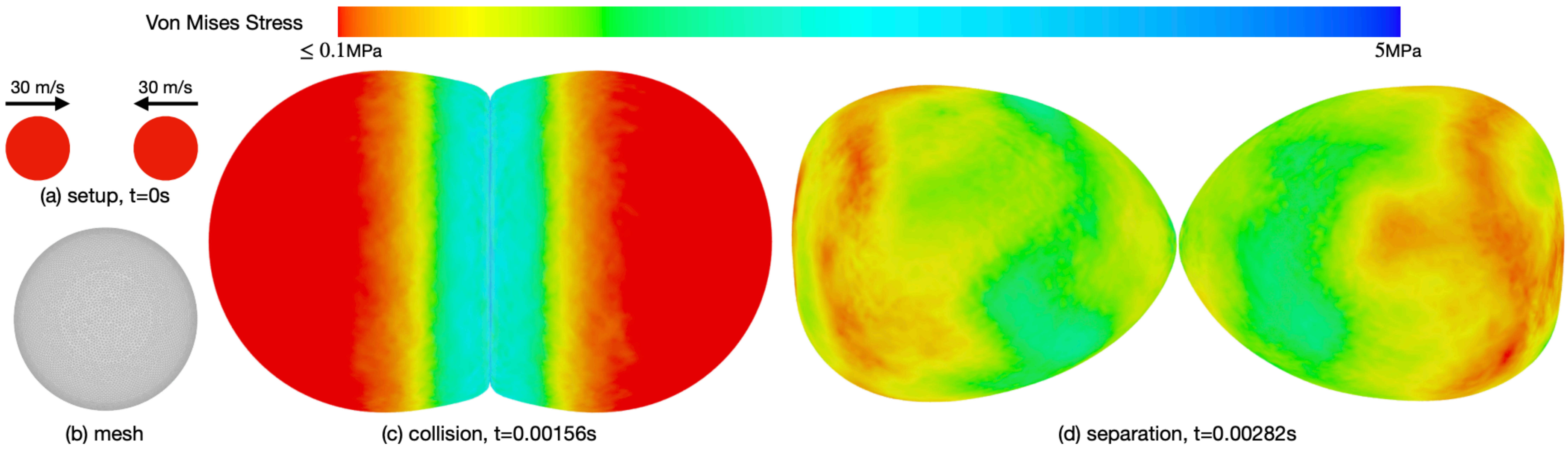
Instructor: Minchen Li



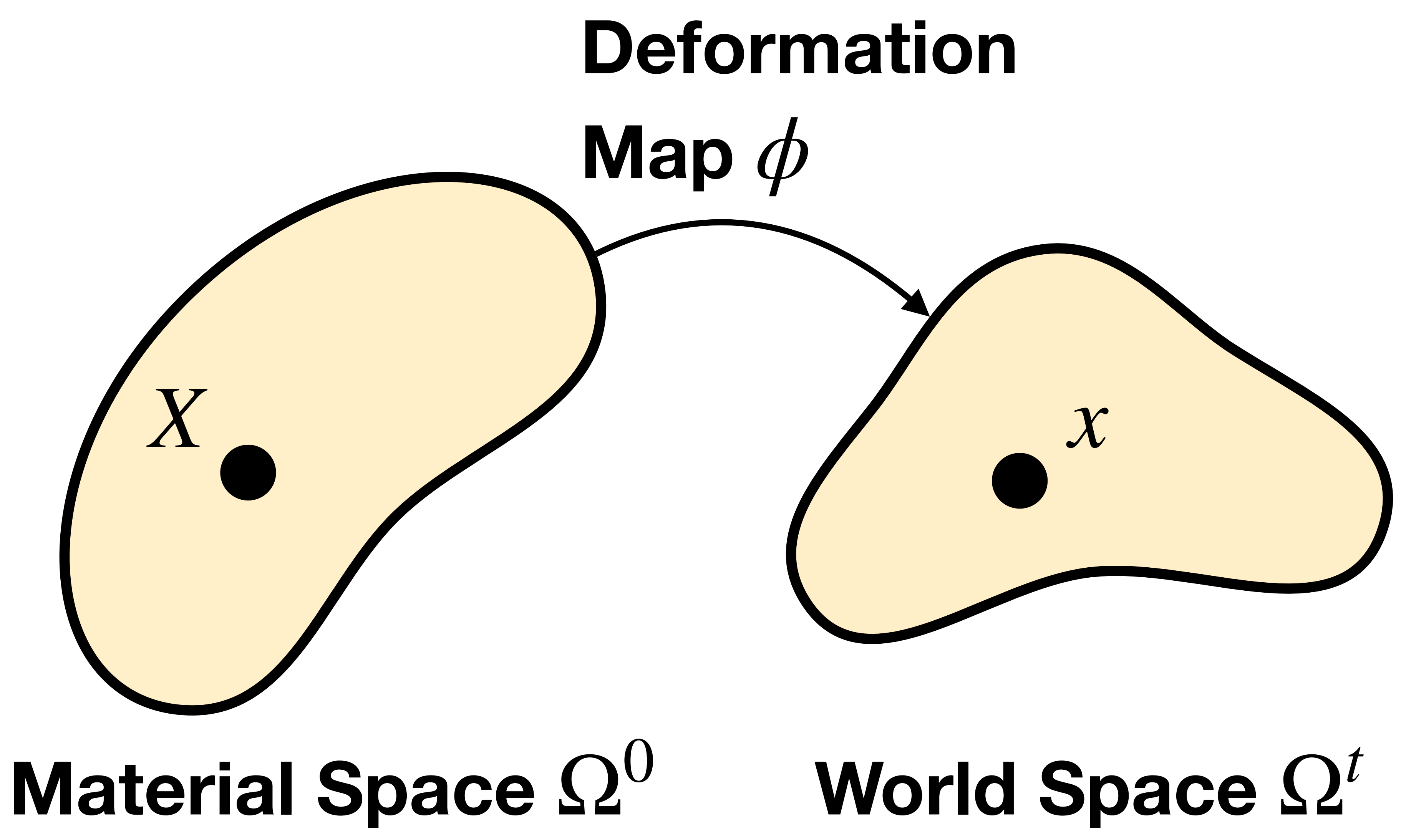
Lec 9: Stress and Its Derivative

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

Recap: Strain Energy

Continuum View and Deformation Gradient

- Treating materials (solid, liquid, or gas) as **continuous pieces of matter**



$$\mathbf{x} = \mathbf{x}(\mathbf{X}, t) = \phi(\mathbf{X}, t)$$

$$x(X, 0) = X$$

- **Deformation Gradient:**

$$\mathbf{F}(\mathbf{X}, t) = \frac{\partial \phi}{\partial \mathbf{X}}(\mathbf{X}, t) = \frac{\partial \mathbf{x}}{\partial \mathbf{X}}(\mathbf{X}, t)$$

$$F_{ij} = \frac{\partial \phi_i}{\partial X_j} = \frac{\partial x_i}{\partial X_j}, \quad i, j = 1, \ldots, d$$

- **Volume change:** $J = \det(F)$

- **Strain Energy:** $P_e = \int_{\Omega_0} \Psi(\mathbf{F}) d\mathbf{X}$

Recap: Strain Energy

Examples and Properties

- **Strain Energy:** $P_e = \int_{\Omega_0} \Psi(\mathbf{F}) d\mathbf{X}$

a square matrix $\mathbf{F}$ is a rotation matrix if and only if

$$\mathbf{F}^T = \mathbf{F}^{-1} \quad \text{and} \quad J \equiv \det(\mathbf{F}) = 1.$$

- **Rigid Null Space:**

$$\Psi(\mathbf{F}) = 0 \quad \forall\, \mathbf{F} = \mathbf{R}$$

- **e.g. penalizing deviation from rotation:**

$$\Psi(\mathbf{F}) = \frac{\mu}{4}\|\mathbf{F}^T\mathbf{F} - \mathbf{I}\|_F^2 + \frac{\lambda}{2}(J-1)^2$$

μ and λ are the Lame parameters

- **e.g. neo-Hookean elasticity:**

$$\Psi_{\mathrm{NH}}(\mathbf{F}) = \frac{\mu}{2}\left(\mathrm{tr}(\mathbf{F}^T\mathbf{F}) - d\right) - \mu \ln(J) + \frac{\lambda}{2}\ln^2(J)$$

Barrier term on J, so inversion-free!

$$\forall\, \mathbf{F} \in \mathbb{R}^{d\times d} \text{ and } d \times d \text{ rotation matrix } \mathbf{R}$$

- **Rotation-Invariance**

$$\Psi(\mathbf{F}) = \Psi(\mathbf{RF})$$

- **Isotropic Elasticity**

$$\Psi(\mathbf{F}) = \Psi(\mathbf{FR})$$

Recap: Strain Energy

Polar Singular Value Decomposition

Algorithm 6: Polar SVD from Standard SVD

Result: $\mathbf{U}$, $\mathbf{\Sigma}$, $\mathbf{V}$

$(\mathbf{U}, \mathbf{\Sigma}\ \mathbf{V}) \leftarrow \text{StandardSVD}(\mathbf{F})$;

if $\det(\mathbf{U}) < 0$ **then**

  $\mathbf{U}(:, d) \leftarrow -\mathbf{U}(:, d)$;

  $\mathbf{\Sigma}_{dd} \leftarrow -\mathbf{\Sigma}_{dd}$;

if $\det(\mathbf{V}) < 0$ **then**

  $\mathbf{V}(:, d) \leftarrow -\mathbf{V}(:, d)$;

  $\mathbf{\Sigma}_{dd} \leftarrow -\mathbf{\Sigma}_{dd}$;

$$\Psi_{\mathrm{NH}}(\mathbf{F}) = \hat{\Psi}_{\mathrm{NH}}(\mathbf{\Sigma}) = \frac{\mu}{2}\left(\sum_{i}^{d} \sigma_i^2 - d\right) - \mu \ln(J) + \frac{\lambda}{2} \ln^2(J)$$

Recap: Strain Energy

Simplified Models

- **Linear Elasticity**

$$\Psi_{\text{lin}}(\mathbf{F}) = \mu \|\boldsymbol{\epsilon}\|_{\text{F}}^2 + \frac{\lambda}{2}\text{tr}^2(\boldsymbol{\epsilon})$$

$\boldsymbol{\epsilon} = \frac{1}{2}(\mathbf{F} + \mathbf{F}^T) - \mathbf{I}$ is the small strain tensor

- **Consistency to Linear Elasticity**

$$\hat{\Psi}(\mathbf{I}) = 0, \quad \frac{\partial \hat{\Psi}}{\partial \sigma_i}(\mathbf{I}) = 0, \quad \text{and} \quad \frac{\partial^2 \hat{\Psi}}{\partial \sigma_i \partial \sigma_j}(\mathbf{I}) = 2\mu\delta_{ij} + \lambda.$$

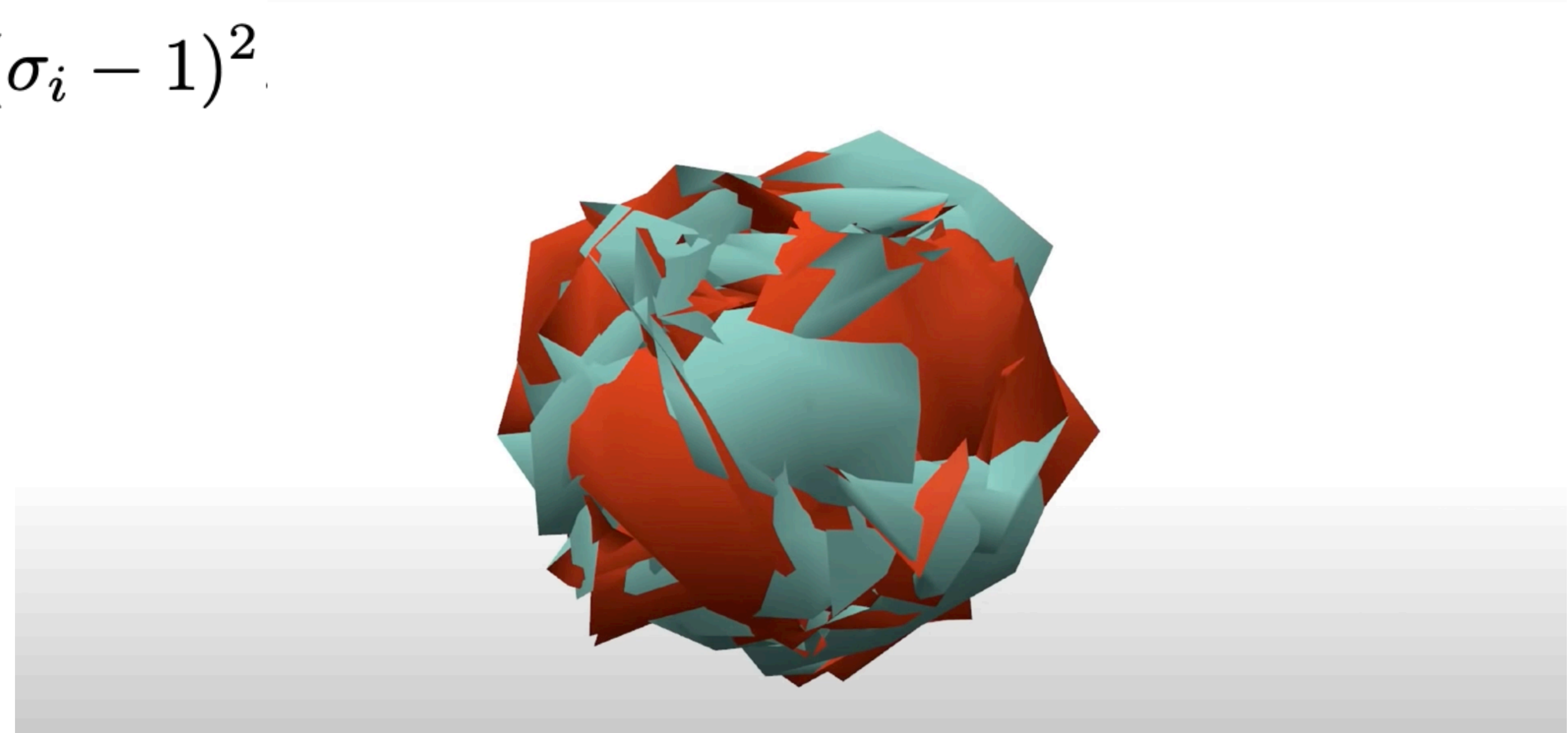
- **Linearly Corotated Elasticity**

$$\Psi_{\text{LC}}(\mathbf{F}) = \Psi_{\text{lin}}((\mathbf{R}^n)^T\mathbf{F})$$

- **As-Rigid-As-Possible (ARAP)**

$$\Psi_{\text{ARAP}}(\mathbf{F}) = \mu \sum_i^a (\sigma_i - 1)^2$$

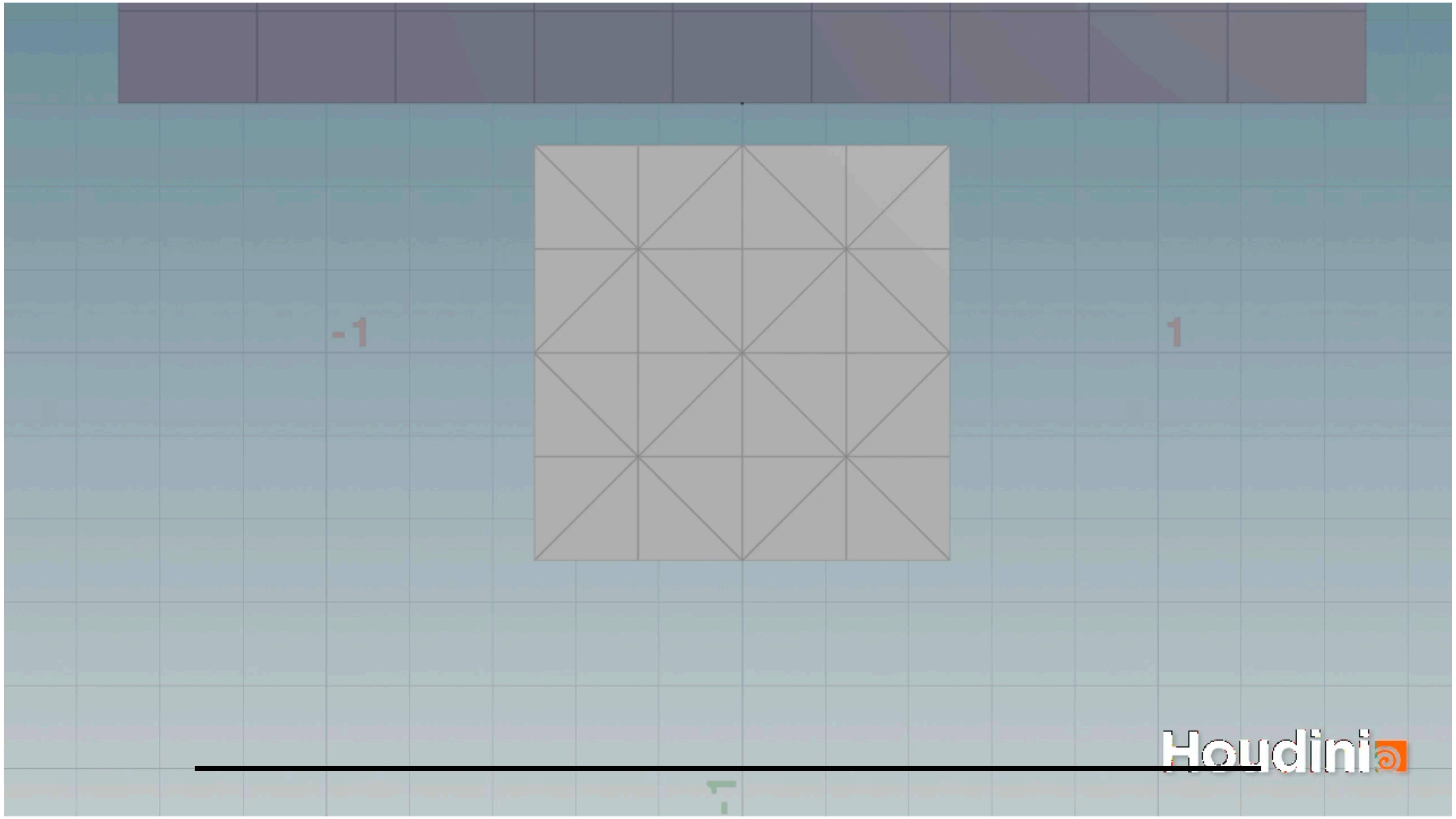
- **Above are all invertible models (allowing det(F) ≤ 0)**
  - **No line search filtering needed**
  - **Can deal with inverted configurations**
  - **Usually smoother — easier to optimize**
  - **More e.g. Stable neo-Hookean [Smith et al. 2018]**



Bunny with randomized vertices

Today: Stress and Its Derivatives

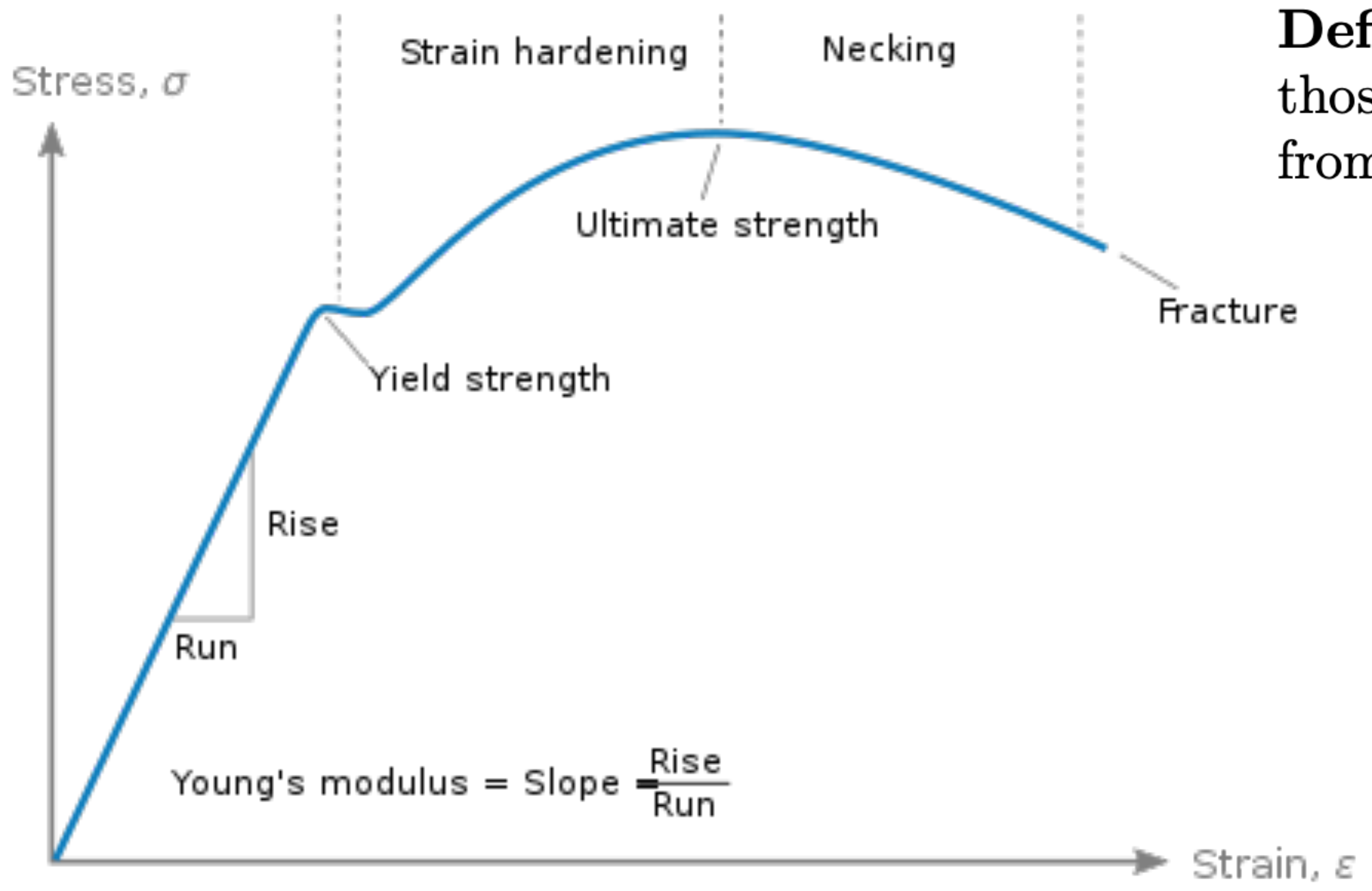
Simulating Inversion-Free Elastodynamics



Stress

Definition and Examples

- a tensor field (like F) measuring pressure (unit: force per area)
- related to F through a constitutive relationship, e.g. neo-Hookean model



Definition (Hyperelastic Materials)**.** Hyperelastic materials are those elastic solids whose **first Piola-Kirchoff stress P** can be derived from an strain energy density function $\Psi(\mathbf{F})$ via

$$\mathbf{P} = \frac{\partial \Psi}{\partial \mathbf{F}} \qquad P_{ij} = \frac{\partial \Psi}{\partial F_{ij}}$$

- **Cauchy stress**

$$\sigma = \frac{1}{J}\mathbf{P}\mathbf{F}^T = \frac{1}{\det(\mathbf{F})}\frac{\partial \Psi}{\partial \mathbf{F}}\mathbf{F}^T$$

Stress

Calculating P in the Diagonal Space for Isotropic Materials

$$\mathbf{P} = \mathbf{U}\hat{\mathbf{P}}\mathbf{V}^T$$

where $\quad \mathbf{F} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T, \quad \Psi(\mathbf{F}) = \hat{\Psi}(\mathbf{\Sigma}), \quad$ and $\quad \hat{\mathbf{P}}_{ij} = \frac{\partial \hat{\Psi}}{\partial \sigma_i}\delta_{ij}$

Example For the Neo-Hookean model

$$\hat{\Psi}_{\mathrm{NH}}(\mathbf{\Sigma}) = \frac{\mu}{2}(\sum_i^d \sigma_i^2 - d) - \mu \ln(J) + \frac{\lambda}{2}\ln^2(J).$$

Thus, we can first perform SVD on $\mathbf{F} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}$ and derive

$$\hat{\mathbf{P}}_{ii} = \mu(\sigma_i - \frac{1}{\sigma_i}) + \lambda \ln(J)\frac{1}{\sigma_i}$$

to compute $\frac{\partial \Psi}{\partial \mathbf{F}} = \mathbf{P} = \mathbf{U}\hat{\mathbf{P}}\mathbf{V}^T$ without symbolically deriving the derivative of Ψ w.r.t. $\mathbf{F}$.

Stress

Calculating P in the Diagonal Space for Isotropic Materials — Proof

$\delta\Psi = \frac{\partial\Psi}{\partial\mathbf{F}}(\mathbf{F}) : \delta\mathbf{F} = \frac{\partial\Psi}{\partial\mathbf{F}}(\mathbf{RF}) : \delta(\mathbf{RF})$	**Rigid null space**
$(\mathbf{P}(\mathbf{F})) : (\delta\mathbf{F}) = (\mathbf{P}(\mathbf{RF})) : \delta(\mathbf{RF})$	**Hyperelasticity**
$(\mathbf{P}(\mathbf{F})) : (\delta\mathbf{F}) = (\mathbf{P}(\mathbf{RF}))_{ij} R_{ik} \delta F_{kj}$	**Index notation**
$(\mathbf{P}(\mathbf{F})) : (\delta\mathbf{F}) = (\mathbf{R}^T\mathbf{P}(\mathbf{RF})) : \delta\mathbf{F}$	**Associativity**
$\mathbf{P}(\mathbf{F}) = \mathbf{R}^T\mathbf{P}(\mathbf{RF})$	$\forall \delta\mathbf{F}$
$\mathbf{RP}(\mathbf{F}) = \mathbf{P}(\mathbf{RF})$	**Multiply $\mathbf{R}$ on both sides**

Similarly, we can prove $P(F)R = P(FR)$ for Isotropic Elasticity.

$$\mathbf{P}(\mathbf{F}) = \mathbf{P}(\mathbf{U}\Sigma\mathbf{V}^T) = \mathbf{U}\mathbf{P}(\Sigma)\mathbf{V}^T = \mathbf{U}\hat{\mathbf{P}}\mathbf{V}^T.$$

Stress Derivative

Derivation for Diagonal Space Calculation

$$\mathbf{P}(\mathbf{F}) = \mathbf{P}(\mathbf{R}\mathbf{R}^T\mathbf{F}\mathbf{Q}\mathbf{Q}^T) = \mathbf{R}\mathbf{P}(\mathbf{R}^T\mathbf{F}\mathbf{Q})\mathbf{Q}^T$$

For arbitrary rotation matrices $\mathbf{R}$ and $\mathbf{Q}$

Call $\mathbf{K} = \mathbf{R}^T\mathbf{F}\mathbf{Q}$, we have

$$\mathbf{P}(\mathbf{F}) = \mathbf{R}\mathbf{P}(\mathbf{K})\mathbf{Q}^T$$

$$\delta\mathbf{P} = \mathbf{R}\left[\frac{\partial\mathbf{P}}{\partial\mathbf{F}}(\mathbf{K}) : \delta(\mathbf{K})\right]\mathbf{Q}^T = \mathbf{R}\left[\frac{\partial\mathbf{P}}{\partial\mathbf{F}}(\mathbf{K}) : (\mathbf{R}^T\delta\mathbf{F}\mathbf{Q})\right]\mathbf{Q}^T$$

Use $\delta\mathbf{F}$

$$\delta\mathbf{P} = \mathbf{U}\left[\frac{\partial\mathbf{P}}{\partial\mathbf{F}}(\Sigma) : (\mathbf{U}^T\delta\mathbf{F}\mathbf{V})\right]\mathbf{V}^T$$

Set $\mathbf{R} = \mathbf{U}$ and $\mathbf{Q} = \mathbf{V}$

$$(\delta\mathbf{P})_{ij} = U_{ik}\left(\frac{\partial\mathbf{P}}{\partial\mathbf{F}}(\Sigma)\right)_{klmn} U_{rm}\delta F_{rs}V_{sn}V_{jl}, \quad \text{and} \quad (\delta\mathbf{P})_{ij} = \left(\frac{\partial\mathbf{P}}{\partial\mathbf{F}}(\mathbf{F})\right)_{ijrs}\delta F_{rs}$$

$$\left(\frac{\partial\mathbf{P}}{\partial\mathbf{F}}(\mathbf{F})\right)_{ijrs} = \left(\frac{\partial\mathbf{P}}{\partial\mathbf{F}}(\Sigma)\right)_{klmn} U_{ik}U_{rm}V_{sn}V_{jl}$$

$\forall\delta\mathbf{F}$

Stress Derivative

Diagonal Space Derivatives

$$(\delta \mathbf{P})_{ij} = U_{ik} \left(\frac{\partial \mathbf{P}}{\partial \mathbf{F}}(\Sigma) \right)_{klmn} U_{rm} \delta F_{rs} V_{sn} V_{jl}, \quad \text{and} \quad (\delta \mathbf{P})_{ij} = \left(\frac{\partial \mathbf{P}}{\partial \mathbf{F}}(\mathbf{F}) \right)_{ijrs} \delta F_{rs}$$

$$\frac{\partial \mathbf{P}}{\partial \mathbf{F}}(\Sigma) = \begin{bmatrix} A & & & \\ & B_{12} & & \\ & & B_{23} & \\ & & & B_{31} \end{bmatrix}$$

(With flattening and permutation)

$$\mathbf{A} = \begin{pmatrix} \hat{\Psi}_{,\sigma_1\sigma_1} & \hat{\Psi}_{,\sigma_1\sigma_2} & \hat{\Psi}_{,\sigma_1\sigma_3} \\ \hat{\Psi}_{,\sigma_2\sigma_1} & \hat{\Psi}_{,\sigma_2\sigma_2} & \hat{\Psi}_{,\sigma_2\sigma_3} \\ \hat{\Psi}_{,\sigma_3\sigma_1} & \hat{\Psi}_{,\sigma_3\sigma_2} & \hat{\Psi}_{,\sigma_3\sigma_3} \end{pmatrix}$$

$$\mathbf{B}_{ij} = \frac{1}{\sigma_i^2 - \sigma_j^2} \begin{pmatrix} \sigma_i \hat{\Psi}_{,\sigma_i} - \sigma_j \hat{\Psi}_{,\sigma_j} & \sigma_j \hat{\Psi}_{,\sigma_i} - \sigma_i \hat{\Psi}_{,\sigma_j} \\ \sigma_j \hat{\Psi}_{,\sigma_i} - \sigma_i \hat{\Psi}_{,\sigma_j} & \sigma_i \hat{\Psi}_{,\sigma_i} - \sigma_j \hat{\Psi}_{,\sigma_j} \end{pmatrix}$$

$$\mathbf{B}_{ij} = \frac{1}{2} \frac{\hat{\Psi}_{,\sigma_i} - \hat{\Psi}_{,\sigma_j}}{\sigma_i - \sigma_j} \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} + \frac{1}{2} \frac{\hat{\Psi}_{,\sigma_i} + \hat{\Psi}_{,\sigma_j}}{\sigma_i + \sigma_j} \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$$

- **Other ways to compute: Analytic Eigensystems for Isotropic Distortion Energies [Smith et al. 2019]**
  - **Modes with negative Eigenvalues are directly projected out**

Stress Derivative

Implementation

NeoHookeanEnergy.py

```python
import utils
import numpy as np
import math

def polar_svd(F):
    [U, s, VT] = np.linalg.svd(F)
    if np.linalg.det(U) < 0:
        U[:, 1] = -U[:, 1]
        s[1] = -s[1]
    if np.linalg.det(VT) < 0:
        VT[1, :] = -VT[1, :]
        s[1] = -s[1]
    return [U, s, VT]
```

$$\mathbf{F} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$$

```python
def dPsi_div_dsigma(s, mu, lam):
    ln_sigma_prod = math.log(s[0] * s[1])
    inv0 = 1.0 / s[0]
    dPsi_dsigma_0 = mu * (s[0] - inv0) + lam * inv0 * ln_sigma_prod
    inv1 = 1.0 / s[1]
    dPsi_dsigma_1 = mu * (s[1] - inv1) + lam * inv1 * ln_sigma_prod
    return [dPsi_dsigma_0, dPsi_dsigma_1]
```

$$\hat{\mathbf{P}}_{ii} = \mu(\sigma_i - \frac{1}{\sigma_i}) + \lambda \ln(J)\frac{1}{\sigma_i}$$

$$\mathbf{A} = \begin{pmatrix} \hat{\Psi}_{,\sigma_1\sigma_1} & \hat{\Psi}_{,\sigma_1\sigma_2} & \hat{\Psi}_{,\sigma_1\sigma_3} \\ \hat{\Psi}_{,\sigma_2\sigma_1} & \hat{\Psi}_{,\sigma_2\sigma_2} & \hat{\Psi}_{,\sigma_2\sigma_3} \\ \hat{\Psi}_{,\sigma_3\sigma_1} & \hat{\Psi}_{,\sigma_3\sigma_2} & \hat{\Psi}_{,\sigma_3\sigma_3} \end{pmatrix}$$

```python
def d2Psi_div_dsigma2(s, mu, lam):
    ln_sigma_prod = math.log(s[0] * s[1])
    inv2_0 = 1 / (s[0] * s[0])
    d2Psi_dsigma2_00 = mu * (1 + inv2_0) - lam * inv2_0 * (ln_sigma_prod - 1)
    inv2_1 = 1 / (s[1] * s[1])
    d2Psi_dsigma2_11 = mu * (1 + inv2_1) - lam * inv2_1 * (ln_sigma_prod - 1)
    d2Psi_dsigma2_01 = lam / (s[0] * s[1])
    return [[d2Psi_dsigma2_00, d2Psi_dsigma2_01], [d2Psi_dsigma2_01, d2Psi_dsigma2_11]]

def B_left_coef(s, mu, lam):
    sigma_prod = s[0] * s[1]
    return (mu + (mu - lam * math.log(sigma_prod)) / sigma_prod) / 2
```

$$\mathbf{B}_{ij} = \frac{1}{2}\frac{\hat{\Psi}_{,\sigma_i} - \hat{\Psi}_{,\sigma_j}}{\sigma_i - \sigma_j}\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} + \frac{1}{2}\frac{\hat{\Psi}_{,\sigma_i} + \hat{\Psi}_{,\sigma_j}}{\sigma_i + \sigma_j}\begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$$

Stress Derivative

Implementation (Cont.)

$$\left(\frac{\partial \mathbf{P}}{\partial \mathbf{F}}(\mathbf{F})\right)_{ijrs} = \left(\frac{\partial \mathbf{P}}{\partial \mathbf{F}}(\Sigma)\right)_{klmn} U_{ik}U_{rm}V_{sn}V_{jl}$$

$$\frac{\partial \mathbf{P}}{\partial \mathbf{F}}(\Sigma) = \begin{bmatrix} A & \\ & B_{12} \end{bmatrix}$$ **in 2D**

With flattening permutation

$\mathbf{F}_{11}, \mathbf{F}_{22}, \mathbf{F}_{21}, \mathbf{F}_{12}$

$\mathbf{P}_{11}$

$\mathbf{P}_{22}$

$\mathbf{P}_{21}$

$\mathbf{P}_{12}$

NeoHookeanEnergy.py

```
def d2Psi_div_dF2(F, mu, lam):
    [U, sigma, VT] = polar_svd(F)

    Psi_sigma_sigma = utils.make_PD(d2Psi_div_dsigma2(sigma, mu, lam))

    B_left = B_left_coef(sigma, mu, lam)
    Psi_sigma = dPsi_div_dsigma(sigma, mu, lam)
    B_right = (Psi_sigma[0] + Psi_sigma[1]) / (2 * max(sigma[0] + sigma[1], 1e-6))
    B = utils.make_PD([[B_left + B_right, B_left - B_right], [B_left - B_right, B_left + B_right]])

    M = np.array([[0, 0, 0, 0]] * 4)
    M[0, 0] = Psi_sigma_sigma[0, 0]
    M[0, 3] = Psi_sigma_sigma[0, 1]
    M[1, 1] = B[0, 0]
    M[1, 2] = B[0, 1]
    M[2, 1] = B[1, 0]
    M[2, 2] = B[1, 1]
    M[3, 0] = Psi_sigma_sigma[1, 0]
    M[3, 3] = Psi_sigma_sigma[1, 1]

    dP_div_dF = np.array([[0, 0, 0, 0]] * 4)
    for j in range(0, 2):
        for i in range(0, 2):
            ij = j * 2 + i
            for s in range(0, 2):
                for r in range(0, 2):
                    rs = s * 2 + r
                    dP_div_dF[ij, rs] = M[0, 0] * U[i, 0] * VT[0, j] * U[r, 0] * VT[0, s] \
                        + M[0, 3] * U[i, 0] * VT[0, j] * U[r, 1] * VT[1, s] \
                        + M[2, 2] * U[i, 0] * VT[1, j] * U[r, 0] * VT[1, s] \
                        + M[2, 1] * U[i, 0] * VT[1, j] * U[r, 1] * VT[0, s] \
                        + M[1, 2] * U[i, 1] * VT[0, j] * U[r, 0] * VT[1, s] \
                        + M[1, 1] * U[i, 1] * VT[0, j] * U[r, 1] * VT[0, s] \
                        + M[3, 0] * U[i, 1] * VT[1, j] * U[r, 0] * VT[0, s] \
                        + M[3, 3] * U[i, 1] * VT[1, j] * U[r, 1] * VT[1, s]
    return dP_div_dF
```

$\mathbf{F}_{11}, \mathbf{F}_{21}, \mathbf{F}_{12}, \mathbf{F}_{22}$

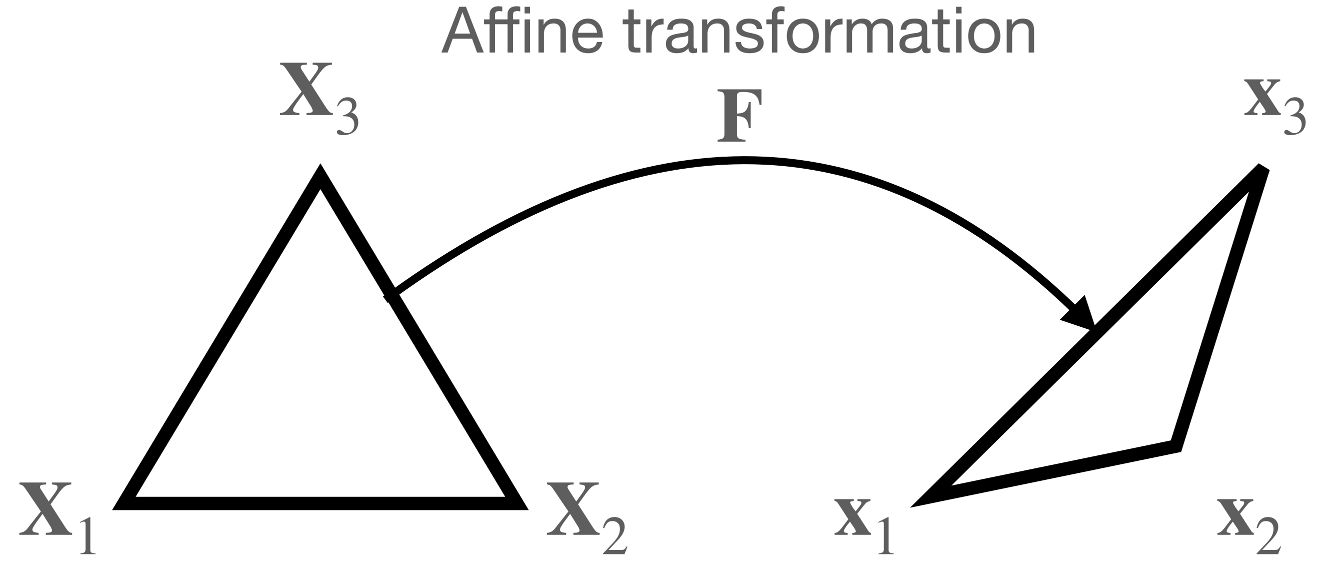
$\mathbf{P}_{11}$

$\mathbf{P}_{21}$

$\mathbf{P}_{12}$

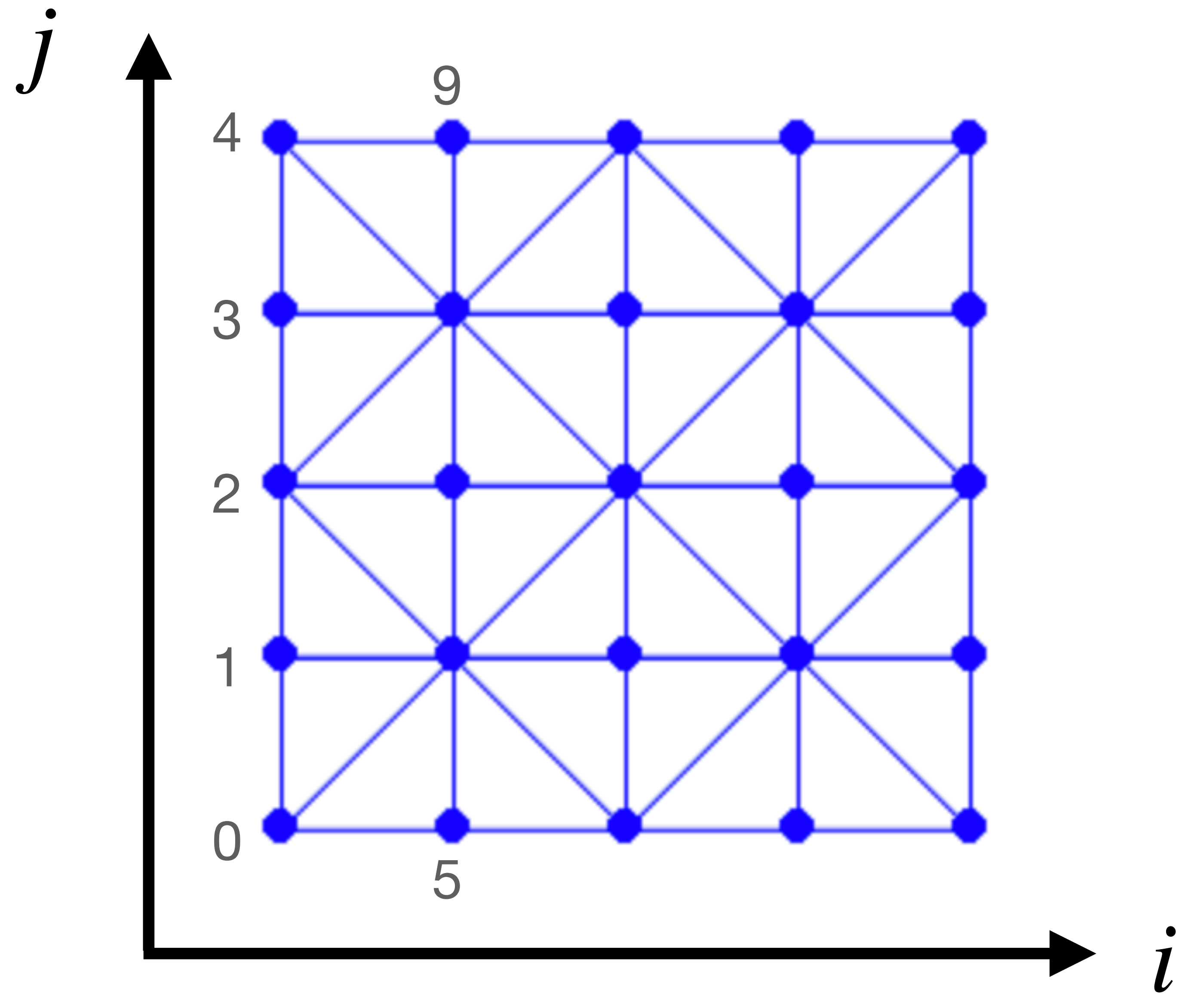
$\mathbf{P}_{22}$

Deformation Gradient on Triangle Meshes



$$\mathbf{x}_2 - \mathbf{x}_1 = \mathbf{F}(\mathbf{X}_2 - \mathbf{X}_1) \quad \text{and} \quad \mathbf{x}_3 - \mathbf{x}_1 = \mathbf{F}(\mathbf{X}_3 - \mathbf{X}_1)$$

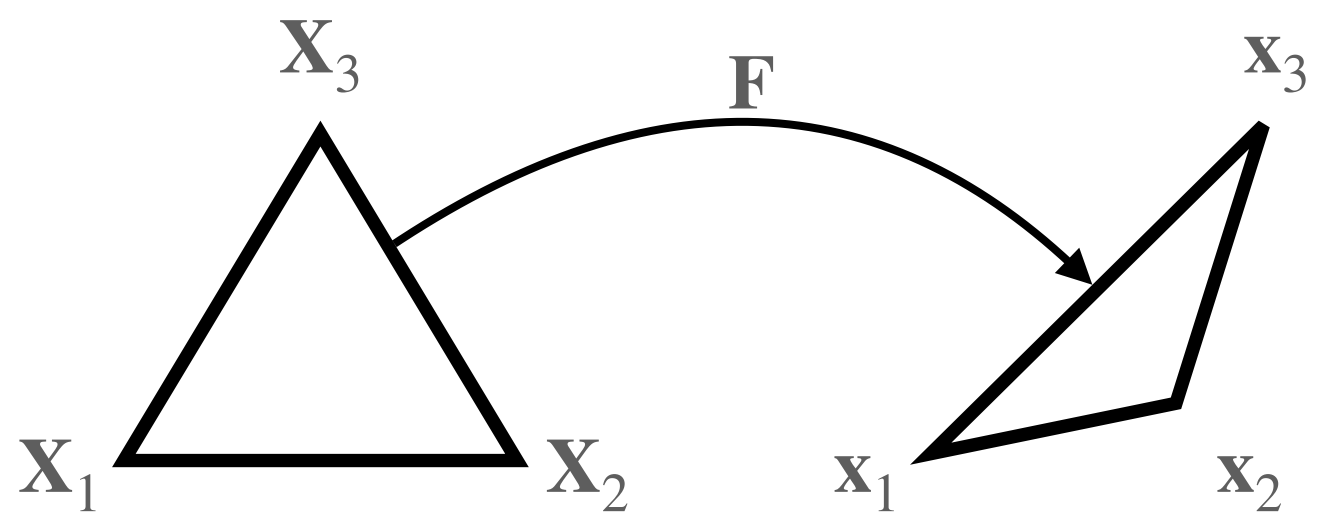
$$\mathbf{F} = [\mathbf{x}_2 - \mathbf{x}_1, \mathbf{x}_3 - \mathbf{x}_1][\mathbf{X}_2 - \mathbf{X}_1, \mathbf{X}_3 - \mathbf{X}_1]^{-1}$$




```python
    # connect the nodes with triangle elements
    e = []
    for i in range(0, n_seg):
        for j in range(0, n_seg):
            # triangulate each cell following a symmetric pattern:
            if (i % 2)^(j % 2):
                e.append([i * (n_seg + 1) + j, (i + 1) * (n_seg + 1) + j, i * (n_seg + 1) + j + 1])
                e.append([(i + 1) * (n_seg + 1) + j, (i + 1) * (n_seg + 1) + j + 1, i * (n_seg + 1) + j + 1])
            else:
                e.append([i * (n_seg + 1) + j, (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, i * (n_seg + 1) + j + 1])
```

square_mesh.py

Elasticity Gradient and Hessian per Triangle



$$\mathbf{x}_2 - \mathbf{x}_1 = \mathbf{F}(\mathbf{X}_2 - \mathbf{X}_1) \quad \text{and} \quad \mathbf{x}_3 - \mathbf{x}_1 = \mathbf{F}(\mathbf{X}_3 - \mathbf{X}_1)$$

$$\mathbf{F} = [\mathbf{x}_2 - \mathbf{x}_1, \mathbf{x}_3 - \mathbf{x}_1][\mathbf{X}_2 - \mathbf{X}_1, \mathbf{X}_3 - \mathbf{X}_1]^{-1}$$

$$\frac{\partial[\mathbf{F}_{11}, \mathbf{F}_{21}, \mathbf{F}_{12}, \mathbf{F}_{22}]^T}{\partial[\mathbf{x}_1^T, \mathbf{x}_2^T, \mathbf{x}_3^T]^T}$$
$$= \begin{bmatrix} -\boldsymbol{B}_{11} - \boldsymbol{B}_{21} & & \boldsymbol{B}_{11} & & \boldsymbol{B}_{21} & \\ & -\boldsymbol{B}_{11} - \boldsymbol{B}_{21} & & \boldsymbol{B}_{11} & & \boldsymbol{B}_{21} \\ -\boldsymbol{B}_{12} - \boldsymbol{B}_{22} & & \boldsymbol{B}_{12} & & \boldsymbol{B}_{22} & \\ & -\boldsymbol{B}_{12} - \boldsymbol{B}_{22} & & \boldsymbol{B}_{12} & & \boldsymbol{B}_{22} \end{bmatrix} \in \mathbb{R}^{4\times 6}$$

$$\boldsymbol{B} = [\mathbf{X}_2 - \mathbf{X}_1, \mathbf{X}_3 - \mathbf{X}_1]^{-1}$$

Elasticity Gradient: $\dfrac{\partial \Psi}{\partial x} = \dfrac{\partial \Psi}{\partial F}\dfrac{\partial F}{\partial x}$

Elasticity Hessian: $\dfrac{\partial^2 \Psi}{\partial x^2} = (\dfrac{\partial F}{\partial x})^T \dfrac{\partial^2 \Psi}{\partial F^2}\dfrac{\partial F}{\partial x}$

Elasticity Gradient and Hessian per Triangle

Implementation

Elasticity Gradient: $$\frac{\partial \Psi}{\partial x} = \frac{\partial \Psi}{\partial F}\frac{\partial F}{\partial x}$$

```python
def dPsi_div_dx(P, IB):  # applying chain-rule, dPsi_div_dx = dPsi_div_dF * dF_div_dx
    dPsi_dx_2 = P[0, 0] * IB[0, 0] + P[0, 1] * IB[0, 1]
    dPsi_dx_3 = P[1, 0] * IB[0, 0] + P[1, 1] * IB[0, 1]
    dPsi_dx_4 = P[0, 0] * IB[1, 0] + P[0, 1] * IB[1, 1]
    dPsi_dx_5 = P[1, 0] * IB[1, 0] + P[1, 1] * IB[1, 1]
    return [np.array([-dPsi_dx_2 - dPsi_dx_4, -dPsi_dx_3 - dPsi_dx_5]), np.array([dPsi_dx_2, dPsi_dx_3]), np.array([dPsi_dx_4, dPsi_dx_5])]

```

NeoHookeanEnergy.py

Elasticity Hessian: $$\frac{\partial^2 \Psi}{\partial x^2} = (\frac{\partial F}{\partial x})^T \frac{\partial^2 \Psi}{\partial F^2}\frac{\partial F}{\partial x}$$

```python
def d2Psi_div_dx2(dP_div_dF, IB):  # applying chain-rule, d2Psi_div_dx2 = dF_div_dx^T * d2Psi_div_dF2 * dF_div_dx (note that d2F_div_dx2 = 0)
    intermediate = np.array([[0.0, 0.0, 0.0, 0.0]] * 6)
    for colI in range(0, 4):
        _000 = dP_div_dF[0, colI] * IB[0, 0]
        _010 = dP_div_dF[0, colI] * IB[1, 0]
        _101 = dP_div_dF[2, colI] * IB[0, 1]
        _111 = dP_div_dF[2, colI] * IB[1, 1]
        _200 = dP_div_dF[1, colI] * IB[0, 0]
        _210 = dP_div_dF[1, colI] * IB[1, 0]
        _301 = dP_div_dF[3, colI] * IB[0, 1]
        _311 = dP_div_dF[3, colI] * IB[1, 1]
        intermediate[2, colI] = _000 + _101
        intermediate[3, colI] = _200 + _301
        intermediate[4, colI] = _010 + _111
        intermediate[5, colI] = _210 + _311
        intermediate[0, colI] = -intermediate[2, colI] - intermediate[4, colI]
        intermediate[1, colI] = -intermediate[3, colI] - intermediate[5, colI]
    result = np.array([[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]] * 6)
    for colI in range(0, 6):
        _000 = intermediate[colI, 0] * IB[0, 0]
        _010 = intermediate[colI, 0] * IB[1, 0]
        _101 = intermediate[colI, 2] * IB[0, 1]
        _111 = intermediate[colI, 2] * IB[1, 1]
        _200 = intermediate[colI, 1] * IB[0, 0]
        _210 = intermediate[colI, 1] * IB[1, 0]
        _301 = intermediate[colI, 3] * IB[0, 1]
        _311 = intermediate[colI, 3] * IB[1, 1]
        result[2, colI] = _000 + _101
        result[3, colI] = _200 + _301
        result[4, colI] = _010 + _111
        result[5, colI] = _210 + _311
        result[0, colI] = -_000 - _101 - _010 - _111
        result[1, colI] = -_200 - _301 - _210 - _311
    return result
```

Elasticity Gradient and Hessian

Summation Over All Triangles

NeoHookeanEnergy.py

```python
def val(x, e, vol, IB, mu, lam):
    sum = 0.0
    for i in range(0, len(e)):
        F = deformation_grad(x, e[i], IB[i])
        sum += vol[i] * Psi(F, mu[i], lam[i])
    return sum

```

```python
def grad(x, e, vol, IB, mu, lam):
    g = np.array([[0.0, 0.0]] * len(x))
    for i in range(0, len(e)):
        F = deformation_grad(x, e[i], IB[i])
        P = vol[i] * dPsi_div_dF(F, mu[i], lam[i])
        g_local = dPsi_div_dx(P, IB[i])
        for j in range(0, 3):
            g[e[i][j]] += g_local[j]
    return g

def hess(x, e, vol, IB, mu, lam):
    IJV = [[0] * (len(e) * 36), [0] * (len(e) * 36), np.array([0.0] * (len(e) * 36))]
    for i in range(0, len(e)):
        F = deformation_grad(x, e[i], IB[i])
        dP_div_dF = vol[i] * d2Psi_div_dF2(F, mu[i], lam[i])
        local_hess = d2Psi_div_dx2(dP_div_dF, IB[i])
        for xI in range(0, 3):
            for xJ in range(0, 3):
                for dI in range(0, 2):
                    for dJ in range(0, 2):
                        ind = i * 36 + (xI * 3 + xJ) * 4 + dI * 2 + dJ
                        IJV[0][ind] = e[i][xI] * 2 + dI
                        IJV[1][ind] = e[i][xJ] * 2 + dJ
                        IJV[2][ind] = local_hess[xI * 2 + dI, xJ * 2 + dJ]
    return IJV
```

Inversion-Free Line Search Filtering

Derivation

$V(\mathbf{x}_i + \alpha^I \boldsymbol{p}_i) = 0$

For all triangle, find α^I, and then take their minimum.

$$\det([\mathbf{x}^{\alpha}_{21}, \mathbf{x}^{\alpha}_{31}]) \equiv \mathbf{x}^{\alpha}_{21,1}\mathbf{x}^{\alpha}_{31,2} - \mathbf{x}^{\alpha}_{21,2}\mathbf{x}^{\alpha}_{31,1} = 0$$

$$\text{with } \mathbf{x}^{\alpha}_{ij} = \mathbf{x}_{ij} + \alpha^I \boldsymbol{p}_{ij} \text{ and } \mathbf{x}_{ij} = \mathbf{x}_i - \mathbf{x}_j,\ \boldsymbol{p}_{ij} = \boldsymbol{p}_i - \boldsymbol{p}_j$$

$$\frac{\det([\boldsymbol{p}_{21}, \boldsymbol{p}_{31}])}{\det([\mathbf{x}_{21}, \mathbf{x}_{31}])}(\alpha^I)^2 + \frac{\det([\mathbf{x}_{21}, \boldsymbol{p}_{31}]) + \det([\boldsymbol{p}_{21}, \mathbf{x}_{31}])}{\det([\mathbf{x}_{21}, \mathbf{x}_{31}])}\alpha^I + 1 = 0$$

Inversion-Free Line Search Filtering

Implementation

$$\frac{\det([\boldsymbol{p}_{21}, \boldsymbol{p}_{31}])}{\det([\mathbf{x}_{21}, \mathbf{x}_{31}])}(\alpha^I)^2 + \frac{\det([\mathbf{x}_{21}, \boldsymbol{p}_{31}]) + \det([\boldsymbol{p}_{21}, \mathbf{x}_{31}])}{\det([\mathbf{x}_{21}, \mathbf{x}_{31}])}\alpha^I + 1 = 0$$

```
def init_step_size(x, e, p):
    alpha = 1
    for i in range(0, len(e)):
        x21 = x[e[i][1]] - x[e[i][0]]
        x31 = x[e[i][2]] - x[e[i][0]]
        p21 = p[e[i][1]] - p[e[i][0]]
        p31 = p[e[i][2]] - p[e[i][0]]
        detT = np.linalg.det(np.transpose([x21, x31]))
        a = np.linalg.det(np.transpose([p21, p31])) / detT
        b = (np.linalg.det(np.transpose([x21, p31])) + np.linalg.det(np.transpose([p21, x31]))) / detT
        c = 0.9  # solve for alpha that first brings the new volume to 0.1x the old volume for slackness
        critical_alpha = utils.smallest_positive_real_root_quad(a, b, c)
        if critical_alpha > 0:
            alpha = min(alpha, critical_alpha)
    return alpha
```

```
def smallest_positive_real_root_quad(a, b, c, tol = 1e-6):
    # return negative value if no positive real root is found
    t = 0
    if abs(a) <= tol:
        if abs(b) <= tol: # f(x) = c > 0 for all x
            t = -1
        else:
            t = -c / b
    else:
        desc = b * b - 4 * a * c
        if desc > 0:
            t = (-b - math.sqrt(desc)) / (2 * a)
            if t < 0:
                t = (-b + math.sqrt(desc)) / (2 * a)
        else: # desv<0 ==> imag, f(x) > 0 for all x > 0
            t = -1
    return t
```

```
        alpha = min(BarrierEnergy.init_step_size(x, n, o, p), NeoHookeanEnergy.init_step_size(x, e, p))  # avoid interpenetration, tunneling, and inversion
```

Demo

github.com/liminchen/solid-sim-tutorial /6_inv_free

Next Lecture: Governing Equations

$$R(\mathbf{X}, t) J(\mathbf{X}, t) = R(\mathbf{X}, 0) \quad \text{Conservation of mass}$$

$$R(\mathbf{X}, 0) \frac{\partial \mathbf{V}}{\partial t}(\mathbf{X}, t) = \nabla^{\mathbf{X}} \cdot \mathbf{P}(\mathbf{X}, t) + R(\mathbf{X}, 0)\boldsymbol{g} \quad \text{Conservation of momentum}$$

Weak form:

$$\int_{\Omega^0} R^0(\mathbf{X}) Q_i^n(\mathbf{X}) A_i^n(\mathbf{X}) d\mathbf{X}$$
$$= \int_{\partial\Omega^0} Q_i^n(\mathbf{X}) T_i^n(\mathbf{X}) ds(\mathbf{X}) - \int_{\Omega^0} Q_{i,j}^n(\mathbf{X}) P_{ij}^n(\mathbf{X}) d\mathbf{X}$$

This Thursday: Project Proposal Presentation

- 10 ~ 15 minutes presentation + 5 ~ 10 minutes Q&A
- Try to Cover:
  - Problem statement / goals, Related works, Approach, Resources, Evaluation, Timeline (See the Piazza post for details)
- Presentations:
  1. Sarah Di, Olga Guțan, Zoë Marschner
  2. Ruben Partono, Daniel Zeng, Shilin Ma
  3. Kevin You

Image Sources

- https://www.youtube.com/watch?v=WV-J7u9aoHk
- https://en.wikipedia.org/wiki/Stress%E2%80%93strain_curve