User Guide: A generalized poroelastic model using FEniCS

The user guide for the model presented in "A generalized poroelastic model using FEniCS with insights into Noordbergum effect" published in *Computers & Geosciences* (Haagenson et al., 2019).

Ryan Haagenson
Civil, Environmental, and Architectural Engineering
University of Colorado Boulder
ryan.haagenson@colorado.edu

July 25th, 2019

Table of Contents

1 Introduction

This document is the user guide associated with the model presented in "A generalized poroelastic model using FEniCS with insights into the Noordbergum effect" published in *Computers & Geosciences* (Haagenson et al., 2019). The contents are not meant to replace any user guide or tutorials for the FEniCS software itself. This document is written with the assumption that the reader already has a working knowledge of the FEniCS software. For more information on FEniCS, including how to install and use the software, please refer to fenicsproject.org or *Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book* (Logg et al., 2012).

The code presented here was developed using the python module of FEniCS version 2017.2.0, which can be installed through the stable Docker images as explained on the FEniCS project webpage (Alnaes et al., 2014; Alnaes et al., 2015; Kirby, 2004; Kirby and Logg, 2006; Kirby et al., 2012; Logg and Wells, 2010). To be made compatible with other versions of FEniCS, small edits to the code's syntax may be required.

The mathematical theory and numerical approach of the model is well described in the associated journal article (Haagenson et al., 2019). This document provides a step-by-step explanation of the code's contents (i.e. what each line or section of code is for and how to manipulate it for customized use). Section 2 presents the code's contents in the order they appear in the code. Section 3 gives a simple tutorial on how to edit the general code's content to achieve a three dimensional simulation of soil heaving.

As with any numerical model, the accuracy of this model is not guaranteed for any given simulation and should be verified by the user in all cases.

For questions or concerns about the model presented here, you may contact the author at ryan.haagenson@colorado.edu. Alteration of the model's code is encouraged. Indeed, one of the main benefits of writing a model in the FEniCS framework is its flexibility to be customized for the needs of various simulations. If you would like any alterations or improvements to become part of the permanent model structure, please contact the author to discuss. Please cite the associated journal article (i.e. "A generalized poroelastic model using FEniCS with insights into the Noordbergum effect" published in *Computers & Geosciences*) for any use of the model's code, in addition to the appropriate citations of the FEniCS software and it's various constituents (Haagenson et al., 2019).

2 The Code

There are two versions of the model that the user may employ: the linear model and the nonlinear model. Both are solving a standard form of the poroelastic equations as described by the associated journal article (Haagenson et al., 2019). While the linear model employs a constant fluid density and porosity, the nonlinear model allows for a variable fluid density and employs a porosity relationship that depends on fluid pressure and volumetric strain. The generalizations provided by the nonlinearities may not always be of utmost importance, depending of the simulation's parameters.

To handle the nonlinearities, the model uses Picard iteration. This iterative method has been written using MPI commands to allow for simulations run in parallel (i.e. with multiple processors). To run the code in parallel, please refer to the regular FEniCS software documentation. The code may also be run in serial without issue.

This section describes the structure of the code General_Poroelastic_Model.py, which is intended to provide the user a general template for the poroelastic model. The file itself does not run any specific simulation. In fact, when the script is run it will return an error. Using this template, the user can customize each section to their specific needs and troubleshoot any errors encountered along the way. A tutorial on how to alter each section for a simple three-dimensional soil-heaving problem is given in Section 3.

The code is initialized with the FEniCS python module, numpy, and ufl to use the conditional workaround for the porosity relationship of the nonlinear model. The date and time of when the code began running is printed to the terminal window.

```
######### INITIALIZE CODE ##################################################

from fenics import *
import numpy as np
import ufl
from time import gmtime, strftime

ufl.algorithms.apply_derivatives.CONDITIONAL_WORKAROUND = True

if MPI.rank(mpi_comm_world()) == 0:
    print "Start date and time:"
    print strftime("%Y-%m-%d %H:%M:%S", gmtime())

############################################################################
```

Next, the user must declare which model they wish to use: the linear or nonlinear model. To do this, set either `Linear` or `Nonlinear` equal to 1. This decision should be based on the model parameters and simulation scenario. The model ensures that only one model type is chosen, and prints that model type to the terminal window.

```
######### DEFINE MODEL TYPE ##################################################

Linear = 0
Nonlinear = 1

if Linear + Nonlinear != 1:
    if MPI.rank(mpi_comm_world()) == 0:
        print "You must choose a model type."
    quit()

if Linear == 1:
    if MPI.rank(mpi_comm_world()) == 0:
        print "Running the Linear Poroelastic Model."
    weight = Constant(0.0)
    linear_flag = 1

if Nonlinear == 1:
    if MPI.rank(mpi_comm_world()) == 0:
        print "Running the Nonlinear Poroelastic Model."
    weight = Constant(1.0)
    linear_flag = 0

##############################################################################
```

Below is where the domain constants are defined. For the each individual simulation, these will be altered. Here, the domain is defined as a cube with one corner at the origin (0,0,0) and the other at (x_1, y_1, z_1) with grid spacing in the x, y, and z directions of n_x, n_y, and n_z respectively. However, these constants can be altered to describe any domain. (eg. a two dimensional rectangle or a three dimensional sphere).

```
######### DOMAIN AND SUBDOMAINS ##############################################

### Domain Constants ###
x0 = 0.0          # Minimum x
x1 = 1.0          # Maximum x
y0 = 0.0          # Minimum y
y1 = 1.0          # Maximum y
z0 = 0.0          # Minimum z
z1 = 1.0          # Maximum z
nx = 1            # Number of cells in x-direction
ny = 1            # Number of cells in y-direction
nz = 1            # Number of cells in z-direction

##############################################################################
```

After defining the domain constants, the model uses the internal meshing tool provided by FEniCS called mshr. If desired, a third party meshing tool (such as gmsh) can be used instead. Mesh files from third party meshing tools can be converted to FEniCS compatible files using the dolfin-convert tool provided with any FEniCS installation.

Here, the converted mesh file would be loaded into the model instead of generated with mshr.

```
######### MESHING ##################################################################

if MPI.rank(mpi_comm_world()) == 0:
    print ('Building mesh...')
mesh = BoxMesh(Point(x0,y0,z0),Point(x1,y1,z1),nx,ny,nz)

####################################################################################
```

The next section of code is used to define the mixed function space of the poroelastic formulation. These are described in the associated journal article, and should not be altered in most circumstances (Haagenson et al., 2019). Altering these function spaces runs the risk that the numerical formulation no longer satisfies the so-called inf-sup condition, and the model accuracy will be in jeopardy. Only alter this section when absolutely sure that the alterations are necessary and still provide a stable selection of function spaces. Function spaces for additional variables (such as fluid density and porosity) are defined here as well.

```
######### FUNCTION SPACES ##########################################################

### Field variable function spaces ###
ele_p  = FiniteElement("DG",  mesh.ufl_cell(), 0) # pressure
ele_q = FiniteElement("BDM", mesh.ufl_cell(), 1) # fluid flux
ele_us  = VectorElement("CG",  mesh.ufl_cell(), 1) # solid displacement
W = MixedElement([ele_p, ele_q, ele_us])
W = FunctionSpace(mesh, W)

### Function spaces for other variables ###
Z = FunctionSpace(mesh, "CG", 1)
V = VectorFunctionSpace(mesh, "DG", 0)
S = TensorFunctionSpace(mesh, "DG", 0)
P = FunctionSpace(mesh, "DG", 0)

####################################################################################
```

The next section of code begins by defining the model parameters for the given simulation. These include the hydraulic parameters (permeability, `k`, fluid viscosity, `mu`, initial porosity, `phi_0`, minimum porosity, `phi_min`, the reference fluid pressure, `p_0`, and the reference fluid density, `rho_0`) and the mechanical parameters (matrix compressibility, `beta_m`, fluid compressibility, `beta_f`, solid grain compressibility, `beta_s`, and Poission's ratio, `nu`). Here, only a single material type is defined; however, multiple material types can be defined and each would require it's own material properties defined in this section. Multiple materials within a simulation requires the use of subdomains. For more information, please refer to the regular FEniCS software documentation. Notice that the bulk modulus (`K`), shear modulus (`G`), and Biot's coefficient (`alpha`) are calculated rather than stipulated.

The minimum porosity set in this section is forcibly applied in the code through conditional statements to avoid having a negative porosity in the nonlinear model; although, this scenario is rarely encountered. The reference fluid pressure and density are used in the nonlinear model only. For the linear model, fluid density and porosity are constant values given by `rho_0` and `phi_0`.

```
######### DEFINE PARAMETERS AND RELATIONSHIPS ############################

### Physical Constants ###
g = 9.81                                # Gravity               (m/sec^2)

### Hydraulic Parameters ###
k = 1.0                                 # Permeability          (m^2)
mu = 1.0                                # Viscosity             (Pa*s)
phi_0 = 0.5                             # Initial Porosity      (-)
phi_min = 0.01                          # Minimum allowed Porosity
p_0 = 0.0                               # Reference Pressure    (Pa)
rho_0 = 1.0                             # Reference Density     (kg/m^3)

### Mechanical Parameters ###
beta_m = 1.0                            # Matrix Compressibility (Pa^-1)
beta_f = 1.0                            # Fluid Compressibility  (Pa^-1)
beta_s = 1.0                            # Solid Compressibility  (Pa^-1)
nu = 0.25                               # Poisson's Ratio        (-)
K = beta_m**(-1)                        # Bulk Modulus           (Pa)
G = 3.0/2.0*K*(1.0-2.0*nu)/(1.0+nu)     # Shear Modulus          (Pa)
alpha = 1.0 - beta_s/beta_m             # Biot Coefficient       (-)
```

This section of code then ends by defining the hydrostatic pressure profile and solution dependent relationships. The hydrostatic pressure profile is used only for converting the pressure perturbation variable to total pressure, as described in the associated journal article (Haagenson et al., 2019). The solution dependent variables include the definitions of stress and strain, which can be saved as model outputs if desired. By defining these variables as python functions, the user has the ability to easily alter these relationships as desired. For example, the stress-strain constitutive relationship could easily be changed to something other than standard elasticity. Moreover, nonlinearities other than fluid density and porosity could be included in the model here (and subsequently included in the model's weak form).

```
### Hydrostatic Condition ###
p_h = project(Expression('rho_0*g*(z1-x[2])',degree=1,rho_0=rho_0,g=g,\
    z1=z1),Z)

### Solution dependent properties ###
# Total Stress
def sigma(us,p,alpha,K,G):
    sigma_val = sigma_e(us,K,G) + alpha*p*Identity(len(us))
    return sigma_val

# Effective Stress
def sigma_e(us,K,G):
    sigma_e_val = -2.0*G*epsilon(us) - (K-2.0/3.0*G)*epsilon_v(us)\
    *Identity(len(us))
    return sigma_e_val

# Strain
def epsilon(us):
    epsilon_val = 1.0/2.0*(grad(us)+grad(us).T)
    return epsilon_val

# Volumetric Strain
def epsilon_v(us):
    epsilon_v_val = div(us)
    return epsilon_v_val

if Linear == 1:
    # Fluid Density
    def rho_f(p,p_0,p_h,rho_0,beta_f):
        rho_val = Constant(rho_0)
        return rho_val

    # Porosity
    def phi(alpha,us,p,beta_s,phi_0,phi_min,t):
        phi_val = Constant(phi_0)
        return phi_val

if Nonlinear == 1:
    # Fluid Density
    def rho_f(p,p_0,p_h,rho_0,beta_f):
        rho_val = Constant(rho_0)*exp(Constant(beta_f)*(p+p_h-\
            Constant(p_0)))
        return rho_val

    # Porosity
    def phi(alpha,us,p,beta_s,phi_0,phi_min,t):
        phi_val = alpha - (alpha-phi_0)*exp(-(epsilon_v(us)+beta_s*p))
        phi_val = conditional(ge(phi_val,phi_min),phi_val,\
            Constant(phi_min))
        return phi_val

##########################################################################
```

The next section defines the time parameters of the simulation. Here, the user should alter the end of the simulation (`tend`) and the number of time steps within the simulation (`nsteps`).

```
######### TIME PARAMETERS  ##############################################

### Time parameters ###
tend = 1.0
nsteps = 10
dt = tend/nsteps

#########################################################################
```

The poroelastic formulation, as described in the associated journal article, considers the unknown field variables to be perturbations from the initial hydrostatic and lithostatic condition, as is common in the theory of poroelasticity (Haagenson et al., 2019). One result of using this formulation is that the initial conditions must be hydrostatic and lithostatic. Otherwise, the formulation using the perturbation variables breaks down mathematically. Therefore, the initial conditions of this model must be left as they are shown below (i.e. with zero perturbation for all field variables, indicating that the system is at hydrostatic and lithostatic conditions). Altering the contents of this section risks the accuracy of the model results significantly.

```
######### INITIAL CONDITIONS  ###########################################

### Initial Condition ###
X_i = Expression(
        (
            "0.0",              # p
            "0.0","0.0","0.0",  # (q1, q2, q3)
            "0.0","0.0","0.0"   # (us1, us2, us3)
        ),degree = 2)
X_n = interpolate(X_i,W)

# Initial porosity
phi_n = interpolate(Constant(phi_0),Z)

# Initial Picard solution estimate
X_m = interpolate(X_i,W)

#########################################################################
```

One of the most important parts of the script is the section containing the boundary conditions. Here, the domain boundaries are defined using the subdomain method; although, other methods exist within the FEniCS framework. Defining these boundaries will depend on the exact domain the user defines for the simulation.

```
######### BOUNDARY CONDITIONS  ##############################################

### Boundary Conditions ###
class LeftBoundary(SubDomain):
    def inside(self, x, on_boundary):
        return near(x[0],x0)
left_boundary = LeftBoundary()

class RightBoundary(SubDomain):
    def inside(self, x, on_boundary):
        return near(x[0],x1)
right_boundary = RightBoundary()

class BackBoundary(SubDomain):
    def inside(self, x, on_boundary):
        return near(x[1],y0)
back_boundary = BackBoundary()

class FrontBoundary(SubDomain):
    def inside(self, x, on_boundary):
        return near(x[1],y1)
front_boundary = FrontBoundary()

class BottomBoundary(SubDomain):
    def inside(self, x, on_boundary):
        return near(x[2],z0)
bottom_boundary = BottomBoundary()

class TopBoundary(SubDomain):
    def inside(self, x, on_boundary):
        return near(x[2],z1)
top_boundary = TopBoundary()

boundary_facet_function = MeshFunction('size_t', mesh, 2)
boundary_facet_function.set_all(0)
left_boundary.mark(boundary_facet_function,1)
right_boundary.mark(boundary_facet_function,2)
back_boundary.mark(boundary_facet_function,3)
front_boundary.mark(boundary_facet_function,4)
bottom_boundary.mark(boundary_facet_function,5)
top_boundary.mark(boundary_facet_function,6)
```

Once the domain boundaries are defined with the `boundary_facet_function` (which may be renamed to `boundary_edge_function` for simulations in two dimensions), the code defines a python function called `GetBoundaryConditions`. This python function takes the current simulation time as an input parameter so that it can handle time dependent boundary conditions. The python function returns a list of the Dirichlet boundary conditions for the simulation. Notice that only flux or displacement boundary conditions can be set in this section. This is because fluid pressure and stress boundary conditions are Neumann conditions in this particular formulation of

poroelasticity. Therefore, fluid pressure and stress boundary conditions are applied through the weak form.

The Brezzi-Douglas-Marini (BDM) elements are vector-valued, and therefore must be assigned a vector-valued object (Brezzi et al., 1985; Brezzi and Fortrin, 1991). Any vector in a BDM element is normal to the facet or edge that it exists on. Hence, the vector stipulated in the Dirichlet boundary condition must be normal to the facets or edges on the particular boundary it is applied. See the FEniCS tutorials for methods on defining vector valued objects that are normal to an arbitrary facet or edge orientation.

```
def GetBoundaryConditions(t):

    bcs = []

    # Left Boundary
    # # Flux Boundary (normal)
    # bcs.append(DirichletBC(W.sub(1), Constant((0.0,0.0,0.0)), \
    #     boundary_facet_function, 1))
    # # Displacement Boundary
    # bcs.append(DirichletBC(W.sub(2), Constant((0.0,0.0,0.0)), \
    #     boundary_facet_function, 1))

    # Right Boundary
    # # Flux Boundary (normal)
    # bcs.append(DirichletBC(W.sub(1), Constant((0.0,0.0,0.0)), \
    #    boundary_facet_function, 2))
    # # Displacement Boundary
    # bcs.append(DirichletBC(W.sub(2), Constant((0.0,0.0,0.0)), \
    #    boundary_facet_function, 2))

    # Back Boundary
    # # Flux Boundary (normal)
    # bcs.append(DirichletBC(W.sub(1), Constant((0.0,0.0,0.0)), \
    #    boundary_facet_function, 3))
    # # Displacement Boundary
    # bcs.append(DirichletBC(W.sub(2), Constant((0.0,0.0,0.0)), \
    #    boundary_facet_function, 3))

    # Front Boundary
    # # Flux Boundary (normal)
    # bcs.append(DirichletBC(W.sub(1), Constant((0.0,0.0,0.0)), \
    #    boundary_facet_function, 4))
    # # Displacement Boundary
    # bcs.append(DirichletBC(W.sub(2), Constant((0.0,0.0,0.0)), \
    #    boundary_facet_function, 4))
```

```
    # Top Boundary
    # # Flux Boundary (normal)
    # bcs.append(DirichletBC(W.sub(1), Constant((0.0,0.0,0.0)), \
    #    boundary_facet_function, 6))
    # # Displacement Boundary
    # bcs.append(DirichletBC(W.sub(2), Constant((0.0,0.0,0.0)), \
    #    boundary_facet_function, 6))

    return bcs

##########################################################################
```

The solver set-up section contains all the required information to build the model's weak form. The delineation of `dx` can be included if the model is set up to run with multiple material types.

```
######### SOLVER SET-UP  ###############################################

U = TrialFunction(W)
V = TestFunction(W)

n = FacetNormal(mesh)
norm = as_vector([n[0], n[1], n[2]])

ff = Constant(0.0)  # fluid source

X = Function(W)

density_save = Function(Z)
porosity_save = Function(Z)

# dx = Measure("dx")(subdomain_data=***subdomain-face-function-name***)
ds = Measure("ds")(subdomain_data=boundary_facet_function)

# Function of ones to evaluate grad(rho_f) in weak form
ones_func = project(Constant(1.0),Z,solver_type='gmres')
```

The python function `WeakForm` takes the test and trial functions as inputs, along with the solution from the previous time step (`X_n`) and the simulation time (`t`). The python function returns the left and right side of the weak form as `A` and `B`. Refer to the associated journal article for details on the weak form (Haagenson et al., 2019). If multiple material types are desired, a weak form for each material type must be defined here and summed together before returning `A` and `B`. The user has great control of the weak form in the FEniCS framework. For example, the flow law (now given as Darcy's law) could be easily altered. Or, if desired, other nonlinearities could be included in the same manner that the nonlinear fluid density and porosity have been already. Here, the weak form is defined in parts (i.e. the conservation of mass, Darcy's law, and the conservation of momentum) before being summed together. The right hand side of Darcy's law and the conservation of momentum are Neumann boundary condition terms for fluid pressure and stress respectively.

```
def WeakForm(U,V,X_n,t):

    p, q, us = split(U)
    Pt, Qt, Ust = split(V)
    p_n, q_n, us_n = split(X_n)
    p_m, q_m, us_m = split(X_m)

    ### Weak Forms ###
    # Conservation of Mass
    CoMass_l_1 = rho_f(p_m,p_0,p_h,rho_0,beta_f)*Constant(alpha)\
        *epsilon_v(us)*Pt*dx
    CoMass_l_2 = rho_f(p_m,p_0,p_h,rho_0,beta_f)*((Constant(alpha)\
        -phi(alpha,us_m,p_m,beta_s,phi_0,phi_min,t))*Constant(beta_s)\
        + phi(alpha,us_m,p_m,beta_s,phi_0,phi_min,t)*Constant(beta_f))*p\
        *Pt*dx
    CoMass_l_3 = dt*rho_f(p_m,p_0,p_h,rho_0,beta_f)*div(q)*Pt*dx
    CoMass_l_4 = dt*Constant(weight)*inner(q,\
        grad(rho_f(p_m,p_0,p_h,rho_0,beta_f)*ones_func))*Pt*dx
    CoMass_l = CoMass_l_1 + CoMass_l_2 + CoMass_l_3 + CoMass_l_4
    CoMass_r_1 = dt*ff*Pt*dx
    CoMass_r_2 = rho_f(p_m,p_0,p_h,rho_0,beta_f)*Constant(alpha)\
        *epsilon_v(us_n)*Pt*dx
    CoMass_r_3 = rho_f(p_m,p_0,p_h,rho_0,beta_f)*((Constant(alpha)\
        -phi(alpha,us_m,p_m,beta_s,phi_0,phi_min,t))*Constant(beta_s) \
        + phi(alpha,us_m,p_m,beta_s,phi_0,phi_min,t)*Constant(beta_f))*p_n\
        *Pt*dx
    CoMass_r = CoMass_r_1 + CoMass_r_2 + CoMass_r_3

    # Darcy's Law
    DL_l = mu/k*inner(q,Qt)*dx - p*div(Qt)*dx
    # DL_r = -Constant(***pressure***)*inner(Qt,norm)*ds(-)

    # Conservation of Momentum
    CoMom_l = inner(sigma(us,p,alpha,K,G),grad(Ust))*dx
    # CoMom_r = inner(Constant((0.0,***loading***)),Ust)*ds(-)

    A = CoMass_l + DL_l + CoMom_l
    B = CoMass_r #+ DL_r + CoMom_r

    return A,B
```

The part of this section defines a python function called `LinearSolver`, which takes the test and trial functions as inputs, in addition to the solution at the previous time step (`X_n`), the current simulation time (`t`), and the list of Dirichlet boundary conditions (`bcs`). The function builds the weak form, assembles the system of equations, and solves that system with the solving method stipulated. Here, the solving method has been set to MUMPS, which is a direct solver capable of solving a system while running in parallel. For larger systems, other solving methods may be more efficient. The python function then returns the solution as `X`.

```
def LinearSolver(U,V,X_n,t,bcs):
    a,L = WeakForm(U,V,X_n,t)
    A, b = assemble_system(a,L,bcs)
    solve(A, X.vector(), b, 'mumps')
    return X

###########################################################################
```

The output section works in conjunction with the post-processing section of the time loop. Here, the files for the three field variables (fluid pressure, specific flux, and solid displacement) are defined, as well as files for the fluid density and porosity (i.e. the nonlinearities for the nonlinear model). Other variables can be saved, such as stress or strain, by defining a file for that variable here. Then, the user can project that variable to the appropriate function space defined in the FUNCTION SPACES section and save it during the post-processing section of the time loop. The xdmf file type can be viewed in either Visit or Paraview for visualization and further post-processing of the model results.

```
######### OUTPUT  ###########################################################

pressure_file = XDMFFile('General/pressure.xdmf')
flux_file = XDMFFile('General/flux.xdmf')
disp_file = XDMFFile('General/disp.xdmf')
density_file = XDMFFile('General/density.xdmf')
porosity_file = XDMFFile('General/porosity.xdmf')

###########################################################################
```

Finally, the script ends with the time loop. Within the time loop, the code employs a Picard iteration (which has its own loop), calls the solver, evaluates the solution for convergence, and updates the previous estimate of the solution and the solution from the previous time step (if the solution has indeed converged). Lastly, the code performs all the post-processing for the model, which is highly customizable by the model user.

After updating the model time and printing it to the terminal window, the code defines the Picard iteration parameters (including a relaxation parameter – `omega`).

```
######### TIME LOOP ##########################################################

t = 0.0

if MPI.rank(mpi_comm_world()) == 0:
    print ('Starting Time Loop...')

### Time Loop ###
for n in range(nsteps):

    MPI.barrier(mpi_comm_world())

    t += dt

    if MPI.rank(mpi_comm_world()) == 0:
        print "###############"
        print ""
        print "NEW TIME STEP"
        print "Time =", t
        print ""
        print np.round(t/tend*100.0,6),"% Complete"

    ### Convergence criteria ###
    reltol = 1E-4               # Relative error tolerance for Picard
    rel_error_max_global = 9999 # Initialize relative error
    max_iter = 100              # Maximum Picard iterations
    omega = 1.0                 # Relaxation coefficient

    iter = 0
```

Next comes the Picard loop where the actual solve call comes in. Notice that the boundary conditions are found for this specific time before entering the Picard loop. If the simulation is set to run with the linear model, the Picard loop is not necessary. Still, the code enters the Picard loop (without printing any of the Picard information to the terminal) and then immediately exits after the first solution is found. In the linear case, this first solution is the precise solution.

```
    if linear_flag == 0:
        if MPI.rank(mpi_comm_world()) == 0:
            print "Entering Picard Iteration:"
            print ""

    # Get boundary conditions for this time
    bcs = GetBoundaryConditions(t)

    ### Picard Iteration Loop ###
    while (rel_error_max_global > reltol):

        if linear_flag == 0:
            if MPI.rank(mpi_comm_world()) == 0:
                print "ITERATE"

        iter += 1

        if linear_flag == 0:
            if MPI.rank(mpi_comm_world()) == 0:
                print "iteration = ", iter
                print ""

        ### Solve ###
        X = LinearSolver(U,V,X_n,t,bcs)

        if MPI.rank(mpi_comm_world()) == 0:
            print ""

        if linear_flag == 0:
            if MPI.rank(mpi_comm_world()) == 0:
                print "Solved for a new solution estimate."
                print ""
```

Next, the solution is evaluated for convergence based on the pressure solution alone. The previous estimate in the Picard iteration is updated with the new solution estimate. If user stipulates that this is the linear model, then the code immediately exits the Picard iteration (as a single iteration with a linear solver will produce the correct result for the linear model).

```
        p, q, us = X.split(True)
        p_m, q_m, us_m = X_m.split(True)
        p_n, q_n, us_n = X_n.split(True)

        # Evaluate for convergence of pressure solution on each processor
        if linear_flag == 1:
            rel_error_max_local = 0.0
        if linear_flag == 0:
            if MPI.rank(mpi_comm_world()) == 0:
                print "Evaluate for Convergence"
                print "--------------"
            cell_values_p = p.vector().get_local()
            cell_values_p_m = p_m.vector().get_local()
            rel_error_max_local = np.nanmax(np.divide(np.abs(cell_values_p \
                - cell_values_p_m),np.abs(cell_values_p_m)))

        # Find the global maximum value of the relative error
        rel_error_max_global = MPI.max(mpi_comm_world(),rel_error_max_local)

        if MPI.rank(mpi_comm_world()) == 0:
            print "Relative Error = ",rel_error_max_global
            print "--------------"
            print ""

        # Update estimate
        X_new = X_m + omega*(X - X_m)
        X_m.assign(X_new)

        if iter == max_iter:
            if MPI.rank(mpi_comm_world()) == 0:
                print "Maximum iterations met"
                print "Solution doesn't converge"
            quit()

    if linear_flag == 0:
        if MPI.rank(mpi_comm_world()) == 0:
            print "The solution has converged."
            print "Total iterations = ", iter
            print ""
```

Finally, the variables are saved to their respective files and the solution from the previous time step is updated. This is the post-processing section, and the user can customize this section significantly. For example, a simulation of contaminate transport could be placed in this section. Another possibility would be to use the pressure and stress solutions to evaluate a failure criterion within the model domain.

```
    if MPI.rank(mpi_comm_world()) == 0:
        print "Saving solutions."
        print ""
    pressure_file.write(p,t)
    flux_file.write(q,t)
    disp_file.write(us,t)
    density_save.assign(project(rho_f(p,p_0,p_h,rho_0,beta_f),P))
    porosity_save.assign(project(phi(alpha,us,p,beta_s,phi_0,phi_min,t),P))
    density_file.write(density_save,t)
    porosity_file.write(porosity_save,t)

    # Update solution at last time step
    X_n.assign(X)
    phi_n.assign(project(phi(alpha,us,p,beta_s,phi_0,phi_min,t),P))
    if MPI.rank(mpi_comm_world()) == 0:
        print "Just updated last solution."
        print ""
        print "###############"
        print ""

##############################################################################

if MPI.rank(mpi_comm_world()) == 0:
    print "This code finished at"
    print strftime("%Y-%m-%d %H:%M:%S", gmtime())
```

3 Tutorial: Soil Heaving

The code described Section 2 is only a template for the poroelastic model. The user must edit this template in order to properly run a simulation. This section presents a simple step-by-step way to update the template. Here, the simulation is a three dimensional column of soil, which heaves upward as fluid flows in through the top boundary. This example could represent soil heaving during water ponding on the ground surface.

To start, create a directory for the test problem:

```
$ mkdir Soil_Heaving_Tutorial
```

and copy the general template into this directory:

```
$ cp General_Poroelastic_Model.py Soil_Heaving_Tutorial/Soil_Heaving.py
```

Open the script in the soil heaving tutorial directory with any text editing software. Even though we could switch the model to the nonlinear version, the problem parameters indicate that the fluid density and porosity are unlikely to change much during the simulation. Hence, we can leave the simulation as the linear model for this tutorial.

Edit the script with the following steps:

1. Update the domain constants to reflect the three dimensional column:

```
######### DOMAIN AND SUBDOMAINS ##########################################

### Domain Constants ###
x0 = 0.0         # Minimum x
x1 = 1.0        # Maximum x
y0 = 0.0         # Minimum y
y1 = 1.0        # Maximum y
z0 = 0.0         # Minimum z
z1 = 10.0       # Maximum z
nx = 5          # Number of cells in x-direction
ny = 5          # Number of cells in y-direction
nz = 100         # Number of cells in z-direction
```

2. Update the model parameters to reflect a realistic soil:

```
######### DEFINE PARAMETERS AND RELATIONSHIPS ############################

### Physical Constants ###
g = 9.81                                # Gravity                (m/sec^2)

### Hydraulic Parameters ###
k = 1.0e-12                             # Permeability           (m^2)
mu = 8.9e-4                             # Viscosity              (Pa*s)
phi_0 = 0.2                             # Initial Porosity       (-)
phi_min = 0.01                          # Minimum allowed Porosity
p_0 = 0.0                               # Reference Pressure     (Pa)
rho_0 = 1000.                           # Reference Density      (kg/m^3)

### Mechanical Parameters ###
beta_m = 1.0e-10                        # Matrix Compressibility (Pa^-1)
beta_f = 4.4e-10                        # Fluid Compressibility  (Pa^-1)
beta_s = 1.0e-11                        # Solid Compressibility  (Pa^-1)
nu = 0.25                               # Poisson's Ratio        (-)
K = beta_m**(-1)                        # Bulk Modulus           (Pa)
G = 3.0/2.0*K*(1.0-2.0*nu)/(1.0+nu)     # Shear Modulus          (Pa)
alpha = 1.0 - beta_s/beta_m             # Biot Coefficient       (-)
```

3. Update the time parameters for the simulation:

```
######### TIME PARAMETERS  ###############################################

### Time parameters ###
tend = 10.0
nsteps = 10
dt = tend/nsteps
```

4. Update the OUTPUT section to put the output files in a more descriptive directory:

```
######### OUTPUT  ########################################################

pressure_file = XDMFFile('Soil_Heaving_Tutorial/pressure.xdmf')
flux_file = XDMFFile('Soil_Heaving_Tutorial/flux.xdmf')
disp_file = XDMFFile('Soil_Heaving_Tutorial/disp.xdmf')
density_file = XDMFFile('Soil_Heaving_Tutorial/density.xdmf')
porosity_file = XDMFFile('Soil_Heaving_Tutorial/porosity.xdmf')

##########################################################################
```

5. Update the boundary conditions. The front, back, left and right boundaries are being set to zero flux and zero normal displacement boundaries. The bottom boundary is also zero flux and is pinned in place. The top boundary is a free surface with a constant pressure applied. The top boundary conditions are applied later in the weak form, so no need to add any Dirichlet boundary conditions for the top boundary at this point.

```
def GetBoundaryConditions(t):

    bcs = []

    # Left Boundary
    # Flux Boundary (normal)
    bcs.append(DirichletBC(W.sub(1), Constant((0.0,0.0,0.0)), \
        boundary_facet_function, 1))
    # Displacement Boundary
    bcs.append(DirichletBC(W.sub(2).sub(0), Constant(0.0), \
        boundary_facet_function, 1))

    # Right Boundary
    # Flux Boundary (normal)
    bcs.append(DirichletBC(W.sub(1), Constant((0.0,0.0,0.0)), \
        boundary_facet_function, 2))
    # Displacement Boundary
    bcs.append(DirichletBC(W.sub(2).sub(0), Constant(0.0), \
        boundary_facet_function, 2))
```

```
    # Back Boundary
    # Flux Boundary (normal)
    bcs.append(DirichletBC(W.sub(1), Constant((0.0,0.0,0.0)), \
        boundary_facet_function, 3))
    # Displacement Boundary
    bcs.append(DirichletBC(W.sub(2).sub(1), Constant(0.0), \
        boundary_facet_function, 3))

    # Front Boundary
    # Flux Boundary (normal)
    bcs.append(DirichletBC(W.sub(1), Constant((0.0,0.0,0.0)), \
        boundary_facet_function, 4))
    # Displacement Boundary
    bcs.append(DirichletBC(W.sub(2).sub(1), Constant(0.0), \
        boundary_facet_function, 4))

    # Bottom Boundary
    # Flux Boundary (normal)
    bcs.append(DirichletBC(W.sub(1), Constant((0.0,0.0,0.0)), \
        boundary_facet_function, 5))
    # Displacement Boundary
    bcs.append(DirichletBC(W.sub(2), Constant((0.0,0.0,0.0)), \
        boundary_facet_function, 5))
```

2. Since we are now using a stipulated pressure boundary condition, we need to uncomment the right hand side of Daryc's law. Update this line as the follows:

```
    # Darcy's Law
    DL_l = mu/k*inner(q,Qt)*dx - p*div(Qt)*dx
    DL_r = -Constant(100.0)*inner(Qt,norm)*ds(6)
```

Here, the constant pressure is 100 Pa, and it is being applied to the top boundary, which is marked with the number six on the boundary facet function.

3. We are technically applying a stipulated stress boundary condition (or a traction boundary condition) to the top boundary. This is a zero stress boundary to represent the free surface. However, since the stipulated stress is zero, we can simply leave the right hand side of the conservation of momentum equation commented out. Update the line defining the right hand side of the weak form (`B`) to include the right hand side of Darcy's law:

```
    B = CoMass_r + DL_r #+ CoMom_r
```

Now the model is ready to run. From the model's directory, simply run:

```
$ python Soil_Heaving.py
```

To run the simulation in parallel with 4 processors, simply run:

```
$ mpirun –n 4 python Soil_Heaving.py
```

If set up properly, the model results should show fluid pressure diffusing downward into the column from the top boundary. The fluid flow (or specific discharge) should be, therefore, downward along the soil column. As fluid pressure in the upper portions of the column increase, the soil will heave upward. The solid displacements should be upward, with larger displacements occurring near the top of the column. Some results from this tutorial are shown below in Figure 1.

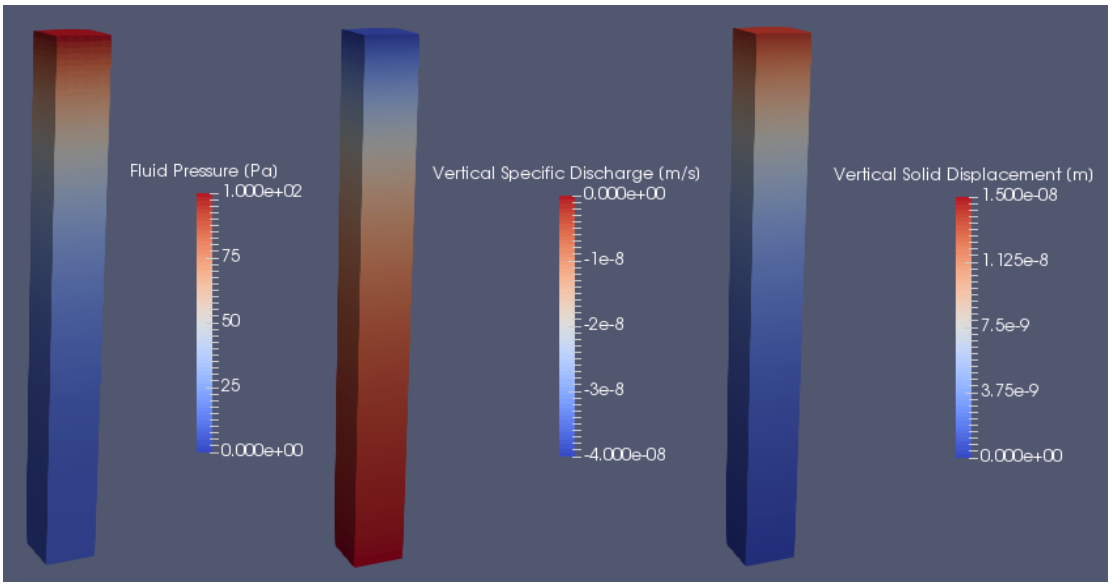

Figure 1. The model results for fluid pressure, the vertical component of specific discharge, and the vertical component of solid displacement for the soil heaving tutorial.

4 References

1. Alnaes, M., Logg, A., Ølgaard, K. B., Rognes, M. E., Wells, G. N., 2014. Unified form language: a domain-specific language for weak formulations of partial differential equations. ACM Transactions on Mathematical Software 40(9).

2. Alnaes, M. Blechta, J., Hake, J., Johansson, A., Kehlet, B., Logg, A., Richardson, C., Ring, J., Rognes, M. E., Wells, G. N., 2015. The FEniCS project version 1.5. Archive of Numerical Software 3(100).

3. Brezzi, F., Douglas, J. Jr., Marini, L. D., 1985. Two families of mixed finite elements for second order elliptic problems. Numerische Mathematik 47(2), 217-235.

4. Brezzi, F., Fortin, M., 1991. Mixed and Hybrid Finite Element Methods. Springer-Verlag, New York, NY, 352 pp.

5. Haagenson, R., Rajaram, H., Allen, J., 2019. A generalized poroelastic model using FEniCS with insights into the Noordbergum effect. Computers & Geosciences.

6. Kirby, R. C., 2004. FIAT: a new paradigm for computing finite element basis functions. ACM Trans- actions on Mathematical Software 30, 502-516.

7. Kirby, R. C., Logg, A., 2006. A compiler for variational forms. ACM Transactions on Mathematical Software 32.

8. Kirby, R. C., Logg, A., Rognes, M. E., Terrel, A. R., 2012. Common and unusual finite elements. In: Automated Solution of Differential Equations by the Finite Element Method. Springer, Berlin, Germany. pp. 91-116.

9. Logg, A., Wells, G. N., 2010. DOLFIN: automated finite element computing. ACM Transactions on Mathematical Software 32(20).

10. Logg, A., Mardal, K., Wells, G., 2012. Automated Solution of Differential Equations by the Finite Element Method. Springer.