Implementing XPBD

JEAN-LOUP ARETTE-HOURQUET, Telecom Paris

Extended Position-Based Dynamics (XPBD) is a widely used method for simulating soft and deformable bodies in real-time. This report begins by summarizing the core principles of XPBD and its key improvements over traditional Position-Based Dynamics (PBD), particularly in handling constraints with stiffness and time integration. I describe the implementation of XPBD for specific physical bodies and constraints, along with an analysis comparing multiple solver iterations and sub-stepping for accuracy and stability.

Additional Key Words and Phrases: Extended Position-Based Dynamics, Position-Based Dynamics, real-time simulation, constraints, deformable bodies, physics-based animation

1 INTRODUCTION

Position-Based Dynamics (PBD) is a technique in real-time simulation, widely used in computer graphics for its simplicity, robustness, and efficiency. Introduced by [Müller et al. 2007], PBD focuses on directly manipulating the positions of particles to satisfy constraints, bypassing the need to solve complex systems of differential equations. This approach makes it particularly appealing for applications like cloth simulation, soft bodies, and fluid dynamics, where real-time performance is crucial. However, while PBD excels in speed and stability, it lacks physical accuracy and the stiffness of the constraints varies with the number of iterations. The last point can be an issue in video games for example, where the frames per second varies, so can create unconsistent results.

Extended Position-Based Dynamics (XPBD), introduced by [Macklin et al. 2016], addresses these limitations by modifying the constraints definition and the solver loop. This enhancement also allows infinitely stiff constraints alongside soft constraints while keeping a fast and stable simulation. Then, the method has been used in more various situations such as rigid bodies or fluid simulations.

This report explores the implementation of XPBD, detailing the constraints and physical bodies modeled.

2 THEORITICAL OVERVIEW

2.1 Description of the algorithm

XPBD algorithm relies on the correction of positions by using constraints. A constraint is function that takes as an input the positions of the system and returns a scalar. The goal of XPBD is to satisfy the constraints, i.e. we can impose it to be equal to or greater than a constant (often 0). The only condition is that the constraint needs to be differentiable. So the algorithm is flexible because it allows various constraints.

Each constraint is associated with a inverse stiffness parameter $\tilde{\alpha}$. It is defined by $\tilde{\alpha} = \alpha/dt^2$. XPBD supports infinite stiffness by setting α to 0, which was not possible with the original PBD algorithm. When it is not 0, α usually ranges from 10^{-10} to 10^{-2}.

Each position is associated with a mass, or rather an inverse mass w. So the algorithm supports infinite masses by setting w to 0.

The algorithm is described in 1. It is composed of three steps:

(1) Predict: Apply velocity and external forces to the system
(2) Correct the constraints
(3) Update velocity

The most important part of the algorithm is solving the constraints. We define local constraints and we try to solve them iteratively. At one point, they will be satisfied. To do this, we first compute a Lagrangian coefficient and then modify the positions of the system.

The strenght of XPBD is that the solver converges; the simulation will have the same behavior as long as there is enough iterations. It was not the case with the original PBD algorithm, as the stiffness of the constraints increased as we increased the number of iterations.

ALGORITHM 1: XPBD simulation loop

Predict position $\tilde{x} \leftarrow x^n + \Delta t v^n + \Delta t^2 M^{-1} f_{\text{ext}}(x^n)$
Initialize the solver variables $x_0 \leftarrow \tilde{x}$
Initialize multipliers $\lambda_0 \leftarrow 0$
for $i \leftarrow 0$ ***to*** *solverIterations* **do**
  for ***each*** *constraint j* **do**
    $\Delta\lambda \leftarrow \dfrac{-C_j(x_i) - \tilde{\alpha}_j \lambda_{ij}}{\nabla C_j M^{-1} \nabla C_j^T + \tilde{\alpha}_j}$
    $\Delta x \leftarrow M^{-1} \nabla C(x_i)^T \Delta\lambda$
    update $\lambda_{i+1j} \leftarrow \lambda_{ij} + \Delta\lambda$
    update $x_{i+1} \leftarrow x_i + \Delta x$
  end
end
update positions $x^{n+1} \leftarrow x_i$
update velocities $v^{n+1} \leftarrow \dfrac{1}{\Delta t}\left(x^{n+1} - x^n\right)$

2.2 Substeps

Recent works ([Macklin et al. 2019]) have shown that we can simplify the algorithm by using substeps instead of iterating multiple times to solve the constraints. The new algorithm is described in algorithm 2. As we can see, we no longer have to keep track of λ.

However implementing directly this algorithm makes the solver explode. This is because substepping brings numerical errors. To fix this, we must include damping in the solver loop, by modifying the $\Delta\lambda$ calculation.

$$\Delta\lambda = \frac{-C_j(x_i) - \gamma_i \nabla C_j \cdot (x - x^n)}{(1+\gamma_i)\nabla C_j M^{-1} \nabla C_j^T + \tilde{\alpha}_j} \tag{1}$$

Here we define γ_i a damping constant, by $\gamma_i = \frac{\tilde{\alpha}_i \tilde{\beta}_i}{\Delta t_s}$ with $\tilde{\beta}_i = \Delta t^2 \beta$. A value of $3 \cdot 10^{-8}$ for β worked well in the implementation.

This report is submitted as a part of project for Advanced 3D Computer Graphics (IMA904/IG3DA), Telecom Paris.

The original work is introduced by [Macklin et al. 2016].

ALGORITHM 2: XPBD simulation loop with substeps

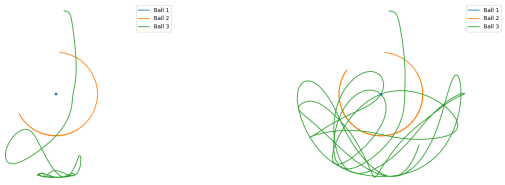
$\Delta t_s \leftarrow \Delta t / \text{solverIterations}$
for $i \leftarrow 0$ ***to*** *solverIterations* **do**
  Predict position $\tilde{x} \leftarrow x^n + \Delta t_s v^n + \Delta t_s^2 M^{-1} f_{\text{ext}}(x^n)$
  for ***each*** *constraint* j **do**
    $\Delta\lambda \leftarrow \dfrac{-C_j(x_i)}{\nabla C_j M^{-1} \nabla C_j^T + \tilde{\alpha}_j}$
    $\Delta x \leftarrow w \nabla C(x_i)^T \Delta\lambda$
    update $x_{i+1} \leftarrow x_i + \Delta x$
  end
  update positions $x_{i+1} \leftarrow x_i$
  update velocities $v_{i+1} \leftarrow \dfrac{1}{\Delta t_s}\left(x^{n+1} - x^n\right)$
end

3 IMPLEMENTATION

3.1 Cord

The simplest constraint is a distance constraint, i.e. two positions must have a constant distance. With this we can create a cord by using successive distance constraint. We can fix one particle by setting its inverse mass to 0.

Having only 3 particles allows to see the difference between the original implementation and substepping. In figure 1, the position of the particles over time is represented in the plane. The cord has been released from an almost vertical position. We see that the energy is quickly lost with the iterations, and the cord goes to a rest pose. Here, no damping has been added, so that means the loss is unevitable. However with the substeps, we see a more chaotic behavior, that is closer to a real double pendulum. We couldn't achieve this result before, even by increasing the stiffness of the constraints. The simulation also runs for much longer. Both methods used 20 iterations or substeps per frame.



(a) Iterations (b) Substeps

Fig. 1. Position of a 2-segments cord over time

So if we want a more elastic simulation, we can use multiple iterations, and if we want to keep the energy we can use substeps. But we can go further and mix the two methods to have intermediate results. Figure 2 shows what happens with 4 substeps and 5 iterations per frame, so again with 20 steps per frame in total. We still have the chaos and energy of a double pendulum, but the energy is dissipated quicker.

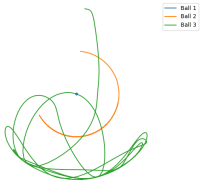


Fig. 2. Hybrid method with iterations and substeps

3.2 Cloth

To represent a cloth, we can create a rectangular mesh with many subdivisions, and add a distance constraint on each segment of the mesh. However this gives the impression of a very thin cloth, because there are many small wrinkles. The effect is much more visible at the bottom of the cloth. To fix this, we can add a bending constraint. The comparison can observed in figure 4.

There are two approaches for the bending constraint. We can first impose two adjacent faces to have a fixed angle, and the other is to add a distance constraint on the opposing vertices of adjacent faces. This is illustrated in figure 3.

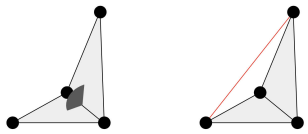
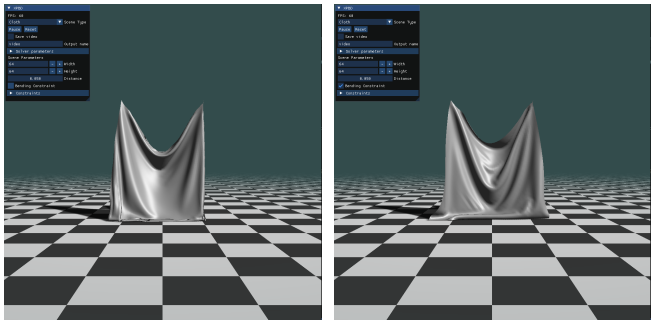


Fig. 3. Bending constraint: angle based (left) vs distance based (right)

The distance based method is one used because it is simpler and faster to compute. However it allows the faces to flip, so keeping two faces almost in a plane is difficult. But for a cloth it works well.



(a) No bending constraint (b) With bending constraint

Fig. 4. Adding a bending constraint

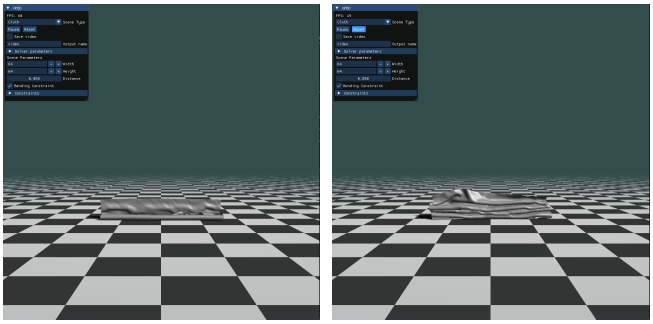
We can also add self collision, but we have to take some precautions. To do this, we followed the recommendations of [Physics 2022].

To accelerate collision detection, we can use a spatial hash to make a broad detection. It is enough to do it once per frame. In the

code, the hash is in *src/utils/SpatialGrid.hpp*. To use it, we can add particles to the grid and they will be added to a cell of the dimension of collision distance. Then we can get the neighbors of a particle by looking at those who are in the neighboring cells. We can finally add a constraint that keeps neighbors at a minimal distance. Thoses constraints must be reset at every frame.

However this is not enough, as particles may go through the cloth be stuck. To prevent this, we can use substepping and add a maximum velocity to avoid particles to go too far away at each step. This avoids us to use a continuous collsion detection.

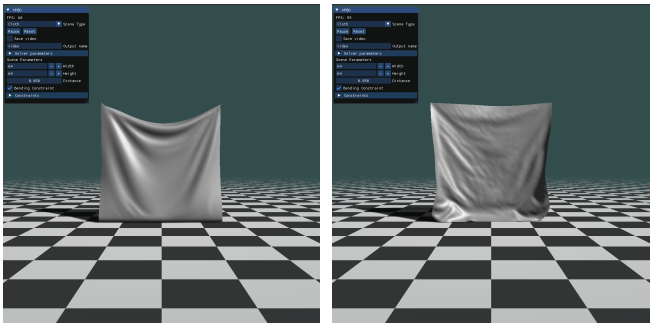
The result can be observed in figure 5. The cloth was dropped in a vertical position and the image was taken after it fell completely on the floor. Without self collision constraint, we can see that the cloth is not flat because of the bending constraints.



(a) No collision constraint (b) With collision constraint

Fig. 5. Adding a collision constraint

Finally, using substeps also increases the stiffness of the constraints, as we can see in figure 6. Here, the top line becomes almost straight.



(a) Iterations (b) Substeps

Fig. 6. Coth: iterations vs substeps

3.3 Soft body

A soft body is a deformable solid. Distance and bending constraints are not enough to simulate a soft body, because nothing prevents it from collapsing. There are no internal forces. To fix this, we can add a volume constraint.

To keep the internal structure, we need to use a mesh that includes tetrahedrons, similar to the type used in Finite Element Method (FEM) simulations. We can add a distance constraint on the edges of the tetrahedrons and a volume constraint on each tetrahedron. The mesh and the tetrahedrons used were found here.

The small features of the soft body are conserved, such as ears as we can see in figure 7. But we can play with constraints to have a softer body. The most important constraints here is the distance constraint, as it's what keeps the structure. If we decrease the stiffness, the edges can stretch and the body be more deformed, as seen in figure 8.

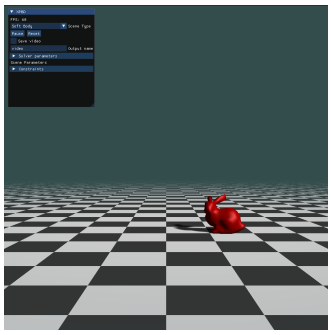
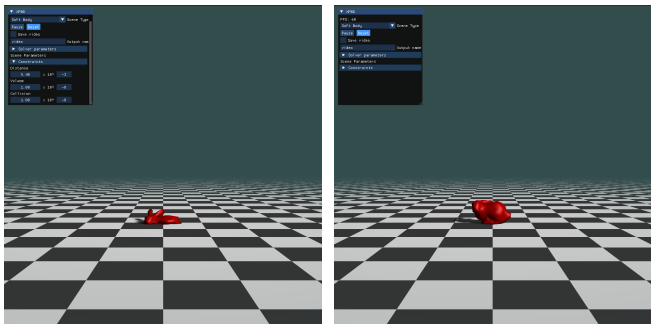


Fig. 7. Soft body



(a) Low stiffness (b) High stiffness

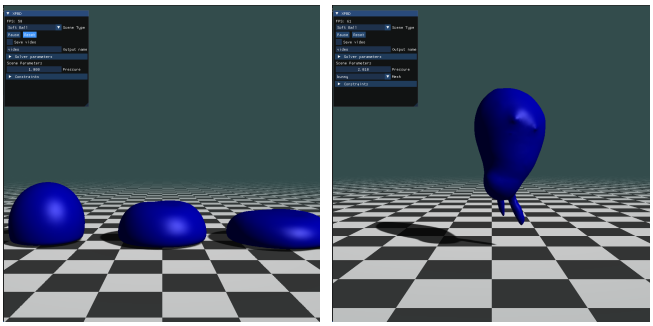
Fig. 8. Soft body after a drop

Some features such as the ears a quite mobile, even if we increase the distance stiffness. They keep their shape but because are attached to the body with a thin base, the constraints are not strong enough from preventing from moving. But that is a natural behaviour we can expect from a soft body. If we want the ears to be fixed, we can add bending constraints. Also, using substeps allows them to be fixed. So we can choose the mode depending on the behaviour we want.

3.4 Balloon

Instead of having a local constraint for the volume, we can try to have a global constraint, by keeping the volume of the whole closed shape to be constant. This can be used to simulate an inflatable

balloon. We have to add a distance constraint on each segment of the mesh to keep the balloon shape. Also, we can play with the deformation of the balloon if we modify the stiffness parameter. The result is seen in figure 9.



(a) Decreasing distance constraint stiffness (b) Bunny mesh

Fig. 9. Balloon

As we can see, small features are not as well preserved as previously. The face of the bunny and the paws are flattened. But this is normal, as there is no bending constraint that maintain them. However this constraint is interesting because it is unique, and don't require an internal structure, so it is faster to compute.

We can also add more pressure inside the balloon by forcing the constraint to keep a bigger value. Then it will fight even more with the distance constraints to keep the shape of the object.

4 TECHNICAL DETAILS

The implementation has been done in C++ and rendered with OpenGL. Shadows have been created with a shadow map. The checker pattern on the floor is done in a shader.

The code had the objective to be modular and well structured. Each scene and each constraint is a children of a virtual class, that allows the solver to be very generic, and the program to be flexible. Each scene has its own setup and render function. It also allows to change parameters the parameters of the scene with a UI made with ImGui. All constraints have a function to compute them or to compute the gradient. They also store the particles they affect.

The scenes are located in the *src/scenes/* folder. The *src/simulation/* folder contains the solver and a scene manager that allows to reset and change scenes, as well as updating them.

The simulation runs and is capped at 60 fps, but can be slower with a lot of constraints or with cloth self-collision. But the robustness of PBD handles it well.

5 DIFFICULTIES ENCOUNTERED

I struggled at first to implement substeps, as I did not know that a damping was needed to make it work.

The angle-based bending constraint was also difficult to implement, as the formula was a bit complicated and also introduced some approximation error.

The soft body and the balloon starts to rotate when on the ground, I don't know the cause. I tried to shuffle the constraints but it does not work.

I tried to implement rigid bodies and a fluid simulation, but couldn't make it work on time. For the rigid bodies, I wanted to use a shape matching algorithm, that tries to find the best rotation and translation to match the deformed shape. However those situations may require to write a dedicated solver, as they need some tweaks to work correctly.

6 FUTURE WORK

Position based dynamics is wide simulation tool, so we can imagine infinitely many situations. The algorithm can be extended to rigid bodies, fluids, but it can also simulate sand or muscles for example.

Also, we could try to break some constraints if they are too much violated. Or we could at least modify them to have a plastic behaviour.

Finally, we could add some friction between the objects.

7 CONCLUSION

This project made me discover an all-purpose simulation tool that is easy to understand yet very powerfull. I explored many situations and constraints, and also compared iterating and substepping in the solver loop.

8 SUPPLEMENTARY MATERIALS

The code of the project is available here.

ACKNOWLEDGMENTS

The author would like to thank Professor Kiwon Um from Telecom Paris for for his guidance and support throughout his course and this project.

REFERENCES

Miles Macklin, Matthias Müller, and Nuttapong Chentanez. 2016. XPBD: position-based simulation of compliant constrained dynamics. In *Proceedings of the 9th International Conference on Motion in Games (MIG '16)*. Association for Computing Machinery, New York, NY, USA, 49–54. DOI: https://doi.org/10.1145/2994258.2994272

Miles Macklin, Kier Storey, Michelle Lu, Pierre Terdiman, Nuttapong Chentanez, Stefan Jeschke, and Matthias Müller. 2019. Small steps in physics simulation. In *Proceedings of the 18th Annual ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA '19)*. Association for Computing Machinery, New York, NY, USA, Article 2, 7 pages. DOI: https://doi.org/10.1145/3309486.3340247

Matthias Müller, Bruno Heidelberger, Marcus Hennix, and John Ratcliff. 2007. Position based dynamics. *J. Vis. Comun. Image Represent.* 18, 2 (April 2007), 109–118. DOI: https://doi.org/10.1016/j.jvcir.2007.01.005

Ten Minutes Physics. 2022. 15 - Self-collisions, solving the hardest problem in animation. (2022). https://www.youtube.com/watch?v=XY3dLpgOk4Q See also the related PDF: https://matthias-research.github.io/pages/tenMinutePhysics/15-selfCollision.pdf.