INL/EXT-16-38931

Light Water Reactor Sustainability Program

Development of Three-Field Poromechanics Capability in MOOSE

M. V. Sivaselvan
Nicholas Oliveto
Andrew Whittaker

April 2016

DOE Office of Nuclear Energy

INL/EXT-16-38931

Light Water Reactor Sustainability Program

Development of Three-Field Poromechanics Capability in MOOSE

M. V. Sivaselvan – University at Buffalo
Nicholas Oliveto – University at Buffalo
Andrew Whittaker – University at Buffalo

April 2016

Idaho National Laboratory
Idaho Falls, Idaho 83415

http://www.inl.gov/lwrs

Prepared for the
U.S. Department of Energy
Office of Nuclear Energy
Under DOE Idaho Operations Office
Contract DE-AC07-05ID14517

ABSTRACT

This report describes the implementation of a three-field `poromechanics` formulation in the Multiphysics Object Oriented Simulation Environment (MOOSE). A three-field formulation is necessary when the acceleration of the pore fluid relative to the porous skeleton is significant, and the compressibility of the skeleton material and pore fluid is much smaller than the deformability of the medium as a whole. This is the case, for example, when modeling seismic response of saturated soils. The implementation described in this report builds on kernels and boundary conditions that are already in MOOSE in the `tensor_mechanics`, (two-field) `poromechanics` and `navier_stokes` modules, and adds three new kernels and a boundary condition. For time integration, Newmark's method is used, following the auxiliary kernel strategy in the `tensor_mechanics` module; an additional auxiliary kernel is implemented. The implementation is verified using a challenging numerical example, for which most results presented in the literature are erroneous. The MOOSE solution is verified by comparing it with reliable solutions obtained using (a) a boundary element method, (b) a three-field mixed finite element method, and (c) the commercial software, COMSOL.

CONTENTS

ABSTRACT iii

FIGURES v

1. INTRODUCTION 1

2. Governing equations 1

3. Weak form and mapping to kernels 3

4. Time discretization and auxiliary kernels 3

5. Kernel and boundary condition implementation 4
   - 5.1 Registration 4
   - 5.2 Kernel `PoreFluidInertialForceCoupling` 4
     - 5.2.1 Residual 4
     - 5.2.2 Jacobian 5
   - 5.3 Kernel `DynamicDarcyFlow` 5
     - 5.3.1 Residual 5
     - 5.3.2 Jacobian 6
   - 5.4 Kernel `MassConservationNewmark` 7
     - 5.4.1 Residual 7
     - 5.4.2 Jacobian 7
   - 5.5 Boundary Condition `PorePressureBC` 8

6. Verification Example 8
   - 6.1 Preliminary analysis with skeleton alone 9
   - 6.2 Analysis of porous medium 9
     - 6.2.1 Mesh Generation 9
     - 6.2.2 Additional items in the input file 10
     - 6.2.3 Results 10

7. Summary 14

8. REFERENCES 15

APPENDIX A – SOURCE CODE FOR THREE-FIELD POROMECHANICS IMPLEMENTATION 17

APPENDIX B – INPUT FILES FOR VERIFICATION EXAMPLE 32

FIGURES

Figure 1. Geometry and loading for verification example (points labeled nodes 1 and 2 indicate locations where the displacement response is plotted) 9

Figure 2. Displacement of node 2 from dynamic analysis of skeleton alone – comparison of MOOSE and ABAQUS solutions 10

Figure 3. MOOSE solution using new three-field formulation – displacements at nodes 1 and 2 11

Figure 4. Solution using three-field mixed finite element model [26] – displacements at nodes 1 and 2 12

Figure 5. COMSOL solution – displacements at nodes 1 and 2 13

Development of Three-Field Poromechanics Capability in MOOSE

1. INTRODUCTION

The goal of this project is to implement a *three-field* formulation of the dynamics of saturated porous media in the Multiphysics Object Oriented Simulation Framework (MOOSE) [1]. Dynamics of porous media are of interest in a variety of fields including mechanics of saturated and partially saturated soil, porous biological materials and petroleum reservoirs. In particular, our motivation here is to simulate and understand the dynamic response of saturated soil under earthquakes and the subsequent soil liquefaction process [2-4].

Partial differential equation (PDE) models of the dynamics of a saturated porous medium are based on representing it as a solid skeleton filled with fluid. Such models contain three fields — the skeleton displacement (u), the velocity of the fluid relative to the skeleton, or Darcy velocity (w), and the excess pressure of the fluid contained in the skeleton, or the pore pressure (p). Various finite element formulations are possible for these PDE models – three-field (u-w-p) formulations and two-field (u-p or u-w) formulations [2]. A majority of current implementations are based on two-field formulations obtained by eliminating either the Darcy velocity w, resulting in a u-p formulation [4-11], or eliminating the pressure, leading to a u-w formulation [12-16]. However, here we pursue a three-field formulation motivated by the following considerations.

1. The Darcy velocity, w, cannot be eliminated from the formulation unless the relative acceleration of the fluid is zero ($\dot{w}$ in the second line of equation (1) below). Thus a u-p formulation is reasonable only for scenarios in which the acceleration of the fluid relative to the solid skeleton is small. Similarly, the pore pressure, p, cannot be eliminated, and thus a u-w formulation is not possible, unless the combined flexibility of the skeleton material and the fluid is non-zero (see the third line of equation (1) below). Therefore, to realistically model phenomena such as saturated soil-foundation-structure interaction in earthquakes, liquefaction and cyclic mobility, where the relative acceleration of the fluid could be substantial, and the skeleton material and fluid are relatively incompressible, a three-field formulation is necessary. This has also been pointed out by Jeremić et al. [3, 17, 18].
2. Since the u-p formulation amounts to solving for pressure and skeleton displacement as primary fields, post-processing is needed to compute the velocity field from the pressure gradient. The Darcy velocity obtained from a u-p formulation is consequently of lower accuracy [19, 20].
3. Both pressure and flux boundary conditions can be applied directly in a three-field formulation, but require special treatment with a two-field formulation [21].
4. Presence of the fluid velocity field in an explicit fashion in the formulation will also allow for poromechanics to be more readily coupled with transport phenomena in MOOSE.

In this report, we describe our implementation of a three-field formulation of the dynamics of porous media in MOOSE, and present a numerical example verifying the implementation.

2. GOVERNING EQUATIONS

We begin with the set of equations developed by Biot [22] to describe dynamics of porous media. We impose the following restrictions, which are appropriate for the response of saturated soils: (i) the skeleton material (i.e., soil particles) and the fluid are incompressible; (ii) there is a single fluid phase (i.e., the soil is fully saturated); (iii) the skeleton has isotropic linear elastic behavior (nonlinear material models can be easily substituted later in MOOSE); (iv) fluid flow follows a linear isotropic form of Darcy's law; (v) kinematics are linearized; and (vi) the additional apparent mass used in Biot's equations is negligible. Then the governing equations are [23]

$$\begin{aligned} &\rho\ddot{u} + \rho^{\mathrm{f}}\dot{w} - \nabla\cdot(\sigma - pI) = 0 \\ &\rho^{\mathrm{f}}\ddot{u} + \frac{\rho^{\mathrm{f}}}{\phi}\dot{w} + \frac{\rho^{\mathrm{f}} g}{K} w + \nabla p = 0 \\ &\nabla\cdot\dot{u} + \nabla\cdot w = 0 \end{aligned} \tag{1}$$

These represent the momentum balance of the porous medium, the dynamic extension of Darcy's flow law, and conservation of fluid mass, respectively. The three fields, as described above, are the skeleton displacement, *u*, the Darcy velocity, *w* and pore fluid pressure, *p*. The Darcy velocity, more precisely, is defined as the relative rate of volume discharge per unit area of the medium, $w = \phi(\dot{u}_{\mathrm{f}} - \dot{u})$, where $\dot{u}_{\mathrm{f}}$ refers to the absolute velocity of the fluid and ϕ is the porosity. ρ^{s} and ρ^{f} are the densities of the skeleton material and the fluid, while $\rho = (1-\phi)\rho^{\mathrm{s}} + \phi\rho^{\mathrm{f}}$ is the wet density of the porous medium; *K* is the hydraulic conductivity of the medium and *g* is acceleration due to gravity. The operators ∇, $\nabla\cdot$ and the superposed dot denote gradient, divergence derivative with respect to time, and *I* is the 3×3 identity matrix.

In the first line of equation (1), σ is the *effective stress* in the skeleton. The total stress in the porous medium, $\sigma^{\mathrm{t}} = \sigma - \alpha_{\mathrm{B}} pI$, where α_{B}, the Biot-Willis coefficient [24], is commonly taken as 1.0 for soil. The constitutive equation relates the effective stress to the strain, $\varepsilon = \frac{1}{2}\left(\nabla u + \nabla u^{\mathrm{T}}\right)$. If the skeleton is modeled as isotropic and linear elastic, this constitutive equation is

$$\sigma = \frac{E}{1+\nu}\left(\varepsilon + \frac{\nu}{1-2\nu}\mathrm{trace}(\varepsilon) I\right) \tag{2}$$

where *E* is modulus of elasticity and ν is Poisson's ratio, both in the drained condition. Other constitutive equations can be used or implemented easily in MOOSE without any modifications of the poromechanics implementation itself.

Introducing, in MOOSE terms, the *auxiliary* variables,

$$v^{\mathrm{s}} = \dot{u}; \quad a^{\mathrm{s}} = \ddot{u}; \quad a^{\mathrm{f}} = \dot{w} \tag{3}$$

the set of equations (1) becomes

$$\begin{aligned} &\rho a^{\mathrm{s}} + \rho^{\mathrm{f}} a^{\mathrm{f}} - \nabla\cdot\sigma + \nabla p = 0 \\ &\rho^{\mathrm{f}} v^{\mathrm{s}} + \frac{\rho^{\mathrm{f}}}{\phi} a^{\mathrm{f}} + \frac{\rho^{\mathrm{f}} g}{K} w + \nabla p = 0 \\ &\nabla\cdot v^{\mathrm{s}} + \nabla\cdot w = 0 \end{aligned} \tag{4}$$

Equations (4) form a system of coupled partial differential equations with *u*, *w*, and *p* as the three unknown fields on a domain Ω. The following are the appropriate boundary conditions,

$$\begin{aligned} &\text{Skeleton boundary conditions}: u = u_{\mathrm{prsc}} \text{ on } \Gamma_u; \quad \sigma^{\mathrm{t}}\cdot\hat{n} = t_{\mathrm{prsc}} \text{ on } \Gamma_t \\ &\text{Fluid boundary conditions} \quad : w\cdot\hat{n} = q_{\mathrm{qrsc}} \text{ on } \Gamma_q; \quad p = p_{\mathrm{prsc}} \text{ on } \Gamma_p \end{aligned} \tag{5}$$

where Γ_u, Γ_t, Γ_q and Γ_p are, respectively, the parts of the boundary, Γ, of Ω, where the skeleton displacement, traction, fluid flux and fluid pressure are prescribed. The corresponding prescribed values are $u_{\mathrm{prsc}}, t_{\mathrm{prsc}}$, q_{qrsc} and p_{prsc}; $\hat{n}$ is the outward unit normal field to the boundary Γ.

We now proceed to develop the weak form of these equations, which is the starting point for a MOOSE implementation.

3. WEAK FORM AND MAPPING TO KERNELS

To build the weak-form representation of equations (4), the three component equations are multiplied by the test functions δu , dw and δp respectively. After appropriate application of integration by parts (divergence theorem), we obtain the weak form

$$\underbrace{\int_\Omega \rho a^{\mathrm{s}} \cdot \delta u \Omega}_{\substack{\text{InertialForce}\\ \text{with } \alpha=0}} + \boxed{\underbrace{\int_\Omega \rho^{\mathrm{f}} a^{\mathrm{f}} \cdot \delta u \Omega}_{\text{PoreFluidInertialForceCoupling}}} + \underbrace{\int_\Omega \sigma \cdot \delta \varepsilon \Omega}_{\text{StressDivergenceTensors}} - \underbrace{\int_{\Gamma_t} (\sigma \hat{n}) \cdot \delta u \Gamma \left(\int_{\Gamma_t} t_{\mathrm{prsc}} \cdot \delta u \Gamma \right)}_{\text{Pressure}}$$

$$- \underbrace{\int_\Omega p \nabla \cdot \delta u \Omega}_{\substack{\text{PoroMechanicsCoupling}\\ \text{with } \alpha_{\mathrm{B}}=1}} + \boxed{\underbrace{\int_\Omega \left(\rho^{\mathrm{f}} a^{\mathrm{s}} + \frac{\rho^{\mathrm{f}}}{\phi} a^{\mathrm{f}} + \frac{\rho^{\mathrm{f}} g}{K} w \right) \cdot \delta w \Omega}_{\text{DynamicDarcyFlow}}} \tag{6}$$

$$- \underbrace{\int_\Omega p \nabla \cdot \delta w \Omega}_{\substack{\text{PoroMechanicsCoupling}\\ \text{with } \alpha_{\mathrm{B}}=1}} + \boxed{\underbrace{\int_{\Gamma_p} p_{\mathrm{prsc}} \hat{n} \cdot \delta w \Gamma}_{\text{PorePressureBC}}} + \boxed{\underbrace{-\int_\Omega \nabla \cdot v^{\mathrm{s}} \delta p \Omega}_{\text{MassConservationNewmark}}} - \underbrace{\int_\Omega \nabla \cdot w \delta p \Omega}_{\text{INSMass}}$$

The text below each term in the weak form shows how the term maps to a *kernel* or *boundary condition* in a MOOSE implementation. We recognize that a number of terms map to existing MOOSE kernels and boundary conditions. The `StressDivergenceTensors`, `InertialForce` and `PoroMechanicsCoupling` kernels and the `Pressure` boundary condition belong to the `tensor_mechanics` module, whereas the `INSMass` kernel is from the `navier_stokes` module. Three new kernels – `PoreFluidInertialForceCoupling`, `DynamicDarcyFlow` and `MassConservationNewmark`, and a new boundary condition — `PorePressureBC` can then be identified and are shown boxed. These need to be implemented for a three-field dynamic poromechanics formulation. Section 5 presentes the implementation. The `MassConservationNewmark` kernel appears very similar to the `INSMass` kernel; however, it uses an auxiliary variable v^{s} , and therefore has to implemented separately. We note that the kinematic boundary conditions $u = u_{\mathrm{prsc}}$ on Γ_u and $w \cdot \hat{n} = q_{\mathrm{qrsc}}$ on Γ_q can be imposed using the `PresetBC` boundary condition in MOOSE.

4. TIME DISCRETIZATION AND AUXILIARY KERNELS

We discretize the formulation in time using Newmark's method. The strong form (4) and the weak form (6) are expressed at time $n+1$. In Newmark's method, the kinematics are approximated by

$$\begin{aligned} v^{\mathrm{s}}_{n+1} &= v^{\mathrm{s}}_n + (1-\gamma)\Delta t a^{\mathrm{s}}_n + \gamma \Delta t a^{\mathrm{s}}_{n+1} \\ u_{n+1} &= u_n + v^{\mathrm{s}}_n \Delta t + \left(\tfrac{1}{2} - \beta\right) \Delta t^2 a^{\mathrm{s}}_n + \beta \Delta t^2 a^{\mathrm{s}}_{n+1} \\ w_{n+1} &= w_n + (1-\gamma)\Delta t a^{\mathrm{f}}_n + \gamma \Delta t a^{\mathrm{f}}_{n+1} \end{aligned} \tag{7}$$

We follow the clever implementation of Newark's method in the `tensor_mechanics` module in MOOSE using *auxiliary kernels*. The following are defined in the existing auxiliary kernels `NewmarkAccelAux` and `NewmarkVelAux` respectively in the tensor_mechanics module.

$$\begin{aligned} a^{\mathrm{s}}_{n+1} &= \frac{1}{\beta \Delta t^2} \left(u_{n+1} - u_n - v^{\mathrm{s}}_n \Delta t - \left(\tfrac{1}{2} - \beta\right) \Delta t^2 a^{\mathrm{s}}_n \right) \\ v^{\mathrm{s}}_{n+1} &= v^{\mathrm{s}} n + (1-\gamma)\Delta t a^{\mathrm{s}}_n + \gamma \Delta t a^{\mathrm{s}}_{n+1} \end{aligned} \tag{8}$$

In line with this approach, a new auxiliary kernel `NewmarkPoreFluidAccelAux` is defined, implementing

$$a^{\mathrm{f}}_{n+1} = \frac{1}{\gamma \Delta t} \left(w_{n+1} - w_n - (1-\gamma)\Delta t a^{\mathrm{f}}_n \right) \tag{9}$$

This is implemented in the auxiliary kernel as follows.

```
Real
NewmarkPoreFluidAccelAux::computeValue()
{
  if (!isNodal())
    mooseError("NewmarkPoreFluidAccelAux must run on a nodal variable");

  Real af_old = _u_old[_qp];
  if (_dt == 0)
    return af_old;
  return 1.0/_gamma*((_w[_qp]-_w_old[_qp])/_dt - af_old*(1.0-_gamma));
}
```

The codes `NewmarkPoreFluidAccel.h` (Listing 1) and NewmarkPoreFluidAccel.C (Listing 2) implementing these kernels is provided in Appendix A.

5. KERNEL AND BOUNDARY CONDITION IMPLEMENTATION

5.1 Registration

As discussed in Section 3, three new kernels

- `PoreFluidInertialForceCoupling`
- `DynamicDarcyFlow`
- `MassConservationNewmark`

and a new boundary condition, `PorePressureBC`, are needed in MOOSE for a three-field poromechanics formulation. These and the auxiliary kernel, `NewmarkPoreFluidAccelAux` discussed in Section 4, are implemented in a new module called `three_field_poromechanics`. They are registered as

```
// External entry point for dynamic object registration
extern "C" void ThreeFieldPoromechanicsApp__registerObjects(Factory & factory) {
ThreeFieldPoromechanicsApp::registerObjects(factory); }
void
ThreeFieldPoromechanicsApp::registerObjects(Factory & factory)
{
  registerAux(NewmarkPoreFluidAccelAux);

  registerKernel(PoreFluidInertialForceCoupling);
  registerKernel(DynamicDarcyFlow);
  registerKernel(MassConservationNewmark);

  registerBoundaryCondition(PorePressureBC);
}
```

in `ThreeFieldPoromechanics.C` (Listing 3 and Listing 4 in Appendix A).

The implementations of the residuals and Jacobians for the new kernels and boundary condition are described in the following.

5.2 Kernel `PoreFluidInertialForceCoupling`

5.2.1 Residual

The kernel `PoreFluidInertialForceCoupling` represents the term

$$\rho^{\mathrm{f}} a^{\mathrm{f}} \cdot \delta u$$

in the weak form (see equation (6)). The primary variable is the skeleton displacement, u, and the Darcy velocity, w, is a coupled variable appearing through the auxiliary variable a^{f}. This term is programmed in MOOSE as

```
Real
PoreFluidInertialForceCoupling::computeQpResidual()
{
  if (_dt == 0)
    return 0.0;
  Real af=1/_gamma*((_w[_qp] - _w_old[_qp])/_dt - (1.0-_gamma)*_af_old[_qp]);
  return _test[_i][_qp]*_rhof[_qp]*af;
}
```

5.2.2 Jacobian

The kernel does not contain the skeleton displacement variable u. Therefore, the diagonal block of the Jacobian is zero.

```
Real
PoreFluidInertialForceCoupling::computeQpJacobian()
{
  return 0.0;
}
```

However, there is a nonzero off diagonal block, because the kernel contains the Darcy velocity variable through the acceleration. Differentiating equation (9) with respect to w, we see that the u-w off diagonal block of the Jacobian is

$$\frac{\rho^{\mathrm{f}}}{\gamma\Delta t}\delta u_i \delta w_j$$

In MOOSE, this takes the form,

```
Real
PoreFluidInertialForceCoupling::computeQpOffDiagJacobian(unsigned int jvar)
{
  if (_dt == 0)
    return 0.0;
  if (jvar != _w_var_num) // only u-w block is nonzero
    return 0.0;
  return _test[_i][_qp]*_rhof[_qp]/(_gamma*_dt)*_phi[_j][_qp];
}
```

The input file must instantiate this kernel as many times as the physical dimension of the problem. For example, for a 2D problem, there must be two instances of the kernel with the corresponding variables as the cartesian components of the skeleton displacement, u. The jacobian block is diagonal; therefore, the coupled variable for each instance is only the corresponding cartesian component of the Darcy velocity w.

The input parameter for the kernel is the constant, γ, in Newmark's method. The kernel is implemented in the files `PoreFluidInertialForceCoupling.h` and `PoreFluidInertialForceCoupling.C` (Listing 5 and Listing 6 in Appendix A).

5.3 Kernel `DynamicDarcyFlow`

5.3.1 Residual

This kernel represents the term

$$\left(\rho^{\mathrm{f}} a^{\mathrm{s}} + \frac{\rho^{\mathrm{f}}}{\phi} a^{\mathrm{f}} + \frac{\rho^{\mathrm{f}} g}{K} w \right) \cdot \delta w$$

in the weak form equation (6). The primary variable is the Darcy velocity, *w*, and the skeleton displacement, *u*, appears as a coupled variable though the auxiliary variable, a^{s}. The residual takes the following form in MOOSE.

```
Real
DynamicDarcyFlow::computeQpResidual()
{
  if (_dt == 0)
    return 0.0;
  Real as=1/_beta*(((_us[_qp]-_us_old[_qp])/(_dt*_dt)) - _vs_old[_qp]/_dt -
_as_old[_qp]*(0.5-_beta));
  Real af=1/_gamma*((_u[_qp]-_u_old[_qp])/_dt - (1.0-_gamma)*_af_old[_qp]);
  return _test[_i][_qp]*_rhof[_qp]*( as + af/_nf[_qp] + _gravity/_K[_qp]*_u[_qp] );
}
```

5.3.2 Jacobian

The Jacobian has diagonal and off diagonal blocks. Differentiating the kernel as well as a^{f} in equation (9) with respect to *w*, we obtain for the diagonal block of the Jacobian

$$\rho^{\mathrm{f}} \left(\frac{1}{\phi \gamma \Delta t} + \frac{g}{K} \right) \delta w_i \delta w_j$$

which in MOOSE becomes

```
Real
DynamicDarcyFlow::computeQpJacobian()
{
  if (_dt == 0)
    return 0.0;
  return _test[_i][_qp]*_rhof[_qp]*(1.0/(_nf[_qp]*_gamma*_dt) +
_gravity/_K[_qp])*_phi[_j][_qp];
}
```

The Jacobian also contains the *u*-*w* off diagonal block. This is obtained by differentiating a^{s} in equation (8) to *u* as

$$\frac{\rho^{\mathrm{f}}}{\beta \Delta t^2} \delta w_i \delta u_j$$

which in MOOSE is

```
Real
DynamicDarcyFlow::computeQpOffDiagJacobian(unsigned int jvar)
{
  if (_dt == 0)
    return 0.0;
  if (jvar != _us_var_num) // only u-w block is nonzero
    return 0.0;
  return _test[_i][_qp]*_rhof[_qp]/(_beta*_dt*_dt)*_phi[_j][_qp];
}
```

Again, there should be one instance of the kernel for each physical dimension of the problem, with the Cartesian component of the Darcy velocity, *w*, as the primary variable. The off-diagonal Jacobian block being a diagonal matrix, the coupled variable is only the corresponding component of the skeleton displacement, *u*.

The kernel requires the parameters acceleration due to gravity *g*, and the constants β and γ of Newmark's time integration method, and is implemented in `DynamicDarcyFlow.h` and `DynamicDarcyFlow.C` (Listing 7 and Listing 8 in Appendix A).

5.4 Kernel `MassConservationNewmark`

This kernel corresponds to the term

$$-\nabla \cdot v^s \delta p$$

As noted earlier in Section 3, even though this is very similar to the `INSMass` kernel that already exists in MOOSE, it has to be implemented separately, since the coupled variable, *u*, appears through the auxiliary variable v^s.

5.4.1 Residual

The residual is implemented as

```
Real
MassConservationNewmark::computeQpResidual()
{
  if (_dt == 0)
    return 0.0;
  Real div_u = _grad_ux[_qp](0) + _grad_uy[_qp](1) + _grad_uz[_qp](2);
  Real div_u_old = _grad_ux_old[_qp](0) + _grad_uy_old[_qp](1) + _grad_uz_old[_qp](2);
  Real div_v_old = _grad_vx_old[_qp](0) + _grad_vy_old[_qp](1) + _grad_vz_old[_qp](2);
  Real div_a_old = _grad_ax_old[_qp](0) + _grad_ay_old[_qp](1) + _grad_az_old[_qp](2);
  Real div_v = (_gamma/_beta/_dt)*(div_u - div_u_old) + (1.0-_gamma/_beta)*div_v_old
             - (1.0 - _gamma/2.0/_beta)*div_a_old;
  return -_test[_i][_qp]*div_v;
}
```

Not surprisingly, the code resembles the residual in the `INSMass` kernel.

5.4.2 Jacobian

The primary variable, the excess pore pressure *p*, does not appear explicitly in the kernel. Therefore, the diagonal block of the Jacobian is zero. The *u*-*p* off diagonal block is obtained by differentiating v^s in equation (8) with respect to *u*. This gives

$$-\frac{1}{\gamma \Delta t} \delta p \delta u_j$$

implemented as

```
Real
MassConservationNewmark::computeQpOffDiagJacobian(unsigned int jvar)
{
  if (_dt == 0)
    return 0.0;
  else if (jvar == _ux_var)
    return -(_gamma/_beta/_dt)*_grad_phi[_j][_qp](0)*_test[_i][_qp];
  else if (jvar == _uy_var)
    return -(_gamma/_beta/_dt)*_grad_phi[_j][_qp](1)*_test[_i][_qp];
  else if (jvar == _uz_var)
    return -(_gamma/_beta/_dt)*_grad_phi[_j][_qp](2)*_test[_i][_qp];
  else
    return 0.0;
}
```

The kernel uses the Newmark constant, γ, as an input parameter, and is implemented in `MassConservationNewmark.h` and `MassConservationNewmark.C` (Listing 9 and Listing 10 in Appendix A).

5.5 Boundary Condition `PorePressureBC`

The residual corresponding to the prescribed excess pore pressure boundary condition term,

$$p_{\text{prsc}}\hat{n}\cdot\delta w$$

is programmed as

```
Real
PorePressureBC::computeQpResidual()
{
  return _test[_i][_qp]*_normals[_qp](_component)*_specified_pore_pressure;
}
```

The Jacobian is zero. The full implementation in the files PorePressureBC.h and PorePressureBC.C is provided in Listing 11 and Listing 12 in Appendix A.

6. VERIFICATION EXAMPLE

A 2D-plane strain model of a block of soil with a strip load on the surface is selected as a verification example. This example has been presented in [4, 25]. It is particularly interesting because numerical results presented in the literature for this problem are incorrect; they fail to even qualitatively resemble the correct solution, incorrectly estimating basic frequency and dissipation characteristics. We computed solutions for this problem using a three-field mixed finite element [26], and verified them using boundary element solutions obtained using the tools of [23]. The boundary element solutions employ a completely different approach, operating in the frequency domain, and discretizing only the boundary. We therefore have confidence in using these numerical results as reference to verify our MOOSE implementation. We also computed solutions to this problem using COMSOL [27] based on a *u-p* formulation, and obtained similar results, adding further confidence; however, the COMSOL solution shows instability for low values of hydraulic conductivity.

The model is shown in Figure 1. Due to symmetry, only half of the domain is represented. Displacements perpendicular to the left, right and bottom walls are constrained. A load of $15\,\text{kPa}$ is applied instantaneously on half of the top surface. The left, right and bottom walls are impermeable, and the fluid pressure at the top surface is zero. The material properties of the skeleton under drained conditions are $E = 14.5\times10^3\ \text{kPa}$ and $\nu = 0.3$. The desities of the skeleton material and fluid are $\rho^{\text{s}} = 2700\text{kg}/\text{m}^3$ and $\rho^{\text{f}} = 1000\text{kg}/\text{m}^3$, and the porosity is $\phi = 0.42$ [4, 10]. Two values of hydraulic conductivity are used

$K = 10^{-1}$ m / s and $K = 10^{-4}$ m / s, which represent extremes of high and low permeability respectively. These extremes are used to explore the stability of the MOOSE three-field implementation under nearly incompressible conditions.

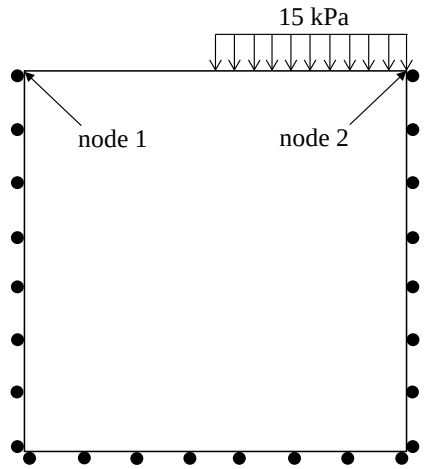


Figure 1. Geometry and loading for verification example (points labeled nodes 1 and 2 indicate locations where the displacement response is plotted)

6.1 Preliminary analysis with skeleton alone

As a first step, we consider the dynamics of the skeleton alone, without the fluid (and with slightly different material properties). We model this in MOOSE using the `tensor_mechanics` module and a 20×20 grid of 4-node quadrilateral Lagrange finite elements, generated using the built in mesh generator. The input file for this model is listed in Appendix B. The load is applied only on the right half of the top boundary and with a small ramp in time using a `ParsedFunction`. The MOOSE solution for the displacement of node 2 is shown in Figure 2 together with a solution from ABAQUS using an identical finite element discretization. Since the first two frequencies of the model are close to each other, a beating phenomenon is observed. There is a slight discrepancy between the MOOSE and ABAQUS solutions. This is because for the 4-node quadrilateral element, ABAQUS uses a lumped mass matrix; since MOOSE uses the weak form directly, it amounts to using a consistent mass matrix. Equipped with this preliminary verification, the full porous medium model is considered next.

6.2 Analysis of porous medium

6.2.1 Mesh Generation

In the full porous medium model, besides the applied load, the top surface also has different boundary conditions on the left and right halves. On the left half, the excess pore pressure is zero (fully drained), while on the right half, the normal component of the Darcy velocity is zero. Therefore, MOOSE's built-in mesh generator cannot be used. We use GMSH [28]. The GMSH geometry file for the model is listed in Appendix B. A regular grid of 20×20 elements is found to be sufficient. The skeleton displacement and Darcy velocity fields are discretized using 9-node quadratic Lagrange elements, while the excess pore pressure field is discretized using 4-node bilinear Lagrange elements. A lower degree approximation must be used for the pressure field in this manner to ensure stability.

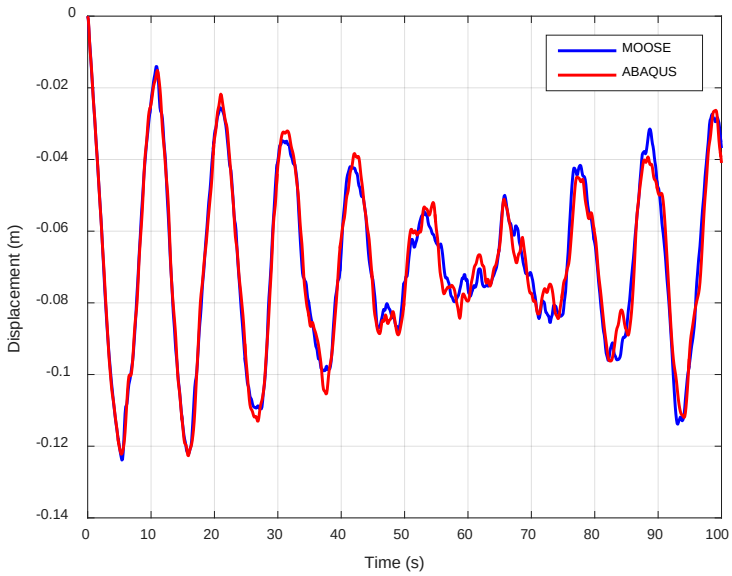


Figure 2. Displacement of node 2 from dynamic analysis of skeleton alone – comparison of MOOSE and ABAQUS solutions

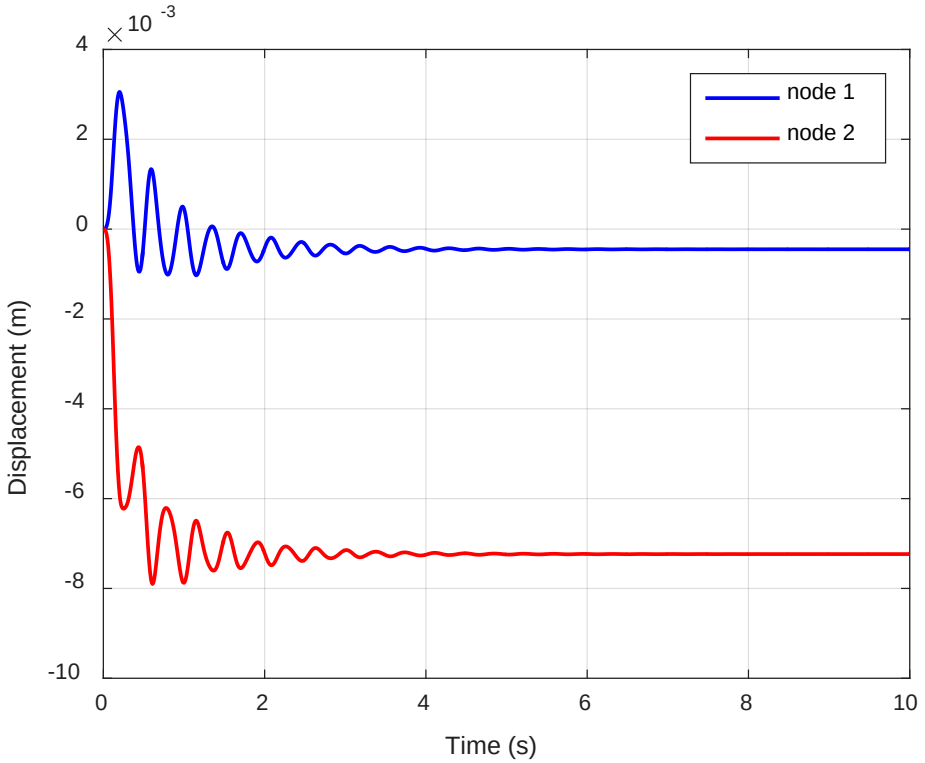
6.2.2 Additional items in the input file

The input file is listed in Appendix B. The following are additional items corresponding to the new kernels for the three-field poromechanics formulation.

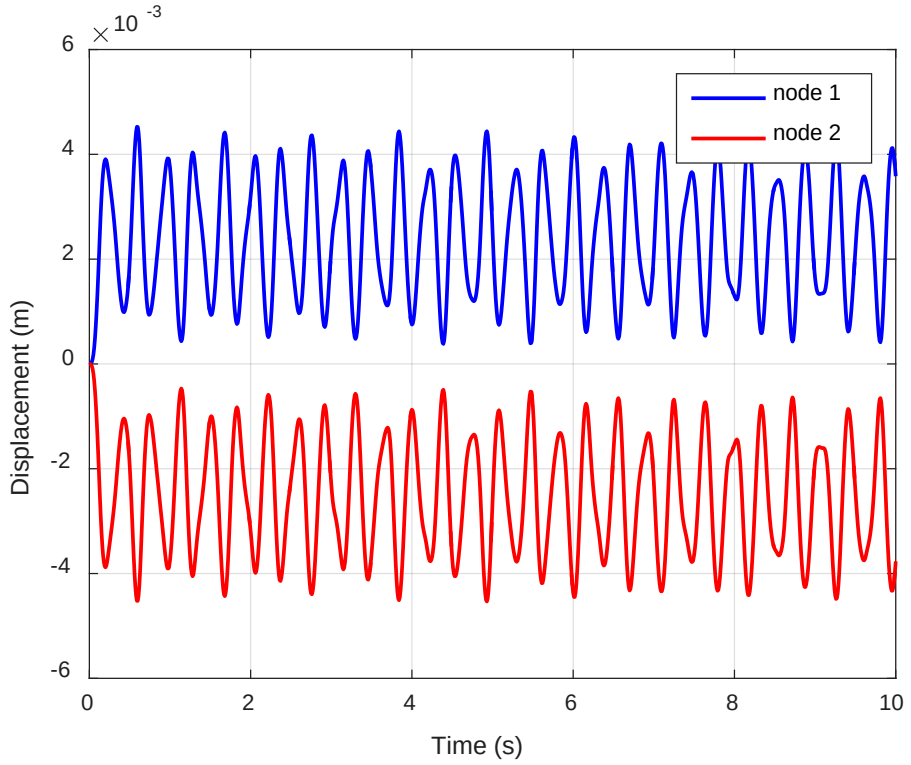
- Variables: In addition to the skeleton displacements `u_x` and `u_y`, and the excess pore pressure `p`, the components of the Darcy velocity field, `w_x` and `w_y` are also included as variables.
- Auxiliary variables: The auxiliary variables `fluidaccel_x` and `fluidaccel_y` are introduced corresponding to the auxiliary kernel `NewmarkPoreFluidAccel`.
- Kernels: The kernels identified in equation (6) are defined.
- Material properties: Pore fluid density, ρ^f (`rho_f`), porosity, ϕ (`porosity`) and hydraulic conductivity, K (`hydconductivity`) are defined.
- Boundary conditions: The normal component of the Darcy velocity is set to zero at the left, right, bottom and top right half boundaries using `PresetBC`s; similarly the pore pressure is set to zero on the top left half boundary
- Preconditioner: A fully couple Jacobian is used.

6.2.3 Results

Figure 3 shows results obtained using the new MOOSE implementation. In Figure 3(a) the displacements of the two corner nodes 1 and 2 (see Figure 1) are shown for the high permeability case (K=10^{-1} m/s). The same displacements are shown for the high permeability case (K=10^{-4} m/s). In the high permeability case, the apparent mechanical damping resulting from the diffusion of the pore fluid is greater, and the eventual consolidation displacement is attained. On the other hand, in the low permeability case, the mechanical damping is lower, and oscillations persist about the consolidation displacement. The frequency of oscillation does not change notably between the two cases.

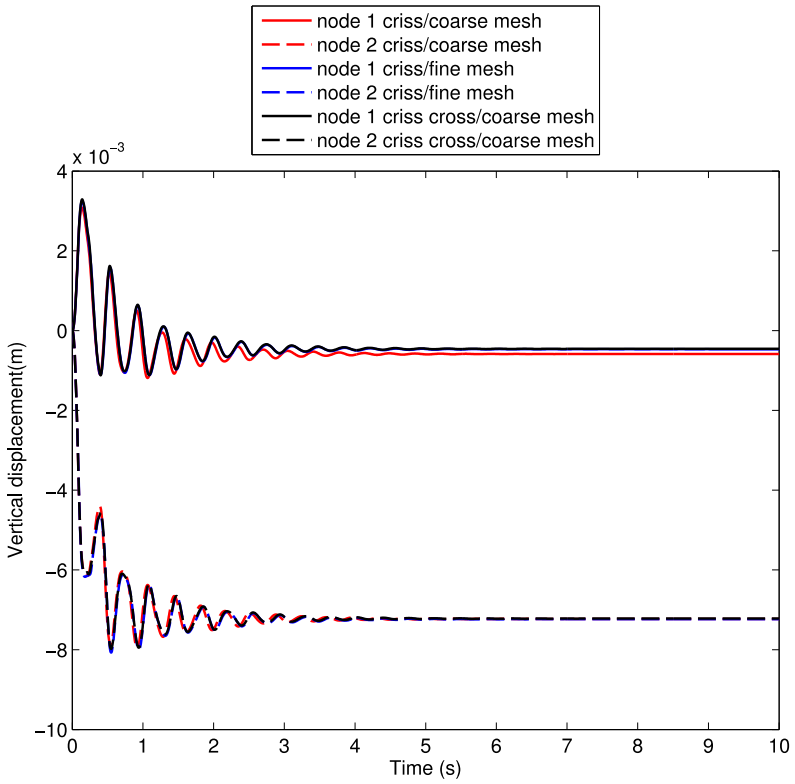


(a) Large hydraulic conductivity, $K = 10^{-1}$ m/s

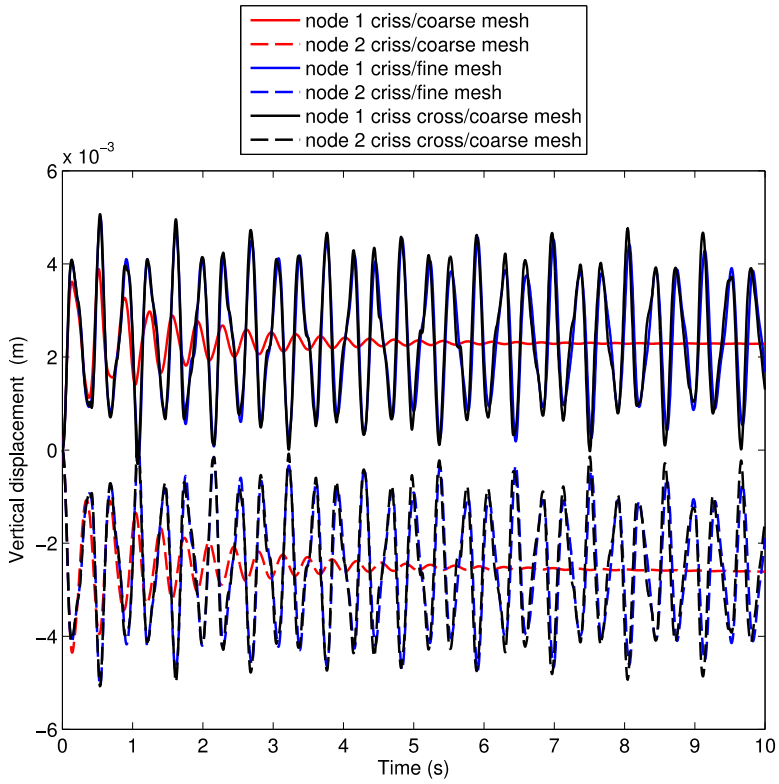


(b) Very small hydraulic conductivity, $K = 10^{-4}$ m/s

Figure 3. MOOSE solution using new three-field formulation – displacements at nodes 1 and 2

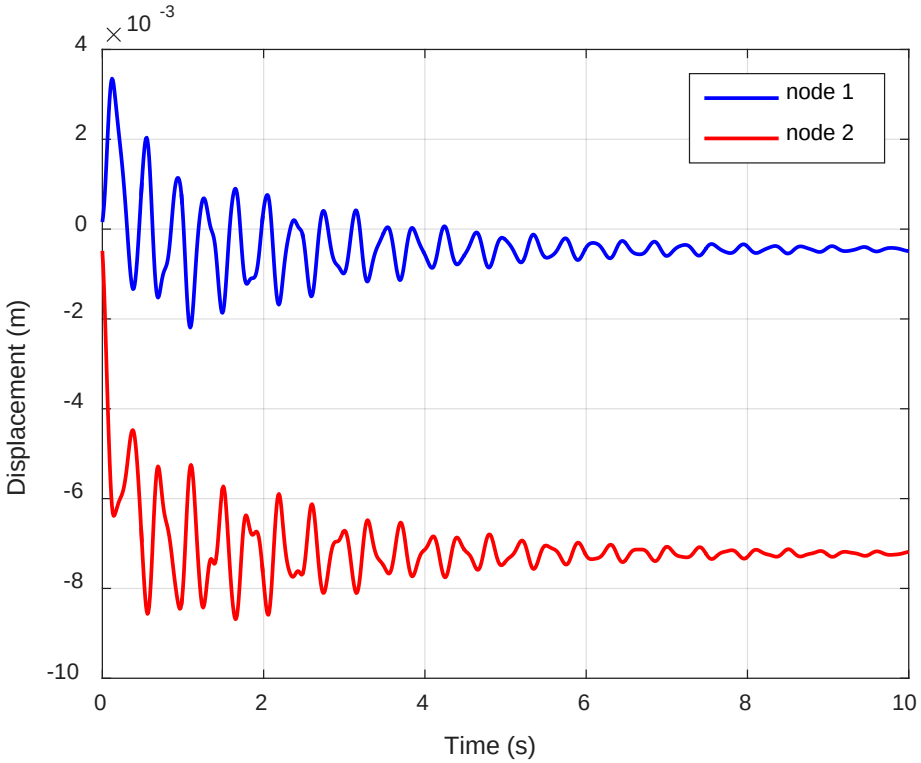


(a) Large hydraulic conductivity, $K = 10^{-1}$ m/s

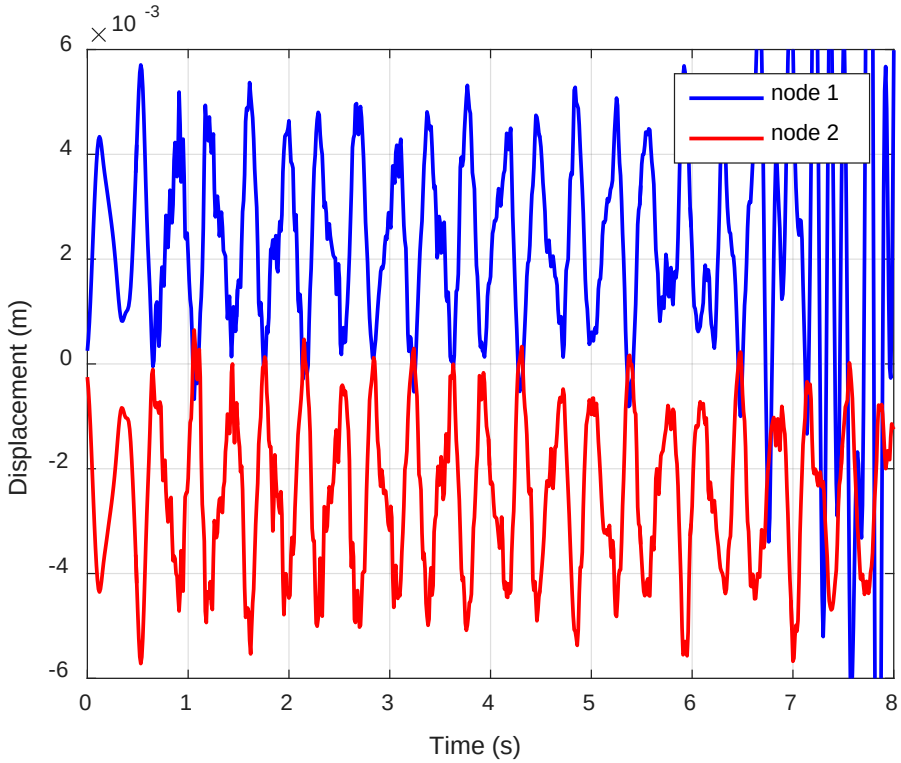


(b) Very small hydraulic conductivity, $K = 10^{-4}$ m/s (the coarse mesh solutions in red do not represent dissipation correctly).

Figure 4. Solution using three-field mixed finite element model [26] – displacements at nodes 1 and 2



(a) Large hydraulic conductivity, $K = 10^{-1}$ m/s



(b) Very small hydraulic conductivity, $K = 10^{-4}$ m/s

Figure 5. COMSOL solution – displacements at nodes 1 and 2

This fact that apparent mechanical damping increases with increasing permeability is not intuitive, for an argument can be made to support the opposite behavior as well. Damping increases with increasing permeability because there is more diffusion of pore fluid, and consequently more dissipation. On the flip side, one might expect higher damping with lower permeability, since the dissipation is greater per unit flow. In fact, results reported in the literature show this reverse behavior [4, 25]. They also do not show the correct frequency of oscillation. We resolved this inconsistency when verifying our previous mixed finite element poromechanics formulation [26]. Results obtained using this formulation are shown in Figure 4. It can be seen that when the permeability is low (Figure 4(b)), the damping characteristics are again not captured correctly if the finite element mesh is not sufficiently fine.

We obtained exact results for this problem using the boundary element method with the tools of [23]. This approach works in the frequency domain, and since only the boundary is discretized, does not suffer from spatial discretization artifacts. Our mixed method [26] with sufficient mesh refinement, matches the boundary element solution exactly. The MOOSE solution also matches this solution exactly (Figure 3 and Figure 4).

Figure 5 shows results obtained for this problem from the commercial software COMSOL [27]. These results are qualitatively similar. However, the COMSOL solution does not represent damping accurately for the high permeability case, and shows instability in the low permeability case. We conclude that the MOOSE implementation performs well in this challenging numerical example.

7. SUMMARY

A three-field poromechanics formulation is implemented in MOOSE. The implementation builds on existing kernels and boundary conditions in the `tensor_mechanics`, (two-field *u-p*) `poromechanics` and `navier_stokes` modules, and adds three new kernels and a boundary condition (see equation (6)). Newmark's method is used for time integration, using the strategy in the `tensor_mechanics` module results in a further auxiliary kernel. The implementation is verified using a challenging numerical example; results compare well with those obtained using very different methods – a boundary element approach, and a mixed finite element method.

8. REFERENCES

1. Gaston, D., Newman, C., Hansen, G., and Lebrun-Grandié, D., *MOOSE: A parallel computational framework for coupled systems of nonlinear equations.* Nuclear Engineering and Design, 2009. 239(10): p. 1768-1778.

2. Zienkiewicz, O. and Shiomi, T., *Dynamic behavior of saturated porous media; the generalized \uppercaseBiot formulation and its numerical solution.* International Journal for Numerical and Analytical Methods in Geomechanics, 1984. 8(1): p. 71-96.

3. Jeremić, B., Cheng, Z., Taiebat, M., and Dafalias, Y., *Numerical simulation of fully saturated porous materials.* International Journal for Numerical and Analytical Methods in Geomechanics, 2008. 32(13): p. 1635-1660.

4. Li, C., Borja, R.I., and Regueiro, R.A., *Dynamics of porous media at finite strain.* Computer Methods in Applied Mechanics and Engineering, 2004. 193(36-38): p. 3837-3870.

5. Liu, R., *Discontinuous Galerkin finite element solution for poromechanics*. 2004, University of Texas at Austin.

6. Murad, M.A. and Loula, A.F.D., *On stability and convergence of finite-element approximations of Biot's consolidation problem.* International Journal for Numerical Methods in Engineering, 1994. 37(4): p. 645-667.

7. Oka, F., Yashima, A., Shibata, T., Kato, M., and Uzuoka, R., *FEM-FDM coupled liquefaction analysis of a porous soil using an elasto-plastic model.* Applied Scientific Research, 1994. 52(3): p. 209-245.

8. Zhu, Q. and Suh, J.F., *Dynamic biphasic poroviscoelastic model simulation of hydrated soft tissues and its potential applications for brain impact study.* American Society of Mechanical Engineering, 2001. 50: p. 835-836.

9. Zienkiewicz, O. *Basic formulation of static and dynamic behavior of soil and other porous media*. in *Numerical Methods in Geomechanics*. 1982. Springer.

10. Zienkiewicz, O., Chan, A., Pastor, M., Paul, D., and Shiomi, T., *Static and dynamic behaviour of soils: a rational approach to quantitative solutions. I. Fully saturated problems.* Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences, 1990. 429(1877): p. 285-309.

11. Zienkiewicz, O., Chang, C., and Bettess, P., *Drained, undrained, consolidating and dynamic behaviour assumptions in soils.* Geotechnique, 1980. 30(4): p. 385-395.

12. Engelman, M.S., Sani, R.L., Gresho, P.M., and Bercovier, M., *Consistent vs reduced integration penalty methods for incompressible media using several old and new elements.* International Journal for Numerical Methods in Fluids, 1982. 2(1): p. 25-42.

13. Prévost, J.H., *Nonlinear transient phenomena in saturated porous media.* Computer Methods in Applied Mechanics and Engineering, 1982. 30(1): p. 3-18.

14. Prévost, J.H., *Wave propagation in fluid-saturated porous media: an efficient finite element procedure.* International Journal of Soil Dynamics and Earthquake Engineering, 1985. 4(4): p. 183-202.

15. Spilker, R.L. and Maxian, T.A., *A mixed penalty finite element formulation of the linear biphasic theory for soft tissues.* International Journal for Numerical Methods in Engineering, 1990. 30(5): p. 1063-1082.

16. Suh, J.-K., Spilker, R., and Holmes, M., *A penalty finite element analysis for nonlinear mechanics of biphasic hydrated soft tissue under large deformation.* International Journal for Numerical Methods in Engineering, 1991. 32(7): p. 1411-1439.

17. Tasiopoulou, P., Taiebat, M., Tafazzoli, N., and Jeremić, B., *On validation of fully coupled behavior of porous media using centrifuge test results.* Coupled Systems Mechanics, 2015. 4: p. 37-65.

18. Tasiopoulou, P., Taiebat, M., Tafazzoli, N., and Jeremić, B., *Solution verification procedures for modeling and simulation of fully coupled porous media: static and dynamic behavior.* Coupled Systems Mechanics, 2015. 4: p. 67-98.

19. Gajo, A., Saetta, A., and Vitaliani, R., *Evaluation of 3-field and 2-field finite-element methods for the dynamic-response of saturated soil.* International Journal for Numerical Methods in Engineering, 1994. 37(7): p. 1231-1247.

20. Simon, B., Wu, J.-S., and Zienkiewicz, O., *Evaluation of higher order, mixed and \uppercaseHermitean finite element procedures for dynamic analysis of saturated porous media using one-dimensional models.* International Journal for Numerical and Analytical Methods in Geomechanics, 1986. 10(5): p. 483-499.

21. Gresho, P.M. and Sani, R.L., *On pressure boundary conditions for the incompressible \uppercasenavier-\uppercasestokes equations.* International Journal for Numerical Methods in Fluids, 1987. 7(10): p. 1111-1145.

22. Biot, M.A., *Theory of Propagation of Elastic Waves in a Fluid-Saturated Porous Solid. I. Low-Frequency Range.* The Journal of the Acoustical Society of America, 1956. 28(2): p. 168-178.

23. Chen, J. and Dargush, G., *Boundary element method for dynamic poroelastic and thermoelastic analyses.* International Journal of Solids and Structures, 1995. 32(15): p. 2257-2278.

24. Rice, J.R. *Elasticity of Fluid-Inflitrated Porous Solids (Poroelasticity)*. 1998 [cited 2016; Available from: http://esag.harvard.edu/rice/e2_Poroelasticity.pdf.

25. Diebels, S. and Ehlers, W., *Dynamic analysis of a fully saturated porous medium accounting for geometrical and material non-linearities.* International Journal for Numerical Methods in Engineering, 1996. 39(1): p. 81-97.

26. Lotfian, Z. and Sivaselvan, M.V., *Mixed finite element formulation for dynamics of porous media.* International Journal for Numerical Methods in Engineering, Submitted.

27. COMSOL Multiphysics v. 5.0, www.comsol.com, COMSOL AB, Stockholm, Sweden.

28. Geuzaine, C. and Remacle, J.-F., *Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities.* International Journal for Numerical Methods in Engineering, 2009. 79(11): p. 1309-1331.

APPENDIX A – SOURCE CODE FOR THREE-FIELD POROMECHANICS IMPLEMENTATION

Listing 1. `NewmarkPoreFluidAccelAux.h`

```
/****************************************************************/
/* MOOSE - Multiphysics Object Oriented Simulation Environment  */
/*                                                              */
/*          All contents are licensed under LGPL V2.1           */
/*             See LICENSE for full restrictions                */
/****************************************************************/


#ifndef NEWMARKPOREFLUIDACCELAUX_H
#define NEWMARKPOREFLUIDACCELAUX_H

#include "AuxKernel.h"


//Forward Declarations
class NewmarkPoreFluidAccelAux;

template<>
InputParameters validParams<NewmarkPoreFluidAccelAux>();

/**
 * Accumulate values from one auxiliary variable into another
 */
class NewmarkPoreFluidAccelAux : public AuxKernel
{
public:

  NewmarkPoreFluidAccelAux(const InputParameters & parameters);

  virtual ~NewmarkPoreFluidAccelAux() {}

protected:
  virtual Real computeValue();

  VariableValue & _w_old;  // W is Darcy velocity
  VariableValue & _w;
  Real _gamma;

};

#endif //NewmarkPoreFluidAccelAux_H
```

Listing 2. NewmarkPoreFluidAccelAux.C

```
/****************************************************************/
/* MOOSE - Multiphysics Object Oriented Simulation Environment  */
/*                                                              */
/*          All contents are licensed under LGPL V2.1           */
/*             See LICENSE for full restrictions                */
/****************************************************************/


#include "NewmarkPoreFluidAccelAux.h"

template<>
InputParameters validParams<NewmarkPoreFluidAccelAux>()
{
  InputParameters params = validParams<AuxKernel>();
    params.addRequiredCoupledVar("darcyvel","Darcy Velocity");
    params.addRequiredParam<Real>("gamma","gamma parameter");
  return params;
}

NewmarkPoreFluidAccelAux::NewmarkPoreFluidAccelAux(const InputParameters & parameters) :
  AuxKernel(parameters),
   _w_old(coupledValueOld("darcyvel")),
   _w(coupledValue("darcyvel")),
   _gamma(getParam<Real>("gamma"))
{
}

Real
NewmarkPoreFluidAccelAux::computeValue()
{
  if (!isNodal())
    mooseError("NewmarkPoreFluidAccelAux must run on a nodal variable");

  Real af_old = _u_old[_qp];
  if (_dt == 0)
    return af_old;
  return 1.0/_gamma*((_w[_qp]-_w_old[_qp])/_dt - af_old*(1.0-_gamma));
}
```

Listing 3. `ThreeFieldPoromechanicsApp.h`

```
#ifndef THREE_FIELD_POROMECHANICSAPP_H
#define THREE_FIELD_POROMECHANICSAPP_H

#include "MooseApp.h"

class ThreeFieldPoromechanicsApp;

template<>
InputParameters validParams<ThreeFieldPoromechanicsApp>();

class ThreeFieldPoromechanicsApp : public MooseApp
{
public:
  ThreeFieldPoromechanicsApp(const InputParameters & parameters);
  virtual ~ThreeFieldPoromechanicsApp();

  static void registerApps();
  static void registerObjects(Factory & factory);
  static void associateSyntax(Syntax & syntax, ActionFactory & action_factory);
};

#endif /* THREE_FIELD_POROMECHANICSAPP_H */
```

Listing 4. `ThreeFieldPoromechanicsApp.C`

```
#include "ThreeFieldPoromechanicsApp.h"
#include "Moose.h"
#include "AppFactory.h"

#include "NewmarkPoreFluidAccelAux.h"
#include "PoreFluidInertialForceCoupling.h"
#include "DynamicDarcyFlow.h"
#include "MassConservationNewmark.h"

#include "PorePressureBC.h"

template<>
InputParameters validParams<ThreeFieldPoromechanicsApp>()
{
  InputParameters params = validParams<MooseApp>();

  params.set<bool>("use_legacy_uo_initialization") = false;
  params.set<bool>("use_legacy_uo_aux_computation") = false;
  return params;
}

ThreeFieldPoromechanicsApp::ThreeFieldPoromechanicsApp(const InputParameters &
parameters) :
    MooseApp(parameters)
{
  srand(processor_id());

  Moose::registerObjects(_factory);
  ThreeFieldPoromechanicsApp::registerObjects(_factory);

  Moose::associateSyntax(_syntax, _action_factory);
  ThreeFieldPoromechanicsApp::associateSyntax(_syntax, _action_factory);
}

ThreeFieldPoromechanicsApp::~ThreeFieldPoromechanicsApp()
{
}

// External entry point for dynamic application loading
extern "C" void ThreeFieldPoromechanicsApp__registerApps() {
ThreeFieldPoromechanicsApp::registerApps(); }
void
ThreeFieldPoromechanicsApp::registerApps()
{
  registerApp(ThreeFieldPoromechanicsApp);
}
```

```
// External entry point for dynamic object registration
extern "C" void ThreeFieldPoromechanicsApp__registerObjects(Factory & factory) {
ThreeFieldPoromechanicsApp::registerObjects(factory); }
void
ThreeFieldPoromechanicsApp::registerObjects(Factory & factory)
{
  registerAux(NewmarkPoreFluidAccelAux);

  registerKernel(PoreFluidInertialForceCoupling);
  registerKernel(DynamicDarcyFlow);
  registerKernel(MassConservationNewmark);

  registerBoundaryCondition(PorePressureBC);
}

// External entry point for dynamic syntax association
extern "C" void ThreeFieldPoromechanicsApp__associateSyntax(Syntax & syntax,
ActionFactory & action_factory) { ThreeFieldPoromechanicsApp::associateSyntax(syntax,
action_factory); }
void
ThreeFieldPoromechanicsApp::associateSyntax(Syntax & syntax, ActionFactory &
action_factory)
{
}
```

Listing 5. `PoreFluidInertialForceCoupling.h`

```
/****************************************************************/
/* MOOSE - Multiphysics Object Oriented Simulation Environment  */
/*                                                              */
/*          All contents are licensed under LGPL V2.1           */
/*             See LICENSE for full restrictions                */
/****************************************************************/
#ifndef POREFLUIDINERTIALFORCECOUPLING_H
#define POREFLUIDINERTIALFORCECOUPLING_H

#include "Kernel.h"
#include "Material.h"

//Forward Declarations
class PoreFluidInertialForceCoupling;

template<>
InputParameters validParams<PoreFluidInertialForceCoupling>();

class PoreFluidInertialForceCoupling : public Kernel
{
public:

  PoreFluidInertialForceCoupling(const InputParameters & parameters);

protected:
  virtual Real computeQpResidual();

  virtual Real computeQpJacobian();
  virtual Real computeQpOffDiagJacobian(unsigned int jvar);

private:
  const MaterialProperty<Real> & _rhof; // fluid density
  const VariableValue & _af_old; // previous value of fluid acceleration
  const VariableValue & _w; // Darcy velocity
  const VariableValue & _w_old;
  unsigned int _w_var_num; // id of the Darcy vel variable
  const Real _gamma;

};
#endif //POREFLUIDINERTIALFORCECOUPLING_H
```

Listing 6. PoreFluidInertialForceCoupling.C

```
/****************************************************************/
/* MOOSE - Multiphysics Object Oriented Simulation Environment  */
/*                                                              */
/*          All contents are licensed under LGPL V2.1           */
/*             See LICENSE for full restrictions                */
/****************************************************************/
#include "PoreFluidInertialForceCoupling.h"
#include "SubProblem.h"

template<>
InputParameters validParams<PoreFluidInertialForceCoupling>()
{
  InputParameters params = validParams<Kernel>();
  params.set<bool>("use_displaced_mesh") = false;
  params.addRequiredCoupledVar("fluidaccel","fluid relative acceleration variable");
  params.addRequiredCoupledVar("darcyvel","Darcy velocity variable");
  params.addRequiredParam<Real>("gamma","gamma parameter");
  return params;
}

PoreFluidInertialForceCoupling::PoreFluidInertialForceCoupling(const InputParameters &
parameters)
  :Kernel(parameters),
   _rhof(getMaterialProperty<Real>("rhof")),
   _af_old(coupledValueOld("fluidaccel")),
   _w(coupledValue("darcyvel")),
   _w_old(coupledValueOld("darcyvel")),
   _w_var_num(coupled("darcyvel")),
   _gamma(getParam<Real>("gamma"))
{}

Real
PoreFluidInertialForceCoupling::computeQpResidual()
{
  if (_dt == 0)
    return 0.0;
  Real af=1/_gamma*((_w[_qp] - _w_old[_qp])/_dt - (1.0-_gamma)*_af_old[_qp]);
  return _test[_i][_qp]*_rhof[_qp]*af;
}

Real
PoreFluidInertialForceCoupling::computeQpJacobian()
{
  return 0.0;
}

Real
PoreFluidInertialForceCoupling::computeQpOffDiagJacobian(unsigned int jvar)
{
  if (_dt == 0)
    return 0.0;
  if (jvar != _w_var_num) // only u-w block is nonzero
    return 0.0;
  return _test[_i][_qp]*_rhof[_qp]/(_gamma*_dt)*_phi[_j][_qp];
}
```

Listing 7. `DynamicDarcyFlow.h`

```
/****************************************************************/
/* MOOSE - Multiphysics Object Oriented Simulation Environment  */
/*                                                              */
/*          All contents are licensed under LGPL V2.1           */
/*             See LICENSE for full restrictions                */
/****************************************************************/
#ifndef DYNAMICDARCYFLOW_H
#define DYNAMICDARCYFLOW_H

#include "Kernel.h"
#include "Material.h"

//Forward Declarations
class DynamicDarcyFlow;

template<>
InputParameters validParams<DynamicDarcyFlow>();

class DynamicDarcyFlow : public Kernel
{
public:

  DynamicDarcyFlow(const InputParameters & parameters);

protected:
  virtual Real computeQpResidual();

  virtual Real computeQpJacobian();
  virtual Real computeQpOffDiagJacobian(unsigned int jvar);

private:
  const MaterialProperty<Real> & _rhof; // fluid density
  const MaterialProperty<Real> & _nf; // porosity
  const MaterialProperty<Real> & _K; // hydraulic conductivity
  const VariableValue & _us; // skeleton displacement
  const VariableValue & _us_old;
  const VariableValue & _vs_old; // skeleton velocity
  const VariableValue & _as_old; // skeleton acceleration
  const VariableValue & _u_old; // Darcy velocity
                                // this is actually w, but is called u in this kernel
                                // because this the variable for this kernel
  const VariableValue & _af_old; // fluid relative acceleration
  unsigned int _us_var_num; // id of skeleton displacement variable
  const Real _gravity; // acceleration due to gravity
  const Real _beta;
  const Real _gamma;

};
#endif //DYNAMICDARCYFLOW_H
```

Listing 8. `DynamicDarcyFlow.C`

```
/****************************************************************/
/* MOOSE - Multiphysics Object Oriented Simulation Environment  */
/*                                                              */
/*          All contents are licensed under LGPL V2.1           */
/*             See LICENSE for full restrictions                */
/****************************************************************/
#include "DynamicDarcyFlow.h"
#include "SubProblem.h"

template<>
InputParameters validParams<DynamicDarcyFlow>()
{
  InputParameters params = validParams<Kernel>();
  params.set<bool>("use_displaced_mesh") = false;
  params.addRequiredCoupledVar("skeletondisp","skeleton displacement variable");
  params.addRequiredCoupledVar("skeletonvel","skeleton velocity variable");
  params.addRequiredCoupledVar("skeletonaccel","skeleton acceleration variable");
  params.addRequiredCoupledVar("fluidaccel","fluid relative acceleration variable");
  params.addRequiredParam<Real>("gravity","acceleration due to gravity");
  params.addRequiredParam<Real>("beta","beta parameter");
  params.addRequiredParam<Real>("gamma","gamma parameter");
  return params;
}

DynamicDarcyFlow::DynamicDarcyFlow(const InputParameters & parameters)
  :Kernel(parameters),
   _rhof(getMaterialProperty<Real>("rhof")),
   _nf(getMaterialProperty<Real>("porosity")),
   _K(getMaterialProperty<Real>("hydconductivity")),
   _us(coupledValue("skeletondisp")),
   _us_old(coupledValueOld("skeletondisp")),
   _vs_old(coupledValueOld("skeletonvel")),
   _as_old(coupledValueOld("skeletonaccel")),
   _u_old(valueOld()),
   _af_old(coupledValueOld("fluidaccel")),
   _us_var_num(coupled("skeletondisp")),
   _gravity(getParam<Real>("gravity")),
   _beta(getParam<Real>("beta")),
   _gamma(getParam<Real>("gamma"))
{}

Real
DynamicDarcyFlow::computeQpResidual()
{
  if (_dt == 0)
    return 0.0;
  Real as=1/_beta*(((_us[_qp]-_us_old[_qp])/(_dt*_dt)) - _vs_old[_qp]/_dt -
_as_old[_qp]*(0.5-_beta));
  Real af=1/_gamma*((_u[_qp]-_u_old[_qp])/_dt - (1.0-_gamma)*_af_old[_qp]);
  return _test[_i][_qp]*_rhof[_qp]*( as + af/_nf[_qp] + _gravity/_K[_qp]*_u[_qp] );
}
```

```
Real
DynamicDarcyFlow::computeQpJacobian()
{
  if (_dt == 0)
    return 0.0;
  return _test[_i][_qp]*_rhof[_qp]*(1.0/(_nf[_qp]*_gamma*_dt) +
_gravity/_K[_qp])*_phi[_j][_qp];
}

Real
DynamicDarcyFlow::computeQpOffDiagJacobian(unsigned int jvar)
{
  if (_dt == 0)
    return 0.0;
  if (jvar != _us_var_num) // only u-w block is nonzero
    return 0.0;
  return _test[_i][_qp]*_rhof[_qp]/(_beta*_dt*_dt)*_phi[_j][_qp];
}
```

Listing 9. `MassConservationNewmark.h`

```
/****************************************************************/
/* MOOSE - Multiphysics Object Oriented Simulation Environment  */
/*                                                              */
/*          All contents are licensed under LGPL V2.1           */
/*             See LICENSE for full restrictions                */
/****************************************************************/
#ifndef MASSCONSERVATIONNEWMARK_H
#define MASSCONSERVATIONNEWMARK_H

#include "Kernel.h"

//Forward Declarations
class MassConservationNewmark;

template<>
InputParameters validParams<MassConservationNewmark>();

class MassConservationNewmark : public Kernel
{
public:

  MassConservationNewmark(const InputParameters & parameters);

protected:
  virtual Real computeQpResidual();

  virtual Real computeQpJacobian();
  virtual Real computeQpOffDiagJacobian(unsigned int jvar);

private:
  unsigned int _ndisp; // number of displacement components (1D, 2D or 3D)
  unsigned int _ux_var; // id of displacement components
  unsigned int _uy_var;
  unsigned int _uz_var;
  VariableGradient &_grad_ux; // gradient of displacement
  VariableGradient &_grad_uy;
  VariableGradient &_grad_uz;
  VariableGradient &_grad_ux_old; // gradient of previous displacement
  VariableGradient &_grad_uy_old;
  VariableGradient &_grad_uz_old;
  VariableGradient &_grad_vx_old; // gradient of previous velocity
  VariableGradient &_grad_vy_old;
  VariableGradient &_grad_vz_old;
  VariableGradient & _grad_ax_old; // gradient of previous acceleration
  VariableGradient & _grad_ay_old;
  VariableGradient & _grad_az_old;
  const Real _beta;
  const Real _gamma;

};
#endif //MASSCONSERVATIONNEWMARK_H
```

Listing 10. `MassConservationNewmark.C`

```
/****************************************************************/
/* MOOSE - Multiphysics Object Oriented Simulation Environment  */
/*                                                              */
/*          All contents are licensed under LGPL V2.1           */
/*             See LICENSE for full restrictions                */
/****************************************************************/
#include "MassConservationNewmark.h"

template<>
InputParameters validParams<MassConservationNewmark>()
{
  InputParameters params = validParams<Kernel>();
  params.set<bool>("use_displaced_mesh") = false;
  params.addCoupledVar("displacements", "String of displacement components");
  params.addCoupledVar("velocities", "String of velocity components");
  params.addCoupledVar("accelerations", "String of acceleration components");
  params.addRequiredParam<Real>("beta","beta parameter");
  params.addRequiredParam<Real>("gamma","gamma parameter");
  return params;
}

MassConservationNewmark::MassConservationNewmark(const InputParameters & parameters)
  :Kernel(parameters),

  _ux_var(coupled("displacements",0)),
  _uy_var(_mesh.dimension() >= 2 ? coupled("displacements",1) : libMesh::invalid_uint),
  _uz_var(_mesh.dimension() == 3 ? coupled("displacements",2) : libMesh::invalid_uint),

  _grad_ux(coupledGradient("displacements",0)),
  _grad_uy(_mesh.dimension() >= 2 ? coupledGradient("displacements",1) : _grad_zero),
  _grad_uz(_mesh.dimension() == 3 ? coupledGradient("displacements",2) : _grad_zero),

  _grad_ux_old(coupledGradientOld("displacements",0)),
  _grad_uy_old(_mesh.dimension() >= 2 ? coupledGradientOld("displacements",1) :
_grad_zero),
  _grad_uz_old(_mesh.dimension() == 3 ? coupledGradientOld("displacements",2) :
_grad_zero),

  _grad_vx_old(coupledGradientOld("velocities",0)),
  _grad_vy_old(_mesh.dimension() >= 2 ? coupledGradientOld("velocities",1) : _grad_zero),
  _grad_vz_old(_mesh.dimension() == 3 ? coupledGradientOld("velocities",2) : _grad_zero),

  _grad_ax_old(coupledGradientOld("accelerations",0)),
  _grad_ay_old(_mesh.dimension() >= 2 ? coupledGradientOld("accelerations",1) :
_grad_zero),
  _grad_az_old(_mesh.dimension() == 3 ? coupledGradientOld("accelerations",2) :
_grad_zero),

   _beta(getParam<Real>("beta")),
   _gamma(getParam<Real>("gamma"))
{}
```

```
Real
MassConservationNewmark::computeQpResidual()
{
  if (_dt == 0)
    return 0.0;
  Real div_u = _grad_ux[_qp](0) + _grad_uy[_qp](1) + _grad_uz[_qp](2);
  Real div_u_old = _grad_ux_old[_qp](0) + _grad_uy_old[_qp](1) + _grad_uz_old[_qp](2);
  Real div_v_old = _grad_vx_old[_qp](0) + _grad_vy_old[_qp](1) + _grad_vz_old[_qp](2);
  Real div_a_old = _grad_ax_old[_qp](0) + _grad_ay_old[_qp](1) + _grad_az_old[_qp](2);
  Real div_v = (_gamma/_beta/_dt)*(div_u - div_u_old) + (1.0-_gamma/_beta)*div_v_old
             - (1.0 - _gamma/2.0/_beta)*div_a_old;
  return -_test[_i][_qp]*div_v;
}

Real
MassConservationNewmark::computeQpJacobian()
{
  // Derivative wrt p is zero
  return 0.0;
}

Real
MassConservationNewmark::computeQpOffDiagJacobian(unsigned int jvar)
{
  if (_dt == 0)
    return 0.0;
  else if (jvar == _ux_var)
    return -(_gamma/_beta/_dt)*_grad_phi[_j][_qp](0)*_test[_i][_qp];
  else if (jvar == _uy_var)
    return -(_gamma/_beta/_dt)*_grad_phi[_j][_qp](1)*_test[_i][_qp];
  else if (jvar == _uz_var)
    return -(_gamma/_beta/_dt)*_grad_phi[_j][_qp](2)*_test[_i][_qp];
  else
    return 0.0;
}
```

Listing 11. `PorePressureBC.h`

```
/****************************************************************/
/* MOOSE - Multiphysics Object Oriented Simulation Environment  */
/*                                                              */
/*          All contents are licensed under LGPL V2.1           */
/*             See LICENSE for full restrictions                */
/****************************************************************/
#ifndef POREPRESSUREBC_H
#define POREPRESSUREBC_H

#include "IntegratedBC.h"

// Forward Declarations
class PorePressureBC;

template<>
InputParameters validParams<PorePressureBC>();

class PorePressureBC : public IntegratedBC
{

public:
  PorePressureBC(const InputParameters &parameters);

  virtual ~PorePressureBC(){};

protected:

  virtual Real computeQpResidual();
  virtual Real computeQpJacobian();

private:

  const unsigned int _component;
  const Real _specified_pore_pressure;
};

#endif // POREPRESSUREBC_H
```

Listing 12. `PorePressureBC.C`

```
/****************************************************************/
/* MOOSE - Multiphysics Object Oriented Simulation Environment  */
/*                                                              */
/*          All contents are licensed under LGPL V2.1           */
/*             See LICENSE for full restrictions                */
/****************************************************************/
#include "PorePressureBC.h"

template<>
InputParameters validParams<PorePressureBC>()
{
  InputParameters params = validParams<IntegratedBC>();

  params.addRequiredParam<unsigned int>("component", "An integer corresponding to the
direction the variable this kernel acts in. (0 for x, 1 for y, 2 for z)");
  params.addRequiredParam<unsigned int>("porepressure", "Specified value of pore
pressure");

  return params;
}

PorePressureBC::PorePressureBC(const InputParameters &parameters)
    : IntegratedBC(parameters),
        _component(getParam<unsigned int>("component")),
      _specified_pore_pressure(getParam<Real>("porepressure"))
{
}

Real
PorePressureBC::computeQpResidual()
{
  return _test[_i][_qp]*_normals[_qp](_component)*_specified_pore_pressure;
}

Real
PorePressureBC::computeQpJacobian()
{
  return 0.0;
}
```

APPENDIX B – INPUT FILES FOR VERIFICATION EXAMPLE

Input file for dynamic analysis of skeleton alone

```
[Mesh]
    type = GeneratedMesh
    dim = 2
    nx = 20
    ny = 20
    xmin = 0
    xmax = 10
    ymin = 0
    ymax = 10
    elem_type = QUAD4
[]

[Variables]
    active = 'u_x u_y'

    [./u_x]
        order = FIRST
        family = LAGRANGE
    [../]
    [./u_y]
        order = FIRST
        family = LAGRANGE
    [../]
[]

[AuxVariables]
    [./v_x]
        order = FIRST
        family = LAGRANGE
    [../]

    [./v_y]
        order = FIRST
        family = LAGRANGE
    [../]

    [./a_x]
        order = FIRST
        family = LAGRANGE
    [../]

    [./a_y]
        order = FIRST
        family = LAGRANGE
    [../]
[]
```

```
[Kernels]
    active = 'stressdivx stressdivy inertia_x inertia_y'

    [./stressdivx]
        type = StressDivergenceTensors
        variable = u_x
        component = 0
        displacements = 'u_x u_y'
        use_displaced_mesh = false
    [../]

    [./stressdivy]
        type = StressDivergenceTensors
        variable = u_y
        component = 1
        displacements = 'u_x u_y'
        use_displaced_mesh = false
    [../]

    [./inertia_x]
        type = InertialForce
        variable = u_x
        velocity = v_x
        acceleration = a_x
        beta = 0.25
        gamma = 0.5
        use_displaced_mesh = false
    [../]

    [./inertia_y]
        type = InertialForce
        variable = u_y
        velocity = v_y
        acceleration = a_y
        beta = 0.25
        gamma = 0.5
        use_displaced_mesh = false
    [../]
[]

[AuxKernels]
    [./accel_x]
        type = NewmarkAccelAux
        variable = a_x
        displacement = u_x
        velocity = v_x
        beta = 0.25
        execute_on = timestep_end
    [../]

     [./accel_y]
        type = NewmarkAccelAux
        variable = a_y
        displacement = u_y
        velocity = v_y
        beta = 0.25
        execute_on = timestep_end
    [../]
```

```
     [./vel_x]
        type = NewmarkVelAux
        variable = v_x
        acceleration = a_x
        gamma = 0.5
        execute_on = timestep_end
    [../]

     [./vel_y]
        type = NewmarkVelAux
        variable = v_y
        acceleration = a_y
        gamma = 0.5
        execute_on = timestep_end
    [../]
[]

[Materials]
    [./elasticity_tensor]
        type = ComputeIsotropicElasticityTensor
        youngs_modulus = 1.0
        poissons_ratio = 0.3
        block = 0
    [../]

    [./strain]
        type = ComputeSmallStrain
        displacements = 'u_x u_y'
        block = 0
    [../]

    [./stress]
        type = ComputeLinearElasticStress
        block = 0
    [../]

    [./density]
        type = GenericConstantMaterial
        block = 0
        prop_names = 'density'
        prop_values = '0.1'
   [../]
[]

[Functions]
    active = 'bc_func'

    [./bc_func]
        type = ParsedFunction
        value = 'if(x<5.0,0.0,10.0)'
    [../]
[]
```

```
[BCs]
    [./bottom_y]
        type = PresetBC
        variable = u_y
        boundary = 'bottom'
        value = 0
    [../]

    [./top_y]
        type = Pressure
        variable = u_y
        boundary = 'top'
        component = 1 #y
        factor = 1.0
        function = bc_func
        use_displaced_mesh = false
    [../]

    [./left_x]
        type = PresetBC
        variable = u_x
        boundary = 'left'
        value = 0
    [../]

    [./right_x]
        type = PresetBC
        variable = u_x
        boundary = 'right'
        value = 0
    [../]
[]

[Executioner]
    type = Transient
    solve_type = 'PJFNK'
    l_max_its = 20
    nl_max_its = 10
    l_tol = 1.0e-7
    nl_rel_tol = 1.0e-12
    start_time = 0
    end_time = 100
    dtmax = 0.1
    dtmin = 0.1
    [./TimeStepper]
        type = ConstantDT
        dt = 0.1
    [../]
[]
[Outputs]
    exodus = true
    output_on = 'timestep_end'
    [./console]
        type = Console
        perf_log = true
        execute_on = 'initial timestep_end failed nonlinear'
    [../]
[]
```

GMSH geometry file for mesh generation

```
lc = DefineNumber[ 0.5, Name "Parameters/lc" ];
Point(1) = {0, 0, 0, lc};
Point(2) = {10, 0, 0, lc};
Point(3) = {10, 10, 0, lc};
Point(4) = {5, 10, 0, lc};
Point(5) = {0, 10, 0, lc};

Line(1) = {1, 2};
Line(2) = {2, 3};
Line(3) = {3, 4};
Line(4) = {4, 5};
Line(5) = {5, 1};

Line Loop(6) = {5, 1, 2, 3, 4};
Plane Surface(7) = {6};

Transfinite Line{1,2,5} = 21;
Transfinite Line{3, 4} = 11;
Transfinite Surface{7} = {1, 2, 3, 5};
Recombine Surface{7};

Physical Surface(8) = {7};
Physical Line(9) = {1};
Physical Line(10) = {2};
Physical Line(11) = {3};
Physical Line(12) = {4};
Physical Line(13) = {5};
```

Input file for dynamic analysis of porous medium

```
[Mesh]
    type = FileMesh
    file = test.msh
    block_id = '8'
    block_name = 'domain'
    boundary_id = '9 10 11 12 13'
    boundary_name = 'bottom right topright topleft left'
[]

[Variables]
    [./u_x]
        order = SECOND
        family = LAGRANGE
    [../]

    [./u_y]
        order = SECOND
        family = LAGRANGE
    [../]

    [./w_x]
        order = SECOND
        family = LAGRANGE
    [../]

    [./w_y]
        order = SECOND
        family = LAGRANGE
    [../]

    [./p]
        order = FIRST
        family = LAGRANGE
    [../]
[]

[AuxVariables]
    [./v_x]
        order = SECOND
        family = LAGRANGE
    [../]

    [./v_y]
        order = SECOND
        family = LAGRANGE
    [../]

    [./a_x]
        order = SECOND
        family = LAGRANGE
    [../]

    [./a_y]
        order = SECOND
        family = LAGRANGE
    [../]
```

```
    [./af_x]
        order = SECOND
        family = LAGRANGE
    [../]

    [./af_y]
        order = SECOND
        family = LAGRANGE
    [../]
[]

[Kernels]
    [./stressdiv_x]
        type = StressDivergenceTensors
        variable = u_x
        component = 0
        displacements = 'u_x u_y'
        # use_displaced_mesh = false
    [../]

    [./stressdiv_y]
        type = StressDivergenceTensors
        variable = u_y
        component = 1
        displacements = 'u_x u_y'
        # use_displaced_mesh = false
    [../]

    [./skeletoninertia_x]
        type = InertialForce
        variable = u_x
        velocity = v_x
        acceleration = a_x
        beta = 0.25
        gamma = 0.5
        use_displaced_mesh = false
    [../]

    [./skeletoninertia_y]
        type = InertialForce
        variable = u_y
        velocity = v_y
        acceleration = a_y
        beta = 0.25
        gamma = 0.5
        use_displaced_mesh = false
    [../]

    [./porefluidIFcoupling_x]
        type = PoreFluidInertialForceCoupling
        variable = u_x
        fluidaccel = af_x
        darcyvel = w_x
        gamma = 0.5
    [../]
```

```
[./porefluidIFcoupling_y]
    type = PoreFluidInertialForceCoupling
    variable = u_y
    fluidaccel = af_y
    darcyvel = w_y
    gamma = 0.5
[../]

[./darcyflow_x]
    type = DynamicDarcyFlow
    variable = w_x
    skeletondisp = u_x
    skeletonvel = v_x
    skeletonaccel = a_x
    fluidaccel = af_x
    gravity = 9.81
    beta = 0.25
    gamma = 0.5
[../]

[./darcyflow_y]
    type = DynamicDarcyFlow
    variable = w_y
    skeletondisp = u_y
    skeletonvel = v_y
    skeletonaccel = a_y
    fluidaccel = af_y
    gravity = 9.81
    beta = 0.25
    gamma = 0.5
[../]

[./poromechskeletoncoupling_x]
    type = PoroMechanicsCoupling
    variable = u_x
    porepressure = p
    component = 0
[../]

[./poromechskeletoncoupling_y]
    type = PoroMechanicsCoupling
    variable = u_y
    porepressure = p
    component = 1
[../]

[./poromechfluidcoupling_x]
    type = PoroMechanicsCoupling
    variable = w_x
    porepressure = p
    component = 0
[../]
[./poromechfluidcoupling_y]
    type = PoroMechanicsCoupling
    variable = w_y
    porepressure = p
    component = 1
[../]
```

```
    [./massconservationskeleton]
        type = MassConservationNewmark
        variable = p
        displacements = 'u_x u_y'
        velocities = 'v_x v_y'
        accelerations = 'a_x a_y'
        beta = 0.25
        gamma = 0.5
    [../]

    [./massconservationfluid]
        type = INSMass
        variable = p
        u = w_x
        v = w_y
        p = p
    [../]
[]

[AuxKernels]
    [./accel_x]
        type = NewmarkAccelAux
        variable = a_x
        displacement = u_x
        velocity = v_x
        beta = 0.25
        execute_on = timestep_end
    [../]

     [./accel_y]
        type = NewmarkAccelAux
        variable = a_y
        displacement = u_y
        velocity = v_y
        beta = 0.25
        execute_on = timestep_end
    [../]

     [./vel_x]
        type = NewmarkVelAux
        variable = v_x
        acceleration = a_x
        gamma = 0.5
        execute_on = timestep_end
    [../]

     [./vel_y]
        type = NewmarkVelAux
        variable = v_y
        acceleration = a_y
        gamma = 0.5
        execute_on = timestep_end
    [../]
```

```
    [./fluidaccel_x]
        type = NewmarkPoreFluidAccelAux
        variable = af_x
        darcyvel = w_x
        gamma = 0.5
        execute_on = timestep_end
    [../]

    [./fluidaccel_y]
        type = NewmarkPoreFluidAccelAux
        variable = af_y
        darcyvel = w_y
        gamma = 0.5
        execute_on = timestep_end
    [../]
[]

[Materials]
    [./elasticity_tensor]
        type = ComputeIsotropicElasticityTensor
        youngs_modulus = 14.5e6
        poissons_ratio = 0.3
        block = 'domain'
    [../]

    [./strain]
        type = ComputeSmallStrain
        displacements = 'u_x u_y'
        block = 'domain'
    [../]

    [./stress]
        type = ComputeLinearElasticStress
        block = 'domain'
    [../]

    [./density]
        type = GenericConstantMaterial
        block = 'domain'
        prop_names = density
        prop_values = 1986
    [../]

    [./rhof]
        type = GenericConstantMaterial
        block = 'domain'
        prop_names = rhof
        prop_values = 1000
    [../]

    [./porosity]
        type = GenericConstantMaterial
        block = 'domain'
        prop_names = porosity
        prop_values = 0.42
    [../]
```

```
    [./hydconductivity]
        type = GenericConstantMaterial
        block = 'domain'
        prop_names = hydconductivity
        prop_values = 0.0001
    [../]

    [./biotcoeff]
        type = GenericConstantMaterial
        block = 'domain'
        prop_names = biot_coefficient
        prop_values = 1.0
    [../]
[]

[Functions]
    active = 'bc_func'

    [./bc_func]
        type = ParsedFunction
        #value = 'if(x<5.0,0.0,15.0)*if(t<0.1,10*t,1.0)'
        value = 'if(t<0.1,10*t,1.0)'
    [../]
[]

[BCs]
    [./bottom_y]
        type = PresetBC
        variable = u_y
        boundary = 'bottom'
        value = 0
    [../]

    [./topright_y]
        type = Pressure
        variable = u_y
        boundary = 'topright'
        component = 1 #y
        factor = 15.0e3
        function = bc_func
        use_displaced_mesh = false
    [../]

    [./topleft_y]
        type = Pressure
        variable = u_y
        boundary = 'topleft'
        component = 1 #y
        factor = 0.0
        use_displaced_mesh = false
    [../]

    [./left_x]
        type = PresetBC
        variable = u_x
        boundary = 'left'
        value = 0
    [../]
```

```
    [./right_x]
        type = PresetBC
        variable = u_x
        boundary = 'right'
        value = 0
    [../]

    [./fluidbottom_y]
        type = PresetBC
        variable = w_y
        boundary = 'bottom'
        value = 0
    [../]

    [./fluidleft_x]
        type = PresetBC
        variable = w_x
        boundary = 'left'
        value = 0
    [../]

    [./fluidright_x]
        type = PresetBC
        variable = w_x
        boundary = 'right'
        value = 0
    [../]

    [./fluidtopright_y]
        type = PresetBC
        variable = w_y
        boundary = 'topright'
        value = 0
    [../]

    [./porepressure]
        type = PresetBC
        variable = p
        boundary = 'topleft'
        value = 0
    [../]
[]

[Preconditioning]
  [./smp]
    type = SMP
    full = true
    petsc_options_iname = '-pc_type -pc_factor_mat_solver_type -pc_factor_shift_type'
    petsc_options_value = 'lu umfpack NONZERO'
  [../]
[]

[Postprocessors]
   [./rightcornerdisp]
     type = PointValue
     point = '10 10 0'
     variable = u_y
   [../]
```

```
   [./leftcornerdisp]
     type = PointValue
     point = '0 10 0'
     variable = u_y
   [../]

   [./rightcornerdarcy]
     type = PointValue
     point = '10 10 0'
     variable = w_y
   [../]

   [./leftcornerdarcy]
     type = PointValue
     point = '0 10 0'
     variable = w_y
   [../]
[]


[Executioner]
    type = Transient
    solve_type = 'PJFNK'
    l_max_its = 1
    nl_max_its = 1
    l_tol = 1.0e-6
    nl_rel_tol = 0.1
    nl_abs_tol = 1.0
    start_time = 0
    end_time = 10.0
    dtmax = 0.002
    dtmin = 0.002
    [./TimeStepper]
        type = ConstantDT
        dt = 0.002
    [../]
[]

[Outputs]
    exodus = true
    output_on = 'timestep_end'
    active = 'csv'
    [./console]
        type = Console
        #perf_log = true
        execute_on = 'initial timestep_end failed nonlinear' # linear
    [../]
    [./csv]
        type = CSV
        execute_on = 'initial timestep_end'
    [../]
[]
```