Reliable Simulation of Elastodynamics
15-462/662 (S24) Guest Lecture

Geometry → Appearance → Animation
Animation

Keyframe Animation

Skinning Animation

Physics-based Animation

\[ f = m \frac{dv}{dt} \]
Physics-based Animation

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

Granular media

Fluids
Today: Elasticity of Deformable Solids
A Reliable Simulation Approach based on Numerical Optimization
Spatial Discretization

\[ \Omega \]

\[ x^n = \begin{bmatrix} x_{0x}^n \\ x_{0y}^n \\ x_{0z}^n \\ x_{1x}^n \\ x_{1y}^n \\ x_{1z}^n \\ \vdots \end{bmatrix} \]

\[ v^n = \begin{bmatrix} v_{0x}^n \\ v_{0y}^n \\ v_{0z}^n \\ v_{1x}^n \\ v_{1y}^n \\ v_{1z}^n \\ \vdots \end{bmatrix} \]
Governing Equation (Conservation of Momentum)

- The spatially discrete, temporally continuous form

\[ \frac{dx}{dt} = v, \]
\[ M \frac{dv}{dt} = f. \]

- Mass matrix (for now)

\[
M = \begin{pmatrix}
  m_1 & m_1 & m_2 \\
  m_1 & m_1 & m_2 \\
  m_1 & m_1 & m_2
\end{pmatrix}
\]
Time Stepping (Time Integration)

\[ x^{n+1} = x^n + \Delta t \cdot v^n \]

\[ v^{n+1} = v^n + \Delta t \cdot \frac{\Delta x}{\Delta t} \]
Governing Equation (Temporally Discrete)
Forward Difference, Forward Euler

- Forward difference approximation on velocity and acceleration

\[
\begin{align*}
\left( \frac{dx}{dt} \right)_n & \approx \frac{x^{n+1} - x^n}{\Delta t} \\
\left( \frac{dv}{dt} \right)_n & \approx \frac{v^{n+1} - v^n}{\Delta t} \\
(f(t^n + \Delta t) & = f(t^n) + \frac{df}{dt}(t^n) \Delta t + O(\Delta t^2) )
\end{align*}
\]

Taylor’s expansion

\[
\begin{align*}
\frac{x^{n+1} - x^n}{\Delta t} & = v^n, \\
M \frac{v^{n+1} - v^n}{\Delta t} & = f^n.
\end{align*}
\]

\[
\begin{align*}
x^{n+1} & = x^n + \Delta tv^n, \\
v^{n+1} & = v^n + \Delta tM^{-1} f^n.
\end{align*}
\]
Newton’s 2nd Law (Temporally Discrete)
Forward and Backward Difference, Symplectic Euler

- Forward difference on acceleration, backward difference on velocity

\[ x^{n+1} = x^n + \Delta t v^{n+1} \]
\[ v^{n+1} = v^n + \Delta t M^{-1} f^n \]
Newton’s 2nd Law (Temporally Discrete)
Backward Difference, Backward Euler (or Implicit Euler)

• Backward difference approximation on velocity and acceleration

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

\[ f^{n+1} = f(x^{n+1}) \]

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

\[ x_{n+1} = x_n + \Delta tv^n, \]
\[ v_{n+1} = v_n + \Delta tM^{-1} f^n. \]

\[ x_{n+1} = x_n + \Delta tv^{n+1}, \]
\[ v_{n+1} = v_n + \Delta tM^{-1} f^{n+1}. \]

\[ x_{n+1} = x_n + \Delta tv^{n+1}, \]
\[ v_{n+1} = v_n + \Delta tM^{-1} f^{n+1}. \]
Newton’s Method for Backward Euler Formulation

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

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 \( g(x) = M(x - (x^n + \Delta tv^n)) - \Delta t^2 f(x) \)

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

Newton’s method in 1D:
- Start from initial guess \( x^0 \)
- For each iteration (until convergence)
  - \( x^{i+1} \leftarrow x^i - g(x^i)/g'(x^i) \)

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 $\| M(x^i - (x^n + \Delta tv^n) - \Delta t^2 f(x^i)) \| > \epsilon$ do
3. \[
    \text{solve } M(x - (x^n + \Delta tv^n) - \Delta t^2 (f(x^i) + \nabla f(x^i)(x - x^i)) = 0 \text{ 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

\[
g'(x_i) = \frac{g(x_{i+1})}{x_{i+2} - x_{i+1}} - g'(x_i)
\]

Good initial guess

Bad initial guess

Simulation explodes!
Optimization Time Integration

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

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

\[ \tilde{x}^n = x^n + \Delta tv^n \]

\[ \frac{1}{2} \| x - \tilde{x}^n \|_M^2 = \frac{1}{2} (x - \tilde{x}^n)^T M (x - \tilde{x}^n) \]

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

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

\[ M(x^{n+1} - (x^n + \Delta tv^n)) - \Delta t^2 f(x^{n+1}) = 0. \]
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 $p$:
\[
\min_{\alpha} E(x^i + \alpha p)
\]
then
\[
x^{i+1} \leftarrow x^i + \alpha p
\]

**Theory:**
If $p$ is a descent direction at $x = x^i$ (like $-\nabla E(x^i)$),
\[
\exists \alpha > 0, \text{ s.t. } E(x^i + \alpha p) < E(x^i)
\]
— 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

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

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

Algorithm 2: Backtracking Line Search

\textbf{Result: } \alpha

1. \( \alpha \leftarrow 1; \)
2. \( \text{while } E(x^i + \alpha p) > E(x^i) \text{ do} \)
3. \( \quad \alpha \leftarrow \alpha / 2; \)
Case Study — Mass-Spring Systems
Case Study — Mass-Spring Simulation
An Initially Stretched Elastic Square

Rest shape

Initially stretched:
Mass-Spring Representation of Solids

- Mass particles connected by springs

```
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 + 1) * (n_seg + 1) + j, (i + 1) * (n_seg + 1) + j + 1)]
            e.append([(i * (n_seg + 1) + j, (i + 1) * (n_seg + 1) + j + 1)])

    return [x, e]
```
Time Integration
Optimization-based Implicit Euler

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

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

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

Algorithm 3: Projected Newton Method for Backward Euler Time Integration

Result: \( x^{n+1}, v^{n+1} \)

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. do
3. while \( E(x^i + \alpha p) > E(x^i) \) do
4. \( \alpha \leftarrow \alpha/2; \)
Incremental Potential

Inertia Term

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

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

\[ \nabla^2 E_I(x) = M - \text{ SPD} \]

```python
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: \( \Delta x \)

- In higher dimensions:
  \[ \frac{1}{2} k (\|x_1 - x_2\| - l)^2 \]
  - Current length: \( \|x_1 - x_2\| \)
  - Rest length: \( l \)

- To avoid computing square root, we define

  \[ P_e(x) = \frac{1}{2} k \left( \frac{\|x_1 - x_2\|^2}{l^2} - 1 \right)^2 \]

  - Elasticity energy density
  - (elasticity energy per unit area)

Continuous setting:
\[ P = \int_{\Omega^0} \Psi dX \]
Incremental Potential

Mass-Spring Elasticity Energy Gradient and Hessian

\[ P_e(x) = l^2 \frac{1}{2} k \left( \frac{\|x_1 - x_2\|^2}{l^2} - 1 \right)^2 \]

\[ \frac{\partial P_e}{\partial x_1} (x) = -\frac{\partial P_e}{\partial x_2} (x) = 2k \left( \frac{\|x_1 - x_2\|^2}{l^2} - 1 \right) (x_1 - x_2) \]

\[ \frac{\partial^2 P_e}{\partial x_1^2} (x) = -\frac{\partial^2 P_e}{\partial x_2^2} (x) = \frac{\partial^2 P_e}{\partial x_2 x_1} (x) \]

\[ = \frac{4k}{l^2} (x_1 - x_2)(x_1 - x_2)^T + 2k \left( \frac{\|x_1 - x_2\|^2}{l^2} - 1 \right) I \]

\[ = \frac{2k}{l^2} (2(x_1 - x_2)(x_1 - x_2)^T + (\|x_1 - x_2\|^2 - l^2) I) \]

---

**MassSpringEnergy.py**

```python
import numpy as np
import utils

def val(x, e, 12, k):
    sum = 0.0
    for i in range(0, len(e)):
        diff = x[e[i][0]] - x[e[i][1]]
        sum += 12[i] * 0.5 * k[i] * (diff.dot(diff) / 12[i] - 1) ** 2
    return sum

def grad(x, e, 12, 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) / 12[i] - 1) *
        g[e[i][0]] += g_diff
        g[e[i][1]] -= g_diff
    return g
```
Incremental Potential
Mass-Spring Elasticity Energy Hessian Implementation

\[
\frac{\partial^2 P_e}{\partial x_i^2}(x) = \frac{\partial^2 P_e}{\partial x_1 \partial x_2}(x) = \frac{\partial^2 P_e}{\partial x_2 \partial x_1}(x) = -\frac{\partial^2 P_e}{\partial x_2 x_1}(x)
\]

\[
= \frac{4k}{l^2} (x_1 - x_2)(x_1 - x_2)^T + 2k\left(\frac{||x_1 - x_2||^2}{l^2} - 1\right)I
\]

\[
= \frac{2k}{l^2} (2(x_1 - x_2)(x_1 - x_2)^T + (||x_1 - x_2||^2 - l^2)I)
\]

---

**MassSpringEnergy.py**

```python
def hess(x, e, 12, 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] / 12[i] * (2 * np.outer(diff, diff)
        + (diff.dot(diff) - 12[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)**

\[
\begin{align*}
\min_{P} \|P - \nabla^2 E(x^i)\|_F \quad \text{s.t.} \quad & v^T P v \geq 0 \quad \forall v \neq 0 \\
\text{Solution:} \quad & \hat{A} = Q \hat{\Lambda} Q^{-1}, \quad \hat{\Lambda}_{ij} = \Lambda_{ij} > 0 \quad \text{?} \quad \Lambda_{ij} : 0
\end{align*}
\]

**Definition (Eigendecomposition).** The eigendecomposition of a square matrix \( A \in \mathbb{R}^{n \times n} \) 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 \( A q_i = \lambda_i q_i \).

```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, 12, k, h):
    return InertiaEnergy.val(x, x_tilde, m) + h * h *
    MassSpringEnergy.val(x, e, 12, k)  # implicit Euler

def IP_grad(x, e, x_tilde, m, 12, k, h):
    return InertiaEnergy.grad(x, x_tilde, m) + h * h *
    MassSpringEnergy.grad(x, e, 12, k)  # implicit Euler

def IP_hess(x, e, x_tilde, m, 12, k, h):
    IJV_In = InertiaEnergy.hess(x, x_tilde, m)
    IJV_MS = MassSpringEnergy.hess(x, e, 12, 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. $x^i \leftarrow x^i + \alpha p;$
3. while $\|p\|_{\infty} / h > \epsilon;$
4. $x^{n+1} \leftarrow x^i;$
5. $v^{n+1} \leftarrow (x^{n+1} - x^n) / \Delta t;$

---

```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]

def search_dir(x, e, x_tilde, m, l2, k, h):
    projected_grad = 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_grad, -reshaped_grad).reshape(len(x), 2)
```

---

*import copy from cmath import inf*
*import numpy as np import scipy.linalg as LA*
*import scipy.sparse as sparse*
*from scipy.sparse.linalg import spsolve*
*import InertiaEnergy import MassSpringEnergy*
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 = 1e6       # 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 = 12 = []
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

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 i in e:
        pygame.draw.aaline(screen, (0, 0, 255), screen_projection(x[i[0]]), screen_projection(x[i[1]]))
    for i in x:
        pygame.draw.circle(screen, (0, 0, 255), screen_projection(x[i]), 0.1 * side_len / n_seg * scale)
    pygame.display.flip()  # flip the display

    # step forward simulation and wait for screen refresh
    [x, 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

Plasticity

Viscoelasticity
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
Instructor: Jim McCann

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
- 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
15-769: 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://dreamfarmstudios.com/blog/what-is-3d-rigging/
- https://drive.google.com/file/d/1oxeQ9L_DX_u_3nig-DMoW_GHZGO1Sva9/preview
- 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/