TECHNICAL UNIVERSITY BERGAKADEMIE FREIBERG

NON-LINEAR FINITE ELEMENT METHODS

PROJECT 2019

Ghogul gopal Kalpana – 63942

Ghogul.gopal-kalpana@student.tu-freiberg.de

Venkatasubramanian Sundaramoorthy – 64143

venkatasubramanian.sundaramoorthy@student.tu-freiberg.de

Contents

1.SHORT NARRATION OF THE GIVEN PROBLEM:

In the given problem, the sphere is the elastic-plastic material and an inclusion inside the sphere. Phase transformation in the inclusion affects the outer material to deform. This phase transformation is purely dilatational (Volumetric strain). Because of this phase transformation, there will be a displacements, stresses and strains will be created in the sphere, which can be calculated using the finite element methods.

In FEM we usually solve from strong form and converted in to weak form and finding the external force and internal force.

Strong form of the given problem:

$$0 = \frac{\partial(r^2\sigma_{rr})}{\partial r} - r(\sigma_{\Phi\Phi} + \sigma_{\theta\theta})$$

Weak form of the given problem:

$$0 = \delta w = \underbrace{\int_{r_i}^{r_o} \delta\varepsilon^T \ \sigma r^2 dr}_{\text{Internal Force}} - \underbrace{[r^2\sigma_{rr}\delta u_r]_{r_i}^{r_o}}_{\text{External Force}}$$

We know that, $\delta\varepsilon^T = B^T\delta u^T$

$$B^T = \left(\frac{\partial N}{\partial r}\right)^T \ (For\ One\ Dimentional\ case)$$

In our case,

$$\varepsilon = \begin{bmatrix} \frac{\partial u}{\partial r} \\ \frac{u}{r} \\ \frac{u}{r} \end{bmatrix} = \begin{bmatrix} \frac{\partial Nu}{\partial r} \\ \frac{Nu}{r} \\ \frac{Nu}{r} \end{bmatrix} = \begin{bmatrix} \frac{\partial N}{\partial r} \\ \frac{N}{r} \\ \frac{N}{r} \end{bmatrix} = \begin{bmatrix} \frac{\partial N}{\partial \xi} * \frac{\partial \xi}{\partial r} \\ \frac{N}{r} \\ \frac{N}{r} \end{bmatrix} = \boxed{\begin{bmatrix} \frac{\partial N}{\partial \xi} * \frac{\partial \xi}{\partial r} \\ \frac{N}{r} \\ \frac{N}{r} \end{bmatrix}}$$

B Matrix

Quadrature with single Gauss point is given, ($\xi = 0$) and weight = 2 (To evaluate the Internal force)

Shape function is given by, $[N](\xi) = \left[\frac{1}{2}(1-\xi), \frac{1}{2}(1+\xi)\right] = \left[\frac{1}{2}, \frac{1}{2}\right]$

Internal Force

$$\int_{r_i}^{r_o} \delta\varepsilon^T \ \sigma r^2 dr = weight * B^T * \ \sigma r^2$$

And we are discretizing the domain(radius) in to several elements (using the mesh refinement condition $h^e(r = r_i) = \frac{1}{5}h^e(r = r_0)$). We can find the stress, strain and displacements in each node and with the help of shape functions we can easily count the displacements over elements.

In our case external force is zero because physically external force acts on the first node only and other nodes are zero while reducing external is totally zero.

As we are dealing with non-linearity numerical methods is one of the ways to solve this problem. For this problem, Newton-Raphson method has been used to find the displacements from this we can easily find the strain and stress for the applied force.

In this elastic-plastic problem, stiffness matrix (K) is depending on the tangent stiffness matrix (C or C_t). Tangent stiffness matrix can be calculated based on the stress (trail stress) values which can be calculated using the initial guess of the displacements. And respective equivalence stress is calculated($\sigma^{eq} = \sqrt{\frac{3}{2}{\sigma_{trail}}^d {\sigma_{trail}}^d}$).The calculated equivalence stress is lesser than the yield stress then elastic case will be called, else plastic case will be called and the corresponding tangent stiffness matrix (C or C_t) will be returned.

Boundary Conditions,

Time frame $\tau \in [0,1]$

Initial condition $u_r(r = r_i) = \frac{1}{3}\tau\epsilon_v r_i, \sigma_{rr}(r = r_o) = 0$

2. AN OVERVIEW OVER THE PROGRAM STRUCTURE

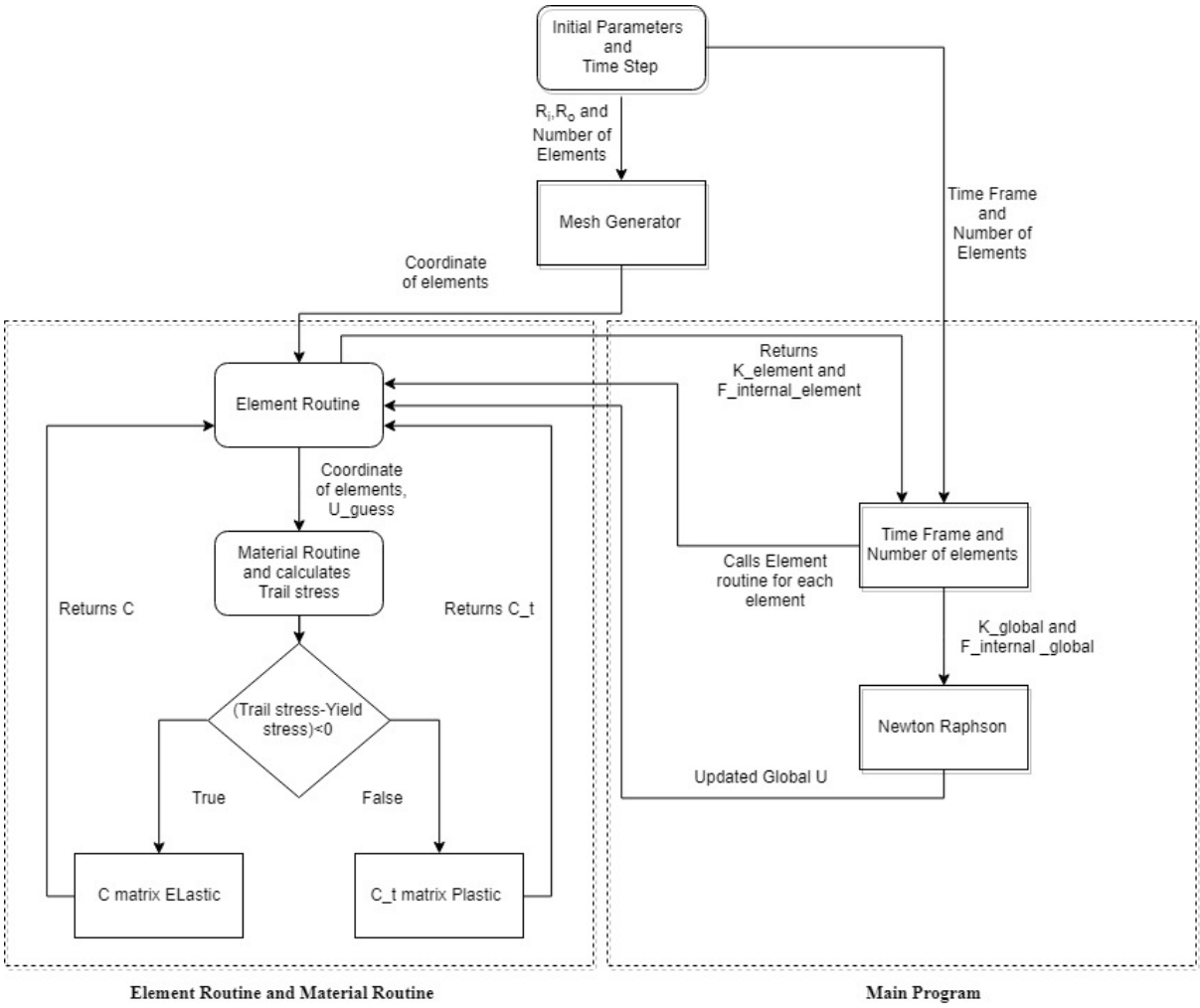


Fig_1: program flow chart

2.1 Initial Parameters and Time step:

```
#Time step parameters
time_step=0.01
time_step_array=np.arange(time_step,1,time_step)
```

Time step is taken as 0.01 and with in the given time frame (0,1),the total time is evenly spaced with a time step

```
#INPUT PARAMETERS
poissons_ratio=0.3
E=0.1
volumetric_strain=-0.01
yield_stress=100e-6

#Mu and Lamda
mu=E/(2*(1+poissons_ratio))
Lamda=poissons_ratio*E/((1-2*poissons_ratio)*(1+poissons_ratio))

#MESH PARAMETERS
inner_radius = 25
outer_radius = 100
no_of_ele = 10
```

Material Parameters, Element parameters and total number of elements are initialized.

Mesh Generator:

```
meshrefinementfactor = 5 #ratio of element sizes at outer and inner radius

#MESH REFINEMENT
q = meshrefinementfactor**(1/(no_of_ele-1))
l_ele = (outer_radius-inner_radius)*(1-q)/(1-meshrefinementfactor*q)

r_node = inner_radius
coordinate=np.array([inner_radius])

#MESH GENERATION
for i in range(no_of_ele):
    r_node = r_node +l_ele
    coordinate = np.append(coordinate,r_node)
    l_ele=l_ele*q
```

The whole domain is discretized into number of elements as per the given condition ($h^e(r = r_i) = \frac{1}{5}h^e(r = r_0)$).

Element Routine and Material Routine:

1.Element Routine:

```
# Element Routine
def Element_fn(coordinate,Lambda,mu,tau):
    der_shape_fn=np.array([-1/2,1/2])

    jacobian=der_shape_fn@np.array([[coordinate[i]],[coordinate[i+1]]])

    jacobian_inverse=np.asscalar(1/jacobian)

    B=np.array([
                [-1/2*jacobian_inverse,1/2*jacobian_inverse],
                [1/(coordinate[i]+coordinate[i+1]),1/(coordinate[i]+coordinate[i+1])],
                [1/(coordinate[i]+coordinate[i+1]),1/(coordinate[i]+coordinate[i+1])]
                ])

    B_T=np.transpose(B)

    C,new_sigma=Material_fn(coordinate,global_u)

    print('Stress',new_sigma)
    K_ele=2*B_T@C@B*jacobian*((coordinate[i]+coordinate[i+1])/2)**2

    f_int_ele=2*(B_T@new_sigma)*jacobian*((coordinate[i]+coordinate[i+1])/2)**2

    return K_ele,f_int_ele
```

B Matrix Calls Material Routine to get the elastic or Plastic Tangent stiffness matrix and returns Element Stiffness matrix(K_ele) and Element F_internal.

2.Material Routine:

```
# #Material Routine
def Material_fn(coordinate,global_u):
    der_shape_fn=np.array([-1/2,1/2])

    jacobian=der_shape_fn@np.array([[coordinate[i]],[coordinate[i+1]]])

    jacobian_inverse=np.asscalar(1/jacobian)

    B=np.array([
                [-1/2*jacobian_inverse,1/2*jacobian_inverse],
                [1/(coordinate[i]+coordinate[i+1]),1/(coordinate[i]+coordinate[i+1])],
                [1/(coordinate[i]+coordinate[i+1]),1/(coordinate[i]+coordinate[i+1])]
                ])

    epsilon = B @ np.array([global_u[i],global_u[i+1]])

    sigma_trail=C@(epsilon-Global_epsilon_p[i])

    sigma_trail_dvt = sigma_trail-(1/3*np.sum(sigma_trail))

    sigma_trail_eql=np.sqrt(3/2*(np.sum(np.square(sigma_trail_dvt))))

    print('Eq',sigma_trail_eql)

    sigma_trail_dvt_tensor=np.diagflat(sigma_trail_dvt)

    sigma_trail_dvt_outer=np.outer(sigma_trail_dvt_tensor,sigma_trail_dvt_tensor)
```

Elastic strain (ε) is calculated based on the initial guess of displacement and trail stress($\sigma_{trail} = C(\varepsilon - \varepsilon_p)$) is calculated based on the elastic strain and initially plastic strain (ε_p) is assumed to be zero. And respective equivalence stress is calculated $\sigma^{eq} = \sqrt{\frac{3}{2}{\sigma_{trail}}^d {\sigma_{trail}}^d}$.

3.Condition Checking:

```
if (sigma_trail_eql-yield_stress) < 0:
    sigma_rr[i]=np.asscalar(sigma_trail[0])
    sigma_phi[i]=np.asscalar(sigma_trail[1])
    return C,sigma_trail
else:
    delta_lamda=(sigma_trail_eql-yield_stress)/(3*mu)
    print('Plastic',tau,i)
    print('delta_lamda',delta_lamda)
    epsilon_p[i]=Global_epsilon_p[i]+delta_lamda*1.5*(sigma_trail_dvt/
    sigma_trail_eql)

    new_trial_stress=C@(epsilon-epsilon_p[i])

    sigma_rr[i]=np.asscalar(new_trial_stress[0])
    sigma_phi[i]=np.asscalar(new_trial_stress[1])

    new_trial_stress_dev=new_trial_stress-(1/3*np.sum(new_trial_stress))

    new_stress_eql=np.sqrt(3/2*(np.sum(np.square(new_trial_stress_dev))))

    C_plastic = k/3*(delta_ij_delta_kl)+2*mu*((sigma_trail_eql-3*mu*delta_lamda)/
    sigma_trail_eql)*I_ijkl-(3*mu*(1/sigma_trail_eql**2)*sigma_trail_dvt_outer)

    value = C_plastic[0::4]
    print(value)
    C_t = value[np.nonzero(value)].reshape(3,3)

    return C_t,new_trial_stress
```

($\sigma_{trail} < yield\_stress$) Elastic C is returned else Plastic C will be returned and the plastic strain is calculated using Euler method (${\varepsilon_p}^{m+1} = \varepsilon_p + \Delta\text{lamda} * 1.5 * (\frac{{\sigma_{trail}}^d}{\sigma^{eq}})$). The new stress is calculated using the elastic strain (ε) and plastic strain (ε_p).

Elastic C matrix:

```
C = np.array([[Lamda+2*mu,Lamda,Lamda],
              [Lamda,Lamda+2*mu,Lamda],
              [Lamda,Lamda,Lamda+2*mu]])
```

Plastic C matrix:

```
C_plastic = k/3*(delta_ij_delta_kl)+2*mu*((sigma_trail_eql-3*mu*delta_lamda)/
sigma_trail_eql)*I_ijkl-(3*mu*(1/sigma_trail_eql**2)*sigma_trail_dvt_outer)
```

Plastic C matrix parameters are initialized before.

```
#Plasticity Parameters
k=3*Lamda+2*mu
delta_ij=np.eye(3)
delta_ij_delta_kl=np.outer(delta_ij,delta_ij)
delta_ik_delta_jl=np.eye(9)
delta_il_delta_jk=np.zeros((9,9))
delta_il_delta_jk[0,0]=1;delta_il_delta_jk[1,3]=1;delta_il_delta_jk[2,6]=1;
delta_il_delta_jk[3,1]=1;delta_il_delta_jk[4,4]=1;delta_il_delta_jk[5,7]=1;
delta_il_delta_jk[6,2]=1;delta_il_delta_jk[7,5]=1;delta_il_delta_jk[8,8]=1
I_ijkl=(1/2*(delta_ik_delta_jl+delta_il_delta_jk))-(1/3*delta_ij_delta_kl)
```

Main program:

Time frame and Number of Elements:

```
for time,tau in  enumerate(time_step_array):
    sigma_rr=np.zeros_like(coordinate)
    sigma_phi=np.zeros_like(coordinate)
    global_u[0]=1/3*tau*(-volumetric_strain)*inner_radius
    delta_u=1
    G_red = np.array([1,1])
    # print('TIME',(time+1))
    epsilon_p=np.zeros((no_of_ele,3,1))
```

For each time frame the first value is replaced by the given condition.

```
 for i in range(no_of_ele):
     K_ele,f_int_ele = Element_fn(coordinate,Lamda,mu,tau)
     Ae=np.zeros((2,no_of_ele+1))
     Ae[0,i]=1
     Ae[1,i+1]=1
     AeT=np.transpose(Ae)


     #Global K Matrix
     K=AeT@K_ele@Ae
     K_global=np.add(K_global,K)

     #Global F_int matrix
     F_int=AeT@f_int_ele
     # print(F_int)
     F_int_global=np.add(F_int_global,F_int)

 K_global_red=np.delete(K_global,0,axis=0)
 K_global_red=np.delete(K_global_red,0,axis=1)

 F_ext_global=np.zeros((no_of_ele+1,1))
```

For every elements K_ele and F_int_ele is calculated and Global K matrix($A_e{}^T K\_ele\_A_e$) and F_int_global($A_e{}^T F\_int\_ele$) is calculated.

Newton Raphson:

Condition for the newton Raphson method

```
 while np.linalg.norm(delta_u)>(0.005*np.linalg.norm(u_reduced)) or np.linalg.norm
 (G_matrix_reduced)>(0.005*np.linalg.norm(F_int_global)) :
```

```
#Newton Raphson Method
G_matrix = F_int_global-F_ext_global

G_matrix_reduced = np.delete(G_matrix,(0),axis=0)

delta_u=np.linalg.inv(K_global_red)@G_matrix_reduced
u_reduced=u_reduced-delta_u

global_u = np.insert(u_reduced,(0),1/3*tau*(-volumetric_strain)*inner_radius).reshape(no_of_ele+1,1)
```

Which calculates the Global u and initial value will be updated and the next time frame will be calculated.

3.MANUAL FOR PROGRAM:

The program file is in the python format so that we can open the code in any python editor. The parameters are already defined in the program and the parameters are according to given variants.

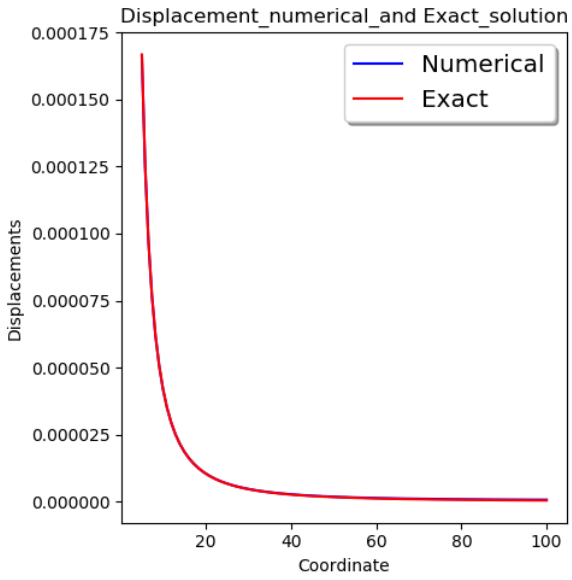
```
#INPUT PARAMETERS
poissons_ratio=0.3
E=0.1
volumetric_strain=-0.01
yield_stress=100e-6

#Mu and Lamda
mu=E/(2*(1+poissons_ratio))
Lamda=poissons_ratio*E/((1-2*poissons_ratio)*(1+poissons_ratio))

#MESH PARAMETERS
inner_radius = 25
outer_radius = 100
no_of_ele = 10
```

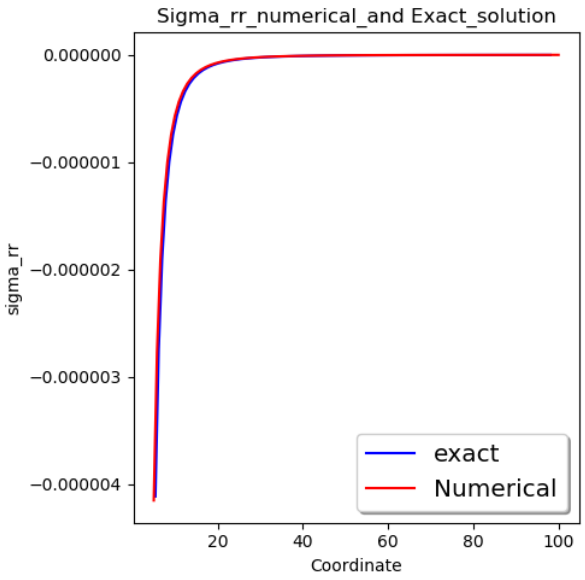
The program generates both analytical and numerical solutions. The outputs are displacements along the nodes, stress along the nodes and finally it gives graphical representation of the radial stress in various time frames. Graphical representation of the outputs will save in directory where the program file is located.

4.VERFICATION AND RESULTS

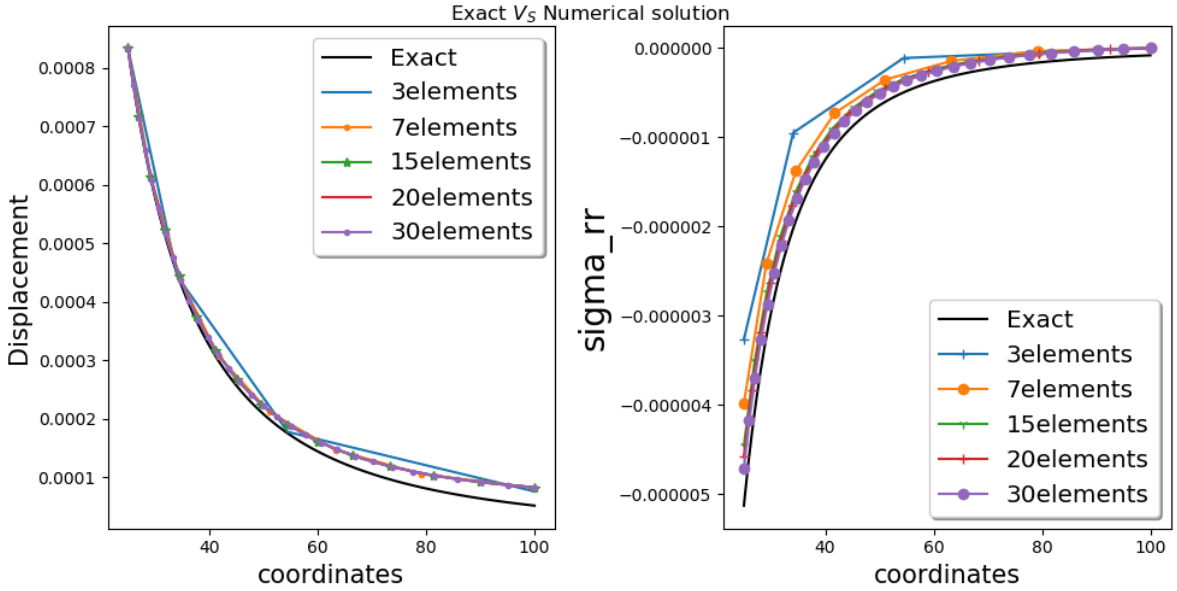
Verification:

Elastic Displacement and Sigma_rr with the limiting case $(r_0/r_i) \rightarrow \infty$



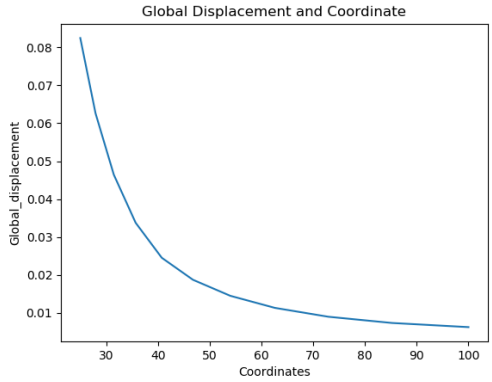


Convergence study with number of elements and exact solution:

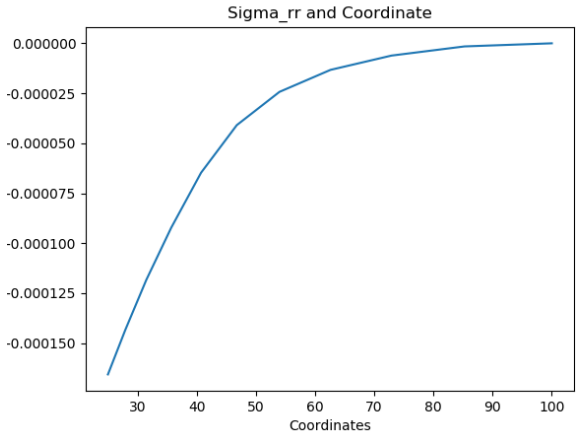


Results:

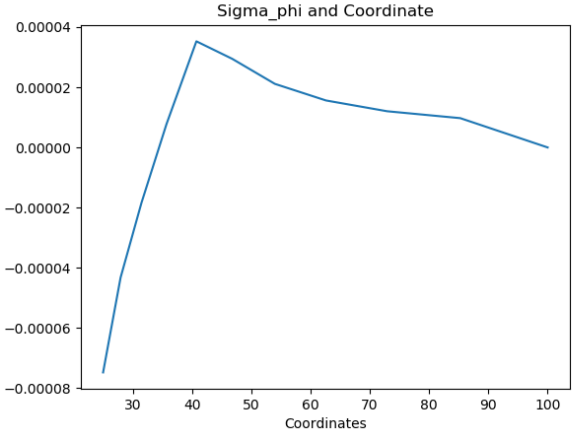
Displacements and Coordinates:



Sigma_rr and Coordinates:



Sigma_phi and Coordinates:



Sigma_rr at ri and Time_frame: