lecture02

Zeffiretti Hiesh

2021 年 6 月 7 日

0.0.1 Lagrangian v.s. Eulerian: Two Views of Continuums

1. Lagrangian View, 拉格朗日视角:
   - Sensors that move passively with the simulated material(随波逐流)
   - 粒子
2. Eulerian View, 欧拉视角:
   - Still sensors that never moves.(岿然不动)
   - 网格

0.0.2 Mass-Spring System, 弹簧-质点模型

- Extremely ordinary
- But very useful!
  - Cloth
  - Elastic objects
  - ...

Mathematical Model

$$
\begin{aligned}
\mathrm{f}_{ij} &= -k(||\mathrm{x}_i - \mathrm{x}_j||_2 - l_{ij})(\widehat{\mathrm{x}_i - \mathrm{x}_j}) \quad (Hooke's Law) \\
\mathrm{f}_i &= \sum_{j}^{j \neq i} \mathrm{f}_{ij} \\
\frac{\partial \mathrm{v}}{\partial t} &= \frac{1}{m_i}\mathrm{f}_i
\end{aligned}
$$

k: spring stiffness;

l_{ij}: spring rest length between particle i and particle j;

m_i: the mass of particle i.

$(\widehat{\mathrm{x}_i - \mathrm{x}_j})$: direction vector from particle i to particle j;

Time integration

- Forward Euler (explicit)

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

- Semi-implicit Euler (aka. sympletic Euler, explicit)

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

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

Implementing a mass-spring system with sympletic Euler

Steps: 1. Compute new velocity using $\mathrm{v}t+1 = \mathrm{v}_t + \Delta t \frac{\mathrm{f}_t}{m}$ 2. Collision with ground 3. Compute new position using $\mathrm{x}_{t+1} = \mathrm{x}_t + \Delta t \mathrm{v}_{t_1}$

[ ]:
```
# Showcase
# Tutorials (Chinese):
# - https://www.bilibili.com/video/BV1UK4y177iH
# - https://www.bilibili.com/video/BV1DK411A771

import taichi as ti

ti.init(arch=ti.gpu)

spring_Y = ti.field(dtype=ti.f32, shape=())  # Young's modulus
paused = ti.field(dtype=ti.i32, shape=())
drag_damping = ti.field(dtype=ti.f32, shape=())
dashpot_damping = ti.field(dtype=ti.f32, shape=())

max_num_particles = 1024
particle_mass = 1.0
dt = 1e-3
substeps = 10

num_particles = ti.field(dtype=ti.i32, shape=())
```

```
x = ti.Vector.field(2, dtype=ti.f32, shape=max_num_particles)
v = ti.Vector.field(2, dtype=ti.f32, shape=max_num_particles)
f = ti.Vector.field(2, dtype=ti.f32, shape=max_num_particles)
fixed = ti.field(dtype=ti.i32, shape=max_num_particles)

# rest_length[i, j] == 0 means i and j are NOT connected
rest_length = ti.field(dtype=ti.f32,
                       shape=(max_num_particles, max_num_particles))


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

    # Compute force
    for i in range(n):
        # Gravity
        f[i] = ti.Vector([0, -9.8]) * particle_mass
        for j in range(n):
            if rest_length[i, j] != 0:
                x_ij = x[i] - x[j]
                d = x_ij.normalized()

                # Spring force
                f[i] += -spring_Y[None] * (x_ij.norm() / rest_length[i, j] -
                                           1) * d

                # Dashpot damping
                v_rel = (v[i] - v[j]).dot(d)
                f[i] += -dashpot_damping[None] * v_rel * d

    # We use a semi-implicit Euler (aka symplectic Euler) time integrator
    for i in range(n):
        if not fixed[i]:
            v[i] += dt * f[i] / particle_mass
            v[i] *= ti.exp(-dt * drag_damping[None])  # Drag damping
```

```
            x[i] += v[i] * dt
        else:
            v[i] = ti.Vector([0, 0])

        # Collide with four walls
        for d in ti.static(range(2)):
            # d = 0: treating X (horizontal) component
            # d = 1: treating Y (vertical) component

            if x[i][d] < 0:  # Bottom and left
                x[i][d] = 0  # move particle inside
                v[i][d] = 0  # stop it from moving further

            if x[i][d] > 1:  # Top and right
                x[i][d] = 1  # move particle inside
                v[i][d] = 0  # stop it from moving further


@ti.kernel
def new_particle(pos_x: ti.f32, pos_y: ti.f32, fixed_: ti.i32):
    # Taichi doesn't support using vectors as kernel arguments yet, so we pass␣
 ↪scalars
    new_particle_id = num_particles[None]
    x[new_particle_id] = [pos_x, pos_y]
    v[new_particle_id] = [0, 0]
    fixed[new_particle_id] = fixed_
    num_particles[None] += 1

    # Connect with existing particles
    for i in range(new_particle_id):
        dist = (x[new_particle_id] - x[i]).norm()
        connection_radius = 0.15
        if dist < connection_radius:
            # Connect the new particle with particle i
            rest_length[i, new_particle_id] = 0.1
```

```
            rest_length[new_particle_id, i] = 0.1


@ti.kernel
def attract(pos_x: ti.f32, pos_y: ti.f32):
    for i in range(num_particles[None]):
        p = ti.Vector([pos_x, pos_y])
        v[i] += -dt * substeps * (x[i] - p) * 100


def main():
    gui = ti.GUI('Explicit Mass Spring System',
                 res=(512, 512),
                 background_color=0xDDDDDD)

    spring_Y[None] = 1000
    drag_damping[None] = 1
    dashpot_damping[None] = 100

    new_particle(0.3, 0.3, False)
    new_particle(0.3, 0.4, False)
    new_particle(0.4, 0.4, False)

    while True:
        for e in gui.get_events(ti.GUI.PRESS):
            if e.key in [ti.GUI.ESCAPE, ti.GUI.EXIT]:
                exit()
            elif e.key == gui.SPACE:
                paused[None] = not paused[None]
            elif e.key == ti.GUI.LMB:
                new_particle(e.pos[0], e.pos[1],
                             int(gui.is_pressed(ti.GUI.SHIFT)))
            elif e.key == 'c':
                num_particles[None] = 0
                rest_length.fill(0)
            elif e.key == 'y':
```

```
            if gui.is_pressed('Shift'):
                spring_Y[None] /= 1.1
            else:
                spring_Y[None] *= 1.1
        elif e.key == 'd':
            if gui.is_pressed('Shift'):
                drag_damping[None] /= 1.1
            else:
                drag_damping[None] *= 1.1
        elif e.key == 'x':
            if gui.is_pressed('Shift'):
                dashpot_damping[None] /= 1.1
            else:
                dashpot_damping[None] *= 1.1

    if gui.is_pressed(ti.GUI.RMB):
        cursor_pos = gui.get_cursor_pos()
        attract(cursor_pos[0], cursor_pos[1])

    if not paused[None]:
        for step in range(substeps):
            substep()

    X = x.to_numpy()
    n = num_particles[None]

    # Draw the springs
    for i in range(n):
        for j in range(i + 1, n):
            if rest_length[i, j] != 0:
                gui.line(begin=X[i], end=X[j], radius=2, color=0x444444)

    # Draw the particles
    for i in range(n):
        c = 0xFF0000 if fixed[i] else 0x111111
        gui.circle(pos=X[i], color=c, radius=5)
```

```
        gui.text(
            content=
            f'Left click: add mass point (with shift to fix); Right click:␣
 ↪attract',
            pos=(0, 0.99),
            color=0x0)
        gui.text(content=f'C: clear all; Space: pause',
                 pos=(0, 0.95),
                 color=0x0)
        gui.text(content=f'Y: Spring Young\'s modulus {spring_Y[None]:.1f}',
                 pos=(0, 0.9),
                 color=0x0)
        gui.text(content=f'D: Drag damping {drag_damping[None]:.2f}',
                 pos=(0, 0.85),
                 color=0x0)
        gui.text(content=f'X: Dashpot damping {dashpot_damping[None]:.2f}',
                 pos=(0, 0.8),
                 color=0x0)
        gui.show()


if __name__ == '__main__':
    main()
```

Explicit v.s. implicit time integrators

- Explicit (forward Euler, sympletic Euler, RK, ...):
  - Feature depends only on past
  - Easy to implement
  - Easy to explode: $\Delta t \leq c\sqrt{m/k}$, $(c \sim 1)$
  - Bad for stiff materials
- Implicit (backword Euler, middle-point, ...):
  - Future denpends on both future and past
  - Chicken-egg problem: need to solve a system of (linear) equations
  - Ingeneral harder to implement
  - Each step is more expensive but time steps are larger

    - Sometimes brings you benefits
    - ... but sometimes not
  - Numerical damping and locking

Implementing

- Implicit time integration:

$$\begin{aligned} \mathrm{x}_{t+1} &= \mathrm{x}_t + \Delta t \mathrm{v}_{t+1} \\ \mathrm{v}_{t+1} &= \mathrm{v}_t + \Delta t \mathrm{M}^{-1} \mathrm{f}(\mathrm{x}_{t+1}) \end{aligned}$$

- Eliminate x_{t+1}:

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

- Linearize (one step of Newton's method):

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

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

How to solve it?

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

雅可比迭代法

对于矩阵 $Ax = b$, A非奇异，且对角元不为0，可以将原方程组改写为：

$$\begin{cases} x_1 = \dfrac{1}{a_{11}} \left(b_1 - a_{11} x_2 - ... - a_{1n} x_n \right), \\ x_2 = \dfrac{1}{a_{22}} \left(b_2 - a_{21} x_1 - ... - a_{2n} x_n \right), \\ ..... \\ x_n = \dfrac{1}{a_{nn}} \left(b_n - a_{n1} x_1 - ... - a_{(n,n-1)} x_{n-1} \right), \end{cases}$$

[ ]:
```python
import taichi as ti
import random

ti.init(arch=ti.cpu)

n = 20

A = ti.field(dtype=ti.f32, shape=(n, n))
x = ti.field(dtype=ti.f32, shape=n)
new_x = ti.field(dtype=ti.f32, shape=n)
b = ti.field(dtype=ti.f32, shape=n)


# 单步雅可比迭代
@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]


# 计算误差
@ti.kernel
def residual() -> ti.f32:
    res = 0.0

    for i in range(n):
        r = b[i] * 1.0
        for j in range(n):
```

```
            r -= A[i, j] * x[j]
        res += r * r

    return res


for i in range(n):
    for j in range(n):
        A[i, j] = random.random() - 0.5

    A[i, i] += n * 0.1

    b[i] = random.random() * 100

for i in range(100):
    iterate()
    print(f'{i}, residual={residual():0.10f}')

for i in range(n):
    lhs = 0.0
    for j in range(n):
        lhs += A[i, j] * x[j]
    assert abs(lhs - b[i]) < 1e-4
```

for such an equation:

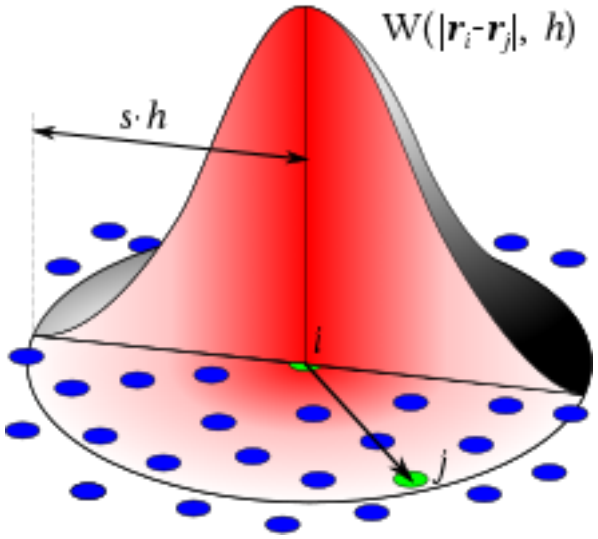
$$\left[\mathrm{I} - \beta \Delta t^2 \mathrm{M}^{-1} \frac{\partial \mathrm{f}}{\partial \mathrm{x}}(\mathrm{x}_t)\right] \mathrm{v}_{t+1} = \mathrm{v}_t \Delta t \mathrm{M}^{-1} \mathrm{f}(\mathrm{x}_t)$$

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

0.0.3 Smoothed particle hydrodynamics (SPH)

- **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. Derivaties: different story)

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



Wikipedia | MIT | 维基百科

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

Implenting SPH using th Equation of States (EOS) Also known as Weakly Compressible SPH (WCSPH). Momentum equation: (ρ: density; B: bulk modulus(体积模量); γ: constant, usually $\sim$ 7)

$$\begin{aligned} \frac{D\mathrm{v}}{Dt} &= -\frac{1}{\rho}\nabla p + g, & p &= B\left(\left(\frac{\rho}{\rho_0}\right)^\gamma - 1\right) \\ A(\mathrm{x}) &= \sum_i A_i \frac{m_i}{\rho_i} W(||\mathrm{x} - \mathrm{x}_j||_2, h), & \rho_i &= \sum_j m_j W(||\mathrm{x}_i - \mathrm{x}_j||_2, h), \end{aligned}$$

Note: the WCSPH paper should have used material derivatives.

Gradients in SPH

$$\begin{aligned} A(\mathrm{x}) &= \sum_i A_i \frac{m_i}{\rho_i} W(||\mathrm{x} - \mathrm{x}_j||_2, h) \\ \nabla A_i &= \rho_i \sum_j m_j \left(\frac{A_i}{\rho_i^2} + \frac{A_j}{\rho_j^2}\right) \nabla_{\mathrm{x}_i} W(||\mathrm{x}_i - \mathrm{x}_j||_2, h) \end{aligned}$$

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

SPH Simulation Cycle

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

$$\begin{aligned} \mathrm{v}_{t+1} &= \mathrm{v}_t + \Delta t \frac{D\mathrm{v}}{Dt} \\ \mathrm{x}_{t+1} &= \mathrm{x}_t + \Delta t \mathrm{v}_{t+1} \end{aligned}$$

Courant-Friedrichs-Levy (CFL) condition One upper bound of time step size:

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

- 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: 0.3~1
- FLIP fluid (smoke): 1~5+

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