GEL02_3d_swell_A.Flowers_Comments

July 20, 2025

1 Cube swelling

- Swelling of a gel cube
- This is a three-dimensional simulation

Accompanies the book, - L. Anand, E.M. Stewart, S.A. Chester. *Introduction to coupled theories in solid mechanics.* 2025, in preparation.

2 Degrees of freedom

- Displacement: u
- pressure: p
- chemical potential: mu
- concentration: c

3 Units

- Length: mm
- Mass: kg
- Time: s
- Mass density: kg/mm^3
- Force: milliN
- Stress: kPa
- Energy: microJ
- Temperature: K
- Amount of substance: mol
- Species concentration: mol/mm^3
- Chemical potential: milliJ/mol
- Molar volume: mm^3/mol
- Species diffusivity: mm^2/s
- Gas constant: microJ/(mol K)

3.0.1 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/

4 Import modules

[21]:
```
# 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, exp, eq, cos, sin,␣
 ↪acos, ge, le, outer, tanh,\
                 cosh, atan, atan2)

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

5 Define geometry

[ ]:
```
# Create mesh
L0 = 2.5 # Edge length of box, mm
domain = mesh.create_box(MPI.COMM_WORLD, [[0.0, 0.0, 0.0], [L0, L0, L0]],\
                         [5, 5, 5], mesh.CellType.hexahedron)

# Note that the simulation results shown in the textbook use a 10 x 10 x 10␣
↪mesh.
# here, we use a coarser 5 x 5 x 5 mesh to get a similar result with␣
↪significantly less computation time.

x = ufl.SpatialCoordinate(domain)
```

Identify boundaries of the domain

[23]:
```
# Identify the planar boundaries of the  box mesh
def xBot(x):
    return np.isclose(x[0], 0)
def xTop(x):
    return np.isclose(x[0], L0)
def yBot(x):
    return np.isclose(x[1], 0)
def yTop(x):
    return np.isclose(x[1], L0)
def zBot(x):
    return np.isclose(x[2], 0)
def zTop(x):
    return  np.isclose(x[2], L0)

# Mark the sub-domains
boundaries = [(1,xBot),(2,xTop),(3,yBot),(4,yTop),(5,zBot),(6,zTop)]

# build collections of facets on each subdomain and mark them appropriately.
facet_indices, facet_markers = [], [] # initalize empty collections of indices␣
↪and markers.
```

```
fdim = domain.topology.dim - 1 # geometric dimension of the facet (mesh␣
 ↪dimension - 1)
for (marker, locator) in boundaries:
    facets = mesh.locate_entities(domain, fdim, locator) # an array of all the␣
 ↪facets in a
                                                          # given subdomain␣
 ↪("locator")
    facet_indices.append(facets)                          # add these facets to␣
 ↪the collection.
    facet_markers.append(np.full_like(facets, marker))    # mark them with the␣
 ↪appropriate index.

# Format the facet indices and markers as required for use in dolfinx.
facet_indices = np.hstack(facet_indices).astype(np.int32)
facet_markers = np.hstack(facet_markers).astype(np.int32)
sorted_facets = np.argsort(facet_indices)
#
# Add these marked facets as "mesh tags" for later use in BCs.
facet_tags = mesh.meshtags(domain, fdim, facet_indices[sorted_facets],␣
 ↪facet_markers[sorted_facets])
```

Print out the unique facet index numbers

[24]:
```
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 numbering:
# boundaries = [(1,xBot),(2,xTop),(3,yBot),(4,yTop),(5,zBot),(6,zTop)]
```

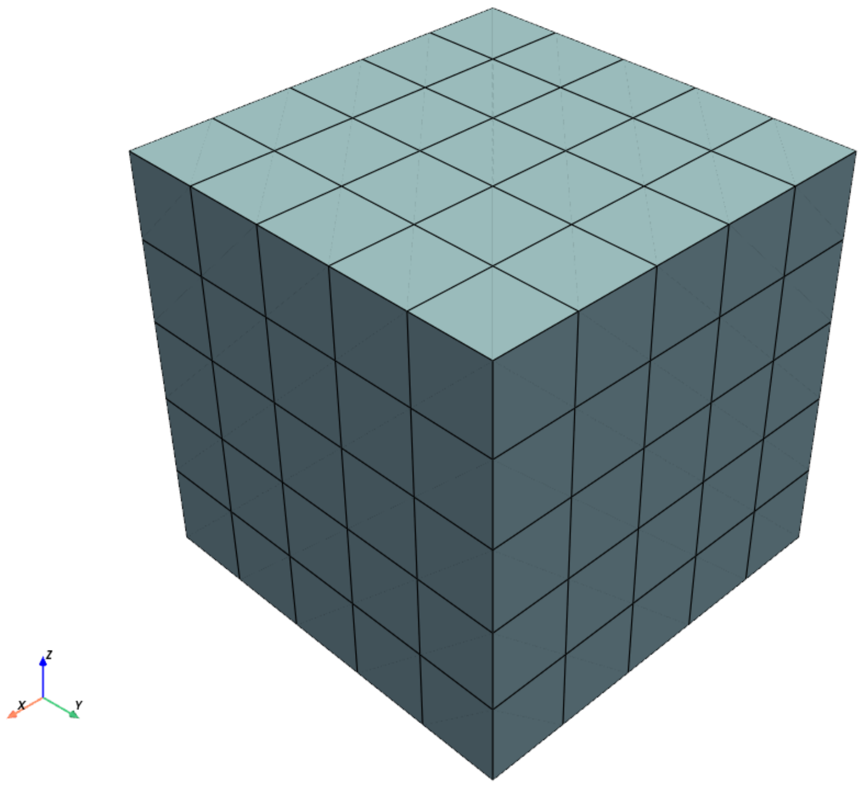
```
[1 2 3 4 5 6]
```

Visualize reference configuration

[25]:
```
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.
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=0.25)

# Add colored 2D surfaces for the named surfaces
xBot_surf = pyvista.UnstructuredGrid(*vtk_mesh(domain, domain.topology.
 ↪dim-1,facet_tags.indices[facet_tags.values==1]) )
yBot_surf = pyvista.UnstructuredGrid(*vtk_mesh(domain, domain.topology.
 ↪dim-1,facet_tags.indices[facet_tags.values==3]) )
zBot_surf = pyvista.UnstructuredGrid(*vtk_mesh(domain, domain.topology.
 ↪dim-1,facet_tags.indices[facet_tags.values==5]) )
#
actor  = plotter.add_mesh(xBot_surf, show_edges=True,color="yellow") # xBot␣
 ↪face is blue
actor2 = plotter.add_mesh(yBot_surf, show_edges=True,color="red")    # yBot is␣
 ↪red
actor3 = plotter.add_mesh(zBot_surf, show_edges=True,color="blue")   # zBot is␣
 ↪green

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

plotter.screenshot("results/cube_mesh.png")

from IPython.display import Image
Image(filename='results/cube_mesh.png')

# #Use the following  commands for a  zoom-able  view
# if not pyvista.OFF_SCREEN:
#     plotter.show()
# else:
#     plotter.screenshot("cube_mesh.png")
```

[25]:



5.1 Define boundary and volume integration measure

[26]:
```
# 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)
n = ufl.FacetNormal(domain)
```

6 Material parameters

[27]:
```
# Set the locking stretch to a large number to model a Neo-Hookean material
#
Gshear_0= Constant(domain,PETSc.ScalarType(1000.0))         # Shear modulus, kPa
lambdaL = Constant(domain,PETSc.ScalarType(100))            # Locking stretch,␣
 ↪Neo_hookean material
Kbulk   = Constant(domain,PETSc.ScalarType(1000*Gshear_0))  # Bulk modulus, kPa
Omega   = Constant(domain,PETSc.ScalarType(1.00e5))         # Molar volume of␣
 ↪fluid
D       = Constant(domain,PETSc.ScalarType(5.00e-3))        # Diffusivity
chi     = Constant(domain,PETSc.ScalarType(0.1))            # Flory-Huggins␣
 ↪mixing parameter
theta0  = Constant(domain,PETSc.ScalarType(298) )           # Reference␣
 ↪temperature
R_gas   = Constant(domain,PETSc.ScalarType(8.3145e6))       # Gas constant
RT      = Constant(domain,PETSc.ScalarType(8.3145e6*theta0))
#
phi0    = Constant(domain,PETSc.ScalarType(0.999))                # Initial␣
 ↪polymer volume fraction
mu0     = Constant(domain,PETSc.ScalarType(ln(1.0-phi0) + phi0 )) # Initial␣
 ↪chemical potential
c0      = Constant(domain,PETSc.ScalarType((1/phi0) - 1))         # Initial␣
 ↪concentration
```

7 Simulation time-control related parameters

[28]:
```
t    = 0.0         # initialization of time
Ttot = 3600*3      # total simulation time
ttd  = 400         # Decay time constant
dt   = 200         # Fixed step size

# Boundary condition expression for increasing  the chemical potential
#
def muRamp(t):
    return mu0*exp(-t/ttd)
```

8 Function spaces

[29]:
```
# 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, chemical␣
 ↪potential and  species concentration
#
```

```
TH = mixed_element([U2, P1, P1, 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, mu, c = split(w)  # displacement u, pressure p, chemical potential mu,␣
 ↪and concentration c

# A copy of functions to store values in the previous step for time-stepping
w_old = Function(ME)
u_old,  p_old, mu_old, c_old = split(w_old)

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

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

9 Initial conditions

- The initial conditions for **u** and p are zero everywhere.
- These are imposed automatically, since we have not specified any non-zero initial conditions.
- We do, however, need to impose the uniform initial conditions for $\mu = \mu_0$ and $\hat{c} = \hat{c}_0$ which correspond to $\phi_0 = 0.999$. This is done below.

[30]:
```
# Assign initial  normalized chemical potential  mu0 to the domain
w.sub(2).interpolate(lambda x: np.full((x.shape[1],),  mu0))
w_old.sub(2).interpolate(lambda x: np.full((x.shape[1],), mu0))

# Assign initial  value of normalized concentration  c0 to the domain
w.sub(3).interpolate(lambda x: np.full((x.shape[1],),  c0))
w_old.sub(3).interpolate(lambda x: np.full((x.shape[1],), c0))
```

10 Subroutines for kinematics and constitutive equations

[31]:
```
#----------------------------------------------------
# Deformation gradient
#----------------------------------------------------
def F_calc(u):

    Id = Identity(3)          # 3D Identity tensor

    F = Id + grad(u)            # 3D Deformation gradient

    return F

```

```python
#---------------------------------------------------
# Effective stretch lambdaBar
#---------------------------------------------------
def lambdaBar_calc(u):
    F = F_calc(u)
    C = F.T*F
    I1 = tr(C)
    lambdaBar = sqrt(I1/3.0)
    return lambdaBar

#---------------------------------------------------
# Calculate zeta
#---------------------------------------------------
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

#---------------------------------------------------
# Calculate zeta0
#---------------------------------------------------
def zeta0_calc():
    # Use Pade approximation of Langevin inverse (A. Cohen, 1991)
    z    = 1/lambdaL
    z    = conditional(gt(z,0.95), 0.95, z) # Keep from blowing up
    beta0 = z*(3.0 - z**2.0)/(1.0 - z**2.0)
    zeta0 = (lambdaL/3)*beta0
    return zeta0

#---------------------------------------------------
# Subroutine for calculating the elastic jacobian Je
#---------------------------------------------------
def Je_calc(u,c):
    F = F_calc(u)
    detF = det(F)
    #
    detFs = 1.0 + c          # = Js
    Je    = (detF/detFs)     # = Je
    return   Je

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

```
def Piola_calc(u,p):
    F     = F_calc(u)
    zeta  = zeta_calc(u)
    zeta0 = zeta0_calc()
    Piola = (zeta*F - zeta0*inv(F.T) ) - J*p*inv(F.T)/Gshear_0
    return Piola

#---------------------------------------------------------------
# Subroutine for calculating the normalized species flux
#---------------------------------------------------------------
def Flux_calc(u, mu, c):
    F = F_calc(u)
    #
    Cinv = inv(F.T*F)
    #
    Mob = (D*c)/(Omega*RT)*Cinv
    #
    Jmat = - RT* Mob * grad(mu)
    return Jmat
```

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

#----------------------------------------------------
# Deformation gradient
#----------------------------------------------------
def F_calc(u):
    Id = Identity(3)          # 3D Identity tensor
##Identity tensor; 3D matrix
    F = Id + grad(u)            # 3D Deformation gradient
##Identity tensor, displacement gradient, and total deformation gradient
    return F
##Gives tensor F; used to compute volume change and Cauchy-Green tensor for␣
 ↪strain measures

#----------------------------------------------------
# Effective stretch lambdaBar
#----------------------------------------------------
def lambdaBar_calc(u):
    F = F_calc(u)
##Helper function compute deformation gradient
    C = F.T*F
##Caughy-Green tensor computed; C is symmetric and captures stretch without␣
 ↪rotation (common in hyperelasticity for strain energy density functions)
    I1 = tr(C)
##Computes first invariant of C; this is the sum of squares of principal␣
 ↪stretches
    lambdaBar = sqrt(I1/3.0)
```

```
##Gives average stretch normalized over 3 dimensions (Frobenius-norm)
    return lambdaBar
##Gives scalar used for energy model depending on stretch magnitude

#---------------------------------------------------
# Calculate zeta
#---------------------------------------------------
def zeta_calc(u):
    lambdaBar = lambdaBar_calc(u)
##Computes average stretch invariant; measure of isotropic deformation
    # Use Pade approximation of Langevin inverse
    z    = lambdaBar/lambdaL
##Locking-stretch;f max. average chain extension before infinite stiffness
    z    = conditional(gt(z,0.95), 0.95, z) # Keep simulation from blowing up
##Used due to FEniCS to keep computation differentiable and stable
    beta = z*(3.0 - z**2.0)/(1.0 - z**2.0)
##Pade approx. to the inverse Langevin function; inverse with functions for␣
 ↪efficiency and stability
    zeta = (lambdaL/(3*lambdaBar))*beta
##Gives correct chain force multiplier used in Gent/Arruda-Boyce energy density␣
 ↪functions
    return zeta
##Gives non-linear stretch function; scales stress/energy terms to prevent␣
 ↪over-stretching beyond material phycial limits

#---------------------------------------------------
# Calculate zeta0
#---------------------------------------------------
def zeta0_calc():
    # Use Pade approximation of Langevin inverse (A. Cohen, 1991)
    z    = 1/lambdaL
##Normalized reference stretch
    z    = conditional(gt(z,0.95), 0.95, z) # Keep from blowing up
##Used due to FEniCS to keep computation differentiable and stable
    beta0 = z*(3.0 - z**2.0)/(1.0 - z**2.0)
##Pade approx. of inverse Langevin
    zeta0 = (lambdaL/3)*beta0
##Full expression for inverse Langevin stretch; matches structure of stretched␣
 ↪form
    return zeta0
##Energy normalization and stress-free reference

#---------------------------------------------------
# Subroutine for calculating the elastic jacobian Je
#---------------------------------------------------
def Je_calc(u,c):
    F = F_calc(u)
```

```
##Computes deformation gradient tensor F in displacement field u
    detF = det(F)
##Computes total volume ratio; how much element has expanded / compressed␣
 ↪compared to reference volume
    detFs = 1.0 + c          # = Js
##Define swelling volume ratio
##c= swlling-related variable; 1+c= volume expansion due to chemical absorption
    Je    = (detF/detFs)     # = Je
##Computes elastic volume change; mechanical deformation after accounting for␣
 ↪volume change from swelling
    return   Je
##Gives mechanical output of deformation
##Used for elastic energy density functions, stress calculations, and␣
 ↪thermodynamic coupling with chemical potentials
#---------------------------------------------
# Subroutine for calculating the Piola  stress
#---------------------------------------------
def Piola_calc(u,p):
    F     = F_calc(u)
##Computes deformation gradient tensor F in displacement field u
    zeta  = zeta_calc(u)
##Computes scalar energy quantity (either osmotic pressure of chemical␣
 ↪potential energy)
    zeta0 = zeta0_calc()
##Reference value, in undeformed or initial state
    Piola = (zeta*F - zeta0*inv(F.T) ) - J*p*inv(F.T)/Gshear_0
##Enforce incompressibility (volumetric constraints) common in gel mechanics or␣
 ↪incompressible elasticity
    return Piola
##Gives full, first Piola-Kirchhoff stress tensor P

#--------------------------------------------------------------
# Subroutine for calculating the normalized species flux
#--------------------------------------------------------------
def Flux_calc(u, mu, c):
    F = F_calc(u)
##Computes deformation gradient tensor F in displacement field u
    Cinv = inv(F.T*F)
##Computes inverse of right Cauchy-Green deformation tensor
##Geometric nonlinearity due to large deformation; helps transform diffusivity /
 ↪ mobility from reference configuration to the current
    Mob = (D*c)/(Omega*RT)*Cinv
##Defines effective mobility tensor M; modifies diffusion based on deformation
    Jmat = - RT* Mob * grad(mu)
##Computes mass flux vector from chemical potential gradient
    return Jmat
```

```
##Gives reference configuration material flux vector
```

11 Evaluate kinematics and constitutive relations

[32]:
```
# Kinematics
F = F_calc(u)
J = det(F)  # Total volumetric jacobian
#
lambdaBar = lambdaBar_calc(u)
#
# Elastic volumetric Jacobian
Je     = Je_calc(u,c)
Je_old = Je_calc(u_old,c_old)

#  Normalized Piola stress
Piola = Piola_calc(u, p)

#  Normalized species  flux
Jmat = Flux_calc(u, mu, c)
```

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

# Kinematics
F = F_axi_calc(u)
##Computes deformation gradient tensor F from displacement field in␣
 ↪axisymmetric coordinates; captures stretch and shear from displacement
J = det(F)  # Total volumetric jacobian
##Total Jacobian determinant measuring volume change due to deformation; ratio␣
 ↪of current to referene volume
##J couples with pressure in incompressibility and appears in transport models␣
 ↪(such as osmosis) as coupling between defomration and chemical concetration

lambdaBar = lambdaBar_calc(u)
# Elastic volumetric Jacobian
Je     = Je_calc(u,c)
##Elastic Jacobian at present time; mechanical volume change, excluding swelling
Je_old = Je_calc(u_old,c_old)
##Elastic Jacobian at previous time; shows evolution

#  Normalized Piola stress
Piola = Piola_calc(u, p)
##Computes total stress in material using displacement and pressure field; then␣
 ↪used in momentum balace, determining response under deformation and swelling

#  Normalized species  flux
Jmat = Flux_calc(u, mu, c)
```

```
##Computes species flux vector; considers deformation, chemical potential, and␣
↪concentration. Pertinent to chemical transport in deforming materials␣
↪(swelling gels) where mechanics and diffusion are coupled
##Flux of mobile species (solvent and ions); how fast they are moving through␣
↪material, driven by chemical potential gradients and modified by mechanical␣
↪deformation
##C evolves through mass conservation (diffusion)
```

12 Weak forms

[33]:
```
# Residuals:
# Res_0: Balance of forces (test fxn: u)
# Res_1: Pressure variable (test fxn: p)
# Res_2: Balance of mass   (test fxn: mu)
# Res_3: Auxiliary variable (test fxn: c)

# Time step field, constant within body
dk = Constant(domain, PETSc.ScalarType(dt))

# The weak form for the equilibrium equation
Res_0 = inner(Piola, grad(u_test) )*dx

# The weak form for the auxiliary pressure variable definition
Res_1 = dot((p*Je/Kbulk + ln(Je)) , p_test)*dx

# The weak form for the mass balance of solvent
Res_2 = dot((c - c_old)/dk, mu_test)*dx \
        -  Omega*dot(Jmat , grad(mu_test) )*dx

# The weak form for the concentration
fac = 1/(1+c)
fac1 =  mu - ( ln(1.0-fac)+ fac + chi*fac*fac)
fac2 = - (Omega*Je/RT)*p
fac3 = - (1./2.) * (Omega/(Kbulk*RT)) * ((p*Je)**2.0)
fac4 = fac1 + fac2 + fac3
#
Res_3 = dot(fac4, c_test)*dx

# Total weak form
Res = Res_0 + Res_1 + Res_2 + Res_3

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

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

```
# Residuals:
# Res_0: Balance of forces (test fxn: u)
# Res_1: Pressure variable (test fxn: p)
# Res_2: Balance of mass   (test fxn: mu)
# Res_3: Auxiliary variable (test fxn: c)

# Time step field, constant within body
dk = Constant(domain, PETSc.ScalarType(dt))
##Defines constant scalar due to FEniCS (time step size)

# The weak form for the equilibrium equation
Res_0 = inner(Piola, grad(u_test) )*dx
##Solid mechanics for nonlinear elasticity (swelling gels); gradient of test␣
 ↪function (tensor representing virtual strain)
##Residual form for momentum balance equation

# The weak form for the auxiliary pressure variable definition
Res_1 = dot((p*Je/Kbulk + ln(Je)) , p_test)*dx
##Standard 2D/3D weak form; axisymmetry not accounted for
##Adds term to pressure residual enforcing compressibility / volume constraint␣
 ↪via pressure, elastic volume change, and bulk modulus
##p=pressure field (unknown) acting as Lagrange multiplier; apperas when␣
 ↪pressure is solved as a field, and is not given (balancing swelling and␣
 ↪elasticity)


# The weak form for the mass balance of solvent
Res_2 = dot((c - c_old)/dk, mu_test)*dx \
##Weak form of transient accumulation of mobile species
        -  Omega*dot(Jmat , grad(mu_test) )*dx
####Weak form of divergence of flux; drives species redistribution due to␣
 ↪chemical gradients and deformation
##Overall enforces chemical species conservation; change in concentration =␣
 ↪inflow/outflow due to chemical potential gradients accounting for␣
 ↪deformation of medium and axisymmetric geometry
##Axisymmetry not acccounted for in both terms per 3D

# The weak form for the concentration
fac = 1/(1+c)
##Computes volume fraction of polymer
fac1 =  mu - ( ln(1.0-fac)+ fac + chi*fac*fac)
##Encodes Flory-Huggins chemical potential
fac2 = - (Omega*Je/RT)*p
##Accounts for mechanical pressure contribution due to chemical potential
fac3 = - (1./2.) * (Omega/(Kbulk*RT)) * ((p*Je)**2.0)
##Compressibility correction due to bulk deformation from free energy stored
fac4 = fac1 + fac2 + fac3
```

```
##Residual contributions; difference between computed chemical potential and␣
 ↪its theoretical value based on thermodynamics, mechanics, and compressibility

Res_3 = dot(fac4, c_test)*dx
##Residual term in variational form for solving chemical␣
 ↪potential-concentration coupling in swelling gel model
##Thermodynamic consistency condition; ensures chemical potential used in flux␣
 ↪equation is thermodynamically correct, given concentration and mechanical␣
 ↪state
##Nonlinear coupled system, with momentum balance, mass transport, and␣
 ↪thermodynamic closure

# Total weak form
Res = Res_0 + Res_1 + Res_2 + Res_3
##Total residual of FE; combining mechanics, thermodynamics, and mass transport␣
 ↪in a swelling gel model
##Formula enabling coupled multiphysics behavior; characteristics in swelling␣
 ↪gels, ionic polymers, hydrogel actuators

# Automatic differentiation tangent:
a = derivative(Res, w, dw)
##Solves nonlinear variational problem in FEniCS, computing Jacobian matrix
```

13 Set-up output files

[34]:
```
# results file name
results_name = "gel_3d_swell"

# Function space for projection of results
U1 = element("DG", domain.basix_cell(), 1, shape=(3,))  # For displacement
P0 = element("DG", domain.basix_cell(), 1)              # For  pressure,␣
 ↪chemical potential, and concentration
T1 = element("DG", domain.basix_cell(), 1, shape=(3,3)) # For stress tensor

V1 = fem.functionspace(domain, P0) # Scalar function space
V2 = fem.functionspace(domain, U1) # Vector function space
V3 = fem.functionspace(domain, T1) # Tensor function space

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

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

mu_vis = Function(V1)
```

```
mu_vis.name = "mu"

c_vis = Function(V1)
c_vis.name = "c"

# calculated fields to write to output file
phi = 1/(1+c)
phi_vis = Function(V1)
phi_vis.name = "phi"
phi_expr = Expression(phi,V1.element.interpolation_points())

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

# Mises stress
T   = Piola*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, mu_vis, c_vis, phi_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))
       mu_vis.interpolate(w.sub(2))
       c_vis.interpolate(w.sub(3))
       phi_vis.interpolate(phi_expr)
       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)
```

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

[35]:
```
# This is actually not neede here for this simulation


# # Identify point for reporting displacement and temperature at a given point
# pointForDisp = np.array([L0,L0,L0])

# 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

# # boundaries = [(1, xBot),(2,xTop),(3,yBot),(4,yTop)]
# # Compute the reaction force using the Piola stress field
# RxnForce = fem.form(2*np.pi*P22*x[0]*ds(4))
```

15 Analysis Step

[36]:
```
# Give the step a descriptive name
step = "Swell"
```

15.1 Boundary conditions

[37]:
```
# Constant for applied  chemical potential
mu_cons = Constant(domain,PETSc.ScalarType(muRamp(0)))

# Recall the sub-domains names and numbers
# boundaries = [(1,xBot),(2,xTop),(3,yBot),(4,yTop),(5,zBot),(6,zTop)]

# 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(1))
yBot_u2_dofs = fem.locate_dofs_topological(ME.sub(0).sub(1), facet_tags.dim,␣
 ↪facet_tags.find(3))
zBot_u3_dofs = fem.locate_dofs_topological(ME.sub(0).sub(2), facet_tags.dim,␣
 ↪facet_tags.find(5))
#
xTop_mu_dofs = fem.locate_dofs_topological(ME.sub(2), facet_tags.dim,␣
 ↪facet_tags.find(2))
yTop_mu_dofs = fem.locate_dofs_topological(ME.sub(2), facet_tags.dim,␣
 ↪facet_tags.find(4))
zTop_mu_dofs = fem.locate_dofs_topological(ME.sub(2), facet_tags.dim,␣
 ↪facet_tags.find(6))

# Dirichlet BCs for displacement
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(0.0, zBot_u3_dofs, ME.sub(0).sub(2))  # u3 fix - zBot
#
# Dirichlet BCs for chemical potential
bcs_4 = dirichletbc(mu_cons, xTop_mu_dofs, ME.sub(2))  # mu_cons - xBot
bcs_5 = dirichletbc(mu_cons, yTop_mu_dofs, ME.sub(2))  # mu_cons - yTop
bcs_6 = dirichletbc(mu_cons, zTop_mu_dofs, ME.sub(2))  # mu_cons - zTop

# Complete set of Dirichlet bcs
bcs = [bcs_1, bcs_2, bcs_3, bcs_4, bcs_5, bcs_6]
```

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

# Constant for applied  chemical potential
mu_cons = Constant(domain,PETSc.ScalarType(muRamp(0)))
##Defines chemically driven control parameters; simulates ramping up the␣
 ↪chemical potential at the boundary during swelling (constant field)

# Recall the sub-domains names and numbers
# boundaries = [(1,xBot),(2,xTop),(3,yBot),(4,yTop),(5,zBot),(6,zTop)]

# 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(1))
yBot_u2_dofs = fem.locate_dofs_topological(ME.sub(0).sub(1), facet_tags.dim,␣
 ↪facet_tags.find(3))
zBot_u3_dofs = fem.locate_dofs_topological(ME.sub(0).sub(2), facet_tags.dim,␣
 ↪facet_tags.find(5))
##DOFs for each displacement component of bottom surfaces

xTop_mu_dofs = fem.locate_dofs_topological(ME.sub(2), facet_tags.dim,␣
 ↪facet_tags.find(2))
yTop_mu_dofs = fem.locate_dofs_topological(ME.sub(2), facet_tags.dim,␣
 ↪facet_tags.find(4))
zTop_mu_dofs = fem.locate_dofs_topological(ME.sub(2), facet_tags.dim,␣
 ↪facet_tags.find(6))
##DOFs for chemical potential on top surfaces

# Dirichlet BCs for displacement
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(0.0, zBot_u3_dofs, ME.sub(0).sub(2))  # u3 fix - zBot
##Applies Dirichlet boundary condition; fix bottom surface of domain in all␣
 ↪spatial directions; fully cleamped base preventing rigid body motion

# Dirichlet BCs for chemical potential
bcs_4 = dirichletbc(mu_cons, xTop_mu_dofs, ME.sub(2))  # mu_cons - xBot
bcs_5 = dirichletbc(mu_cons, yTop_mu_dofs, ME.sub(2))  # mu_cons - yTop
bcs_6 = dirichletbc(mu_cons, zTop_mu_dofs, ME.sub(2))  # mu_cons - zTop
##Given value of chemical potential , modeling contact with external␣
 ↪environment (i.e. fluid in swelling gel)
##Represent a controlled environment dictating solvent movement in/out of gel;␣
 ↪essential for stimulating swelling in external field

# Complete set of Dirichlet bcs
bcs = [bcs_1, bcs_2, bcs_3, bcs_4, bcs_5, bcs_6]
##Implements boundary conditions to solver
```

15.2 Define the nonlinear variational problem

[38]:
```
# 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"
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()
```

16 Initialize arrays for storing output history

[39]:
```
# # Arrays for storing output history
# totSteps = 100000
# timeHist0 = np.zeros(shape=[totSteps])
# timeHist1 = np.zeros(shape=[totSteps])
# timeHist2 = np.zeros(shape=[totSteps])
# timeHist3 = np.zeros(shape=[totSteps])
# #
# timeHist3[0] = mu0 # Initialize the chemical potential

# Initialize a counter for reporting data
ii=0

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

16.1 Start calculation loop

[40]:
```
# Print  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
    mu_cons.value = float(muRamp(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

    # 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: Swell | Increment: 1, Iterations: 7
      Simulation Time: 200.0 s  of  10800 s
```

```
Step: Swell | Increment: 2, Iterations: 6
      Simulation Time: 400.0 s  of  10800 s

Step: Swell | Increment: 3, Iterations: 6
      Simulation Time: 600.0 s  of  10800 s

Step: Swell | Increment: 4, Iterations: 6
      Simulation Time: 800.0 s  of  10800 s

Step: Swell | Increment: 5, Iterations: 6
      Simulation Time: 1000.0 s  of  10800 s

Step: Swell | Increment: 6, Iterations: 6
      Simulation Time: 1200.0 s  of  10800 s

Step: Swell | Increment: 7, Iterations: 5
      Simulation Time: 1400.0 s  of  10800 s

Step: Swell | Increment: 8, Iterations: 5
      Simulation Time: 1600.0 s  of  10800 s

Step: Swell | Increment: 9, Iterations: 5
      Simulation Time: 1800.0 s  of  10800 s

Step: Swell | Increment: 10, Iterations: 5
      Simulation Time: 2000.0 s  of  10800 s

Step: Swell | Increment: 11, Iterations: 5
      Simulation Time: 2200.0 s  of  10800 s

Step: Swell | Increment: 12, Iterations: 5
      Simulation Time: 2400.0 s  of  10800 s

Step: Swell | Increment: 13, Iterations: 5
      Simulation Time: 2600.0 s  of  10800 s

Step: Swell | Increment: 14, Iterations: 5
      Simulation Time: 2800.0 s  of  10800 s

Step: Swell | Increment: 15, Iterations: 4
      Simulation Time: 3000.0 s  of  10800 s

Step: Swell | Increment: 16, Iterations: 4
      Simulation Time: 3200.0 s  of  10800 s

Step: Swell | Increment: 17, Iterations: 4
      Simulation Time: 3400.0 s  of  10800 s
```

```
Step: Swell | Increment: 18, Iterations: 4
      Simulation Time: 3600.0 s  of  10800 s

Step: Swell | Increment: 19, Iterations: 4
      Simulation Time: 3800.0 s  of  10800 s

Step: Swell | Increment: 20, Iterations: 4
      Simulation Time: 4000.0 s  of  10800 s

Step: Swell | Increment: 21, Iterations: 4
      Simulation Time: 4200.0 s  of  10800 s

Step: Swell | Increment: 22, Iterations: 4
      Simulation Time: 4400.0 s  of  10800 s

Step: Swell | Increment: 23, Iterations: 4
      Simulation Time: 4600.0 s  of  10800 s

Step: Swell | Increment: 24, Iterations: 4
      Simulation Time: 4800.0 s  of  10800 s

Step: Swell | Increment: 25, Iterations: 4
      Simulation Time: 5000.0 s  of  10800 s

Step: Swell | Increment: 26, Iterations: 4
      Simulation Time: 5200.0 s  of  10800 s

Step: Swell | Increment: 27, Iterations: 4
      Simulation Time: 5400.0 s  of  10800 s

Step: Swell | Increment: 28, Iterations: 4
      Simulation Time: 5600.0 s  of  10800 s

Step: Swell | Increment: 29, Iterations: 4
      Simulation Time: 5800.0 s  of  10800 s

Step: Swell | Increment: 30, Iterations: 4
      Simulation Time: 6000.0 s  of  10800 s

Step: Swell | Increment: 31, Iterations: 4
      Simulation Time: 6200.0 s  of  10800 s

Step: Swell | Increment: 32, Iterations: 4
      Simulation Time: 6400.0 s  of  10800 s

Step: Swell | Increment: 33, Iterations: 4
      Simulation Time: 6600.0 s  of  10800 s
```

```
Step: Swell | Increment: 34, Iterations: 4
      Simulation Time: 6800.0 s  of  10800 s

Step: Swell | Increment: 35, Iterations: 4
      Simulation Time: 7000.0 s  of  10800 s

Step: Swell | Increment: 36, Iterations: 4
      Simulation Time: 7200.0 s  of  10800 s

Step: Swell | Increment: 37, Iterations: 4
      Simulation Time: 7400.0 s  of  10800 s

Step: Swell | Increment: 38, Iterations: 4
      Simulation Time: 7600.0 s  of  10800 s

Step: Swell | Increment: 39, Iterations: 4
      Simulation Time: 7800.0 s  of  10800 s

Step: Swell | Increment: 40, Iterations: 4
      Simulation Time: 8000.0 s  of  10800 s

Step: Swell | Increment: 41, Iterations: 4
      Simulation Time: 8200.0 s  of  10800 s

Step: Swell | Increment: 42, Iterations: 4
      Simulation Time: 8400.0 s  of  10800 s

Step: Swell | Increment: 43, Iterations: 4
      Simulation Time: 8600.0 s  of  10800 s

Step: Swell | Increment: 44, Iterations: 4
      Simulation Time: 8800.0 s  of  10800 s

Step: Swell | Increment: 45, Iterations: 4
      Simulation Time: 9000.0 s  of  10800 s

Step: Swell | Increment: 46, Iterations: 4
      Simulation Time: 9200.0 s  of  10800 s

Step: Swell | Increment: 47, Iterations: 4
      Simulation Time: 9400.0 s  of  10800 s

Step: Swell | Increment: 48, Iterations: 4
      Simulation Time: 9600.0 s  of  10800 s

Step: Swell | Increment: 49, Iterations: 4
      Simulation Time: 9800.0 s  of  10800 s
```

```
Step: Swell | Increment: 50, Iterations: 4
      Simulation Time: 10000.0 s  of  10800 s

Step: Swell | Increment: 51, Iterations: 4
      Simulation Time: 10200.0 s  of  10800 s

Step: Swell | Increment: 52, Iterations: 4
      Simulation Time: 10400.0 s  of  10800 s

Step: Swell | Increment: 53, Iterations: 4
      Simulation Time: 10600.0 s  of  10800 s

Step: Swell | Increment: 54, Iterations: 4
      Simulation Time: 10800.0 s  of  10800 s

-----------------------------------------
End computation
------------------------------------------
Elapsed real time:  0:00:35.774578
------------------------------------------
```