

PERSONAL NOTES

NONLINEAR F.E.A.

Hanfeng Zhai

Disclaimer: These notes are intended solely for personal reference and study purposes. They represent my own understanding of the course material and may contain errors or inaccuracies. The content presented here should not be considered as an authoritative source, and reliance solely on these materials is not recommended. If you notice any materials that potentially infringe upon the copyright of others, please contact me at hz253@cornell.edu so that appropriate action can be taken. Your feedback is greatly appreciated.

#FEM (Nonlinear)

▷ What is FEA?

Method to solve PDE.

▷ Why Nonlinear FEA important?

large deformation.

(linearize the strain)

↓
plastic deformation.

↳ material response
hyperelastic, etc. . . .

▷ tools for linear FEA. - relevant?

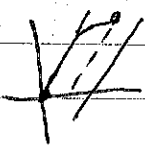
Strong form → weak form

↓
discretize

↓
shape function

* ~~local~~ local support.

Key difference:
the solution



Prescribe the behavior within the element.

← (interpolation)

↳ derivatives

solution ← assembly.
↳ postprocessing.

← Gauss Quadrature
(integration).

Sources of nonlinearities in solids

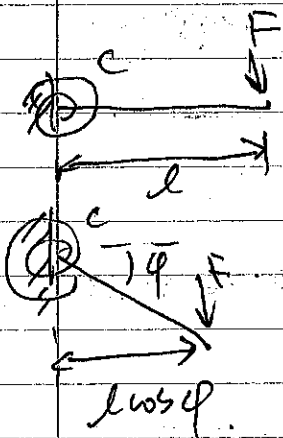
- geometric nonlinearities
- deformation
- material responses
- instabilities
- BCs
- coupled problem
 - force, multiphysics

large deformation (disp.) of rigid beam

⇒ Example:

rigid beam, rotational spring stiffness c

$$\sum M = 0 \rightarrow Fl \cos \varphi = c \varphi$$

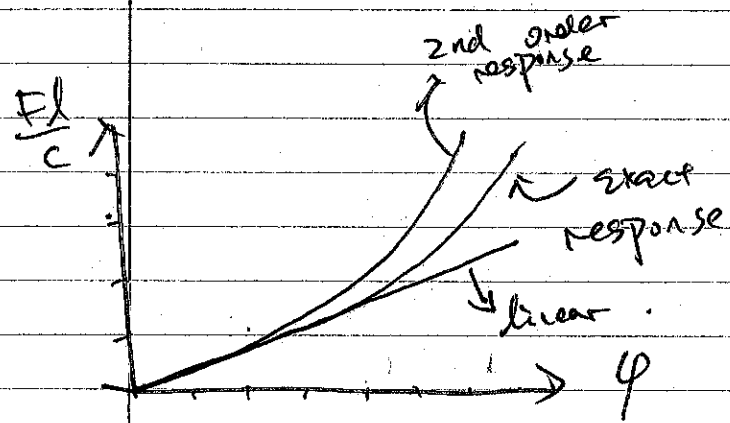


For small rotations $\cos \varphi \rightarrow 1$

$$F = c\varphi/l \quad \varphi \rightarrow 0$$

to capture the nonlinearity, → 2nd order theory
expand the function into Taylor series:

$$\cos \varphi \approx 1 - \frac{\varphi^2}{2}$$



$$F = \frac{c\varphi}{2(1 - \frac{\varphi^2}{2})}$$

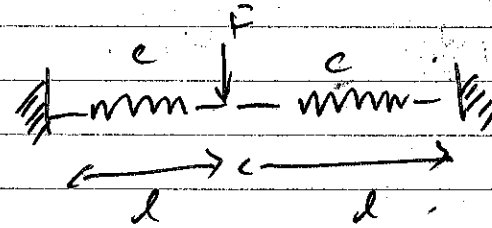
"2nd order"

* geometric nonlinearities

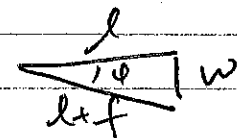
general motion

* this is not deformation - no strain tensor involved

large deformation of elastic system



$$\sum \vec{F} = 0$$



elongation

- kinematics

- equilibrium

- constitutive law

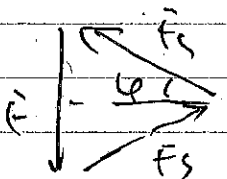
Kinematics

$$w^2 + l^2 = (l+f)^2$$

$$\Rightarrow \sin \varphi = \frac{w}{l+f}$$

Equilibrium

$$\geq F_s \sin \varphi = F$$



Constitutive law:

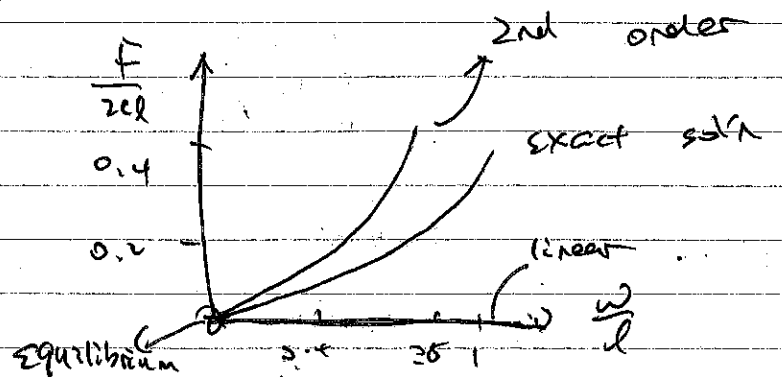
$$F_s = c f \leftarrow \text{elongation}$$

↑ Spring constant

exact sol'n:

$$\frac{w}{l} \left[1 - \frac{1}{\sqrt{1 + \left(\frac{w}{l}\right)^2}} \right] = \frac{F}{2cl}$$

the exact sol'n:



Taylor expansion:

$$\frac{1}{\sqrt{1 + \left(\frac{w}{l}\right)^2}} \approx 1 - \frac{1}{2} \left(\frac{w}{l}\right)^2$$

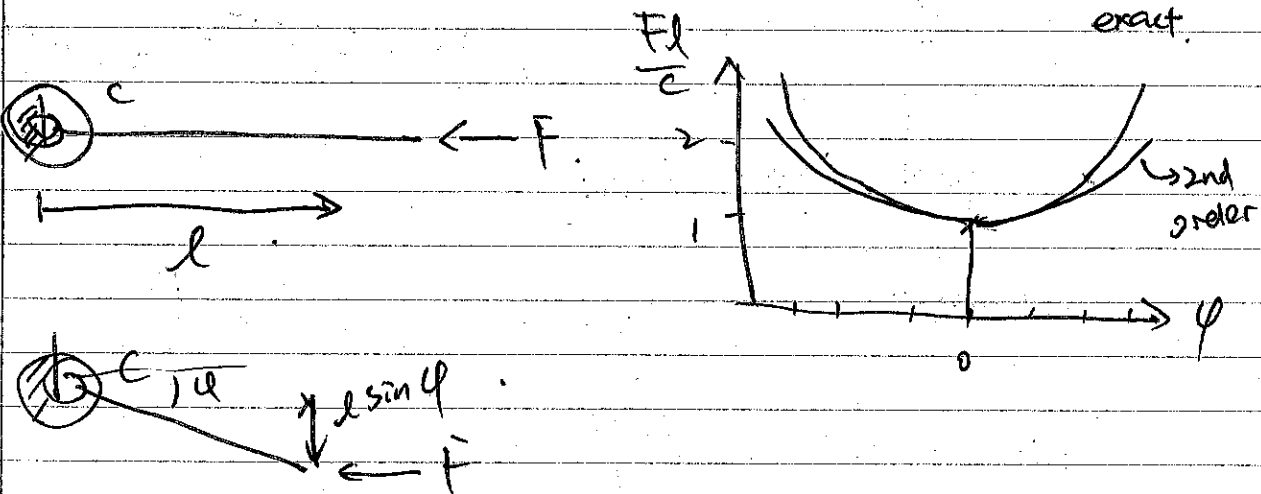
small values of $w \rightarrow \frac{w}{l} \ll 1$

$$\frac{w}{l} \left[\frac{1}{2} \left(\frac{w}{l}\right)^2 \right] = \frac{F}{2cl}$$

source of nonlinearities: deformation & rotation

Bifurcation

$$Fl \sin \varphi = c \varphi$$



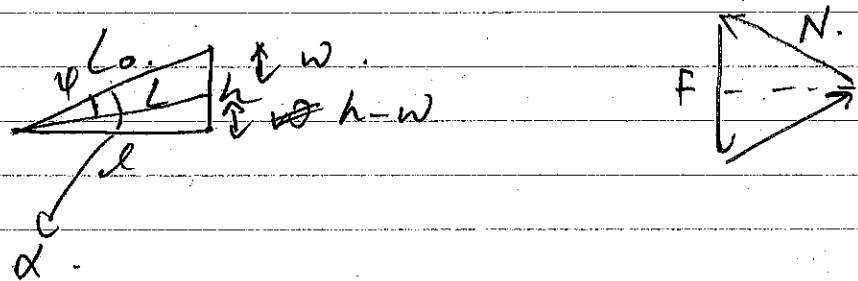
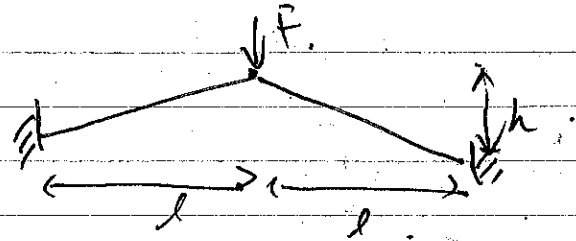
Approximation:

$$\text{Taylor expansion: } \sin \varphi \approx \varphi - \frac{\varphi^3}{6}$$

$$\& \frac{1}{(1-x)} \approx 1+x$$

Snap-through.

Geometry: $(h-w)^2 + l^2 = L^2$ & $L^2 + l^2 = L_0^2$



from this we can write length change:

$$f = L - L_0 = l \left[\sqrt{1 + \left(\frac{h-w}{l}\right)^2} - \sqrt{1 + \left(\frac{h}{l}\right)^2} \right]$$

↓ final length ↑ initial

Equilibrium: $N \sin(\alpha - \varphi) = -\frac{F}{2}$

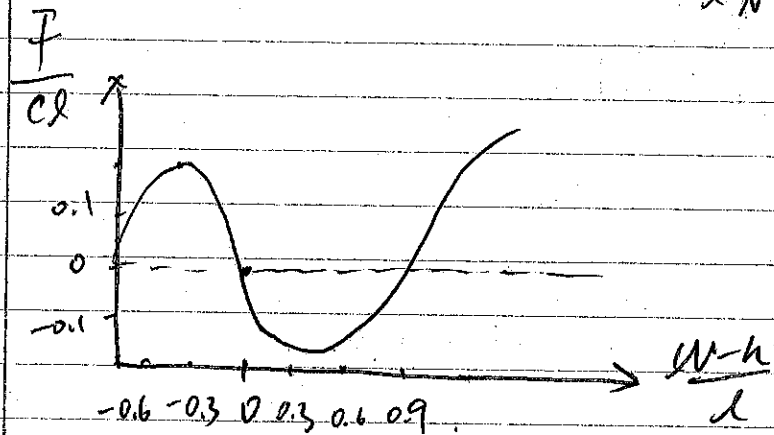
$$\sin(\alpha - \varphi) = \frac{h-w}{L}$$

$$\hookrightarrow N \frac{h-w}{L} = -\frac{F}{2}$$

Constitutive law: $F_s = cf$ → final expression

$$c(h-w) \frac{L-L_0}{L} = -\frac{F}{2} \rightarrow$$

$$\frac{wh}{l} \left[1 - \frac{L_0}{l \sqrt{1 + \left(\frac{h-w}{l}\right)^2}} \right] - \frac{F}{2cl} = u$$



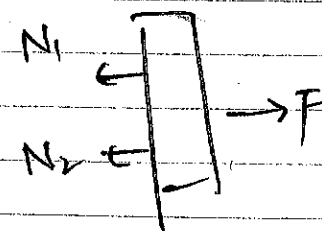
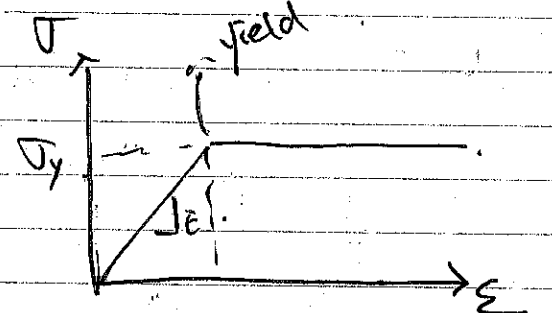
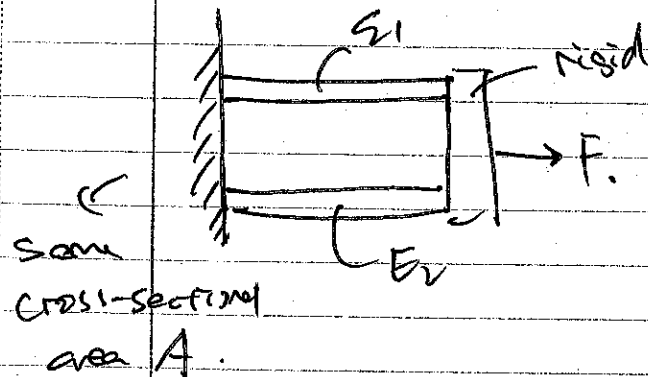
(*) F = applied force

Material nonlinearities

elastic moduli: $E_1 = 2E_2 = 2E$

their yield stresses:

$$\sigma_{y1} = 3\sigma_{y2} = 3\sigma_y$$



$$N_1 + N_2 = F$$

$$\sigma_1 + \sigma_2 = \frac{F}{A}$$

$$u_1 = u_2 = u, \quad \epsilon = \frac{u}{l}$$

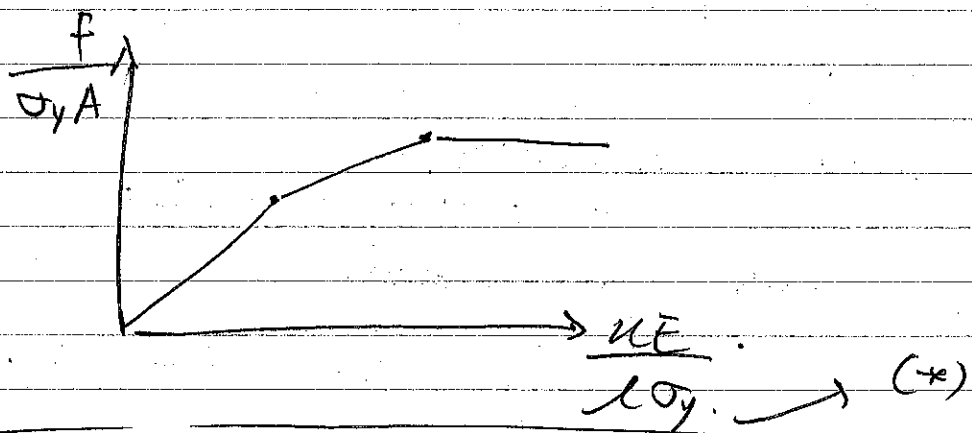
comece thru Hooke's law.

$$\sigma_i = E_i \epsilon = E_i \frac{u}{l}$$

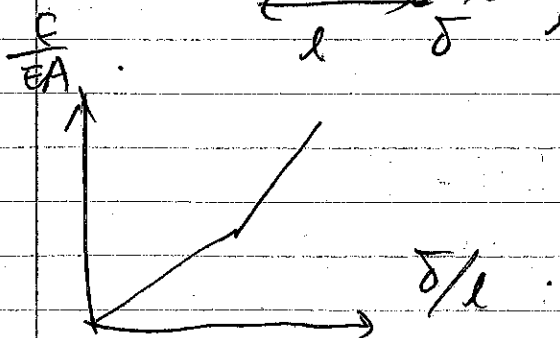
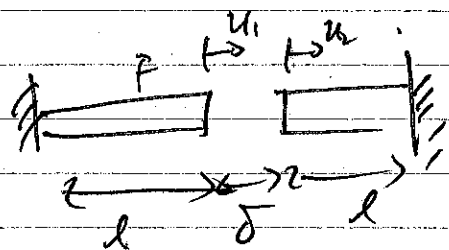
$$u = \frac{Fl}{(E_1 + E_2)A}$$

$$\sigma_2 < \sigma_1$$

Bar 2 yields $\rightarrow F = 3A\sigma_y$



BC induced nonlinearity



HW: notes

$$\frac{\partial (A_{ik}^{-1} A_{kj})}{\partial A_m} = -A_{ik}^{-1} \frac{\partial A_{ik}}{\partial A_m} A_{kj}^{-1} + A_{ik}^{-1} \frac{\partial A_{kj}}{\partial A_m} A_{kj}$$

$$\rightarrow \frac{\partial A^{-1}}{\partial A} \quad \text{hint: } A_{ik}^{-1} A_{kj} = \delta_{ij}$$

$$\frac{\partial A^{-1} A}{\partial A_{kj}} = \frac{\partial A_{ik}^{-1}}{\partial A_{kj}} A_{ik} + A_{ik}^{-1} \frac{\partial A_{kj}}{\partial A_{kj}} = \delta_{ij}$$

$$\frac{\partial A_{ij}^{-1}}{\partial A_{pq}} = -A_{ip}^{-1} A_{jq}^{-1} \quad \frac{\partial (ab)}{\partial c} = \frac{\partial a}{\partial c} b + \frac{\partial b}{\partial c} a$$

$$\frac{\partial \det A}{\partial A} = \det(A) \cdot A^{-T}$$

$$\frac{\partial \det(A_{ij})}{\partial A_{kl}} =$$

Prob 1. $\frac{\partial (A_{ik}^{-1} A_{kj})}{\partial A_{pq}}$

$$= \frac{\partial A_{ik}^{-1}}{\partial A_{pq}} A_{kj} + \frac{\partial A_{kj}}{\partial A_{pq}} A_{ik}^{-1}$$

$\frac{\partial \delta_{ij}}{\partial A_{pq}} = \frac{\partial A_{kj}}{\partial A_{pq}}$

$\frac{\partial A_{ik}^{-1}}{\partial A_{pq}} A_{kj} = -\frac{\partial A_{kj}}{\partial A_{pq}} A_{ik}^{-1}$
 $\frac{\partial A_{ik}^{-1}}{\partial A_{pq}} = \dots A_{kj}^{-1}$

$A_{ik} A_{kj}^{-1} = \delta_{ij}$

$$\frac{\partial A_{ij}^{-1}}{\partial A_{pq}} A_{jk} + A_{ij}^{-1} \frac{\partial A_{jk}}{\partial A_{pq}} = \frac{\partial A_{ik}^{-1}}{\partial A_{pq}} A_{kk} + A_{ik}^{-1} \frac{\partial A_{kk}}{\partial A_{pq}}$$

$$\frac{\partial A_{ij}^{-1}}{\partial A_{pq}} = -A_{ik}^{-1} \frac{\partial A_{kk}}{\partial A_{pq}} A_{kj}^{-1}$$

$$= -A_{ik}^{-1} \delta_{kp} \delta_{kq} A_{kj}^{-1}$$

$$= -A_{ip}^{-1} A_{qj}^{-1}$$

Prob. 2

$$\frac{\partial \det(A)}{\partial A} = \frac{\partial \det(A)}{\partial A_{ij}} = \frac{\partial \|A_{ij}\|}{\partial A_{ij}}$$

$$= \sum \text{sgn} \Pi a$$

direct notation:

$$\det A = \begin{vmatrix} a_{11} & \dots \\ \dots & \dots \end{vmatrix}$$

Invariant:

$$\det A = \|A_{ij}\|$$

$$\sum_i \sum_j \frac{\partial A}{\partial A_{ij}} \cdot \frac{\partial A_{ki}}{\partial A_{ij}}$$

$$\begin{cases} \frac{\partial Z_1}{\partial A} = I \\ \frac{\partial Z_2}{\partial A} = Z_1 I - A^T \\ \frac{\partial Z_3}{\partial A} = (A^T)^T - Z_2 A^T \end{cases}$$

$$\frac{\partial \det A}{\partial A_{ij}} = \sum_k \sum_l C_{kl} \delta_{ik} \delta_{jl} = Z_3 A^T$$

$$\frac{\partial \det A}{\partial A_{ij}} = \sum_k \sum_l \det(A)$$

$$f(A) = \det(A)$$

$$A_{11} A_{22} - A_{12} A_{21}$$

$$\det(A) = \frac{1}{6} \epsilon_{ijk} \epsilon_{abc} A_{ia} A_{jb} A_{kc}$$

$$\frac{\partial \frac{1}{6} \sum_{ijk} \epsilon_{ijk} \det A_{ij} A_{jk} A_{ki}}{\partial A_{ij}}$$

$$\frac{\partial \det A_{ij}}{\partial A_{ij}} = (A_{ij}^T)^2 - \mathbb{I} [A_{ij}^T + \mathbb{I} \mathbb{I}]$$

$$= (A_{ij}^T)^2 - \det \operatorname{tr} A_{ij} A_{ji}$$

$$+ \frac{1}{2} [(\operatorname{tr} A_{ij})^2 - \operatorname{tr} A_{ij}^2]$$

$$L_3 = \det A_{ij}$$

$$= A_{ji}^2 - \operatorname{tr} A_{ij} A_{ji}$$

$$L_1 = \operatorname{tr} A_{ij}$$

$$+ \frac{1}{2} [\dots]$$

$$L_2 = \frac{1}{2} [(\operatorname{tr} A)^2 - \operatorname{tr} (A^2)]$$

$$= \det A_{ij} A_{ji}^{-1}$$

$$= A_{ji}^2 - \operatorname{tr} A_{ij} A_{ji} + \frac{1}{2} [(\operatorname{tr} A_{ij})^2 - \operatorname{tr} A_{ij}^2]$$

$$\frac{\partial (A_{11} A_{22} - A_{12} A_{21})}{\partial A_{ij}} = A_{22} \quad A_{11}$$

∂

Prob. 3

$$\underline{D} = \underline{a} \otimes \underline{b}$$

$$= \begin{bmatrix} a_1 b_1 & a_1 b_2 & a_1 b_3 \\ a_2 b_1 & a_2 b_2 & a_2 b_3 \\ \dots & \dots & \dots \end{bmatrix}$$

$$\det(\underline{A}) \uparrow \frac{\partial (\underline{A}^T \partial \underline{A})}{\partial \underline{A}}$$

$$\frac{\partial \det(A)}{\partial A} = \lim_{h \rightarrow 0} \frac{\det(A+h) - \det A}{h}$$

$$= \det A \lim_{h \rightarrow 0} \frac{\det [A^{-1}(A+h)] - 1}{h}$$

$$= \det$$

$$\det(\partial A^T) = \det A$$

Prob 4. $\underline{A} = \alpha(\underline{I} - \underline{e}_1 \otimes \underline{e}_1) + \beta(\underline{e}_1 \otimes \underline{e}_2 + \underline{e}_2 \otimes \underline{e}_1)$

compute eigen values

- eigenvectors

we can first write out the matrix form

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} \alpha & \beta \\ \beta & \alpha \end{bmatrix}$$

Prob 5. - vector form

$$\oint_C \underline{\nabla} \times \underline{F} d\underline{s} = \iint_S \text{curl } \underline{F} d\underline{S}$$

$$\iint_S \text{curl } \underline{F} d\underline{S} = \iiint_V \text{div}(\text{curl } \underline{F}) dV$$

$$\therefore \text{curl}(\text{curl } \underline{F}) = \nabla(\nabla \cdot \underline{F}) - \nabla^2 \underline{F}$$

$$\iint_S \text{div}(\text{curl } \underline{F}) dV = \iiint_V [\nabla(\nabla \cdot \underline{F}) - \nabla^2 \underline{F}] dV$$

$$\iiint_V [\nabla(\nabla \cdot \underline{F}) - \nabla^2 \underline{F}] dV = \iint_S (\nabla \cdot \underline{F}) \underline{n} dS - \iint_S \underline{F} \cdot \underline{n} dS$$

$$\oint_C \underline{\nabla} \times \underline{F} d\underline{s} = \iint_S (\nabla \cdot \underline{F}) \underline{n} dS - \iint_S \underline{F} \cdot \underline{n} dS$$

Prob. 5

$$\oint_C \phi d\underline{x} = \int_S \underline{n} \times \text{grad } \phi dS$$

$$\oint_C \underline{u} \times d\underline{x} = \int_S [(\text{div } \underline{u}) \underline{n} - (\text{grad } \underline{u}) \cdot \underline{n}] dS$$

$$\oint_C \phi d\underline{x} = \int_S \underline{n} \times \text{grad } \phi dS$$

$$\oint_C \underline{u} \times d\underline{x}$$

$$\int_C \underline{A} \cdot d\underline{x} = \int_S \underline{A} \times d\underline{S}$$

$$\oint_C \phi d\underline{x} = \int_S \underline{n} \times \text{grad } \phi$$

Proof for 1:

$$\int_C \phi d\underline{x} = \int_C d(\underline{x} \cdot \text{grad } \phi)$$

$$= \int_C d\underline{x} \times \text{grad } \phi dS$$

$$= \int_S \underline{n} \times \text{grad } \phi dS$$

$d(\underline{x} \cdot \text{grad } \phi)$
 $d\underline{x} \cdot \text{grad } \phi$
 scalar $\times \underline{x}$

Proof for 2:

$$\int_C \underline{u} \times d\underline{x} = \int_C d(\underline{x} \times \underline{u})$$

$$= \int_S d \underline{x} \times \underline{u} dS$$

$$= \int_S [\text{div } \underline{u} \cdot \underline{n} - \text{grad} \cdot (\underline{u} \otimes \underline{n})] dS$$

Gauss's Theorem

$$\int_S \underline{\phi} \cdot d\underline{S} = \int_V \text{div } \underline{\phi} dV$$

$$\int_C \underline{\phi} d\underline{x} = \int_S \underline{n} \times \text{grad } \phi dS$$

Consider efflux $dE = \underline{\phi} \cdot d\underline{S}$
 $\text{div } \underline{\phi} dV = dE = \underline{\phi} \cdot d\underline{S}$

Stokes' Theorem

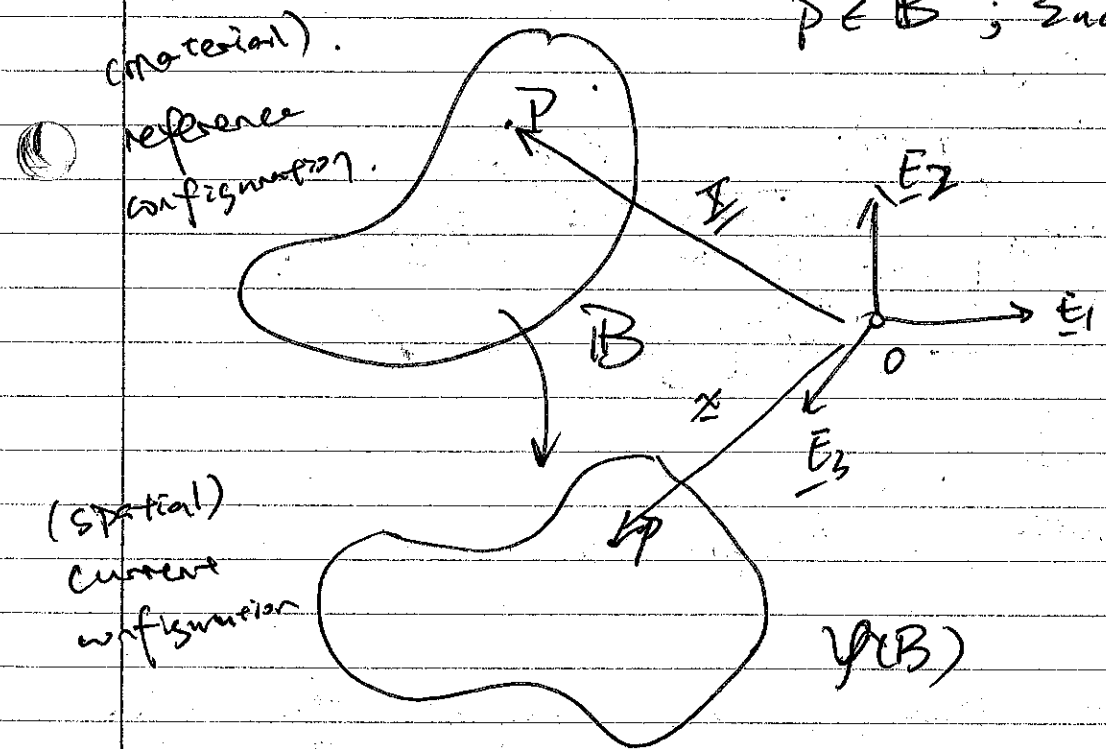
$$\oint_C \underline{a} \cdot d\underline{I} = \int_S \text{curl } \underline{a} \cdot d\underline{S}$$

$$\sum_{\text{de loop}} \underline{a} \cdot d\underline{I} = (\nabla \times \underline{a}) \cdot d\underline{S}$$

Lecture 2. Continuum Mechanics - important

- Kinematics (motion)
- Strain measures.
- vector & tensor transformation
- Balance equations.
- stress measures
- Constitutive relations.

$P \in B$; Euclidean space \mathbb{R}^3

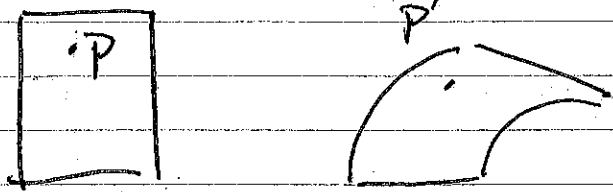


configuration $\varphi: B \rightarrow \mathbb{R}^3$

$$\underline{x} = \varphi(\underline{X}, t)$$

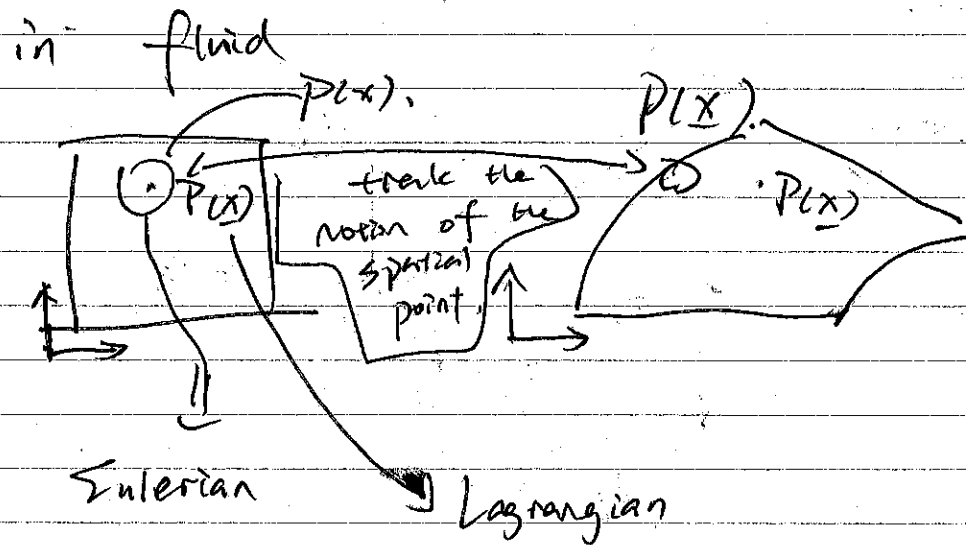
$\underline{x}, \underline{X}$ - positions in \mathbb{R}^3
 w/ origin 0

$$\underline{X} =$$

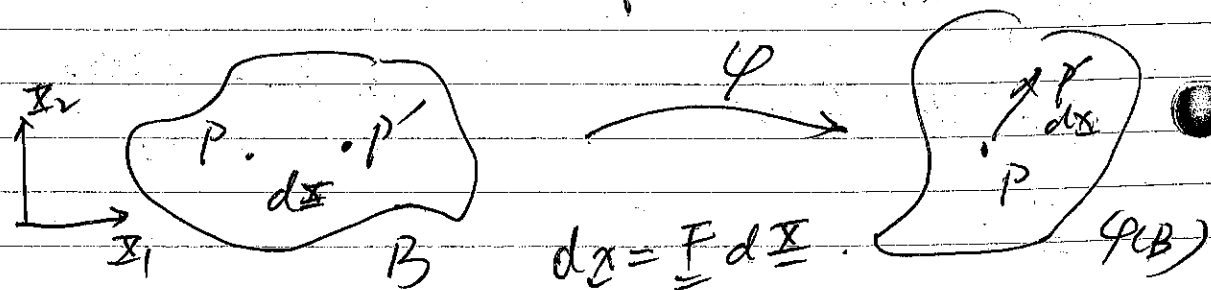


$$\underline{x} = \underline{x}_0 + \underline{\epsilon}A$$

in linear FEM - Lagrangian
but does not make a diff.



- How line elements mapped from reference to current configs.



$$\underline{F} = \frac{\partial \underline{x}}{\partial \underline{X}} \rightarrow \frac{\partial x_i}{\partial X_A} = F_{iA}$$

$$\underline{F} = \text{Grad } \varphi(\underline{X}, t) \rightarrow \underline{x} = \varphi(\underline{X})$$

\underline{F} tensor vs. matrix
 ↓ preserves basis properties
 ↓ numerical for n.

$$\underline{A} = \underline{a} \otimes \underline{b}$$

$$\underline{a} = a_i \underline{e}_i$$

$$\underline{b} = b_j \underline{e}_j$$

$$[A] = [\dots]$$

$$dx_1 = F_{11} dX_1 + F_{12} dX_2 + F_{13} dX_3$$

$$[F_{iA}] = \begin{bmatrix} \frac{\partial x_1}{\partial X_1} & \frac{\partial x_1}{\partial X_2} & \dots \\ \dots & \dots & \dots \\ \frac{\partial x_3}{\partial X_1} & \dots & \dots \end{bmatrix}$$

(cont'd)

Kinematics — deformation gradients.

\underline{F} has to be invertible.

$$d\underline{X} = \underline{F}^{-1} d\underline{x}$$

$$\underline{F}^{-1} = (F^{-1})_A \underline{E}_A \otimes \underline{e}_i$$

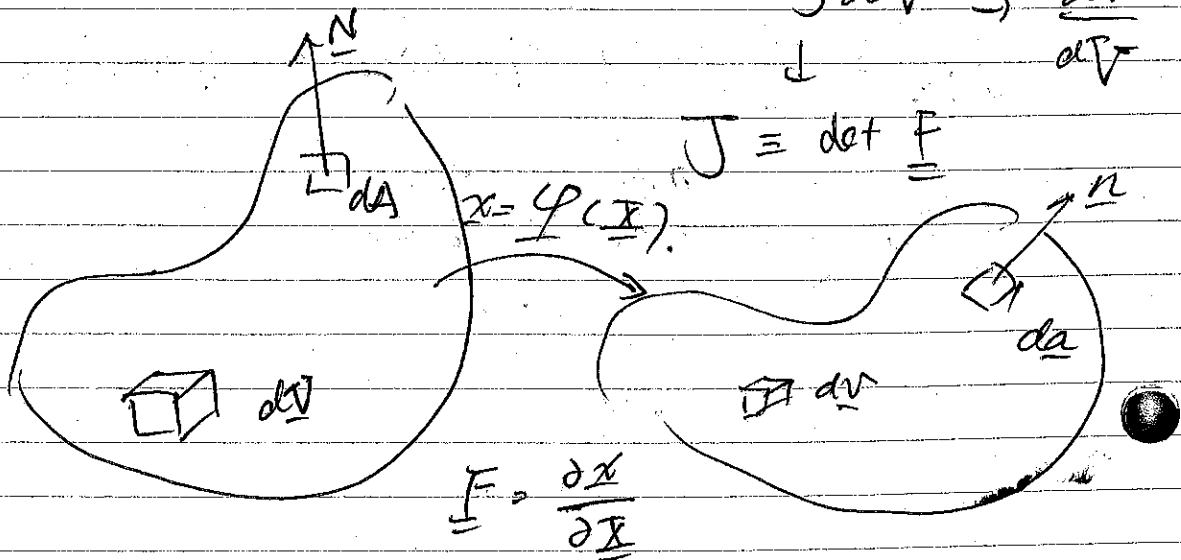
$$\hookrightarrow \frac{\partial X_A}{\partial x_i}$$

* Nanson's formula. Capture transformation of surface element in ref. config. dA

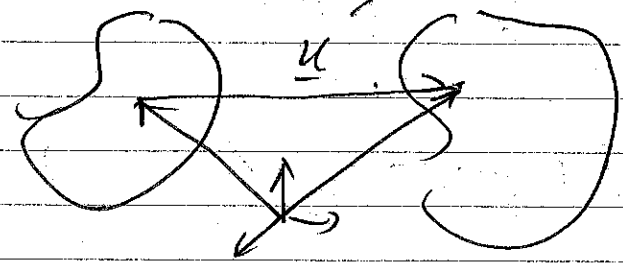
$$d\underline{a} = \underline{n} d\underline{a} = \underline{J} \underline{F}^{-T} \underline{N} dA = \underline{J} \underline{F}^{-T} d\underline{A}$$

$$dV = \underline{J} dV \rightarrow \frac{dV}{dV}$$

$$\underline{J} \equiv \det \underline{F}$$



— Define the displacement \rightarrow unknown.



$$\underline{u}(\underline{X}, t) = \underline{\varphi}(\underline{X}, t) - \underline{X} = \underline{x} - \underline{X}$$

$$\underline{F} = \text{Grad} [\underline{X} + \underline{u}(\underline{X}, t)]$$

$$= \underline{I} + \text{Grad} \underline{u} = \underline{I} + \underline{H}$$

↓ displacement gradient

$$* \text{Grad} = \nabla_{\underline{X}}$$

— material gradient operation

$$\text{grad} = \nabla_{\underline{x}}$$

— spatial gradient operation

$$\underline{I} = \delta_{ij} \underline{E}_i \otimes \underline{E}_j$$

— Strain measures

Green-Lagrange strain tensor

Right Cauchy-Green strain tensor

$$\underline{E} \equiv \frac{1}{2} (\underline{F}^T \underline{F} - \underline{I}) = \frac{1}{2} (\underline{C} - \underline{I})$$

high-order terms
get rid of

$$\underline{\underline{E}} = \frac{1}{2} (\underline{\underline{H}}^T + \underline{\underline{H}} + \underline{\underline{H}}^T \underline{\underline{H}})$$

$$\underline{\underline{\zeta}} = \frac{1}{2} (\underline{\underline{H}} + \underline{\underline{H}}^T) = \frac{1}{2} (\underline{\underline{u}})$$

Define rotation $\underline{\underline{R}}$ ($\underline{\underline{R}}^{-1} = \underline{\underline{R}}^T$) &

corresponding symmetric stretch tensors

$$\underline{\underline{U}} \ \& \ \underline{\underline{V}} : \quad \underline{\underline{F}} = \underline{\underline{R}} \underline{\underline{U}} = \underline{\underline{V}} \underline{\underline{R}}$$

$$\begin{cases} F_{iB} \underline{e}_i \otimes \underline{E}_B = (R_{iA} \underline{e}_i \otimes \underline{E}_A) (U_{\alpha B} \underline{E}_\alpha \otimes \underline{E}_B) \\ F_{iB} \underline{e}_i \otimes \underline{E}_B = (V_{iK} \underline{e}_i \otimes \underline{e}_K) (R_{mB} \underline{e}_m \otimes \underline{E}_B) \end{cases}$$

generalized strain measure:

$$\underline{\underline{E}}^\alpha = \frac{1}{2} (\underline{\underline{U}}^\alpha - \underline{\underline{I}}) \ \& \ \underline{\underline{e}} = \frac{1}{2} (\underline{\underline{V}}^\alpha - \underline{\underline{I}})$$

$$\underline{\underline{E}} = \frac{1}{2} (\underline{\underline{U}}^2 - \underline{\underline{I}})$$

$$\underline{\underline{U}} = \sum_i \lambda_i \underline{N}_i \otimes \underline{N}_i ; \quad \underline{\underline{V}} = \sum_i \lambda_i \underline{n}_i \otimes \underline{n}_i$$

$$\underline{n}_i = \underline{\underline{R}} \underline{N}_i$$

- Strain measures.

Almansi strain tensor.

$$\underline{\underline{e}} = \underline{\underline{e}}^{(-2)} = \frac{1}{2} (\underline{\underline{I}} - \underline{\underline{V}}^{-2}) = \frac{1}{2} (\underline{\underline{I}} - \underline{\underline{b}}^{-1}) = \frac{1}{2} (\underline{\underline{I}} - \underline{\underline{F}}^T \underline{\underline{F}}^{-1})$$

$$\underline{\underline{b}} \equiv \underline{\underline{F}} \underline{\underline{F}}^T = \underline{\underline{V}} \underline{\underline{R}} \underline{\underline{R}}^T \underline{\underline{V}}^T = \underline{\underline{V}}^2$$

* pull back ←
push forward →

$$G(\underline{\underline{X}}) = g(\underline{\underline{x}})$$

$$\text{Grad } G = \underline{\underline{F}}^T \text{grad } g \Leftrightarrow \frac{\partial G}{\partial X_A} = \frac{\partial g}{\partial x_i} \frac{\partial x_i}{\partial X_A}$$

$$\text{grad } g = \underline{\underline{F}}^{-T} \text{Grad } G$$

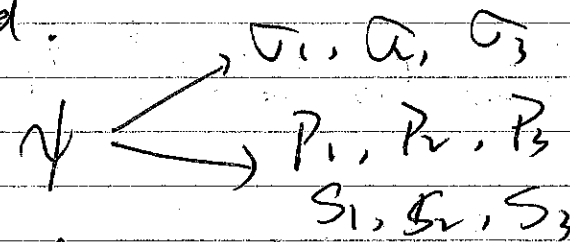
$$\frac{\partial g}{\partial x_i} = \frac{\partial g}{\partial X_A} \frac{\partial X_A}{\partial x_i}$$

pull back
+
push forward

Vector & tensor transformation:

HW continued.

Prob. 9.



Strain Energy function $\Psi \rightarrow \lambda_1, \lambda_2, \lambda_3$.

$$\frac{\partial \Psi}{\partial \lambda_i} = \sigma_i \lambda_i, \quad i=1, 2, 3.$$

$$P_i = \frac{\partial \Psi}{\partial \lambda_i} \quad \sigma_i = \lambda_i P_i.$$

$$\underline{\underline{S}} = 2\rho_0 \frac{\partial \Psi(\underline{\underline{C}})}{\partial \underline{\underline{C}}}$$

$$\underline{\underline{S}} = \underline{\underline{F}}^{-1} \underline{\underline{P}} = J \underline{\underline{F}}^{-1} \underline{\underline{\sigma}} \underline{\underline{F}}^{-T}.$$

Ψ — incompressible isotropic hyperelastic.

i.t.o. \checkmark principal stretches.

$$\Psi = \frac{1}{2} k (\lambda_1^2 + \lambda_2^2 + \lambda_3^2 - 3)$$

$$+ \frac{k}{\beta} (\lambda_1 \lambda_2 \lambda_3)^\beta$$

\Downarrow
3 Cauchy stress.

$$\begin{cases} \sigma_1 = k \lambda_1 (\lambda_1^2 + \lambda_2^2 + \lambda_3^2 - 3) + (\lambda_1 \lambda_2 \lambda_3)^\beta k \beta \lambda_1 \\ \sigma_2 = k \lambda_2 (\quad \quad \quad) + \quad \quad \quad \lambda_2 \\ \sigma_3 = k \lambda_3 (\quad \quad \quad) + \quad \quad \quad \lambda_3 \end{cases}$$

$$\Psi = (I_1, I_2, I_3) = \frac{\mu}{\nu} (I_1 - 3) + \frac{\lambda}{\nu} (I_1 - 3)^2$$

$$\Psi = \frac{\mu}{\nu} (I_1 - 3)$$

$$= \frac{\mu}{\nu} (\lambda_1^2 + \lambda_2^2 + \lambda_3^2 - 3)$$

↓

$$\sigma_1, \sigma_2, \sigma_3 = \dots$$

$$\underline{P} = -p \underline{F}^{-T} + 2 \left[\left(\frac{\partial \Psi}{\partial I_1} + I_1 \frac{\partial \Psi}{\partial I_2} \right) \underline{F} - \frac{\partial \Psi}{\partial I_2} \underline{F} \underline{F} \right]$$

Prob 10

$$\chi_1 = \lambda_1 \underline{I}_1, \quad \chi_2 = \lambda_2 \underline{I}_2, \quad \chi_3 = \frac{1}{\lambda_1 \lambda_2} \underline{I}_3$$

★ Membrane theory:

Ogden's theory: $\sigma = \mu (\lambda^n - 1)$

$$W = \sum_i \frac{\alpha_i}{n_i} \left[(\lambda_1^2 + \lambda_2^2 + \frac{1}{\lambda_1^2} + \frac{1}{\lambda_2^2})^{\frac{n_i}{2}} - 2 \right]$$

Strain energy (Wikip) $W(\lambda_1, \lambda_2) = \sum_i \frac{\alpha_i}{n_i} (\lambda_1^{\alpha_i} + \lambda_2^{\alpha_i} + \lambda_1^{-\alpha_i} + \lambda_2^{-\alpha_i} - 4)$

We know λ_1 & λ_2 & $\det \underline{F} = 1$

HW note

Prob 7: reverse to get $\underline{X}(\underline{x})$

$$\begin{cases} \chi_1 = e^t \underline{I}_1 - e^{-t} \underline{I}_2 \\ \chi_2 = e^t \underline{I}_1 + e^{-t} \underline{I}_2 \\ \chi_3 = \underline{I}_3 \end{cases}$$

$$\underline{u} = \underline{x} - \underline{X}$$

$$\underline{F} = \frac{\partial \underline{x}}{\partial \underline{X}} = \frac{\partial (\underline{u} + \underline{X})}{\partial \underline{X}}$$

$$\begin{cases} \chi_1 + \chi_2 = 2e^t \underline{I}_1 \rightarrow \underline{I}_1 = \frac{1}{2e^t} (\chi_1 + \chi_2) \\ \chi_2 - \chi_1 = 2e^{-t} \underline{I}_2 \rightarrow \underline{I}_2 = \frac{1}{2e^{-t}} (\chi_2 - \chi_1) \\ \underline{I}_3 = \chi_3 \end{cases}$$

in the material description:

$$\underline{A}(\underline{X}, t) = \frac{\partial \underline{V}(\underline{X}, t)}{\partial t} = \begin{cases} \frac{\partial \chi_1}{\partial t} = e^t \underline{I}_1 - e^{-t} \underline{I}_2 \\ \frac{\partial \chi_2}{\partial t} = e^t \underline{I}_1 + e^{-t} \underline{I}_2 \\ \frac{\partial \chi_3}{\partial t} = 0 \end{cases}$$

$$\underline{V}(\underline{X}, t) = \frac{\partial \underline{x}(\underline{X}, t)}{\partial t} = \begin{cases} \frac{\partial \chi_1}{\partial t} = e^t \underline{I}_1 + e^{-t} \underline{I}_2 \\ \frac{\partial \chi_2}{\partial t} = e^t \underline{I}_1 - e^{-t} \underline{I}_2 \\ \frac{\partial \chi_3}{\partial t} = 0 \end{cases}$$

$$\frac{1}{2e^t} \frac{\partial \chi_1}{\partial t} - \frac{1}{2e^{-t}} \frac{\partial \chi_2}{\partial t} = \begin{bmatrix} e^t \underline{I}_1 + e^{-t} \underline{I}_2 \\ e^t \underline{I}_1 - e^{-t} \underline{I}_2 \\ 0 \end{bmatrix}$$

Prob. 8:

$$du = (F - I) dX$$

$$\frac{du}{dX} = F - I$$

$$\frac{du}{dX} + I = F$$

$$X - X = u$$

$$\frac{\partial X}{\partial X} \frac{\partial X}{\partial X} = \frac{\partial u}{\partial X}$$

$$I - F^{-1} = \frac{\partial u}{\partial X}$$

$$I - \frac{\partial u}{\partial X} = F^{-1}$$

$$F^{-1} = \begin{bmatrix} 1 & -\frac{1}{4} & 0 \\ 1 & 2 & 0 \\ 0 & 0 & -3 \end{bmatrix}$$

$$F^{-1} = \begin{bmatrix} 0 & -\frac{1}{4} & 0 \\ 1 & -1 & 0 \\ 0 & 0 & 4 \end{bmatrix}$$

$$F = \begin{bmatrix} 1 & 0 & 0 \\ -\frac{1}{4} & -\frac{1}{4} & 0 \\ 0 & 0 & -4 \end{bmatrix}$$

$$F = \frac{1}{\det(F)} \text{adj}(F)$$

$$\text{adj}(F^{-1}) = \begin{bmatrix} 4 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 4 \end{bmatrix}$$

Prob. 9 & 10.

$$\frac{\partial \psi}{\partial c} = \sum_i^3 \frac{\partial \psi}{\partial \lambda_i^2} \frac{\partial \lambda_i^2}{\partial c}$$

$$\frac{\partial \lambda_i^2}{\partial c} = \frac{n_i}{N_i}$$

$$\frac{\partial \psi}{\partial c} = \sum_i^3 \frac{\partial \psi}{\partial \lambda_i^2} \frac{n_i}{N_i}$$

$$S = 2 \frac{\partial \psi}{\partial c} = \sum_i^3 \frac{2 \frac{\partial \psi}{\partial \lambda_i^2} n_i}{\lambda_i^2} = \sum_i^3 \frac{\lambda_i^2 (n_i^2 + \lambda_i^2 n_i - 3)}{\lambda_i^2} n_i$$

$$= \frac{\lambda_1 n_1}{n_1 n_1} + \frac{\lambda_2 n_2}{n_2 n_2} + \frac{\lambda_3 n_3}{n_3 n_3}$$

$$P = F S$$

$$n(n_1 n_1 + n_2 n_2 + n_3 n_3)$$

$$F N_i = \lambda_i n_i$$

$$F_i N_i = \lambda_i n_i$$

$$P = F \sum_i^3 \frac{\partial \psi}{\partial \lambda_i^2} N_i \otimes N_i = \sum_i^3 \frac{\partial \psi}{\partial \lambda_i^2} N_i \otimes N_i$$

$$S = J^{-1} F P = \frac{1}{\lambda_1 \lambda_2 \lambda_3} \sum_i^3 \frac{\partial \psi}{\partial \lambda_i^2} \lambda_i N_i \otimes N_i$$

Prob 10.

Ogden model: plane stress = $\sigma_3 = 0$

$$\psi = \sum_i \frac{2\mu_i}{\alpha_i} (\bar{\lambda}_1^{\alpha_i} + \bar{\lambda}_2^{\alpha_i} + \bar{\lambda}_3^{\alpha_i} - 3) + \frac{K_1}{2} (J-1)^2$$

$$\frac{\partial \psi}{\partial \lambda_i} = \sum_i \frac{2\mu_i}{\alpha_i} \left(\frac{\lambda_i}{J^{1/3}}\right)^{\alpha_i}$$

↳ const. $\alpha_i \lambda_i^{\alpha_i-1}$

$$\frac{2\mu_i}{\alpha_i} \left(\frac{1}{J^{1/3}}\right)^{\alpha_i} \alpha_i \lambda_i^{\alpha_i-1}$$

$$= \frac{2\mu_i}{\alpha_i} \left(\frac{1}{J^{1/3}}\right)^{\alpha_i} \lambda_i^{\alpha_i-1}$$

$$\underline{F} = \frac{\partial \psi}{\partial \underline{x}} = \begin{bmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{bmatrix}$$

$$J = 1$$

Week 3 - Lecture 1.

weak form $\int \dots$
 Integrability \uparrow
 BCs \uparrow

$w = \begin{cases} w|_{\Omega} \in H^1, w=0 & \text{on } \Gamma_D \\ \text{on } \Gamma_N \end{cases}$

$u \Rightarrow u = \{u \mid u \in H^1, u=g \text{ on } \Gamma_D\}$

$\int_{\Omega} (\nabla w)^2 dV < \infty$
 does not blow up

* cannot prescribe both force & displacement

"Spring"-type BCs - Robin

essential vs. natural

Dirichlet vs. Neumann

$$u = g \quad - q \cdot n_i = h$$

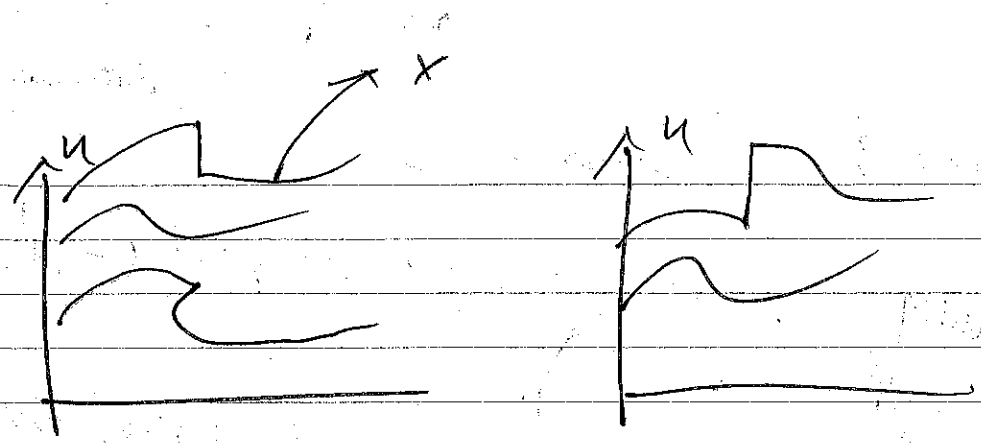
1st: multiply both sides by test function

$\Gamma = \Gamma_D \cup \Gamma_N$

$\int_{\Omega} w \operatorname{div} q = \int_{\Omega} w f$

$\int_{\Omega} w \operatorname{div} q dV = \int_{\Omega} w f dV$

$\int_{\Omega} w q \cdot n dS - \int_{\Omega} q \cdot \operatorname{grad} w dV = \int_{\Omega} w f dV$



∴ we find u has to be smooth continuous

$C^0 \rightarrow$ zeroth derivative is continuous

$C^1 \rightarrow$ first derivative is continuous

$C^n \rightarrow$ n^{th} derivative is continuous.

1) \star Strong form: point-wise
weak form: or an average sense.

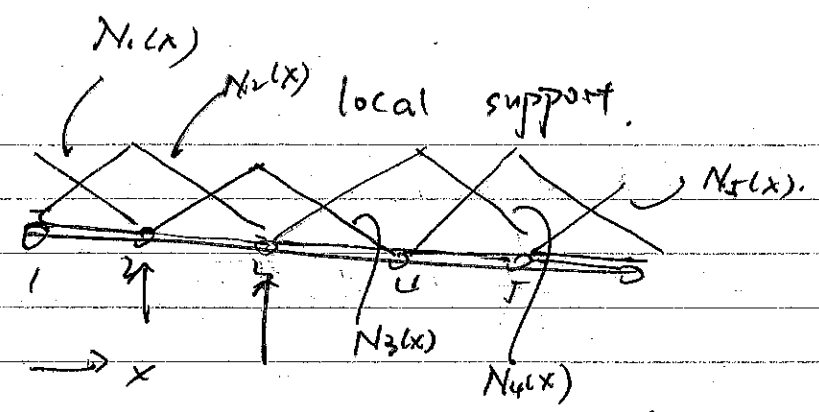
Order
2) Strong form: $a_{ij} = k_{ij} u_{,ij}$ - 2nd order

weak form: - 1st order deriv.

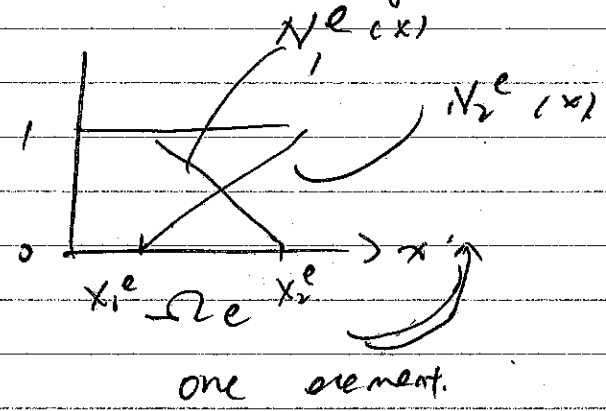
↳ symmetry between w & u

- Shape function example:

1D domain:



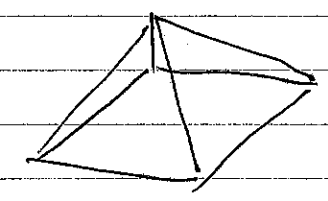
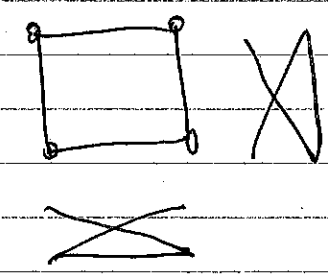
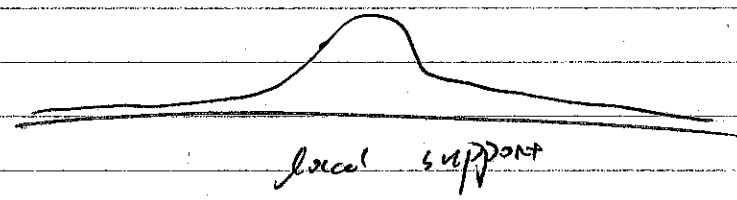
all the shape functions look the same.



- linear shape functions.

"bubble" function

one element.



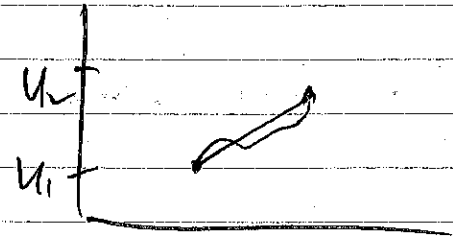
$$\rightarrow (W^h, u)_p = \sum_i C_A(N_A, u),$$

many integrals.

$$\sum_{A \in \mathcal{T}_T} C_A \left\{ \int_T N_1 c_1 ds + \int_T N_2 c_2 ds + \dots \right\} = 0 \rightarrow u \text{ can set}$$

$$u \approx u_1 N_1 + u_2 N_2$$

$$\frac{\partial u}{\partial x} \approx u_1 \frac{\partial N_1}{\partial x} + u_2 \frac{\partial N_2}{\partial x}$$

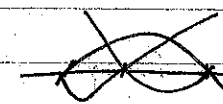
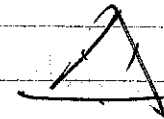


#FEM

Week 3 - lecture 2.

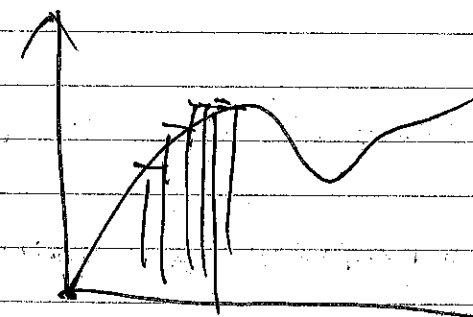
Galerkin formulation

$$[\quad] \begin{Bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{Bmatrix} = \{ \quad \} \rightarrow \text{solve for } d$$



quadratic

quadrature



Gauss quadrature

sample \rightarrow integration method

↑↑ Same shape function

Bubnov-Galerkin → same s.f.

Petrov-Galerkin → different s.f.

↳ subq.

how to solve PDE → tune the shape functions

Galerkin method vs. Ritz-Ritz method

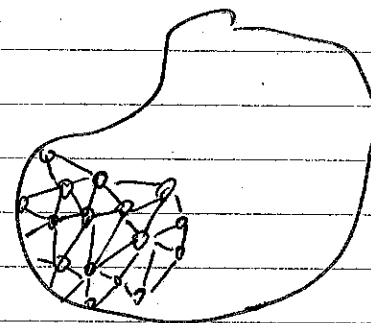
↓
Shape function
has to be global

Co: derivatives have a jump.

① what order derivatives in weak form

② what order continuities if required.

Week 4. Lecture 1.



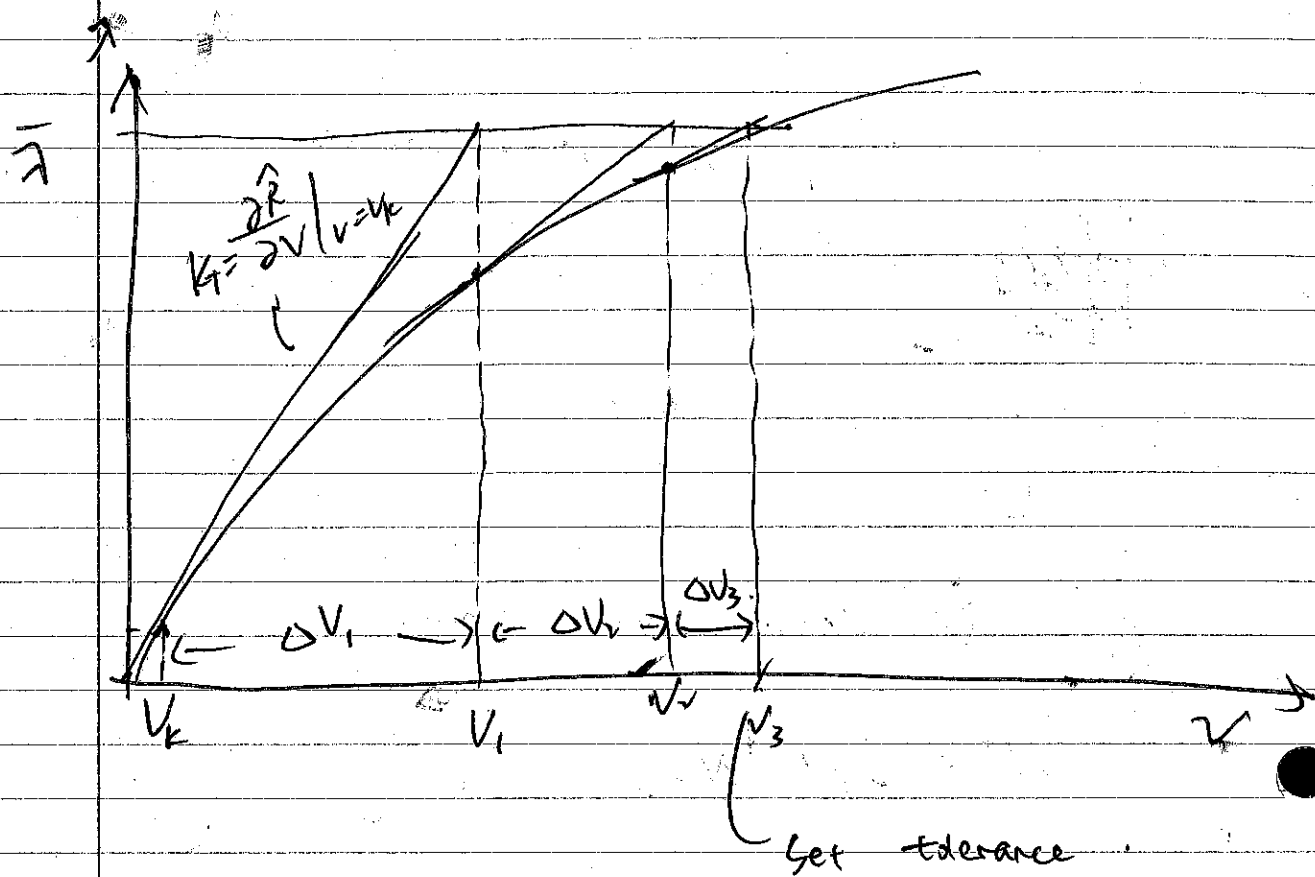
$u, u^h \rightarrow$ ele. size

to L^2

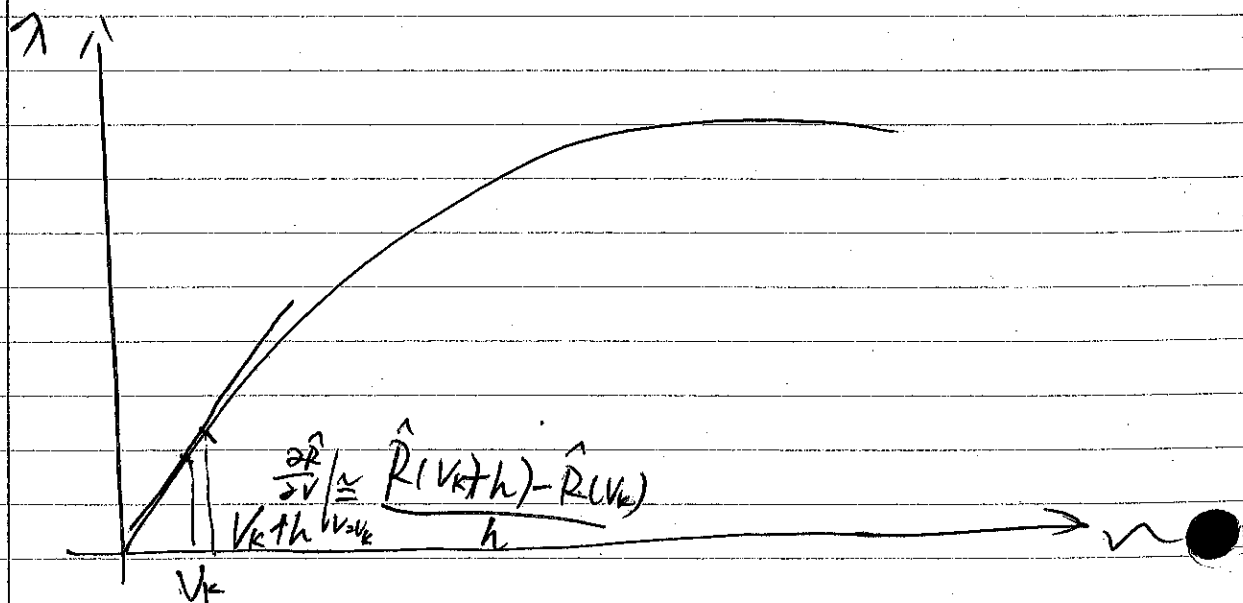
eg. 1D linear shape func.

$$\underline{N}^e = [N_1, N_2] \quad \underline{L}^e = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Week 5, Lecture 1

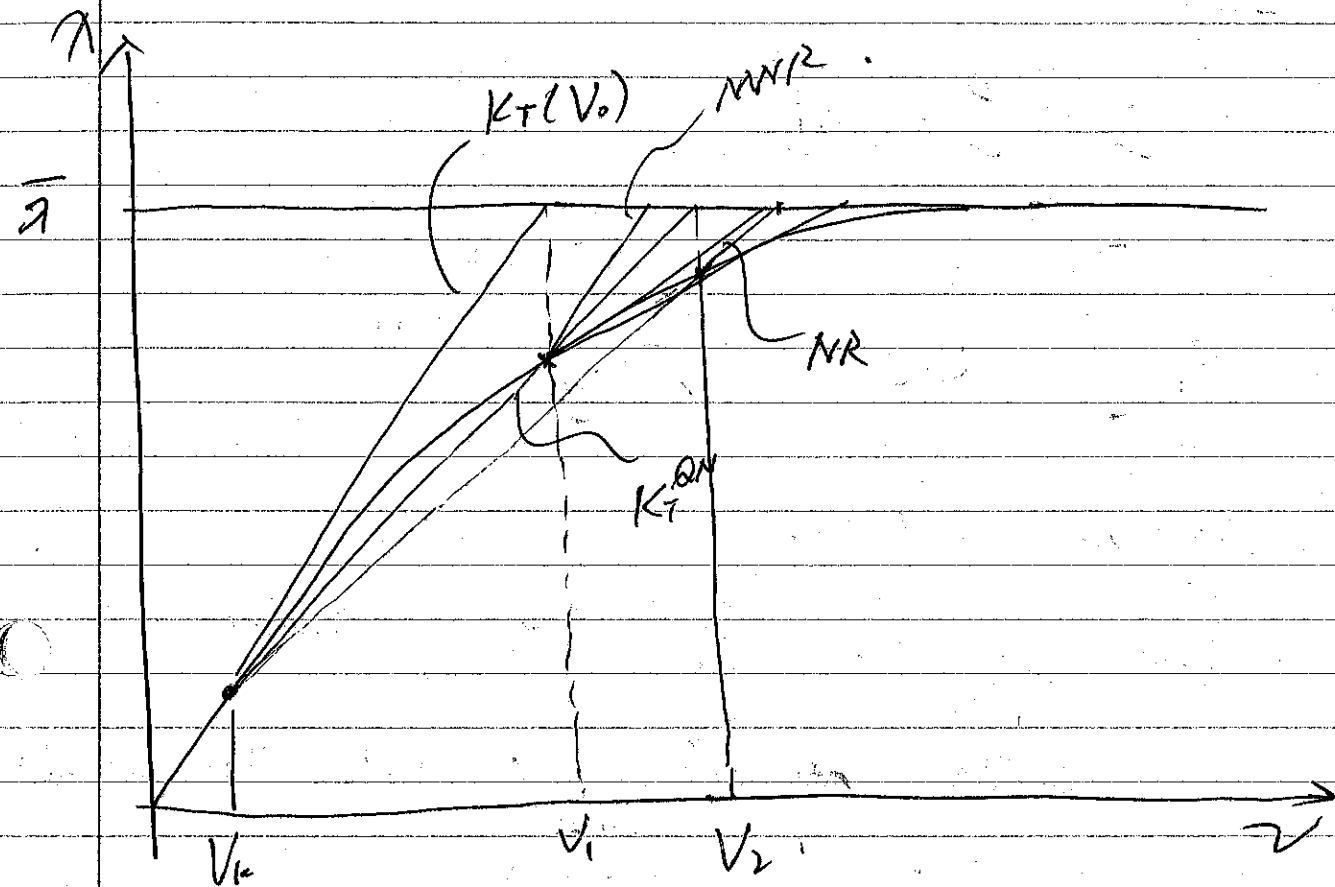


"Quadratic Convergence"



Lecture 2

Comparison between 3 methods



HW 2 :

Newton - Raphson

$$\underline{G} = \underline{R} - \lambda \underline{P}$$

$$\begin{bmatrix} G_1 \\ G_2 \end{bmatrix} = \begin{bmatrix} R_1 \\ R_2 \end{bmatrix} - \lambda \begin{bmatrix} P_1 \\ P_2 \end{bmatrix} \rightarrow 0$$

$V_0 = V_k$ (ICs).

For λ_i in λ [0, ..., 10] ← initialize

while $\Sigma \leq 1e-4$ and iter ≤ 15 .

$$\underline{G} + \underline{D}_G \underline{K}_T \Delta \underline{V}$$

$$\begin{bmatrix} G_1 \\ G_2 \end{bmatrix} + \begin{bmatrix} D_1 G_1 & D_1 G_2 \\ D_2 G_1 & D_2 G_2 \end{bmatrix} \begin{bmatrix} \Delta V_1 \\ \Delta V_2 \end{bmatrix}$$

equiv.

$$\underline{K}_T \Delta \underline{V} = -\underline{G} \quad \dots (1)$$

$$\underline{V} = \underline{V} + \Delta \underline{V} \quad \dots (2)$$

$$\|\underline{G}\| = \sqrt{G[0]^2 + G[1]^2} \quad (3)$$

for $j = [0, 1, 2, \dots, 40]$

$$\begin{cases} \Delta \lambda = 0.25 \\ \lambda = \Delta \lambda * j \end{cases}$$

$$\underline{G}_{j=0} = \underline{K}_{T0} \Delta \underline{V}_0$$

while $\Sigma > 1e-4$ or iter ≤ 15 .

$$\underline{G} = \underline{R} - \lambda \underline{P}$$

$$\underline{K}_T = [\quad]$$

$$\underline{V} = \underline{V} + \Delta \underline{V}$$

$$\|\underline{G}\| = \sqrt{\dots}$$

$$\underline{G} = - \begin{bmatrix} 0.6V_1 \Delta V_1 + 6 \Delta V_1 - 2xV_2 \Delta V_2 \\ - \Delta V_1 + \Delta V_2 \end{bmatrix}$$

$$\underline{K}_T = \begin{bmatrix} 0.6V_1 + 6 & -2xV_2 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} \Delta V_1 \\ \Delta V_2 \end{bmatrix}$$

Newton-Raphson implementation.

Initialization

for λ
 while $V_0 = V_k = \begin{bmatrix} \cdot \\ \cdot \end{bmatrix}$ $\rightarrow i \leq 15$.

compute G

$$G = R - \lambda P \Rightarrow \begin{bmatrix} G_1 \\ G_2 \end{bmatrix} = \begin{bmatrix} R_1 \\ R_2 \end{bmatrix} - \lambda \begin{bmatrix} P_1 \\ P_2 \end{bmatrix}$$

compute K_T

$$K_T = DG = \begin{bmatrix} \lambda G_1 & \lambda P_1 \\ \lambda G_2 & \lambda P_2 \end{bmatrix}$$

compute ΔV_{i+1}

$$\Delta V_{i+1} = \frac{G}{K_T} = \frac{G}{K_T} - \frac{G}{K_T} \cdot K_T^{-1} \cdot G$$

compute new V

$$V_{i+1} = V_i + \Delta V_{i+1}$$

$i = i + 1$

compute convergence

$$E = \sqrt{G_1^2 + G_2^2}$$

* How to solve I.C.S?

$$\begin{bmatrix} 0.2V_1^3 - \alpha V_2^2 + 6V_1 \\ V_2 = V_1 \end{bmatrix} = 0 \quad \rightarrow \quad 0.2V^3 - \alpha V^2 + 6V = 0$$

HW Line Search Method

\downarrow

select α_i .

$$g(\alpha_i) = \Delta V_{i+1}^T \left[G(V_i + \alpha_i \Delta V_{i+1}, \lambda) \right] = 0$$

$$G = \begin{bmatrix} 0.2V_1^3 - \alpha V_2^2 + 6V_1 \\ V_2 - V_1 \end{bmatrix} - \lambda_i \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$g(\alpha_i) = \begin{bmatrix} \Delta V_{i+1}^T \\ \Delta V_{i+1}^T \end{bmatrix} \left\{ \begin{array}{l} 0.2(V_i + \alpha_i \Delta V_i)^3 - \alpha(V_i + \alpha_i \Delta V_i)^2 + 6(V_i + \alpha_i \Delta V_i) \\ V_i + \alpha_i \Delta V_i - V_i - \alpha_i \Delta V_i \end{array} \right\} - \lambda_i \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

to solve for α_{i+1} next α

$$\alpha_i^{k+1} = \alpha_i^k - \frac{g(\alpha_i^k)}{g'(\alpha_i^k)} = \frac{\alpha_i^k - \alpha_i^{k-1}}{g(\alpha_i^k) - g(\alpha_i^{k-1})}$$

BFGS Method

Initialization

$$V_0 = V_k$$

Compute G

... (# same)

Compute K

(# same) $\Rightarrow K(0)$

For loop ...

- compute G

- compute H^{-1}

$$\underline{w}_i = \underline{v}_i - \underline{v}_{i-1}$$

$$[\] = [\] - [\]$$

$$\underline{g}_i = \underline{G}_i - \underline{G}_{i-1}$$

$$[\] = [\] - [\]$$

$$\underline{a} = \frac{1}{\text{np.transpose}(\underline{g}_i) \cdot \underline{w}_i \cdot \underline{w}_i}$$

$$\underline{b}_i = - \left\{ \underline{g}_i - \left[\frac{-\underline{w}_i^T \underline{g}_i}{\underline{w}_i^T \underline{G}_{i-1}} \right] \underline{G}_{i-1} \right\}$$

BFGS

def: compute $V(H, G, V)$

- $\Delta V = -HG$

- $V = V + \Delta V$

$\rightarrow V_{i+1}, V_i$

def: compute $G(R, \lambda(P), V)$

$$G_{\text{new}} \leftarrow G = R - \lambda P$$

$$G \leftarrow G_{\text{new}} = R_{\text{new}} - \lambda P$$

$\rightarrow G_{i+1}, G_i$

$$G_i, G_{i-1} \leftarrow$$

$$V_i, V_{i-1} \leftarrow$$

$$\underline{w}_i = \underline{v}_i - \underline{v}_{i-1}$$

$$\underline{g}_i = \underline{G}_i - \underline{G}_{i-1}$$

$$a_i = \frac{1}{\underline{g}_i^T \underline{w}_i \underline{w}_i}$$

$$b_i = - \left\{ \underline{g}_i - \left[\frac{-\underline{w}_i^T \underline{g}_i}{\underline{w}_i^T \underline{G}_{i-1}} \right] \underline{G}_{i-1} \right\}$$

$$\rightarrow H_i = H_i(a_i, b_i, H_{i-1})$$

for BFGS.

we have to solve exactly for

v & v_{prev} and G & G_{prev}

before the i^{th} loop:

v_{new} exact solution

def compute $G_i(\lambda, v)$.

→ G_i

def compute $H(v_i, v_{i-1}, G_i, G_{i-1}, \lambda)$.

→ H

def solve $v(\lambda)$.

"solve v analytically"

→ v

for $\lambda = [0, 0.25, 0.5, \dots, 1.0]$

$v_{i-1}, v_i, G_{i-1}, G_i$

while not converge:

$H = \text{compute } H$

$\Delta v = H \cdot G_i$

→ $v_{\text{new}} = v_i + \Delta v_i$; $v_i = v_{i-1}$

Solve for next v_i :

$$\begin{bmatrix} 0.2v_1^3 - \lambda v_2^2 + 6v_1 \\ v_2 - v_1 \end{bmatrix} - 0.25 \begin{bmatrix} 1 \\ 0 \end{bmatrix} = 0$$

$$0.2v_1^3 - \lambda v_2^2 + 6v_1 - 0.25 = 0$$

Week 6: Lecture 1.

Weak form & variational principles

(Lecture 11)

- finite deformations

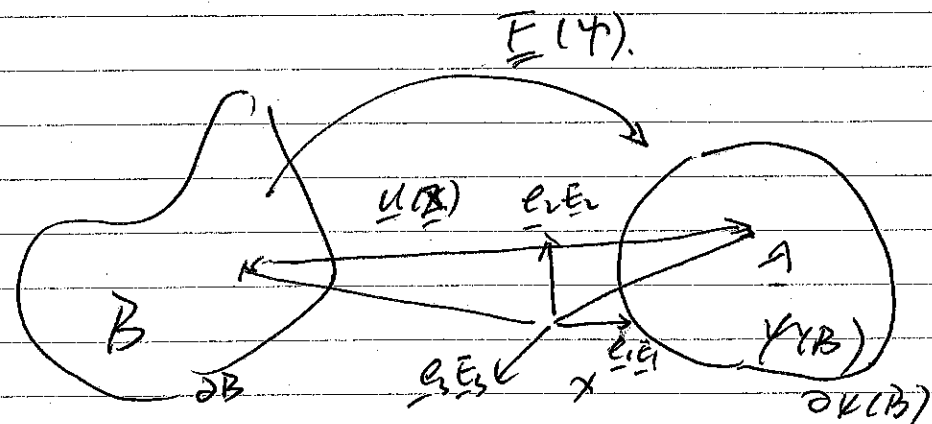
↳ geometric nonlinearities

↳ material nonlinearities (const. law)

Strong form \rightarrow Weak form

{ material description
spatial disp.

{ material ~
spat. ~



Strong form: hyperelasticity

- Kinematics:

$$\underline{x} = \underline{\psi}(\underline{X}) = \underline{u} + \underline{X} \quad \rightarrow \text{reference config.}$$

$$\underline{F} = \text{Grad } \varphi(\underline{X}, t) = \underline{I} + \text{Grad } \underline{u}$$

- Constitutive law

$$\underline{P} = \frac{\partial W(\underline{F})}{\partial \underline{F}} \quad \boxed{\underline{\epsilon} \cdot \underline{B}}$$

Equilibrium:

body forces per unit ref. vol.

strain energy function

$$\text{Div } \underline{P} + \underline{b} = \rho_0 \underline{\dot{v}} \quad \boxed{\underline{\epsilon} \cdot \underline{B}}$$

principle invariants of $\underline{C} = \underline{F}^T \underline{F}$

Material divergence

"2nd order"

- BCs:

$$\partial B_u \cup \partial B_\sigma = \partial B \quad \underline{u} = \bar{u} \quad \text{on } \partial B_u \quad \text{essential}$$

$$\partial B_u \cap \partial B_\sigma = \emptyset \quad \underline{P} \underline{n} = \bar{t} \quad \text{on } \partial B_\sigma \quad \text{natural}$$

Q.L. strain tensor

- kinematics
$$\underline{\underline{E}} = \frac{1}{2} (\underline{\underline{F}}^T + \underline{\underline{F}} - \underline{\underline{I}}) \quad \underline{\underline{F}} = \underline{\underline{I}} + \text{Grad} \underline{\underline{u}}$$

- const. mod.
$$\underline{\underline{\Sigma}} = \frac{\partial W}{\partial \underline{\underline{F}}}$$

- equilibrium
$$\text{Div} (\underline{\underline{F}} \underline{\underline{\Sigma}}) + \underline{\underline{b}} = \rho_0 \underline{\underline{v}}$$

Reference density

- BCS.

current config.

$\underline{\underline{b}}$ - per unit current volume

ρ - current density

$$\underline{\underline{\sigma}} = \frac{1}{J} \underline{\underline{T}} \leftarrow \text{Kirchhoff stress tensor}$$

lecture 13.

$\eta \cdot f \rightarrow \text{Scalar}$

$\eta_i f_i$

Single contraction
$$\underline{\underline{A}} \cdot \underline{\underline{B}} = A_{ik} B_{kj} = M_{ij}$$

$$\underline{\underline{A}} \cdot \underline{\underline{B}} = A_{ij} B_{ij} = \alpha$$

↓ trace of matrix

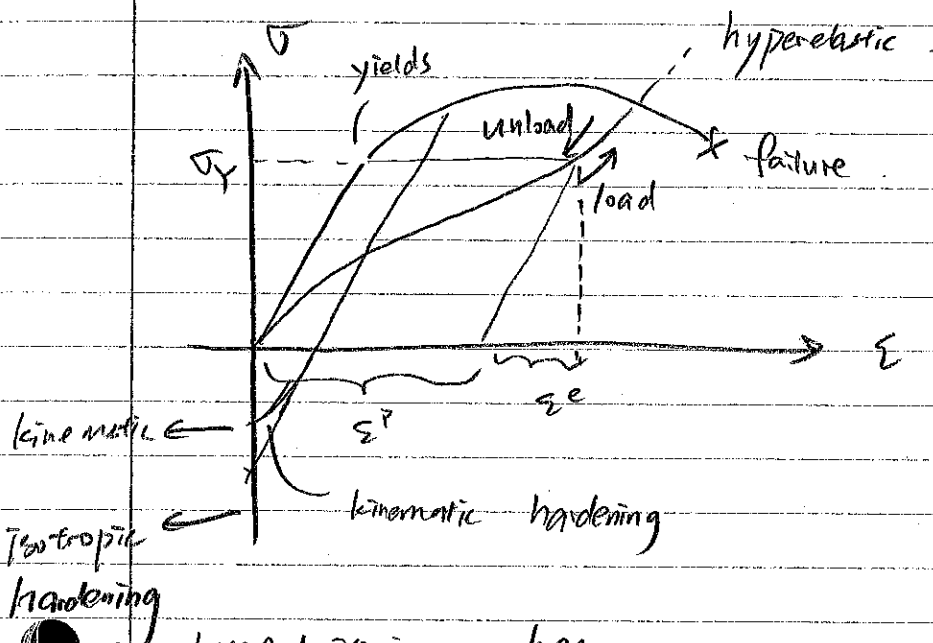
$$\underline{\underline{A}} \underline{\underline{B}} = \underline{\underline{M}}$$

$$\underline{\underline{A}} \cdot \underline{\underline{B}} = \alpha$$

Conclude: Course Notes for FEM

Additional Notes on Plasticity

plasticity vs. hyper-elasticity



hyper-elasticity has a one-on-one mapping between the stress and strains

Hyper-elastic: $\underline{\underline{S}} = \gamma \frac{\partial W(\underline{\underline{C}})}{\partial \underline{\underline{C}}} = \Psi(\underline{\underline{C}})$

FEA solves
 $u = \bar{u} + \Delta u$
 \downarrow
 $\underline{\underline{C}}$

$$G(\bar{\varphi}, \gamma) + \Delta G \cdot \Delta u \approx 0$$

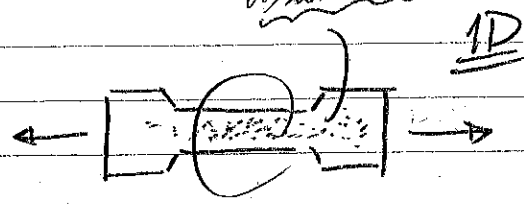
↑
 Calculate stresses in this term.

plasticity - easiest version

↓
 irreversibility.

dissipate energy to induce plastic deformation.

thinking an experiment
 assume macroscopic perfect



homogeneous stress state

why we like it

Assume small deformations: or kinematic
 "empirical assumption from obs."

$$\underline{\underline{\epsilon}} = \underline{\underline{\epsilon}}^e + \underline{\underline{\epsilon}}^p$$

elastic plastic

(Additive strain decomposition)

diviatoric

$$\underline{\underline{e}} = \underline{\underline{\epsilon}} - \frac{1}{3} \text{tr} \underline{\underline{\epsilon}} \underline{\underline{I}}$$

$$\underline{\underline{S}} = \underline{\underline{\sigma}} - \frac{1}{3} \text{tr} \underline{\underline{\sigma}} \underline{\underline{I}}$$

conserv. angular momentum

the same way we define a strain energy

$$\underline{\underline{\sigma}} = \frac{\partial \psi(\underline{\underline{\epsilon}}^e, \underline{\underline{\alpha}})}{\partial \underline{\underline{\epsilon}}^e}$$

Stress & strain → External

$$\underline{\underline{q}} = \frac{\partial \psi(\underline{\underline{\epsilon}}^e, \underline{\underline{\alpha}})}{\partial \underline{\underline{\alpha}}}$$

microstructural

Internal - not obs.

$\underline{\underline{q}}$ - internal vars - we cannot directly observe

Some dissipation mechanism, has to be

mathematically described by some thermodynamic conjugate variables - hardening.

$\underline{\underline{q}}$ is the thermodynamic conjugate of $\underline{\underline{\alpha}}$

Strain energy → elastic tensor
 → hardening variables

$$\psi(\underline{\underline{\epsilon}}^e, \underline{\underline{\alpha}}) = W_e(\underline{\underline{\epsilon}}^e) + W_h(\underline{\underline{\alpha}})$$

From linear elasticity, for small deformation.

$$W_e = \frac{1}{2} \underline{\underline{\epsilon}}^e : \underline{\underline{C}}^e [\underline{\underline{\epsilon}}^e]$$

Hook's law

$$\Delta E = \underline{\underline{C}}^e [\Delta \underline{\underline{\epsilon}}^e]$$

Variational form

For hardening potential

$$W_h = \frac{1}{2} H \hat{\alpha}^2 + \frac{1}{3} H |\underline{\underline{\alpha}}|^2$$

kinematic 2nd order scalar

isotropic hardening

$$\underline{\underline{q}} = -\frac{2}{3} H \underline{\underline{\alpha}}, \quad q_{ij} = -\frac{2}{3} H \alpha_{ij}$$

and $\hat{q} = -H \hat{\alpha}$

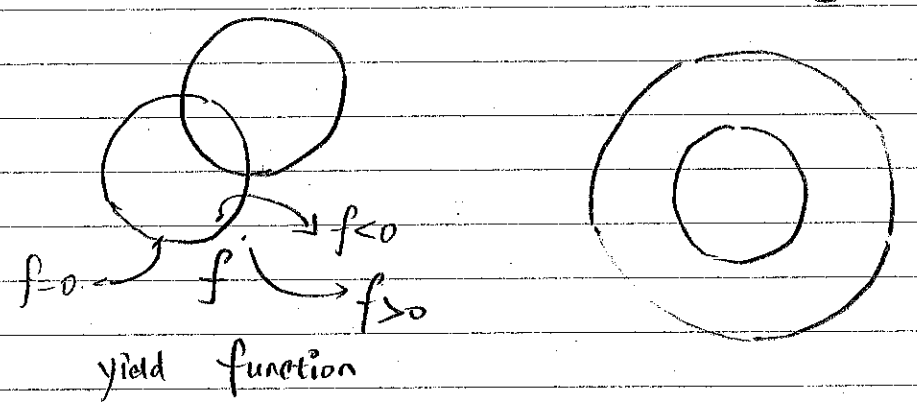
isotropic hardening
 kinematic hardening

$$\begin{cases} \underline{\underline{q}} = - \frac{\partial W_r}{\partial \underline{\underline{\alpha}}} \\ \dot{\underline{\underline{q}}} = - \frac{\partial W_r}{\partial \dot{\underline{\underline{q}}}} \end{cases}$$

→ simplified version, not the only form

the yield condition: $f(\underline{\underline{\Sigma}}, \underline{\underline{q}}, \dot{\underline{\underline{q}}}) \leq 0$

↓ ↓
kinematic isotropic
hardening



in 1D: $f = |\underline{\underline{\sigma}}| - \underline{\underline{\sigma}}_Y \dot{\underline{\underline{q}}}$ modulate the yield stress

$\underline{\underline{D}} = \text{dev } \underline{\underline{\Sigma}}, \quad (\underline{\underline{\Sigma}} - \underline{\underline{q}}) \cdot (\underline{\underline{\Sigma}} - \underline{\underline{q}})$
 $\underline{\underline{I}}_1^{\underline{\underline{\Sigma}} - \underline{\underline{q}}} = \text{tr}(\underline{\underline{\Sigma}} - \underline{\underline{q}})$ pressure independent

linear isotropic & kinematic hardening

$$f(\underline{\underline{\Sigma}}, \underline{\underline{q}}, \dot{\underline{\underline{q}}}) = \|\underline{\underline{\Sigma}} - \underline{\underline{q}}\| - \sqrt{\frac{2}{3}} (\underline{\underline{\sigma}}_0 - \dot{\underline{\underline{q}}}) \leq 0$$

Von Mises plasticity.

$$f(\underline{\underline{\Sigma}}, \underline{\underline{q}}, \dot{\underline{\underline{q}}}) = \sqrt{(\underline{\underline{\Sigma}} - \underline{\underline{q}}) \cdot (\underline{\underline{\Sigma}} - \underline{\underline{q}})} - k(\dot{\underline{\underline{q}}}) \leq 0$$

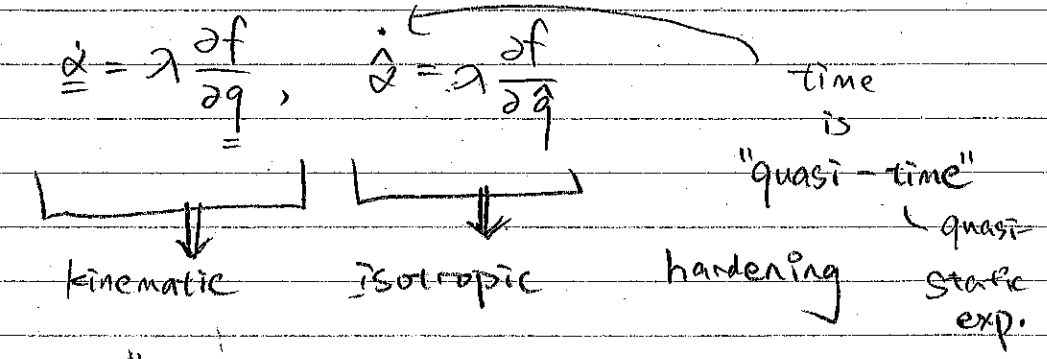
↓
if no internal variables
↓
perfect plasticity

Elasto-plastic material laws — Flow rules

associate flow rule: → plastic flow
 induce plasticity
 $\underline{\underline{\dot{P}}} = \lambda \frac{\partial f}{\partial \underline{\underline{\Sigma}}}$

$$\begin{cases} \underline{\underline{\Sigma}}(\underline{\underline{\epsilon}}^e, \underline{\underline{\alpha}}, \dot{\underline{\underline{q}}}) \\ \underline{\underline{q}} \end{cases}$$

"associate plasticity" — define the plastic strain rate



flow direction:

$$\underline{n} = \frac{\partial f}{\partial \underline{\underline{S}}} = \frac{\underline{\underline{S}} - \underline{\underline{q}}}{\|\underline{\underline{S}} - \underline{\underline{q}}\|}$$

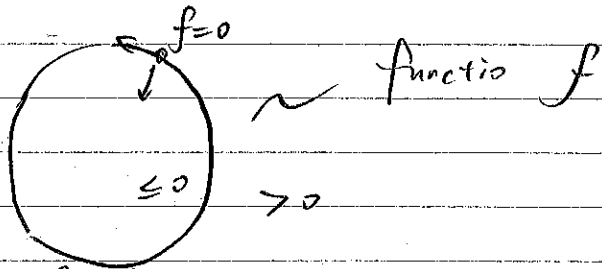
$$\frac{\partial f}{\partial \underline{\underline{q}}} = -\underline{n}$$

the evolution equations:

$$\dot{\underline{\underline{e}}}^p = \lambda \underline{n}, \quad \dot{\underline{\underline{q}}} = -\lambda \underline{n}, \quad \dot{\hat{\alpha}} = \lambda \sqrt{\frac{2}{3}}$$

the latter yields plastic strain inc.

$$\dot{\hat{\alpha}} = \sqrt{\frac{2}{3}} \|\dot{\underline{\underline{e}}}^p\|, \quad \|\dot{\underline{\underline{e}}}^p\| = \lambda$$



$$f = f(\underline{\underline{S}}, \underline{\underline{q}}, \hat{\alpha})$$

When stress is on $f=0$, distinguish 3 cases:

- elastic unloading. $\lambda=0, f<0$
- neutral loading $\lambda=0$ specifically for 3D case $f=0$
- plastic flow. $\lambda>0, f=0$

KKT condition summarizes the 3 cases.

$$\lambda \geq 0, \quad f \leq 0, \quad \lambda f = 0$$

→ for any dissipative problem.

& consistency condition:

$$\dot{\lambda} f = 0, \quad \text{if } f=0$$

- $f < 0, \lambda = 0$
- $f = 0, \lambda = 0$
- $f = 0, \lambda > 0$

if $\lambda = 0$,

derive a incremental form of small deformation plasticity; consistency for isotropic kinematic hardening.

$$\dot{f} = \frac{\partial f}{\partial \underline{\underline{S}}} \dot{\underline{\underline{S}}} + \frac{\partial f}{\partial \underline{\underline{q}}} \dot{\underline{\underline{q}}} + \frac{\partial f}{\partial \hat{\alpha}} \dot{\hat{\alpha}} = 0$$

$$\begin{aligned} \dot{f} &= \frac{\partial f}{\partial \underline{\underline{S}}} \cdot \underline{\underline{C}}^e [\dot{\underline{\underline{e}}} - \dot{\underline{\underline{e}}}^p] + \frac{\partial f}{\partial \underline{\underline{q}}} \dot{\underline{\underline{q}}} + \frac{\partial f}{\partial \hat{\alpha}} \dot{\hat{\alpha}} \\ &= \underline{n} \cdot \underline{\underline{C}}^e [\dot{\underline{\underline{e}}}] - \lambda \left(\underline{n} \cdot \underline{\underline{C}}^e [\underline{\underline{A}}] + \frac{2}{3} \|\underline{n}\| \dot{\hat{\alpha}} + \frac{2}{3} \dot{\hat{\alpha}} \right) = 0 \end{aligned}$$

the incremental rule: $\underline{\underline{\sigma}} = \text{dev} \underline{\underline{\underline{C}}} \underline{\underline{\underline{S}}} + p \underline{\underline{I}}$

$$\underline{\underline{\underline{C}}} = \underline{\underline{\underline{C}}}^e \underline{\underline{\underline{C}}}^p$$

We can solve for plastic multiplier:

$$\lambda = A^{-1} \underline{n} \cdot \underline{\Phi}^e [\underline{\dot{\epsilon}}]$$

linear elasticity "standard" ~

Q: \emptyset when to use specific R?

(?) R_u vs. $R \dots ?$

$\hookrightarrow \bar{v}$ vs. $v \dots ?$

MAE 7750: HW #1

Hanfeng Zhai

February 6, 2023

1. Obtain an expression for $\partial \mathbf{A}^{-1} / \partial \mathbf{A}$, where \mathbf{A} is a second order tensor. (Hint: use indicial notation and start from identity $A_{ik}^{-1} A_{kj} = \delta_{ij}$).

We start from the term $A_{ik}^{-1} A_{kj}$, by applying partial derivative on this term:

$$\frac{\partial (A_{ik}^{-1} A_{kj})}{\partial A_{pq}} = \frac{\partial A_{ik}^{-1}}{\partial A_{pq}} A_{kj} + A_{ik}^{-1} \frac{\partial A_{kj}}{\partial A_{pq}}$$

Due to $A_{ik}^{-1} A_{kj} = \delta_{ij}$, we have $\partial \delta_{ij} / \partial A_{pq} = 0$. We then have

$$\begin{aligned} \frac{\partial A_{ik}^{-1}}{\partial A_{pq}} A_{kj} + A_{ik}^{-1} \frac{\partial A_{kj}}{\partial A_{pq}} &= 0 \\ \frac{\partial A_{ik}^{-1}}{\partial A_{pq}} A_{kj} &= -A_{ik}^{-1} \frac{\partial A_{kj}}{\partial A_{pq}} \end{aligned}$$

We thus obtain

$$\begin{aligned} \frac{\partial A_{ik}^{-1}}{\partial A_{pq}} &= -A_{ik}^{-1} \frac{\partial A_{kj}}{\partial A_{pq}} A_{kj}^{-1} \\ &= -A_{ik}^{-1} \delta_{kp} \delta_{jq} A_{kj}^{-1} \\ &= -A_{ip}^{-1} A_{kq}^{-1} \end{aligned}$$

Hence, we obtain

$$\frac{\partial \mathbf{A}^{-1}}{\partial \mathbf{A}} = -\mathbf{A}^{-1} \mathbf{A}^{-1}$$

2. Obtain the expression $\partial \det \mathbf{A} / \partial \mathbf{A} = \det(\mathbf{A}) \mathbf{A}^{-T}$ using direct notation.

According to the definition of derivative, consider $\partial \det \mathbf{A} / \partial \mathbf{A}$ multiplies another tensor \mathbf{B}^1 :

¹Ref.: [https://en.wikipedia.org/wiki/Tensor_derivative_\(continuum_mechanics\)](https://en.wikipedia.org/wiki/Tensor_derivative_(continuum_mechanics))

$$\begin{aligned}
\frac{\partial \det \mathbf{A}}{\partial \mathbf{A}} : \mathbf{B} &= \left. \frac{d}{d\alpha} \det(\mathbf{A} + \alpha \mathbf{B}) \right|_{\alpha=0} \\
&= \left. \frac{d}{d\alpha} \det \left[\alpha \mathbf{A} \left(\frac{1}{\alpha} \mathbf{I} + \mathbf{A}^{-1} \cdot \mathbf{B} \right) \right] \right|_{\alpha=0} \\
&= \left. \frac{d}{d\alpha} \left[\alpha^3 \det(\mathbf{A}) \det \left(\frac{1}{\alpha} \mathbf{I} + \mathbf{A}^{-1} \cdot \mathbf{B} \right) \right] \right|_{\alpha=0}.
\end{aligned} \tag{1}$$

writing the determinant in the form of invariants:

$$\det(\lambda \mathbf{I} + \mathbf{A}) = \lambda^3 + I_1(\mathbf{A})\lambda^2 + I_2(\mathbf{A})\lambda + I_3(\mathbf{A}) \tag{2}$$

Substitute equation (2) back into equation (1) we have:

$$\begin{aligned}
\frac{\partial \det \mathbf{A}}{\partial \mathbf{A}} : \mathbf{B} &= \left. \frac{d}{d\alpha} \left[\alpha^3 \det(\mathbf{A}) \left(\frac{1}{\alpha^3} + I_1(\mathbf{A}^{-1} \cdot \mathbf{B}) \frac{1}{\alpha^2} + I_2(\mathbf{A}^{-1} \cdot \mathbf{B}) \frac{1}{\alpha} + I_3(\mathbf{A}^{-1} \cdot \mathbf{B}) \right) \right] \right|_{\alpha=0} \\
&= \left. \det(\mathbf{A}) \frac{d}{d\alpha} \left[1 + I_1(\mathbf{A}^{-1} \cdot \mathbf{B}) \alpha + I_2(\mathbf{A}^{-1} \cdot \mathbf{B}) \alpha^2 + I_3(\mathbf{A}^{-1} \cdot \mathbf{B}) \alpha^3 \right] \right|_{\alpha=0} \\
&= \left. \det(\mathbf{A}) \left[I_1(\mathbf{A}^{-1} \cdot \mathbf{B}) + 2I_2(\mathbf{A}^{-1} \cdot \mathbf{B}) \alpha + 3I_3(\mathbf{A}^{-1} \cdot \mathbf{B}) \alpha^2 \right] \right|_{\alpha=0} \\
&= \det(\mathbf{A}) I_1(\mathbf{A}^{-1} \cdot \mathbf{B}) \\
&= \det(\mathbf{A}) \text{tr}(\mathbf{A}^{-1} \cdot \mathbf{B}) \\
&= \det(\mathbf{A}) [\mathbf{A}^{-1}]^T : \mathbf{B}
\end{aligned}$$

3. Consider the dyad $\mathbf{D} = \mathbf{a} \otimes \mathbf{a}$.

- Write out the components of \mathbf{D} in matrix form.

Let $\mathbf{a} = [a_1, a_2, \dots, a_n]^T$, the components of the dyad \mathbf{D} writes:

$$\mathbf{D} = \begin{bmatrix} a_1 a_1 & a_1 a_2 & \cdots & a_1 a_n \\ a_2 a_1 & a_2 a_2 & \cdots & a_2 a_n \\ \vdots & \vdots & \ddots & \vdots \\ a_n a_1 & a_n a_2 & \cdots & a_n a_n \end{bmatrix}$$

- Compute the three principal invariants of \mathbf{D} simplifying as much as possible.

The first principal invariant I_1 is the trace of \mathbf{D} :

$$I_1 = \sum_{i=1}^n \mathbf{D}_{ii} = \sum_{i=1}^n a_i^2$$

The second principal invariant I_2 is the determinant of the \mathbf{D} :

$$I_2 = \det(\mathbf{D}) = \det(\mathbf{a} \otimes \mathbf{a}) = (\det(\mathbf{a}))^2 = \left(\prod_{i=1}^n a_i\right)^2$$

The third principal invariant I_3 is the product of the non-zero eigenvalues of \mathbf{D} , which is the product of all the components of \mathbf{a} :

$$I_3 = \prod_{i=1}^n a_i$$

- Compute the eigenvalues of \mathbf{D} .

The eigenvalue equation writes:

$$\sum_{j=1}^n a_j^2 v_j = \lambda v_i, \text{ for } i = 1, 2, \dots, n$$

Since \mathbf{D} is a scalar multiple of $\mathbf{a} \otimes \mathbf{a}$, all the eigenvalues are equal to $\mathbf{a} \cdot \mathbf{a} = \sum_{i=1}^n a_i^2$. Hence, the eigenvalues of $\mathbf{D} = \mathbf{a} \otimes \mathbf{a}$ are

$$\lambda = \sum_{i=1}^n a_i^2$$

with corresponding eigenvectors $\mathbf{v} = \mathbf{a}$

4. Let tensor \mathbf{A} be given by $\mathbf{A} = \alpha(\mathbf{I} - \mathbf{e}_1 \otimes \mathbf{e}_1) + \beta(\mathbf{e}_1 \otimes \mathbf{e}_2 + \mathbf{e}_2 \otimes \mathbf{e}_1)$. where α, β are non-zero scalars and $\mathbf{e}_1, \mathbf{e}_2$ orthonormal vectors.

- Show that the eigenvalues of \mathbf{A} are

$$\begin{aligned} \lambda_1 &= \alpha \\ \lambda_{2,3} &= \alpha/2 \pm (\alpha^2/4 + \beta^2)^{1/2} \end{aligned}$$

Since $\mathbf{A} = \alpha(\mathbf{I} - \mathbf{e}_1 \otimes \mathbf{e}_1) + \beta(\mathbf{e}_1 \otimes \mathbf{e}_2 + \mathbf{e}_2 \otimes \mathbf{e}_1)$, then

Given a second-order tensor \mathbf{A} , the eigenvalues λ_i are found by solving the characteristic equation $\det(\mathbf{A} - \lambda_i \mathbf{I}) = 0$, where \mathbf{I} is the identity tensor.

\mathbf{A} can be expanded as

$$[\mathbf{A}] = \alpha \left(\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} - \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \right) + \beta \left(\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \right) = \begin{bmatrix} 0 & \beta & 0 \\ \beta & \alpha & 0 \\ 0 & 0 & \alpha \end{bmatrix}$$

Solving the characteristic equation we have

$$\det \begin{bmatrix} -\lambda & \beta & 0 \\ \beta & \alpha - \lambda & 0 \\ 0 & 0 & \alpha - \lambda \end{bmatrix} = 0 \rightarrow (-\lambda(\alpha - \lambda) - \beta^2)(\alpha - \lambda) = 0$$

We can hence obtain the three eigenvalues:

$$\lambda_1 = \alpha$$

$$\lambda_{2,3} = \alpha/2 \pm (\alpha^2/4 + \beta^2)^{1/2}$$

- Compute the associated eigenvectors.

Solving the three linear equations by substituting the eigenvalues we have:

$$\left\{ \begin{array}{l} \left(\begin{bmatrix} 0 & \beta & 0 \\ \beta & \alpha & 0 \\ 0 & 0 & \alpha \end{bmatrix} - \begin{bmatrix} \alpha & 0 & 0 \\ 0 & \alpha & 0 \\ 0 & 0 & \alpha \end{bmatrix} \right) \mathbf{v}_1 = 0 \\ \left(\begin{bmatrix} 0 & \beta & 0 \\ \beta & \alpha & 0 \\ 0 & 0 & \alpha \end{bmatrix} - \begin{bmatrix} \frac{\alpha}{2} + (\frac{\alpha^2}{4} + \beta^2)^{1/2} & 0 & 0 \\ 0 & \frac{\alpha}{2} + (\frac{\alpha^2}{4} + \beta^2)^{1/2} & 0 \\ 0 & 0 & \frac{\alpha}{2} + (\frac{\alpha^2}{4} + \beta^2)^{1/2} \end{bmatrix} \right) \mathbf{v}_2 = 0 \\ \left(\begin{bmatrix} 0 & \beta & 0 \\ \beta & \alpha & 0 \\ 0 & 0 & \alpha \end{bmatrix} - \begin{bmatrix} \frac{\alpha}{2} - (\frac{\alpha^2}{4} + \beta^2)^{1/2} & 0 & 0 \\ 0 & \frac{\alpha}{2} - (\frac{\alpha^2}{4} + \beta^2)^{1/2} & 0 \\ 0 & 0 & \frac{\alpha}{2} - (\frac{\alpha^2}{4} + \beta^2)^{1/2} \end{bmatrix} \right) \mathbf{v}_3 = 0 \end{array} \right.$$

Solving these equations we then obtain the three eigenvectors:

```

1 from sympy import *
2 from sympy.solvers.solveset import linsolve
3
4 alpha, beta = symbols('alpha beta')
5
6 A = Matrix([[0, beta, 0], [beta, alpha, 0], [0, 0, alpha]])
7 B1 = Matrix([[alpha, 0, 0], [0, alpha, 0], [0, 0, alpha]])
8 B2 = Matrix([[alpha/2 + sqrt(alpha**2/4 + beta**2), 0, 0], [0, alpha/2 + sqrt(alpha
**2/4 + beta**2), 0], [0, 0, alpha/2 + sqrt(alpha**2/4 + beta**2)]])
9 B3 = Matrix([[alpha/2 - sqrt(alpha**2/4 + beta**2), 0, 0], [0, alpha/2 - sqrt(alpha
**2/4 + beta**2), 0], [0, 0, alpha/2 - sqrt(alpha**2/4 + beta**2)]])
10
11 M1 = A - B1; M2 = A - B2; M3 = A - B3;
12 v1 = Matrix([Symbol('v1(1)'), Symbol('v1(2)'), Symbol('v1(3)')])
13 v2 = Matrix([Symbol('v2(1)'), Symbol('v2(2)'), Symbol('v2(3)')])
14 v3 = Matrix([Symbol('v3(1)'), Symbol('v3(2)'), Symbol('v3(3)')])
15
16 sol1 = linsolve((M1*v1), (v1[0], v1[1], v1[2]))
17 sol2 = linsolve((M2*v2), (v2[0], v2[1], v2[2]))
18 sol3 = linsolve((M3*v3), (v3[0], v3[1], v3[2]))

```

Written in symbolic forms as:

$$\left\{ \begin{array}{l} \mathbf{v}_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \\ \mathbf{v}_2 = \begin{bmatrix} \frac{2\beta}{\alpha + \sqrt{\alpha^2 + 4\beta^2}} \\ 1 \\ 0 \end{bmatrix} \\ \mathbf{v}_3 = \begin{bmatrix} \frac{2\beta}{\alpha - \sqrt{\alpha^2 + 4\beta^2}} \\ 1 \\ 0 \end{bmatrix} \end{array} \right.$$

- Under which conditions on α, β is \mathbf{A} positive definite.

We can compute $\det \mathbf{A} = -\alpha\beta^2$.

To satisfy the positive-definite conditions $-\alpha\beta^2 > 0$, one should obtain $\alpha < 0$ & $\beta \neq 0$.

5. Let ϕ and \mathbf{u} be smooth scalar and vector fields defined on the surface \mathcal{S} and curve \mathcal{C} , and let \mathbf{n} the unit outward normal on \mathcal{S} . Show that

$$\oint_{\mathcal{C}} \phi d\mathbf{x} = \int_{\mathcal{S}} \mathbf{n} \times \text{grad}\phi ds$$

$$\oint_{\mathcal{C}} \mathbf{u} \times d\mathbf{x} = \int_{\mathcal{S}} [(\text{div}\mathbf{u})\mathbf{n} - (\text{grad}^T \mathbf{u})\mathbf{n}] ds$$

For the first equation, recall Stoke's theorem, considering the definition of a circulation per unit area², we have:

$$\sum \phi d\mathbf{x} = (\nabla \times \phi) ds$$

we can then write:

$$\begin{aligned} \oint_{\mathcal{C}} \phi d\mathbf{x} &= \oint_{\mathcal{C}} d\mathbf{x} \times \text{grad}\phi ds \\ &= \int_{\mathcal{S}} \mathbf{n} \times \text{grad}\phi ds \end{aligned}$$

²Ref.: <https://www.lehman.edu/faculty/anchordoqui/VC-4.pdf>

For the second equation, we can start with the LHS:

$$\begin{aligned}\oint_C \mathbf{u} \times d\mathbf{x} &= \oint_C d(\mathbf{x} \times \mathbf{u}) \\ &= \int_S d\mathbf{x} \times \mathbf{u} ds \\ &= \int_S [(\operatorname{div}\mathbf{u})\mathbf{n} - (\operatorname{grad}^T \mathbf{u})\mathbf{n}] ds\end{aligned}$$

6. Apply the operator ∇ to product of smooth tensor fields \mathbf{A} , \mathbf{B} to establish the identity:

$$\operatorname{div}(\mathbf{AB}) = \operatorname{grad}\mathbf{A} : \mathbf{B} + \mathbf{A}\operatorname{div}\mathbf{B}$$

Starting with the LHS:

$$\begin{aligned}\operatorname{div}(\mathbf{AB}) &= \nabla \cdot (\mathbf{AB}) \\ &= \sum_{i=1}^n \frac{\partial(A_j B_j)}{\partial x_i} \\ &= \sum_{i=1}^n \frac{\partial A_j}{\partial x_i} B_j + \sum_{i=1}^n A_j \frac{\partial B_j}{\partial x_i} \\ &= \operatorname{grad}\mathbf{A} : \mathbf{B} + \mathbf{A}\operatorname{div}\mathbf{B}\end{aligned}$$

7. A certain motion of a continuum body in the material description is given in the form

$$x_1 = e^t X_1 - e^{-t} X_2, x_2 = e^t X_1 + e^{-t} X_2, x_3 = X_3$$

for $t > 0$. Find the velocity and acceleration components in terms of the material and spatial coordinates and time.

The velocity in the material description:

$$\mathbf{V}(\mathbf{X}, t) = \frac{\partial \mathbf{x}}{\partial t} = \begin{cases} \frac{\partial x_1}{\partial t} = e^t X_1 + e^{-t} X_2 \\ \frac{\partial x_2}{\partial t} = e^t X_1 - e^{-t} X_2 \\ \frac{\partial x_3}{\partial t} = 0 \end{cases}$$

which can also be written in matrix form

$$\mathbf{V} = \begin{bmatrix} e^t X_1 + e^{-t} X_2 \\ e^t X_1 - e^{-t} X_2 \\ 0 \end{bmatrix}$$

Acceleration in the material description writes:

$$\mathbf{A}(\mathbf{X}, t) = \frac{\partial \mathbf{V}(\mathbf{X}, t)}{\partial t} = \begin{cases} \frac{\partial V_1}{\partial t} = e^t X_1 + e^{-t} X_2 \\ \frac{\partial V_2}{\partial t} = e^t X_1 - e^{-t} X_2 \\ \frac{\partial V_3}{\partial t} = 0 \end{cases}$$

which can also be written in matrix form

$$\mathbf{A} = \begin{bmatrix} e^t X_1 - e^{-t} X_2 \\ e^t X_1 + e^{-t} X_2 \\ 0 \end{bmatrix}$$

Rearranging the given condition we can write:

$$\mathbf{X} = \begin{cases} X_1 = \frac{1}{2e^t}(x_1 + x_2) \\ X_2 = \frac{1}{2e^{-t}}(x_2 - x_1) \\ X_3 = x_3 \end{cases}$$

we can hence write the velocity and acceleration in the spatial description:

$$\mathbf{v}(\mathbf{x}, t) = \begin{bmatrix} -\frac{1}{2e^t}(x_1 + x_2) \\ \frac{1}{2e^{-t}}(x_2 - x_1) \\ 0 \end{bmatrix}, \quad \mathbf{a}(\mathbf{x}, t) = \begin{bmatrix} \frac{1}{2e^t}(x_1 + x_2) \\ \frac{1}{2e^{-t}}(x_2 - x_1) \\ 0 \end{bmatrix}$$

8. In a deformation of a three-dimensional problem, the displacement components of \mathbf{u} are found to be

$$u_1 = x_1 - \frac{1}{4}x_2, u_2 = x_1 + 2x_2, u_3 = -3x_3$$

- Compute the matrix representations of the deformation gradient and its inverse and show the deformation is isochoric.

Considering $\mathbf{u} = \mathbf{x} - \mathbf{X}$, we have

$$\mathbf{I} - \mathbf{F}^{-1} = \frac{\partial \mathbf{u}}{\partial \mathbf{x}}$$

We can thence calculate \mathbf{F}^{-1} :

$$\mathbf{F}^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} - \begin{bmatrix} 1 & -\frac{1}{4} & 0 \\ 1 & 2 & 0 \\ 0 & 0 & -3 \end{bmatrix} = \begin{bmatrix} 0 & \frac{1}{4} & 0 \\ -1 & -1 & 0 \\ 0 & 0 & 4 \end{bmatrix}$$

By calculating the inverse we get \mathbf{F} :

$$\mathbf{F} = \begin{bmatrix} -4 & -1 & 0 \\ 4 & 0 & 0 \\ 0 & 0 & \frac{1}{4} \end{bmatrix}$$

We can then calculate the determinant: $\det \mathbf{F} = 4 \frac{1}{4} = 1$. So we can say that the deformation is isochoric.

- Determine the components of the material and spatial strain tensors \mathbf{C} , \mathbf{E} and \mathbf{b} , \mathbf{e} .

We can calculate material strain tensor from \mathbf{F} :

$$\mathbf{E} = \frac{1}{2} (\mathbf{F}^T \mathbf{F} - \mathbf{I}) = \begin{bmatrix} 15.5 & 2 & 0 \\ 2 & 0 & 0 \\ 0 & 0 & -0.46875 \end{bmatrix}$$

The spatial strain tensor:

$$\mathbf{e} = \frac{1}{2} (\mathbf{I} - \mathbf{F}^T \mathbf{F}^{-1}) = \begin{bmatrix} 2.5 & 2.5 & 0 \\ 0 & 0.625 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

The right Cauchy–Green deformation tensor

$$\mathbf{C} = \mathbf{F}^T \mathbf{F} = \begin{bmatrix} 17 & -16 & 0 \\ -16 & 16 & 0 \\ 0 & 0 & 0.0625 \end{bmatrix}$$

The left Cauchy–Green deformation tensor

$$\mathbf{b} = \mathbf{F} \mathbf{F}^T = \begin{bmatrix} 32 & 4 & 0 \\ 4 & 1 & 0 \\ 0 & 0 & 0.0625 \end{bmatrix}$$

The results are calculated from the following codes:

```

1 F = np.linalg.inv(np.array([[0, 1/4, 0], [-1, -1, 0], [0, 0, 4]]))
2 E = .5*(np.dot(np.transpose(F), F) - np.eye(3)); print(E)
3 e = .5*(np.eye(3) - np.dot(np.transpose(F), np.linalg.inv(F))); print(e)
4 b = np.dot(F, np.transpose(F)); print(b)
5 C = np.dot(np.transpose(F), F); print(C)

```

9. Since the strain energy function, Ψ of an isotropic hyperelastic material is an invariant, we may regard it a function of the principal stretches λ_a , $a = 1, 2, 3$ and thus write

$$\Psi = \Psi(\mathbf{C}) = \Psi(\lambda_1, \lambda_2, \lambda_3)$$

- Stating the necessary conditions derive the three principal Cauchy stress components (note: σ_a), along with the principal 1st and 2nd Piola-Kirchhoff stresses.

To derive the three principal Cauchy stress components, one can first write out the first Piola-Kirchhoff stress tensor from the strain energy function Ψ by³:

$$\mathbf{P} = \frac{\partial \Psi}{\partial \mathbf{F}}$$

One can then get the Cauchy stress⁴:

$$\sigma = J^{-1} \mathbf{P} \mathbf{F}^T$$

where $J = \det \mathbf{F}$. By computing the eigenvalues of the σ one can hence obtain the three principal Cauchy stress components.

Based on σ , one can also compute the second Piola-Kirchhoff stress:

$$\mathbf{S} = J \mathbf{F}^{-1} \sigma \mathbf{F}^{-T}$$

From the above equations, we know that one should know \mathbf{F} and its relationship to Ψ to obtain the corresponding components.

- Formulate the strain energy function Ψ of an incompressible isotropic hyperelastic material in terms of principal stretches and obtain the three principal Cauchy stress components, along with the principal 1st and 2nd Piola-Kirchhoff stresses.

Recall the strain energy function of an incompressible isotropic hyperelastic material, we may assume a simple strain energy function

$$\begin{aligned} \Psi(I_1, I_2, I_3) &= \frac{\mu}{2}(I_1 - 3) + \frac{\lambda}{2}(I_2 - 3)^2 \\ &= \frac{\mu}{2}(\lambda_1^2 + \lambda_2^2 + \lambda_3^2 - 3) \end{aligned}$$

According to the chain rule, we have

$$\frac{\partial \Psi}{\partial \mathbf{C}} = \sum_i^3 \frac{\partial \Psi}{\partial \lambda_i^2} \frac{\partial \lambda_i^2}{\partial \mathbf{C}}$$

Since $\frac{\partial \lambda_i^2}{\partial \mathbf{C}} = \mathbf{N}_i \otimes \mathbf{N}_i$, we have

$$\frac{\partial \Psi}{\partial \mathbf{C}} = \sum_i^3 \frac{\partial \Psi}{\partial \lambda_i^2} \mathbf{N}_i \otimes \mathbf{N}_i$$

³Ref.: <https://www.cs.toronto.edu/jacobson/seminar/sifakis-course-notes-2012.pdf>

⁴Ref.: https://pke1015.connect.amazon.auckland.ac.nz/SolidMechanicsBooks/Part_III/Chapter_3_Stress_Mass_Momentum/Stress_Balance_Principles_05_Stress_Measures_NonLinear.pdf

We can then compute the second Piola-Kirchhoff stress

$$\begin{aligned}
\mathbf{S} &= \sum_i^3 \frac{1}{\lambda_i} \frac{\partial \Psi}{\partial \lambda_i} \mathbf{N}_i \otimes \mathbf{N}_i \\
&= \frac{\mu}{2} \sum_i^3 \frac{1}{\lambda_i} \frac{\partial (\lambda_1^2 + \lambda_2^2 + \lambda_3^2 - 3)}{\partial \lambda_i} \mathbf{N}_i \otimes \mathbf{N}_i \\
&= \mu (\mathbf{N}_1 \otimes \mathbf{N}_1 + \mathbf{N}_2 \otimes \mathbf{N}_2 + \mathbf{N}_3 \otimes \mathbf{N}_3)
\end{aligned}$$

from which we can also compute the first Piola-Kirchhoff stress

$$\begin{aligned}
\mathbf{P} &= \mathbf{F}\mathbf{S} \\
&= \sum_i^3 \frac{\partial \Psi}{\partial \lambda_i} \mathbf{N}_i \otimes \mathbf{N}_i \\
&= \mu (\lambda_1 \mathbf{N}_1 \otimes \mathbf{N}_1 + \lambda_2 \mathbf{N}_2 \otimes \mathbf{N}_2 + \lambda_3 \mathbf{N}_3 \otimes \mathbf{N}_3)
\end{aligned}$$

We can then compute the Cauchy stress

$$\begin{aligned}
\sigma &= J^{-1} \mathbf{F}\mathbf{P}^\top \\
&= \frac{1}{\lambda_1 \lambda_2 \lambda_3} \sum_i^3 \frac{\partial \Psi}{\partial \lambda_i} \lambda_i \mathbf{N}_i \otimes \mathbf{N}_i \\
&= \frac{\mu}{\lambda_1 \lambda_2 \lambda_3} (\lambda_1^2 \mathbf{N}_1 \otimes \mathbf{N}_1 + \lambda_2^2 \mathbf{N}_2 \otimes \mathbf{N}_2 + \lambda_3^2 \mathbf{N}_3 \otimes \mathbf{N}_3)
\end{aligned}$$

For the general case of incompressible hyperelastic materials⁵, we have

$$\begin{aligned}
\sigma &= -p\mathbf{I} + 2\mathbf{F} \cdot \frac{\partial \Psi}{\partial \mathbf{C}} \cdot \mathbf{F}^\top \\
\mathbf{P} &= -p\mathbf{F}^\top + 2\mathbf{F} \cdot \frac{\partial \Psi}{\partial \mathbf{C}} \\
\mathbf{S} &= -\mathbf{F}^{-1} p \mathbf{F}^\top + 2 \frac{\partial \Psi}{\partial \mathbf{C}}
\end{aligned}$$

10. Consider an incompressible hyperelastic membrane under biaxial deformation with the following kinematic assumptions:

$$x_1 = \lambda_1 X_1, x_2 = \lambda_2 X_2, x_3 = \frac{1}{\lambda_1 \lambda_2} X_3$$

In particular, the two principal stretches λ_1 and λ_2 are given. According to the membrane theory, assume a plane stress state (out-of-plane stress is zero) and specify the Cauchy stresses (note: use results from problem 9) in the plane of the membrane by applying Ogden's strain energy function.

⁵Ref.: https://en.wikipedia.org/wiki/Hyperelastic_material

Applying Ogden's strain energy function⁶, we have

$$\Psi = \sum_i^3 \frac{2\mu_i}{\alpha_i^2} \left(\left(\frac{\lambda_1}{J^{1/3}} \right)^{\alpha_i} + \left(\frac{\lambda_2}{J^{1/3}} \right)^{\alpha_i} + \left(\frac{\lambda_3}{J^{1/3}} \right)^{\alpha_i} \right) + \frac{K_1}{2} (J - 1)^2$$

According to the plane stress assumption, we have $\sigma_3 = 0$.

From the given deformation gradient tensor \mathbf{F} , we know that $J = \det \mathbf{F} = 1$.

From the solution in Prob. 9, we can write out the Cauchy stress:

$$\begin{aligned} \frac{\partial \Psi}{\partial \mathbf{C}} &= \sum_i^3 \frac{\partial \Psi}{\partial \lambda_i^2} \mathbf{N}_i \otimes \mathbf{N}_i \\ &= \sum_i^3 \frac{2\mu_i}{\alpha_i} \left(\frac{1}{J^{1/3}} \right)^{\alpha_i} (\lambda_1^{\alpha_i-2} + \lambda_2^{\alpha_i-2} + \lambda_3^{\alpha_i-2}) \\ &= \sum_i^3 \frac{2\mu_i}{\alpha_i} \left(\lambda_1^{\alpha_i-2} + \lambda_2^{\alpha_i-2} + \left(\frac{1}{\lambda_1 \lambda_2} \right)^{\alpha_i-2} \right) \\ \rightarrow 2\mathbf{F} \frac{\partial \Psi}{\partial \mathbf{C}} \mathbf{F}^T &= \sum_i^3 \frac{4\mu_i}{\alpha_i} \left(\lambda_1^{\alpha_i} + \lambda_2^{\alpha_i} + \left(\frac{1}{\lambda_1 \lambda_2} \right)^{\alpha_i} \right) \end{aligned}$$

We can hence write out the matrix form of the Cauchy stress:

$$\sigma = \begin{bmatrix} \sum_i^3 \frac{4\mu_i}{\alpha_i} \lambda_1 - p & 0 & 0 \\ 0 & \sum_i^3 \frac{4\mu_i}{\alpha_i} \lambda_2 - p & 0 \\ 0 & 0 & \sum_i^3 \frac{4\mu_i}{\alpha_i} \frac{1}{\lambda_1 \lambda_2} - p \end{bmatrix}$$

Since we know that $\sigma_3 = 0$, we have $\sum_i^3 \frac{4\mu_i}{\alpha_i} \frac{1}{\lambda_1 \lambda_2} = p$. We can then write out the form of Cauchy stress:

$$\sigma = \begin{bmatrix} \sum_i^3 \frac{4\mu_i}{\alpha_i} \left(\lambda_1 - \frac{1}{\lambda_1 \lambda_2} \right) & 0 & 0 \\ 0 & \sum_i^3 \frac{4\mu_i}{\alpha_i} \left(\lambda_2 - \frac{1}{\lambda_1 \lambda_2} \right) & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

⁶Ref.: http://solidmechanics.org/text/Chapter3_5/Chapter3_5.htm

MAE7750_HW2

March 10, 2023

1 MAE 7750: HW 2

Hanfeng Zhai

Cornell University

1.1 Hands-on Numerical Implementation

1.1.1 Scenario 1: $x = 1.8$

Newton-Raphson Method

```
[47]: import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
v1_values_NR = [];v2_values_NR = []
G1_values_NR = [];G2_values_NR = []
R1_values_NR = [];R2_values_NR = []
num_iter_NR = []

v_0 = np.transpose(np.array([0.042,0.042]))
lamb = np.arange(0.25, 10, 0.25)
P1 = 1; P2 = 0
v = v_0
x = 1.8
TOL = 1

R0 = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0], v[1]-v[0]]))
P0 = np.transpose(np.array([1,0]))
G0 = R0 - 0 * P0
G_benchmark = np.sqrt(G0[0]**2 + G0[1]**2); print(G_benchmark)
# K_T = np.array([[0.6*v[0]+6, -2*x*v[1]], [-1, 1]]); #print("K_T",K_T)

for lamb_i in lamb:
    ii = 1
    num_iter = 0
    while True:
        # print("check lambda:", lamb_i)
        R = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0],
→v[1]-v[0]])); #print("R value:", R)
```

```

P = np.transpose(np.array([1,0]))
G = R - lamb_i * P;
K_T = np.array([[0.6*v[0]**2+6, -2*x*v[1]], [-1, 1]])
Delta_v = -np.dot(np.linalg.inv(K_T),G);
v = v + Delta_v; #print("v:",v)
abs_G = np.sqrt(G[0]**2 + G[1]**2); #print("abs G:", abs_G)
TOL = abs_G; #print("tol:", TOL); #print("<=====>")
num_iter = num_iter+1
if TOL <= 1e-4 * G_benchmark or ii > 15:
    v1_values_NR.append(v[0]);v2_values_NR.append(v[1])
    G1_values_NR.append(G[0]);G2_values_NR.append(G[1])
    R1_values_NR.append(R[0]);R2_values_NR.append(R[1])
    num_iter_NR.append(num_iter)
    break
ii += 1

v1_values_NR.append(v[0]);v2_values_NR.append(v[1])
G1_values_NR.append(G[0]);G2_values_NR.append(G[1])
R1_values_NR.append(R[0]);R2_values_NR.append(R[1])

v1_loop_NR = np.array(v1_values_NR);v2_loop_NR = np.array(v2_values_NR)
G1_loop_NR = np.array(G1_values_NR);G2_loop_NR = np.array(G2_values_NR)
R1_loop_NR = np.array(R1_values_NR);R2_loop_NR = np.array(R2_values_NR)

```

0.2488396176

Modified Newton-Raphson Method

```

[49]: v1_values_MNR = [];v2_values_MNR = []
G1_values_MNR = [];G2_values_MNR = []
R1_values_MNR = [];R2_values_MNR = []
num_iter_MNR = []

v = v_0

G_benchmark = np.sqrt(G0[0]**2 + G0[1]**2);
K_T = np.array([[0.6*v[0]**2+6, -2*x*v[1]], [-1, 1]]);
Delta_v = []; inv_K = np.linalg.inv(K_T);
for lamb_i in lamb:
    ii = 1
    num_iter = 0
    while True:
        R = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0],
→v[1]-v[0]])); #print("R value:", R)
        P = np.transpose(np.array([1,0]))
        G = R - lamb_i * P; #print(G)
        Delta_v = -np.dot(inv_K,G);
        v = v + Delta_v;

```

```

abs_G = np.sqrt(G[0]**2 + G[1]**2);
TOL = abs_G;
num_iter = num_iter+1
if TOL <= 1e-4 * G_benchmark or ii > 15:
    v1_values_MNR.append(v[0]);v2_values_MNR.append(v[1])
    G1_values_MNR.append(G[0]);G2_values_MNR.append(G[1])
    R1_values_MNR.append(R[0]);R2_values_MNR.append(R[1])
    num_iter_MNR.append(num_iter)
    break
ii += 1

v1_values_MNR.append(v[0]);v2_values_MNR.append(v[1])
G1_values_MNR.append(G[0]);G2_values_MNR.append(G[1])
R1_values_MNR.append(R[0]);R2_values_MNR.append(R[1])

v1_loop_MNR = np.array(v1_values_MNR);v2_loop_MNR = np.array(v2_values_MNR)
G1_loop_MNR = np.array(G1_values_MNR);G2_loop_MNR = np.array(G2_values_MNR)
R1_loop_MNR = np.array(R1_values_MNR);R2_loop_MNR = np.array(R2_values_MNR)

```

Modified Newton-Raphson Method with Line Search

```

[50]: import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl

x=1.8
lamb = np.arange(0.25, 10, 0.25)

v1_values_MNRLS = [];v2_values_MNRLS = []
G1_values_MNRLS = [];G2_values_MNRLS = []
R1_values_MNRLS = [];R2_values_MNRLS = []
num_iter_MNRLS = []
v_0 = np.transpose(np.array([0,0]))
v = v_0
R0 = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0], v[1]-v[0]]))
P0 = np.transpose(np.array([1,0]))
G0 = R0 - 0 * P0
G_benchmark = np.sqrt(G0[0]**2 + G0[1]**2);
K_T = np.array([[0.6*v[0]+6, -2*x*v[1]], [-1, 1]]);
Delta_v = []; inv_K = np.linalg.inv(K_T);

alpha_0 = 0; alpha_1 = 1; alpha = []

def g_func(alpha, Delta_v, lamb, vv):
    R = np.transpose(np.array([0.2*(vv[0] + alpha*Delta_v[0])**3 - x*(vv[1] +
    ↪alpha*Delta_v[1])**2 + 6*(vv[0] + alpha*Delta_v[0]),\

```

```

                                (vv[1] + alpha*Delta_v[1])-(vv[0] +
→alpha*Delta_v[0]))
    P = np.transpose(np.array([1,0])); G = R - lamb * P;
→#print("R,G,P,alpha,Delta_v:",R,G,P,alpha,Delta_v)
    output = np.dot(Delta_v, G); #print("g function:",output)
    return output

for lamb_i in lamb:
    ii = 1
    alpha_prev = alpha_0; alpha = alpha_1

    Delta_v = -np.dot(inv_K, G0)
    num_iter = 0

    while True:
        R = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0],
→v[1]-v[0]]));
        P = np.transpose(np.array([1,0]))
        G = R - lamb_i * P; Delta_v = -np.dot(inv_K,G);
        denom = g_func(alpha, Delta_v, lamb_i, v) - g_func(alpha_prev, Delta_v,
→lamb_i, v)
        interval = g_func(alpha, Delta_v, lamb_i, v) * ( (alpha - alpha_prev) /
→denom )
        alpha_prev = alpha
        alpha = alpha - interval
        v = v + alpha * Delta_v
        num_iter = num_iter + 1
        if np.abs(g_func(alpha, Delta_v, lamb_i, v)) > 0.5 * np.abs(g_func(0,
→Delta_v, lamb_i, v)) or ii > 5:
            v1_values_MNRLS.append(v[0]);v2_values_MNRLS.append(v[1])
            G1_values_MNRLS.append(G[0]);G2_values_MNRLS.append(G[1])
            R1_values_MNRLS.append(R[0]);R2_values_MNRLS.append(R[1])
            num_iter_MNRLS.append(num_iter); #print(np.abs(g_func(alpha,
→Delta_v, lamb_i, v)), "/", 0.5 * np.abs(g_func(0, Delta_v, lamb_i, v)))
            break
        ii += 1

    v1_values_MNRLS.append(v[0]);v2_values_MNRLS.append(v[1])
    G1_values_MNRLS.append(G[0]);G2_values_MNRLS.append(G[1])
    R1_values_MNRLS.append(R[0]);R2_values_MNRLS.append(R[1])

v1_loop_MNRLS = np.array(v1_values_MNRLS);v2_loop_MNRLS = np.
→array(v2_values_MNRLS)
G1_loop_MNRLS = np.array(G1_values_MNRLS);G2_loop_MNRLS = np.
→array(G2_values_MNRLS)

```

```
R1_loop_MNRLS = np.array(R1_values_MNRLS);R2_loop_MNRLS = np.
↳array(R2_values_MNRLS)
```

BFGS Method

```
[52]: import numpy as np
import sympy as sym
import matplotlib.pyplot as plt
import matplotlib as mpl

v1_values_BFGS = [];v2_values_BFGS = []
G1_values_BFGS = [];G2_values_BFGS = []
R1_values_BFGS = [];R2_values_BFGS = []
num_iter_BFGS = []

v_0 = np.transpose(np.array([0,0]))
lamb = np.arange(0.5, 10, 0.25)
v = v_0
x = 1.8
TOL = 1

R0 = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0], v[1]-v[0]]))
P0 = np.transpose(np.array([1,0]))
G0 = R0 - 0 * P0
G_benchmark = np.sqrt(G0[0]**2 + G0[1]**2)
P = np.transpose(np.array([1,0]))

left_temp_ab = np.ones([2,2]);
right_temp_ba = np.ones([2,2]);

for lamb_i in lamb:
    ii = 1; K_T = np.array([[0.6*v[0]**2+6, -2*x*v[1]], [-1, 1]])
    num_iter = 0
    # We need to initialize the BFGS with Newton-Raphson
    H_prev = np.linalg.inv(K_T);
    R = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0], v[1]-v[0]]));
    G = R - lamb_i * P;
    Delta_v_NR = -np.dot(np.linalg.inv(K_T),G);
    v_prev = v; v = v + Delta_v_NR;
    R = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0], v[1]-v[0]]));
    G_prev = G; G = R - lamb_i * P;
    while True:
        w = v - v_prev; g = G - G_prev;

        a_temp_1 = np.matmul(np.transpose(g),w);
        a_temp_2 = 1/a_temp_1;
        a = a_temp_2*w;
```



```

b_temp_1 = - np.matmul(np.transpose(w), g);
b_temp_2 = np.matmul(np.transpose(w), G_prev);
b_temp_3 = b_temp_1/b_temp_2;
b_temp_4 = np.sqrt(b_temp_3);
b_temp_5 = b_temp_4 * G_prev
b = - (g - b_temp_5);

left_temp_ab[0][0] = a[0]*b[0]; left_temp_ab[0][1] = a[0]*b[1];
left_temp_ab[1][0] = a[1]*b[0]; left_temp_ab[1][1] = a[1]*b[1];
left_temp = np.eye(2)+left_temp_ab;

right_temp_ba[0][0] = b[0]*a[0];right_temp_ba[0][1] = b[0]*a[1];
right_temp_ba[1][0] = b[1]*a[0];right_temp_ba[1][1] = b[1]*a[1];
right_temp = np.eye(2)+right_temp_ba;

H = left_temp @ H_prev @ right_temp; H_prev = H;

Delta_v_BFGS = -np.matmul(H, G)
v_prev = v; v = v + Delta_v_BFGS
R = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0],
→v[1]-v[0]])); G = R - lamb_i * P
abs_G = np.sqrt(G[0]**2 + G[1]**2);
TOL = abs_G;num_iter = num_iter+1
if TOL <= 1e-4 * G_benchmark or ii > 15:
    v1_values_BFGS.append(v[0]); v2_values_BFGS.append(v[1])
    G1_values_BFGS.append(G[0]); G2_values_BFGS.append(G[1])
    R1_values_BFGS.append(R[0]); R2_values_BFGS.append(R[1])
    num_iter_BFGS.append(num_iter)
    break
ii += 1

v1_values_BFGS.append(v[0]);v2_values_BFGS.append(v[1])
G1_values_BFGS.append(G[0]);G2_values_BFGS.append(G[1])
R1_values_BFGS.append(R[0]);R2_values_BFGS.append(R[1])

v1_loop_BFGS = np.array(v1_values_BFGS);v2_loop_BFGS = np.array(v2_values_BFGS)
G1_loop_BFGS = np.array(G1_values_BFGS);G2_loop_BFGS = np.array(G2_values_BFGS)
R1_loop_BFGS = np.array(R1_values_BFGS);R2_loop_BFGS = np.array(R2_values_BFGS)

```

Analytical Solution

```

[44]: import sympy as sym

v_analy, lamb_i= sym.symbols('v_analy lamb_i')

eq = sym.Eq(0.2*v_analy**3 - 1.8*v_analy**2 + 6*v_analy - lamb_i, 0)
v_analy_solns = []

```

```

lamb = np.arange(0, 10, 0.25)
for lamb_i_val in lamb:
    solns = sym.solve(eq.subs(lamb_i, lamb_i_val), v_analy)
    solns = solns[0]; #print(solns)
    v_analy_solns.append(solns)

v_analy_solns = np.array(v_analy_solns)

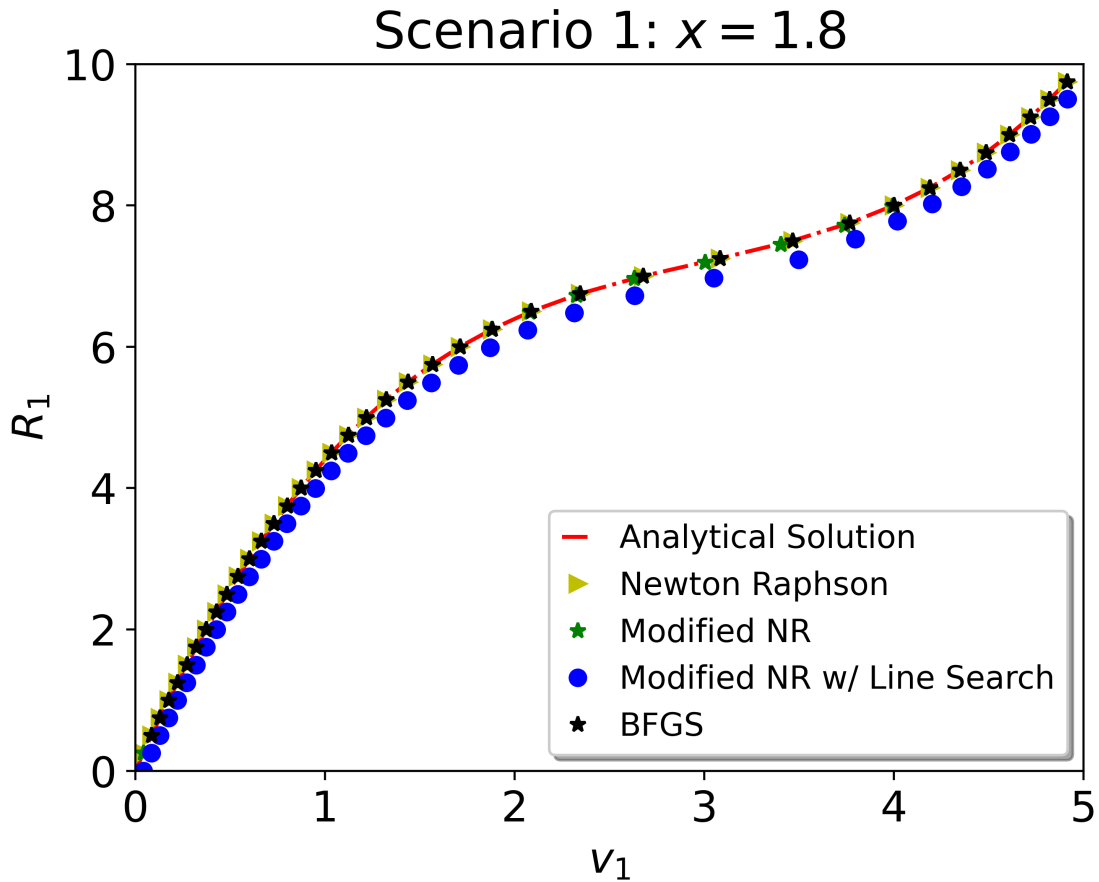
R_analy = 0.2*v_analy_solns**3 - 1.8*v_analy_solns**2 + 6*v_analy_solns

```

```

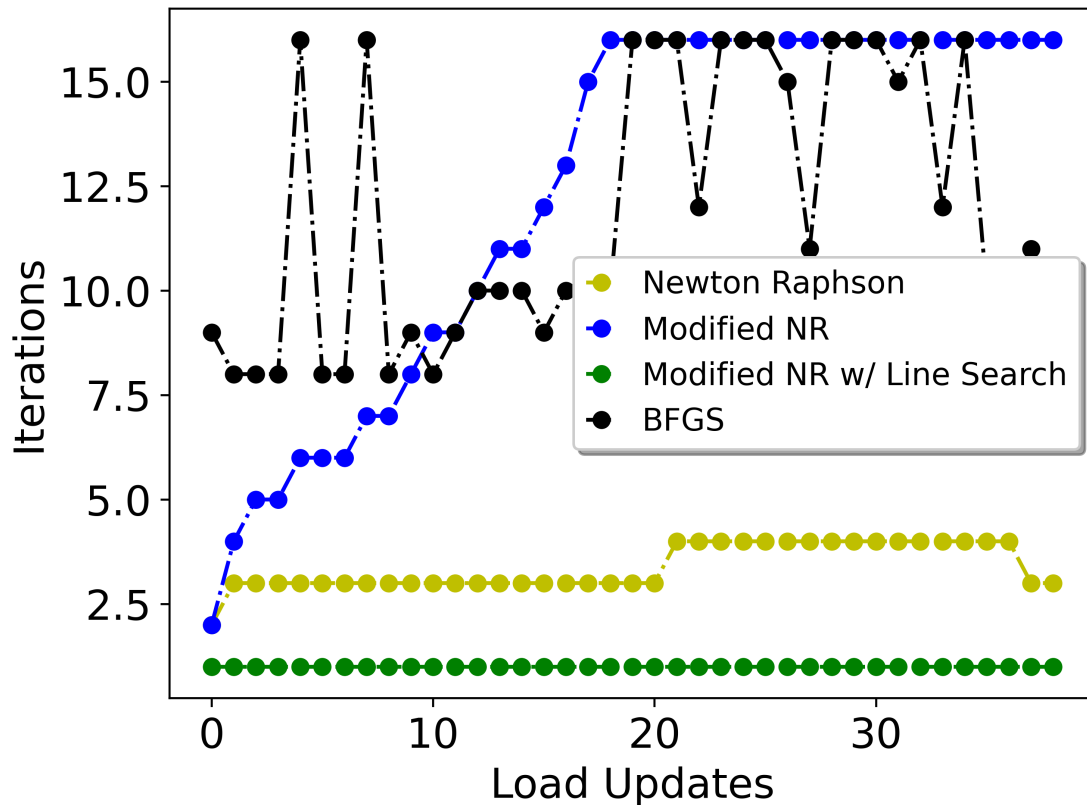
[53]: import matplotlib as mpl
plt.plot(v_analy_solns,R_analy,'r-.',label='Analytical Solution')
plt.plot(v1_loop_NR,R1_loop_NR,'y>',label='Newton Raphson')
plt.plot(v1_loop_MNR,R1_loop_MNR,'g*',label='Modified NR')
plt.plot(v1_loop_MNRLS,R1_loop_MNRLS,'bo',label='Modified NR w/ Line Search')
plt.plot(v1_loop_BFGS,R1_loop_BFGS,'k*',label='BFGS')
plt.xlabel("$v_1$")
plt.ylabel("$R_1$")
plt.ylim([0,10])
plt.xlim([0,5])
plt.title("Scenario 1: $x = 1.8$")
plt.legend(shadow=True, handlelength=1, fontsize=12)
plt.rcParams['figure.dpi'] = 500
plt.show()
plt.figure(figsize=(5, 3)); mpl.rcParams.update({'font.size': 16})

```



<Figure size 2500x1500 with 0 Axes>

```
[56]: plt.plot(num_iter_NR, 'yo-.', label='Newton Raphson')
plt.plot(num_iter_MNR, 'bo-.', label='Modified NR')
plt.plot(num_iter_MNRLS, 'go-.', label='Modified NR w/ Line Search')
plt.plot(num_iter_BFGS, 'ko-.', label='BFGS')
plt.xlabel("Load Updates")
plt.ylabel("Iterations")
plt.legend(shadow=True, handlelength=1, fontsize=12)
plt.rcParams['figure.dpi'] = 500
```



1.1.2 Scenario 2: $x = 2.1$

Newton-Raphson Method

```
[25]: import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
v1_values_NR = [];v2_values_NR = []
G1_values_NR = [];G2_values_NR = []
R1_values_NR = [];R2_values_NR = []
num_iter_NR = []

v_0 = np.transpose(np.array([0.042,0.042]))
lamb = np.arange(0.25, 10, 0.25)
v = v_0
x = 2.1
TOL = 1

R0 = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0], v[1]-v[0]]))
P0 = np.transpose(np.array([1,0]))
G0 = R0 - 0 * P0
G_benchmark = np.sqrt(G0[0]**2 + G0[1]**2); print(G_benchmark)
```

```

for lamb_i in lamb:
    ii = 1
    num_iter = 0
    while True:
        # print("check lambda:", lamb_i)
        R = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0],
→v[1]-v[0]])); #print("R value:", R)
        P = np.transpose(np.array([1,0]))
        G = R - lamb_i * P; #print("G value:", G)
        K_T = np.array([[0.6*v[0]**2+6, -2*x*v[1]], [-1, 1]])
        Delta_v = -np.dot(np.linalg.inv(K_T),G); #print("Delta v:", Delta_v)#-np.
→dot(np.linalg.inv(K_T), G);
        # print("K_T:", K_T); print("inverse K_T:", np.linalg.inv(K_T))
        v = v + Delta_v; #print("v:",v)
        abs_G = np.sqrt(G[0]**2 + G[1]**2); #print("abs G:", abs_G)
        TOL = abs_G; #print("tol:", TOL); #print("<=====>")
        num_iter = num_iter+1
        if TOL <= 1e-4 * G_benchmark or ii > 15:
            v1_values_NR.append(v[0]);v2_values_NR.append(v[1])
            G1_values_NR.append(G[0]);G2_values_NR.append(G[1])
            R1_values_NR.append(R[0]);R2_values_NR.append(R[1])
            num_iter_NR.append(num_iter)
            break
        ii += 1

    v1_values_NR.append(v[0]);v2_values_NR.append(v[1])
    G1_values_NR.append(G[0]);G2_values_NR.append(G[1])
    R1_values_NR.append(R[0]);R2_values_NR.append(R[1])

v1_loop_NR = np.array(v1_values_NR);v2_loop_NR = np.array(v2_values_NR)
G1_loop_NR = np.array(G1_values_NR);G2_loop_NR = np.array(G2_values_NR)
R1_loop_NR = np.array(R1_values_NR);R2_loop_NR = np.array(R2_values_NR)

```

0.2483104176

Modified Newton-Raphson Method

```

[26]: import numpy as np

v1_values_MNR = [];v2_values_MNR = []
G1_values_MNR = [];G2_values_MNR = []
R1_values_MNR = [];R2_values_MNR = []
num_iter_MNR = []
x = 2.1
v_0 = np.transpose(np.array([0.042,0.042]))
lamb = np.arange(0.25, 10, 0.25)
v = v_0

```

```

R0 = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0], v[1]-v[0]]))
P0 = np.transpose(np.array([1,0]))
G0 = R0 - 0 * P0
G_benchmark = np.sqrt(G0[0]**2 + G0[1]**2); #print("tolerance:",
→G_benchmark*1e-4)

K_T = np.array([[0.6*v[0]**2+6, -2*x*v[1]], [-1, 1]]);
Delta_v = []; inv_K = np.linalg.inv(K_T); #print("inv K:", inv_K)
for lamb_i in lamb:
    ii = 1
    num_iter = 0
    while True:
        # print("check lambda:", lamb_i)
        R = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0],
→v[1]-v[0]])); #print("R value:", R)
        P = np.transpose(np.array([1,0]))
        G = R - lamb_i * P; #print("G value:", G, "/", "inv_K:", inv_K)
        Delta_v = -np.dot(inv_K,G);
        v = v + Delta_v; #print("v value:", v, "/", "Delta v:", Delta_v)
        abs_G = np.sqrt(G[0]**2 + G[1]**2);
        TOL = abs_G; #print("=====", TOL, "=====")
        num_iter = num_iter + 1
        if TOL <= 1e-4 * G_benchmark or ii > 15:
            v1_values_MNR.append(v[0]);v2_values_MNR.append(v[1])
            G1_values_MNR.append(G[0]);G2_values_MNR.append(G[1])
            R1_values_MNR.append(R[0]);R2_values_MNR.append(R[1])
            num_iter_NR.append(num_iter)
            break
        ii += 1

    v1_values_MNR.append(v[0]);v2_values_MNR.append(v[1])
    G1_values_MNR.append(G[0]);G2_values_MNR.append(G[1])
    R1_values_MNR.append(R[0]);R2_values_MNR.append(R[1])

v1_loop_MNR = np.array(v1_values_MNR);v2_loop_MNR = np.array(v2_values_MNR)
G1_loop_MNR = np.array(G1_values_MNR);G2_loop_MNR = np.array(G2_values_MNR)
R1_loop_MNR = np.array(R1_values_MNR);R2_loop_MNR = np.array(R2_values_MNR)

```

Modified Newton-Raphson Method with Line Search

```

[27]: import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl

lamb = np.arange(0.25, 10, 0.25)

```

```

v1_values_MNRLS = [];v2_values_MNRLS = []
G1_values_MNRLS = [];G2_values_MNRLS = []
R1_values_MNRLS = [];R2_values_MNRLS = []
num_iter_MNRLS = []
v_0 = np.transpose(np.array([0,0]))
v = v_0
R0 = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0], v[1]-v[0]]))
P0 = np.transpose(np.array([1,0]))
G0 = R0 - 0 * P0
G_benchmark = np.sqrt(G0[0]**2 + G0[1]**2);
K_T = np.array([[0.6*v[0]**2+6, -2*x*v[1]], [-1, 1]]);
Delta_v = []; inv_K = np.linalg.inv(K_T);

alpha_0 = 0; alpha_1 = 1; alpha = []

def g_func(alpha, Delta_v, lamb, vv):
    R = np.transpose(np.array([0.2*(vv[0] + alpha*Delta_v[0])**3 - x*(vv[1] +
→alpha*Delta_v[1])**2 + 6*(vv[0] + alpha*Delta_v[0]),\
                                (vv[1] + alpha*Delta_v[1])-(vv[0] +
→alpha*Delta_v[0])]))
    P = np.transpose(np.array([1,0])); G = R - lamb * P;
→#print("R,G,P,alpha,Delta_v:",R,G,P,alpha,Delta_v)
    output = np.dot(Delta_v, G); #print("g function:",output)
    return output

for lamb_i in lamb:
    ii = 1
    alpha_prev = alpha_0; alpha = alpha_1
    Delta_v = -np.dot(inv_K, G0)
    num_iter = 0

    while True:
        R = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0],
→v[1]-v[0]]));
        P = np.transpose(np.array([1,0]))
        G = R - lamb_i * P; Delta_v = -np.dot(inv_K,G);
        denom = g_func(alpha, Delta_v, lamb_i, v) - g_func(alpha_prev, Delta_v,
→lamb_i, v)
        interval = g_func(alpha, Delta_v, lamb_i, v) * ( (alpha - alpha_prev) /
→denom )

        alpha_prev = alpha
        alpha = alpha - interval
        v = v + alpha * Delta_v
        num_iter = num_iter+1

```

```

        if np.abs(g_func(alpha, Delta_v, lamb_i, v)) > 0.5 * np.abs(g_func(0,
→Delta_v, lamb_i, v)) or ii > 5:
            v1_values_MNRLS.append(v[0]);v2_values_MNRLS.append(v[1])
            G1_values_MNRLS.append(G[0]);G2_values_MNRLS.append(G[1])
            R1_values_MNRLS.append(R[0]);R2_values_MNRLS.append(R[1])
            num_iter_MNRLS.append()
            break
        ii += 1

    v1_values_MNRLS.append(v[0]);v2_values_MNRLS.append(v[1])
    G1_values_MNRLS.append(G[0]);G2_values_MNRLS.append(G[1])
    R1_values_MNRLS.append(R[0]);R2_values_MNRLS.append(R[1])

v1_loop_MNRLS = np.array(v1_values_MNRLS);v2_loop_MNRLS = np.
→array(v2_values_MNRLS)
G1_loop_MNRLS = np.array(G1_values_MNRLS);G2_loop_MNRLS = np.
→array(G2_values_MNRLS)
R1_loop_MNRLS = np.array(R1_values_MNRLS);R2_loop_MNRLS = np.
→array(R2_values_MNRLS)

```

BFGS Method

```

[61]: import numpy as np
import sympy as sym
import matplotlib.pyplot as plt
import matplotlib as mpl

v1_values_BFGS = [];v2_values_BFGS = []
G1_values_BFGS = [];G2_values_BFGS = []
R1_values_BFGS = [];R2_values_BFGS = []
num_iter_BFGS = []
v_0 = np.transpose(np.array([0.042,0.042]))
lamb = np.arange(0.5, 10, 0.25)
v = v_0
TOL = 1

R0 = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0], v[1]-v[0]]))
P0 = np.transpose(np.array([1,0]))
G0 = R0 - 0 * P0
G_benchmark = np.sqrt(G0[0]**2 + G0[1]**2); print(G_benchmark)
P = np.transpose(np.array([1,0]))

left_temp_ab = np.ones([2,2]);
right_temp_ba = np.ones([2,2]);

for lamb_i in lamb:
    ii = 1; K_T = np.array([[0.6*v[0]**2+6, -2*x*v[1]], [-1, 1]])

```



```

num_iter = 0
# We need to initialize the BFGS with Newton-Raphson
H_prev = np.linalg.inv(K_T);
R = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0], v[1]-v[0]]));
G = R - lamb_i * P;
Delta_v_NR = -np.dot(np.linalg.inv(K_T),G);
v_prev = v; v = v + Delta_v_NR;
R = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0], v[1]-v[0]]));
G_prev = G; G = R - lamb_i * P;
while True:
    w = v - v_prev; g = G - G_prev; #print("w:", w, "|", "g:", g)

    a_temp_1 = np.matmul(np.transpose(g),w); #print("a temp 1 val.:",  

→a_temp_1)
    a_temp_2 = 1/a_temp_1; #print("a temp 2 val.:", a_temp_2)
    a = a_temp_2*w; #print("a val.:", a)

    b_temp_1 = - np.matmul(np.transpose(w), g); #print("b temp 1 val.:",  

→b_temp_1)
    b_temp_2 = np.matmul(np.transpose(w), G_prev); #print("b temp 2 val.:",  

→b_temp_2)
    b_temp_3 = b_temp_1/b_temp_2; #print("b temp 3 val.:", b_temp_3)
    b_temp_4 = np.sqrt(b_temp_3); #print("b temp 4 val.:", b_temp_4)
    b_temp_5 = b_temp_4 * G_prev#np.matmul(b_temp_4, G_prev)
    b = - (g - b_temp_5); #print("b val.:", b)

    left_temp_ab[0][0] = a[0]*b[0]; left_temp_ab[0][1] = a[0]*b[1];
    left_temp_ab[1][0] = a[1]*b[0]; left_temp_ab[1][1] = a[1]*b[1];
    left_temp = np.eye(2)+left_temp_ab;

    right_temp_ba[0][0] = b[0]*a[0];right_temp_ba[0][1] = b[0]*a[1];
    right_temp_ba[1][0] = b[1]*a[0];right_temp_ba[1][1] = b[1]*a[1];
    right_temp = np.eye(2)+right_temp_ba;

    H = left_temp @ H_prev @ right_temp; H_prev = H;

    Delta_v_BFGS = -np.matmul(H, G)
    v_prev = v; v = v + Delta_v_BFGS
    R = np.transpose(np.array([0.2*v[0]**3 - x*v[1]**2 + 6*v[0],  

→v[1]-v[0]])); G = R - lamb_i * P
    abs_G = np.sqrt(G[0]**2 + G[1]**2);
    TOL = abs_G;
    num_iter = num_iter + 1
    if TOL <= 1e-4 * G_benchmark or ii > 15:
        v1_values_BFGS.append(v[0]); v2_values_BFGS.append(v[1])
        G1_values_BFGS.append(G[0]); G2_values_BFGS.append(G[1])

```

```

        R1_values_BFGS.append(R[0]); R2_values_BFGS.append(R[1])
        num_iter_BFGS.append(num_iter)
        break
    ii += 1

v1_values_BFGS.append(v[0]);v2_values_BFGS.append(v[1])
G1_values_BFGS.append(G[0]);G2_values_BFGS.append(G[1])
R1_values_BFGS.append(R[0]);R2_values_BFGS.append(R[1])

v1_loop_BFGS = np.array(v1_values_BFGS);v2_loop_BFGS = np.array(v2_values_BFGS)
G1_loop_BFGS = np.array(G1_values_BFGS);G2_loop_BFGS = np.array(G2_values_BFGS)
R1_loop_BFGS = np.array(R1_values_BFGS);R2_loop_BFGS = np.array(R2_values_BFGS)

```

0.2488396176

Analytical Solution

```

[30]: import sympy as sym
v_analy, lamb_i= sym.symbols('v_analy lamb_i')

eq = sym.Eq(0.2*v_analy**3 - 2.1*v_analy**2 + 6*v_analy - lamb_i, 0)
v_analy_solns = []
lamb = np.arange(0, 10, 0.25)
for lamb_i_val in lamb:
    solns = sym.solve(eq.subs(lamb_i, lamb_i_val), v_analy)
    solns = solns[0];
    v_analy_solns.append(solns)

v_analy = np.array([0.0, 0.0422901040533185, 0.0858944598074010, 0.
→130924633802590, 0.177508467636853, 0.225793572228660, 0.275951844787766, 0.
→328185398807908, 0.382734486316489, 0.439888294466496, 0.500000000000000, 0.
→563508326896292, 0.630969399311139, 0.703105604423934, 0.780884053088076, 0.
→865649959190717, 0.959370638050928, 1.06512801561128, 1.18826230851010, 1.
→33968215690638, 1.55051025721682, 6.51227858156787, 6.57176672599949, 6.
→62837364425313, 6.68243824065754, 6.73423887266998, 6.78400740883051, 6.
→83193935969245, 6.87820134291152, 6.92293669160797, 6.96626974021598, 7.
→00830914952264, 7.04915052130478, 7.08887847939054, 7.12756834425001, 7.
→16528749395310, 7.20209648029536, 7.23804995175773, 7.27319742257035, 7.
→30758391806220])

R_analy = 0.2*v_analy**3 - x*v_analy**2 + 6*v_analy

```

```

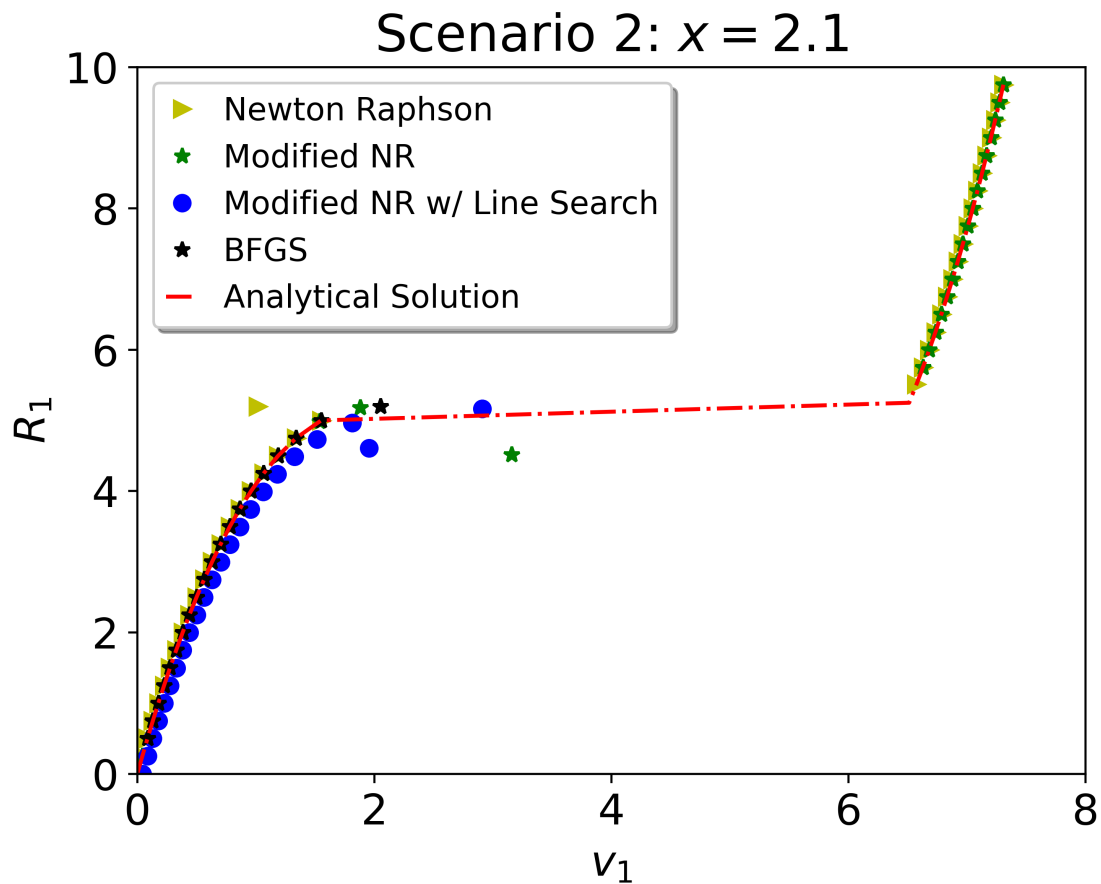
[33]: import matplotlib as mpl
plt.plot(v1_loop_NR,R1_loop_NR,'y>',label='Newton Raphson')
plt.plot(v1_loop_MNR,R1_loop_MNR,'g*',label='Modified NR')
plt.plot(v1_loop_MNRLS,R1_loop_MNRLS,'bo',label='Modified NR w/ Line Search')
plt.plot(v1_loop_BFGS,R1_loop_BFGS,'k*',label='BFGS')

```

```

plt.plot(v_analy,R_analy,'r-.',label='Analytical Solution')
plt.xlabel("$v_1$")
plt.ylabel("$R_1$")
plt.ylim([0,10])
plt.xlim([0,8])
plt.title("Scenario 2: $x=2.1$")
plt.legend(shadow=True, handlelength=1, fontsize=12)
plt.rcParams['figure.dpi'] = 500
plt.show()
plt.figure(figsize=(5, 3)); mpl.rcParams.update({'font.size': 16})

```



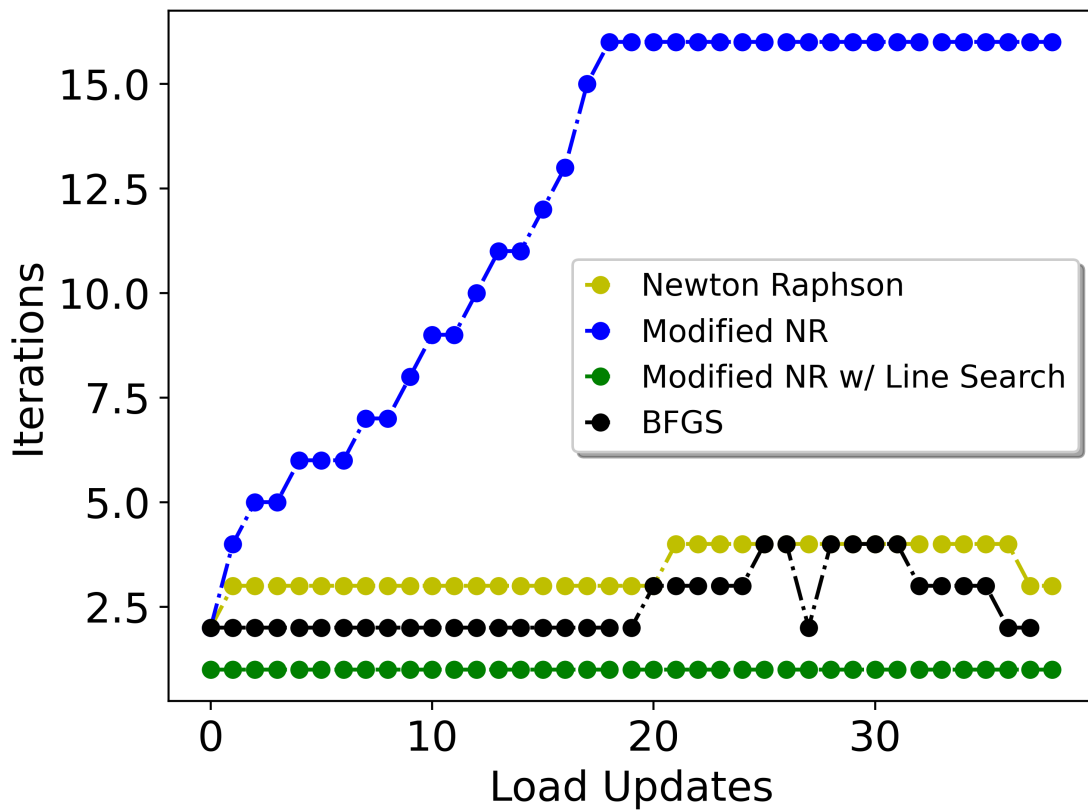
<Figure size 2500x1500 with 0 Axes>

```

[62]: plt.plot(num_iter_NR, 'yo-.', label='Newton Raphson')
plt.plot(num_iter_MNR, 'bo-.', label='Modified NR')
plt.plot(num_iter_MNRLS, 'go-.', label='Modified NR w/ Line Search')
plt.plot(num_iter_BFGS, 'ko-.', label='BFGS')
plt.xlabel("Load Updates")
plt.ylabel("Iterations")

```

```
plt.legend(shadow=True, handlelength=1, fontsize=12)
plt.rcParams['figure.dpi'] = 500
```



1.2 ChatGPT Implementation

Language Input

I have a nonlinear problem to be solved using Newton-based solvers. Let's first define the residual

ChatGPT Response

To solve the given problem in Python, we can use the Newton-Raphson method. The method involves

Define the residual function $\mathbf{G}(\mathbf{v}) = \mathbf{R}(\mathbf{v}) - \lambda \mathbf{P} = 0$ as a Py

Define the Jacobian matrix of the residual function, which is the matrix of partial derivatives

Define a function that performs one iteration of the Newton-Raphson method by solving the linear

Implement a loop that performs the Newton-Raphson iterations for each load increment λ_i

Plot the relationship between R_1 and v_1 for the converged solution at each load increment.

Plot the number of iterations taken to converge for each load increment.

Here's the Python code to solve the problem:

```
[14]: import numpy as np
import matplotlib.pyplot as plt

# Define the residual function  $G(v) = R(v) - \lambda P$ 
def R1(v, x):
    return 0.2*v[0]**3 - x*v[1]**2 + 6*v[0]

def R2(v):
    return v[1] - v[0]

def G(v, lam, x=1.8):
    return np.array([R1(v, x) - lam, R2(v)])

# Define the Jacobian matrix  $J = dG/dv$ 
def Jf(v, lam, x=1.8):
    J = np.zeros((2, 2))
    J[0, 0] = 0.6*v[0]**2 + 6
    J[0, 1] = -2*x*v[1]
    J[1, 0] = -1
    J[1, 1] = 1
    return J

# Define the load increments
lam_values = np.linspace(0, 10, 41)

# Define convergence criteria
epsilon = 1e-4
n_max = 15

# Initialize the solution and convergence arrays
num_loadsteps = len(lam_values)
v0 = np.array([0, 0])
v_converged = np.zeros((num_loadsteps, 2))
iter_counts = np.zeros(num_loadsteps)

# Solve for the nonlinear solution using Newton-Raphson method
for i in range(num_loadsteps):
    lam = lam_values[i]
    v = v0
```

```

for n in range(n_max):
    J = Jf(v, lam)
    Gv = G(v, lam)
    norm_Gv = np.linalg.norm(Gv)

    if norm_Gv <= epsilon*np.linalg.norm(G(v, 0)):
        v_converged[i] = v
        iter_counts[i] = n+1
        break

    dv = np.linalg.solve(J, -Gv)
    v += dv

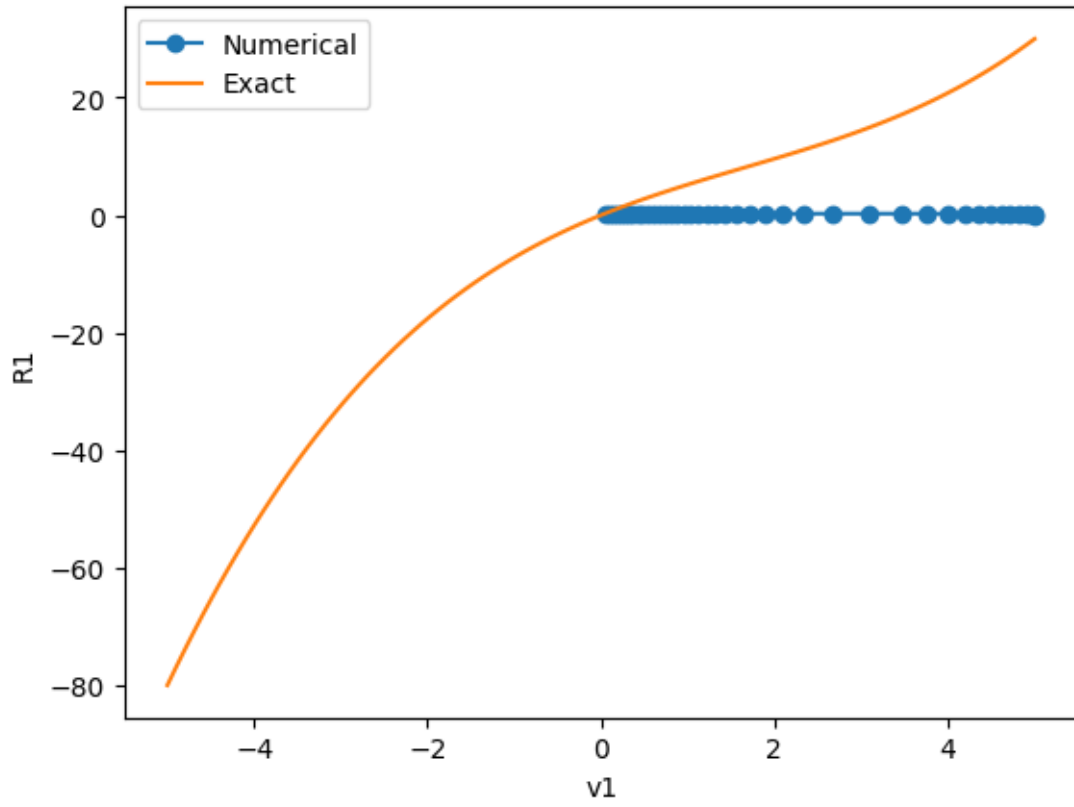
v0 = v_converged[i]

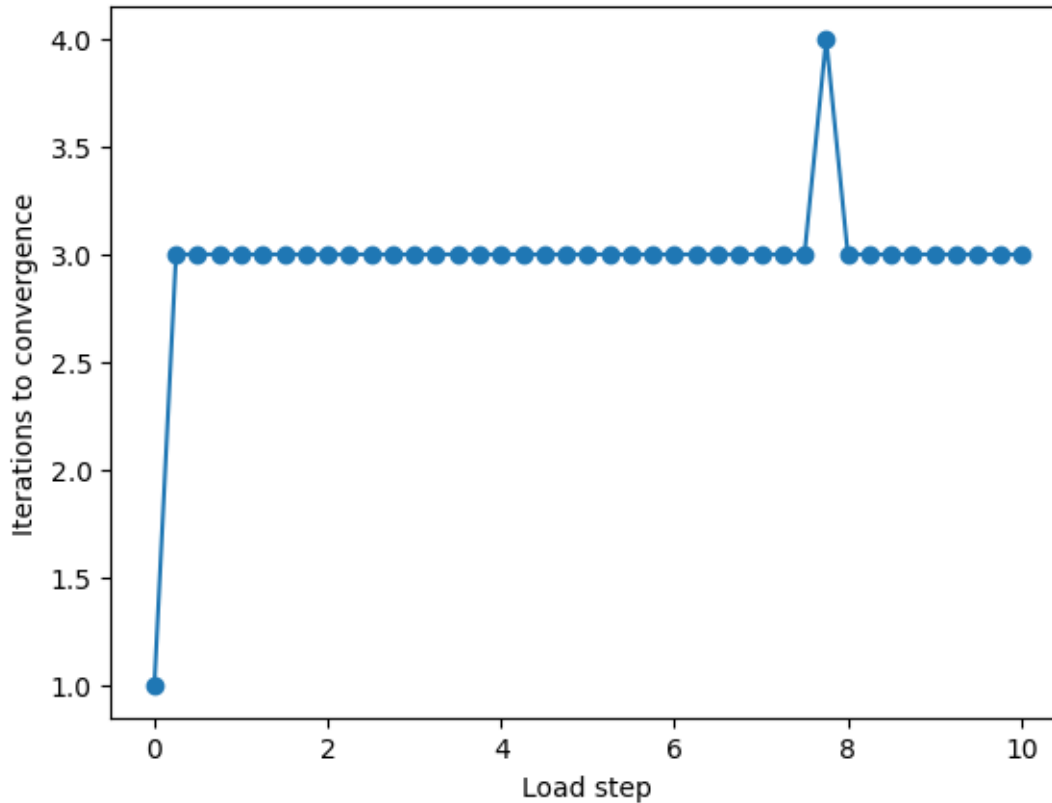
# Exact solution for R1 vs. v1
v1_exact = np.linspace(-5, 5, 100)
R1_exact = 0.2*v1_exact**3 - x*v1_exact**2 + 6*v1_exact

# Plot the nonlinear solution and exact solution
plt.plot(v_converged[:, 0], G(v_converged.T, lam_values).T[:, 0], '-o',
         label='Numerical')
plt.plot(v1_exact, R1_exact, label='Exact')
plt.xlabel('v1')
plt.ylabel('R1')
plt.legend()
plt.show()

# Plot the number of iterations to convergence vs. load step number
plt.plot(lam_values, iter_counts, '-o')
plt.xlabel('Load step')
plt.ylabel('Iterations to convergence')
plt.show()

```





Language Input

Now, solve this problem using the modified Newton-Raphson method and get the two plots.

ChatGPT Response

Sure, here's the code for the modified Newton-Raphson method to solve the given problem and plot

```
[15]: def newton_raphson_modified(G, DG, x0, lam, epsilon=1e-6, max_iter=100):
    x = x0.copy()
    n = 0
    while n < max_iter:
        n += 1
        Gx = G(x, lam)
        DGx = DG(x, lam)
        dx = np.linalg.solve(DGx, -Gx)
        x = x + dx
        if np.linalg.norm(Gx) <= epsilon:
            break
    return x, n
```



```

[16]: # Define the functions G and DG for the nonlinear equation
def G(v, lam):
    G1 = 0.2*v[0]**3 - v[1]*v[0]**2 + 6*v[0] - lam
    G2 = -0.1*v[0]**2 + 0.1*v[1]**2 - v[1] - 1
    return np.array([G1, G2])

def DG(v, lam):
    DG11 = 0.6*v[0]**2 - 2*v[1]*v[0] + 6
    DG12 = -v[0]**2 - 2*v[0]*v[1] - 1
    DG21 = -0.2*v[0]
    DG22 = 0.2*v[1] - 1
    return np.array([[DG11, DG12], [DG21, DG22]])

# Define the starting point and range of lambda values
v0 = np.array([2.0, 2.0])
lam_values = np.linspace(0, 4, 41)

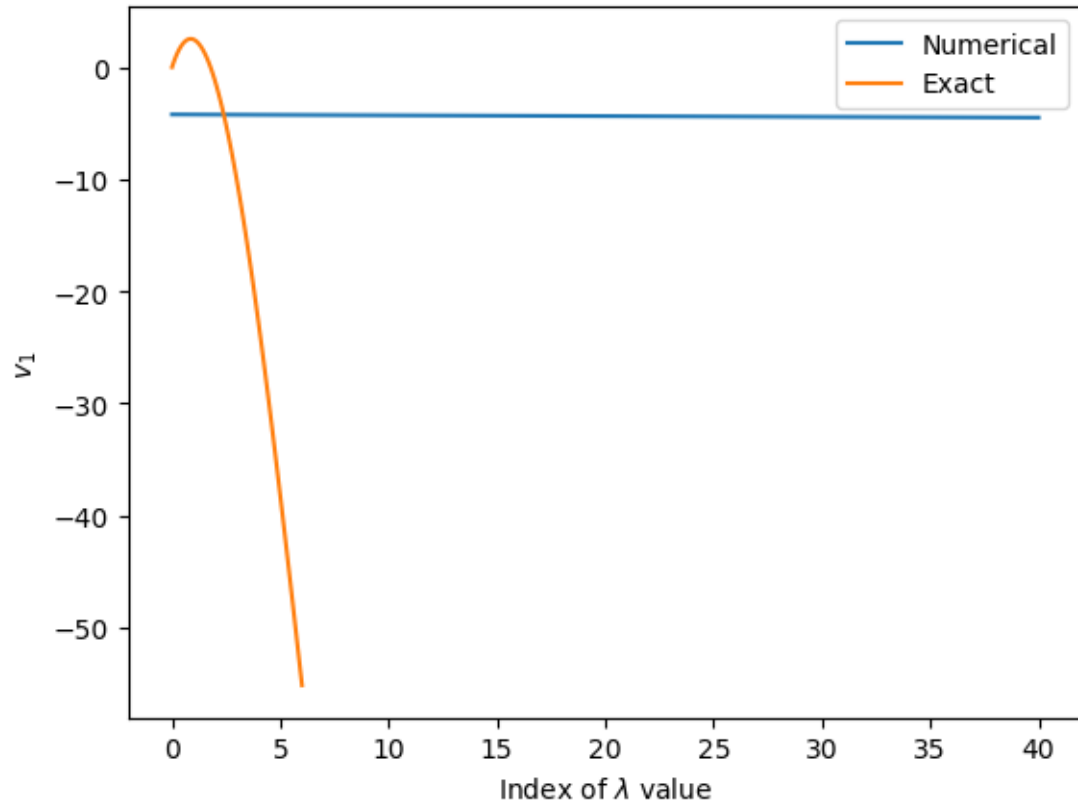
# Use the modified Newton-Raphson method to solve the nonlinear equation
v_converged = np.zeros((len(lam_values), 2))
iter_counts = np.zeros(len(lam_values), dtype=int)
for i, lam in enumerate(lam_values):
    v_converged[i], iter_counts[i] = newton_raphson_modified(G, DG, v0, lam)

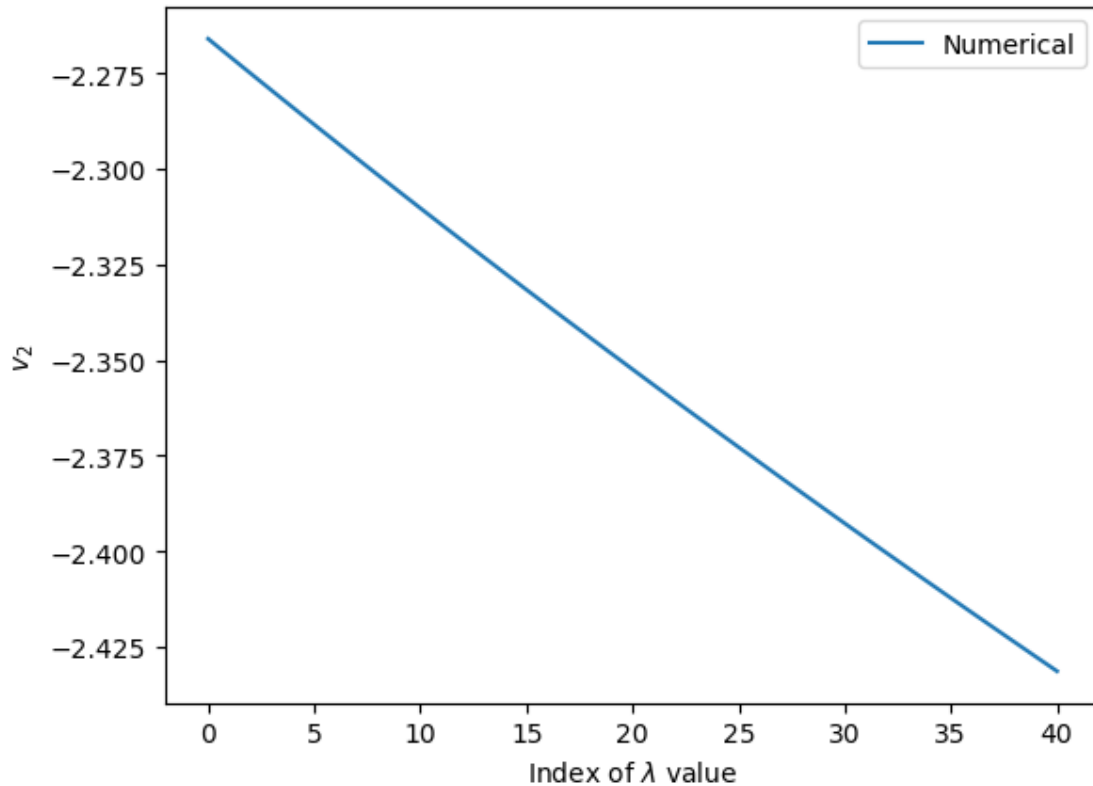
# Calculate the exact solution for v1
v1_exact = np.linspace(0, 6, 100)
R1_exact = 0.2*v1_exact**3 - v1_exact**2*(2 + np.sqrt(3)) + 6*v1_exact

# Plot the numerical and exact solutions for v1
plt.plot(v_converged[:, 0], label='Numerical')
plt.plot(v1_exact, R1_exact, label='Exact')
plt.xlabel('Index of  $\lambda$  value')
plt.ylabel('$v_1$')
plt.legend()
plt.show()

# Plot the corresponding values of v2
plt.plot(v_converged[:, 1], label='Numerical')
plt.xlabel('Index of  $\lambda$  value')
plt.ylabel('$v_2$')
plt.legend()
plt.show()

```





Input Language

Solve it using Modified Newton-Raphson method with Line Search

ChatGPT Output

Sure, here's the code to solve the same system of equations using the modified Newton-Raphson method

```
[7]: import numpy as np

def G(v, lam, x):
    n = len(v)
    G = np.zeros((n, 1))
    G[0] = 0.2*v[0]**3 - x*v[1]**2 + 6*v[0] - v[1] + lam[0]
    G[1] = v[1] - v[0] + lam[1]
    return G

def dG(v, x):
    n = len(v)
    dG = np.zeros((n, n))
    dG[0, 0] = 0.6*v[0]**2 + 6
    dG[0, 1] = -2*x*v[1]
```

```

dG[1, 0] = -1
dG[1, 1] = 1
return dG

def modified_newton_raphson(v0, lam0, x, max_iter=100, tol=1e-6):
    v = v0.copy()
    lam = lam0.copy()
    v_converged = []
    lam_converged = []
    for i in range(len(lam0)):
        Gv = G(v, lam, x)
        n_iter_i = 0
        while np.linalg.norm(Gv) > tol and n_iter_i < max_iter:
            dGv = dG(v, x)
            dv = np.linalg.solve(dGv, -Gv)
            alpha = 1
            while np.linalg.norm(G(v+alpha*dv, lam, x)) >= (1-alpha/2)*np.linalg.
→norm(Gv):
                alpha /= 2
                v += alpha*dv
                lam_new = lam + alpha*Gv
                Gv = G(v, lam_new, x)
                lam = lam_new
                n_iter_i += 1
            v_converged.append(v)
            lam_converged.append(lam)
            n_iter.append(n_iter_i)
    return v_converged, lam_converged, n_iter

# Initial guess for v and lambda
v0 = np.array([0.9, 1.2])
lam0 = np.linspace(0, 10, 41)

# Solve the problem using the modified Newton-Raphson method with line search
v_converged, lam_converged, n_iter = modified_newton_raphson(v0, lam0, x=1.8)

# Plot the relationship between R_1 and v_1
import matplotlib.pyplot as plt
v1_range = np.linspace(0, 1.5, 100)
R1_range = []
for v1 in v1_range:
    v = np.array([v1, v1])
    R1 = 0.2*v[0]**3 - 1.8*v[1]**2 + 6*v[0] - v[1]
    R1_range.append(R1)
plt.plot(v1_range, R1_range)
plt.xlabel('v_1')
plt.ylabel('R_1')

```

```
plt.show()
```

```
-----  
ValueError                                Traceback (most recent call last)  
Cell In[7], line 48  
    45 lam0 = np.linspace(0, 10, 41)  
    47 # Solve the problem using the modified Newton-Raphson method with line_  
    ↪search  
----> 48 v_converged, lam_converged, n_iter =_  
    ↪modified_newton_raphson(v0, lam0, x=1.8)  
    50 # Plot the relationship between R_1 and v_1  
    51 import matplotlib.pyplot as plt  
  
Cell In[7], line 31, in modified_newton_raphson(v0, lam0, x, max_iter, tol)  
    29 dv = np.linalg.solve(dGv, -Gv)  
    30 alpha = 1  
----> 31 while np.linalg.norm(G(v+alpha*dv, lam, x)) >= (1-alpha/2)*np.linalg.  
    ↪norm(Gv):  
    32     alpha /= 2  
    33 v += alpha*dv  
  
Cell In[7], line 6, in G(v, lam, x)  
    4 n = len(v)  
    5 G = np.zeros((n, 1))  
----> 6 G[0] = 0.2*v[0]**3 - x*v[1]**2 + 6*v[0] - v[1] + lam[0]  
    7 G[1] = v[1] - v[0] + lam[1]  
    8 return G  
  
ValueError: could not broadcast input array from shape (2,) into shape (1,)
```

```
[8]: import numpy as np  
from scipy.optimize import minimize  
  
# Define the objective function  
def obj(x):  
    return -(0.2*x[0]**3 - x[1]**2*x[0] + 6*x[0] - x[1] + x[2])  
  
# Define the gradient of the objective function  
def grad_obj(x):  
    return np.array([-0.6*x[0]**2 - x[1]**2, -2*x[0]*x[1] - 1, 1])  
  
# Define the Hessian of the objective function  
def hess_obj(x):  
    return np.array([[ -1.2*x[0], -2*x[1], 0], [-2*x[1], -2*x[0], 0], [0, 0, 0]])  
  
# Define the constraint function
```

```

def constr(x):
    return np.array([x[0] - x[1]])

# Define the Jacobian of the constraint function
def jac_constr(x):
    return np.array([[1, -1, 0]])

# Define the Hessian of the Lagrangian function
def hess_lagrangian(x, lam):
    return hess_obj(x) - lam[0]*jac_constr(x).T@jac_constr(x)

# Define the callback function
def callback(xk):
    global n_iter
    n_iter += 1

# Set the initial guess for the decision variables and Lagrange multiplier
x0 = np.array([0.5, 1.5, 0.0])
lam0 = np.array([0.0])

# Set the bounds for the decision variables
bounds = [(0.0, None), (0.0, None), (None, None)]

# Set the constraints and their bounds
cons = {'type': 'eq', 'fun': constr, 'jac': jac_constr}
bnd_cons = {'type': 'ineq', 'fun': lambda x: x[0] - 1.8}

# Solve the problem using BFGS optimization method
res = minimize(obj, x0, method='SLSQP', jac=grad_obj, hessp=hess_lagrangian,
    →constraints=[cons, bnd_cons], bounds=bounds, callback=callback)

# Print the results
print('Optimization results:')
print('  Converged:', res.success)
print('  Message:', res.message)
print('  Number of iterations:', n_iter)
print('  Optimal objective value:', -res.fun)
print('  Optimal decision variables:', res.x)
print('  Lagrange multiplier:', res.x[-1])

```

```

/lib/python3.10/site-packages/scipy/optimize/_minimize.py:561: RuntimeWarning:
Method SLSQP does not use Hessian-vector product information (hessp).
  warn('Method %s does not use Hessian-vector product '

```

TypeError

Traceback (most recent call last)

Cell In[8], line 45

```

42 bnd_cons = {'type': 'ineq', 'fun': lambda x: x[0] - 1.8}
44 # Solve the problem using BFGS optimization method
---> 45 res =
-> minimize(obj, x0, method='SLSQP', jac=grad_obj, hessp=hess_lagrangian, constraints=[cons, bn
47 # Print the results
48 print('Optimization results:')

File /lib/python3.10/site-packages/scipy/optimize/_minimize.py:708, in
-> minimize(fun, x0, args, method, jac, hess, hessp, bounds, constraints, tol,
-> callback, options)
705     res = _minimize_cobyla(fun, x0, args, constraints, callback=callback,
706                          **options)
707 elif meth == 'slsqp':
--> 708     res = _minimize_slsqp(fun, x0, args, jac, bounds,
709                          constraints, callback=callback, **options)
710 elif meth == 'trust-constr':
711     res = _minimize_trustregion_constr(fun, x0, args, jac, hess, hessp,
712                                     bounds, constraints,
713                                     callback=callback, **options)

File /lib/python3.10/site-packages/scipy/optimize/_slsqp_py.py:438, in
-> _minimize_slsqp(func, x0, args, jac, bounds, constraints, maxiter, ftol,
-> iprint, disp, eps, callback, finite_diff_rel_step, **unknown_options)
435 if majiter > majiter_prev:
436     # call callback if major iteration has incremented
437     if callback is not None:
--> 438         callback(np.copy(x))
440     # Print the status of the current iterate if iprint > 2
441     if iprint >= 2:

Cell In[8], line 31, in callback(xk)
29 def callback(xk):
30     global n_iter
---> 31     n_iter += 1

TypeError: 'int' object is not iterable

```

After so many tries, the ChatGPT really cannot give me the code for Modified Newton-Raphson w/ Line Search and BFGS, so I eventually give up. When I was implementing the code given by ChatGPT, it is usually the case that the code does not work at all in the first time and I have to ask it again and again and keep inputting the error messages, which is extremely time-consuming. It is obviously a long way to go for ChatGPT to do comprehensive computational mechanics.

MAE 7750: HW #3

Hanfeng Zhai

April 11, 2023

Pseudocode *Design a detailed pseudocode for a nonlinear hyperelasticity problem in 2D. The pseudocode should refer to which functions and subroutines will be called for each steps. Discuss what happens in each step and also mention the relevant variables. (the file points to the contents of the "hyper" folder to find other subroutines and read input files. Specific emphasis should be placed on the Newton Raphson iterative solution scheme. What are the convergence criteria?*

In this problem, I write the pseudo code mainly based on "`test_cylinder.py`". The pseudo-code is shown in Algorithm 1. The emphasis on the Newton-Raphson iteration scheme begins in Line 13.

Base BVP *Run the code and plot the deformed mesh for the St. Venant and Neo-Hookean constitutive laws.*

We first do a brief description of the mechanics problem. For the Neo-Hookean model, the strain energy density function writes (according to the codes):

$$\phi = \frac{\mu}{2} (I_1 - 3 - 2 \log J) + \frac{\lambda}{2} \log(J)^2 \quad (1)$$

where μ and λ are the first and second Lamé constants. We can further compute the second Piola-Kirchhoff stress \mathbf{S} and the elasticity tensor $\mathbf{M} \equiv \mathbb{C}$ from $\mathbf{S} = 2 \frac{\partial \phi}{\partial \mathbf{C}}$ and $\mathbf{M} = 2 \frac{\partial \mathbf{S}}{\partial \mathbf{C}}$, respectively, where $\mathbf{C} = \mathbf{F}^T \mathbf{F}$ is the right Cauchy-Green tensor.

We denote the prescribed displacement on the inner surface as u_0 . The boundary condition writes:

$$\begin{aligned} u_r = u_{\text{inner}} : u_r = u_0 = 1 \times 10^{-4} \\ u_r = u_{\text{outer}} : \sigma_r = 0 \\ \sigma_\theta = 0 \end{aligned} \quad (2)$$

Using both the coarse mesh and triangular refined mesh we generate Figure 1: the updated deformed cylinders are shown in red (current configuration \mathbf{x}) and undeformed cylinders are marked in blue (\mathbf{X}).

Alternate BVP *Change the boundary conditions and run an alternate BVP of your choice repeating the two runs of the previous task.*

Algorithm 1 Nonlinear Finite Element Analysis of Hyperelasticity

```
1: function cut(M, bcDofs, freeDofs)
2:     Cut the degree of freedom of the elasticity tensor M into the boundary and free
       degree of freedom parts.
3: function verify(U, Uref, bcDofs, tolerance)
4:     Checks if the values of U at the indices specified in bcDofs are the same as the
       corresponding values in Uref up to the specified tolerance. If they are not, it
       throws an error or warning.
5: Define the material properties: Set Young's modulus and Poisson's ratio for rubber (and
       steel — here we consider rubber). Recall the constitutive model of Neo Hookean, Gent,
       and St. Venant Kirchhoff Hyperelasticity.
6: Define the iteration parameters: Specify the number of steps in the loop, the size of each
       step, and the magnitude of the radial displacement applied to the interior boundary of
       the cylinder (I further change it to the exterior cylinder).
7: Define the solver parameters: Set the precision, tolerance, and the maximum number of
       iterations.
8: Specify the file and folders to save the output files. Create something like “/dirout/”.
9: Generate mesh based on geometry and recall gmsh python module. One can either load
       pre-generated mesh in a specific directory or generate using the existing module.
10: Set boundary conditions and force vectors.
11: Assembly the global stiffness matrix. Solve based on  $\mathbf{M} \cdot \mathbf{U} = \mathbf{F}$  (internal + external).
12: Apply the loading and update the information. Apply the iteration using the Newton-
       Raphson iteration solution scheme. Initialize the Residual, Stiffness, etc.
13: while Residual > Precision do
14:     Compute the internal stiffness matrix K from displacements.
15:     Compute the stiffness matrix M from K.
16:     Solve the unknown displacement U from M and Residual R.
17:     Compute Force F from U; Compute external force  $\mathbf{F}_{\text{ext}} = \mathbf{M}_{11}\mathbf{U}_1 + \mathbf{M}_{12}\mathbf{U}_2$ .
18:     Update Residual  $\mathbf{R} = \mathbf{F}_{\text{ext}} - \mathbf{F}_{\text{int}}$ ; Cut R into R1 & R2.
19:     Verify the boundary conditions satisfy preset tolerance.
20:     if normalized test function < precision then
21:         Update iteration.
22:     end if
23:     for Local nodes in Displacement nodes do
24:         Compute the local displacement u, Obtain the reference configuration X; → Get
           current configuration  $\mathbf{x} = \mathbf{X} + \mathbf{u}$ ; → Obtain radial strain  $\epsilon_r$ .
25:         Update nodal force and internal displacement from  $\epsilon_r$ .
26:     end for
27:     for Local element in Overall elements do
28:         Compute different kinds of Stresses (i.e., Cauchy, First Piola-Kirchhoff, ...), and
           Cauchy Green strains E.
29:     end for
30: end while
31: Plot the figures and postprocessing.
```

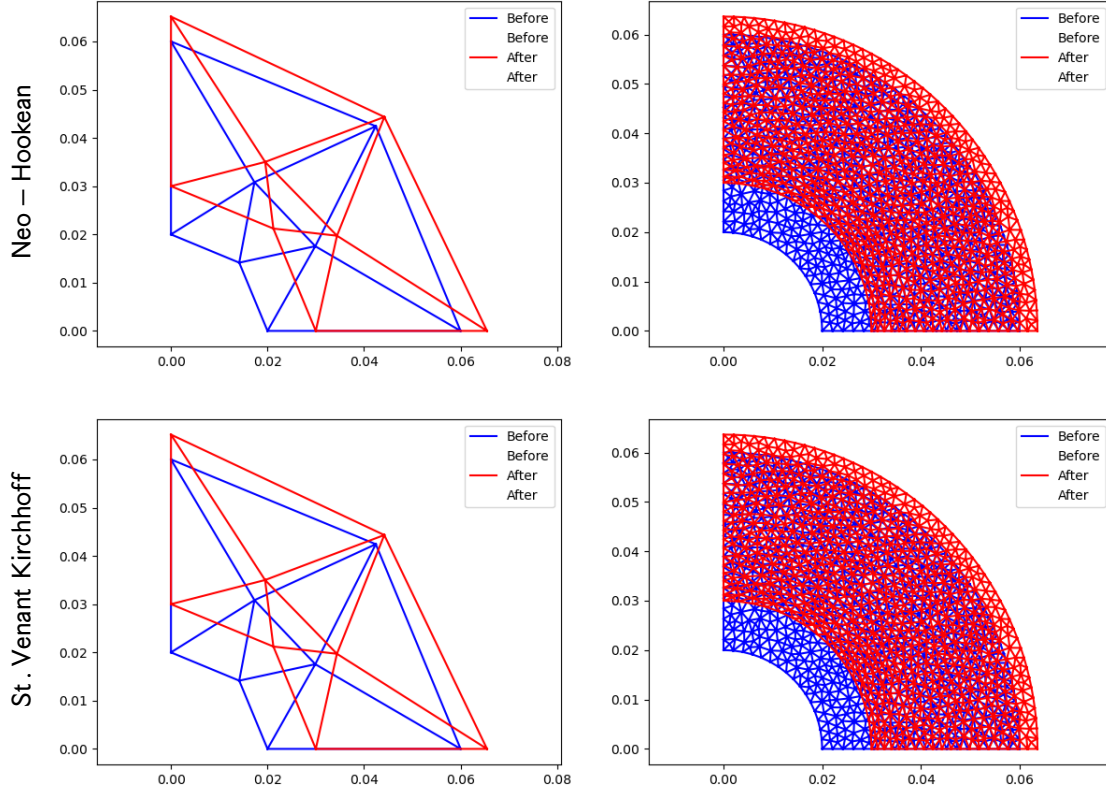


Figure 1: The original (blue) and deformed (red) mesh for the cylinder using both the coarse (left) and fine (right) triangular meshes using the Neo-Hookean (up) and St. Venant (bottom) constitutive laws for the original given boundary conditions.

I modify the new boundary conditions as one applies external displacement towards to radial center, shown in Figure 2.

The new boundary condition writes:

$$\begin{aligned}
 u_r = u_{\text{outer}} : u_r &= -u_0 = -1 \times 10^{-4} \\
 u_r = u_{\text{inner}} : \sigma_r &= 0 \\
 \sigma_\theta &= 0
 \end{aligned} \tag{3}$$

The results for using the Neo-Hookean and St. Venant Kirchhoff hyperelasticity constitutive models are shown in Figure 3. For both the previous and this problem, we adopt the rubber material as the two constitutive laws are mainly used for rubber.

Implementation of another constitutive law *Implement the necessary code for another hyperelastic constitutive law. It is suggested to work with the compressible Gent law, but any other hyperelastic law of your choice (such as Arruda-Boyce 8-chain) is also acceptable.*

Here, we use the Gent hyperelasticity constitutive model. Recalling the potential energy

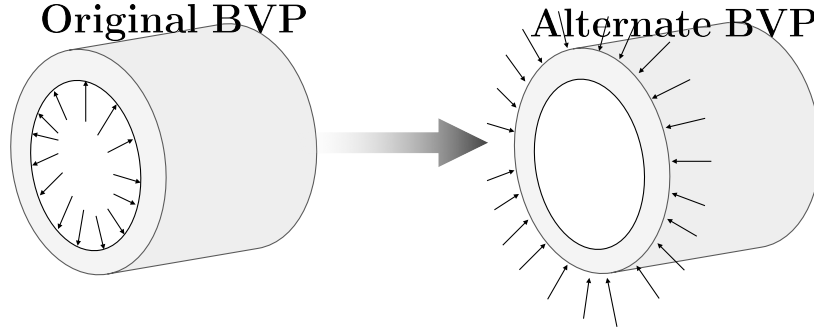


Figure 2: The schematic for my alternate boundary conditions.

model by Horgan and Saccomandi [1]:

$$\phi = -\frac{\mu}{2} \left[J_m \log \left(\left(1 + \frac{3}{J_m} \right) - \frac{\text{tr}(\mathbf{C})}{J_m} \right) + 2 \log J \right] \quad (4)$$

We can hence compute the 2nd Piola-Kirchhoff stress from the strain energy density function:

$$\begin{aligned} \mathbf{S} &= 2 \frac{\partial \phi}{\partial \mathbf{C}} \\ &= -2 \frac{\mu}{2} \left[\frac{-J_m \mathbf{I}}{J_m + 3 - \text{tr}(\mathbf{C})} + 2 \frac{1}{2} \mathbf{C}^{-1} \right] \\ &= -\mu \left[\frac{-J_m \mathbf{I}}{J_m + 3 - \text{tr}(\mathbf{C})} + \mathbf{C}^{-1} \right] \\ &= \frac{\mu J_m \mathbf{I}}{J_m + 3 - \text{tr}(\mathbf{C})} - \mu \mathbf{C}^{-1} \end{aligned} \quad (5)$$

Based on the 2nd PK stress tensor, one can compute the elasticity tensor in terms of \mathbf{C} :

$$\begin{aligned} \mathbf{M} &= 2 \frac{\partial \mathbf{S}}{\partial \mathbf{C}} \\ &= 2 \frac{\partial \left[\frac{\mu J_m \mathbf{I}}{J_m + 3 - \text{tr}(\mathbf{C})} - \mu \mathbf{C}^{-1} \right]}{\partial \mathbf{C}} \\ &= \frac{2\mu J_m \mathbf{I} \otimes \mathbf{I}}{[J_m + 3 - \text{tr}(\mathbf{C})]^2} - 2\mu \frac{\partial \mathbf{C}^{-1}}{\partial \mathbf{C}} \end{aligned} \quad (6)$$

Implement this in Python we generate the following codes:

```

1 class GentElasticity(Elasticity):
2
3     def __init__(self, E, nu):
4         super(GentElasticity, self).__init__(E, nu)
5
6     def potential(self, C):
7         lamda = self.get1LAME()
8         mu = self.get2LAME()

```

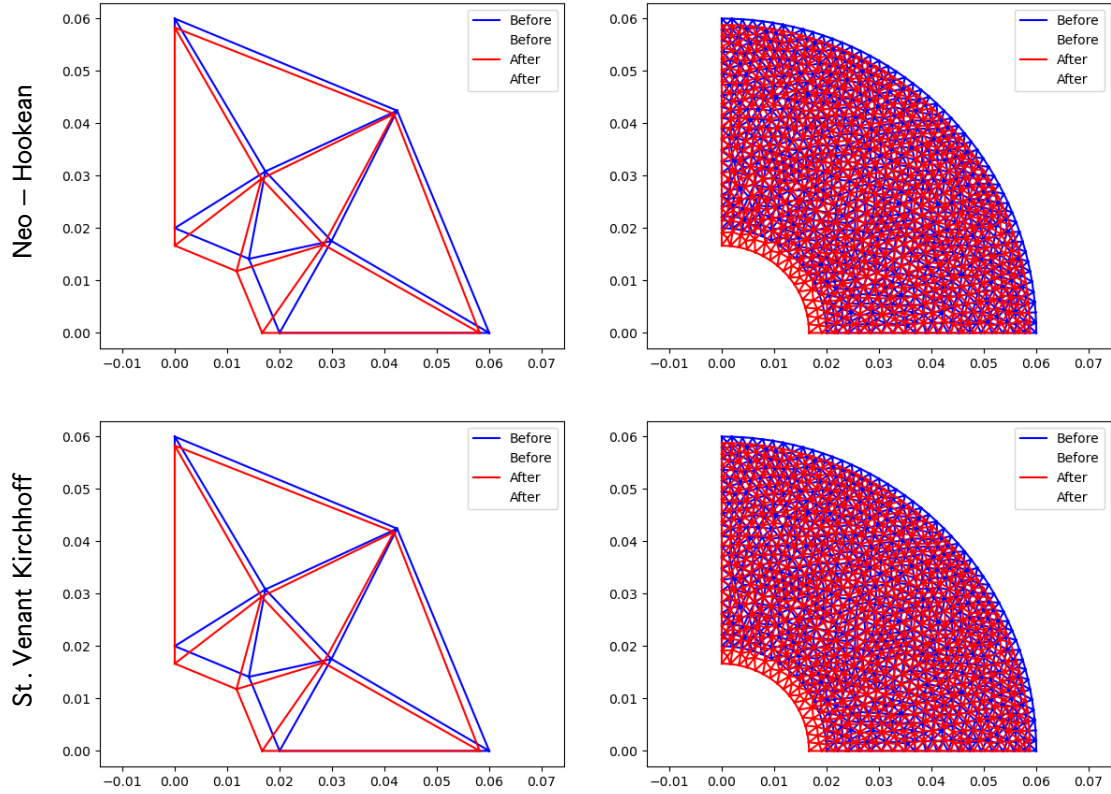


Figure 3: The original (blue) and deformed (red) mesh for the cylinder using both the coarse (left) and fine (right) triangular meshes using the Neo-Hookean (up) and St. Venant (bottom) constitutive laws for the alternate (Figure 2) boundary conditions.

```

9     dim = len(C)
10    if dim == 2:
11        K = np.copy(C)
12        C = np.zeros((3, 3))
13        C[:2, :2] = K[:, :]
14        J = np.sqrt(tensor.det(C))
15        jm = 50
16        phi = - (mu / 2) * ( jm * np.log( (1 + (3/jm)) - (tensor.trace(C)/jm) ) + 2 * np.log
17        (J))
18        return phi
19
20    def stress(self, C):
21        dim = len(C)
22        if dim == 2:
23            K = np.copy(C)
24            C = np.eye(3)
25            C[:2, :2] = K[:, :]
26            PK2 = tensor.tensor(dim)
27            lamda = self.get1LAME()
28            mu = self.get2LAME()
29            invC = tensor.inv(C)
30            I = tensor.I(3)
31            jm = 50
32            PK2 = ( ( mu * jm * I ) / ( jm + 3 - tensor.trace(C) ) ) - mu * invC

```

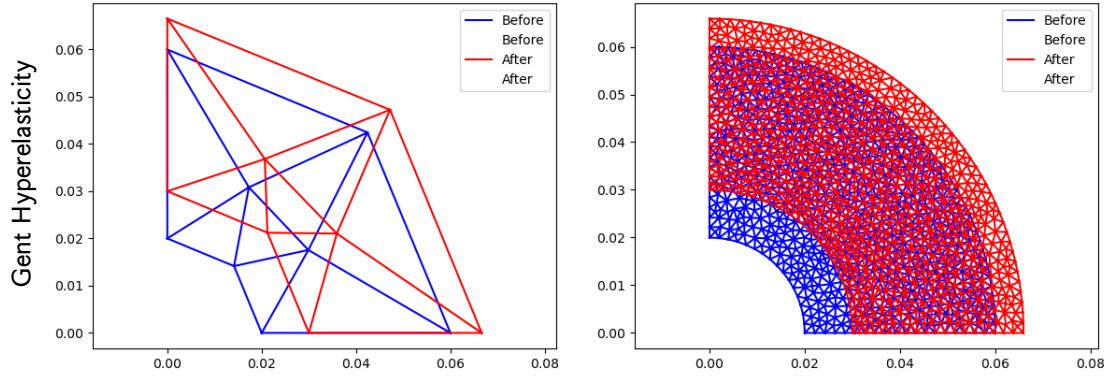


Figure 4: The original (blue) and deformed (red) mesh for the cylinder using both the coarse (left) and fine (right) triangular meshes using the Gent constitutive laws for the original given boundary conditions.

```

32     if dim == 2:
33         return PK2[:2, :2]
34     return PK2
35
36 def stiffness(self, C):
37     d = len(C)
38     dim = len(C)
39     lamda = self.get1LAME()
40     mu = self.get2LAME()
41     invC = tensor.inv(C)
42     dinvC = tensor.tensor4(d)
43     for i in range(d):
44         for j in range(d):
45             for k in range(d):
46                 for l in range(d):
47                     part1 = invC[i, k] * invC[j, l]
48                     part2 = invC[i, l] * invC[j, k]
49                     dinvC[i, j, k, l] = -(part1 + part2) / 2
50
51     I = tensor.I(dim)
52     IxI = tensor.outerProd4(I, I)
53     jm = 50
54     M = ( (2 * mu * jm * IxI) / (jm + 3 - tensor.trace(C)**2) ) - 2 * mu * dinvC
55     return M

```

References

- [1] C. O. HORGAN AND G. SACCOMANDI, *Constitutive models for compressible nonlinearly elastic materials with limiting chain extensibility*, Journal of Elasticity, 77 (2004), pp. 123–138.

MAE 7750: HW #4

Hanfeng Zhai

May 23, 2023

You will consider J_2 plasticity with linear isotropic hardening. The yield function is defined as:

$$f(\bar{\mathbf{s}}, \hat{\alpha}) = \|\bar{\mathbf{s}}\| - \sqrt{\frac{2}{3}} [Y_0 + \hat{H}] \leq 0 \quad (1)$$

and you will consider the following material properties:

- Young's modulus: 200GPa
- Poisson's ratio: 0.3
- Coefficient of linear hardening $\hat{H} = 1930\text{MPa}$
- Coefficient of linear kinematic hardening $H = 1\text{GPa}$
- Initial yield stress $Y_0 = 268\text{MPa}$

Pseudocode Design a detailed pseudocode for a small strain elastoplasticity problem in 2D. The pseudocode should refer to which functions and subroutines will be called for each step. Discuss what happens in each step and also mention the relevant variables. (the file points to the consists of the "hyper" folder to find other subroutines and read input files. Specific emphasis should be placed on the predictor-corrector solution scheme. What are the convergence criteria? Build on the pseudocode from HW3 (this should be the first step before you start coding).

Here, the main file is not majorly modified, whereas the plasticity implementations are mainly controlled by `elasticity` & `fem`, representing the definition of the constitutive models and the FEM iteration updates. The two related algorithms' pseudocode (Algorithms 2 & 3) are also attached after the pseudocodes for the main file (Algorithm 1).

In my implementation, I define two different boundary conditions. For the first boundary condition, I apply a radial displacement of 5×10^{-5} on the interior boundary. For the second boundary condition, I apply a radial displacement of -1×10^{-4} on the exterior boundary.

The convergence is majorly governed by the combination of three factors: **tolerance**, machine precision, and maximum iterations. Also, the incremental loading loop steps and the general meshing structures, the geometry also affect the convergence. In our implementation,

Algorithm 1 Nonlinear Finite Element Analysis for Elastoplasticity

```
1: function cut(M, bcDofs, freeDofs)
2:   Cut the degree of freedom of the elasticity tensor M into the boundary and free
   degree of freedom parts.
3: function verify(U, Uref, bcDofs, tolerance)
4:   Checks if the values of U at the indices specified in bcDofs are the same as the
   corresponding values in Uref up to the specified tolerance. If they are not, it
   throws an error or warning.
5: Define the material properties: Set material parameters for steel:  $E, \nu, \hat{H}, H, Y_0$ . Recall
   the constitutive model of Elastoplasticity (both the Neo-Hookean & St. Venant models)
6: Define the iteration parameters: Specify the number of steps in the loop, the size of each
   step, and the magnitude of the radial displacement applied to the interior boundary of
   the cylinder (I further change it to the exterior cylinder).
7: Define the solver parameters: Set the precision, tolerance, and the maximum number of
   iterations.
8: Specify the file and folders to save the output files. Create something like “/dirout/”.
9: Generate mesh based on geometry and recall gmsh python module. One can either load
   pre-generated mesh in a specific directory or generate using the existing module.
10: Set boundary conditions and force vectors.
11: Assembly the global stiffness matrix. Solve based on  $\mathbf{M} \cdot \mathbf{U} = \mathbf{F}$  (internal + external).
12: Apply the loading and update the information. Apply the iteration using the Newton-
   Raphson iteration solution scheme. Initialize the Residual, Stiffness, etc.
13: while Residual > Precision do
14:   Compute the internal stiffness matrix K from displacements.
15:   Compute the stiffness matrix M from K.
16:   Solve the unknown displacement U from M and Residual R.
17:   Compute Force F from U; Compute external force  $\mathbf{F}_{\text{ext}} = \mathbf{M}_{11} \mathbf{U}_1 + \mathbf{M}_{12} \mathbf{U}_2$ .
18:   Update Residual  $\mathbf{R} = \mathbf{F}_{\text{ext}} - \mathbf{F}_{\text{int}}$ ; Cut R into R1 & R2.
19:   Verify the boundary conditions satisfy preset tolerance.
20:   if normalized test function < precision then
21:     Update iteration.
22:   end if
23:   for Local nodes in Displacement nodes do
24:     Compute the local displacement u, Obtain the reference configuration X; → Get
     current configuration  $\mathbf{x} = \mathbf{X} + \mathbf{u}$ ; → Obtain radial strain  $\epsilon_r$ .
25:     Update nodal force and internal displacement from  $\epsilon_r$ .
26:   end for
27:   for Local element in Overall elements do
28:     Compute different kinds of Stresses (i.e., Cauchy, First Piola-Kirchhoff, ...), and
     Cauchy Green strains E.
29:   end for
30: end while
31: Plot the figures and postprocessing.
```

the tolerance is 1×10^{-7} , the machine precision is 1×10^{-15} , and the maximum iteration is 1000. I discretized my applied radial displacements into 10 loading steps.

Algorithm 2 Constitutive Model Definition

- 1: **class** Plasticity(E, ν, H, \hat{H}, Y_0)
 - 2: Initialize and obtain the corresponding materials' properties, i.e., $G, K, \lambda, \mu, \dots$
 - 3: Obtain the stress and stiffness matrices from the corresponding constitutive laws.
 - 4: Define and obtain the properties related to both elasticity & plasticity.
 - 5: **class** Neo-Hookean Elasto-Plasticity(Plasticity)
 - 6: Obtain the potential, stress, & stiffness from the hyperelasticity theories.
 - 7: Define the yield check and compute the plastic stress based on the passed plasticity parameters. Then compute the materials tangent from the plastic stress.
 - 8: **class** St-Venant Elasto-Plasticity(Plasticity)
 - 9: Obtain the potential, stress, & stiffness from the hyperelasticity theories.
 - 10: Define the yield check and compute the plastic stress based on the passed plasticity parameters. Then compute the materials tangent from the plastic stress.
-

Algorithm 3 Finite Element Iteration Update

- 1: **class** FiniteElement
 - 2: Initialize all the related variables to be calculated, i.e., $\mathbf{F}, \mathbf{E}, \sigma, \mathbf{S}, \mathbf{K}, \dots$
 - 3: Define operators to calculate shapes, obtain dimensions, & the integration points, ...
 - 4: Define the *update* operator:
 - 5: Looping over the integration points: update the related stresses and strains required in the calculations.
 - 6: Check whether the materials reach plasticity:
 - 7: First calculate the yield criteria f — if the material yields, update the new plastic stresses and strains.
 - 8: Compute forces, stiffness, etc.
-

Base BVP *Run the code and plot the deformed mesh for the St. Venant and Neo-Hookean constitutive laws. Plot the evolution of the equivalent plastic strain at select locations. Adjust the loading accordingly to make sure plastic loading occurs while staying within the small deformation limit.*

The deformed mesh (red) and the original mesh (blue) comparison for the Neo-Hookean model is shown in Figure 1. The cumulative plastic strains for the ten different loading steps are shown in Figure 2. The deformed mesh (red) and the original mesh (blue) comparison for the St. Venant model is shown in Figure 3. The cumulative plastic strains for the ten different loading steps are shown in Figure 4.

Alternate BVP *Change the boundary conditions and run an alternate BVP of your choice (consistent with HW3) making sure the response goes in the plastic regime.*

The deformed mesh (red) and the original mesh (blue) comparison for the Neo-Hookean

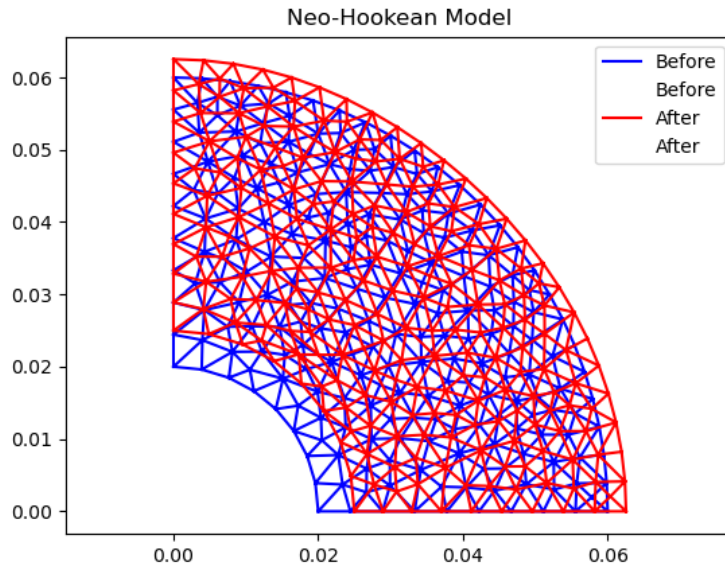


Figure 1: The original and deformed meshes for the Neo-Hookean elastoplasticity model under the first defined boundary conditions.

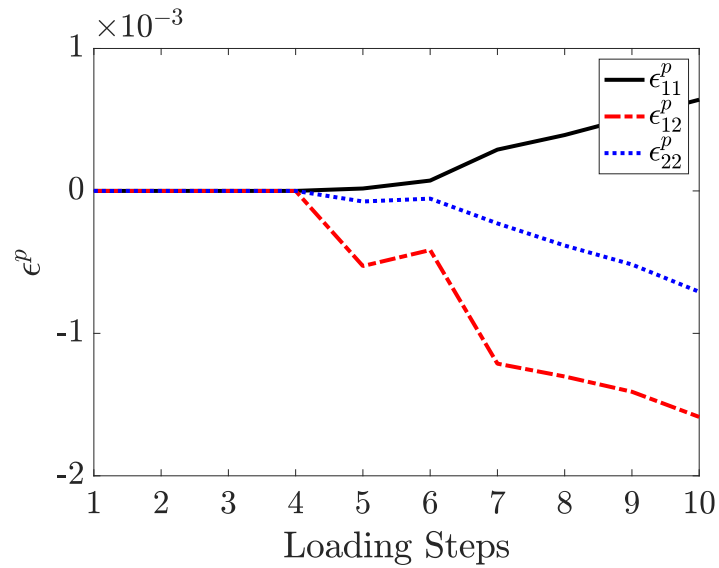


Figure 2: The evolution of the plastic strains (ϵ_{11} , ϵ_{12} , ϵ_{22}) for the Neo-Hookean elastoplasticity model under the first defined boundary conditions.

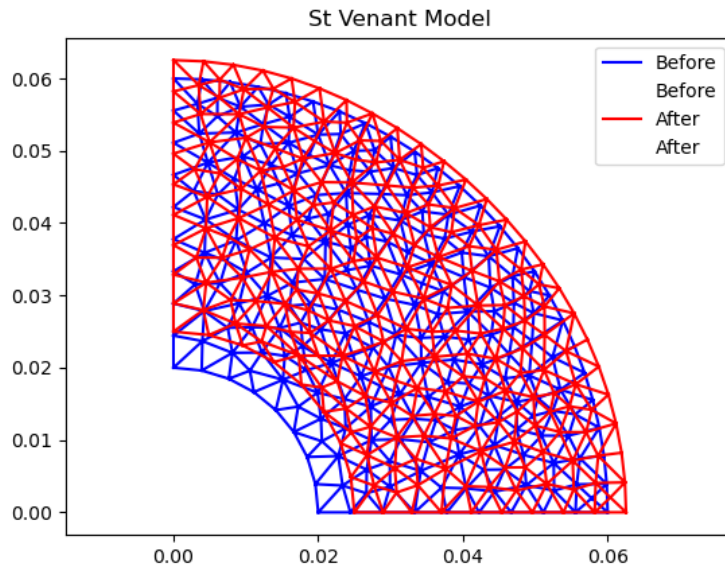


Figure 3: The original and deformed meshes for the St. Venant elastoplasticity model under the first defined boundary conditions.

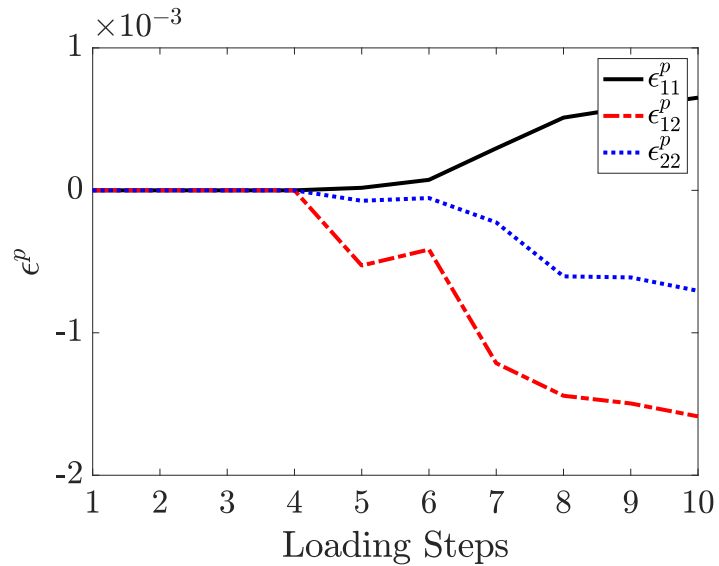


Figure 4: The evolution of the plastic strains (ϵ_{11} , ϵ_{12} , ϵ_{22}) for the St. Venant elastoplasticity model under the first defined boundary conditions.

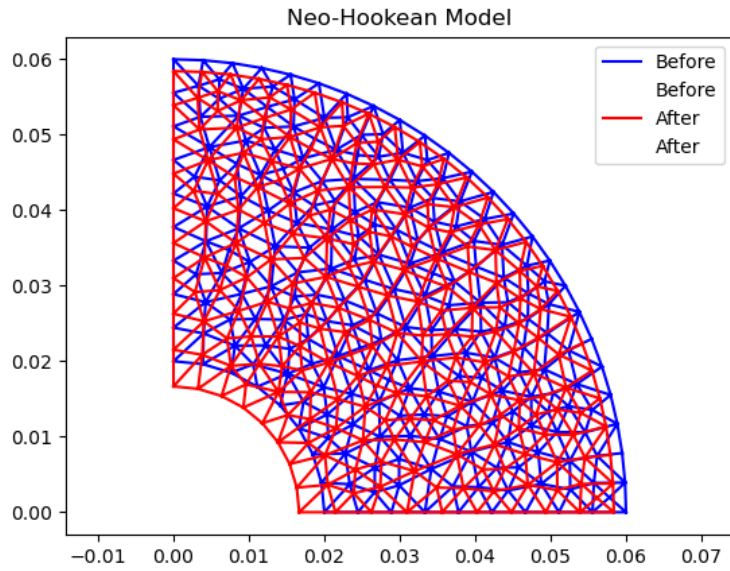


Figure 5: The original and deformed meshes for the Neo-Hookean elastoplasticity model under the second defined boundary conditions.

model is shown in Figure 5. The cumulative plastic strains for the ten different loading steps are shown in Figure 6. The deformed mesh (red) and the original mesh (blue) comparison for the St. Venant model is shown in Figure 7. The cumulative plastic strains for the ten different loading steps are shown in Figure 8.

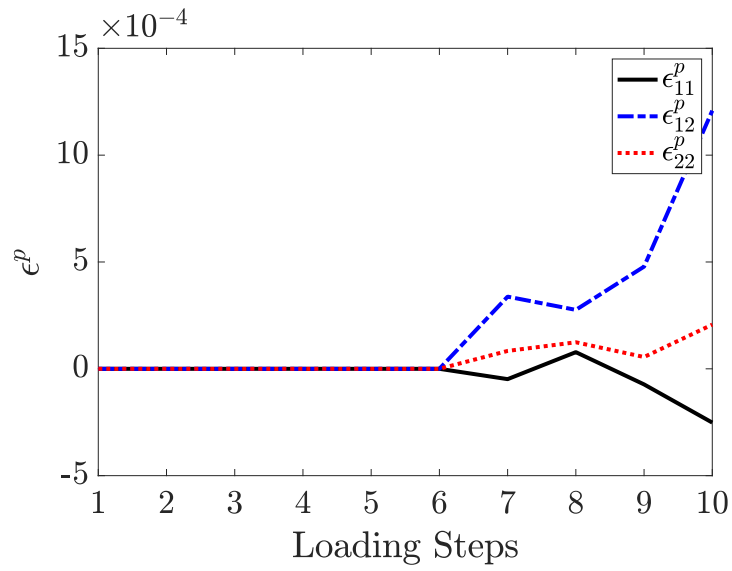


Figure 6: The evolution of the plastic strains (ϵ_{11} , ϵ_{12} , ϵ_{22}) for the Neo-Hookean elastoplasticity model under the second defined boundary conditions.

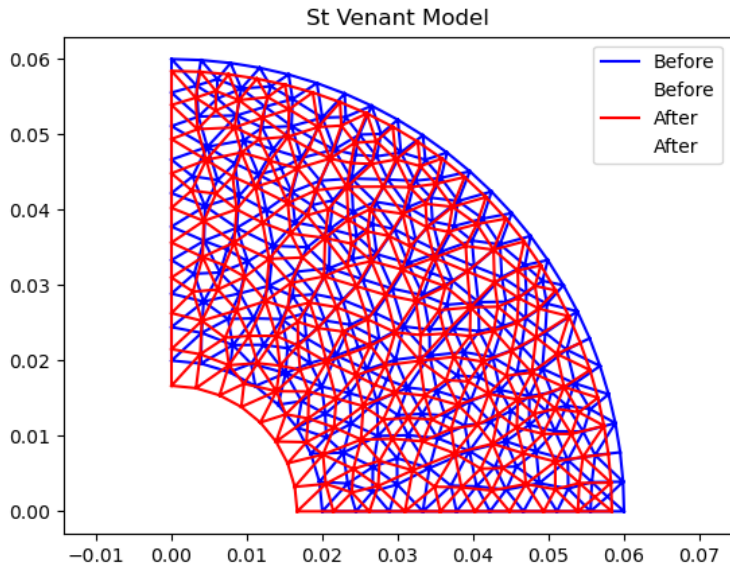


Figure 7: The original and deformed meshes for the St. Venant elastoplasticity model under the second defined boundary conditions.

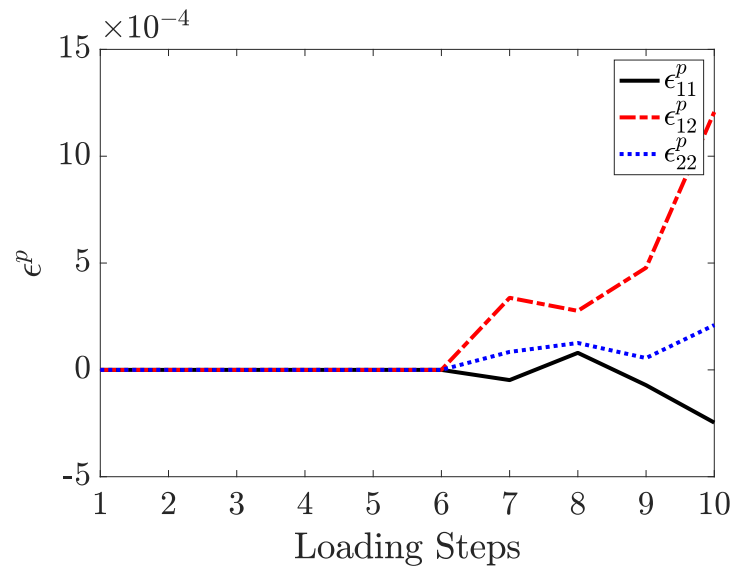


Figure 8: The evolution of the plastic strains (ϵ_{11} , ϵ_{12} , ϵ_{22}) for the St. Venant elastoplasticity model under the second defined boundary conditions.