

# PERSONAL NOTES

## FINITE ELEMENT ANALYSIS

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](mailto:hz253@cornell.edu) so that appropriate action can be taken. Your feedback is greatly appreciated.

2023

# 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  $\mathbf{B}$ <sup>1</sup>:

---

<sup>1</sup>Ref.: [https://en.wikipedia.org/wiki/Tensor\\_derivative\\_\(continuum\\_mechanics\)](https://en.wikipedia.org/wiki/Tensor_derivative_(continuum_mechanics))

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

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:

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

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  $\mathbf{D}$ :

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

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  $\alpha, \beta$  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  $\lambda_i$  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:

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

- 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, 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, 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  $\alpha, \beta$  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  $\phi$  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

$$\begin{aligned} \oint_{\mathcal{C}} \phi d\mathbf{x} &= \int_{\mathcal{S}} \mathbf{n} \times \text{grad} \phi ds \\ \oint_{\mathcal{C}} \mathbf{u} \times d\mathbf{x} &= \int_{\mathcal{S}} [(\text{div} \mathbf{u}) \mathbf{n} - (\text{grad}^\top \mathbf{u}) \mathbf{n}] ds \end{aligned}$$

For the first equation, recall Stoke's theorem, considering the definition of a circulation per unit area<sup>2</sup>, we have:

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

we can then write:

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

---

<sup>2</sup>Ref.: <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  $\nabla$  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,  $\Psi$  of an isotropic hyperelastic material is an invariant, we may regard it a function of the principal stretches  $\lambda_a, a = 1, 2, 3$  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:  $\sigma_a$ ), 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  $\Psi$  by<sup>3</sup>:

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

One can then get the Cauchy stress<sup>4</sup>:

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

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

Based on  $\boldsymbol{\sigma}$ , one can also compute the second Piola-Kirchhoff stress:

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

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

- Formulate the strain energy function  $\Psi$  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

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

---

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

<sup>4</sup>Ref.: [https://pkel015.connect.amazon.auckland.ac.nz/SolidMechanicsBooks/Part\\_III/Chapter\\_3\\_Stress\\_Mass\\_Momentum/Stress\\_Balance\\_Principles\\_05\\_Stress\\_Measures\\_NonLinear.pdf](https://pkel015.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 materials<sup>5</sup>, 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  $\lambda_1$  and  $\lambda_2$  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.

---

<sup>5</sup>Ref.: [https://en.wikipedia.org/wiki/Hyperelastic\\_material](https://en.wikipedia.org/wiki/Hyperelastic_material)

Applying Ogden's strain energy function<sup>6</sup>, 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:

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

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

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

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:

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

---

<sup>6</sup>Ref.: [http://solidmechanics.org/text/Chapter3\\_5/Chapter3\\_5.htm](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], #red
        -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], u
→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)

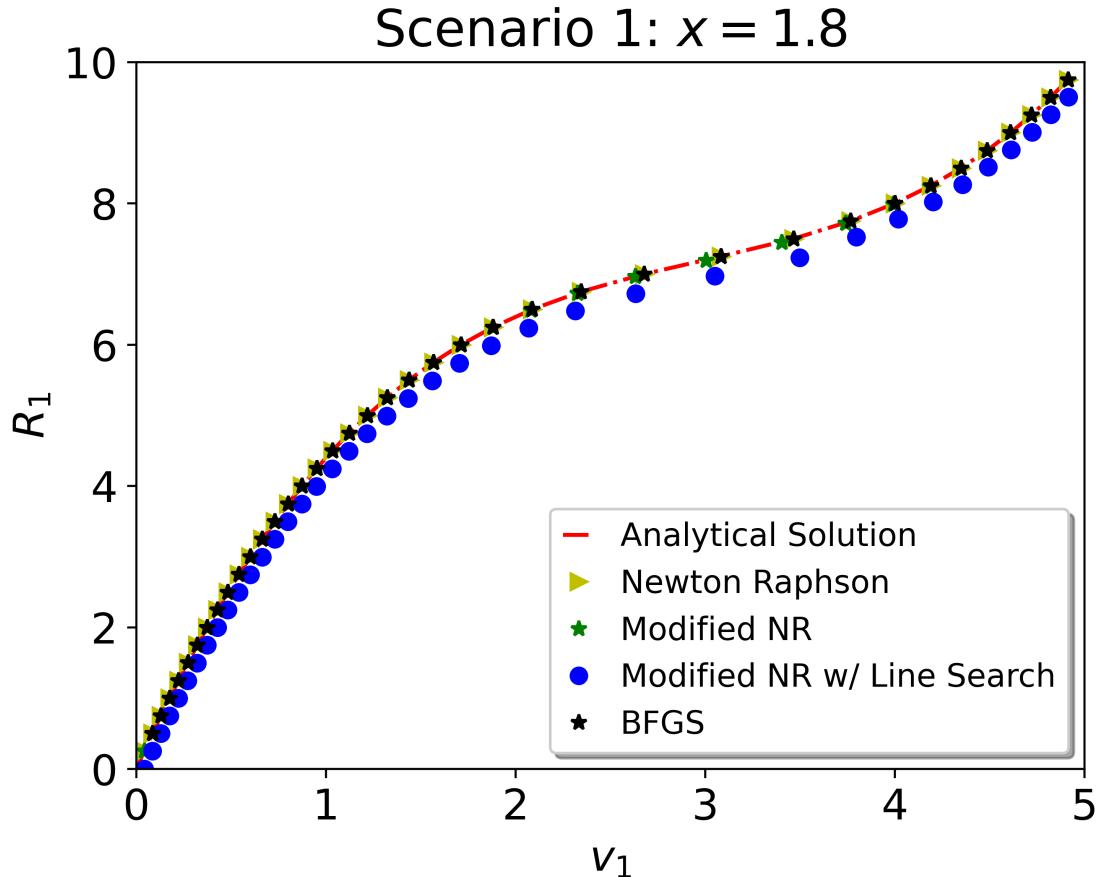
R_analy = 0.2*v_analy_solns**3 - 1.8*v_analy_solns**2 + 6*v_analy_solns

```

```

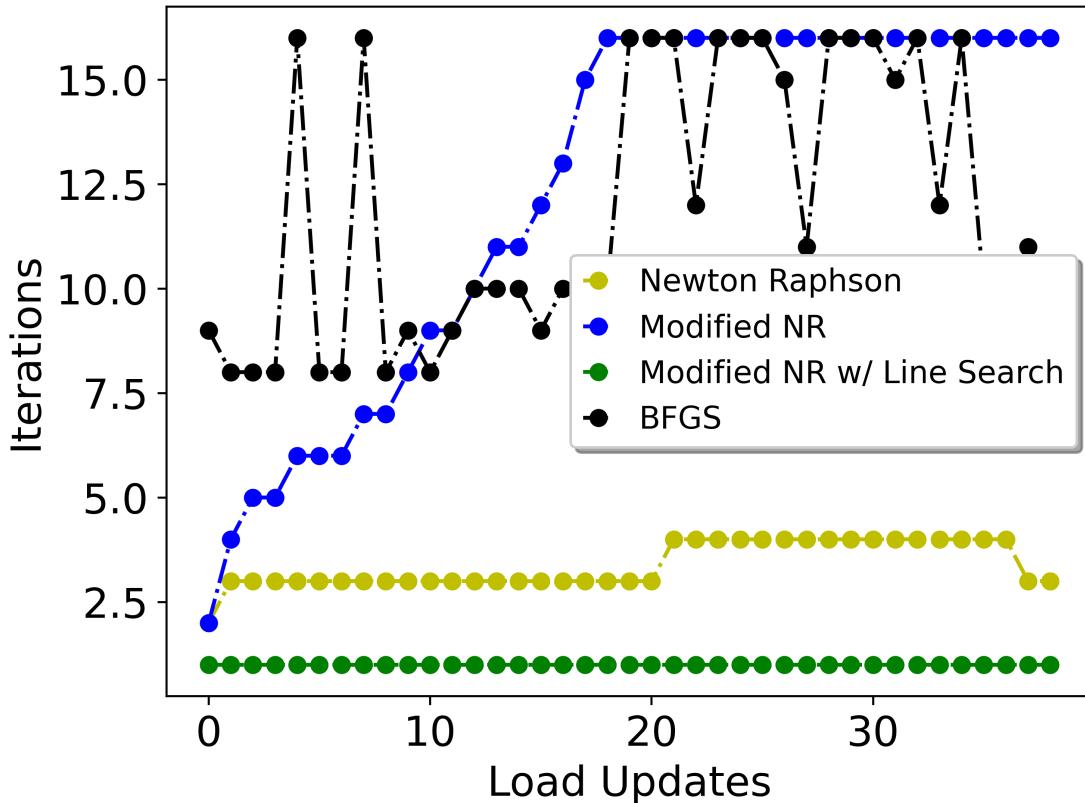
[53]: import matplotlib as mpl
plt.plot(v_analy_solns,R_analy,'r-.',label='Analytical Solution')
plt.plot(v1_loop_NR,R1_loop_NR,'y>',label='Newton Raphson')
plt.plot(v1_loop_MNR,R1_loop_MNR,'g*',label='Modified NR')
plt.plot(v1_loop_MNRLS,R1_loop_MNRLS,'bo',label='Modified NR w/ Line Search')
plt.plot(v1_loop_BFGS,R1_loop_BFGS,'k*',label='BFGS')
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, u
→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, 0.5000000000000000, 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])
```

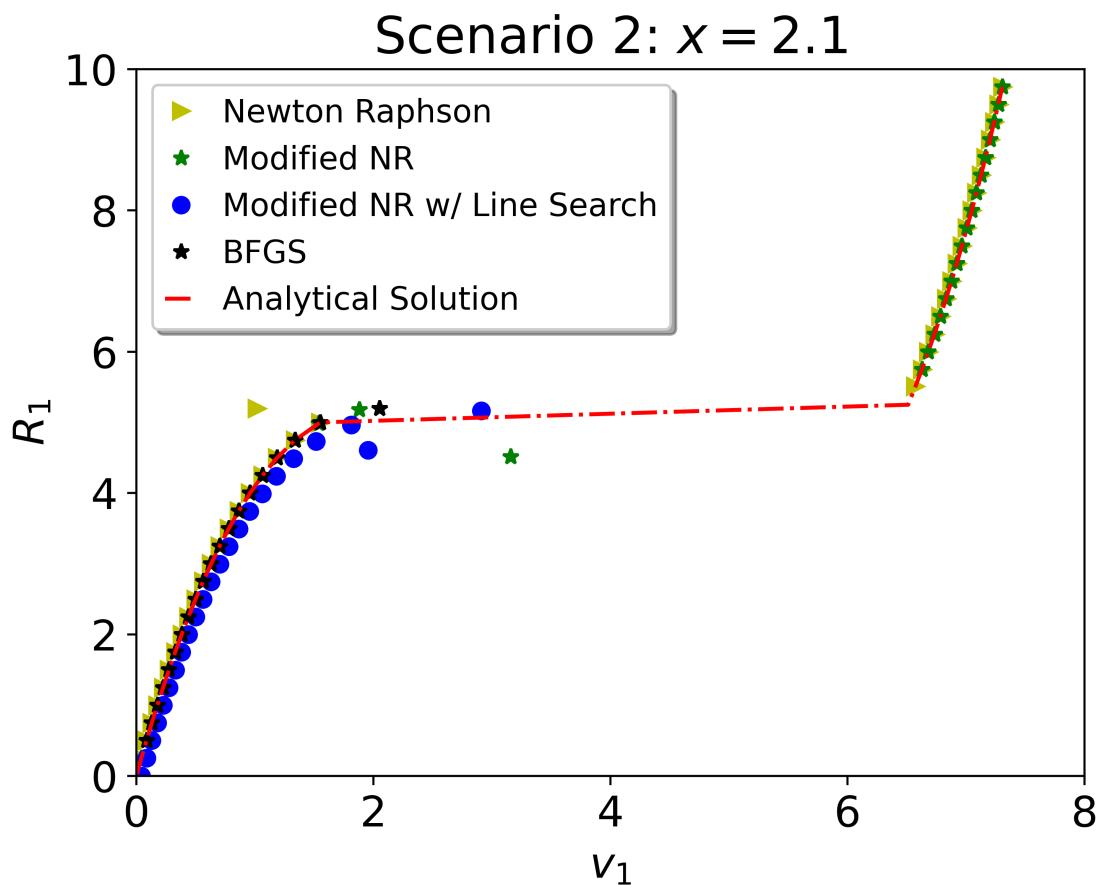
R\_analy = 0.2\*v\_analy\*\*3 - x\*v\_analy\*\*2 + 6\*v\_analy

```
[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>

```

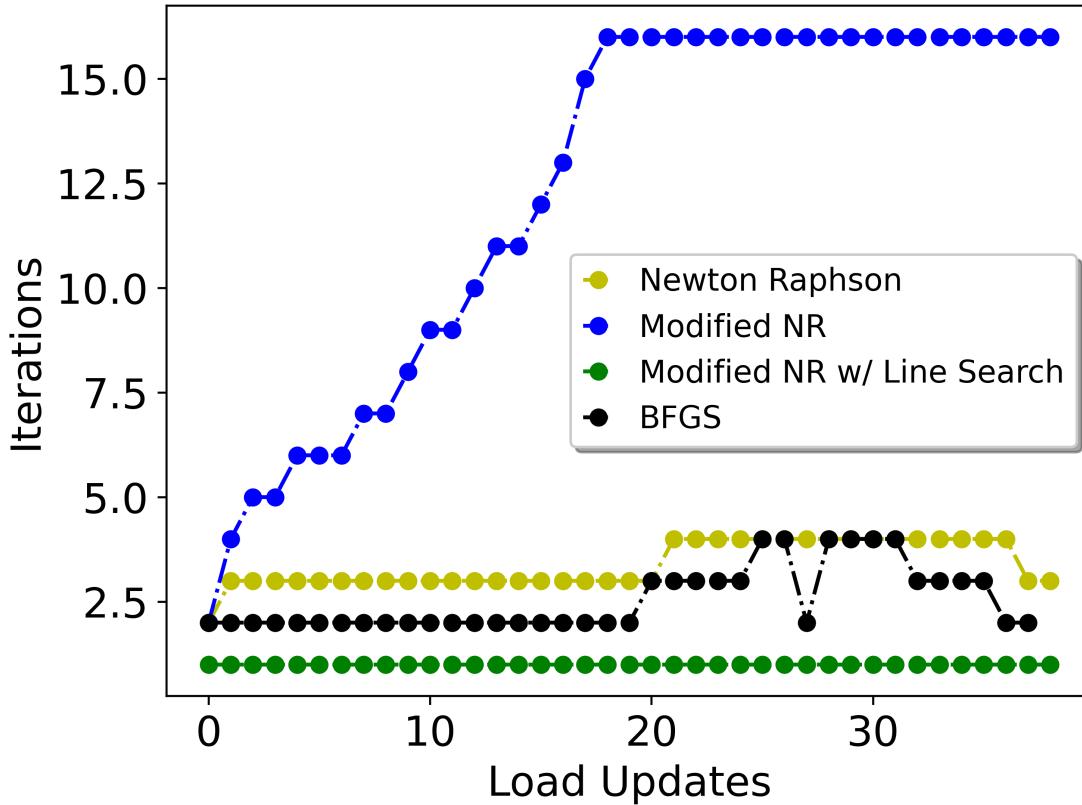
[62]: plt.plot(num_iter_NR,'yo-.',label='Newton Raphson')
plt.plot(num_iter_MNR,'bo-.',label='Modified NR')
plt.plot(num_iter_MNRLS,'go-.',label='Modified NR w/ Line Search')
plt.plot(num_iter_BFGS,'ko-.',label='BFGS')
plt.xlabel("Load Updates")
plt.ylabel("Iterations")

```

```

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 function:

### ChatGPT Response

To solve the given problem in Python, we can use the Newton-Raphson method. The method involves defining the residual function  $\mathbf{G}(\mathbf{v}) = \mathbf{R}(\mathbf{v}) - \lambda \mathbf{P} = 0$  as a Python function.

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 system:

Implement a loop that performs the Newton-Raphson iterations for each load increment  $\lambda_i$ :

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]

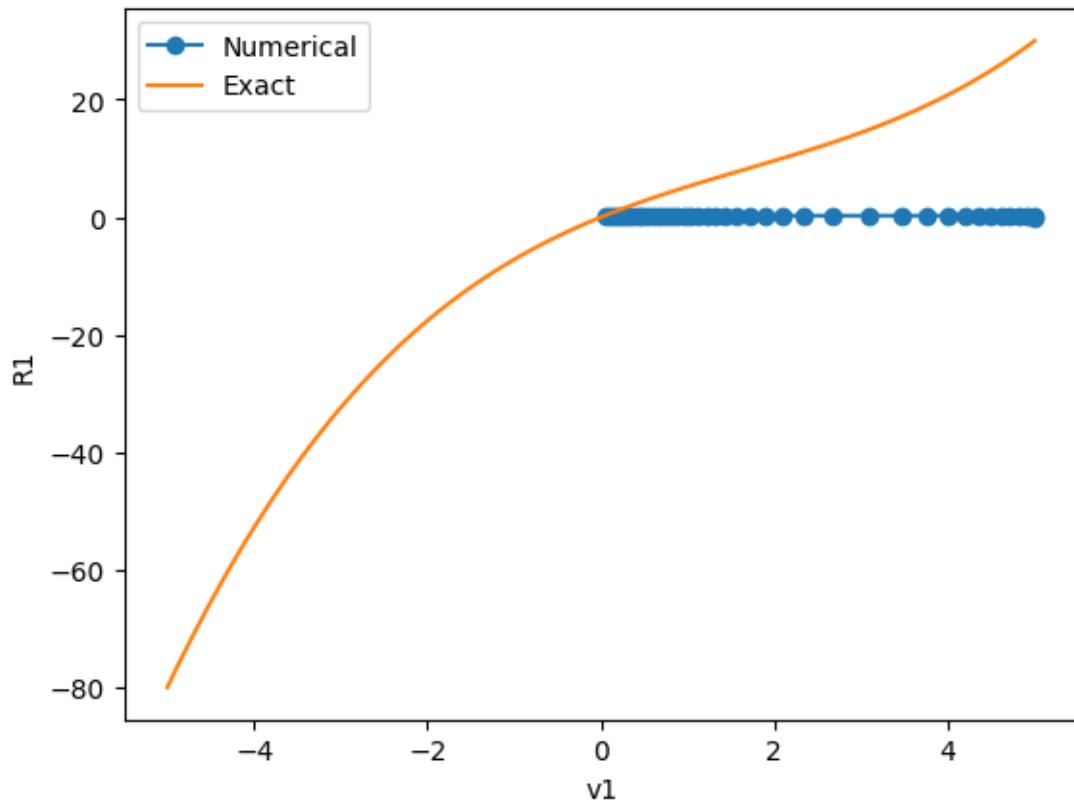
# Exact solution for R1 vs. v1
v1_exact = np.linspace(-5, 5, 100)
R1_exact = 0.2*v1_exact**3 - x*v1_exact**2 + 6*v1_exact

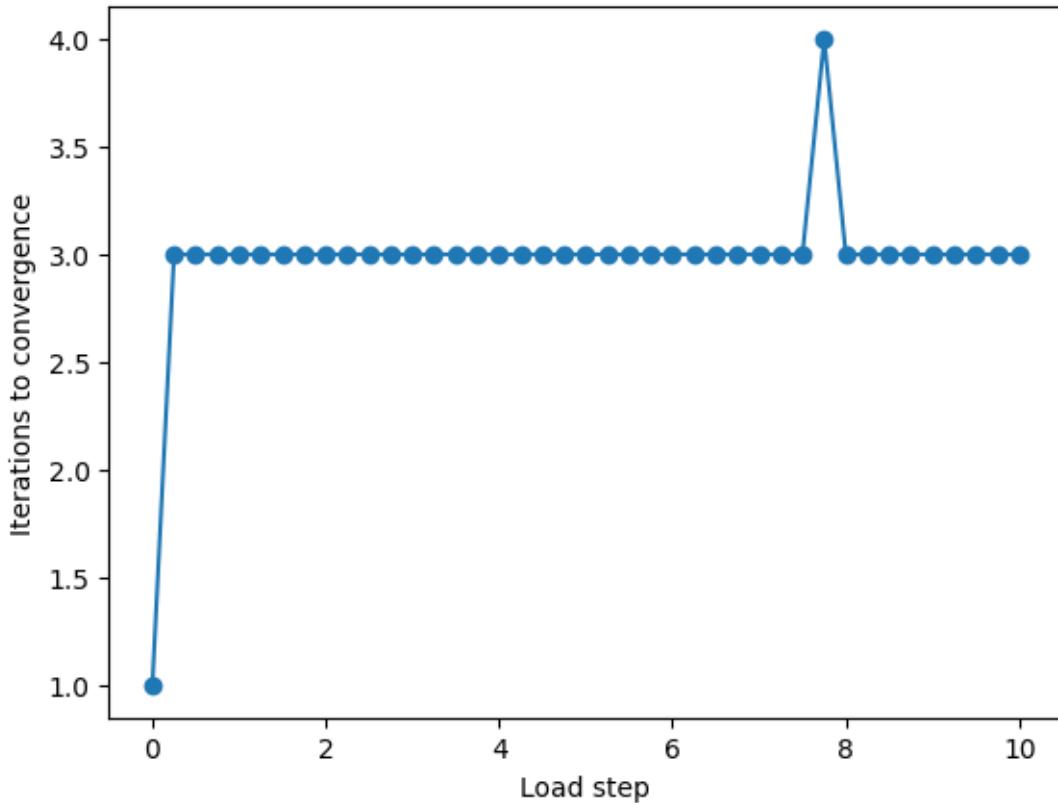
# Plot the nonlinear solution and exact solution
plt.plot(v_converged[:, 0], G(v_converged.T, lam_values).T[:, 0], '-o',  

         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]])

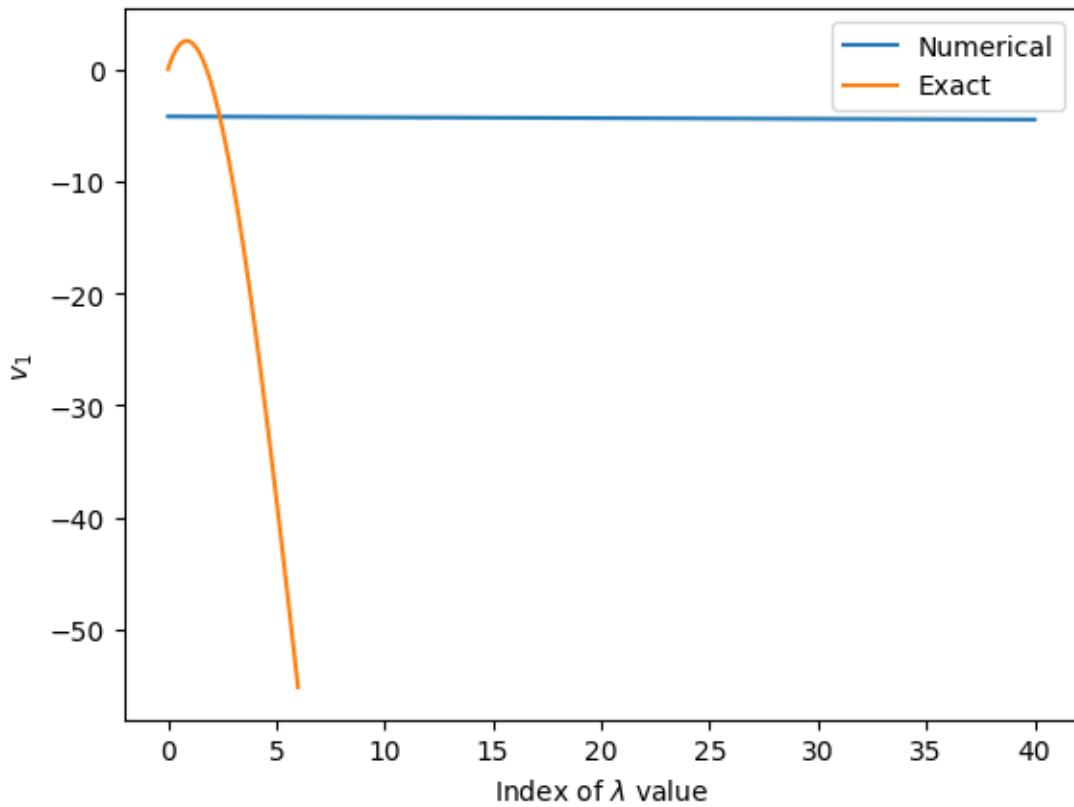
# Define the starting point and range of lambda values
v0 = np.array([2.0, 2.0])
lam_values = np.linspace(0, 4, 41)

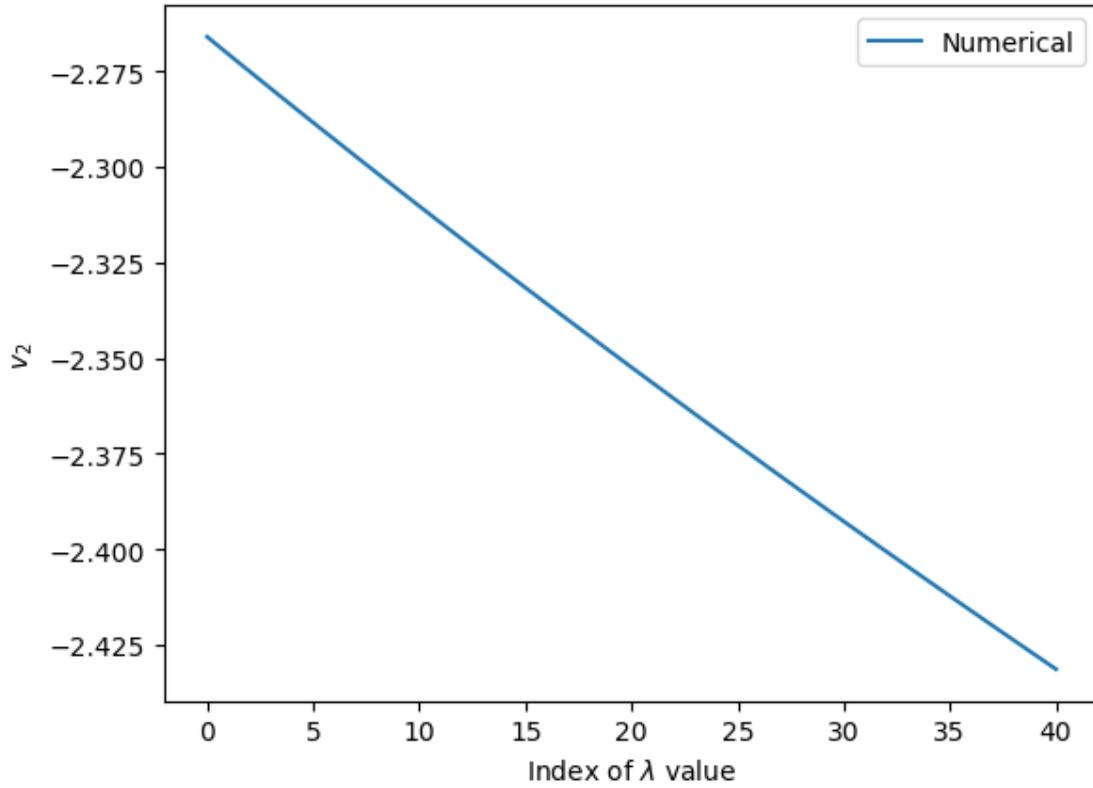
# Use the modified Newton-Raphson method to solve the nonlinear equation
v_converged = np.zeros((len(lam_values), 2))
iter_counts = np.zeros(len(lam_values), dtype=int)
for i, lam in enumerate(lam_values):
    v_converged[i], iter_counts[i] = newton_raphson_modified(G, DG, v0, lam)

# 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 me

```
[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, hess=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

Cell In[8], line 45

Traceback (most recent call last)

```

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, bnd_cons])
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  $\mu$  and  $\lambda$  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*( $\mathbf{M}$ , bcDofs, freeDofs)  
2:     Cut the degree of freedom of the elasticity tensor  $\mathbf{M}$  into the boundary and free degree of freedom parts.
- 3: **function** *verify*( $\mathbf{U}$ ,  $\mathbf{U}_{ref}$ , bcDofs, tolerance)  
4:     Checks if the values of  $\mathbf{U}$  at the indices specified in bcDofs are the same as the corresponding values in  $\mathbf{U}_{ref}$  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  $\mathbf{K}$  from displacements.
- 15:     Compute the stiffness matrix  $\mathbf{M}$  from  $\mathbf{K}$ .
- 16:     Solve the unknown displacement  $\mathbf{U}$  from  $\mathbf{M}$  and Residual  $\mathbf{R}$ .
- 17:     Compute Force  $\mathbf{F}$  from  $\mathbf{U}$ ; Compute external force  $\mathbf{F}_{ext} = \mathbf{M}_{11}\mathbf{U}_1 + \mathbf{M}_{12}\mathbf{U}_2$ .
- 18:     Update Residual  $\mathbf{R} = \mathbf{F}_{ext} - \mathbf{F}_{int}$ ; Cut  $\mathbf{R}$  into  $\mathbf{R}_1$  &  $\mathbf{R}_2$ .
- 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  $\mathbf{u}$ , Obtain the reference configuration  $\mathbf{X}$ ; → 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  $\mathbf{E}$ .
- 29:     **end for**
- 30: **end while**
- 31: *Plot the figures and postprocessing*.

---

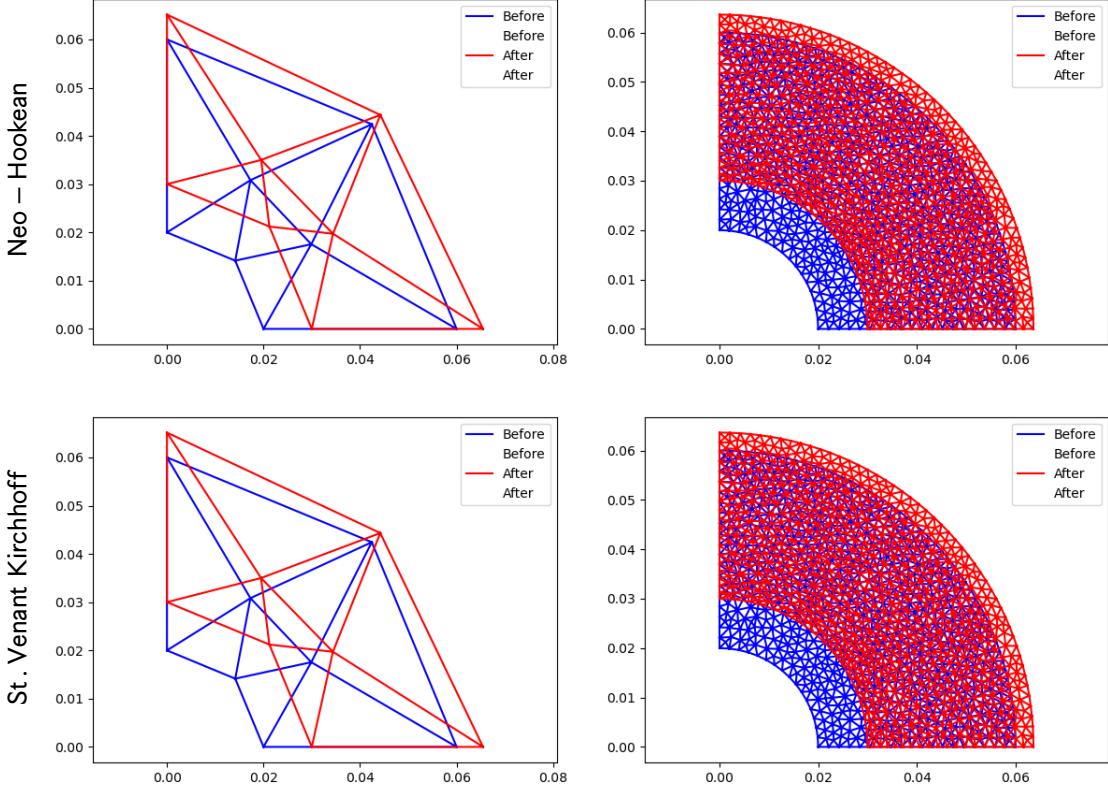


Figure 1: The original (blue) and deformed (red) mesh for the cylinder using both the coarse (left) and fine (right) triangular meshes using the Neo-Hookean (up) and St. Venant Kirchhoff (bottom) constitutive laws for the original given boundary conditions.

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

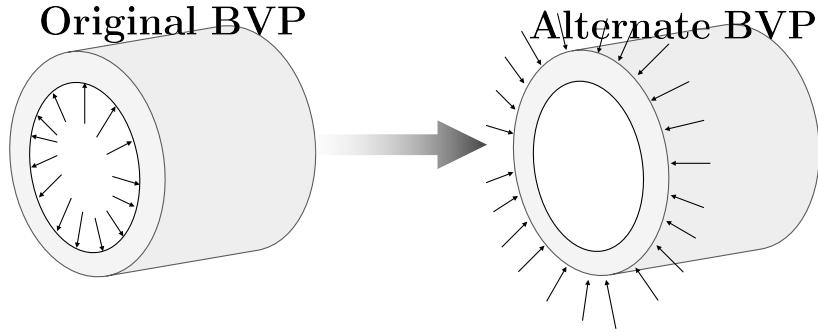


Figure 2: The schematic for my alternate boundary conditions.

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()
```

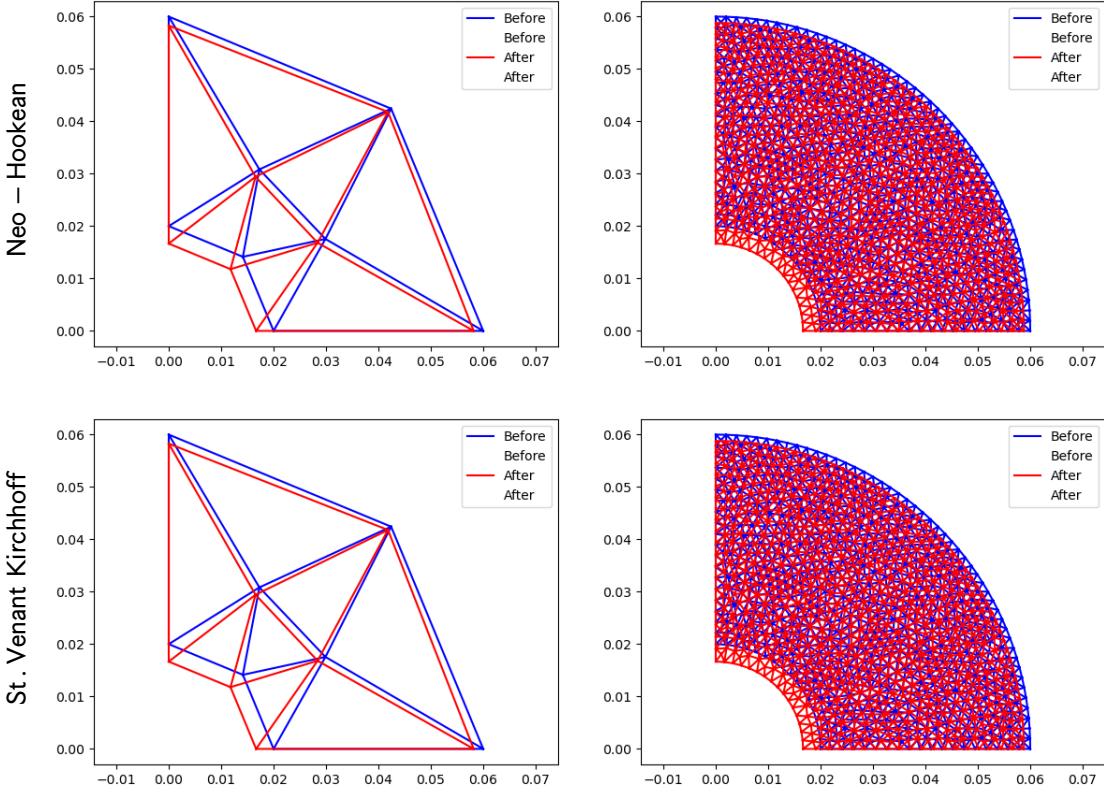


Figure 3: The original (blue) and deformed (red) mesh for the cylinder using both the coarse (left) and fine (right) triangular meshes using the Neo-Hookean (up) and St. Venant (bottom) constitutive laws for the alternate (Figure 2) boundary conditions.

```

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

```

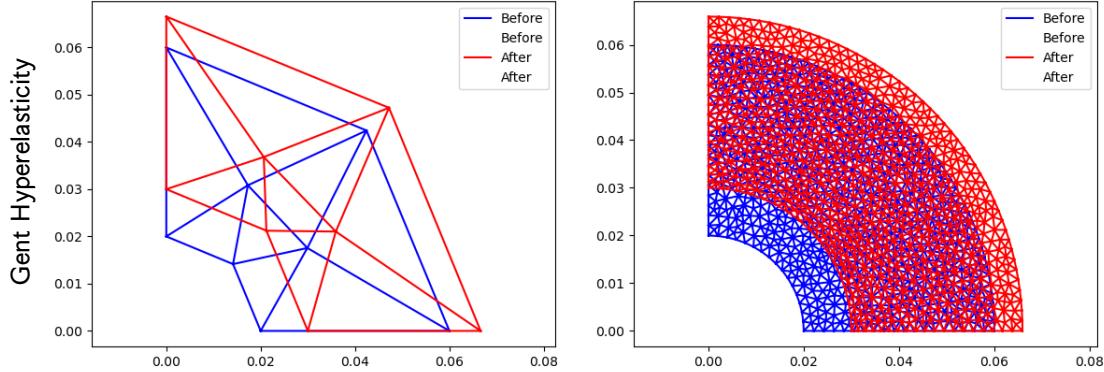


Figure 4: The original (blue) and deformed (red) mesh for the cylinder using both the coarse (left) and fine (right) triangular meshes using the Gent constitutive laws for the original given boundary conditions.

```

32     if dim == 2:
33         return PK2[:, :, :2]
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 J2 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  $\hat{H} = 1930MPa$
- Coefficient of linear kinematic hardening  $H = 1GPa$
- Initial yield stress  $Y_0 = 268MPa$

**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  $5 \times 10^{-5}$  on the interior boundary. For the second boundary condition, I apply a radial displacement of  $-1 \times 10^{-4}$  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*( $\mathbf{M}$ , bcDofs, freeDofs)
- 2:     Cut the degree of freedom of the elasticity tensor  $\mathbf{M}$  into the boundary and free degree of freedom parts.
- 3: **function** *verify*( $\mathbf{U}$ ,  $\mathbf{U}_{ref}$ , bcDofs, tolerance)
- 4:     Checks if the values of  $\mathbf{U}$  at the indices specified in bcDofs are the same as the corresponding values in  $\mathbf{U}_{ref}$  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  $\mathbf{K}$  from displacements.
- 15:     Compute the stiffness matrix  $\mathbf{M}$  from  $\mathbf{K}$ .
- 16:     Solve the unknown displacement  $\mathbf{U}$  from  $\mathbf{M}$  and Residual  $\mathbf{R}$ .
- 17:     Compute Force  $\mathbf{F}$  from  $\mathbf{U}$ ; Compute external force  $\mathbf{F}_{ext} = \mathbf{M}_{11}\mathbf{U}_1 + \mathbf{M}_{12}\mathbf{U}_2$ .
- 18:     Update Residual  $\mathbf{R} = \mathbf{F}_{ext} - \mathbf{F}_{int}$ ; Cut  $\mathbf{R}$  into  $\mathbf{R}_1$  &  $\mathbf{R}_2$ .
- 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  $\mathbf{u}$ , Obtain the reference configuration  $\mathbf{X}$ ;  $\rightarrow$  Get current configuration  $\mathbf{x} = \mathbf{X} + \mathbf{u}$ ;  $\rightarrow$  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  $\mathbf{E}$ .
- 29:     **end for**
- 30: **end while**
- 31: *Plot the figures and postprocessing.*

---

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

---

**Algorithm 2** Constitutive Model Definition

---

```

1: class Plasticity( $E, \nu, H, \hat{H}, Y_0$ )
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

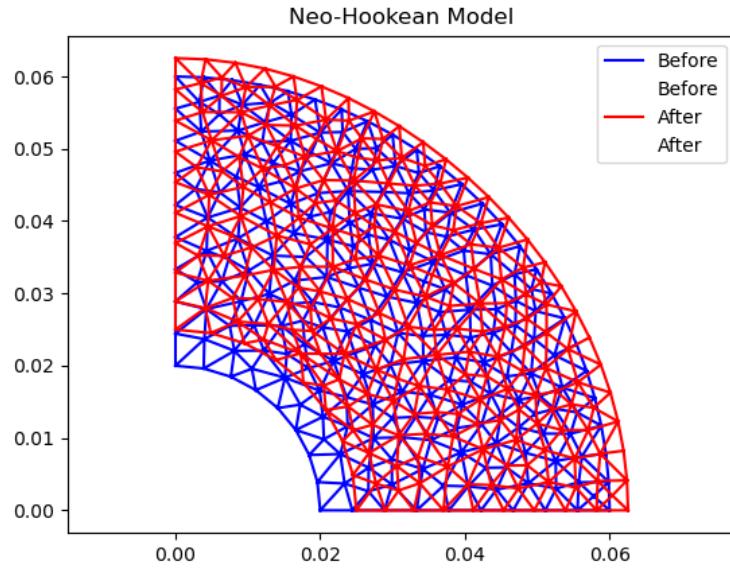


Figure 1: The original and deformed meshes for the Neo-Hookean elastoplasticity model under the first defined boundary conditions.

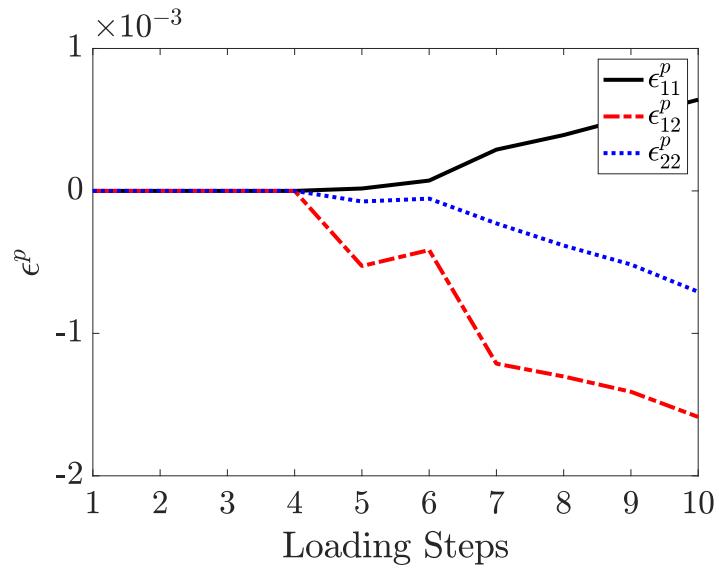


Figure 2: The evolution of the plastic strains ( $\epsilon_{11}$ ,  $\epsilon_{12}$ ,  $\epsilon_{22}$ ) for the Neo-Hookean elastoplasticity model under the first defined boundary conditions.

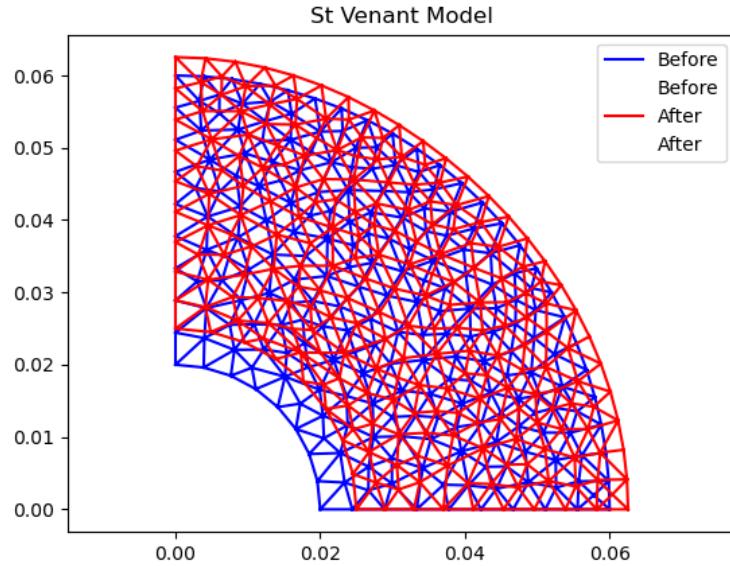


Figure 3: The original and deformed meshes for the St. Venant elastoplasticity model under the first defined boundary conditions.

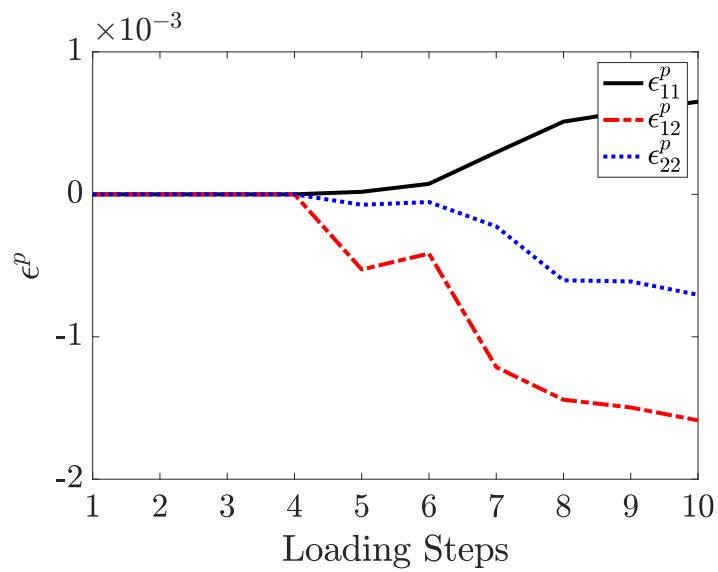


Figure 4: The evolution of the plastic strains ( $\epsilon_{11}$ ,  $\epsilon_{12}$ ,  $\epsilon_{22}$ ) for the St. Venant elastoplasticity model under the first defined boundary conditions.

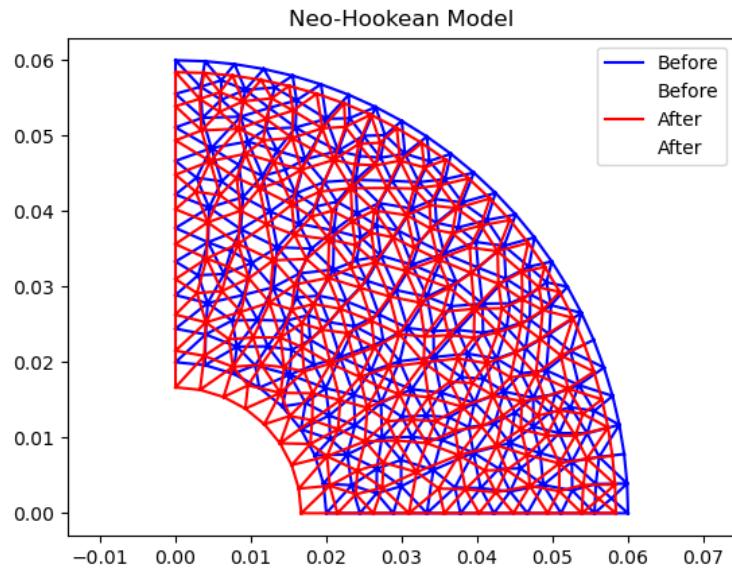


Figure 5: The original and deformed meshes for the Neo-Hookean elastoplasticity model under the second defined boundary conditions.

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.

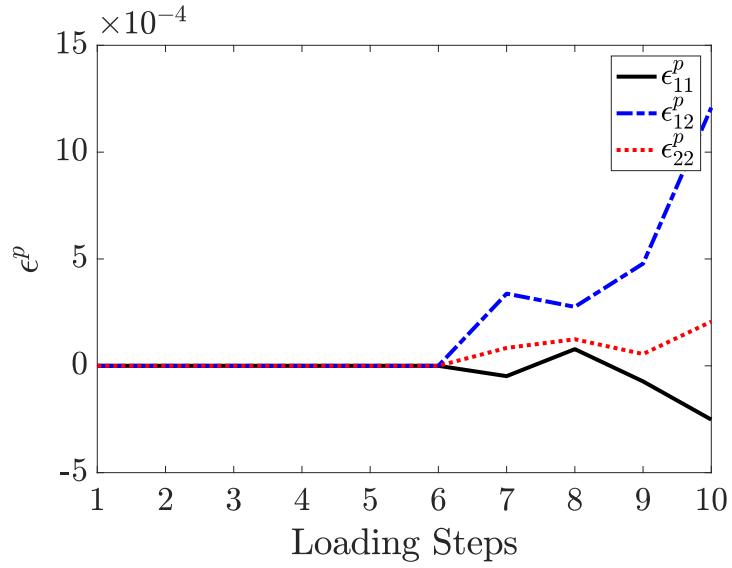


Figure 6: The evolution of the plastic strains ( $\epsilon_{11}$ ,  $\epsilon_{12}$ ,  $\epsilon_{22}$ ) for the Neo-Hookean elastoplasticity model under the second defined boundary conditions.

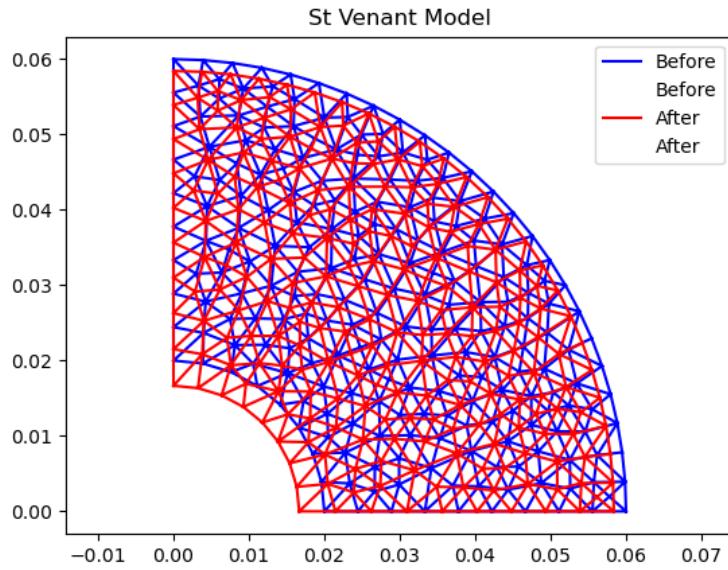


Figure 7: The original and deformed meshes for the St. Venant elastoplasticity model under the second defined boundary conditions.

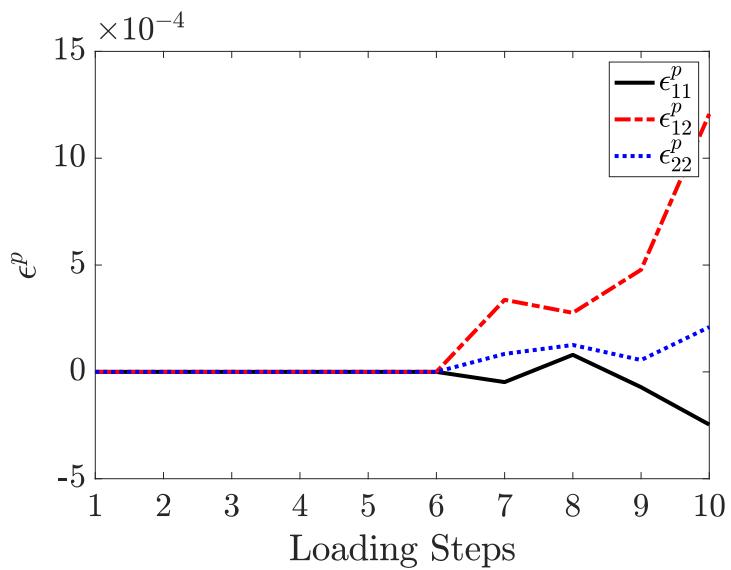


Figure 8: The evolution of the plastic strains ( $\epsilon_{11}$ ,  $\epsilon_{12}$ ,  $\epsilon_{22}$ ) for the St. Venant elastoplasticity model under the second defined boundary conditions.