3D04_hole_in_plate_v0p8-_A.Flowers_Commented

July 6, 2025

1 Tension of a 3D plate with a hole

1.0.1 Units

- Length: mm
- Mass: kg
- Time: s
- Force: milliNewtons
- Stress: kPa

1.0.2 Software:

- Dolfinx v0.8.0

In the collection "Example Codes for Coupled Theories in Solid Mechanics,"

By Eric M. Stewart, Shawn A. Chester, and Lallit Anand.

https://solidmechanicscoupledtheories.github.io/

2 Import modules

[1]:
```
# Import FEnicSx/dolfinx
import dolfinx

# For numerical arrays
import numpy as np

# For MPI-based parallelization
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# PETSc solvers
from petsc4py import PETSc

# specific functions from dolfinx modules
from dolfinx import fem, mesh, io, plot, log
from dolfinx.fem import (Constant, dirichletbc, Function, functionspace,␣
 ↪Expression )
from dolfinx.fem.petsc import NonlinearProblem
from dolfinx.nls.petsc import NewtonSolver
from dolfinx.io import VTXWriter, XDMFFile


# specific functions from ufl modules
import ufl
from ufl import (TestFunctions, TrialFunction, Identity, grad, det, div, dev,␣
 ↪inv, tr, sqrt, conditional ,\
                 gt, dx, inner, derivative, dot, ln, split)

# basix finite elements (necessary for dolfinx v0.8.0)
import basix
from basix.ufl import element, mixed_element, quadrature_element

# Matplotlib for plotting
import matplotlib.pyplot as plt
plt.close('all')

# For timing the code
from datetime import datetime


# Set level of detail for log messages (integer)
# Guide:
# CRITICAL  = 50, // errors that may lead to data corruption
# ERROR     = 40, // things that HAVE gone wrong
# WARNING   = 30, // things that MAY go wrong later
# INFO      = 20, // information of general interest (includes solver info)
# PROGRESS  = 16, // what's happening (broadly)
# TRACE     = 13, // what's happening (in detail)
# DBG       = 10  // sundry
#
log.set_log_level(log.LogLevel.WARNING)
```

3 Define geometry

[2]:
```
# Dimensions of one quarter of hole in plate specimen
#
L0 = 15.0    # Length mm
W0 = 10.0    # Width mm
t0 = 1.0     # Thickness mm

# Pull in the mesh *.xdmf file and read any named domains in the mesh.
```

```
with XDMFFile(MPI.COMM_WORLD,"meshes/3D_hip_v2.xdmf",'r') as infile:
    domain = infile.read_mesh(name="Grid",xpath="/Xdmf/Domain")
    cell_tags = infile.read_meshtags(domain,name="Grid")

# Create facet to cell connectivity required to determine boundary facets.
domain.topology.create_connectivity(domain.topology.dim, domain.topology.dim-1)

# Read in facet tags from an *.xdmf file.
with XDMFFile(MPI.COMM_WORLD, "meshes/facet_3D_hip_v2.xdmf", "r") as xdmf:
    facet_tags = xdmf.read_meshtags(domain, name="Grid")

x = ufl.SpatialCoordinate(domain)
```

[ ]:
```
##Flowers Comments

# Dimensions of one quarter of hole in plate specimen
L0 = 15.0   # Length mm
W0 = 10.0   # Width mm
t0 = 1.0    # Thickness mm
##Plate dimensions

# Pull in the mesh *.xdmf file and read any named domains in the mesh.
with XDMFFile(MPI.COMM_WORLD,"meshes/3D_hip_v2.xdmf",'r') as infile:
##Parallel execution implemented. Mesh file called on to read
    domain = infile.read_mesh(name="Grid",xpath="/Xdmf/Domain")
##Mesh read from XDMF. Domain called on within XDMF structure
    cell_tags = infile.read_meshtags(domain,name="Grid")
##Mesh tags called on. This differenciates change in boundary surface of mesh␣
 ↪(i.e. increase/decrease of elements or element shape)

# Create facet to cell connectivity required to determine boundary facets.
domain.topology.create_connectivity(domain.topology.dim, domain.topology.dim-1)
##Mesh topology (structure) defined and dimension of mesh
##Connectivity constructs cells to facets via FEniCSX

# Read in facet tags from an *.xdmf file.
with XDMFFile(MPI.COMM_WORLD, "meshes/facet_3D_hip_v2.xdmf", "r") as xdmf:
    facet_tags = xdmf.read_meshtags(domain, name="Grid")
##Cells=3D; Facets=2D. Defines boundary conditions within the mesh
###Think Boundary Conditions set in Pointwise-vs-ANSYS

x = ufl.SpatialCoordinate(domain)
##x,y,z spatial coordinates used from boundary conditions
```

Print out the unique facet index numbers

```
[3]: top_imap = domain.topology.index_map(2)      # index map of 2D entities in␣
      ↪domain (facets)
     values = np.zeros(top_imap.size_global)      # an array of zeros of the same␣
      ↪size as number of 2D entities
     values[facet_tags.indices]=facet_tags.values # populating the array with facet␣
      ↪tag index numbers
     print(np.unique(facet_tags.values))          # printing the unique indices

     # Surface labels from gmsh:
     # Physical Surface("xbot", 33)
     # Physical Surface("ybot", 34)
     # Physical Surface("xtop", 35)
```

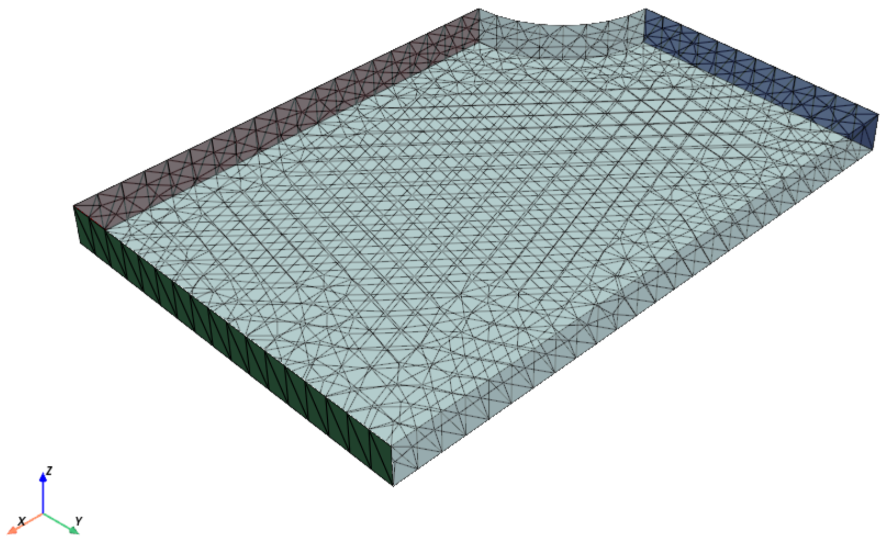
```
[33 34 35]
```

Visualize reference configuration and boundary facets

```
[19]: import pyvista
      pyvista.set_jupyter_backend('html')
      from dolfinx.plot import vtk_mesh
      pyvista.start_xvfb()

      # initialize a plotter
      plotter = pyvista.Plotter()

      # Add the mesh -- I make the 3D mesh opaque, so that 2D surfaces stand out.
      topology, cell_types, geometry = plot.vtk_mesh(domain, domain.topology.dim)
      grid = pyvista.UnstructuredGrid(topology, cell_types, geometry)
      plotter.add_mesh(grid, show_edges=True, opacity=0.5)

      # Add colored 2D surfaces for the named surfaces
      xBot = pyvista.UnstructuredGrid(*vtk_mesh(domain, domain.topology.
       ↪dim-1,facet_tags.indices[facet_tags.values==33]) )
      yBot = pyvista.UnstructuredGrid(*vtk_mesh(domain, domain.topology.
       ↪dim-1,facet_tags.indices[facet_tags.values==34]) )
      xTop = pyvista.UnstructuredGrid(*vtk_mesh(domain, domain.topology.
       ↪dim-1,facet_tags.indices[facet_tags.values==35]) )
      #
      actor = plotter.add_mesh(xBot, show_edges=True,color="blue") # top face is blue
      actor2 = plotter.add_mesh(yBot, show_edges=True,color="red") # sides are red
      actor3 = plotter.add_mesh(xTop, show_edges=True,color="green") # bottom face is␣
       ↪green

      labels = dict(zlabel='Z', xlabel='X', ylabel='Y')
      plotter.add_axes(**labels)

      plotter.screenshot("mesh.png")
```

```
from IPython.display import Image
Image(filename='mesh.png')
```

[19]:



3.1 Define boundary and volume integration measure

[5]:
```
# Surface labels from gmsh:
# Physical Surface("xbot", 33)
# Physical Surface("ybot", 34)
# Physical Surface("xtop", 35)

# Define the boundary integration measure "ds" using the facet tags,
# also specify the number of surface quadrature points.
ds = ufl.Measure('ds', domain=domain, subdomain_data=facet_tags,␣
 ↪metadata={'quadrature_degree':4})

# Define the volume integration measure "dx"
# also specify the number of volume quadrature points.
dx = ufl.Measure('dx', domain=domain, metadata={'quadrature_degree': 4})

#  Define facet normal
```

```
n = ufl.FacetNormal(domain)
```

4 Material parameters

-Arruda-Boyce model

[6]:
```
Gshear_0 = Constant(domain,PETSc.ScalarType(280.0))            # Ground state␣
 ↪shear modulus
lambdaL  = Constant(domain,PETSc.ScalarType(5.12))             # Locking stretch
Kbulk    = Constant(domain,PETSc.ScalarType(1000.0*Gshear_0))
```

5 Simulation time-control related params

[7]:
```
# Initialize time
t = 0.0
#  Stretch parameters
stretch  = 3.0                # axial stretch
dispTot  = (stretch-1)*L0     # axial displacement. Remember L0 is the initial␣
 ↪gage length
rate     = 1.e0
Ttot     = (stretch-1)/rate
numSteps = 100
dt       = Ttot/numSteps      # (fixed) step size

# Function to linearly ramp up displacement on boundary.
def dispRamp(t):
    return dispTot*t/Ttot
```

6 Function spaces

[8]:
```
# Define function space, both vectorial and scalar
U2 = element("Lagrange", domain.basix_cell(), 2, shape=(3,)) # For displacement
P1 = element("Lagrange", domain.basix_cell(), 1)  # For  pressure              ␣
 ↪
#
TH = mixed_element([U2, P1])      # Taylor-Hood style mixed element
ME = functionspace(domain, TH)     # Total space for all DOFs

# Define actual functions with the required DOFs
w    = Function(ME)
u, p = split(w)  # displacement u, pressure p

# A copy of functions to store values in the previous step
w_old         = Function(ME)
u_old,  p_old = split(w_old)
```

```
# Define test functions
u_test, p_test = TestFunctions(ME)

# Define trial functions needed for automatic differentiation
dw = TrialFunction(ME)
```

7 Initial conditions

- The initial conditions for degrees of freedom u and p are zero everywhere
- These are imposed automatically, since we have not specified any non-zero initial conditions.

8 Subroutines for kinematics and constitutive equations

[9]:
```
# Deformation gradient
def F_calc(u):
    Id = Identity(3)
    F  = Id + grad(u)
    return F

def lambdaBar_calc(u):
    F = F_calc(u)
    C = F.T*F
    Cdis = J**(-2/3)*C
    I1 = tr(Cdis)
    lambdaBar = sqrt(I1/3.0)
    return lambdaBar

def zeta_calc(u):
    lambdaBar = lambdaBar_calc(u)
    # Use Pade approximation of Langevin inverse
    z    = lambdaBar/lambdaL
    z    = conditional(gt(z,0.95), 0.95, z) # Keep simulation from blowing up
    beta = z*(3.0 - z**2.0)/(1.0 - z**2.0)
    zeta = (lambdaL/(3*lambdaBar))*beta
    return zeta

# Generalized shear modulus for Arruda-Boyce model
def Gshear_AB_calc(u):
    zeta    = zeta_calc(u)
    Gshear  = Gshear_0 * zeta
    return Gshear

#----------------------------------------------
# Subroutine for calculating the Cauchy stress
#----------------------------------------------
```

```
def T_calc(u,p):
    Id = Identity(3)
    F   = F_calc(u)
    J = det(F)
    B = F*F.T
    Bdis = J**(-2/3)*B
    Gshear  = Gshear_AB_calc(u)
    T = (1/J)* Gshear * dev(Bdis) - p * Id
    return T

#----------------------------------------------
# Subroutine for calculating the Piola  stress
#----------------------------------------------
def Piola_calc(u, p):
    Id = Identity(3)
    F   = F_calc(u)
    J = det(F)
    #
    T   = T_calc(u,p)
    #
    Tmat   = J * T * inv(F.T)
    return Tmat
```

[ ]:
```
##A.Flowers Comments

# Deformation gradient
def F_calc(u):
    Id = Identity(3)
    F  = Id + grad(u)
    return F
##Calculation for deformation gradient tensor F

def lambdaBar_calc(u):
    F = F_calc(u)
    C = F.T*F
    Cdis = J**(-2/3)*C
    I1 = tr(Cdis)
    lambdaBar = sqrt(I1/3.0)
    return lambdaBar
##Scalar stretch measure used in hyperelasticity models

def zeta_calc(u):
    lambdaBar = lambdaBar_calc(u)
##Isochoric stretch from deformation
    # Use Pade approximation of Langevin inverse
    z    = lambdaBar/lambdaL
##Normalizes stretch from polymer network
```

```
    z    = conditional(gt(z,0.95), 0.95, z) # Keep simulation from blowing up
##Prevents numeric instability; Langevin function because singular
    beta = z*(3.0 - z**2.0)/(1.0 - z**2.0)
##Pade approximation; used for stability
    zeta = (lambdaL/(3*lambdaBar))*beta
    return zeta
##Stress scalar from statistical mechanics model for polymers; accounts for␣
 ↪finite chain extensibility. Stress tensors for nonlinear chain elasticity

# Generalized shear modulus for Arruda-Boyce model
def Gshear_AB_calc(u):
##Effective shear for nonlinear hyperelastic material
    zeta    = zeta_calc(u)
##Stretch dependent factor using inverse Langevin. Increasing of stretch=␣
 ↪polymers stiffen
    Gshear  = Gshear_0 * zeta
    return Gshear
##Shear module grows due to deformation
##This is important due to modeling with biological materials (i.e. tissue);␣
 ↪nonlinear and stretch sensitive

#---------------------------------------------
# Subroutine for calculating the Cauchy stress
#---------------------------------------------
def T_calc(u,p):
    Id = Identity(3)
    F   = F_calc(u)
##Deformation gradient
    J = det(F)
##Jacobian (volume change due to deformation)
    B = F*F.T
##Cauchy-Green tensor
    Bdis = J**(-2/3)*B
##Removes volumetric part to get isochoric (volume perserving aspect)
    Gshear  = Gshear_AB_calc(u)
##Stretch dependent shear
    T = (1/J)* Gshear * dev(Bdis) - p * Id
    return T
##Cauchy stress calculation; shape change and pressure separated to obtain␣
 ↪deformed configuration

#----------------------------------------------
# Subroutine for calculating the Piola  stress
#----------------------------------------------
def Piola_calc(u, p):
    Id = Identity(3)
    F   = F_calc(u)
    J = det(F)
    T    = T_calc(u,p)
    Tmat    = J * T * inv(F.T)
    return Tmat
##Piola stress used in weak form of balance equation, with displacement␣
 ↪gradient; defined in terms of reference coordinates
```

9 Evaluate kinematics and constitutive relations

[10]:
```
F =  F_calc(u)
J = det(F)
lambdaBar = lambdaBar_calc(u)

# Piola stress
Tmat = Piola_calc(u, p)
```

[ ]:
```
##A.Flowers Comments

F =  F_calc(u)
##F= deformation gradient tensor; u= displacement (unkown and solving for)
J = det(F)
##J= Jacobian determinant. Volume change during deformation
lambdaBar = lambdaBar_calc(u)
##incompressible hyperelasticity; seperates volumetric deviatoric, shape␣
 ↪changing parts of deformation. Volume corrected stretch is calculated for␣
 ↪use in isochoric strain energy

# Piola stress
Tmat = Piola_calc(u, p)
##Computes Piola stress tensor from displacement field (u) and pressure (p)
```

10 Weak forms

[11]:
```
# Residuals:
# Res_0: Balance of forces (test fxn: u)
# Res_1: Coupling pressure (test fxn: p)

# The weak form for the equilibrium equation. No body force
Res_0 = inner(Tmat , grad(u_test) )*dx

# The weak form for the pressure
fac_p = ln(J)/J
#
Res_1 = dot( (p/Kbulk + fac_p), p_test)*dx

# Total weak form
```

```
Res = Res_0 +  Res_1

# Automatic differentiation tangent:
a = derivative(Res, w, dw)
```

[ ]:
```
##A.Flowers Comments

# Residuals:
# Res_0: Balance of forces (test fxn: u)
# Res_1: Coupling pressure (test fxn: p)

# The weak form for the equilibrium equation. No body force
Res_0 = inner(Tmat , grad(u_test) )*dx
##Mechanical residual of weak form for nonlinear elasticity; used to build the␣
 ↪residual vector. Used in FE for a deforming solid

# The weak form for the pressure
fac_p = ln(J)/J
##Scalar factor used for compressible / incompressible materials due to␣
 ↪pressure / energy in nonlinear elasticity

Res_1 = dot( (p/Kbulk + fac_p), p_test)*dx
##Residual defined for pressure field in FE due to incompressible materials.␣
 ↪Differentiates volumetric strain energy

# Total weak form
Res = Res_0 +  Res_1
##Defines total residual of weak form; from force balance (linear momentum /␣
 ↪mechanical equillibrium) and incompressibility (pressure equation)

# Automatic differentiation tangent:
a = derivative(Res, w, dw)
##Jacobian form to solve for nonlinear PDE
```

11 Set-up output files

[12]:
```
# results file name
results_name = "3D_HIP"

# v0.8.0 syntax:
U1 = element("DG", domain.basix_cell(), 1, shape=(3,)) # For displacement
P0 = element("DG", domain.basix_cell(), 1)  # For  pressure

V2 = fem.functionspace(domain, U1) #Vector function space
V1 = fem.functionspace(domain, P0) #Scalar function space, must be␣
 ↪discontinuous here since materials are discontinuous.
```

```python
# fields to write to output file
u_vis = Function(V2)
u_vis.name = "disp"

p_vis = Function(V1)
p_vis.name = "p"

J_vis = Function(V1)
J_vis.name = "J"
J_expr = Expression(J,V1.element.interpolation_points())

lambdaBar_vis = Function(V1)
lambdaBar_vis.name = "lambdaBar"
lambdaBar_expr = Expression(lambdaBar,V1.element.interpolation_points())

P11 = Function(V1)
P11.name = "P11"
P11_expr = Expression(Tmat[0,0],V1.element.interpolation_points())
P22 = Function(V1)
P22.name = "P22"
P22_expr = Expression(Tmat[1,1],V1.element.interpolation_points())
P33 = Function(V1)
P33.name = "P33"
P33_expr = Expression(Tmat[2,2],V1.element.interpolation_points())

T   = Tmat*F.T/J
T0   = T - (1/3)*tr(T)*Identity(3)
Mises = sqrt((3/2)*inner(T0, T0))
Mises_vis= Function(V1,name="Mises")
Mises_expr = Expression(Mises,V1.element.interpolation_points())


# set up the output VTX files.
file_results = VTXWriter(
    MPI.COMM_WORLD,
    "results/" + results_name + ".bp",
    [  # put the functions here you wish to write to output
        u_vis, p_vis, J_vis, P11, P22, P33, lambdaBar_vis,
        Mises_vis,
    ],
    engine="BP4",
)

def writeResults(t):
       # Output field interpolation
```

```
        u_vis.interpolate(w.sub(0))
        p_vis.interpolate(w.sub(1))
        J_vis.interpolate(J_expr)
        P11.interpolate(P11_expr)
        P22.interpolate(P22_expr)
        P33.interpolate(P33_expr)
        lambdaBar_vis.interpolate(lambdaBar_expr)
        Mises_vis.interpolate(Mises_expr)

        # Write output fields
        file_results.write(t)
```

12 Infrastructure for pulling out time history data (force, displacement, etc.)

[13]:
```
# infrastructure for evaluating functions at a certain point efficiently
pointForDisp = np.array([L0, W0, 0])
bb_tree = dolfinx.geometry.bb_tree(domain,domain.topology.dim)
cell_candidates = dolfinx.geometry.compute_collisions_points(bb_tree,␣
 ↪pointForDisp)
colliding_cells = dolfinx.geometry.compute_colliding_cells(domain,␣
 ↪cell_candidates, pointForDisp).array

# computing the reaction force using the stress field
rxnForce = fem.form(P11*ds(33)) #P11*ds

# Surface labels from gmsh:
# Physical Surface("xbot", 33)
# Physical Surface("ybot", 34)
# Physical Surface("xtop", 35)
```

13 Name the analysis step

[14]:
```
# Give the step a descriptive name
step = "Stretch"
```

13.1 Boundary condtions

[15]:
```
# Constant for applied displacement
# For now set the value zero. This value will be updated each step of the␣
 ↪solution procedure.
disp_cons = Constant(domain,PETSc.ScalarType(dispRamp(0)))

# Find the specific DOFs which will be constrained.
```

```
xBot_u1_dofs = fem.locate_dofs_topological(ME.sub(0).sub(0), facet_tags.dim,␣
 ↪facet_tags.find(33))
yBot_u2_dofs = fem.locate_dofs_topological(ME.sub(0).sub(1), facet_tags.dim,␣
 ↪facet_tags.find(34))
xTop_u1_dofs = fem.locate_dofs_topological(ME.sub(0).sub(0), facet_tags.dim,␣
 ↪facet_tags.find(35))
xTop_u2_dofs = fem.locate_dofs_topological(ME.sub(0).sub(1), facet_tags.dim,␣
 ↪facet_tags.find(35))
xTop_u3_dofs = fem.locate_dofs_topological(ME.sub(0).sub(2), facet_tags.dim,␣
 ↪facet_tags.find(35))

# building Dirichlet BCs
bcs_1 = dirichletbc(0.0, xBot_u1_dofs,ME.sub(0).sub(0))  # u1 fix - xBot
bcs_2 = dirichletbc(0.0, yBot_u2_dofs, ME.sub(0).sub(1))  # u2 fix - yBot
#
bcs_3 = dirichletbc(disp_cons, xTop_u1_dofs, ME.sub(0).sub(0))  # disp ramp -␣
 ↪xTop
bcs_4 = dirichletbc(0.0, xTop_u2_dofs, ME.sub(0).sub(1))  # u2 fix - xTop
bcs_5 = dirichletbc(0.0, xTop_u3_dofs, ME.sub(0).sub(2))  # u3 fix - xTop

bcs = [bcs_1, bcs_2, bcs_3, bcs_4, bcs_5]
```

[ ]:
```
##Flowers Comments

# Constant for applied displacement
# For now set the value zero. This value will be updated each step of the␣
 ↪solution procedure.
disp_cons = Constant(domain,PETSc.ScalarType(dispRamp(0)))
##Parameter defines stretch of cube over time. dispRamp(0) is displacemnet of␣
 ↪time initially set at 0
##Gives constant displacement value; Application of Dirichlet to FE
##Uniform scalar applied to mesh domain

# Find the specific DOFs which will be constrained.
xBot_u1_dofs = fem.locate_dofs_topological(ME.sub(0).sub(0), facet_tags.dim,␣
 ↪facet_tags.find(33))
yBot_u2_dofs = fem.locate_dofs_topological(ME.sub(0).sub(1), facet_tags.dim,␣
 ↪facet_tags.find(34))
xTop_u1_dofs = fem.locate_dofs_topological(ME.sub(0).sub(0), facet_tags.dim,␣
 ↪facet_tags.find(35))
xTop_u2_dofs = fem.locate_dofs_topological(ME.sub(0).sub(1), facet_tags.dim,␣
 ↪facet_tags.find(35))
xTop_u3_dofs = fem.locate_dofs_topological(ME.sub(0).sub(2), facet_tags.dim,␣
 ↪facet_tags.find(35))
##Function in dolfinx used to identify degrees of freedom (DOFs) due to mesh␣
 ↪entities. v=function space of displacement; entity_dim= dimension of geometry
```

```
##DoFs defined for boundary surface of mesh due to displacement field

# building Dirichlet BCs
bcs_1 = dirichletbc(0.0, xBot_u1_dofs,ME.sub(0).sub(0))  # u1 fix - xBot
bcs_2 = dirichletbc(0.0, yBot_u2_dofs, ME.sub(0).sub(1))  # u2 fix - yBot
#Sets boundary condition of x,y (plate edge)

bcs_3 = dirichletbc(disp_cons, xTop_u1_dofs, ME.sub(0).sub(0))  # disp ramp -␣
 ↪xTop
bcs_4 = dirichletbc(0.0, xTop_u2_dofs, ME.sub(0).sub(1))  # u2 fix - xTop
bcs_5 = dirichletbc(0.0, xTop_u3_dofs, ME.sub(0).sub(2))  # u3 fix - xTop
##Time-dependent displacement ramp defined (x) while y,z are fixed

bcs = [bcs_1, bcs_2, bcs_3, bcs_4, bcs_5]
##Command to solver to implement when calculating solution; applying the␣
 ↪boundary conditions
```

[ ]:
```
## Define the nonlinear variational problem
```

[16]:
```
# # Optimization options for the form compiler



# Set up nonlinear problem
problem = NonlinearProblem(Res, w, bcs, a)

# the global newton solver and params
solver = NewtonSolver(MPI.COMM_WORLD, problem)
solver.convergence_criterion = "incremental"
solver.rtol = 1e-8
solver.atol = 1e-8
solver.max_it = 50
solver.report = True

#  The Krylov solver parameters.
ksp = solver.krylov_solver
opts = PETSc.Options()
option_prefix = ksp.getOptionsPrefix()
opts[f"{option_prefix}ksp_type"] = "preonly" # "preonly" works equally well
opts[f"{option_prefix}pc_type"] = "lu" # do not use 'gamg' pre-conditioner
opts[f"{option_prefix}pc_factor_mat_solver_type"] = "mumps"
opts[f"{option_prefix}ksp_max_it"] = 30
ksp.setFromOptions()
```

13.2 Start calculation loop

[17]:
```python
# Variables for storing time history
totSteps = numSteps+1
timeHist0 = np.zeros(shape=[totSteps])
timeHist1 = np.zeros(shape=[totSteps])
timeHist2 = np.zeros(shape=[totSteps])

#Iinitialize a counter for reporting data
ii=0

# Write initial state to file
writeResults(t=0.0)

# Print out message for simulation start
print("------------------------------------")
print("Simulation Start")
print("------------------------------------")
# Store start time
startTime = datetime.now()

# Time-stepping solution procedure loop
while (round(t + dt, 9) <= Ttot):

    # increment time
    t += dt
    # increment counter
    ii += 1

    # update time variables in time-dependent BCs
    disp_cons.value = dispRamp(t)

    # Solve the problem
    try:
        (iter, converged) = solver.solve(w)
    except: # Break the loop if solver fails
        print("Ended Early")
        break

    # Collect results from MPI ghost processes
    w.x.scatter_forward()

    # Write output to file
    writeResults(t)

    # Update DOFs for next step
    w_old.x.array[:] = w.x.array
    # Store time history variables at this time
    timeHist0[ii] = w.sub(0).sub(0).eval([L0, W0, 0.0],colliding_cells[0])[0] #␣
↪time history of displacement
    #
    timeHist1[ii] =  domain.comm.gather(fem.assemble_scalar(rxnForce))[0] #␣
↪time history of engineering stress

    # Print progress of calculation
    if ii%1 == 0:
        now = datetime.now()
        current_time = now.strftime("%H:%M:%S")
        print("Step: {} | Increment: {}, Iterations: {}".\
              format(step, ii, iter))
        print("      Simulation Time: {} s  of  {} s".\
              format(round(t,4), Ttot))
        print()


# close the output file.
file_results.close()

# End analysis
print("------------------------------------------")
print("End computation")
# Report elapsed real time for the analysis
endTime = datetime.now()
elapseTime = endTime - startTime
print("------------------------------------------")
print("Elapsed real time:  {}".format(elapseTime))
print("------------------------------------------")
```

```
------------------------------------
Simulation Start
------------------------------------
Step: Stretch | Increment: 1, Iterations: 4
      Simulation Time: 0.02 s  of  2.0 s

Step: Stretch | Increment: 2, Iterations: 4
      Simulation Time: 0.04 s  of  2.0 s

Step: Stretch | Increment: 3, Iterations: 4
      Simulation Time: 0.06 s  of  2.0 s

Step: Stretch | Increment: 4, Iterations: 4
      Simulation Time: 0.08 s  of  2.0 s
```

```
Step: Stretch | Increment: 5, Iterations: 4
      Simulation Time: 0.1 s  of  2.0 s

Step: Stretch | Increment: 6, Iterations: 4
      Simulation Time: 0.12 s  of  2.0 s

Step: Stretch | Increment: 7, Iterations: 4
      Simulation Time: 0.14 s  of  2.0 s

Step: Stretch | Increment: 8, Iterations: 4
      Simulation Time: 0.16 s  of  2.0 s

Step: Stretch | Increment: 9, Iterations: 4
      Simulation Time: 0.18 s  of  2.0 s

Step: Stretch | Increment: 10, Iterations: 4
      Simulation Time: 0.2 s  of  2.0 s

Step: Stretch | Increment: 11, Iterations: 4
      Simulation Time: 0.22 s  of  2.0 s

Step: Stretch | Increment: 12, Iterations: 4
      Simulation Time: 0.24 s  of  2.0 s

Step: Stretch | Increment: 13, Iterations: 4
      Simulation Time: 0.26 s  of  2.0 s

Step: Stretch | Increment: 14, Iterations: 4
      Simulation Time: 0.28 s  of  2.0 s

Step: Stretch | Increment: 15, Iterations: 4
      Simulation Time: 0.3 s  of  2.0 s

Step: Stretch | Increment: 16, Iterations: 4
      Simulation Time: 0.32 s  of  2.0 s

Step: Stretch | Increment: 17, Iterations: 4
      Simulation Time: 0.34 s  of  2.0 s

Step: Stretch | Increment: 18, Iterations: 4
      Simulation Time: 0.36 s  of  2.0 s

Step: Stretch | Increment: 19, Iterations: 4
      Simulation Time: 0.38 s  of  2.0 s

Step: Stretch | Increment: 20, Iterations: 4
      Simulation Time: 0.4 s  of  2.0 s
```

```
Step: Stretch | Increment: 21, Iterations: 4
      Simulation Time: 0.42 s  of  2.0 s

Step: Stretch | Increment: 22, Iterations: 4
      Simulation Time: 0.44 s  of  2.0 s

Step: Stretch | Increment: 23, Iterations: 4
      Simulation Time: 0.46 s  of  2.0 s

Step: Stretch | Increment: 24, Iterations: 4
      Simulation Time: 0.48 s  of  2.0 s

Step: Stretch | Increment: 25, Iterations: 4
      Simulation Time: 0.5 s  of  2.0 s

Step: Stretch | Increment: 26, Iterations: 4
      Simulation Time: 0.52 s  of  2.0 s

Step: Stretch | Increment: 27, Iterations: 4
      Simulation Time: 0.54 s  of  2.0 s

Step: Stretch | Increment: 28, Iterations: 4
      Simulation Time: 0.56 s  of  2.0 s

Step: Stretch | Increment: 29, Iterations: 4
      Simulation Time: 0.58 s  of  2.0 s

Step: Stretch | Increment: 30, Iterations: 4
      Simulation Time: 0.6 s  of  2.0 s

Step: Stretch | Increment: 31, Iterations: 4
      Simulation Time: 0.62 s  of  2.0 s

Step: Stretch | Increment: 32, Iterations: 4
      Simulation Time: 0.64 s  of  2.0 s

Step: Stretch | Increment: 33, Iterations: 4
      Simulation Time: 0.66 s  of  2.0 s

Step: Stretch | Increment: 34, Iterations: 4
      Simulation Time: 0.68 s  of  2.0 s

Step: Stretch | Increment: 35, Iterations: 4
      Simulation Time: 0.7 s  of  2.0 s

Step: Stretch | Increment: 36, Iterations: 4
      Simulation Time: 0.72 s  of  2.0 s
```

```
Step: Stretch | Increment: 37, Iterations: 4
      Simulation Time: 0.74 s  of  2.0 s

Step: Stretch | Increment: 38, Iterations: 4
      Simulation Time: 0.76 s  of  2.0 s

Step: Stretch | Increment: 39, Iterations: 4
      Simulation Time: 0.78 s  of  2.0 s

Step: Stretch | Increment: 40, Iterations: 4
      Simulation Time: 0.8 s  of  2.0 s

Step: Stretch | Increment: 41, Iterations: 4
      Simulation Time: 0.82 s  of  2.0 s

Step: Stretch | Increment: 42, Iterations: 4
      Simulation Time: 0.84 s  of  2.0 s

Step: Stretch | Increment: 43, Iterations: 4
      Simulation Time: 0.86 s  of  2.0 s

Step: Stretch | Increment: 44, Iterations: 4
      Simulation Time: 0.88 s  of  2.0 s

Step: Stretch | Increment: 45, Iterations: 4
      Simulation Time: 0.9 s  of  2.0 s

Step: Stretch | Increment: 46, Iterations: 4
      Simulation Time: 0.92 s  of  2.0 s

Step: Stretch | Increment: 47, Iterations: 4
      Simulation Time: 0.94 s  of  2.0 s

Step: Stretch | Increment: 48, Iterations: 4
      Simulation Time: 0.96 s  of  2.0 s

Step: Stretch | Increment: 49, Iterations: 4
      Simulation Time: 0.98 s  of  2.0 s

Step: Stretch | Increment: 50, Iterations: 4
      Simulation Time: 1.0 s  of  2.0 s

Step: Stretch | Increment: 51, Iterations: 4
      Simulation Time: 1.02 s  of  2.0 s

Step: Stretch | Increment: 52, Iterations: 4
      Simulation Time: 1.04 s  of  2.0 s
```

```
Step: Stretch | Increment: 53, Iterations: 4
      Simulation Time: 1.06 s  of  2.0 s

Step: Stretch | Increment: 54, Iterations: 4
      Simulation Time: 1.08 s  of  2.0 s

Step: Stretch | Increment: 55, Iterations: 4
      Simulation Time: 1.1 s  of  2.0 s

Step: Stretch | Increment: 56, Iterations: 4
      Simulation Time: 1.12 s  of  2.0 s

Step: Stretch | Increment: 57, Iterations: 4
      Simulation Time: 1.14 s  of  2.0 s

Step: Stretch | Increment: 58, Iterations: 4
      Simulation Time: 1.16 s  of  2.0 s

Step: Stretch | Increment: 59, Iterations: 4
      Simulation Time: 1.18 s  of  2.0 s

Step: Stretch | Increment: 60, Iterations: 4
      Simulation Time: 1.2 s  of  2.0 s

Step: Stretch | Increment: 61, Iterations: 4
      Simulation Time: 1.22 s  of  2.0 s

Step: Stretch | Increment: 62, Iterations: 4
      Simulation Time: 1.24 s  of  2.0 s

Step: Stretch | Increment: 63, Iterations: 4
      Simulation Time: 1.26 s  of  2.0 s

Step: Stretch | Increment: 64, Iterations: 4
      Simulation Time: 1.28 s  of  2.0 s

Step: Stretch | Increment: 65, Iterations: 4
      Simulation Time: 1.3 s  of  2.0 s

Step: Stretch | Increment: 66, Iterations: 4
      Simulation Time: 1.32 s  of  2.0 s

Step: Stretch | Increment: 67, Iterations: 4
      Simulation Time: 1.34 s  of  2.0 s

Step: Stretch | Increment: 68, Iterations: 4
      Simulation Time: 1.36 s  of  2.0 s
```

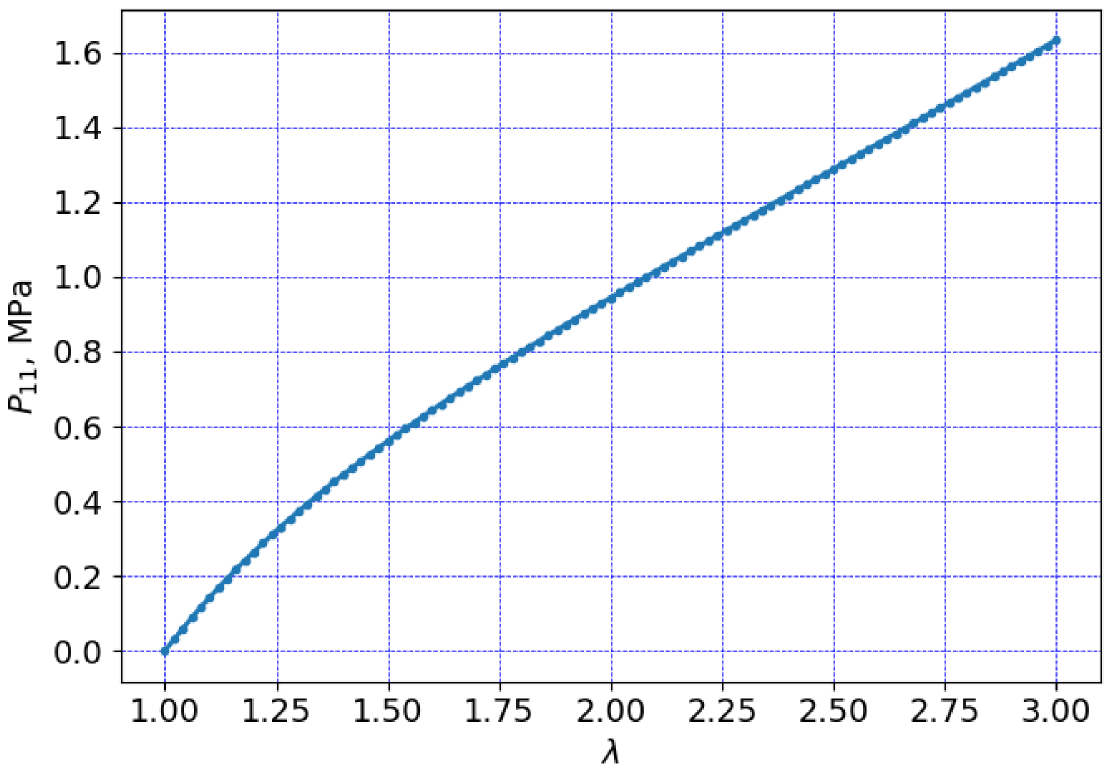
```
Step: Stretch | Increment: 69, Iterations: 4
      Simulation Time: 1.38 s  of  2.0 s

Step: Stretch | Increment: 70, Iterations: 4
      Simulation Time: 1.4 s  of  2.0 s

Step: Stretch | Increment: 71, Iterations: 4
      Simulation Time: 1.42 s  of  2.0 s

Step: Stretch | Increment: 72, Iterations: 4
      Simulation Time: 1.44 s  of  2.0 s

Step: Stretch | Increment: 73, Iterations: 4
      Simulation Time: 1.46 s  of  2.0 s

Step: Stretch | Increment: 74, Iterations: 4
      Simulation Time: 1.48 s  of  2.0 s

Step: Stretch | Increment: 75, Iterations: 4
      Simulation Time: 1.5 s  of  2.0 s

Step: Stretch | Increment: 76, Iterations: 4
      Simulation Time: 1.52 s  of  2.0 s

Step: Stretch | Increment: 77, Iterations: 4
      Simulation Time: 1.54 s  of  2.0 s

Step: Stretch | Increment: 78, Iterations: 4
      Simulation Time: 1.56 s  of  2.0 s

Step: Stretch | Increment: 79, Iterations: 4
      Simulation Time: 1.58 s  of  2.0 s

Step: Stretch | Increment: 80, Iterations: 4
      Simulation Time: 1.6 s  of  2.0 s

Step: Stretch | Increment: 81, Iterations: 4
      Simulation Time: 1.62 s  of  2.0 s

Step: Stretch | Increment: 82, Iterations: 4
      Simulation Time: 1.64 s  of  2.0 s

Step: Stretch | Increment: 83, Iterations: 4
      Simulation Time: 1.66 s  of  2.0 s

Step: Stretch | Increment: 84, Iterations: 4
      Simulation Time: 1.68 s  of  2.0 s
```

```
Step: Stretch | Increment: 85, Iterations: 4
      Simulation Time: 1.7 s  of  2.0 s

Step: Stretch | Increment: 86, Iterations: 4
      Simulation Time: 1.72 s  of  2.0 s

Step: Stretch | Increment: 87, Iterations: 4
      Simulation Time: 1.74 s  of  2.0 s

Step: Stretch | Increment: 88, Iterations: 4
      Simulation Time: 1.76 s  of  2.0 s

Step: Stretch | Increment: 89, Iterations: 4
      Simulation Time: 1.78 s  of  2.0 s

Step: Stretch | Increment: 90, Iterations: 4
      Simulation Time: 1.8 s  of  2.0 s

Step: Stretch | Increment: 91, Iterations: 4
      Simulation Time: 1.82 s  of  2.0 s

Step: Stretch | Increment: 92, Iterations: 4
      Simulation Time: 1.84 s  of  2.0 s

Step: Stretch | Increment: 93, Iterations: 4
      Simulation Time: 1.86 s  of  2.0 s

Step: Stretch | Increment: 94, Iterations: 4
      Simulation Time: 1.88 s  of  2.0 s

Step: Stretch | Increment: 95, Iterations: 4
      Simulation Time: 1.9 s  of  2.0 s

Step: Stretch | Increment: 96, Iterations: 4
      Simulation Time: 1.92 s  of  2.0 s

Step: Stretch | Increment: 97, Iterations: 4
      Simulation Time: 1.94 s  of  2.0 s

Step: Stretch | Increment: 98, Iterations: 4
      Simulation Time: 1.96 s  of  2.0 s

Step: Stretch | Increment: 99, Iterations: 4
      Simulation Time: 1.98 s  of  2.0 s

Step: Stretch | Increment: 100, Iterations: 4
      Simulation Time: 2.0 s  of  2.0 s
```

```
------------------------------------------
End computation
-------------------------------------------
Elapsed real time:  0:01:50.558887
-------------------------------------------
```

14 Plot results

[18]:
```
# set plot font to size 14
font = {'size'   : 14}
plt.rc('font', **font)

# Get array of default plot colors
prop_cycle = plt.rcParams['axes.prop_cycle']
colors = prop_cycle.by_key()['color']

plt.figure()
plt.plot((L0 + timeHist0)/L0, 2*timeHist1/1e3/(W0*t0), linewidth=2.0,\
         color=colors[0], marker='.')
plt.axis('tight')
plt.ylabel(r'$P_{11}$, MPa')
plt.xlabel(r'$\lambda$')
# plt.xlim([1,8])
# plt.ylim([0,8])
plt.grid(linestyle="--", linewidth=0.5, color='b')
# plt.show()

fig = plt.gcf()
fig.set_size_inches(7,5)
plt.tight_layout()
plt.savefig("results/3D_hip_fenicsX.png", dpi=600)
```


1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
P_{11}, MPa
1.00
1.25
1.50
1.75
2.00
2.25
2.50
2.75
3.00
λ