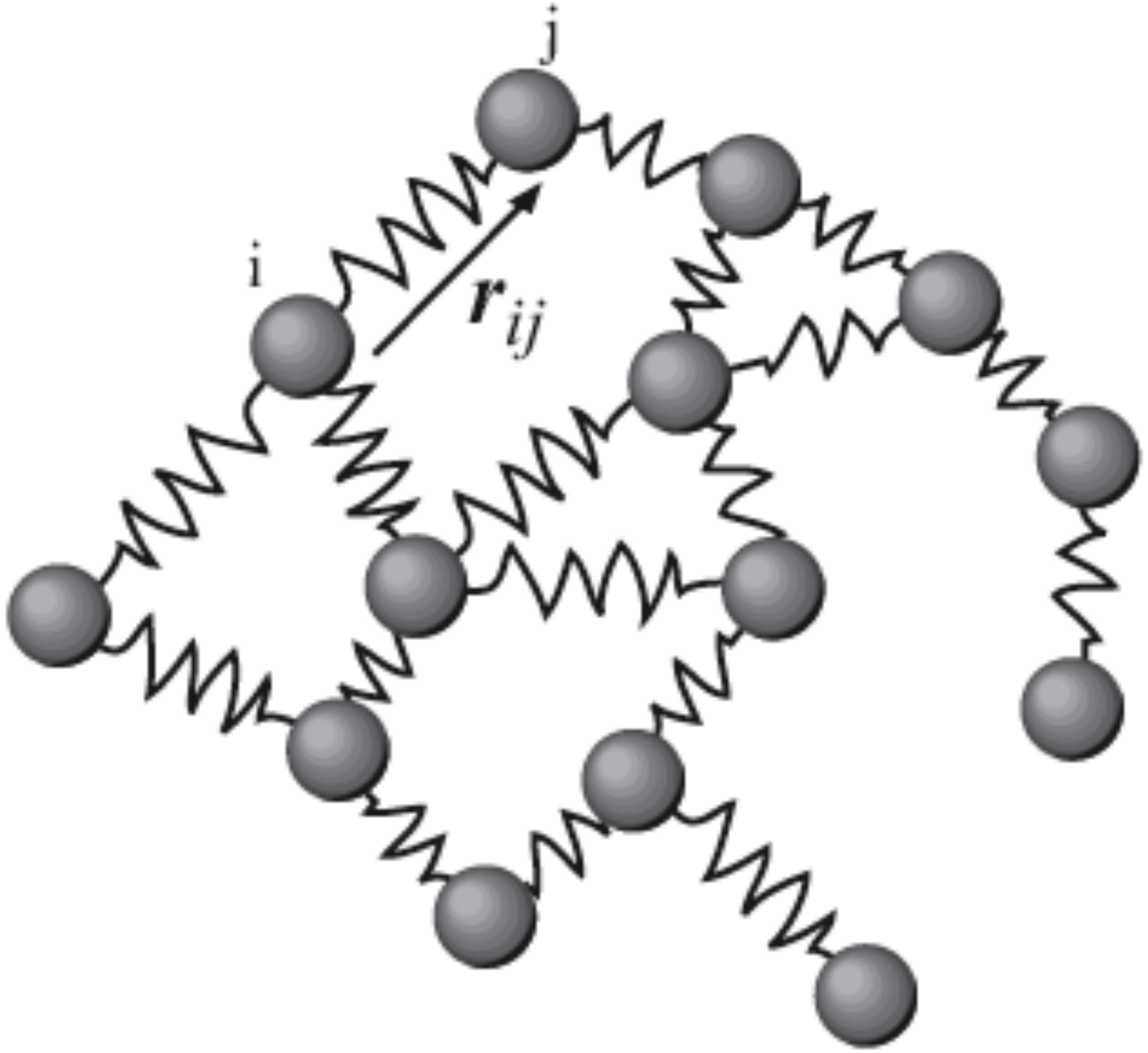
Instructor: Minchen Li

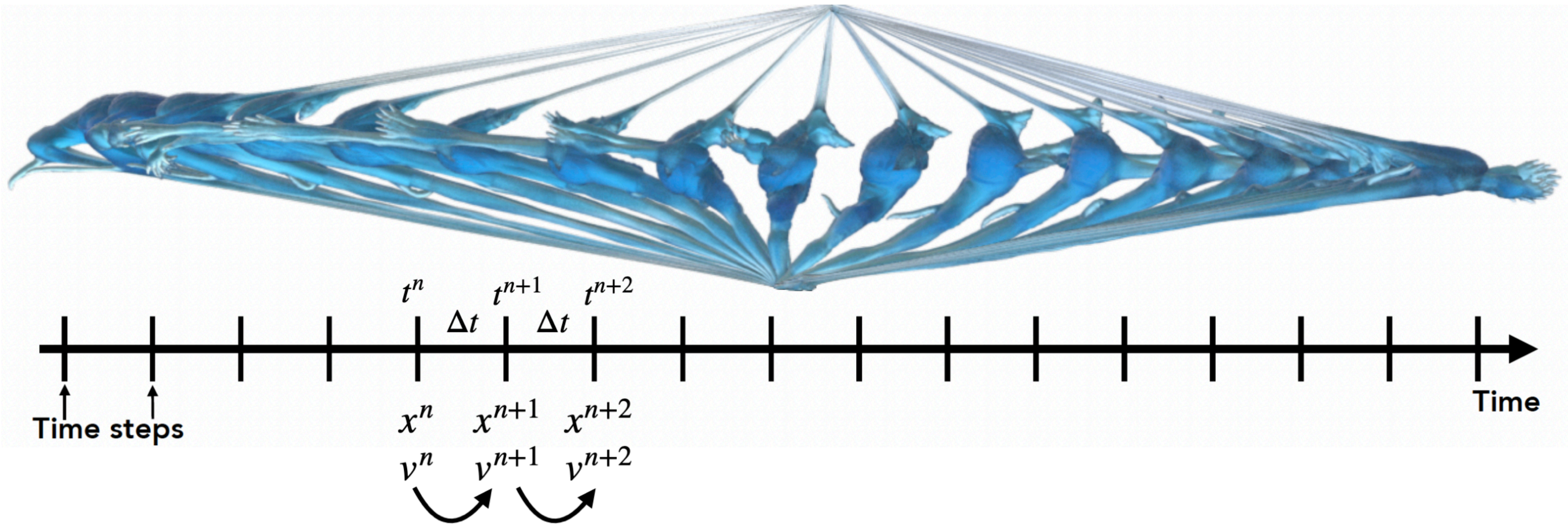


Lec 3: Case Study — Mass-Spring Systems

15-769: Physically-based Animation of Solids and Fluids (F23)

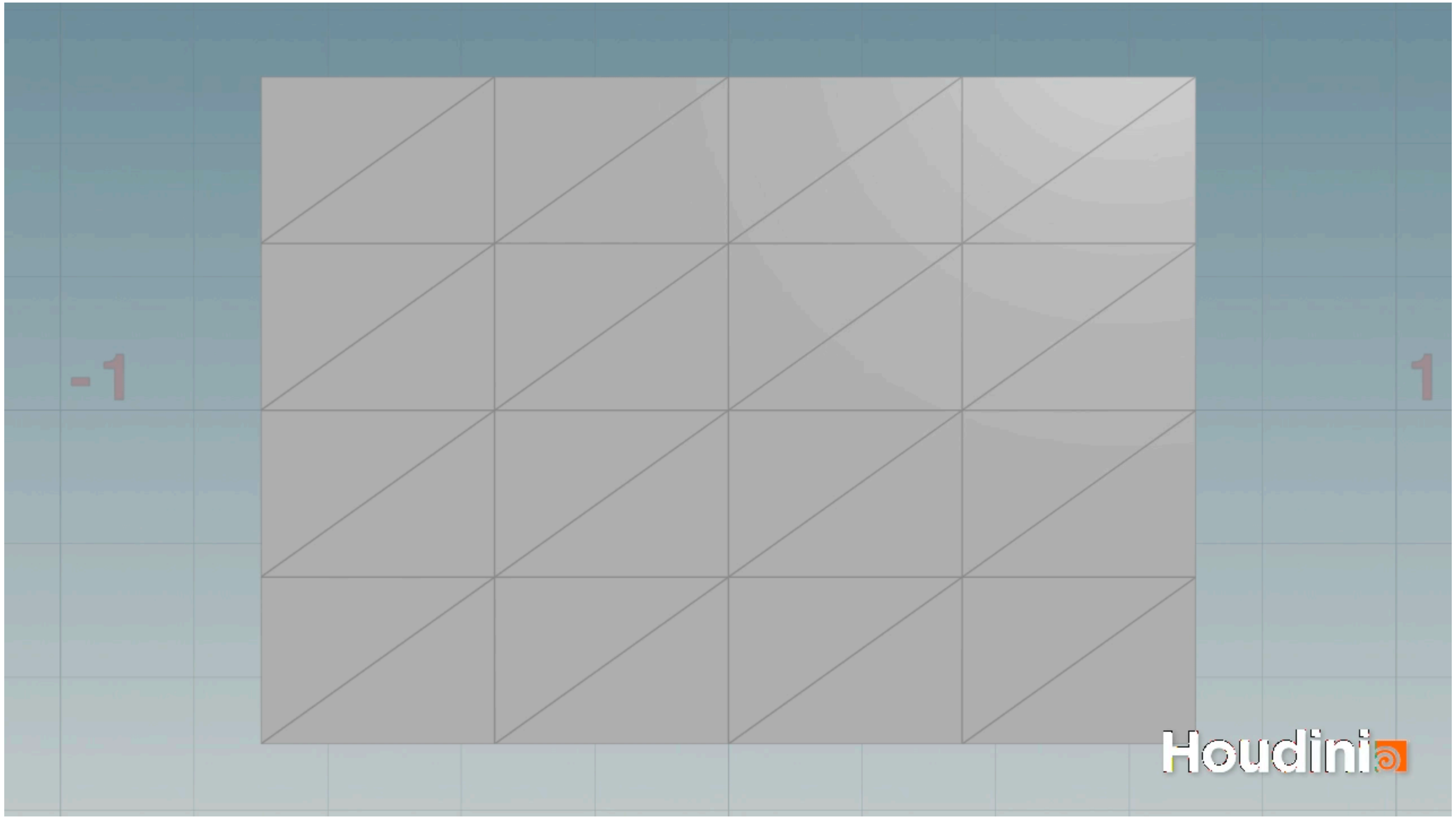
Recap: Time Integration

- Explicit time integration methods are conditionally stable
- Implicit Euler time integration is unconditionally stable
- Optimization-based implicit time integration guarantees solver convergence
  - Line search along descent directions



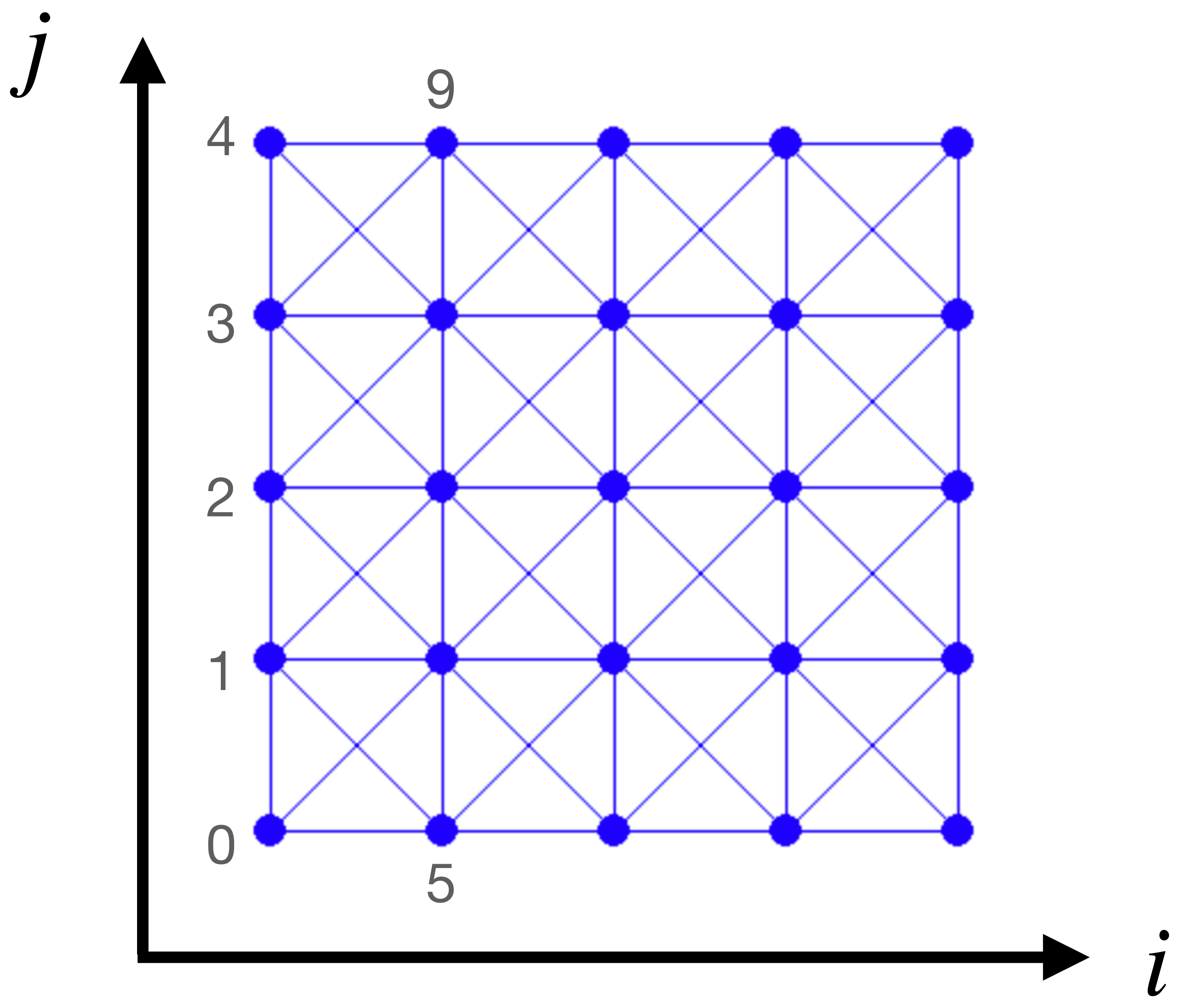
Today: Case Study — Mass-Spring Simulation

An Initially Stretched Elastic Square



Mass-Spring Representation of Solids

- Mass particles connected by springs
    - square_mesh.py




```
import numpy as np

def generate(side_length, n_seg):
    # sample nodes uniformly on a square
    x = np.array([[0.0, 0.0]] * ((n_seg + 1) ** 2))
    step = side_length / n_seg
    for i in range(0, n_seg + 1):
        for j in range(0, n_seg + 1):
            x[i * (n_seg + 1) + j] = [-side_length / 2 + i * step, -side_length / 2 + j * step]

    # connect the nodes with edges
    e = []
    # horizontal edges
    for i in range(0, n_seg):
        for j in range(0, n_seg + 1):
            e.append([i * (n_seg + 1) + j, (i + 1) * (n_seg + 1) + j])
    # vertical edges
    for i in range(0, n_seg + 1):
        for j in range(0, n_seg):
            e.append([i * (n_seg + 1) + j, i * (n_seg + 1) + j + 1])
    # diagonals
    for i in range(0, n_seg):
        for j in range(0, n_seg):
            e.append([i * (n_seg + 1) + j, (i + 1) * (n_seg + 1) + j + 1])
            e.append([(i + 1) * (n_seg + 1) + j, i * (n_seg + 1) + j + 1])

    return [x, e]
```

Time Integration

Optimization-based Implicit Euler

$$x^{n+1} = x^n + \Delta t v^{n+1},$$
$$v^{n+1} = v^n + \Delta t M^{-1} f^{n+1}$$

$$\Longleftrightarrow$$

Inertia term

Elasticity

$$E(x) = \frac{1}{2}\|x - (x^n + hv^n)\|_M^2 + h^2 P(x).$$

Incremental Potential

$$\frac{\partial P}{\partial x}(x) = -f(x)$$

Algorithm 3: Projected Newton Method for Backward Euler Time Integration

```
Result: x^{n+1}, v^{n+1}
x^i ← x^n;
do
  | P ← SPDProjection(∇²E(x^i));
  | p ← −P^{−1}∇E(x^i);
  | α ← BackTrackingLineSearch(x^i, p);  //
  | x^i ← x^i + αp;
while ||p||_∞/h > ε;
x^{n+1} ← x^i;
v^{n+1} ← (x^{n+1} − x^n)/Δt;
```

Energy Hessian: $\nabla^2 E(x^i)$

Energy Gradient: $\nabla E(x^i)$

Algorithm 2: Backtracking Line Search

```
Result: α
α ← 1;
while E(x^i + αp) > E(x^i) do
  └ α ← α/2;
```

Energy Value: $E(x^i + \alpha p)$

Incremental Potential

Inertia Term

with $\tilde{x}^n = x^n + hv^n$

$$E_I(x) = \frac{1}{2}\|x - \tilde{x}^n\|_M^2$$

$$\nabla E_I(x) = M(x - \tilde{x}^n)$$

$$\nabla^2 E_I(x) = M$$ — SPD

InertiaEnergy.py

```
import numpy as np

def val(x, x_tilde, m):
    sum = 0.0
    for i in range(0, len(x)):
        diff = x[i] - x_tilde[i]
        sum += 0.5 * m[i] * diff.dot(diff)
    return sum

def grad(x, x_tilde, m):
    g = np.array([[0.0, 0.0]] * len(x))
    for i in range(0, len(x)):
        g[i] = m[i] * (x[i] - x_tilde[i])
    return g

def hess(x, x_tilde, m):
    IJV = [[0] * (len(x) * 2), [0] * (len(x) * 2), np.array
    ([0.0] * (len(x) * 2))]
    for i in range(0, len(x)):
        for d in range(0, 2):
            IJV[0][i * 2 + d] = i * 2 + d
            IJV[1][i * 2 + d] = i * 2 + d
            IJV[2][i * 2 + d] = m[i]
    return IJV
```

Incremental Potential

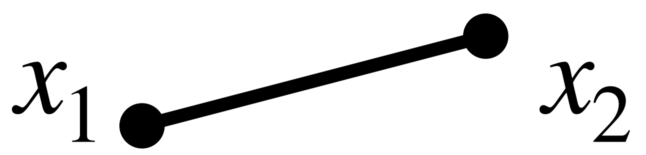
Mass-Spring Elasticity Energy

- Hooke's Law in 1D:
  - $E = \frac{1}{2}k(\Delta x)^2$

    Spring stiffness: k

    Spring displacement: Δx

- In higher dimensions:



  - $\frac{1}{2}k(\|x_1 - x_2\| - l)^2$ or $l^2\frac{1}{2}k(\frac{\|x_1 - x_2\|}{l} - 1)^2$

    Current length: $\|x_1 - x_2\|$

    Rest length: l

    A strain measure: $\frac{\|x_1 - x_2\|}{l}$

- To avoid computing square root, we define

$$P_e(x) = l^2\frac{1}{2}k(\frac{\|\boldsymbol{x}_1 - \boldsymbol{x}_2\|^2}{l^2} - 1)^2$$

Area weighting: l^2

Elasticity energy density (elasticity energy per unit area): $\frac{1}{2}k(\frac{\|\boldsymbol{x}_1 - \boldsymbol{x}_2\|^2}{l^2} - 1)^2$

Continuous setting:

$$P = \int_{\Omega^0} \Psi dX$$

Incremental Potential

Mass-Spring Elasticity Energy Gradient and Hessian

$$P_e(x) = l^2 \frac{1}{2} k \left(\frac{\|\boldsymbol{x}_1 - \boldsymbol{x}_2\|^2}{l^2} - 1 \right)^2$$

$$\frac{\partial P_e}{\partial \boldsymbol{x}_1}(x) = -\frac{\partial P_e}{\partial \boldsymbol{x}_2}(x) = 2k \left(\frac{\|\boldsymbol{x}_1 - \boldsymbol{x}_2\|^2}{l^2} - 1 \right)(\boldsymbol{x}_1 - \boldsymbol{x}_2)$$

$$\begin{aligned}\frac{\partial^2 P_e}{\partial \boldsymbol{x}_1^2}(x) = \frac{\partial^2 P_e}{\partial \boldsymbol{x}_2^2}(x) &= -\frac{\partial^2 P_e}{\partial \boldsymbol{x}_1 \boldsymbol{x}_2}(x) = -\frac{\partial^2 P_e}{\partial \boldsymbol{x}_2 \boldsymbol{x}_1}(x) \\ &= \frac{4k}{l^2}(\boldsymbol{x}_1 - \boldsymbol{x}_2)(\boldsymbol{x}_1 - \boldsymbol{x}_2)^T + 2k \left(\frac{\|\boldsymbol{x}_1 - \boldsymbol{x}_2\|^2}{l^2} - 1 \right) \boldsymbol{I} \\ &= \frac{2k}{l^2} \left(2(\boldsymbol{x}_1 - \boldsymbol{x}_2)(\boldsymbol{x}_1 - \boldsymbol{x}_2)^T + (\|\boldsymbol{x}_1 - \boldsymbol{x}_2\|^2 - l^2) \boldsymbol{I} \right)\end{aligned}$$

MassSpringEnergy.py

```python
import numpy as np
import utils

def val(x, e, l2, k):
    sum = 0.0
    for i in range(0, len(e)):
        diff = x[e[i][0]] - x[e[i][1]]
        sum += l2[i] * 0.5 * k[i] * (diff.dot(diff) / l2[i] - 1) ** 2
    return sum
```

```python
def grad(x, e, l2, k):
    g = np.array([[0.0, 0.0]] * len(x))
    for i in range(0, len(e)):
        diff = x[e[i][0]] - x[e[i][1]]
        g_diff = 2 * k[i] * (diff.dot(diff) / l2[i] - 1) * diff
        g[e[i][0]] += g_diff
        g[e[i][1]] -= g_diff
    return g
```

Incremental Potential

Mass-Spring Elasticity Energy Hessian Implementation

$$
\begin{aligned}
\frac{\partial^2 P_e}{\partial \boldsymbol{x}_1^2}(x) = \frac{\partial^2 P_e}{\partial \boldsymbol{x}_2^2}(x) = -\frac{\partial^2 P_e}{\partial \boldsymbol{x}_1 \boldsymbol{x}_2}(x) = -\frac{\partial^2 P_e}{\partial \boldsymbol{x}_2 \boldsymbol{x}_1}(x) \\
= \frac{4k}{l^2}(\boldsymbol{x}_1 - \boldsymbol{x}_2)(\boldsymbol{x}_1 - \boldsymbol{x}_2)^T + 2k(\frac{\|\boldsymbol{x}_1 - \boldsymbol{x}_2\|^2}{l^2} - 1)\boldsymbol{I} \\
= \frac{2k}{l^2}(2(\boldsymbol{x}_1 - \boldsymbol{x}_2)(\boldsymbol{x}_1 - \boldsymbol{x}_2)^T + (\|\boldsymbol{x}_1 - \boldsymbol{x}_2\|^2 - l^2)\boldsymbol{I})
\end{aligned}
$$

MassSpringEnergy.py

```python
def hess(x, e, l2, k):
    IJV = [[0] * (len(e) * 16), [0] * (len(e) * 16), np.array([0.0] * (len(e) * 16))]
    for i in range(0, len(e)):
        diff = x[e[i][0]] - x[e[i][1]]
        H_diff = 2 * k[i] / l2[i] * (2 * np.outer(diff, diff) + (diff.dot(diff) - l2[i]) * np.identity(2))
        H_local = utils.make_PD(np.block([[H_diff, -H_diff], [-H_diff, H_diff]]))
        # add to global matrix
        for nI in range(0, 2):
            for nJ in range(0, 2):
                indStart = i * 16 + (nI * 2 + nJ) * 4
                for r in range(0, 2):
                    for c in range(0, 2):
                        IJV[0][indStart + r * 2 + c] = e[i][nI] * 2 + r
                        IJV[1][indStart + r * 2 + c] = e[i][nJ] * 2 + c
                        IJV[2][indStart + r * 2 + c] = H_local[nI * 2 + r, nJ * 2 + c]
    return IJV
```

Incremental Potential

Mass-Spring Elasticity Energy Hessian Projection (make_PSD)

$$\min_P \|P - \nabla^2 E(x^i)\|_F \quad s.t. \quad v^T P v \geq 0 \quad \forall v \neq 0$$

Solution: $\hat{A} = Q\hat{\Lambda}Q^{-1}, \quad \hat{\Lambda}_{ij} = \Lambda_{ij} > 0 \; ? \; \Lambda_{ij} : 0$

Definition (Eigendecomposition)**.** The eigendecomposition of a square matrix $A \in R^{n \times n}$ is

$$A = Q\Lambda Q^{-1}$$

where $Q = [q_1, q_2, ..., q_n]$ is composed of the eigenvectors q_i of A, $\|q_i\| = 1$; $\Lambda = [\lambda_1, \lambda_2, ..., \lambda_n]$, $\lambda_1 \geq \lambda_2 \geq ..., \lambda_n$ are the eigenvalues of A; and $Aq_i = \lambda_i q_i$.

utils.py

```python
import numpy as np
import numpy.linalg as LA

def make_PD(hess):
    [lam, V] = LA.eigh(hess)    # Eigen decomposition on symmetric matrix
    # set all negative Eigenvalues to 0
    for i in range(0, len(lam)):
        lam[i] = max(0, lam[i])
    return np.matmul(np.matmul(V, np.diag(lam)), np.transpose(V))
```

Incremental Potential

Gradient and Hessian

time_integrator.py

```python
def IP_val(x, e, x_tilde, m, l2, k, h):
    return InertiaEnergy.val(x, x_tilde, m) + h * h * MassSpringEnergy.val(x, e, l2, k)     # implicit Euler

def IP_grad(x, e, x_tilde, m, l2, k, h):
    return InertiaEnergy.grad(x, x_tilde, m) + h * h * MassSpringEnergy.grad(x, e, l2, k)   # implicit Euler

def IP_hess(x, e, x_tilde, m, l2, k, h):
    IJV_In = InertiaEnergy.hess(x, x_tilde, m)
    IJV_MS = MassSpringEnergy.hess(x, e, l2, k)
    IJV_MS[2] *= h * h    # implicit Euler
    IJV = np.append(IJV_In, IJV_MS, axis=1)
    H = sparse.coo_matrix((IJV[2], (IJV[0], IJV[1])), shape=(len(x) * 2, len(x) * 2)).tocsr()
    return H
```

Time Integration

Algorithm 3: Projected Newton Method for Backward Euler Time Integration

Result: x^{n+1}, v^{n+1}

$x^i \leftarrow x^n$;
do
  $P \leftarrow \text{SPDProjection}(\nabla^2 E(x^i))$;
  $p \leftarrow -P^{-1}\nabla E(x^i)$;
  $\alpha \leftarrow \text{BackTrackingLineSearch}(x^i, p)$; //
  $x^i \leftarrow x^i + \alpha p$;
while $\|p\|_\infty / h > \epsilon$;
$x^{n+1} \leftarrow x^i$;
$v^{n+1} \leftarrow (x^{n+1} - x^n)/\Delta t$;

Algorithm 2: Backtracking Line Search

Result: α

$\alpha \leftarrow 1$;
while $E(x^i + \alpha p) > E(x^i)$ **do**
  $\alpha \leftarrow \alpha/2$;

```python
import copy
from cmath import inf

import numpy as np
import numpy.linalg as LA
import scipy.sparse as sparse
from scipy.sparse.linalg import spsolve

import InertiaEnergy
import MassSpringEnergy
```

time_integrator.py

```python
def step_forward(x, e, v, m, l2, k, h, tol):
    x_tilde = x + v * h     # implicit Euler predictive position
    x_n = copy.deepcopy(x)

    # Newton loop
    iter = 0
    E_last = IP_val(x, e, x_tilde, m, l2, k, h)
    p = search_dir(x, e, x_tilde, m, l2, k, h)
    while LA.norm(p, inf) / h > tol:
        print('Iteration', iter, ':')
        print('residual =', LA.norm(p, inf) / h)

        # line search
        alpha = 1
        while IP_val(x + alpha * p, e, x_tilde, m, l2, k, h) > E_last:
            alpha /= 2
        print('step size =', alpha)

        x += alpha * p
        E_last = IP_val(x, e, x_tilde, m, l2, k, h)
        p = search_dir(x, e, x_tilde, m, l2, k, h)
        iter += 1

    v = (x - x_n) / h   # implicit Euler velocity update
    return [x, v]
```

```python
def search_dir(x, e, x_tilde, m, l2, k, h):
    projected_hess = IP_hess(x, e, x_tilde, m, l2, k, h)
    reshaped_grad = IP_grad(x, e, x_tilde, m, l2, k, h).reshape(len(x) * 2, 1)
    return spsolve(projected_hess, -reshaped_grad).reshape(len(x), 2)
```

Simulator with Visualization

Simulator.py

```python
# Mass-Spring Solids Simulation

import numpy as np  # numpy for linear algebra
import pygame       # pygame for visualization
pygame.init()

import square_mesh   # square mesh
import time_integrator

# simulation setup
side_len = 1
rho = 1000  # density of square
k = 1e5     # spring stiffness
initial_stretch = 1.4
n_seg = 4   # num of segments per side of the square
h = 0.004   # time step size in s

# initialize simulation
[x, e] = square_mesh.generate(side_len, n_seg)  # node positions and edge node indices
v = np.array([[0.0, 0.0]] * len(x))                        # velocity
m = [rho * side_len * side_len / ((n_seg + 1) * (n_seg + 1))] * len(x)  # calculate node mass evenly
# rest length squared
l2 = []
for i in range(0, len(e)):
    diff = x[e[i][0]] - x[e[i][1]]
    l2.append(diff.dot(diff))
k = [k] * len(e)    # spring stiffness
# apply initial stretch horizontally
for i in range(0, len(x)):
    x[i][0] *= initial_stretch

# simulation with visualization
resolution = np.array([900, 900])
offset = resolution / 2
scale = 200
def screen_projection(x):
    return [offset[0] + scale * x[0], resolution[1] - (offset[1] + scale * x[1])]

time_step = 0
screen = pygame.display.set_mode(resolution)
running = True
while running:
    # run until the user asks to quit
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    print('### Time step', time_step, '###')

    # fill the background and draw the square
    screen.fill((255, 255, 255))
    for eI in e:
        pygame.draw.aaline(screen, (0, 0, 255), screen_projection(x[eI[0]]), screen_projection(x[eI[1]]))
    for xI in x:
        pygame.draw.circle(screen, (0, 0, 255), screen_projection(xI), 0.1 * side_len / n_seg * scale)

    pygame.display.flip()   # flip the display

    # step forward simulation and wait for screen refresh
    [x, v] = time_integrator.step_forward(x, e, v, m, l2, k, h, 1e-2)
    time_step += 1
    pygame.time.wait(int(h * 1000))

pygame.quit()
```

Demo!

Code: github.com/liminchen/solid-sim-tutorial

Image Sources

- https://academic-accelerator.com/encyclopedia/spring-system