Lagrangian Simulation Approaches

Mass-Spring Systems and Smoothed Particle Hydrodynamics

Yuanming Hu

MIT CSAIL

June 8, 2020

Table of Contents

Mass-spring Systems

Demo!

Mass-spring systems

$$
\begin{aligned}
\mathbf{f}_{ij} &= -k(||\mathbf{x}_i - \mathbf{x}_j||_2 - l_{ij})(\widehat{\mathbf{x}_i - \mathbf{x}_j}) \quad \text{(Hooke's Law)} \\
\mathbf{f}_i &= \sum_j^{j \neq i} \mathbf{f}_{ij} \\
\frac{\partial \mathbf{v}_i}{\partial t} &= \frac{1}{m_i}\mathbf{f}_i \quad \text{(Newton's second law of motion)} \\
\frac{\partial \mathbf{x}_i}{\partial t} &= \mathbf{v}_i
\end{aligned}
$$

k: spring stiffness; l_{ij}: spring rest length between particle i and particle j; m_i: the mass of particle i. $(\widehat{\mathbf{x}_i - \mathbf{x}_j})$: direction vector from particle i to particle i; $\widehat{\Box}$ means **normalization**.

Table of Contents

Time integration

1. Forward Euler (explicit)

$$\begin{aligned}\mathbf{v}_{t+1} &= \mathbf{v}_t + \Delta t \frac{\mathbf{f}_t}{m} \\ \mathbf{x}_{t+1} &= \mathbf{x}_t + \Delta t \mathbf{v}_t\end{aligned}$$

2. Semi-implicit Euler (aka. symplectic Euler, explicit)

$$\begin{aligned}\mathbf{v}_{t+1} &= \mathbf{v}_t + \Delta t \frac{\mathbf{f}_t}{m} \\ \mathbf{x}_{t+1} &= \mathbf{x}_t + \Delta t \mathbf{v}_{t+1}\end{aligned}$$

3. Backward Euler (often with Newton's method, implicit)

Implementing a mass-spring system with symplectic Euler

1. Compute new velocity using $\mathbf{v}_{t+1} = \mathbf{v}_t + \Delta t \frac{\mathbf{f}_t}{m}$
2. Collision with ground
3. Compute new position using $\mathbf{x}_{t+1} = \mathbf{x}_t + \Delta t \mathbf{v}_{t+1}$

Implementing a mass-spring system with symplectic Euler

Showcase `mass_spring.py`

```
@ti.kernel
def substep():
    n = num_particles[None]

    # Compute force and new velocity
    for i in range(n):
        v[i] *= ti.exp(-dt * damping[None]) # damping
        total_force = ti.Vector(gravity) * particle_mass
        for j in range(n):
            if rest_length[i, j] != 0:
                x_ij = x[i] - x[j]
                total_force += -spring_stiffness[None] * (x_ij.norm() - rest_length[i, j]) * x_ij.
                    normalized()
        v[i] += dt * total_force / particle_mass

    # Collide with ground
    for i in range(n):
        if x[i].y < bottom_y:
            x[i].y = bottom_y
            v[i].y = 0

    # Compute new position
    for i in range(n):
        x[i] += v[i] * dt
```

Explicit v.s. implicit time integrators

Explicit (forward Euler, symplectic Euler, RK, ...):

- Future depends only on past
- Easy to implement
- Easy to explode:

$$\Delta t \leq c\sqrt{\frac{m}{k}} \quad (c \sim 1)$$

- Bad for stiff materials

Implicit (backward Euler, middle-point, ...):

- Future depends on both future and past
- Chicken-egg problem: need to solve a system of (linear) equations
- In general harder to implement
- Each step is more expensive but time steps are larger
  - Sometimes brings you benefits
  - ... but sometimes not
- Numerical damping and locking

Mass-spring systems

Implicit time integration:

$$\mathbf{x}_{t+1} = \mathbf{x}_t + \Delta t \mathbf{v}_{t+1} \tag{1}$$

$$\mathbf{v}_{t+1} = \mathbf{v}_t + \Delta t \mathbf{M}^{-1}\mathbf{f}(\mathbf{x}_{t+1}) \tag{2}$$

Eliminate $\mathbf{x}_{t+1}$:

$$\mathbf{v}_{t+1} = \mathbf{v}_t + \Delta t \mathbf{M}^{-1}\mathbf{f}(\mathbf{x}_t + \Delta t \mathbf{v}_{t+1}) \tag{3}$$

Linearize (one step of Newton's method):

$$\mathbf{v}_{t+1} = \mathbf{v}_t + \Delta t \mathbf{M}^{-1}\left[\mathbf{f}(\mathbf{x}_t) + \frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\mathbf{x}_t)\Delta t \mathbf{v}_{t+1}\right] \tag{4}$$

After linearization

Linearize:

$$\mathbf{v}_{t+1} = \mathbf{v}_t + \Delta t \mathbf{M}^{-1}\left[\mathbf{f}(\mathbf{x}_t) + \frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\mathbf{x}_t)\Delta t \mathbf{v}_{t+1}\right] \tag{5}$$

Clean up:

$$\left[\mathbf{I} - \Delta t^2 \mathbf{M}^{-1}\frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\mathbf{x}_t)\right]\mathbf{v}_{t+1} = \mathbf{v}_t + \Delta t \mathbf{M}^{-1}\mathbf{f}(\mathbf{x}_t) \tag{6}$$

A nice *linear* system!

Solving the system

$$\left[\mathbf{I} - \Delta t^2 \mathbf{M}^{-1} \frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\mathbf{x}_t)\right] \mathbf{v}_{t+1} = \mathbf{v}_t + \Delta t \mathbf{M}^{-1} \mathbf{f}(\mathbf{x}_t) \tag{7}$$

How to solve it?

- Jacobi/Gauss-Seidel iterations (easy to implement!)
- Conjugate gradients (later in this course)

Solving the system

$$
\begin{aligned}
\mathbf{A} &= \left[\mathbf{I} - \Delta t^2 \mathbf{M}^{-1} \frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\mathbf{x}_t)\right] \\
\mathbf{b} &= \mathbf{v}_t + \Delta t \mathbf{M}^{-1} \mathbf{f}(\mathbf{x}_t) \\
\mathbf{A}\mathbf{v}_{t+1} &= \mathbf{b}
\end{aligned}
$$

Solving linear systems with Jacobi iterations (Demo!)


```
A = ti.var(dt=ti.f32, shape=(n, n))
x = ti.var(dt=ti.f32, shape=n)
new_x = ti.var(dt=ti.f32, shape=n)
b = ti.var(dt=ti.f32, shape=n)

@ti.kernel
def iterate():
    for i in range(n):
        r = b[i]
        for j in range(n):
            if i != j:
                r -= A[i, j] * x[j]

        new_x[i] = r / A[i, i]

    for i in range(n):
        x[i] = new_x[i]
```

Unifying explicit and implicit integrators

$$\left[\mathbf{I} - \beta\Delta t^2\mathbf{M}^{-1}\frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\mathbf{x}_t)\right]\mathbf{v}_{t+1} \quad = \quad \mathbf{v}_t + \Delta t\mathbf{M}^{-1}\mathbf{f}(\mathbf{x}_t)$$

1. $\beta = 0$: forward/semi-implicit Euler (explicit)
2. $\beta = 1/2$: middle-point (implicit)
3. $\beta = 1$: backward Euler (implicit)

Solve faster

What if we have millions of mass points and springs?

- Sparse matrices
- Conjugate gradients
- Preconditioning
- Use position-based dynamics[1]

A different (yet much faster) approach: Fast mass-spring system solver[2]

[1] M. Müller et al. (2007). "Position based dynamics". In: *Journal of Visual Communication and Image Representation* 18.2, pp. 109–118.

[2] T. Liu et al. (2013). "Fast simulation of mass-spring systems". In: *ACM Transactions on Graphics (TOG)* 32.6, pp. 1–7.

Table of Contents

Smoothed particle hydrodynamics

High-level idea: use particles carrying samples of physical quantities, and a kernel function W, to approximate continuous fields: (A can be almost any spatially varying physical attributes: density, pressure, etc. Derivatives: different story)

$$A(\mathbf{x}) = \sum_i A_i \frac{m_i}{\rho_i} W(||\mathbf{x} - \mathbf{x}_j||_2, h)$$

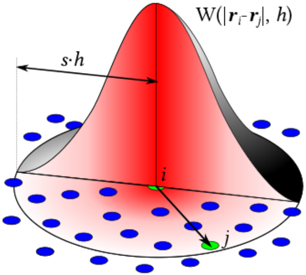

Figure: SPH particles and their kernel (source: Wikipedia)

Smoothed particle hydrodynamics

1. Originally proposed for astrophysical problems[3]
2. No meshes. Very suitable for free-surface flows[4]!
3. Easy to understand intuitively: just imagine each particle is a small parcel of water (although strictly not the case!)

[3] R. A. Gingold and J. J. Monaghan (1977). "Smoothed particle hydrodynamics: theory and application to non-spherical stars". In: *Monthly notices of the royal astronomical society* 181.3, pp. 375–389.

[4] J. J. Monaghan (1994). "Simulating free surface flows with SPH". In: *Journal of computational physics* 110.2, pp. 399–406.

Implementing SPH using the Equation of States (EOS)

Also known as Weakly Compressible SPH (WCSPH)[5].

Momentum equation: (ρ: density; B: bulk modulus; γ: constant, usually ~ 7)

$$\frac{D\mathbf{v}}{Dt} = -\frac{1}{\rho}\nabla p + \mathbf{g}, \quad p = B\left(\left(\frac{\rho}{\rho_0}\right)^{\gamma} - 1\right)$$

$$A(\mathbf{x}) = \sum_i A_i \frac{m_i}{\rho_i} W(||\mathbf{x} - \mathbf{x}_j||_2, h), \quad \rho_i = \sum_j m_j W(||\mathbf{x}_i - \mathbf{x}_j||_2, h)$$

Extras: surface tension, viscosity; (very) nice tutorial[6]

Note: the WCSPH paper should have used material derivatives.

[5] M. Becker and M. Teschner (2007). "Weakly compressible SPH for free surface flows". In: *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation.* Eurographics Association, pp. 209–217.

[6] D. Koschier et al. (2019). "Smoothed Particle Hydrodynamics Techniques for the Physics Based Simulation of Fluids and Solids". In:

Gradients in SPH

$$A(\mathbf{x}) = \sum_i A_i \frac{m_i}{\rho_i} W(||\mathbf{x} - \mathbf{x}_j||_2, h)$$

$$\nabla A_i = \rho_i \sum_j m_j \left(\frac{A_i}{\rho_i^2} + \frac{A_j}{\rho_j^2} \right) \nabla_{\mathbf{x}_i} W(||\mathbf{x}_i - \mathbf{x}_j||_2, h)$$

- Not really accurate...
- but at least symmetric and momentum conserving!

Now we can compute ∇p_i.
Extension: Laplace operator (viscosity etc.)...

SPH Simulation Cycle

Recall:

$$\frac{D\mathbf{v}}{Dt} = -\frac{1}{\rho}\nabla p + \mathbf{g}, p = B\left(\left(\frac{\rho}{\rho_0}\right)^{\gamma} - 1\right)$$

1. For each particle i, compute $\rho_i = \sum_j m_j W(||\mathbf{x}_i - \mathbf{x}_j||_2, h)$
2. For each particle i, compute ∇p_i using the gradient operator
3. Symplectic Euler step (again...):

$$\mathbf{v}_{t+1} = \mathbf{v}_t + \Delta t \frac{D\mathbf{v}}{Dt}$$

$$\mathbf{x}_{t+1} = \mathbf{x}_t + \Delta t \mathbf{v}_{t+1}$$

Variants of SPH

Recent updates:

- ...
- Predictive-Corrective Incompressible SPH (PCI-SPH)[7]
- Position-based fluids (PBF)[8] **Demo:** `ti example pbf2d`
- Divergence-free SPH (DFSPH[9])
- ...

Survey paper: *SPH Fluids in Computer Graphics*[10]

---

[7] B. Solenthaler and R. Pajarola (2009). "Predictive-corrective incompressible SPH". In: *ACM SIGGRAPH 2009 papers*, pp. 1–6.

[8] M. Macklin and M. Müller (2013). "Position based fluids". In: *ACM Transactions on Graphics (TOG)* 32.4, pp. 1–12.

[9] J. Bender and D. Koschier (2015). "Divergence-free smoothed particle hydrodynamics". In: *Proceedings of the 14th ACM SIGGRAPH/Eurographics symposium on computer animation*, pp. 147–155.

[10] M. Ihmsen et al. (2014). "SPH fluids in computer graphics". In:

Courant–Friedrichs–Lewy (CFL) condition

One upper bound of time step size:

$$C = \frac{u\Delta t}{\Delta x} \le C_{\text{max}} \sim 1$$

- C: CFL number (Courant number, or simple the CFL)
- Δt: time step
- Δx: length interval (e.g. particle radius and grid size)
- u: maximum (velocity)

Application: estimating allowed time step in (explicit) time integrations.
Typical C_{max} in graphics:

- SPH: ~ 0.4
- MPM: $0.3 \sim 1$
- FLIP fluid (smoke): $1 \sim 5+$

Accelerating SPH: Neighborhood search

So far, per substep complexity of SPH is $O(n^2)$. This is too costly to be practical. In practice, people build spatial data structure such as voxel grids to accelerate neighborhood search. This reduces time complexity to $O(n)$.

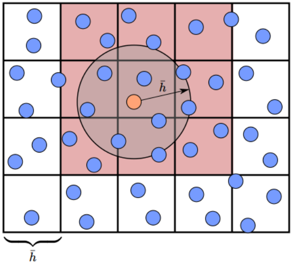

Figure: Neighborhood search with hashing. Source: Koschier et al. 2019.

Reference: Compact hashing

Other particle-based simulation methods

- Discrete element method, e.g.[11]
- Moving Particle Semi-implicit (MPS)[12]
- Power Particles: An incompressible fluid solver based on power diagrams[13]
- A peridynamic perspective on spring-mass fracture[14]
- ...

[11] N. Bell, Y. Yu, and P. J. Mucha (2005). "Particle-based simulation of granular materials". In: *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pp. 77–86.

[12] S. Koshizuka and Y. Oka (1996). "Moving-particle semi-implicit method for fragmentation of incompressible fluid". In: *Nuclear science and engineering* 123.3, pp. 421–434.

[13] F. de Goes et al. (2015). "Power particles: an incompressible fluid solver based on power diagrams.". In: *ACM Trans. Graph.* 34.4, pp. 50–1.

[14] J. A. Levine et al. (2014). "A peridynamic perspective on spring-mass fracture". In: *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation.* Citeseer, pp. 47–55.

Exporting your results `taichi v0.6.8`

Make an `mp4` video out of your frames

1. Use `ti.GUI.show` [doc] to save the screenshots. Or simply use `ti.imwrite(img, filename)` [doc].
2. `ti video` creates `video.mp4` using frames under the current folder. To specify frame rate, use `ti video -f 24` or `ti video -f 60`.
3. Convert `mp4` to `gif` and share it online: `ti gif -i input.mp4`.

Make sure `ffmpeg` works!

- Linux and OS X: with high probability you already have `ffmpeg`.
- Windows: install `ffmpeg` on your own [doc].

More information: [Documentation] Export your results.