth simulator from Chen [2021], we visualize our A matrix (defined precisely later in 12)'s sparcity in Figure 1.

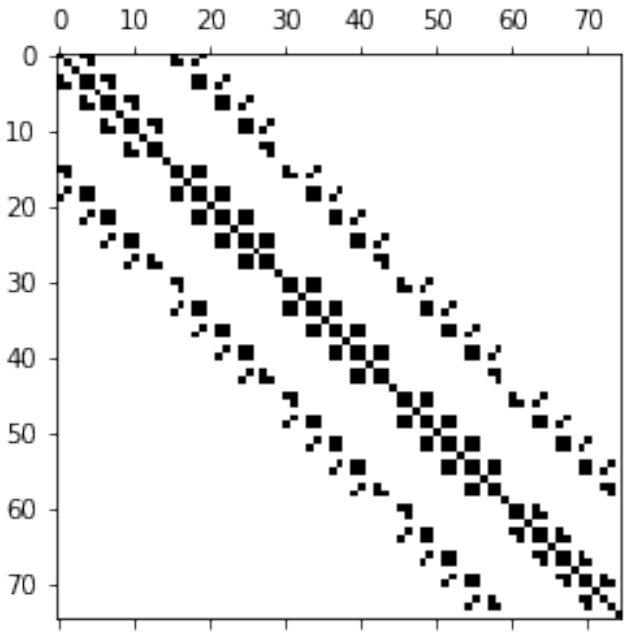

Figure 1: Sparcity of the A matrix for simulating a 4×4-piece of cloth.

How to apply PCG? Before we apply PCG to solve the system, we need to make sure the "A" matrix is symmetric positive definite - i.e. $\left(I - hM^{-1}\frac{\partial f}{\partial \mathbf{v}}^k - h^2 M^{-1}\frac{\partial f}{\partial \mathbf{x}}^k\right)$ is SPD. This condition unfortunately is not guaranteed. Baraff and Witkin claimed that by left-multiplying M we can make it SPD without proving the statement Baraff and Witkin [1998]. So Equation 10 becomes:

$$\left(M - h\frac{\partial f}{\partial \mathbf{v}}^k - h^2\frac{\partial f}{\partial \mathbf{x}}^k\right)\Delta\mathbf{v} = h\left(f(\mathbf{x}^k, \mathbf{v}^k) + h\mathbf{v}^k\frac{\partial f}{\partial \mathbf{x}}^k\right). \tag{11}$$

So now at each iteration we apply PCG on Equation 11 to solve the "$A\mathbf{x} = \mathbf{b}$" problem, having

$$\begin{aligned} A &= \Big(M - h\frac{\partial f}{\partial \mathbf{v}}^k - h^2\frac{\partial f}{\partial \mathbf{x}}^k\Big), \\ \mathbf{b} &= h\Big(f(\mathbf{x}^k, \mathbf{v}^k) + h\mathbf{v}^k\frac{\partial f}{\partial \mathbf{x}}^k\Big). \end{aligned} \tag{12}$$

3 The Conjugate Gradient (CG) Method and Preconditioning

In this section we discuss in details the conjugate gradient (CG) method and how preconditioning works to speed up convergence. Specifically, we elaborate the following:

1. Preconditions the system must satisfy for CG to apply;
2. Setting up a quadratic function and formulating the "solving $Ax = b$" problem as a minimization problem;
3. The method of steepest descent as a "simpler" precursor of the CG method;
4. The CG method and its pseudo-code implementation;
5. How preconditioning speeds up CG's convergence.

The order of the discussions is inspired by Shewchuk's work Shewchuk et al. [1994].

3.1 Preconditions

As discussed earlier in Section 1, the CG method solves a linear system $A\mathbf{x} = \mathbf{b}$ iteratively. A must be square, symmetric and positive-definite (SPD). By being positive-definite, we mean

$$\mathbf{x}^T A\mathbf{x} > 0 \tag{13}$$

for non-zero vector $\mathbf{x}$.

3.2 Solving a Minimization Problem

We pose the problem of solving $A\mathbf{x} = \mathbf{b}$ as a function minimization problem. First of all we define function

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{b}^T\mathbf{x} + c, \tag{14}$$

where c is some constant scalar. Without loss of generality we assume the dimension of the problem $n = 2$ — which is assuming $\mathbf{x}$, $\mathbf{b}$ as 2D column vectors, c as a scalar, and A as a $(2, 2)$ SPD matrix. Now we want to find the minimum of this function.

But how do we know there exists a minimum, not a maximum for $f(\mathbf{x})$? The complete proof would be time consuming to write, but we can answer this question intuitively by making an analogy in 1D: say $\mathbf{x}$ is a scalar, then A and c should be scalars as well. So the function becomes $f(x) = \frac{1}{2}Ax^2 - bx + c$. If A is positive, we immediately notice that the function is concave up, so there should be a global minimum. Now back in 2D: the SPD matrix A is analogous to a positive A in the 1D case, so we do have a global minimum. Guaranteeing a minimum exists is also why we need A to be positive-definite.

To find the minimum of $f(\mathbf{x})$, we take its derivative and solve for the $\mathbf{x}$ which gives zero derivative:

$$\nabla f(\mathbf{x}) = \frac{d}{d\mathbf{x}}\Big[f(\mathbf{x})\Big] = A\mathbf{x} - \mathbf{b} = \mathbf{0}. \tag{15}$$

We notice that Equation 15 is exactly the problem we had before: solving $\mathbf{x}$ for system $A\mathbf{x} = \mathbf{b}$. Thus instead of saying with the CG method we aim to solve $A\mathbf{x} = \mathbf{b}$, we can say we aim to minimize the objective function f as defined in Equation 14.

3.3 The Method of Steepest Descent (SD)

Before we introduce the CG method, we present the method of steepest descent (SD) which is very close to CG but easier to understand.

In a nut shell, to solve $A\mathbf{x} = \mathbf{b}$ with the SD method, we first make an "initial guess" $\mathbf{x}_0$, then at each step $i \geq 1$, we advance $\mathbf{x}_i$ until the error between $\mathbf{x}_i$ and the exact solution $\mathbf{x}$ is sufficiently small. We define two terms, *error* ($\mathbf{e}_i$) and *residual* ($\mathbf{r}_i$):

$$\mathbf{e}_i = \mathbf{x}_i - \mathbf{x}; \tag{16}$$

$$\mathbf{r}_i = \mathbf{b} - A\mathbf{x}_i = A(\mathbf{x} - \mathbf{x}_i) = -A\mathbf{e}_i. \tag{17}$$

Intuitively, $\mathbf{e}_i$ measures how close $\mathbf{x}_i$ is to the exact minimum $\mathbf{x}$; $\mathbf{r}_i$ measures the absolute error between $\mathbf{b}$ and the step-wise approximation $\mathbf{b}_i = A\mathbf{x}_i$. Suppose we have made our initial guess $\mathbf{x}_0$, and correspondingly we have our error $\mathbf{e}_0$ and residual $\mathbf{r}_0$. For the next step, as the method's name suggests, we want to march $\mathbf{x}_i$ in the direction where $f(\mathbf{x}_0)$ descents the most: the gradient $\nabla f(\mathbf{x}_0)$. From Equation 15 we see this is actually the negative of residual - $\nabla f(\mathbf{x}_0) = -\mathbf{r}_0$. So for $\mathbf{x}_1$ we have

$$\mathbf{x}_1 = \mathbf{x}_0 + \alpha_0 \mathbf{r}_0, \tag{18}$$

where α_0 is a scalar which says how much we march down that direction. We would like to pick a α_0 so that $\mathbf{x}_1$ gets to the minimum of f along the direction of $\mathbf{r}_0$. So we take the directional derivative of f at $\mathbf{x}_1$ along the direction of $\mathbf{r}_0$ and set it to 0:

$$\nabla_{\mathbf{r}_0} f(\mathbf{x}_1) = \nabla f(\mathbf{x}_1)^T \mathbf{r}_0 = 0. \tag{19}$$

And we have

$$\nabla f(\mathbf{x}_1) = -\mathbf{r}_1. \tag{20}$$

Now we see that $-\mathbf{r}_1 \mathbf{r}_0 = 0$, meaning $\mathbf{r}_1$ and $\mathbf{r}_0$ are orthogonal. We can then prove by induction that at each step, the direction of descent is orthogonal to the direction of descent of the previous step. From Equation 19 and 20 we can compute α_0:

$$\begin{aligned}
\mathbf{r}_1^T \mathbf{r}_0 &= 0 \\
(\mathbf{b} - A\mathbf{x}_1)^T \mathbf{r}_0 &= 0 \\
(\mathbf{b} - A(\mathbf{x}_0 + \alpha_0 \mathbf{r}_0))^T \mathbf{r}_0 &= 0 \\
\mathbf{b}^T \mathbf{r}_0 - (A\mathbf{x}_0)^T \mathbf{r}_0 - \alpha_0 (A\mathbf{r}_0)^T \mathbf{r}_0 &= 0 \\
-\alpha_0 (A\mathbf{r}_0)^T \mathbf{r}_0 &= (A\mathbf{x}_0)^T \mathbf{r}_0 - \mathbf{b}^T \mathbf{r}_0 \\
\alpha_0 &= \frac{(A\mathbf{x}_0)^T \mathbf{r}_0 - \mathbf{b}^T \mathbf{r}_0}{-(A\mathbf{r}_0)^T \mathbf{r}_0} \\
\alpha_0 &= \frac{\mathbf{b}^T \mathbf{r}_0 - \mathbf{r}_0^T \mathbf{r}_0 - \mathbf{b}^T \mathbf{r}_0}{-(A\mathbf{r}_0)^T \mathbf{r}_0} \\
\alpha_0 &= \frac{\mathbf{r}_0^T \mathbf{r}_0}{\mathbf{r}_0^T A \mathbf{r}_0}.
\end{aligned} \tag{21}$$

Knowing α_0 and $\mathbf{r}_0$, we can proceed to compute $\mathbf{x}_1$ and $\mathbf{r}_1$. We then repeat the above procedure for step $2, 3, \ldots$ until the error $\mathbf{e}_i$ is below a threshold.

How do we know if the method will converge to the true solution x? And if so, after how many steps would the method converge? First of all, as detailed in Shewchuk et al. [1994], let us reformulate the objective function $f(\mathbf{x})$ (Equation 14) as

$$f(\mathbf{x}) = f(\mathbf{x}^*) + \frac{1}{2}(\mathbf{x} - \mathbf{x}^*)^T A(\mathbf{x} - \mathbf{x}^*), \tag{22}$$

where $\mathbf{x}^*$ is the global minimum. So to minimize $f(\mathbf{x})$ we are in fact minimizing the term $\frac{1}{2}(\mathbf{x} - \mathbf{x}^*)^T A(\mathbf{x} - \mathbf{x}^*)$. Now let us multiply this term by 2, take the square root and refer to it as *energy norm* $||\mathbf{e}||_A$. So at each step i,

$$||\mathbf{e}_i||_A = \sqrt{(\mathbf{x}_i - \mathbf{x})^T A(\mathbf{x}_i - \mathbf{x})} = \sqrt{\mathbf{e}_i^T A \mathbf{e}_i}. \tag{23}$$

where $\mathbf{x}$ is the exact global minimum. Obviously, to minimize $f(\mathbf{x})$ we should aim to minimize the energy norm; for convergence, we need $\lim_{i \to \infty} ||\mathbf{e}_i||_A = 0$. Given our initial guess $\mathbf{x}_0$, we can compute the initial energy norm $||\mathbf{e}_0||_A$.

Then in Shewchuk et al. [1994] Shewchuck found we can define an upper bound on $||\mathbf{e}_{i+1}||_A$ with respect to $||\mathbf{e}_0||_A$ in the 2D case:

$$||\mathbf{e}_{i+1}||_A \leq \left(\frac{\kappa - 1}{\kappa + 1}\right)^i ||\mathbf{e}_0||_A, \tag{24}$$

where κ is the condition number of A. So we see $\frac{\kappa-1}{\kappa+1}$ is the growth factor of the error; and since $\kappa - 1 < \kappa + 1$, we know the growth factor would always be less than 1, so the SD method is guaranteed to converge. On the other hand, if κ is too large, $\frac{\kappa-1}{\kappa+1}$ would be close to 1, and convergence would be slow.

Finally we visualize the SD method in a simple 2D case, where

$$\begin{aligned} A &= \begin{bmatrix} 3 & 2 \\ 2 & 6 \end{bmatrix}, \\ b &= \begin{bmatrix} 2 \\ -8 \end{bmatrix}, \end{aligned} \tag{25}$$

where A is SPD, as given in Shewchuk et al. [1994]. We notice from Figure 2 (a) that

- at each step we take the direction of descent orthogonal to the level curve of f;
- at each step the direction of descent is orthogonal to the direction of descent of the previous step;
- the directions of descent form a "zigzag" shape: the same direction is taken at multiple (but non-consecutive) steps. Such behavior becomes more obvious as $\mathbf{x}_i$ approaches the exact minimum.

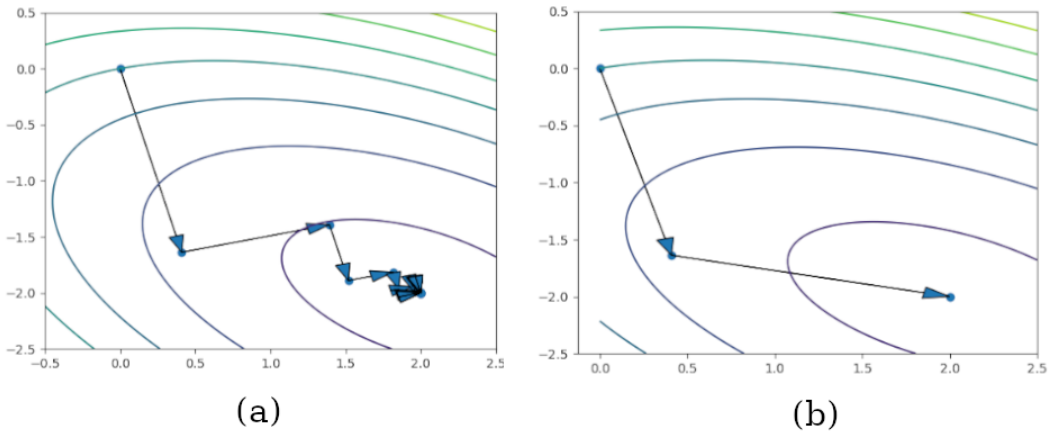

Figure 2: Visualization of (a) the SD method and (b) the CG method for a simple 2D case, starting from the same initial guess $\mathbf{x}_0$. The points are $\mathbf{x}_i$'s at each iteration; the arrows are directions of descent taken at each step.

3.4 The Method of Conjugate Gradient (CG)

While the SD method looks great and can always converge as long as A is SPD, the method has a flaw: it can converge very slowly if A's condition number is very large. If we visualize how $\mathbf{x}_i$ advances we will see the method steps along the same direction of descent over and over again.

A-Conjugacy. The CG method resolves this problem by making sure it steps along each descent direction only once. Here we use $\mathbf{d}_i$ to refer to the descent directions, as opposed to $\mathbf{r}_i$, since we are not going to use exactly the residuals. The directions are A-conjugate to each other:

$$\mathbf{d}_i^T A \mathbf{d}_j = 0 \tag{26}$$

for $i \neq j$.

The Method of Conjugate Direction (CD). We call the method above (in which we step along A-conjugate directions) the CD Method. In the SD method, at each step we have

$$\alpha_i = \frac{\mathbf{r}_i^T \mathbf{r}_i}{\mathbf{r}_i^T A \mathbf{r}_i}. \tag{27}$$

We derived Equation 27 under the assumption that at each step the residual is orthogonal to the residual of the preceding step. This however is not the case for the CD method: here we want to have $\mathbf{e}_{i+1}$ A-orthogonal to $\mathbf{d}_i$, so from step $i+1$ onward we would never have to descend along direction $\mathbf{d}_i$. From here we can derive α_i:

$$\begin{aligned} \mathbf{d}_i^T A \mathbf{e}_{i+1} &= 0 \\ \mathbf{d}_i^T (\mathbf{e}_i + \alpha_i \mathbf{d}_i) &= 0 \\ \alpha_i &= -\frac{\mathbf{d}_i^T A \mathbf{e}_i}{\mathbf{d}_i^T A \mathbf{d}_i} \\ \alpha_i &= \frac{\mathbf{d}_i^T A \mathbf{r}_i}{\mathbf{d}_i^T A \mathbf{d}_i}. \end{aligned} \tag{28}$$

Now we want to ask:

1. Does the CD method converge? How fast would it converge?
2. How do we construct each direction of descent $\mathbf{d}_i$?

To answer the first question, as suggested in Shewchuk et al. [1994], we express error $\mathbf{e}_0$ as a linear combination of descent directions:

$$\mathbf{e}_0 = \sum_{j=0}^{n-1} \delta_j \mathbf{d}_j. \tag{29}$$

Left-multiply both sides of Equation 29 by $\mathbf{d}_i^T$, we have

$$\delta_i = \frac{\mathbf{d}_i^T A \mathbf{e}_i}{\mathbf{d}_i^T A \mathbf{d}_i}. \tag{30}$$

From Equation 30 and Equation 28 we notice that $\alpha_i = -\delta_i$. Therefore for error $\mathbf{e}_i$, we have

$$\begin{aligned} \mathbf{e}_i &= \mathbf{e}_0 + \sum_{j=0}^{n-1} \alpha_j \mathbf{d}_j \\ &= \mathbf{e}_0 - \sum_{j=0}^{i-1} \delta_j \mathbf{d}_j \\ &= \sum_{j=0}^{n-1} \delta_j \mathbf{d}_j - \sum_{j=0}^{i-1} \delta_j \mathbf{d}_j \\ &= \sum_{j=i}^{n-1} \delta_j \mathbf{d}_j. \end{aligned} \tag{31}$$

. So we see at step $i = n$, we are guaranteed to have $\mathbf{e}_i = 0$. So the CD method converges to the exact solution in at most n steps! Now we proceed to think about constructing the descent vectors $\mathbf{d}_i$'s. Suppose we have a set of n linearly independent vectors $\{\mathbf{u}_0, \mathbf{u}_1, \dots \mathbf{u}_{n-1}\}$. We can pick the axis vectors from an Euclidean coordinate system for example. For each descent vector $\mathbf{d}_i$, we build it from $\mathbf{u}_i$ and make sure it is A-conjugate with all previous $\mathbf{d}_i$'s by subtracting parallel components:

$$\mathbf{d}_i = \mathbf{u}_i + \sum_{j=0}^{i-1} \beta_{ij} \mathbf{d}_j, \tag{32}$$

where for $i = 0$ we have $\mathbf{d}_0 = \mathbf{u}_0$. The process of determining each β_{ij}, hence $\mathbf{d}_i$'s is called the Gram-Schmidt Process and is discussed with great visualizations in Shewchuk et al. [1994]. Here we only show how β_{ij} from 32 is derived -

we right-multiply both sides of the equation by $A\mathbf{d}_j$:

$$\begin{aligned} \mathbf{d}_i A\mathbf{d}_j &= \mathbf{u}_i A\mathbf{d}_j + \sum_{k=0}^{i-1} \beta_{ik}\mathbf{d}_k A\mathbf{d}_j \\ 0 &= \mathbf{u}_i A\mathbf{d}_j + \beta_{ij}\mathbf{d}_j A\mathbf{d}_j \\ \beta_{ij} &= \frac{-\mathbf{u}_i A\mathbf{d}_j}{\mathbf{d}_j A\mathbf{d}_j}. \end{aligned} \tag{33}$$

note that here we use k as the index variable to iterate over previous descent vectors. Now with β_{ij} in our possession, we can compute $\mathbf{d}_i$ with Equation 32. Then using Equation 28 we can compute α_i, and then advance $\mathbf{x}_i$ toward the exact solution.

The CG Method. From Equation 33 we notice two problems with the CD method:

1. At each step i, we need to compute β_{ij} for all $0 < j < i$. So the computational overhead is high;
2. We need to store all search vectors $\mathbf{d}_j$ for $0 < j < i$ to compute β_{ij}. While needing to store them is not a big deal if n is low, for high-dimensional problems such as simulating a cloth piece of thousands of particles the storage cost would be huge.

We resolve the two problems by making a simple adjustment to the CD method : set $\mathbf{u}_i = \mathbf{r}_i$ and call it the method of Conjugate Gradient (CG). This adjustment makes $\beta_{ij} = 0$ for all $j < i - 1$ (proof given in Shewchuk et al. [1994]). So we only have non-zero β_{ij} for $j = i - 1$, and as derived in Shewchuk et al. [1994] we have

$$\beta_i = \beta_{i,i-1} = \frac{\mathbf{r}_i^T \mathbf{r}_i}{\mathbf{r}_{i-1}^T \mathbf{r}_{i-1}}. \tag{34}$$

Since β_i now depends only on $\mathbf{r}_i$ and $\mathbf{r}_{i-1}$, we have significantly reduced computational cost. Also apparently now we do not have to store $\mathbf{d}_0, \mathbf{d}_1, \ldots \mathbf{d}_{i-1}$ to compute β_i, so we have less storage cost.

CG: a 2D example. So now say we pick some 2D vector $\mathbf{x}_0$ as our initial guess for the exact solution $\mathbf{x}$. And let us assume that we know beforehand the method takes two steps to converge. We follow the steps below to advance from $\mathbf{x}_0$ to $\mathbf{x}_1$:

$$\begin{aligned} \mathbf{r}_0 &= \mathbf{b} - A\mathbf{x}_0; \\ \mathbf{d}_0 &= \mathbf{r}_0; \\ \alpha_0 &= \frac{\mathbf{r}_0^T \mathbf{r}_0}{\mathbf{d}_0^T A\mathbf{d}_0}; \\ \mathbf{x}_1 &= \mathbf{x}_0 + \alpha_0 \mathbf{d}_0. \end{aligned} \tag{35}$$

Then we proceed to compute $\mathbf{x}_2$:

$$\begin{aligned} \mathbf{r}_1 &= \mathbf{b} - A\mathbf{x}_1; \\ \beta_1 &= \frac{\mathbf{r}_1^T \mathbf{r}_1}{\mathbf{r}_0^T \mathbf{r}_0}; \\ \mathbf{d}_1 &= \mathbf{d}_0 + \mathbf{r}_1 + \beta_1 \mathbf{d}_0; \\ \alpha_1 &= \frac{\mathbf{r}_1^T \mathbf{r}_1}{\mathbf{d}_1^T A\mathbf{d}_1}; \\ \mathbf{x}_2 &= \mathbf{x}_1 + \alpha_1 \mathbf{d}_1. \end{aligned} \tag{36}$$

And we know for sure $\mathbf{x}_2 = \mathbf{x}$, since from Equation 31 we know the CD/CG method converges in at most n steps. For problems in higher dimensions, we repeat the steps from 36 for $\mathbf{x}_3$, $\mathbf{x}_4$, etc until the residual is sufficiently small or until we have repeated by n steps.

Once again we visualize the method with the help of a 2D case. We see in Figure 2 (b) that the method takes only two steps to get to the exact minimum; compared with the SD method visualized in (a), the CG method takes a lot less steps, since it never wastes effort in traversing along a direction which it has descended along before.

3.5 Preconditioning

Here we discuss

1. The problem with CG's rate of convergence;
2. How we construct a preconditioner to mitigate the problem, and how we integrate preconditioning to the CG method.

The problem with CG's rate of convergence. In Section 3.3 we presented how the rate of convergence of the SD method is affected by A's condition number: larger condition numbers lead to slower convergence. The CG method unfortunately suffers from the same problem: as proven in Shewchuk et al. [1994], we can bound the energy norm of error at step i by

$$||\mathbf{e}_i||_A \leq 2\Big(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\Big)^i ||\mathbf{e}_0||_A; \tag{37}$$

and we see that with large condition number κ the growth factor approaches to 1, leading to slow convergence.

We can lower the condition number of A by defining a SPD matrix P that approximates to A and is easily invertible, then multiplying $A\mathbf{x} = \mathbf{b}$ by P^{-1} on both sides:

$$P^{-1}A\mathbf{x} = P^{-1}\mathbf{b}. \tag{38}$$

Often we set up M as a diagonal matrix for which the diagonal entries are the same as those of A Shewchuk et al. [1994]. Since P^{-1} would be approximately A^{-1}, $P^{-1}A$ (which we can intuitively think of as the new "A" matrix) would have a condition number approximately equal to one.

Construct a preconditioner P and put it in the CG method. Recall that for CG to work, we need $P^{-1}A$ to be SPD — but can we get such guarantee? The answer is we cannot; but still we can apply the CG method somehow. In Shewchuk et al. [1994] Shewchuck proved via Cholesky factorization that CG would still work on such matrices. The entire proof is fairly long and is not the focus of this work, so here we just present the gist of it: we factorize P by $P = EE^T$, and we use CG to solve the system $E^{-1}AE^{-T}\hat{\mathbf{x}} = E^{-1}\mathbf{b}$ for $\hat{\mathbf{x}}$, since $E^{-1}AE^{-T}$ is SPD. At last we solve for $\mathbf{x}$ from $\hat{\mathbf{x}}$ by $\hat{\mathbf{x}} = E^T\mathbf{x}$. So in fact, as Ascher and Greif formulated in Ascher and Greif [2011], the system now we should be solving for is

$$\Big(P^{-\frac{1}{2}}AP^{-\frac{1}{2}}\Big)\Big(P^{\frac{1}{2}}\mathbf{x}\Big) = P^{-\frac{1}{2}}\mathbf{b}, \tag{39}$$

. Or, as in Shewchuk et al. [1994] we can eliminate E entirely and solve for $\mathbf{x}$ at each step:

$$\begin{aligned} \mathbf{r}_0 &= \mathbf{b} - A\mathbf{x}_0; \\ \mathbf{d}_0 &= P^{-1}\mathbf{r}_0 \end{aligned} \tag{40}$$

for Step 0. Then for Step $i \geq 0$, we compute

$$\begin{aligned} \alpha_i &= \frac{\mathbf{r}_i^T P^{-1}\mathbf{r}_i}{\mathbf{d}_i^T A\mathbf{d}_i}, \\ \mathbf{x}_{i+1} &= \mathbf{x}_i + \alpha_i\mathbf{d}_i, \\ \mathbf{r}_{i+1} &= \mathbf{r}_i - \alpha_i A\mathbf{d}_i, \\ \beta_{i+1} &= \frac{\mathbf{r}_{i+1}^T P^{-1}\mathbf{r}_{i+1}}{\mathbf{r}_i^T P^{-1}\mathbf{r}_i}, \\ \mathbf{d}_{i+1} &= P^{-1}\mathbf{r}_{i+1} + \beta_{i+1}\mathbf{d}_i \end{aligned} \tag{41}$$

until the error in either $\mathbf{e}_i$ or $\mathbf{r}_i$ becomes sufficiently small.

4 The Modified Preconditioned Conjugate Gradient (MPCG) Method for Cloth Simulation

In Section 1.4 we briefly explained that we need to modify the PCG method for cloth simulation somehow to accommodate kinematic constraints on particles. Here we discuss MPCG in greater detail.

Recall from Equation 11 that at each step we solve for the change in velocities $\Delta\mathbf{v}$ of particles. If a particle i moves freely within a 3D space then we can compute $\Delta\mathbf{v}_i$ exactly from the equation. But if the particle is constrained along, say, the x direction then $\Delta\mathbf{v}_i$ becomes

$$\begin{bmatrix}\Delta v_{ix}\\ \Delta v_{iy}\\ \Delta v_{iz}\end{bmatrix} = \begin{bmatrix}z_{ix}\\ \Delta v_{iy}\\ \Delta v_{iz}\end{bmatrix}, \tag{42}$$

where z_{ix} is a preset change in velocity imposed on the particle by the constraint Baraff and Witkin [1998]. Moreover, constraints are not necessarily applied on axis-aligned directions; we can easily apply constraints along orthogonal directions defined by unit vector $\mathbf{p}$ (two DoF) or $\mathbf{p}$ and $\mathbf{q}$ (one DoF) by setting up filter matrix S_i on each particle:

$$S_i = \begin{cases} I & \text{DoF = 3}\\ I - \mathbf{p}_i\mathbf{p}_i^T & \text{DoF = 2}\\ I - \mathbf{p}_i\mathbf{p}_i^T - \mathbf{q}_i\mathbf{q}_i^T & \text{DoF = 1}\\ 0 & \text{DoF = 0}\end{cases} \tag{43}$$

Baraff and Witkin [1998] Then we multiply S_i with $\Delta\mathbf{v}_i$ to "filter out" changes induced by simulated forces along the constrained directions, so that

$$\Delta\mathbf{v}_{i,\text{modified}} = S_i\Delta\mathbf{v}_i + \mathbf{z}_i \tag{44}$$

If we are using a direct method to solve the system, we would need to augment our constraint matrices S_i's into the inverse mass matrix M^{-1} from Equation 10 to nullify velocity changes when force or force derivatives are not zeroes. We augment by defining matrix W and placing S_i along its diagonals Baraff and Witkin [1998]. Then we make these two changes to Equation 10:

1. Replace M^{-1} by W;
2. Add $\mathbf{z}$ to the RHS of the equation. We see that for a fully-constrained particle, $W = 0$, and we have $\Delta\mathbf{v} = \mathbf{z}$.

Following the two changes above, we have

$$\Big(I - hW\frac{\partial f}{\partial \mathbf{v}}^k - h^2W\frac{\partial f}{\partial \mathbf{x}}^k\Big)\Delta\mathbf{v} = hW\Big(f(\mathbf{x}^k,\mathbf{v}^k) + h\mathbf{v}^k\frac{\partial f}{\partial \mathbf{x}}^k\Big) + \mathbf{z} \tag{45}$$

Baraff and Witkin [1998].

The CG method on the other hand requires the "A" matrix from Equation 45 to be SPD. This unfortunately is not the case Baraff and Witkin [1998]. Also the CG method takes multiple intermediate steps with lots of variables to take care of - residuals, directions of descent, etc, and for each variable we need to apply the constraint for the method to hold. So Baraff and Witkin proposed the Modified Preconditioned Conjugate Gradient (MPCG) Method defined as Algorithm 1 Baraff and Witkin [1998].

We can see that function `FILTER()` takes in a vector and nullifies components of input $\mathbf{u}$ along constrained directions (specified in S_i) to zeroes. Function `MPCG()` takes A, $\mathbf{b}$, $\mathbf{z}$ and computes the change-in-velocity $\Delta\mathbf{v}$. Note that $\mathbf{c}$ is the preconditioned and filtered direction of descent; δ_0 measures the size of $\mathbf{b}$ (and we multiply by P^{-1} here since as shown in Equation 38, we are applying preconditioning to both sides of the equation we are solving for). On each iteration, we compute $\delta = \mathbf{r}^TP^{-1}\mathbf{r}$ which measures how close we are in terms of estimating $\mathbf{b}$, and we terminate the algorithm once we have $\delta \le \epsilon^2\delta_0$, where ϵ is the machine episilon (largest relative error). So essentially we terminate as soon as the magnitude of $\mathbf{r}$ becomes less than the tolerable largest absolute error in $\mathbf{b}$.

5 Simulation Result

5.1 Implementation

We have implemented an MPCG solver as a Python class; the solver follows Algorithm 1. Please refer to Appendix A for the complete source code of the MPCG solver.

To simulate cloth, we simply used Chen's cloth simulator implementation from Chen [2021]. Specifically, the cloth is initialized as a 4×4 (4 faces along each side) square-shaped grid of particles on the $x - y$ plane; in total the cloth has 25 particles. Each square has a side width of 1.5. We fix the four corners of the cloth at their initial positions, and apply a constant force with magnitude of 2 along the $-z$ to the particle at the centre. We simulate for a total

Algorithm 1 The MPCG Algorithm.

procedure FILTER($\mathbf{u}$) ▷ $\mathbf{u}$ is a vector (not necessarily change in velocity)
  for i from 1 to n **do** ▷ n being the number of particles
    $\mathbf{u}_{i,\texttt{filtered}} \leftarrow S_i\ \mathbf{u}_i$
  end for
  return $\mathbf{u}_{\texttt{filtered}}$
end procedure
procedure MPCG(A, $\mathbf{b}$, $\mathbf{z}$)
  $\Delta\mathbf{v} \leftarrow \mathbf{z}$
  $\delta_0 \leftarrow \texttt{FILTER}(\mathbf{b})^T P \texttt{FILTER}(\mathbf{b})$
  $\mathbf{r} \leftarrow \texttt{FILTER}(\mathbf{b} - A\Delta\mathbf{v})$
  $\mathbf{c} \leftarrow \texttt{FILTER}(P^{-1}\mathbf{r})$
  $\delta_{\texttt{new}} \leftarrow \mathbf{r}^T\mathbf{c}$
  while $\delta_{\texttt{new}} > \epsilon^2\delta_0$ **do**
    $\mathbf{q} \leftarrow \texttt{FILTER}(A\mathbf{c})$
    $\alpha \leftarrow \frac{\delta_{\texttt{new}}}{\mathbf{c}^T\mathbf{q}}$
    $\Delta\mathbf{v} \leftarrow \Delta\mathbf{v} + \alpha\mathbf{c}$
    $\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{q}$
    $\mathbf{s} \leftarrow P^{-1}\mathbf{r}$
    $\delta_{\texttt{old}} \leftarrow \delta_{\texttt{new}}$
    $\delta_{\texttt{new}} \leftarrow \mathbf{r}^T\mathbf{s}$
    $\mathbf{c} \leftarrow \texttt{FILTER}(\mathbf{s} + \frac{\delta_{\texttt{new}}}{\delta_{\texttt{old}}}\mathbf{c})$
  end while
  return $\Delta\mathbf{v}$
end procedure

of 0.4 seconds with a time step of 0.02 seconds over 20 steps. For the simulator's implementation, please check `https://github.com/ericchen321/math607e_project`.

Inside the loop where we solved for velocity change and update particle positions, we replaced function call to `np.linalg.inv()` by the MPCG solver, and made changes to the "A" matrix and "$\mathbf{b}$" accordingly.

5.2 Correctness

We confirmed the solver's correctness by testing it against cases in which a particle is unconstrained or constrained along one, two or three directions.

For the unconstrained case: we solved a system by using `np.linalg.inv()` and compared the result with the solution from the solver.

For the constrained cases: we found it quite difficult to construct test cases, since obtaining force matrices and force derivatives were difficult without the cloth simulator. And even if we did obtain those data from the simulator, we could not confirm if the data is correct (since the simulator itself was not properly tested), not to say our implementation of the MPCG solver. So in our unit tests, we checked the correctness in constrained cases by only inspecting if changes in velocity along the constrained directions remain zero. We did however compute the discrepency between $\Delta\mathbf{v}$ solved by `np.linalg.inv()` from Equation 45 and $\Delta\mathbf{v}$ solved by our MPCG solver from Equation 11 at each step. Unfortunately we see that the errors are not close to zero, suggesting bugs either in our MPCG solver or in the cloth simulator.

5.3 Visualization

In Figure 3 we show the cloth mesh at different time steps. We notice that while after 10 steps the cloth's deformation still appears natural, after 20 steps the deformation is completely messed up, suggesting either the simulator is faulty or the MPCG solver is bugged.

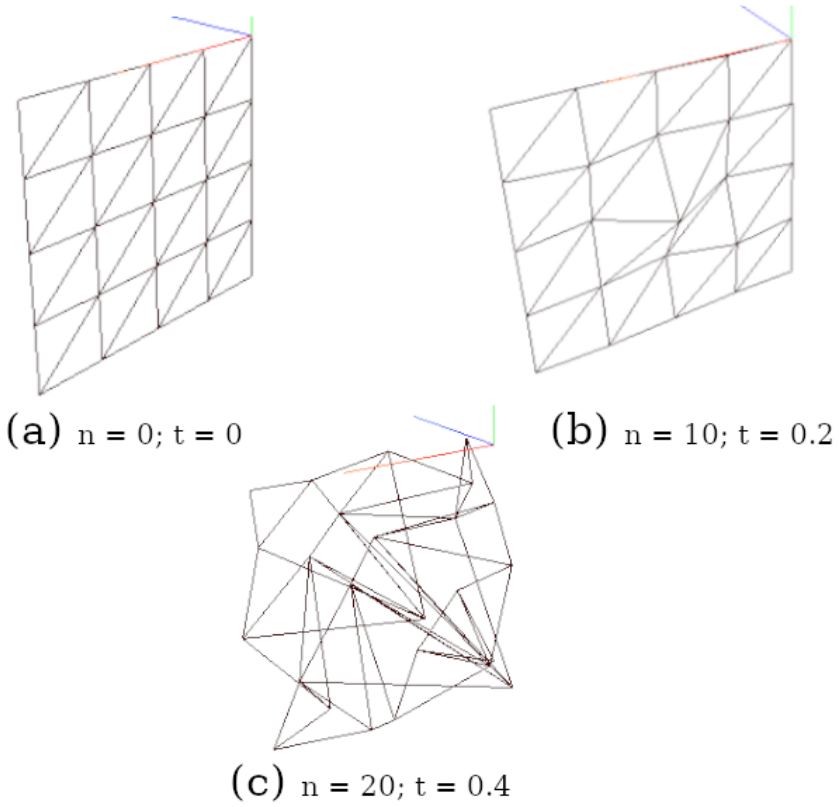

Figure 3: Mesh deformation at (a) $t = 0s$, (b) $t = 0.2s$, and (c) $t = 0.4s$.

6 Conclusion and Future Work

6.1 Conclusion

Following Shewchuck's excellent introductory text, we have explained in details the conjugate gradient (CG) method, preconditioning and how they get to be applied in cloth simulation. Further we elaborated on Baraff and Witkin's Modified PCG (MPCG) method. We also discussed our implementation of the MPCG method and showcased our simulation result on a 4×4-piece of cloth.

6.2 Future Work

We have identified three aspects of this work that should be improved upon:

1. Testing of the simulator and the MPCG solver. From the simulation result we see apparently our method is unstable despite being mathematically sound. So our implementation of either the simulator or the MPCG solver must be bugged. So we should test them further.
2. Performance comparison. We would like to investigate how much speed up we can get by replacing `np.linalg.inv()` (the LU factorization solver) with the MPCG method.
3. Implement Ascher and Boxerman's improved MPCG method Ascher and Boxerman [2003].

A Source Code: MPCG Solver

```
import numpy as np

class MPCGSolver:
    r"""
```

```
Solve system Ax = b with the MPCG method.
"""

def __init__(self, A_in, b_in, S_in, z_in) -> None:
    r"""
    Constructor for MPCGSolver

    :param A_in: A matrix of shape (3n, 3n)
    :param b_in: vector b of shape (3n, )
    :param S_in: constraint list of length n; each element
        should be a tuple of 0/1/2/3 (not-necessarily unitary)
        vectors indicating prohibited directions
    :param z_in: constrained velocity matrix of shape (n, 3);
        zero vectors for unconstrained particles
    """
    self._A = A_in.copy()
    self._b = np.expand_dims(b_in, axis=1)
    self._z = z_in.copy()
    self._num_particles = self._z.shape[0]
    self._S = self.compute_S(S_in)
    self.compute_M()
    self._epsi = 1e-8

@property
def num_particles(self):
    r"""
    Getter for num_particles
    """
    return self._num_particles

@property
def A(self):
    r"""
    Getter for A
    """
    return self._A

@property
def b(self):
    r"""
    Getter for b
    """
    return self._b

@property
def P(self):
    r"""
    Get the preconditioner matrix M.
    """
    return self._M

def compute_M(self):
    r"""
    Compute the preconditioner P from A.
    """
    assert self._A is not None
    self._M = np.diag(np.diag(self._A))

@property
def z(self):
    r"""
    Get the constraint matrix z.
    """
    return self._z
```

```
def compute_S(self, S_in):
    r"""
    Compute constraint matrix S for each particle
    from list of constrained direction tuples.

    :param S_in: list of n tuples of prohibited directions

    :return:
        list of n constraint matrices S_i of shape (3, 3)
    """
    S_out = []
    for particle_index in range(self._num_particles):
        # get the constraint vectors and compute S_i
        S_in_i = S_in[particle_index]
        if len(S_in_i) == 0:
            # no constraints
            S_i = np.eye(3)
        elif len(S_in_i) == 1:
            # one constraint
            p = np.expand_dims(S_in_i[0], axis=0)
            S_i = np.eye(3) - np.matmul(np.transpose(p), p)
        elif len(S_in_i) == 2:
            # two constraints
            p = np.expand_dims(S_in_i[0], axis=0)
            q = np.expand_dims(S_in_i[1], axis=0)
            S_i = np.eye(3) - np.matmul(np.transpose(p), p) - np.matmul(np.transpose(q), q)
        else:
            # three constraints
            S_i = np.zeros((3, 3))
        # add S_i to S
        S_out.append(S_i)
    return S_out

def filter(self, v):
    r"""
    Filter vector v by kinematic constraints.

    :param v: vector of shape (3n, 1)

    :return:
        v filtered by constraints; shape is (3n, 1)
    """
    v_out = np.zeros(v.shape)
    for particle_index in range(self.num_particles):
        # extract v_i
        v_i = v[particle_index*3:(particle_index+1)*3, 0]
        # compute S_i * v_i
        S_i_v_i = np.matmul(self._S[particle_index], v_i)
        # assign S_i*v_i to v_out
        v_out[particle_index*3:(particle_index+1)*3, 0] = S_i_v_i
    return v_out

def solve(self):
    r"""
    Solve A * del_v = b. Return del_v.
    """
    # initialize del_v
    del_v = np.reshape(
        self._z.copy(),
        (self._z.shape[0]*self._z.shape[1], 1))

    delta_0 = np.matmul(
        np.transpose(self.filter(self._b)),
        np.matmul(np.linalg.inv(self._M), self.filter(self._b)))
    r = self.filter(self._b - np.matmul(self._A, del_v)) # (3n, 1)
```

```
c = self.filter(np.matmul(np.linalg.inv(self._M), r)) # (3n, 1)
delta_new = np.matmul(np.transpose(r), c)

while delta_new > np.power(self._epsi, 2)*delta_0:
    # iterate until relative error in ||r||^2 is small enough
    q = self.filter(np.matmul(self._A, c))
    alpha = delta_new / np.matmul(np.transpose(c), q)
    del_v = del_v + alpha*c
    r = r - alpha*q
    s = np.matmul(np.linalg.inv(self._M), r) # (3n, 1)
    delta_old = delta_new
    delta_new = np.matmul(np.transpose(r), s)
    c = self.filter(s + (delta_new/delta_old)*c)

return del_v
```

References

David Baraff and Andrew Witkin. Large steps in cloth simulation. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 43–54, 1998.

Guanxiong Chen. Cpsc 548 assignment: Cloth simulation. https://github.com/ericchen321/cpsc548_assignments/tree/master/a2, 2021.

Uri M Ascher and Chen Greif. *A first course on numerical methods*. SIAM, 2011.

Jonathan Richard Shewchuk et al. An introduction to the conjugate gradient method without the agonizing pain, 1994.

Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi:10.1038/s41586-020-2649-2. URL https://doi.org/10.1038/s41586-020-2649-2.

Uri M Ascher and Eddy Boxerman. On the modified conjugate gradient method in cloth simulation. *The Visual Computer*, 19(7):526–531, 2003.