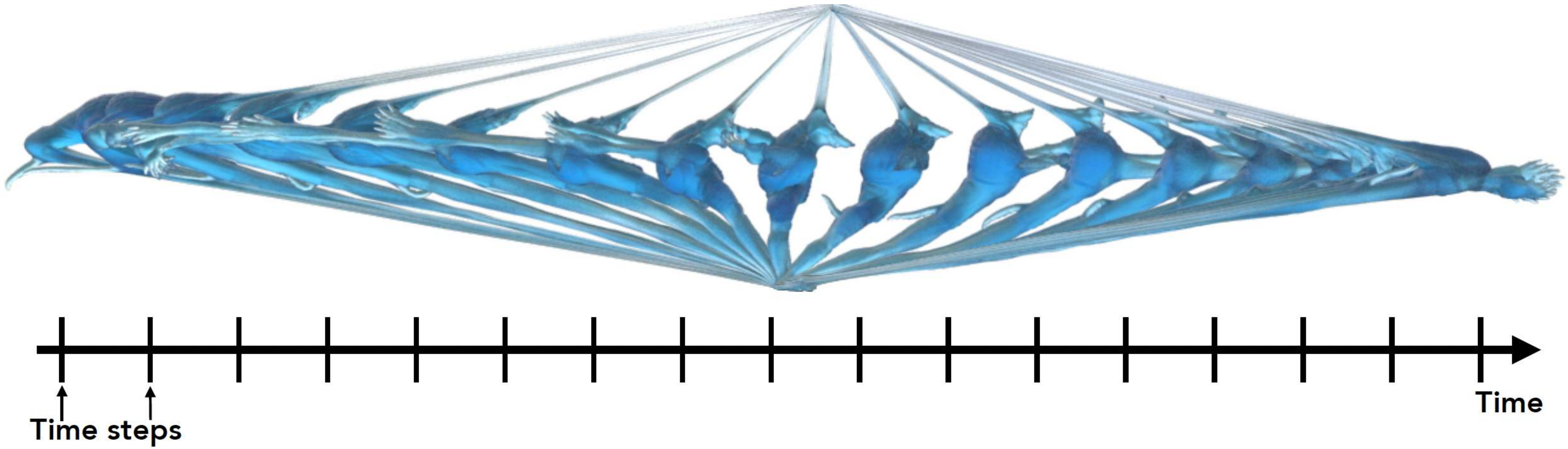
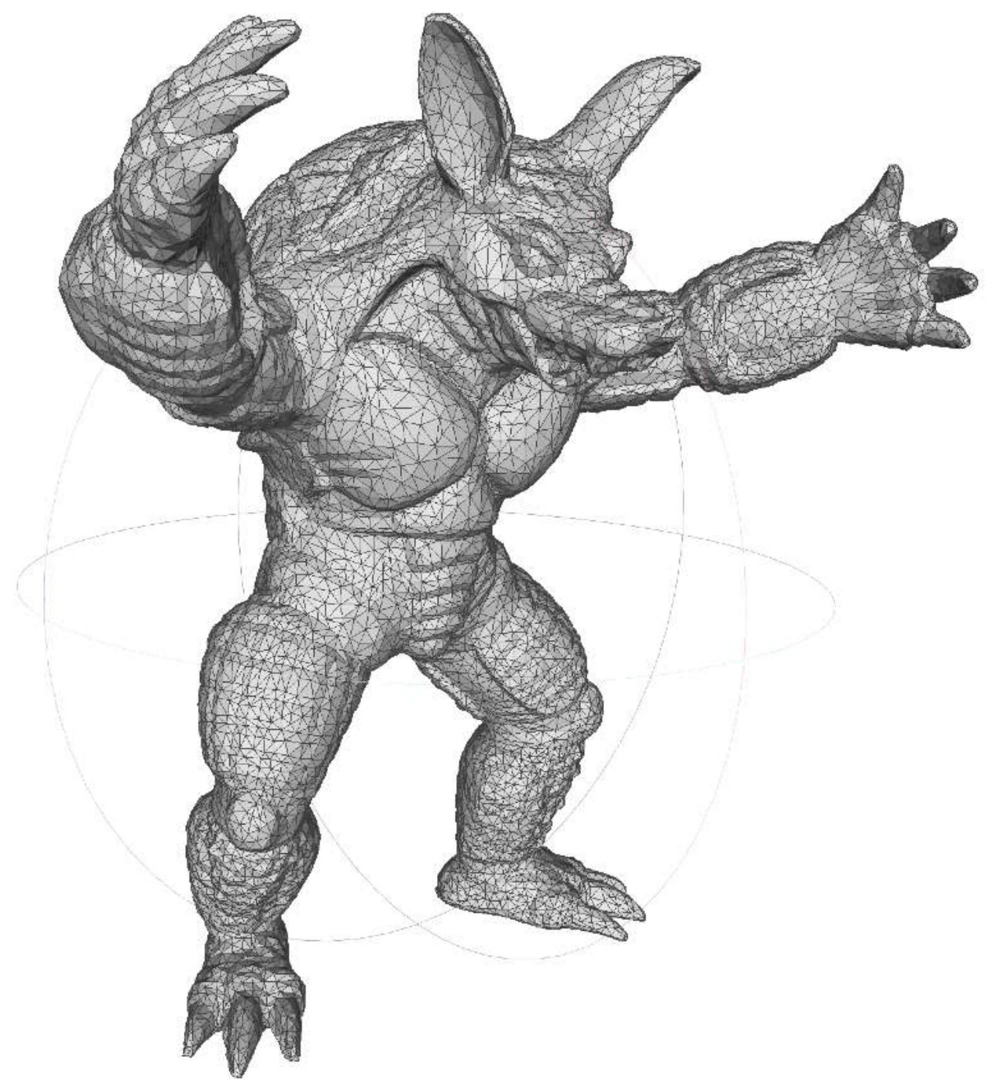


Reliable Simulation of Elastodynamics

15-362/662 (F24) Special Topics # 4

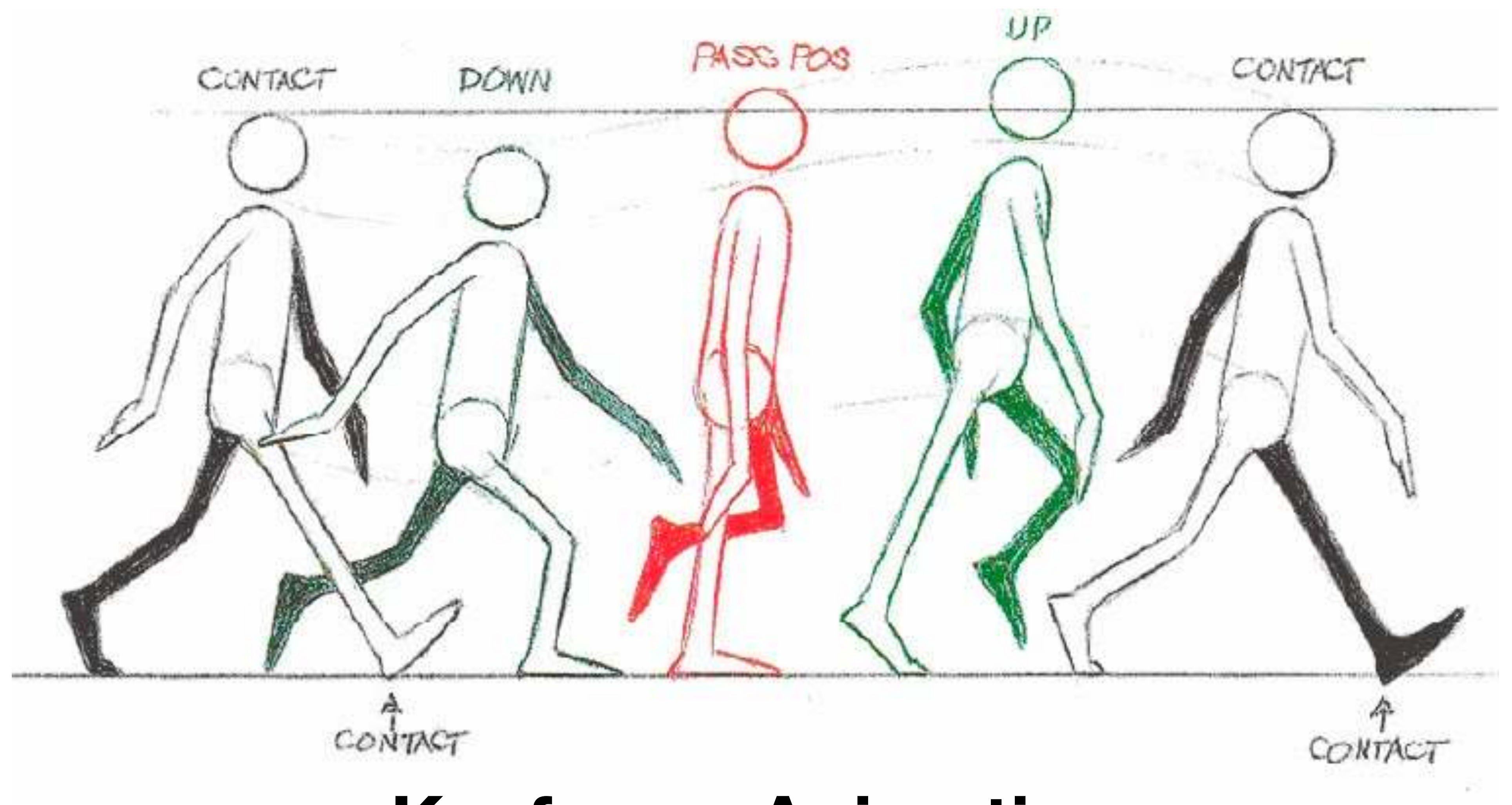
Computer Graphics:

Generating Realistic Visual Effects via Computing

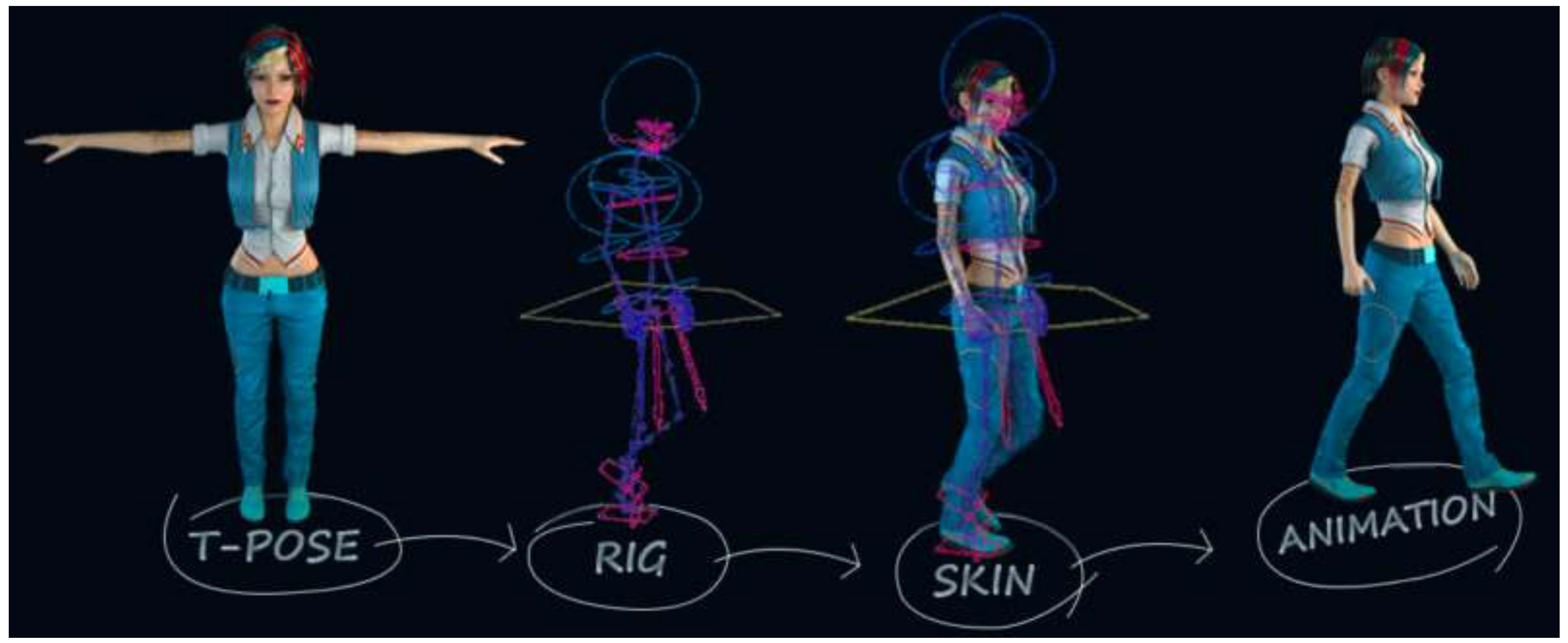


Geometry ⟶ Appearance ⟶ Animation

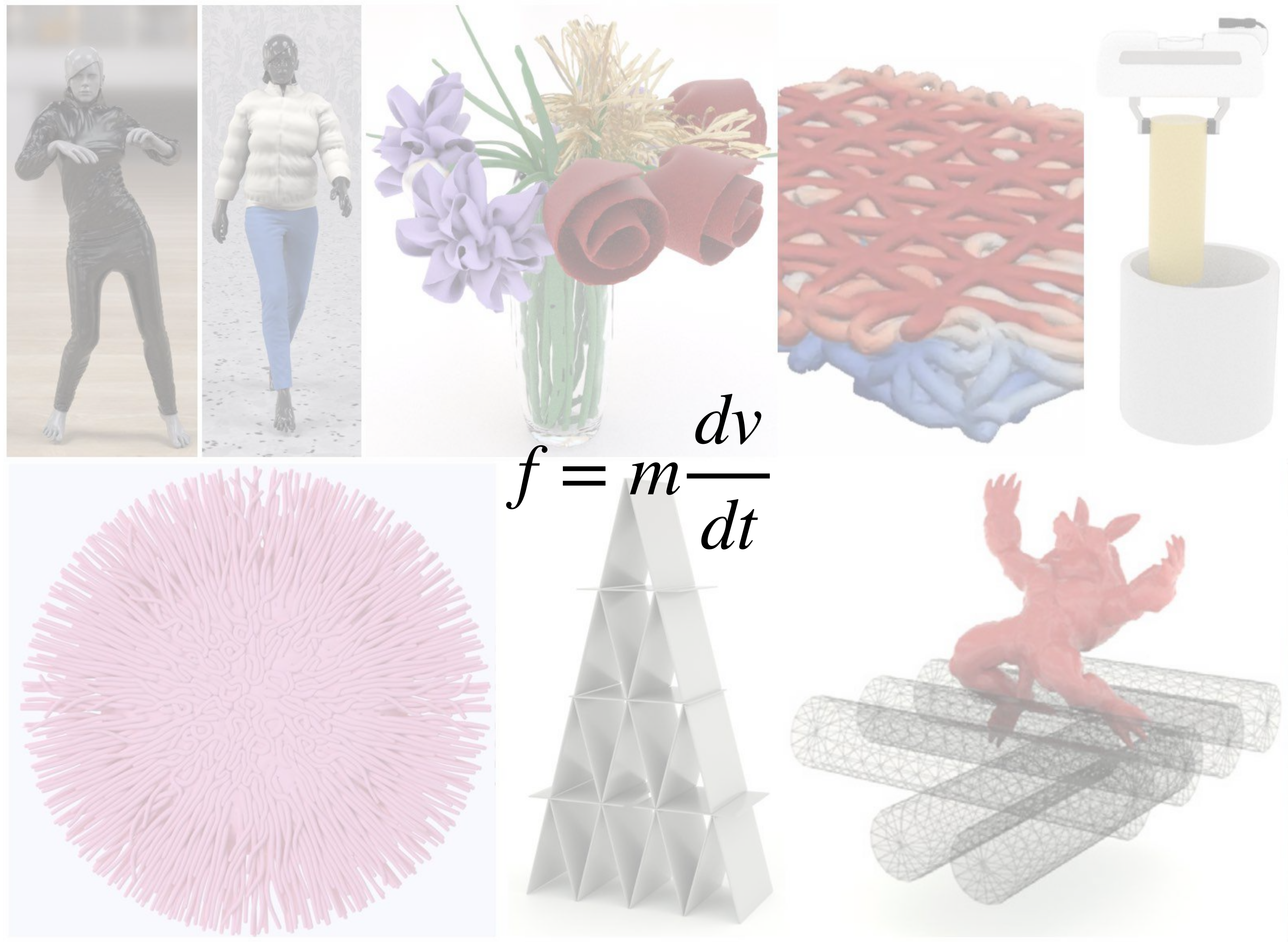
Animation



Keyframe Animation



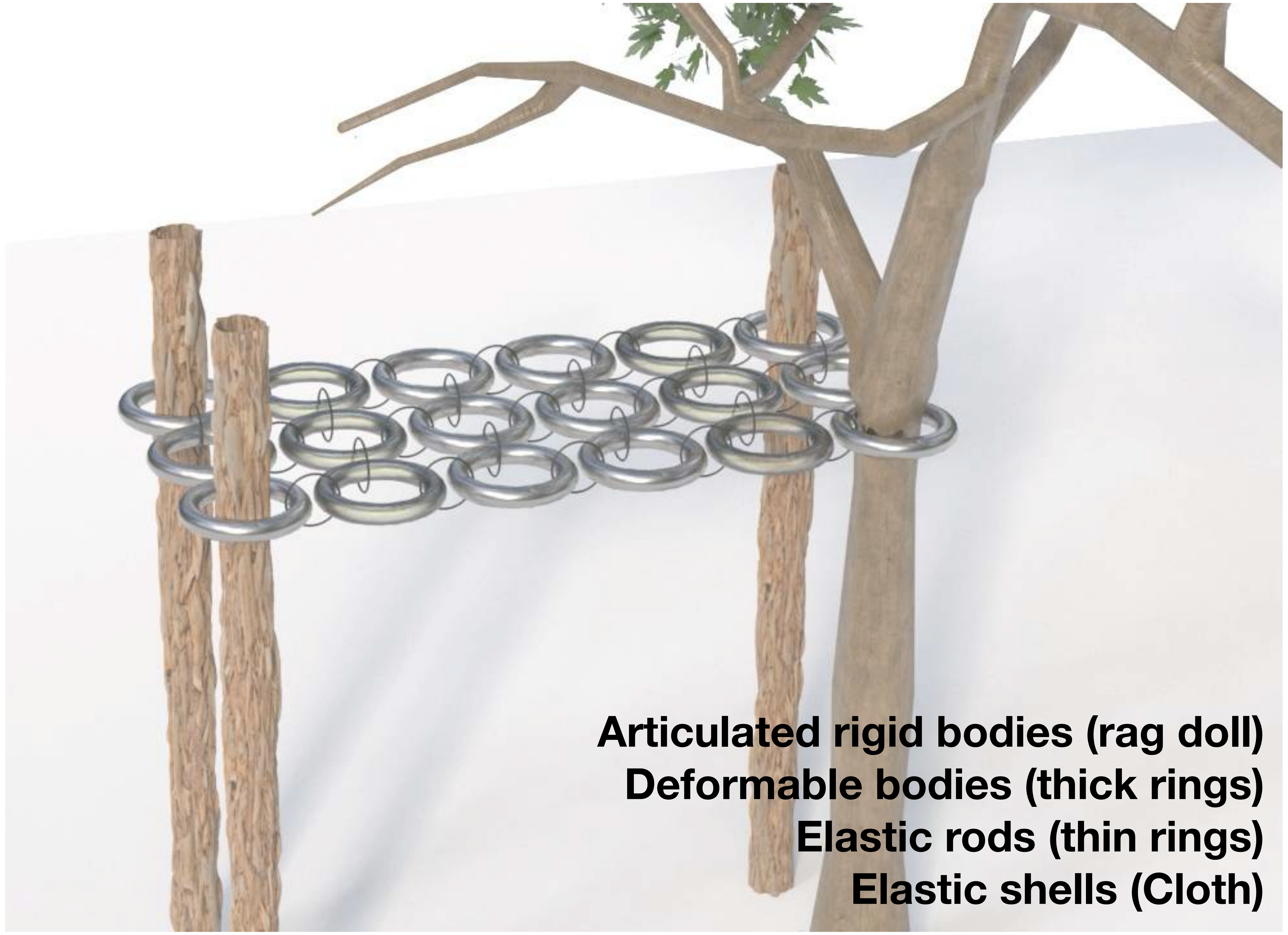
Skinning Animation



$$f = m\frac{dv}{dt}$$

Physics-based Animation

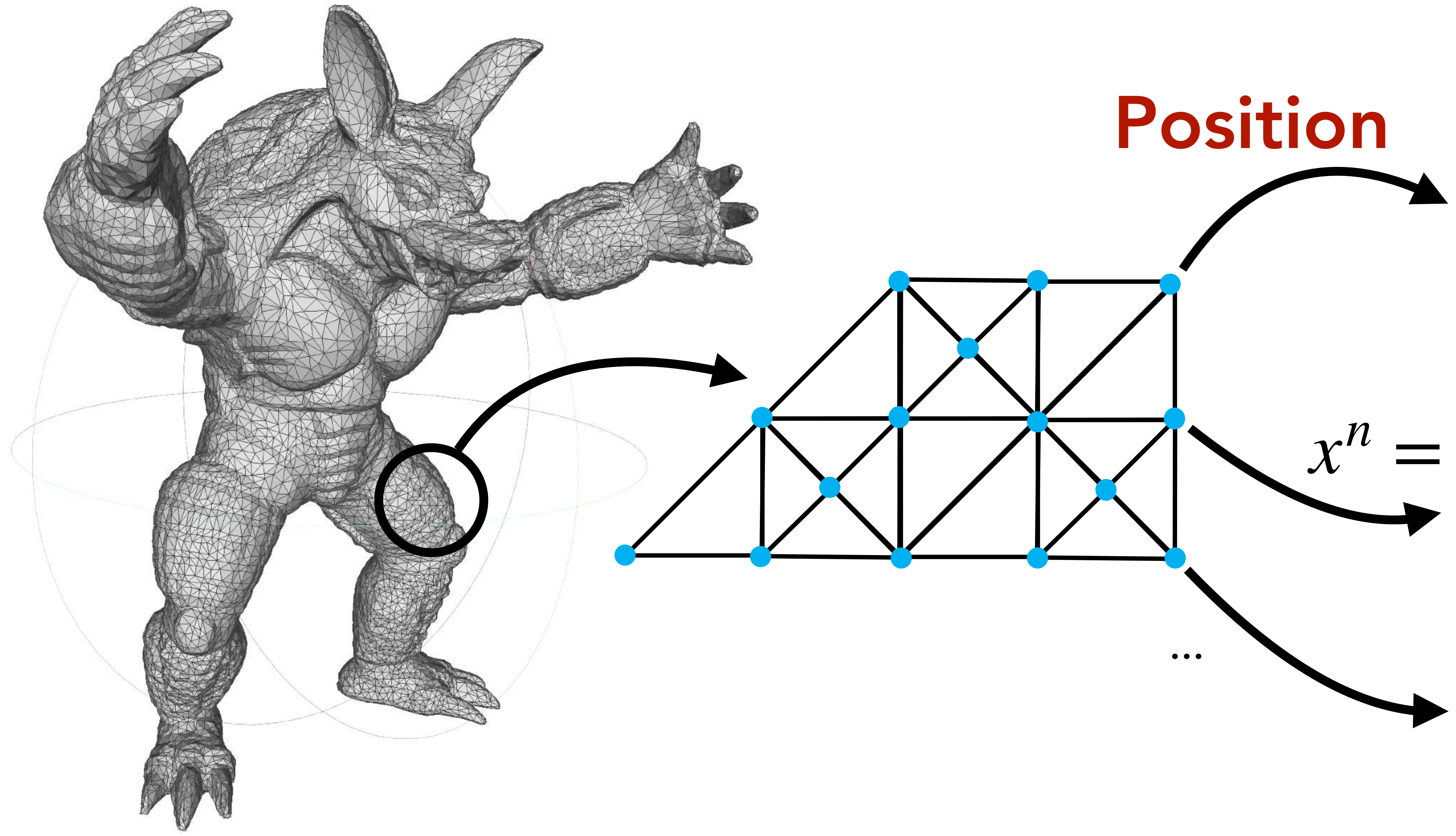
Physics-based Animation


Articulated rigid bodies (rag doll)
Deformable bodies (thick rings)
Elastic rods (thin rings)
Elastic shells (Cloth)

Today: Elasticity of Deformable Solids

A Reliable Simulation Approach based on Numerical Optimization

Spatial Discretization



Position

$$x^n = \begin{bmatrix} x^n_{0x} \\ x^n_{0y} \\ x^n_{0z} \\ x^n_{1x} \\ x^n_{1y} \\ x^n_{1z} \\ \vdots \end{bmatrix} \qquad v^n = \begin{bmatrix} v^n_{0x} \\ v^n_{0y} \\ v^n_{0z} \\ v^n_{1x} \\ v^n_{1y} \\ v^n_{1z} \\ \vdots \end{bmatrix}$$

Velocity

Governing Equation (Conservation of Momentum)

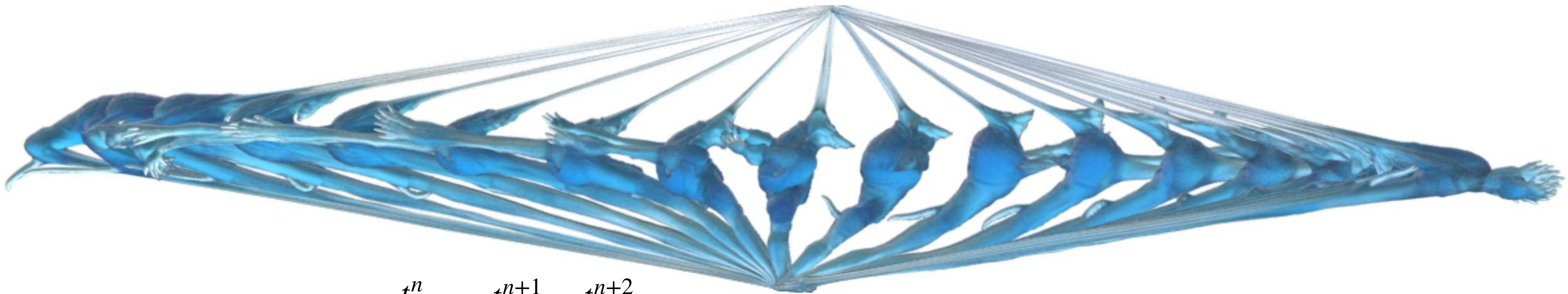
- The spatially discrete, temporally continuous form

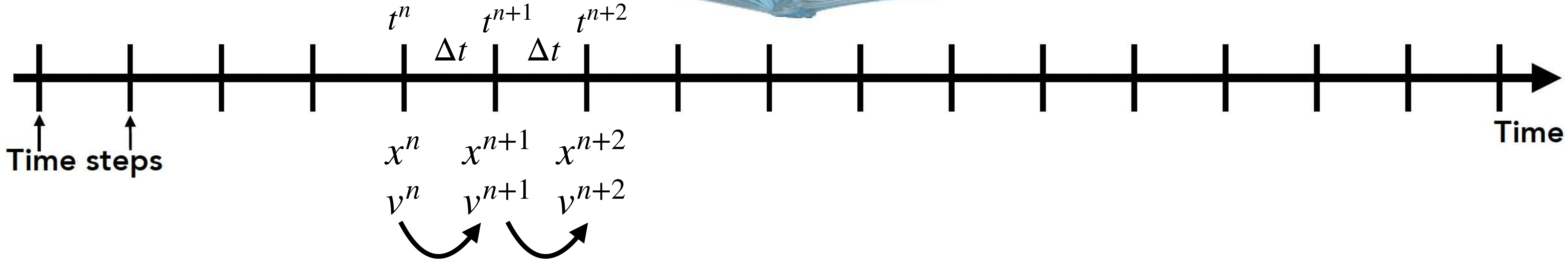
$$\begin{aligned} \frac{dx}{dt} &= v, \\ M\frac{dv}{dt} &= f. \end{aligned}$$

- Mass matrix (for now)

$$M = \begin{pmatrix} m_1 & & & \\ & m_1 & & \\ & & m_2 & \\ & & & m_2 \end{pmatrix}$$

Time Stepping (Time Integration)





Governing Equation (Temporally Discrete)

Forward Difference, Forward Euler

- Forward difference approximation on velocity and acceleration

$$\left(\frac{\mathbf{d}x}{\mathbf{d}t}\right)^n \approx \frac{x^{n+1}-x^n}{\Delta t} \qquad \left(\frac{\mathbf{d}v}{\mathbf{d}t}\right)^n \approx \frac{v^{n+1}-v^n}{\Delta t}$$

$$\left(f(t^n + \Delta t) = f(t^n) + \frac{\mathbf{d}f}{\mathbf{d}t}(t^n)\Delta t + O(\Delta t^2) \right)$$

Taylor's expansion

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

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

Newton’s 2nd Law (Temporally Discrete)

Forward and Backward Difference, Symplectic Euler

- Forward difference on acceleration, backward difference on velocity

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

Newton's 2nd Law (Temporally Discrete)

Backward Difference, Backward Euler (or Implicit Euler)

- Backward difference approximation on velocity and acceleration

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

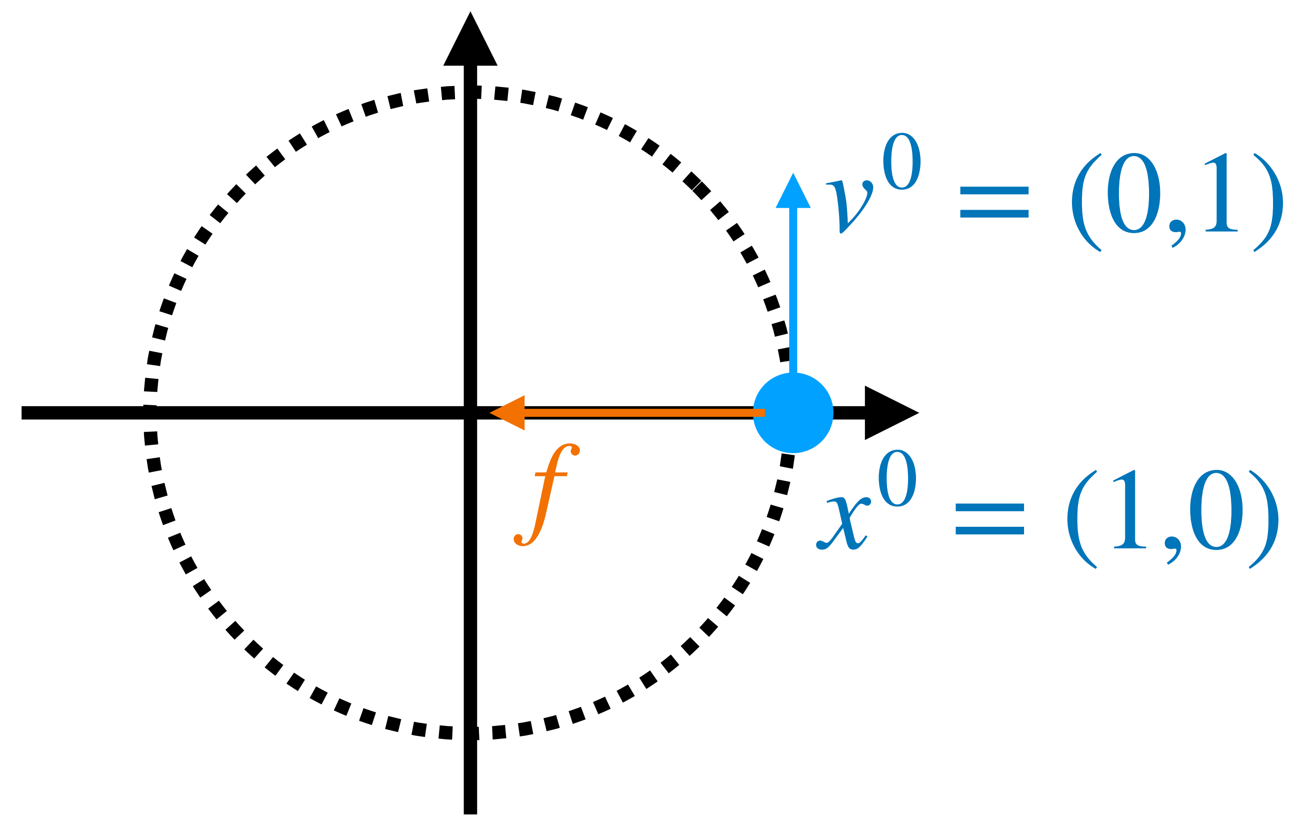
$$f^{n+1} = f(x^{n+1})$$

Needs to solve a system of equations:

$$M(x^{n+1} - (x^n + \Delta t v^n)) - \Delta t^2 f(x^{n+1}) = 0.$$

Stability of Forward, Symplectic, and Backward Euler

Example on a uniform circular motion

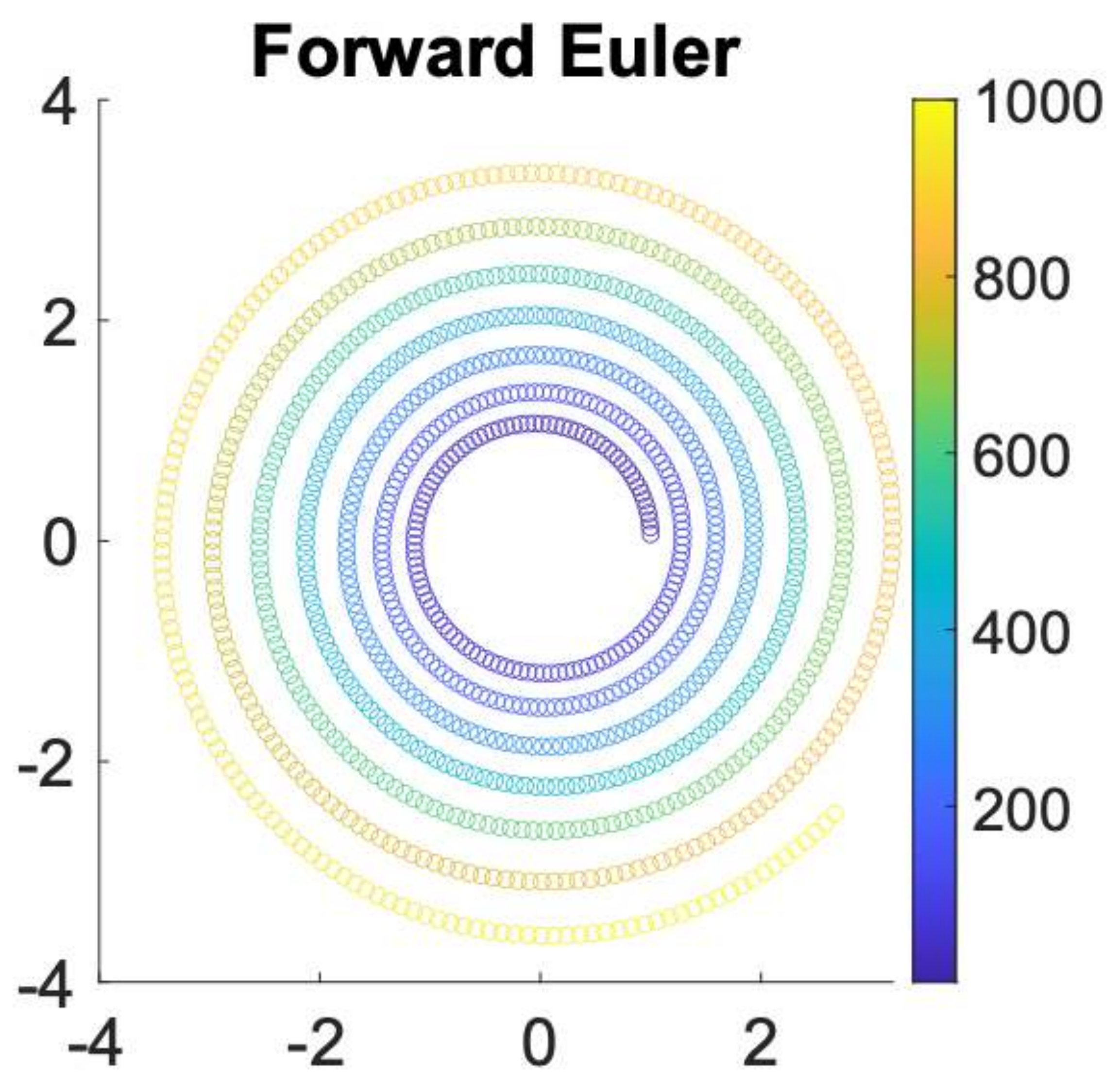


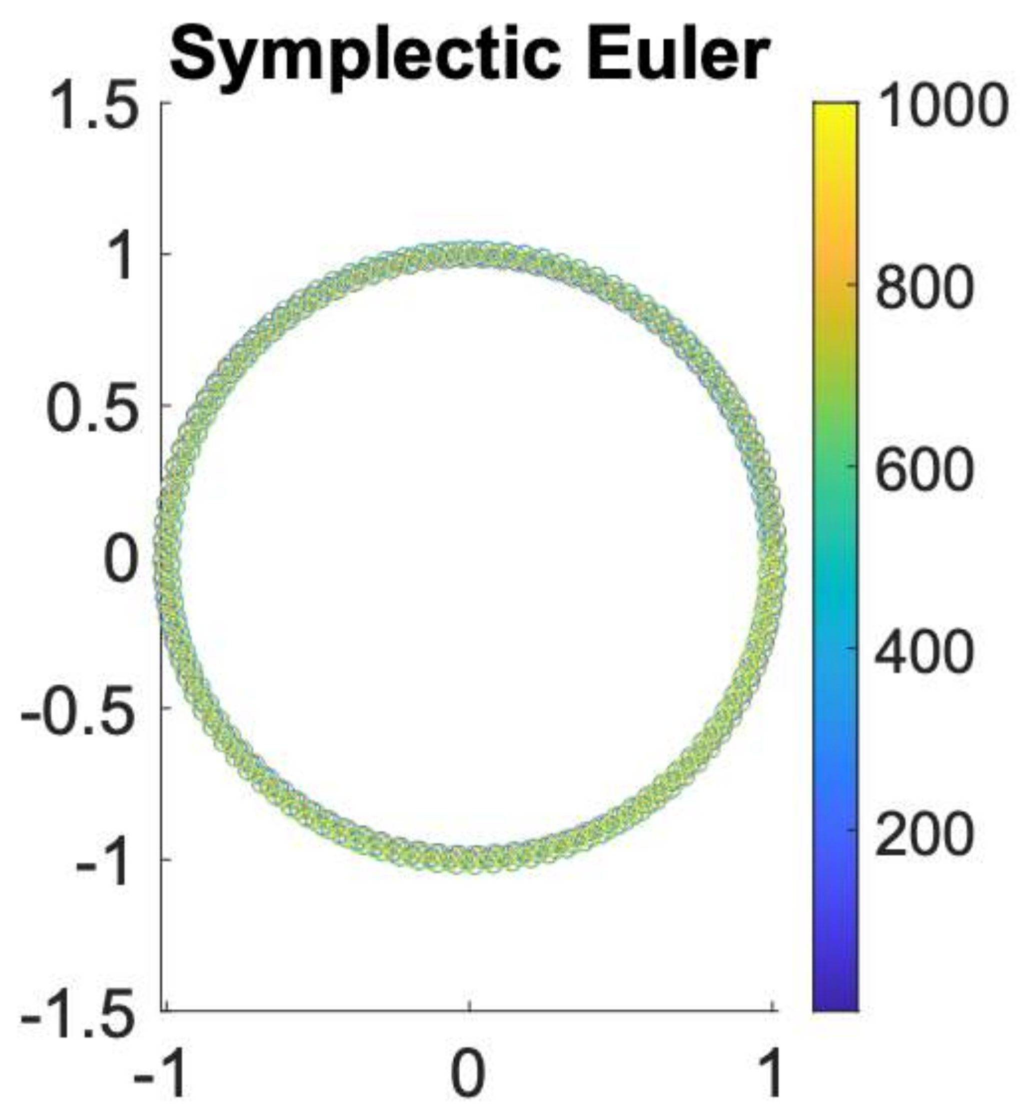
Problem Setup

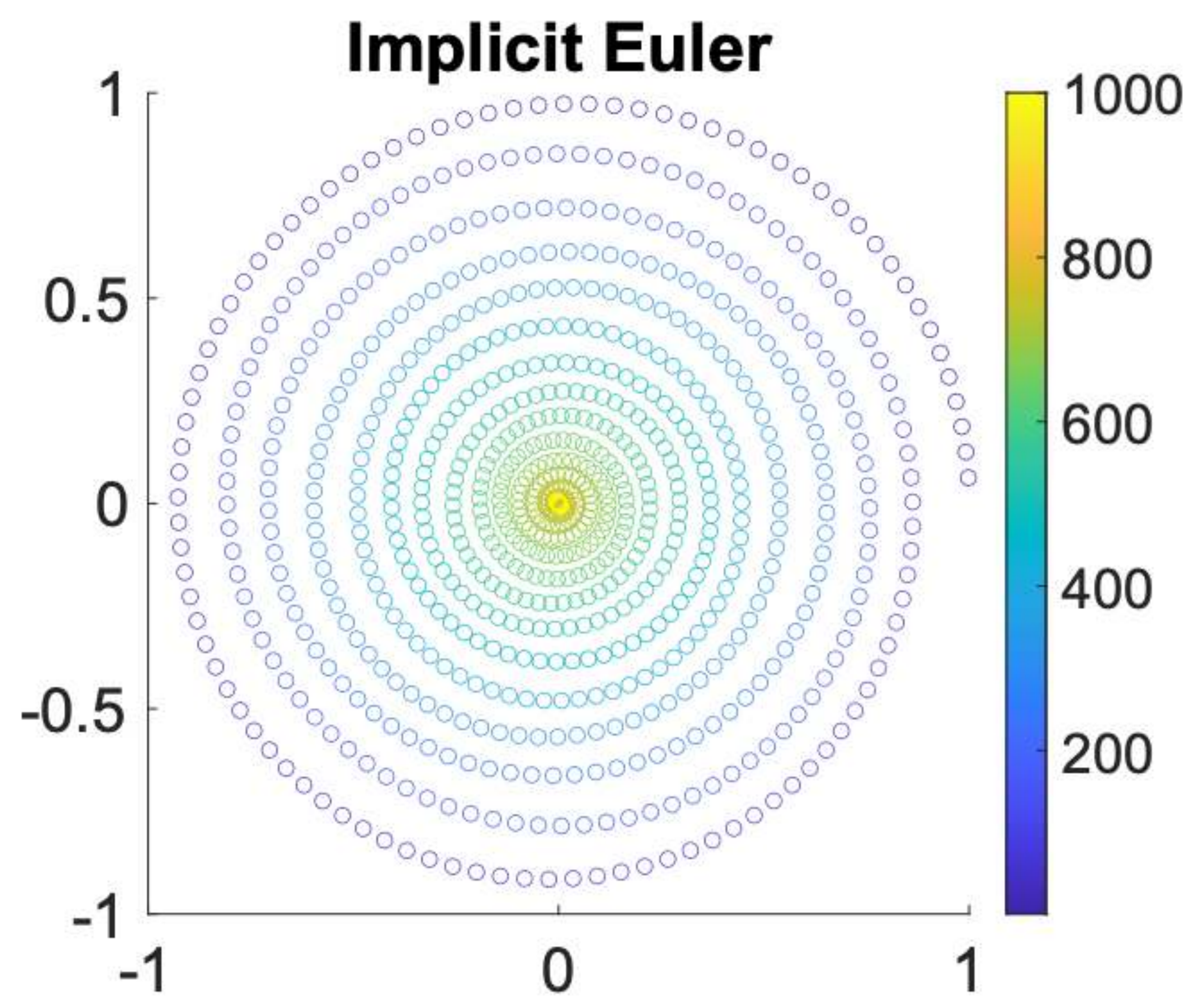
$$x^{n+1} = x^n + \Delta t v^n, \quad v^{n+1} = v^n + \Delta t M^{-1} f^n.$$

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

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







Newton's Method for Backward Euler

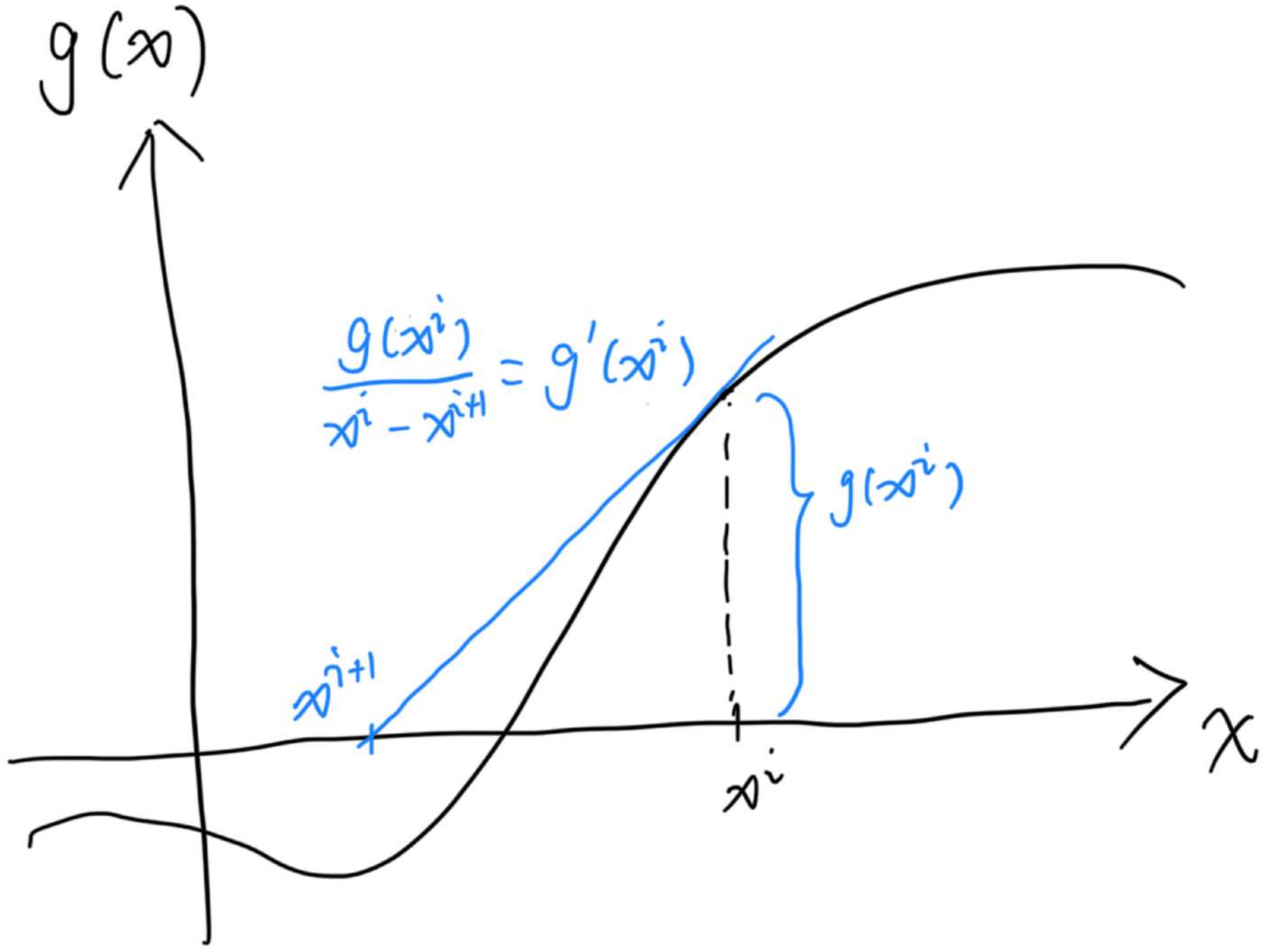
Formulation

Let $g(x) = M(x - (x^n + \Delta t v^n)) - \Delta t^2 f(x)$

We want to solve $g(x) = 0$

Newton's method in 1D:

- **Start from initial guess** x^0
- **For each iteration (until convergence)**
  - $x^{i+1} \leftarrow x^i - g(x^i)/g'(x^i)$



Newton's Method for Backward Euler

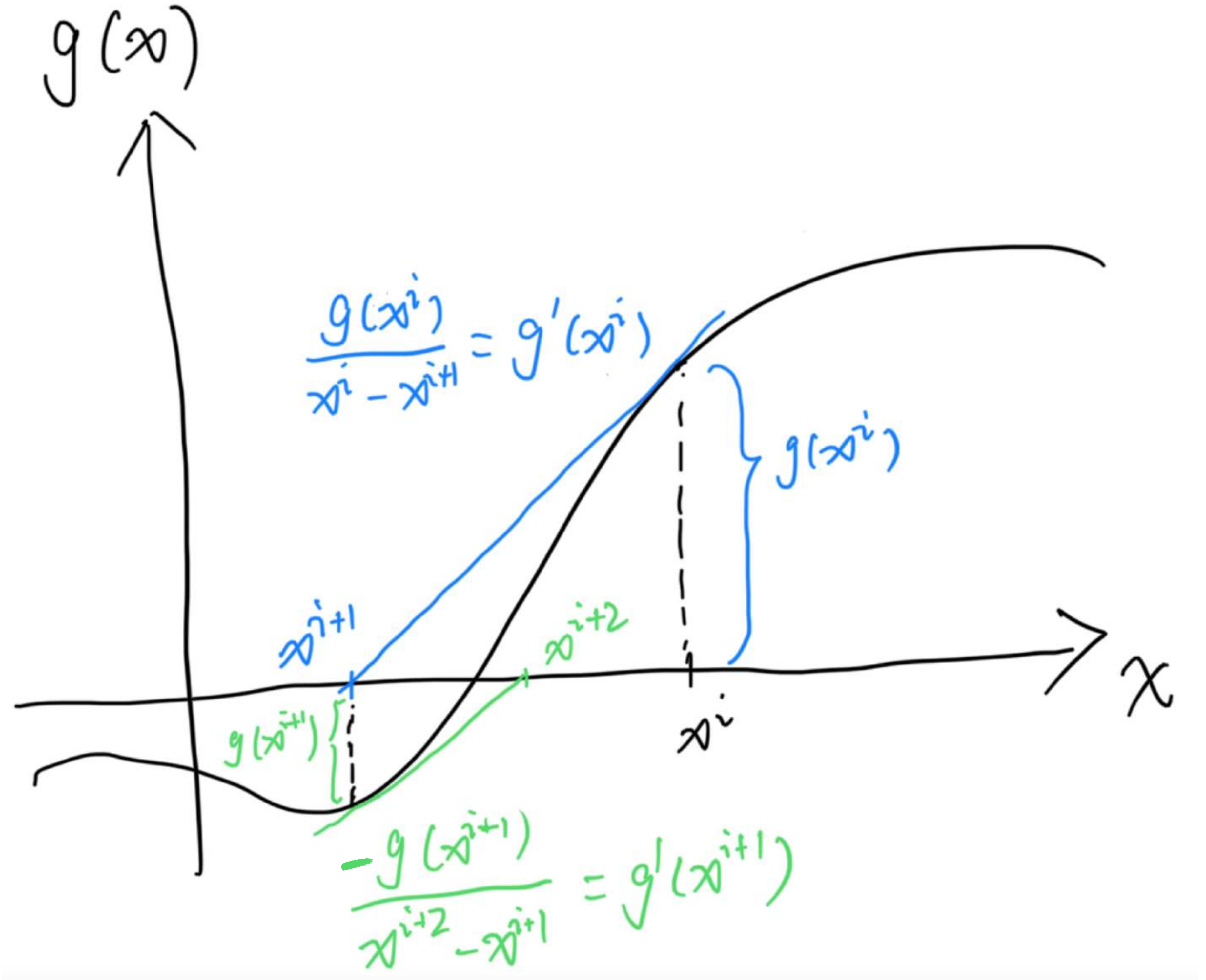
Formulation

Let $g(x) = M(x - (x^n + \Delta t v^n)) - \Delta t^2 f(x)$

We want to solve $g(x) = 0$

Newton's method in 1D:

- **Start from initial guess** x^0
- **For each iteration (until convergence)**
  - $x^{i+1} \leftarrow x^i - g(x^i)/g'(x^i)$



$g(x)$
x
$\frac{g(x^i)}{x^i - x^{i+1}} = g'(x^i)$
$g(x^i)$
x^{i+1}
x^{i+2}
x^i
$g(x^{i+1})$
$\frac{-g(x^{i+1})}{x^{i+2} - x^{i+1}} = g'(x^{i+1})$

In higher dimensions:

$$x^{i+1} \leftarrow x^i - (\nabla g(x^i))^{-1} g(x^i)$$

Derivation:

Linearly approximate $g(x) = 0$ **at** x^i :

$$g(x) = g(x^i) + \nabla g(x^i)(x - x^i)$$

$$g(x^{i+1}) \approx g(x^i) + \nabla g(x^i)(x^{i+1} - x^i) = 0$$

Newton's Method for Backward Euler

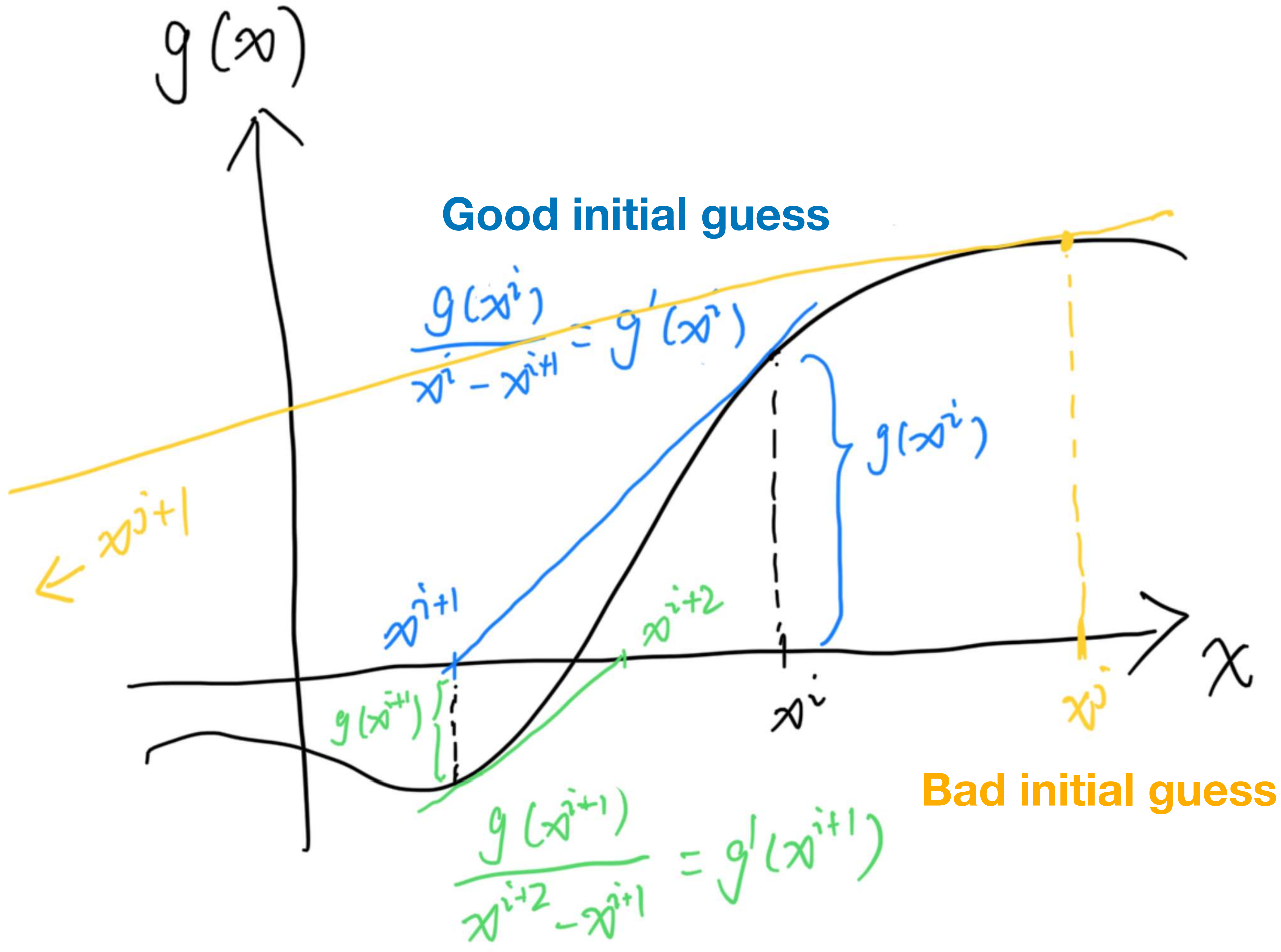
Pseudo-code

Algorithm 1: Newton's Method for Backward Euler Time Integration

```
Result: $x^{n+1}$, $v^{n+1}$
1 $x^i \leftarrow x^n$;
2 while $\|M(x^i - (x^n + \Delta t v^n)) - \Delta t^2 f(x^i)\| > \epsilon$ do
3     solve $M(x - (x^n + \Delta t v^n)) - \Delta t^2(f(x^i) + \nabla f(x^i)(x - x^i)) = 0$
       for $x$;
4     $x^i \leftarrow x$;
5 $x^{n+1} \leftarrow x^i$;
6 $v^{n+1} \leftarrow (x^{n+1} - x^n)/\Delta t$;
```

Convergence Issue of Newton's Method

Over-shooting



$g(x)$
x
Good initial guess
$\frac{g(x^i)}{x^i - x^{i+1}} = g'(x^i)$
$g(x^i)$
x^{i+1}
x^{i+2}
$g(x^{i+1})$
x^i
x^j
x^{j+1}
$\frac{g(x^{i+1})}{x^{i+2} - x^{i+1}} = g'(x^{i+1})$

Bad initial guess → Simulation explodes!

Optimization Time Integration

$$x^{n+1} = \arg\min_x E(x)$$

$$\text{where } E(x) = \frac{1}{2}\|x - \tilde{x}^n\|_M^2 + \Delta t^2 P(x).$$

$$\tilde{x}^n = x^n + \Delta t v^n$$

$$\tfrac{1}{2}\|x - \tilde{x}^n\|_M^2 = \tfrac{1}{2}(x - \tilde{x}^n)^T M (x - \tilde{x}^n)$$

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

At the local minimum of $E(x)$, $\frac{\partial E}{\partial x}(x^{n+1}) = 0$

$$M(x^{n+1} - (x^n + \Delta t v^n)) - \Delta t^2 f(x^{n+1}) = 0.$$

Optimization Time Integration

Newton's Method with Line Search

We want to solve $\nabla E(x) = 0$

Newton's method:

- **Start from initial guess** x^0
- **For each iteration (until convergence)**
  - $x^{i+1} \leftarrow x^i - (\nabla E(x^i))^{-1} \nabla E(x^i)$

Let $p = -(\nabla E(x^i))^{-1} \nabla E(x^i)$

Line Search along direction p**:**

$$\min_{\alpha} E(x^i + \alpha p)$$

$$x^{i+1} \leftarrow x^i + \alpha p$$

Theory:

If p **is a descent direction at** $x = x^i$ **(like** $-\nabla E(x^i)$**),**

$\exists \alpha > 0,\ s.t.\ E(x^i + \alpha p) < E(x^i)$

— need $\nabla^2 E(x)$ **to be symmetric positive-definite**

Idea:

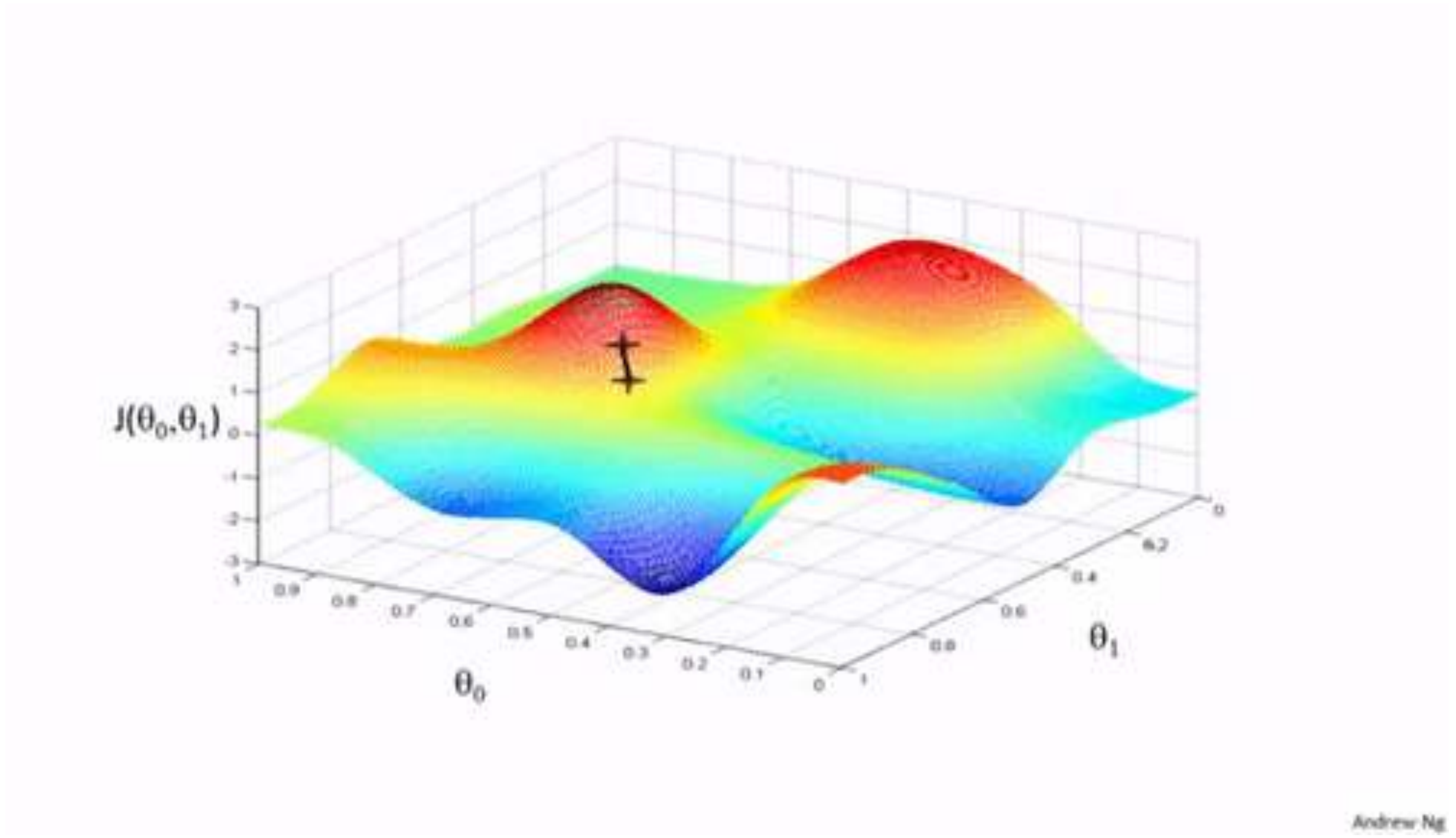
We can project $\nabla^2 E(x)$ **to a nearby SPD matrix for computing** p

Then we can ensure $E(x^{i+1}) < E(x^i)\ \forall i$

— no explosion!

Optimization Time Integration

Newton's Method with Line Search, 2D Illustration



Global Convergence with Line Search

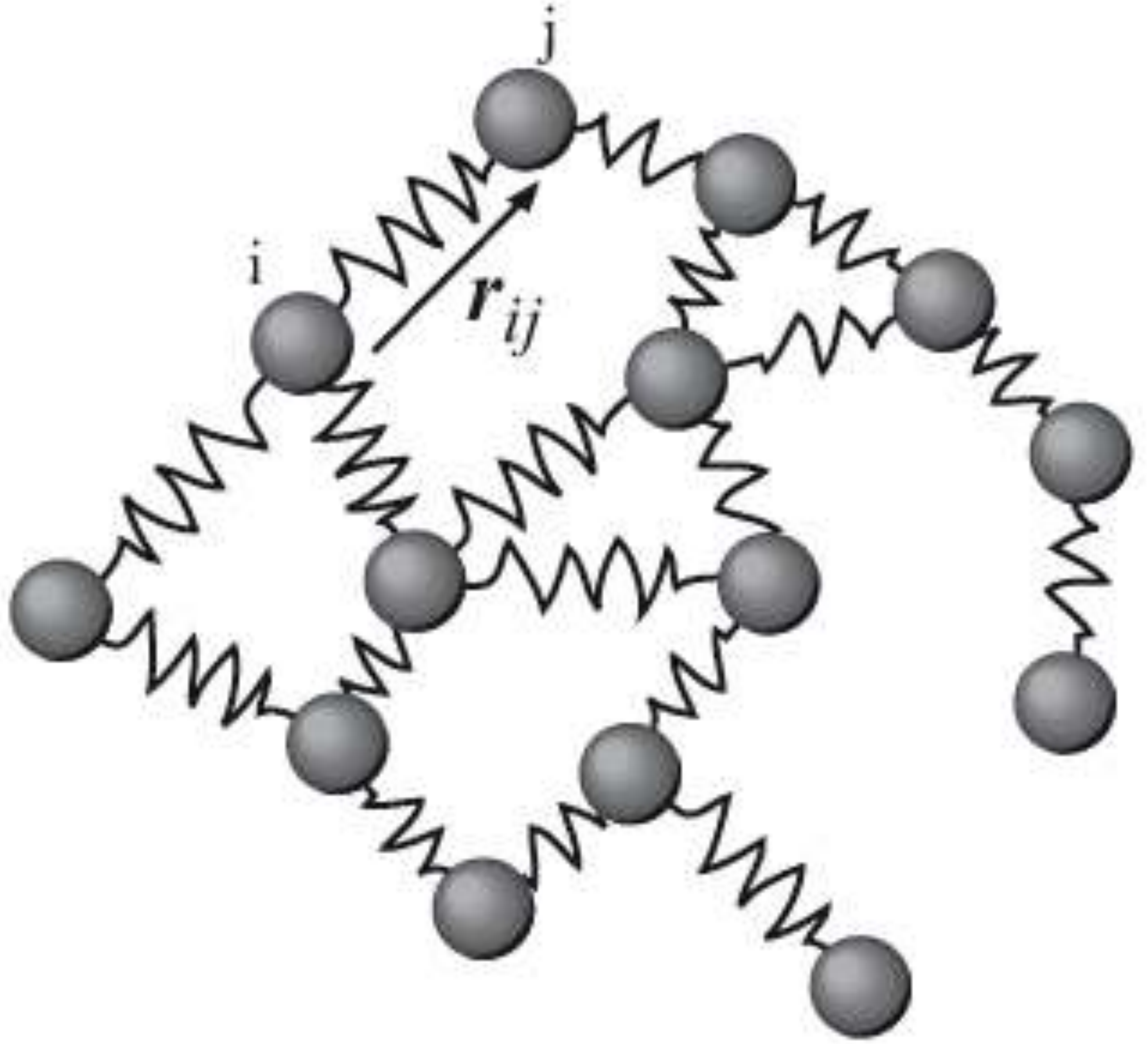
Pseudo-code

Algorithm 3: Projected Newton Method for Backward Euler Time Integration

```
Result: $x^{n+1}$, $v^{n+1}$
1 $x^i \leftarrow x^n$;
2 do
3     $P \leftarrow \text{SPDProjection}(\nabla^2 E(x^i))$;
4     $p \leftarrow -P^{-1} \nabla E(x^i)$;
5     $\alpha \leftarrow \text{BackTrackingLineSearch}(x^i, p)$;  //
6     $\boxed{x^i \leftarrow x^i + \alpha p;}$
7 while $\|p\|_\infty / h > \epsilon$;
8 $x^{n+1} \leftarrow x^i$;
9 $v^{n+1} \leftarrow (x^{n+1} - x^n)/\Delta t$;
```

Algorithm 2: Backtracking Line Search

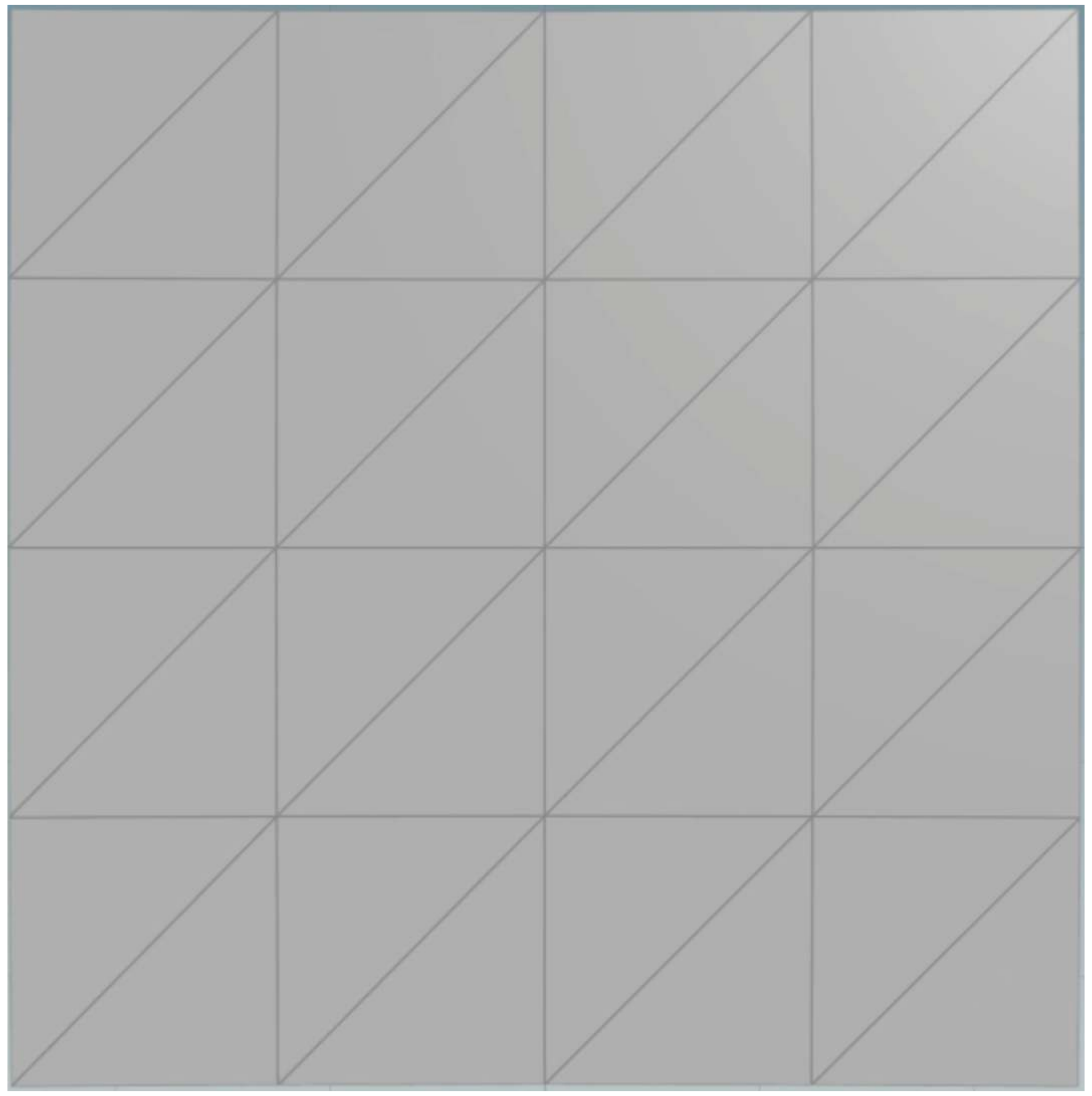
```
Result: $\alpha$
1 $\alpha \leftarrow 1$;
2 while $E(x^i + \alpha p) > E(x^i)$ do
3     $\alpha \leftarrow \alpha/2$;
```



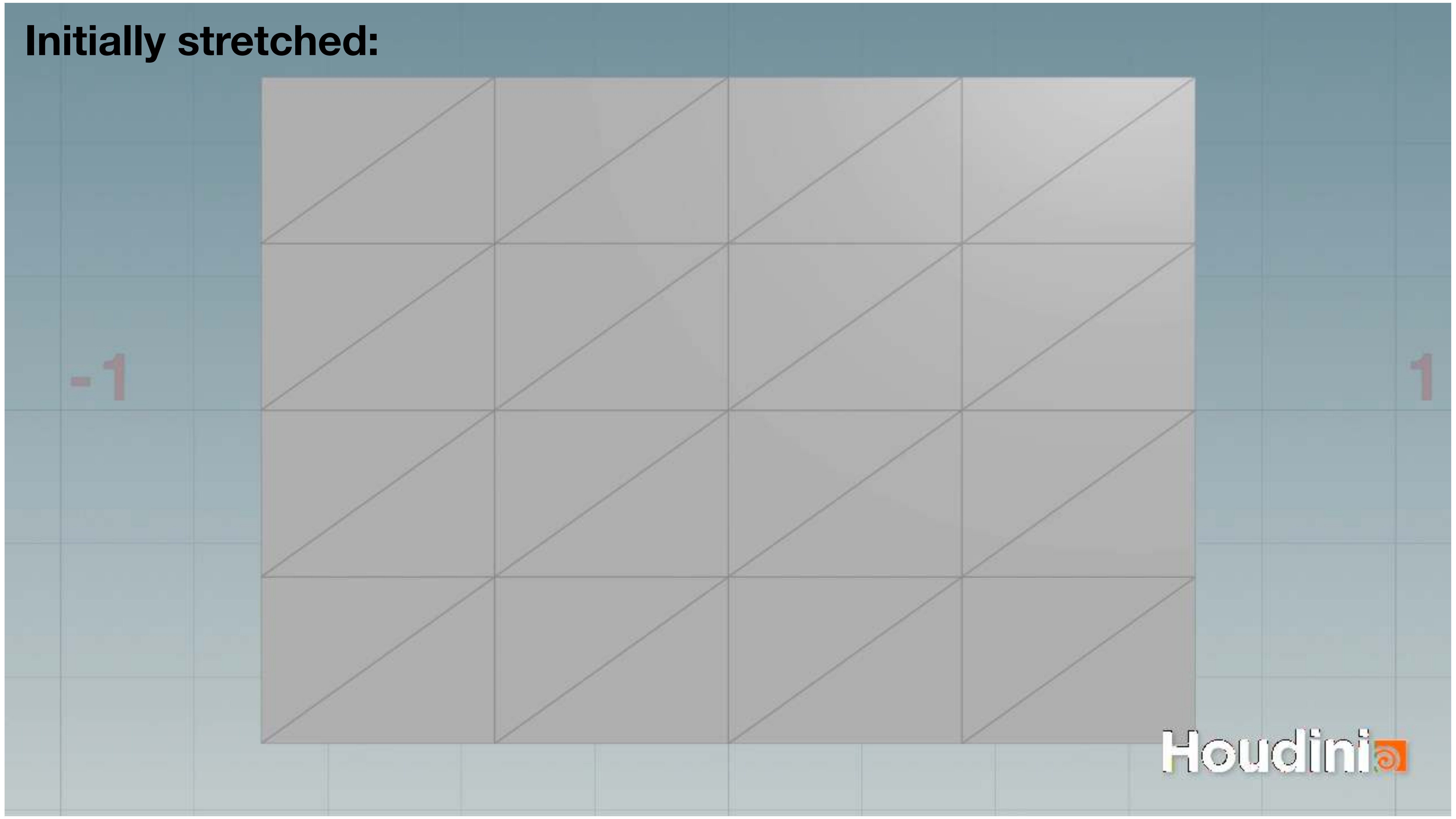
Case Study — Mass-Spring Systems

Case Study — Mass-Spring Simulation

An Initially Stretched Elastic Square

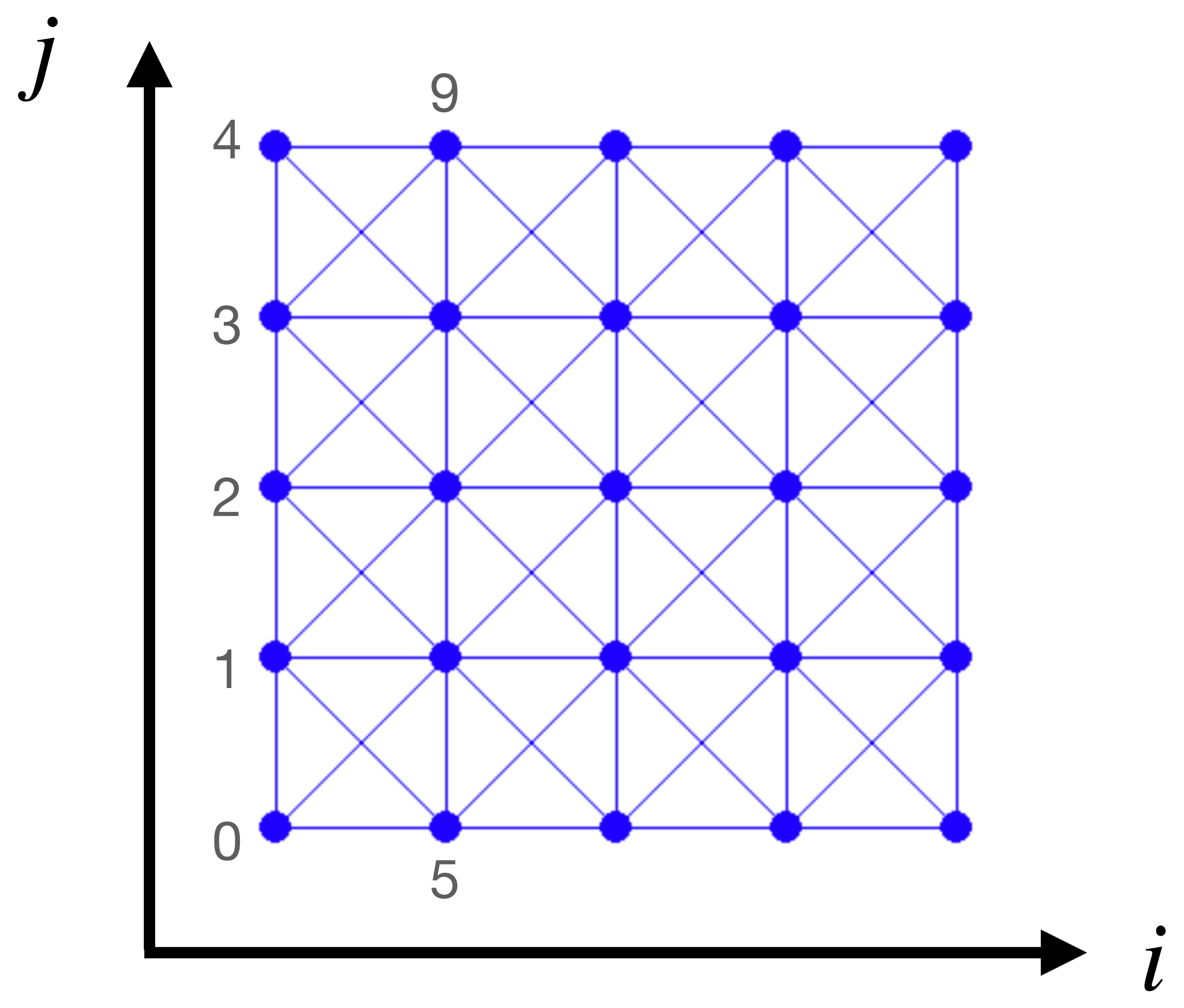


Rest shape



Mass-Spring Representation of Solids

- Mass particles connected by springs
  - square_mesh.py



```python
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

$$\begin{aligned} x^{n+1} &= x^n + \Delta t v^{n+1}, \\ v^{n+1} &= v^n + \Delta t M^{-1} f^{n+1} \end{aligned} \quad \Longleftrightarrow \quad \underbrace{\boxed{E(x)}}_{\text{Incremental Potential}} = \overbrace{\boxed{\frac{1}{2}\|x - (x^n + hv^n)\|_M^2}}^{\text{Inertia term}} + h^2 \overbrace{\boxed{P(x)}}^{\text{Elasticity}}.$$

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

Algorithm 3: Projected Newton Method for Backward Euler Time Integration

```
Result: $x^{n+1}$, $v^{n+1}$
$x^i \leftarrow x^n$;
do
    $P \leftarrow \text{SPDProjection}(\overset{\text{Energy Hessian}}{\boxed{\nabla^2 E(x^i)}})$;
    $p \leftarrow -P^{-1}\underset{\text{Energy Gradient}}{\boxed{\nabla E(x^i)}}$;
    $\alpha \leftarrow \text{BackTrackingLineSearch}(x^i, p)$; //
    $x^i \leftarrow x^i + \alpha p$;
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: $\alpha$
$\alpha \leftarrow 1$;
while $\overset{\text{Energy Value}}{\boxed{E(x^i + \alpha p)}} > E(x^i)$ do
    $\alpha \leftarrow \alpha/2$;
```

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

```python
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

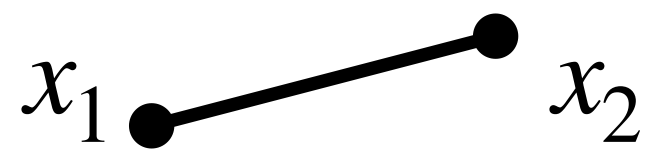
- Hooke's Law in 1D:

Spring stiffness

- $E = \frac{1}{2}\boxed{k}(\boxed{\Delta x})^2$

Spring displacement

- In higher dimensions:



Current length

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

Rest length

A strain measure

- To avoid computing square root, we define

Area weighting

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

Elasticity energy density (elasticity energy per unit area)

Continuous setting:

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

Incremental Potential

Mass-Spring Elasticity Energy Gradient and Hessian

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

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

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

MassSpringEnergy.py

```python
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

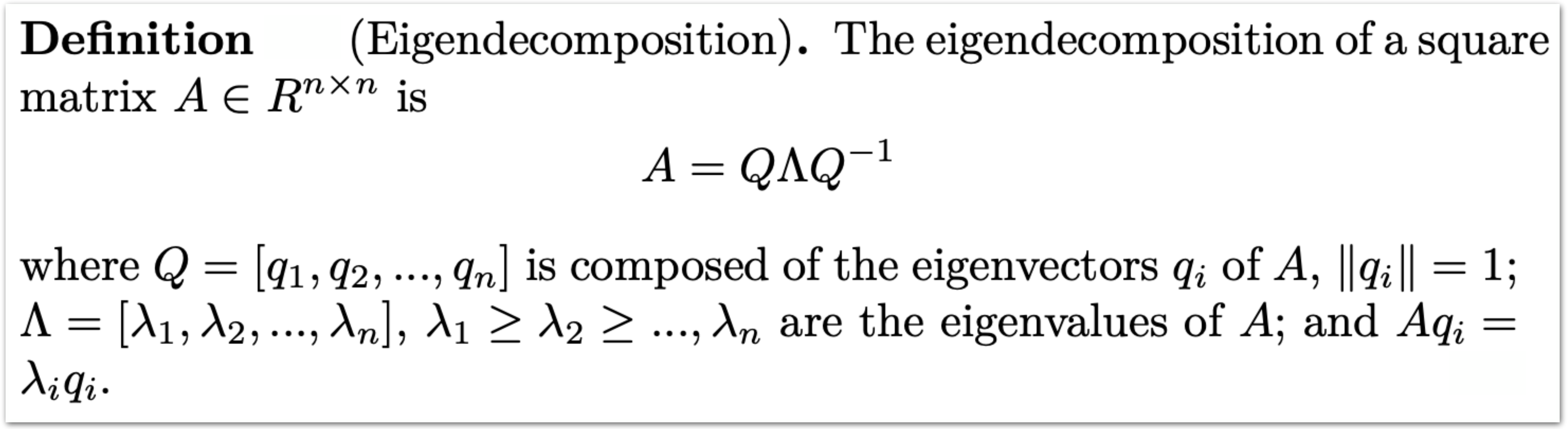
```python
def hess(x, e, l2, k):
    IJV = [[0] * (len(e) * 16), [0] * (len(e) * 16), np.array([0.0] * (len(e) * 16))]
    for i in range(0, len(e)):
        diff = x[e[i][0]] - x[e[i][1]]
        H_diff = 2 * k[i] / l2[i] * (2 * np.outer(diff, diff) + (diff.dot(diff) - l2[i]) * np.identity(2))
        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: $x^{n+1}$, $v^{n+1}$
1 $x^i \leftarrow x^n$;
2 do
3     $P \leftarrow \text{SPDProjection}(\nabla^2 E(x^i))$;
4     $p \leftarrow -P^{-1}\nabla E(x^i)$;
5     $\alpha \leftarrow \text{BackTrackingLineSearch}(x^i, p)$; //
6     $x^i \leftarrow x^i + \alpha p$;
7 while $\|p\|_\infty / h > \epsilon$;
8 $x^{n+1} \leftarrow x^i$;
9 $v^{n+1} \leftarrow (x^{n+1} - x^n)/\Delta t$;
```

Algorithm 2: Backtracking Line Search

```
  Result: $\alpha$
1 $\alpha \leftarrow 1$;
2 while $E(x^i + \alpha p) > E(x^i)$ do
3     $\alpha \leftarrow \alpha/2$;
```

```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

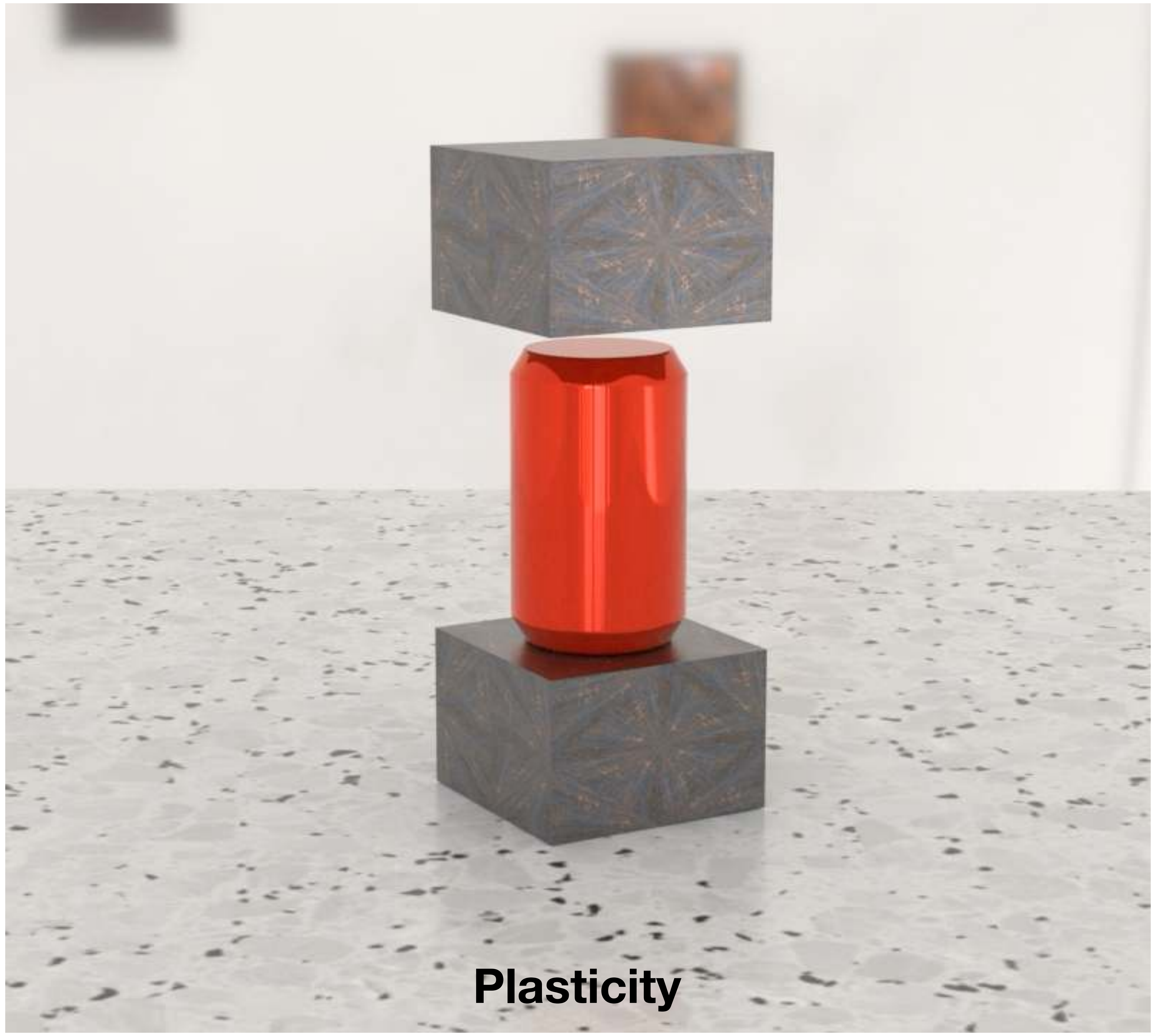
```python
# Mass-Spring Solids Simulation

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
```

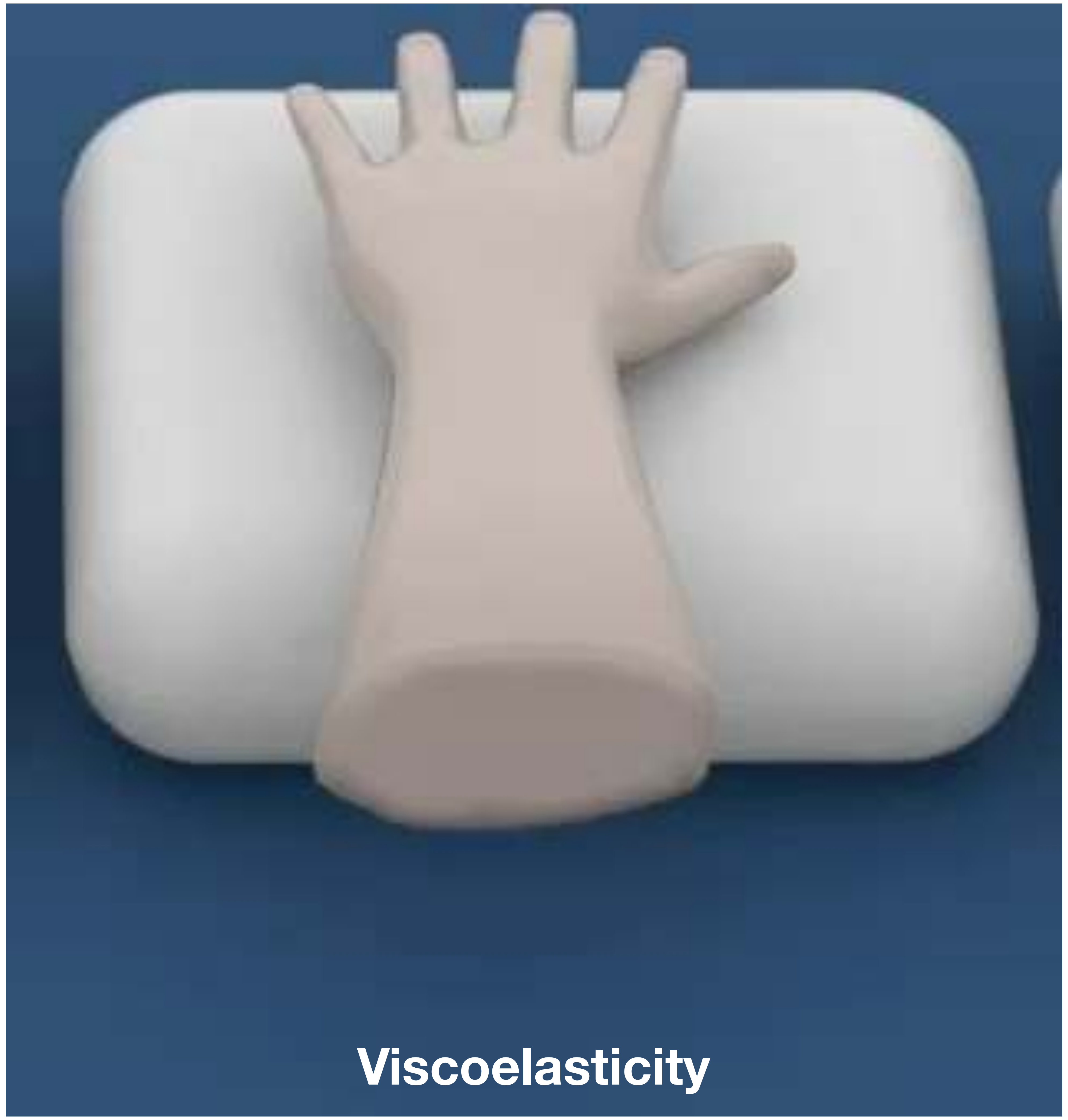
```python
# simulation with visualization
resolution = np.array([900, 900])
offset = resolution / 2
scale = 200
def screen_projection(x):
    return [offset[0] + scale * x[0], resolution[1] - (offset[1] + scale * x[1])]

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

More Topics on Deformable Solids: Inelasticity



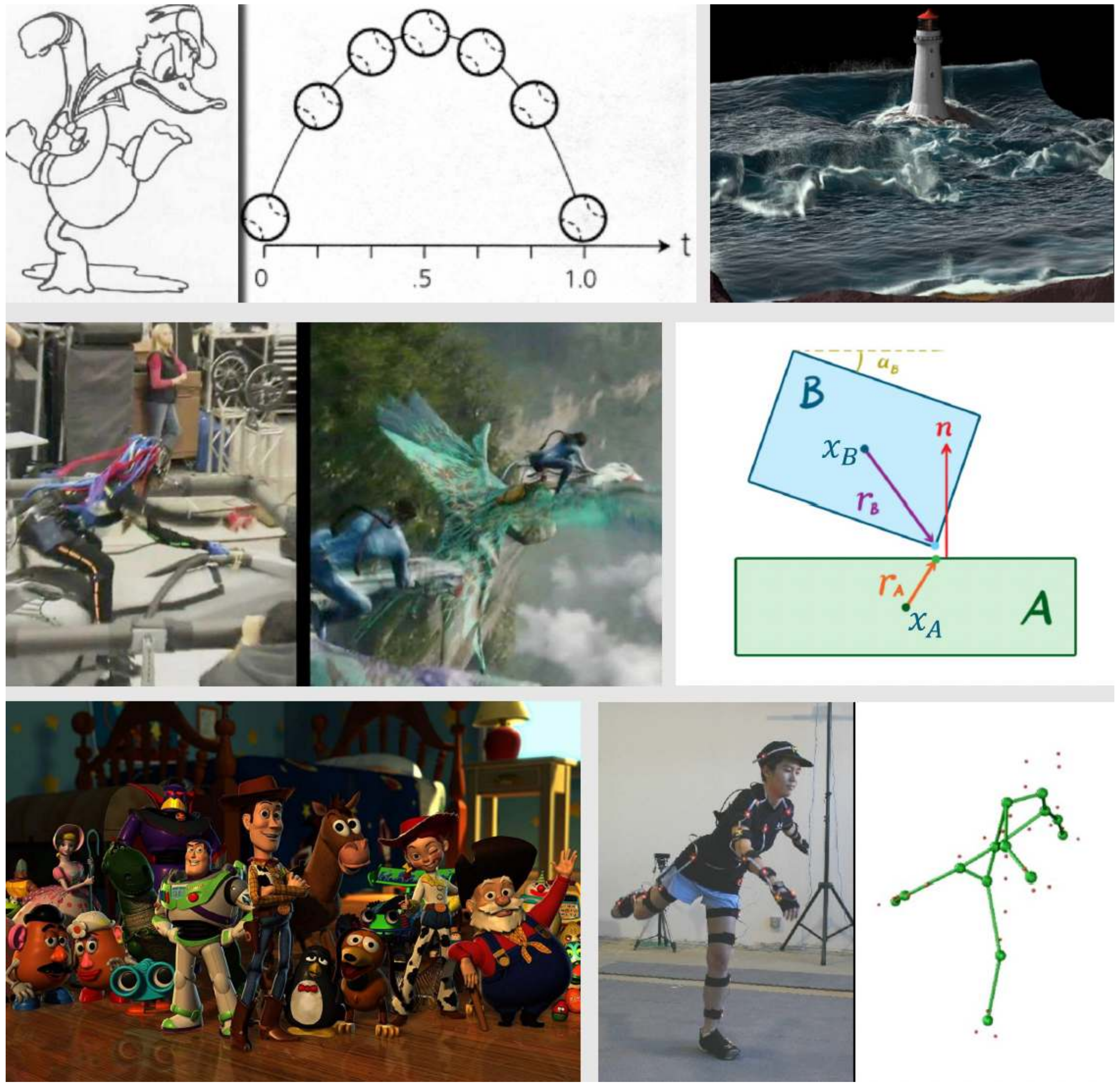


More Topics on Deformable Solids: Contact

More Topics on Deformable Solids: Fracture

15-464/664: Technical Animation

Instructor: Nancy Pollard

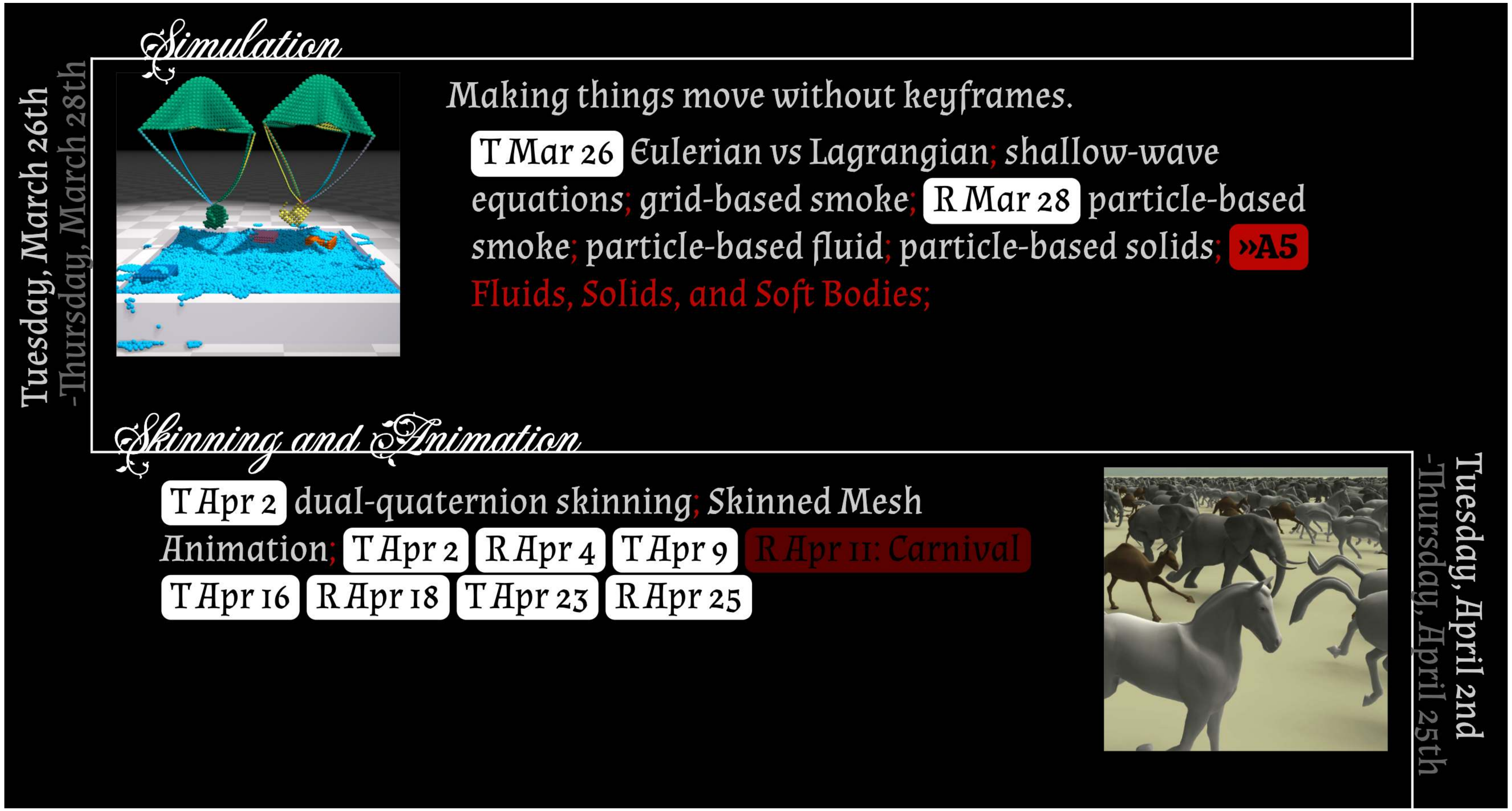


Topics:

- Inverse Kinematics
- Rigging & Skinning
- Motion Capture
- Fluid Simulation
- Cloth Dynamics
- Rigid Body Collisions
- Character Animation

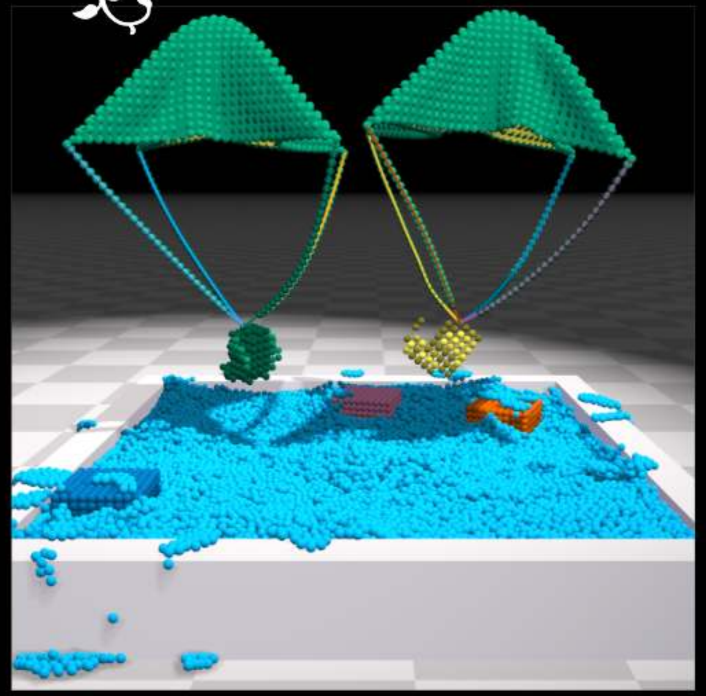
15-472/672/772: Real-Time Computer Graphics

Instructor: Jim McCann



Tuesday, March 26th -Thursday, March 28th

Simulation



Making things move without keyframes.

T Mar 26 Eulerian vs Lagrangian; shallow-wave equations; grid-based smoke; R Mar 28 particle-based smoke; particle-based fluid; particle-based solids; »A5 Fluids, Solids, and Soft Bodies;

Skinning and Animation

T Apr 2 dual-quaternion skinning; Skinned Mesh Animation; T Apr 2 R Apr 4 T Apr 9 R Apr 11: Carnival T Apr 16 R Apr 18 T Apr 23 R Apr 25

Tuesday, April 2nd -Thursday, April 25th

15-763: Physics-based Animation of Solids and Fluids

Instructor: Minchen Li

Topics:

- Optimization Time Integration
- Contact and Friction
- Inversion-Free Elasticity
- Governing Equations
- Finite Element Discretization
- Reduced-Order Models
- Fluids Simulation

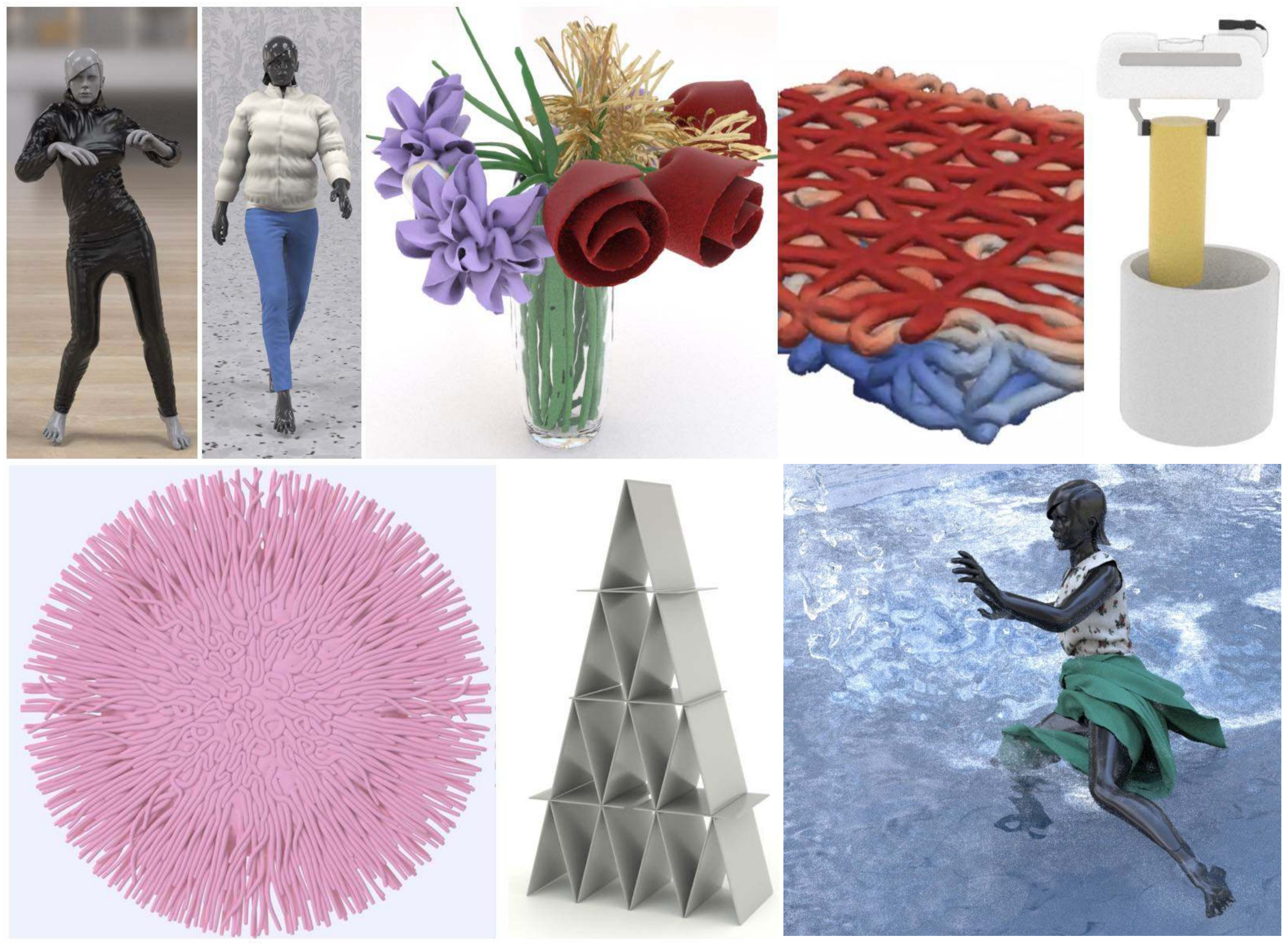


Image Sources

- https://r-wong253249-sp.blogspot.com/2013/11/pose-to-pose-and-frame-by-frame-research.html
- https://dreamfarmstudios.com/blog/what-is-3d-rigging/
- https://drive.google.com/file/d/1oxeQ9L_DX_u_3nig-DMoW_GHZGO1Sva9/preview
- https://medium.com/@jaleeladejumo/gradient-descent-from-scratch-batch-gradient-descent-stochastic-gradient-descent-and-mini-batch-def681187473
- https://academic-accelerator.com/encyclopedia/spring-system
- http://graphics.cs.cmu.edu/nsp/course/15464-s21/www/
- http://graphics.cs.cmu.edu/courses/15-472-s24/