Implementation of solid body stress analysis in OpenFOAM

Tang, Tian

Publication date:
2013

Document Version
Publisher's PDF, also known as Version of record

CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Project work:

Implementation of solid body stress analysis in OpenFOAM

Developed for OpenFOAM-1.6-ext

Author:
Tian Tang

Peer reviewed by:
Florian Vesting
Jelena Andric

November 6, 2012

Contents

Chapter 1

Introduction

During last decades, finite volume method (FVM) has been applied widely and successfully to solve the non-linear fluid flow problems. The fact that solid body mechanics and fluid mechanics share the same governing equation, and differ in constitutive relations only, gives a possibility of applying FVM to simulate solid body dynamics as well [1, 2]. Having the same discretization and approximation procedure for both solid body and fluid flow, it becomes much easier to model the interactions in multi-field problems, for instance, the seabed-wave-structure interaction in offshore engineering [3].

Motivated by the above ideas, this tutorial attempts to show practical implementations of solid body stress analysis in a CFD open source software OpenFOAM. Of course, it could not cover all aspects of the broad field of solid mechanics, therefore only a few topics will be discussed there, and the selection of these topics for inclusion is mainly influenced by the author's particular interests in soil mechanics. It is in a hope that this tutorial could provide the readers with some prospects of using OpenFOAM to solve solid body problems.

The content of this tutorial is arranged as follows. In Chapter 2, an available linear stress analysis solver `stressedFoam` distributed in OpenFOAM-1.6-ext is firstly reviewed. Afterwards, Chapter 3 describes the newly implemented `plasticStressedFoam`, which covers material non-linearity in solid body. And Chapter 4 develops the new `poroPlasticStressedFoam` solver, which performs the stress analysis in porous media. Moreover, Chapter 5 introduces the new block matrix solver, `coupledPoroFoam`, which provides implicit solutions of the pressure-displacement coupling in porous media. In the end, a general closure is given.

Chapter 2

The stressedFoam solver in OpenFOAM-1.6-ext

2.1 Theoretical background

stressedFoam is a linear stress analysis solver for elastic solid, applicable to both steady-state and transient problems. Thermal stress has also been included originally in the solver, so that users can choose to switch on/off the influence of temperature. However, this tutorial decides to focus on the isothermal mechanical stress, the part of temperature effects will therefore been neglected.

Considering a solid body element, the momentum balance states the following vector equations:

$$\frac{\partial^2(\rho\mathbf{u})}{\partial t^2} - \nabla\cdot\boldsymbol{\sigma} = \mathbf{0} \tag{2.1}$$

where $\mathbf{u}$ is the solid displacement vector, ρ is the density, and $\boldsymbol{\sigma}$ is the stress tensor. An assumption of free body force has been made.

In order to close the equation system, a constitutive relation has to been specified, and in the case of stressedFoam solver, it uses linear elasticity:

$$\boldsymbol{\sigma} = 2\mu\boldsymbol{\varepsilon} + \lambda tr(\boldsymbol{\varepsilon})\mathbf{I} \tag{2.2}$$

where $\mathbf{I}$ is the unit tensor, μ and λ are material properties called Lamé parameters.

And, the strain tensor $\boldsymbol{\varepsilon}$ is defined in terms of $\mathbf{u}$:

$$\boldsymbol{\varepsilon} = \frac{1}{2}[\nabla\mathbf{u} + (\nabla\mathbf{u})^T] \tag{2.3}$$

Combining Eq.(2.1-2.3), the governing equation for linear elastic solid body is written with the only unknown variable $\mathbf{u}$:

$$\frac{\partial^2(\rho\mathbf{u})}{\partial t^2} - \nabla\cdot\left[\mu\nabla\mathbf{u} + \underline{\mu(\nabla\mathbf{u})^T + \lambda\mathbf{I}tr(\nabla\mathbf{u})}\right] = \mathbf{0} \tag{2.4}$$

Notice that in the above vector equation, the three displacement components are coupled with each other through the underlined term $\mu(\nabla\mathbf{u})^T + \lambda\mathbf{I}tr(\nabla\mathbf{u})$. In order to handle this issue, the solution procedure in stressedFoam solver follows the so-called *segregated manner*, where each displacement component is solved separately and the inter-component coupling is treated explicitly. In order to recover the explicit coupling, several iterations are used.

After solving the displacement vector, it is then straightforward to obtain the strains and stress distributions in the solid body using Eq.(2.3-2.2).

2.2 A walk through stressedFoam

We will now explore how the stressedFoam solver is implemented in OpenFOAM to solve the above governing equation for elastic solid body. As a start point, Fig 2.1 below provides an overview of the structure of stressedFoam solver:

```
cd $FOAM_APP/solvers/stressAnalysis/stressedFoam
```

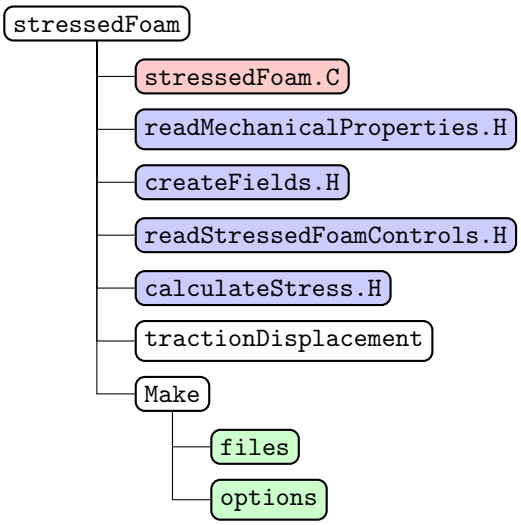


Figure 2.1: The file system of stressedFoam solver

- stressedFoam.C is the main file. Some header files are included in it. We will go through the codes in detail later and explain the included files as well when they are reached.
- The readMechanicalProperties.H file reads the input information of material parameters, e.g. μ and λ.
- The createFields.H file initializes the variable field of our interests, e.g. displacement vector **u**.
- The readStressedFoamControls.H file reads the input parameters for controlling the number of iterations and convergence tolerance.
- The calculateStress.H file is a post-process tool, which calculates the stress field based on Eq 2.3- 2.2.
- The tractionDisplacement directory contains a fixed displacement gradient boundary due to fixed traction, which is a very common boundary condition for stress analysis problem. We will discuss it separately in details in Section 2.3.
- The Make directory has the instructions for the wmake compilation command, including two files: files and options.

Now, let's focus on understanding the main file, stressedFoam.C. It starts with[1]:

[1] A small note for easier reading: codes in the main .C file all has a framebox, codes in included .H files is colored with blue without framebox.

```
#include "fvCFD.H"
```

where `fvCFD.H` contains all the class definitions about finite volume method, it is included from the following path:

`$FOAM_SRC/finiteVolume/lnInclude/fvCFD.H`

Afterwards, inside the `main` function there are case initializations (mesh, time, material parameters, variable fields, etc.):

```
int main(int argc, char *argv[])
{
#   include "setRootCase.H"
#   include "createTime.H"
#   include "createMesh.H"
#   include "readMechanicalProperties.H"
#   include "createFields.H"
```

where the first three files are included generally from OpenFOAM source library:

`$FOAM_SRC/OpenFOAM/lnInclude`

The other two files are included locally with specified case information. The `readMechanicalProperties.H` file simply contains the process of converting the Young's Modulus E and Poisson ratio ν (user inputs) into Lamé parameters μ, λ. The `createFields.H` file creates the displacement vector field by requiring an initialization from the user, and the calculated displacement values will be automatically written out:

```
// createFields.H
Info<< "Reading displacement field U\n" << endl;
volVectorField U
(
    IOobject
    (
        "U",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);
```

After the case initializations, we come to the time-loop:

```
    for (runTime++; !runTime.end(); runTime++)
    {
        Info<< "Iteration: " << runTime.timeName() << nl << endl;

#       include "readStressedFoamControls.H"

        int iCorr = 0;
        scalar initialResidual = 0;
```

Inside the loop, the included `readStressedFoamControls.H` file reads the user-defined convergence requirement and how many iterative corrections for the explicit treatment of inter-component coupling are wanted:

```
//  readStressedFoamControls.H
const dictionary& stressControl = mesh.solutionDict().subDict("stressedFoam");

int nCorr(readInt(stressControl.lookup("nCorrectors")));
scalar convergenceTolerance(readScalar(stressControl.lookup("U")));
```

Then we reach the core part of `stressedFoam`, which is solving the Eq 2.4 using a `do-while` loop:

```
do
      {
          volTensorField gradU = fvc::grad(U);
          fvVectorMatrix UEqn
          (
              fvm::d2dt2(U)
           ==
              fvm::laplacian(2*mu + lambda, U, "laplacian(DU,U)")

            + fvc::div
              (
                  mu*gradU.T() + lambda*(I*tr(gradU)) - (mu + lambda)*gradU,
                  "div(sigma)"
              )
          );

          initialResidual = UEqn.solve().initialResidual();

      } while (initialResidual > convergenceTolerance && ++iCorr < nCorr);
```

The namespace `fvm` denotes for implicit discretization, while `fvc` stands for explicit discretization. We had mentioned before that, the term of $\mu(\nabla \mathbf{u})^T + \lambda \mathbf{I} tr(\nabla \mathbf{u})$ contains the inter-component coupling and therefore shall be taken explicitly. However, the `stressedFoam` solver goes a bit further by doing the following rearrangement to improve the convergence:

$$\nabla \cdot \boldsymbol{\sigma} = \underbrace{\nabla \cdot (\mu \nabla \mathbf{u})}_{implicit} + \underbrace{\nabla \cdot [\mu(\nabla \mathbf{u})^T + \lambda \mathbf{I} tr(\nabla \mathbf{u})]}_{explicit} \tag{2.5}$$

$$= \underbrace{\nabla \cdot [(2\mu + \lambda)\nabla \mathbf{u}]}_{implicit} + \underbrace{\nabla \cdot [\mu(\nabla \mathbf{u})^T + \lambda \mathbf{I} tr(\nabla \mathbf{u}) - (\mu + \lambda)\nabla \mathbf{u}]}_{explicit} \tag{2.6}$$

This mainly helps to balance the implicit and explicit parts and give an equal coefficient matrix for all components of $\mathbf{u}$.

The `while` loop is providing an iterative prediction-correction process as illustrated in Fig 2.2 on next page.

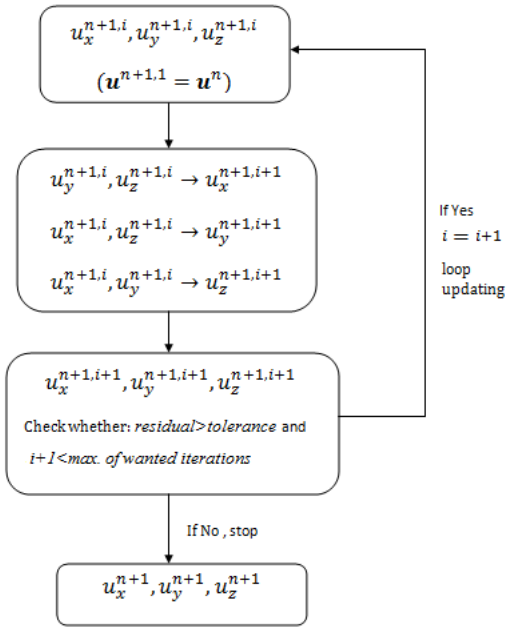


Figure 2.2: The `while` loop diagram.
where u_x, u_y, u_z are the displacement components, n stands for the time step, and i denotes the i^{th} iteration

After solving the converged displacement vector field, the stress is finally calculated through the included `calculateStress.H` file:

```
#       include "calculateStress.H"
```

where basically Eq 2.2 and 2.3 are used:

```
// calculateStress.H
    if (runTime.outputTime())
    {
        volTensorField gradU = fvc::grad(U);

        volSymmTensorField sigma =
            rho*(2.0*mu*symm(gradU) + lambda*I*tr(gradU));
        ... // some more less important codes are hidden
    }
```

In the end of `stressedFoam.C`, there is output information regarding the elapsed time in each time step and finally we close the `runTime` loop:

```
        Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
            << "  ClockTime = " << runTime.elapsedClockTime() << " s"
            << nl << endl;
    }

    Info<< "End\n" << endl;

    return(0);
}
```

So far, we have gone through `stressedFoam.C`. To summarize it in short, there is a while loop to solve the inter-component coupled governing equation (Eq 2.4) and then an included `calculateStress.H` file to obtain the corresponding stresses.

2.3 The `tractionDisplacement` boundary

In case of solid stress problems, we usually encounter a boundary condition with fixed traction force ($\mathbf{T}$), which can be expressed as:

$$\mathbf{T} = \boldsymbol{\sigma} \cdot \mathbf{n} \tag{2.7}$$

where $\mathbf{n}$ is the surface normal on the boundary.
However, since the displacement $\mathbf{u}$ is the computing variable in `stressedFoam` solver, we have to convert the traction force into displacement on the boundary:

$$\mathbf{T} = \boldsymbol{\sigma} \cdot \mathbf{n} = \left[\mu \nabla \mathbf{u} + \mu (\nabla \mathbf{u})^T + \lambda \mathbf{I} tr(\nabla \mathbf{u})\right] \cdot \mathbf{n} \tag{2.8}$$

$$= [\underbrace{(2\mu + \lambda)\nabla \mathbf{u}}_{implicit} + \underbrace{\mu (\nabla \mathbf{u})^T + \lambda \mathbf{I} tr(\nabla \mathbf{u}) - (\mu + \lambda)\nabla \mathbf{u}}_{explicit}] \cdot \mathbf{n} \tag{2.9}$$

$$\rightarrow (\nabla \mathbf{u}) \cdot \mathbf{n} = \frac{\mathbf{T} - [\mu (\nabla \mathbf{u})^T + \lambda \mathbf{I} tr(\nabla \mathbf{u}) - (\mu + \lambda)\nabla \mathbf{u}] \cdot \mathbf{n}}{2\mu + \lambda} \tag{2.10}$$

As a result, the above equation gives a displacement gradient boundary (Neumann type), which can be derived from the given traction force and the explicitly treated terms on the right hand side. This process has been implemented as the `tractionDisplacement` boundary in `stressedFoam`. In the main file `tractionDisplacementFvPatchVectorField.C`, it is the following part which updates the boundary value:

```
void tractionDisplacementFvPatchVectorField::updateCoeffs()
{
    ...  //here some less important codes are ignored

    vectorField n = patch().nf();

    const fvPatchField<tensor>& gradU =
        patch().lookupPatchField<volTensorField, tensor>("grad(U)");

    gradient() =
    (
        (traction_ - pressure_*n)/rho.value()
      - (n & (mu.value()*gradU.T() - (mu + lambda).value()*gradU))
      - n*tr(gradU)*lambda.value()
    )/(2.0*mu + lambda).value();

    fixedGradientFvPatchVectorField::updateCoeffs();
}
```

It is worth mentioning a tricky part of understanding this implementations. Namely in Eq 2.10, only one traction force T is specified, while in the implementation, we have `(traction_ - pressure_*n)`. The explanation is that the `traction_` term in the codes is exactly the T in the equation, while the term of `pressure_*n` actually is just one extra input which helps to specify a normal traction force on the boundary, i.e. a scalar. Thus for general traction force condition, we specify the `traction_`, and set `pressure_` $= 0$, while for a normal traction force condition, we set `traction_` $= (\ 0 \quad 0 \quad 0\ )$ and specify a `pressure_` value.

2.4 Run a plateHole case

Having studied the main solver file and the boundary condition, we will now run the test case called `plateHole`, which is a square plate with a circular hole at its centre loaded by horizontal tension force on the right hand side:

```
cp -r $FOAM_TUTORIALS/stressAnalysis/stressedFoam/plateHole $FOAM_RUN
cd $FOAM_RUN/plateHole
blockMesh
stressedFoam
touch output.foam
paraview output.foam
```

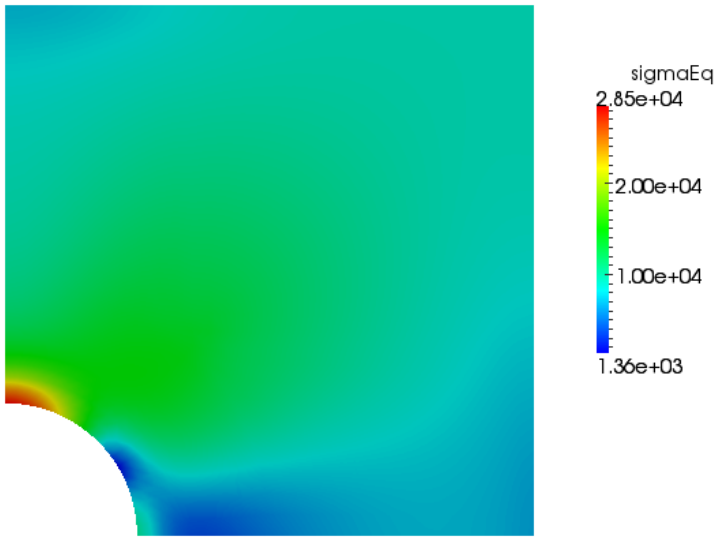


Figure 2.3: The equivalent stress field in the plate
$\sigma_{Eq} = \sqrt{\frac{3}{2}\mathbf{s} : \mathbf{s}}$, $\mathbf{s}$ = deviatoric stress

Here, only a quarter of the plate is simulated due to symmetry. The comparison between the simulation and the analytical solution has been done by Jasak & Weller, 2000. According to their result, the calculation of `stressedFoam` shows remarkable accuracy.

Chapter 3

The `plasticStressedFoam` solver

The `stressedFoam` solver is based on linear elastic theory of solid mechanics. Most processes of solid mechanics in nature, however, are highly non-linear. In this section, we will therefore explore implementation of material non-linearity, i.e. the plasticity in solid body. The new implemented solver is named as `plasticStressedFoam`.

In `plasticStressedFoam` solver, the simplest perfect plasticity is considered. Perfect plasticity is defined as: there exists a maximum level of stress that the solid body can handle elastically, once the maximum stress level is reached, the solid will undergo irreversible deformation without any increase in stresses or loads.

3.1 Mathematical model

After adding plastic feature into the solid, the steady-state governing equation is expressed in the following incremental form:

$$\nabla \cdot \left\{ \mu \nabla(d\mathbf{u}) + \mu[\nabla(d\mathbf{u})]^T + \lambda \mathbf{I} tr[\nabla(d\mathbf{u})] - \underbrace{[2\mu(d\boldsymbol{\varepsilon}_p) + \lambda \mathbf{I} tr\,(d\boldsymbol{\varepsilon}_p)]}_{plasticity} \right\} = \mathbf{0} \tag{3.1}$$

where $d\mathbf{u}$ is the incremental displacement vector, and $d\boldsymbol{\varepsilon}_p$ stands for the incremental plastic strain tensor, which is stress-level dependent: when the stress is below the limit level, only elasticity occurs, $d\boldsymbol{\varepsilon}_p = \mathbf{0}$; while the limit stress level is reached, plasticity occurs, $d\boldsymbol{\varepsilon}_p$ shall then be computed from the stress $\boldsymbol{\sigma}$, and the calculation procedure is described in Appendix A.

To solve Eq 3.1, it is needed to split the terms into implicit and explicit parts. In this case, we will do the following split:

$$\underbrace{\nabla \cdot [(2\mu + \lambda)\nabla(d\mathbf{u})]}_{implicit} = \underbrace{\overbrace{-\nabla \cdot \left\{ \mu[\nabla(d\mathbf{u})]^T + \lambda \mathbf{I} tr[\nabla(d\mathbf{u})] - (\mu + \lambda)\nabla(d\mathbf{u}) \right\}}^{inter-component \quad coupling} + \overbrace{\nabla \cdot [2\mu(d\boldsymbol{\varepsilon}_p) + \lambda \mathbf{I} tr\,(d\boldsymbol{\varepsilon}_p)]}^{plastic \quad terms}}_{explicit} \tag{3.2}$$

3.2 Let's implement it

We will now start to develop our new perfect-plastic solid stress analysis solver based on `stressedFoam`.

Inside your local `solvers` directory, simply copy the original `stressedFoam` solver here, change the corresponding file names, and then compile the solver so that we make sure the basic setup is correct:

```
cp -r $FOAM_APP/solvers/stressAnalysis/stressedFoam .
mv stressedFoam plasticStressedFoam
cd plasticStressedFoam
mv stressedFoam.C plasticStressedFoam.C
sed -i 's/stressedFoam/plasticStressedFoam/g' Make/files
sed -i 's/FOAM_APPBIN/FOAM_USER_APPBIN/g' Make/files
wclean
wmake
```

Once it compiles successfully, we can then start to modify the content of the solver for our own sake. The new `plasticStressedFoam` solver is supposed to have several features listed below:

- The primitive variable becomes incremental displacement $d\mathbf{u}$, instead of $\mathbf{u}$. Moreover, two fields have to be created, i.e. the incremental plastic strain $d\boldsymbol{\varepsilon}_p$, and the stress $\boldsymbol{\sigma}$;
- A material property, e.g. a yield stress parameter is required to represent the limit stress level;
- The plastic terms are added into the governing equation under explicit discretization.
- Both the stress $\boldsymbol{\sigma}$ and incremental plastic strain $d\boldsymbol{\varepsilon}_p$ have to be updated in each time step after solving the governing equation.

We will now implement these features one by one. Firstly, we modify the `createFields.H` file so as to create fields of $d\mathbf{u}$, $\boldsymbol{\sigma}$, $d\boldsymbol{\varepsilon}_p$. The initial and boundary values of $d\mathbf{u}$ will be read from the user. The initial displacement field $\mathbf{u}$ is set equal to the first displacement increment. The initial stress field $\boldsymbol{\sigma}$ can be specified by the user. And if assume no plasticity will happen in the very beginning, so $d\boldsymbol{\varepsilon}_p = \mathbf{0}$.

```
// createFields.H
    Info<< "Reading incremental displacement field dU\n" << endl;
    volVectorField dU
    (
        IOobject
        (
            "dU",
            runTime.timeName(),
            mesh,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh
    );
    Info<< "Reading displacement field dU\n" << endl;
    volVectorField U
    (
        IOobject
        (
            "U",
            runTime.timeName(),
            mesh,
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),
        dU
    );
    Info<< "Reading stress field sigma\n" << endl;
    volSymmTensorField sigma
```

```
(
    IOobject
    (
        "sigma",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);
Info<< "Reading incremental plastic strain field deps_p\n" << endl;
volTensorField deps_p
(
    IOobject
    (
        "deps_p",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    mesh,
    dimensionedTensor("deps_p", dimless, tensor::zero)
);
```

Then, we are requiring a generalized yield stress material parameter: k_y, by adding one line in the `readMechanicalProperties.H` file:

```
dimensionedScalar ky(mechanicalProperties.lookup("ky"));
```

Notice that for this steady-state solver, the density ρ is no longer needed, neither the normalization of Young's Modulus E is required. Therefore we can delete the corresponding lines:

```
dimensionedScalar rho(mechanicalProperties.lookup("rho"));
dimensionedScalar rhoE(mechanicalProperties.lookup("E"));

Info<< "Normalising E : E/rho\n" << endl;
dimensionedScalar E = rhoE/rho;
```

And, we keep the Young's Modulus E in its original form:

```
dimensionedScalar E(mechanicalProperties.lookup("E"));
```

Now, we can modify the main `plasticStressedFoam.C` file to implement the incremental governing equation. We firstly change the variable $\mathbf{u}$ into $d\mathbf{u}$ by typing the following command line in the terminal:

```
sed -i 's/U/dU/g' plasticStressedFoam.C
```

We also rename the included `readStressedFoamControls.H` so as to be consistent:

```
mv readStressedFoamControls.H readPlasticStressedFoamControls.H
sed -i 's/stressedFoam/plasticStressedFoam/g' readPlasticStressedFoamControls.H
sed -i 's/U/dU/g' readPlasticStressedFoamControls.H
sed -i 's/stressedFoamControls/plasticStressedFoamControls/g' plasticStressedFoam.C
```

Then, we could edit the main `.C` file manually to remove the transient term (the second time derivative), add the explicit plastic term, and update the total displacement after solving the incremental displacement. The final modification would be as follows:

```
    do
    {
        volTensorField gradU = fvc::grad(dU);

        fvVectorMatrix dUEqn
        (
            fvm::laplacian(2*mu + lambda, dU, "laplacian(DdU,dU)")

          ==

          - fvc::div
            (
                mu*gradU.T() + lambda*(I*tr(gradU)) - (mu + lambda)*gradU,
                "div(sigmaExp)"
            )

          + fvc::div
            (
               2*mu*deps_p + lambda*I*tr(deps_p),
               "div(sigmaP)"
            )
        );

        initialResidual = dUEqn.solve().initialResidual();

    } while (initialResidual > convergenceTolerance && ++iCorr < nCorr);

    U += dU;
```

So far we haven't implemented the information about how to update the stress $\boldsymbol{\sigma}$ and incremental plastic strain $d\boldsymbol{\varepsilon}_p$. This will be done in the included file `calculateStress.H`.

Firstly, we shall discard the original process of calculating the stress and strain by pure elasticity, shown in below:

```
volTensorField gradU = fvc::grad(U);

volSymmTensorField sigma =
     rho*(2.0*mu*symm(gradU) + lambda*I*tr(gradU));
```

Instead, we will write a new procedure of calculating the elasto-plastic stress and the incremental plastic strain as follows (details refer to the radial return algorithm in Appendix A):

```
 volTensorField graddU = fvc::grad(dU);

 volSymmTensorField sigma_old = sigma;

//Get the trial updated stress
   sigma += 2.0*mu*symm(graddU) + lambda*I*tr(graddU);

//Check the yield condition
   volScalarField sqrtJ2 = sqrt((1.0/2.0)*magSqr(dev(sigma)));
```

```
volScalarField fac = sqrtJ2/k;

forAll(fac, celli)
{
    if (fac[celli] > 1.0)         //Plasticity occurs
     {
        sigma[celli] = 1.0/3.0*I*tr(sigma[celli]) + dev(sigma[celli])/fac[celli];
        symmTensor dsigma = sigma[celli] - sigma_old[celli];
        tensor deps_e = 1.0/3.0*I*tr(dsigma)/(3.0*lambda+2.0*mu).value()
                        + dev(dsigma)/(2.0*mu.value());
        tensor deps = 1.0/2.0*(graddU[celli] + graddU[celli].T());
        deps_p[celli] = deps - deps_e;
     }
     else                         // only elasticity
     {
        deps_p[celli] = tensor::zero;
     }
}
```

At this stage, we have completed the implementation of all new features that the `plasticStressedFoam` solver should have. Type `wmake` to compile the solver, then we are completed.

3.3 Modify the `tractionDisplacement` boundary

Now the `plasticStressedFoam` solver is ready, we still need to think about the boundary condition. We can use those standard boundary conditions in OpenFOAM, such as `fixedValue` and `fixedGradient` for our incremental displacement boundary. We could also modify the `tractionDisplacement` boundary to make it suitable for incremental displacement as well. Recall Eq 2.7, 3.1, 3.2:

$$\mathbf{T} = \boldsymbol{\sigma} \cdot \mathbf{n} \rightarrow d\mathbf{T} = d\boldsymbol{\sigma} \cdot \mathbf{n} \tag{3.3}$$

$$d\boldsymbol{\sigma} = \underbrace{(2\mu + \lambda)\nabla(d\mathbf{u})}_{\text{implicit}} + \underbrace{\mu[\nabla(d\mathbf{u})]^T + \lambda\mathbf{I}tr[\nabla(d\mathbf{u})] - (\mu + \lambda)\nabla(d\mathbf{u})}_{\text{inter-component coupling, explicit}} - \underbrace{[2\mu(d\boldsymbol{\varepsilon}_p) + \lambda\mathbf{I}tr\,(d\boldsymbol{\varepsilon}_p)]}_{\text{plasticity, explicit}} \tag{3.4}$$

Combine Eq 3.3-3.4:

$$(\nabla d\mathbf{u}) \cdot \mathbf{n} = \frac{d\mathbf{T} - \left\{\mu[\nabla(d\mathbf{u})]^T + \lambda\mathbf{I}tr[\nabla(d\mathbf{u})] - (\mu + \lambda)\nabla(d\mathbf{u}) - [2\mu(d\boldsymbol{\varepsilon}_p) + \lambda\mathbf{I}tr\,(d\boldsymbol{\varepsilon}_p)]\right\} \cdot \mathbf{n}}{2\mu + \lambda} \tag{3.5}$$

We will try to implement the above equation as a new `incrementTractionDisplacement` boundary. As a start, reuse the original `tractionDisplacement` source:

```
mv tractionDisplacement incrementTractionDisplacement
cd incrementTractionDisplacement
mv tractionDisplacementFvPatchVectorField.H incrementTractionDisplacementFvPatchVectorField.H
mv tractionDisplacementFvPatchVectorField.C incrementTractionDisplacementFvPatchVectorField.C
sed -i 's/traction/incrementTraction/g' incrementTractionDisplacementFvPatchVectorField*
sed -i 's/pressure/incrementPressure/g' incrementTractionDisplacementFvPatchVectorField*
```

Then, we only need to modify the `updateCoeffs()` function as follows:

```
 void incrementTractionDisplacementFvPatchVectorField::updateCoeffs()
{
    ... //here some unchanged lines are ignored
```

```
    const fvPatchField<tensor>& deps_p =
        patch().lookupPatchField<volTensorField, tensor>("deps_p");

    gradient() =
    (
        (incrementTraction_ - incrementPressure_*n)
      - (n & (mu.value()*graddU.T() - (mu + lambda).value()*graddU))
      - n*tr(graddU)*lambda.value()
      + (n & (2*mu.value()*deps_p))
      + n*tr(deps_p)*lambda.value()
    )/(2.0*mu + lambda).value();

    fixedGradientFvPatchVectorField::updateCoeffs();
}
```

Finally, we try to update the `Make` file so that the new `incrementTractionDisplacement` boundary can be compiled and used. In the terminal, type the following command:

```
sed -i 's/traction/incrementTraction/g' Make/files
sed -i 's/traction/incrementTraction/g' Make/options
wmake
```

Now, the `incrementTractionDisplacement` boundary is also ready.

3.4 Test the solver

In order to know whether the new `plasticStressedFoam` solver and the `incrementTractionDisplacement` boundary condition can work properly or not, and if not what went wrong in our implementations, it's necessary and helpful to set up a test case and give a run.

Hence, we will create a `plasticPlateHole` case based on the original `plateHole` case. The following changes have to be done in general:

- Rename the file `0/U` to `0/dU`, and replace the `TractionDisplacement` boundary by the new `incrementTractionDisplacement` boundary, also change the keywords for `traction` and `pressure` to `incrementTraction` and `incrementPressure`;
- Create a new file `0/sigma` to specify the initial stress field;
- Add one line for the yield parameter k_y in the `constant/mechanicalProperties` file;
- Use `sed` command to replace all `U` by `dU`, and `stressedFoam` by `plasticStressedFoam` in all the files under the `system` folder;
- Add the discretisation method for the plastic terms (`div(sigmaP)`) in the `system/fvSchemes` file under `divSchemes{}`, e.g.:
  `div(sigmaP)  Gauss linear;`

The details of the codes for the `plasticPlateHole` case can be found in Appendix B.

Once the case set-up is completed, we can then give it a run by typing:

```
blockMesh
plasticStressedFoam
```

Fig 3.1 illustrates some results from the simulation.

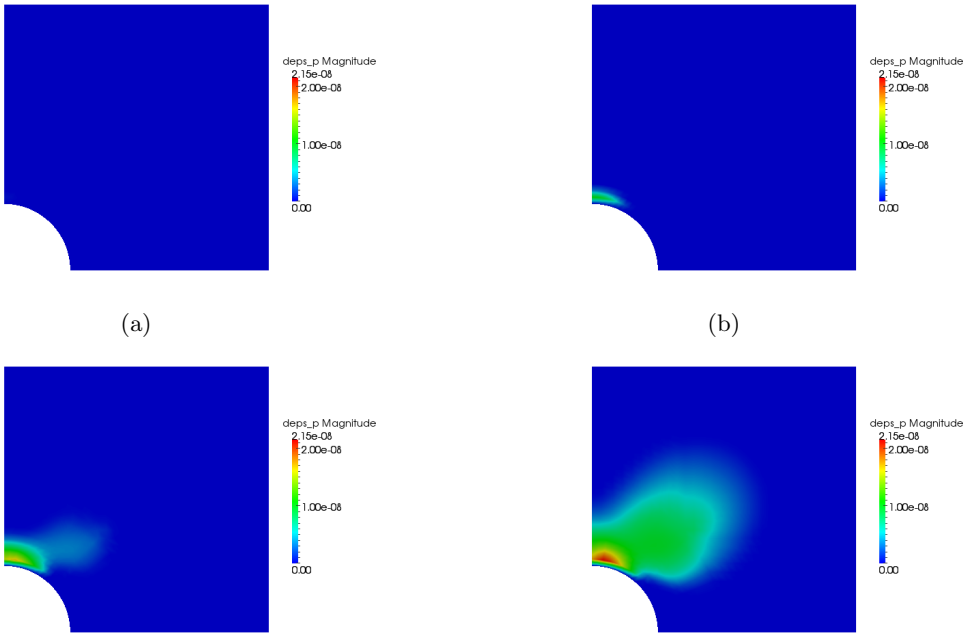


Figure 3.1: The development of plastic zones, represented by `mag(deps_p)`. the test condition is $d\mathbf{T} = 10^3 Pa, k = 10^3 Pa$

Chapter 4

The `poroPlasticStressedFoam` solver

Often we might need to investigate the stress conditions in some porous solid bodies, for instance, the soil in geotechnical engineering, which is consisted of soil grain particles and the pore fluids (e.g. water and air). In this section, we will focus on developing a stress analysis solver for soil saturated by water.

4.1 Extend Biot's equation

The governing equations for the porous soil media is usually based on Biot's consolidation theory, where the soil skeleton deforms elastically and the pore water flows as Darcy's flow:

$$\nabla \cdot \left[\mu \nabla \mathbf{u} + \mu (\nabla \mathbf{u})^T + \lambda \mathbf{I} tr(\nabla \mathbf{u})\right] = \nabla p \tag{4.1}$$

$$\frac{k}{\gamma}\nabla^2 p = \frac{n}{K'}\frac{\partial p}{\partial t} + \frac{\partial}{\partial t}(\nabla \cdot \mathbf{u}) \tag{4.2}$$

where p is the pore water pressure, k is the the permeability coefficient, γ is the specific weight of water,n is the porosity, and K' is the pore water modulus.

If we would instead like to simulate the soil skeleton as a perfect-plastic material, we can extend Eq 4.1 into the following incremental form:

$$\nabla \cdot \left\{ \mu \nabla (d\mathbf{u}) + \mu [\nabla (d\mathbf{u})]^T + \lambda \mathbf{I} tr[\nabla (d\mathbf{u})] - \underbrace{[2\mu (d\boldsymbol{\varepsilon}_p) + \lambda \mathbf{I} tr\,(d\boldsymbol{\varepsilon}_p)]}_{plasticity} \right\} = \nabla (dp) \tag{4.3}$$

Now we will implement a new solver named as `poroPlasticStressedFoam` to solve the above governing equations Eq 4.2-4.3.

4.2 Develop the `poroPlasticStressedFoam` solver

Let us firstly gather the new features of `poroPlasticStressedFoam`:

- There is one more variable, i.e. the pore water pressure p, and one more scalar equation Eq 4.2 for p. Also, noticed that the displacement vector $\mathbf{u}$ and pore pressure p are coupled in the equations, therefore we must do some splits of implicit and explicit parts to solve the equations:

$$\underbrace{\frac{k}{\gamma}\nabla^2 p}_{\text{implicit}} = \underbrace{\frac{n}{K'}\frac{\partial p}{\partial t}}_{\text{implicit}} + \underbrace{\frac{\partial}{\partial t}(\nabla \cdot \mathbf{u})}_{\mathbf{u}\text{-coupling, explicit}} \tag{4.4}$$

$$\underbrace{\nabla \cdot [(2\mu + \lambda)\nabla(d\mathbf{u})]}_{implicit} = \underbrace{-\nabla \cdot \left\{\mu[\nabla(d\mathbf{u})]^T + \lambda \mathbf{I} tr[\nabla(d\mathbf{u})] - (\mu + \lambda)\nabla(d\mathbf{u})\right\}}_{\text{inter-component coupling, explicit}} \tag{4.5}$$

$$+ \underbrace{\nabla \cdot [2\mu(d\boldsymbol{\varepsilon}_p) + \lambda \mathbf{I} tr\,(d\boldsymbol{\varepsilon}_p)]}_{\text{plasticity, explicit}} + \underbrace{\nabla(dp)}_{p\text{-coupling, explicit}} \tag{4.6}$$

- The momentum equation, i.e. Eq 4.3 is formulated incrementally, while Eq. 4.2 is stated in total form. Some conversions from increments to total values are therefore necessary during the solution procedure.

We can then start the implementations. Inside the local `solver` folder, copy the previous `plasticStressedFoam` and do the basic ste-up as before:

```
cp -r plasticStressedFoam poroPlasticStressedFoam
cd poroPlasticStressedFoam
mv plasticStressedFoam.C poroPlasticStressedFoam.C
sed -i 's/plasticStressedFoam/poroPlasticStressedFoam/g' Make/files
wclean
wmake
```

Now, we are ready to modify the solver content. Firstly, we create a scalar field called `p` in the `createFields.H` file:

```
Info<< "Reading pore pressure field p\n" << endl;
    volScalarField p
    (
        IOobject
        (
            "p",
            runTime.timeName(),
            mesh,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh
    );
```

Then we add those extra material parameters in the `readMechanicalProperties.H` file:

```
    dimensionedScalar  k(mechanicalProperties.lookup("k"));
    dimensionedScalar  gamma(mechanicalProperties.lookup("gamma"));
    dimensionedScalar  n(mechanicalProperties.lookup("n"));
    dimensionedScalar  Kprime(mechanicalProperties.lookup("Kprime"));

    dimensionedScalar Dp1 = k/gamma*(Kprime/n);
    dimensionedScalar Dp2= Kprime/n;
```

We also rename the included `readPlasticStressedFoamControls.H` file:

```
mv readPlasticStressedFoamControls.H readPoroPlasticStressedFoamControls.H
sed -i 's/plasticStressedFoam/poroPlasticStressedFoam/g' readPoroPlasticStressedFoamControls.H
sed -i 's/readPlasticStressed/readPoroPlasticStressed/g' poroPlasticStressedFoam.C
```

Since we are solving both the momentum equation and the pore water flow equation, we might want different convergence tolerances for them. Therefore in the `readPoroPlasticStressedFoamControls.H` file we write the following lines:

```
    scalar pTolerance(readScalar(stressControl.lookup("p")));
    scalar dUTolerance(readScalar(stressControl.lookup("dU")));
```

Now we implement the *segregated* approach of solving Eq 1.19 and 1.21 in a `do-while` loop as shown in codes below:

```
int iCorr = 0;
scalar pResidual = 0;
scalar dUResidual = 0;

volScalarField p_old = p;
volVectorField U_old = U;

do
{
                fvScalarMatrix pEqn
    (
     fvm::ddt(p) == fvm::laplacian(Dp1, p, "laplacian(Dp1,p)")
                  - fvc::div(fvc::ddt(Dp2, U), "div(ddt(U))")
    );
    pResidual = pEqn.solve().initialResidual();
    volScalarField dp = p - p_old;

    volTensorField graddU = fvc::grad(dU);
    fvVectorMatrix dUEqn
    (
        fvm::laplacian(2*mu + lambda, dU, "laplacian(DdU,dU)")
     ==
      - fvc::div
        (
            mu*graddU.T() + lambda*(I*tr(graddU)) - (mu + lambda)*graddU,
            "div(sigmaExp)"
        )
      + fvc::div
        (
            2.0*mu*deps_p + lambda*I*tr(deps_p),
            "div(sigmaP)"
        )
      + fvc::grad(dp)
    );
    dUResidual = dUEqn.solve().initialResidual();
    U = U_old + dU;

} while ((pResidual > pTolerance || dUResidual > dUTolerance)
         && ++iCorr < nCorr);
```

Fig 4.1 below helps to understand the solution procedure more clear.

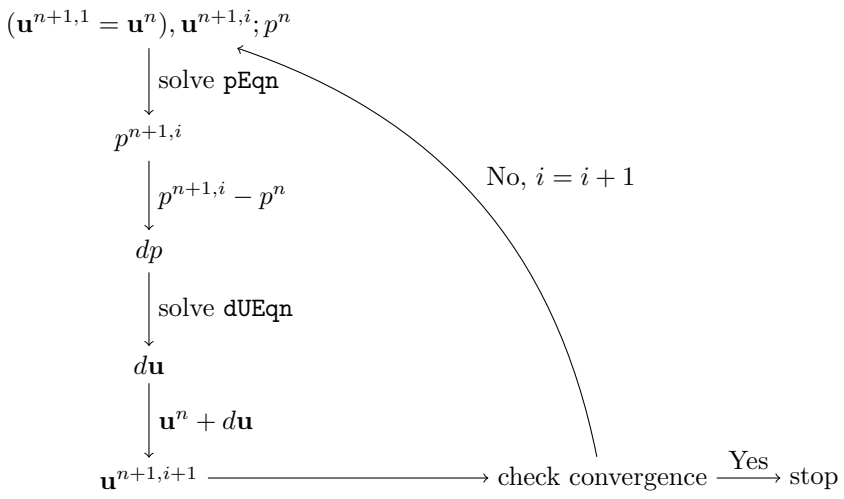


Figure 4.1: The `while` loop in `poroPlasticStressedFoam` solver

We keep the included `calculateStress.H` file unchanged. In the end, we try `wmake`, it compiles without error. The new `poroPlasticStressedFoam` solver is ready.

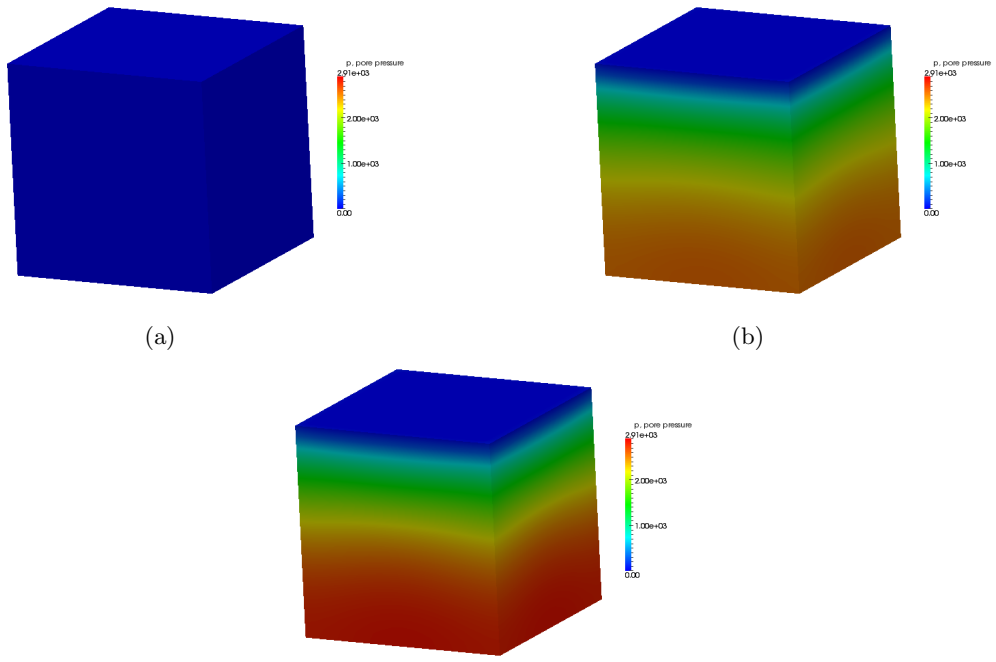
4.3 Simulate `myPoroCase`

Let us simulate a simple case `myPoroCase` to test the solver. Assume a cubic soil sample:

- it is compressed on the top with constant rate of displacement, $d\mathbf{u} = (0 \quad -c \quad 0)$, c is a given positive value;
- the sides and bottom are constrained, so that $d\mathbf{u} = (0 \quad 0 \quad 0)$;
- the top is open for drainage, i.e. $p = 0$;
- the sides and bottom are sealed, which means $\nabla p = 0$.

And we would like to see how the pore pressures will be generated during the test.

The mesh from `icoFoam/cavity` is reused and extended to three dimensions. The details of case set-up codes can be found in Appendix C. Fig 4.2 shows the simulation results.



(a) (b)

(c)

Figure 4.2: Illustration of gradually accumulated pore pressure calculated by `poroPlasticStressedFoam`

Chapter 5

The `coupledPoroFoam` solver

In porous media, the strong interaction between the solid skeleton and pore fluid drives us to think about solving both of the displacement and pore pressure simultaneously. The new *block matrix solver* algorithm then comes to our rescue, since it provides implicit solutions of strongly coupled variables sharing a common mesh.

In this section, we will introduce a new solver, `coupledPoroFoam`, which solves Biot's consolidation equations assuming linear elastic soil skeleton[1]. The `coupedPoroFoam` is developed on the basis of the `blockCoupledScalarTransportFoam` solver available in the OpenFOAM-1.6-ext path of `$Foam_APP/solvers/coupled`.

5.1 Theory of the *block matrix solver*

Let us first have a brief understanding on the theory of the new *block matrix solver* method.

Given a resulting finite volume discretization of a coupled equation set:

$$a_P \mathbf{x}_P + \sum_N a_N \mathbf{x}_N = \mathbf{b} \tag{5.1}$$

where, $\mathbf{x}$ is a vector of m arbitrary variables that we would like to solve from the equation set, and a is the coefficient matrix of dimension $m \times m$. The lower index P stands for the current computing cell and N is the neighbor of cell P.

The *block matrix solver* algorithm will differs from the conventional *segregated* algorithm as follows:

- The *segregated* approach - no coupling between variables:

$$a_P \mathbf{x}_P + \sum_N a_N \mathbf{x}_N = \begin{bmatrix} a_{11} & & \\ & \ddots & \\ & & a_{mm} \end{bmatrix}_P \mathbf{x}_P + \sum_N \begin{bmatrix} a_{11} & & \\ & \ddots & \\ & & a_{mm} \end{bmatrix}_N \mathbf{x}_N \tag{5.2}$$

- The *block matrix solver* approach - coupling between variables in owner and neighbor cells:

$$a_P \mathbf{x}_P + \sum_N a_N \mathbf{x}_N = \begin{bmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mm} \end{bmatrix}_P \mathbf{x}_P + \sum_N \begin{bmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mm} \end{bmatrix}_N \mathbf{x}_N \tag{5.3}$$

We then write out the assembled sparse linear system as:

$$[A][\mathbf{X}] = [B] \tag{5.4}$$

[1] we solve the simple elastic Biot's equations for this simple tutorial, further extension to plasticity of solid skeleton is possible.

- In *segregated* approach, we iteratively solve m small sparse linear systems of $[A]$, each is:

$$[A_{small}] = [\underline{1 \cdot n} \times \underline{1 \cdot n}], [\mathbf{X}] = [\underline{1 \cdot n} \times 1] \tag{5.5}$$

  where, n = number of cells.

- In *block matrix solver* approach, we solve once the large sparse linear system $[A]$:

$$[A] = [\underline{m \cdot n} \times \underline{m \cdot n}], [\mathbf{X}] = [\underline{m \cdot n} \times 1] \tag{5.6}$$

Favorably, the sparseness pattern of block matrix $[A]$ is unchanged from the segregated small scalar matrix $[A_{small}]$.

5.2 Construct our own *block matrix*

Come to our specific case, we are solving the Biot's consolidation equation set (recalled from Eq 4.1-4.2):

$$\nabla \cdot \left[\mu \nabla \mathbf{u} + \mu (\nabla \mathbf{u})^T + \lambda \mathbf{I} tr(\nabla \mathbf{u})\right] = \nabla p \tag{5.7}$$

$$\frac{k}{\gamma} \nabla^2 p = \frac{n}{K'} \frac{\partial p}{\partial t} + \frac{\partial}{\partial t} (\nabla \cdot \mathbf{u}) \tag{5.8}$$

Hence,

$$x = \begin{bmatrix} u_x \\ u_y \\ u_z \\ p \end{bmatrix}, \text{and } a = \begin{bmatrix} a_{11} & \cdots & a_{14} \\ \vdots & \ddots & \vdots \\ a_{41} & \cdots & a_{44} \end{bmatrix} \tag{5.9}$$

Ideally, we could store all the coupling terms inside the coefficient matrix a and implement a fully coupled block matrix solver. However, some restrictions lie in that in all the current OpenFOAM versions no fully implicit discretization for the terms of $\nabla \cdot \left[(\nabla \mathbf{u})^T\right]$ and $\nabla \cdot [tr(\nabla \mathbf{u})\mathbf{I}]$ available yet. Thus the inter-component coupling is still treated explicitly in this tutorial [2]. In the end, we do the following split of implicit and explicit parts:

$$\underbrace{\nabla \cdot [(2\mu + \lambda) \nabla \mathbf{u}]}_{implicit} + \underbrace{\nabla \cdot \left[\mu (\nabla \mathbf{u})^T + \lambda \mathbf{I} tr(\nabla \mathbf{u}) - (\mu + \lambda) \nabla \mathbf{u}\right]}_{explicit} = \underbrace{\nabla p}_{implicit} \tag{5.10}$$

$$\underbrace{\frac{k}{\gamma} \nabla^2 p - \frac{n}{K'} \frac{\partial p}{\partial t}}_{implicit} = \underbrace{\frac{\partial}{\partial t} (\nabla \cdot \mathbf{u})}_{implicit} \tag{5.11}$$

We therefore develop the only pressure-coupled block matrix as:

$$a_P \mathbf{x}_P + \sum_N a_N \mathbf{x}_N = \begin{bmatrix} a_{11} & & & a_{14} \\ & a_{22} & & a_{24} \\ & & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}_P \begin{bmatrix} u_x \\ u_y \\ u_z \\ p \end{bmatrix}_P + \sum_N \begin{bmatrix} a_{11} & & & a_{14} \\ & a_{22} & & a_{24} \\ & & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}_N \begin{bmatrix} u_x \\ u_y \\ u_z \\ p \end{bmatrix}_N \tag{5.12}$$

Moreover, we may also apply a few iterations to recover the 'decoupling' of inter-component terms after solving the block matrix system as follows:

[2] some new implementations are planned for the implicit discretization of laplacian transpose term, i.e. $\nabla \cdot \left[(\nabla \mathbf{u})^T\right]$ (which stores the most part of inter-component coupling), we will see what happens in the future. If it succeeds, the full coupled block matrix can be constructed in the same way

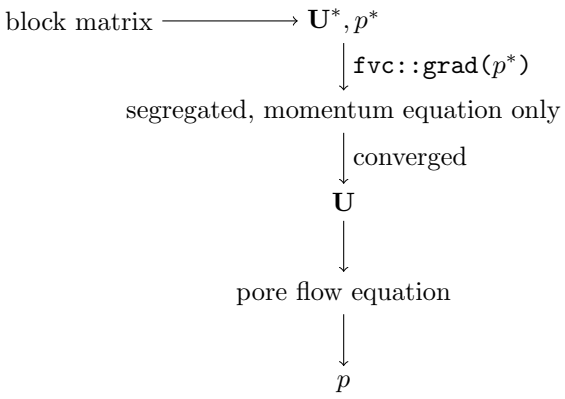


Figure 5.1: The solution routine of `coupledPoroFoam` solver

5.3 The implementation files

After introducing our own block matrix structure and the solution algorithm, we will now describe the practical implementations. Let us first have a general view on the file system of the `coupledPoroFoam` solver, which has been shown in Table 5.1 on next page.

Table 5.1: The file system of `coupledPoroFoam`

solver	file name	description

`coupledPoroFoam`	`vector4Field.H`
	`tensor4Field.H`
	`tensor4Field.C`
	`blockVector4Matrix.H`
	`blockVector4Matrix.C`
	`blockVector4Solvers.C`
		prepare for case specified block-size, define the algebraic method suitable for solving large linear system

	`blockMatrixTools.H` `blockMatrixTools.C`	define the `blockMatrixTools` namespace and functions to fill in the block with information from scalar equations
	`createFields.H`	create the variable fields
	`coupledPoroFoam.C`	The main file, in this solver it simply includes all the necessary files

	`prepareScalarMatrix.H`
	`prepareBlockMatrix.H`
	`solveBlockMatrix.H`
	`loopRecoverExplicit.H`
		The included solution files, which perform the whole procedure of constructing the block matrix from scalar matrices, solving the large linear system, and doing a couple of iterations to recover the explicit inter-component coupling

	`readCoupledPoroFoamControl.H`	define the control parameters for the iterations in `loopRecoverExplicit.H`
	`calculateStress.H`	a post-processing file which calculates elastic stresses from displacement
	`Make` directory	compilation

It is necessary to mention that the original `blockCoupledScalarTransportFoam` solver solves a coupled two-phase fluid/solid heat transfer problem:

$$\nabla \cdot \phi T_f - \nabla \cdot D_{T_f} \nabla T_f = \alpha(T_s - T_f) \tag{5.13}$$

$$-\nabla \cdot D_{T_s} \nabla T_s = \alpha(T_f - T_s) \tag{5.14}$$

where T_s, T_f are the temperatures of solid and fluid. $\phi, D_{T_f}, D_{T_s}, \alpha$ are all material properties.

In this heat transfer problem, the two scalar equations are coupled simply through linear terms, i.e. αT_s in Eq 5.13 and αT_f in Eq 5.14. While, the Biot's equation set is coupled via differential operators, i.e. ∇p in Eq 5.7 and $\frac{\partial}{\partial t}(\nabla \cdot \mathbf{u})$ in Eq 5.8.

It is this difference complicates the implementation, we will therefore focus on describing how to implement these new features[3]. In the original `blockMatrixTools.H` and `blockMatrixTools.C` files, there are only functions to fill in the block diagonals, source terms and working solution variables with the information from `fvScalarMatrix`[4]. We therefore implement one more function called `insertCoupling` to fill in the block off-diagonal terms and also source contribution from the coupling. The corresponding codes in the header file `blockMatrixTools` looks like:

```
// Update coupling of block system
template<class BlockType>
void insertCoupling
(
 const direction dir1,
 const direction dir2,
 const fvScalarMatrix& m,
 BlockLduMatrix<BlockType>& blockM,
 Field<BlockType>& blockB
);
```

The definition of this function in the main file `blockMatrixTools.C` is written as:

```
template<class BlockType>
void insertCoupling
(
    const direction dir1,
    const direction dir2,
    const fvScalarMatrix& m,
    BlockLduMatrix<BlockType>& blockM,
    Field<BlockType>& blockB
)
{
    // Prepare the diagonal and source
    scalarField diag = m.diag();
    scalarField source = m.source();

    // Add boundary source contribution
    m.addBoundaryDiag(diag, 0);
    m.addBoundarySource(source, false);
```

[3]The readers who are interested in the implementation of the original `blockCoupledScalarTransportFoam` solver can refer to Ivor Clifford's presentation slides, "Block-Coupled Simulations Using OpenFOAM", on 6th OpenFOAM Workshop.

[4]It is possible to extend the functions in `blockMatrixTools` namespace so that they can fill in the block matrix with the information from `fvVectorMatrix`, but I haven't managed the whole implementation yet, therefore in this tutorial, we keep using the `fvScalarMatrix`. In future, it is better to fill in the block matrix with `fvVectorMatrix` as well.

```
    if (blockM.diag().activeType() == blockCoeffBase::SQUARE)
    {
        typename CoeffField<BlockType>::squareTypeField& blockDiag =
            blockM.diag().asSquare();

        forAll (diag, i)
        {
            blockDiag[i](dir1, dir2) = -diag[i];
        }
    }

    blockInsert(dir1, source, blockB);

    if (m.hasUpper())
    {
        const scalarField& upper = m.upper();
        if (blockM.upper().activeType() == blockCoeffBase::SQUARE)
        {
            typename CoeffField<BlockType>::squareTypeField& blockUpper =
                blockM.upper().asSquare();

            forAll (upper, i)
            {
                blockUpper[i](dir1, dir2) = -upper[i];
            }
        }
    }

    if (m.hasLower())
    {
        const scalarField& lower = m.lower();
        if (blockM.lower().activeType() == blockCoeffBase::SQUARE)
        {
            typename CoeffField<BlockType>::squareTypeField& blockLower =
                blockM.lower().asSquare();

            forAll (lower, i)
            {
                blockLower[i](dir1, dir2) = -lower [i];
            }
        }
     }

}
```

We also divide the content in the main `.C` file into several included local `.H` files to make the different procedures separate and clear as shown in Fig 5.2:

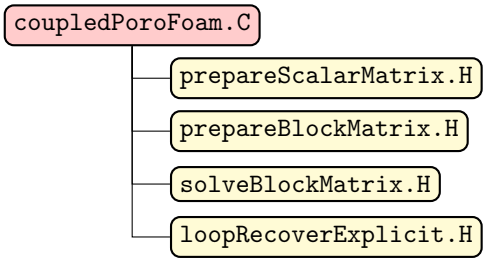


Figure 5.2: The main solution files of `coupledPoroFoam` solver

- The `prepareScalarMatrix.H` file contains the discretization procedure of single scalar equations and builds up scalar matrices. Notice that, because there is no `fvm::grad(p)` available in OpenFOAM yet, we apply a trick to do the implicit discretization of gradient term as follows:

$$\frac{\partial p}{\partial x} = \mathbf{I}_x \cdot \nabla p = \nabla \cdot (\mathbf{I}_x \cdot p), \text{where } \mathbf{I}_x = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix} \tag{5.15}$$

$$\frac{\partial p}{\partial y} = \mathbf{I}_y \cdot \nabla p = \nabla \cdot (\mathbf{I}_y \cdot p), \text{where } \mathbf{I}_x = \begin{pmatrix} 0 & 1 & 0 \end{pmatrix} \tag{5.16}$$

$$\frac{\partial p}{\partial z} = \mathbf{I}_z \cdot \nabla p = \nabla \cdot (\mathbf{I}_z \cdot p), \text{where } \mathbf{I}_z = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix} \tag{5.17}$$

Similarly, for the $\frac{\partial}{\partial t}(\nabla \cdot U)$ term, we apply the following conversions:

$$\frac{\partial}{\partial t}\left(\frac{\partial U_x}{\partial x}\right) = \frac{\partial}{\partial t}\left[\nabla \cdot (\mathbf{I}_x \cdot U_x)\right] = \underbrace{\frac{\nabla \cdot (\mathbf{I}_x \cdot U_x)}{\delta t}}_{\texttt{fvm}} - \underbrace{\frac{1}{\delta t}\left(\frac{\partial U_x^{old}}{\partial x}\right)}_{\texttt{fvc}} \tag{5.18}$$

$$\frac{\partial}{\partial t}\left(\frac{\partial U_y}{\partial y}\right) = \frac{\partial}{\partial t}\left[\nabla \cdot (\mathbf{I}_y \cdot U_y)\right] = \underbrace{\frac{\nabla \cdot (\mathbf{I}_y \cdot U_y)}{\delta t}}_{\texttt{fvm}} - \underbrace{\frac{1}{\delta t}\left(\frac{\partial U_y^{old}}{\partial y}\right)}_{\texttt{fvc}} \tag{5.19}$$

$$\frac{\partial}{\partial t}\left(\frac{\partial U_z}{\partial z}\right) = \frac{\partial}{\partial t}\left[\nabla \cdot (\mathbf{I}_z \cdot U_z)\right] = \underbrace{\frac{\nabla \cdot (\mathbf{I}_z \cdot U_z)}{\delta t}}_{\texttt{fvm}} - \underbrace{\frac{1}{\delta t}\left(\frac{\partial U_z^{old}}{\partial z}\right)}_{\texttt{fvc}} \tag{5.20}$$

where U_x, U_y, U_z are the components of displacement. δt is the time step and $U_x^{old}, U_y^{old}, U_z^{old}$ are the old-time value of displacement components.

The corresponding implementation codes in the `prepareScalarMatrix.H` are:

```
//- prepare the explicit inter-component coupling
volTensorField gradU = fvc::grad(U);
volVectorField divSigExp = fvc::div(mu*gradU.T()+lambda*I*tr(gradU)-
                                    (mu+lambda)*gradU);
//- discretize the momentum equation without coupling of pore pressure
fvScalarMatrix UxEqn
  (
   fvm::laplacian(2*mu+lambda,Ux) == -divSigExp.component(vector::X)
   );

fvScalarMatrix UyEqn
  (
   fvm::laplacian(2*mu+lambda,Uy) == -divSigExp.component(vector::Y)
```

```
    );

fvScalarMatrix UzEqn
  (
   fvm::laplacian(2*mu+lambda,Uz) == -divSigExp.component(vector::Z)
   );

//- discretize the pore flow equation without coupling of displacement
fvScalarMatrix pEqn
  (
   fvm::laplacian(Dp1,p) - fvm::ddt(p)
   );

surfaceScalarField Ixf = fvc::interpolate(Ix) & mesh.Sf();
surfaceScalarField Iyf = fvc::interpolate(Iy) & mesh.Sf();
surfaceScalarField Izf = fvc::interpolate(Iz) & mesh.Sf();

//- discretize the grad(p) coupling in the momentum equation
fvScalarMatrix UxpEqn ( fvm::div(Ixf,p) );
fvScalarMatrix UypEqn ( fvm::div(Iyf,p) );
fvScalarMatrix UzpEqn ( fvm::div(Izf,p) );

//- discretize the ddt(div(U)) coupling in pore flow equation
fvScalarMatrix pUxEqn
(
   fvm::div(Dp2*Ixf/deltaT,Ux) == -1.0/deltaT*Dp2*gradUold.component(tensor::XX)
   );
fvScalarMatrix pUyEqn
  (
   fvm::div(Dp2*Iyf/deltaT,Uy) == -1.0/deltaT*Dp2*gradUold.component(tensor::YY)
   );
fvScalarMatrix pUzEqn
  (
   fvm::div(Dp2*Izf/deltaT,Uz) == -1.0/deltaT*Dp2*gradUold.component(tensor::ZZ)
   );
```

- The `prepareBlockMatrix.H` file creates the block matrix system based on the information from those scalar matrices:

```
//- prepare block system
BlockLduMatrix<vector4> blockA(mesh);

//- grab block diagonal and set it to zero
Field<tensor4>& d = blockA.diag().asSquare();
d = tensor4::zero;

//- grab block off-diagonals and set them to zero
Field<tensor4>& u = blockA.upper().asSquare();
Field<tensor4>& l = blockA.lower().asSquare();
u = tensor4::zero;
l = tensor4::zero;

//- create the source term
vector4Field blockB(mesh.nCells(), vector4::zero);
```

```
//- create the working solution variable
vector4Field blockX(mesh.nCells(), vector4::zero);

//- insert the information from scalar Matrix into the block Matrix system
//- 1. insert the diagonal terms, solution variables, and source terms
blockMatrixTools::insertEquation(0,UxEqn,blockA,blockX,blockB);
blockMatrixTools::insertEquation(1,UyEqn,blockA,blockX,blockB);
blockMatrixTools::insertEquation(2,UzEqn,blockA,blockX,blockB);
blockMatrixTools::insertEquation(3,pEqn,blockA,blockX,blockB);

//- 2. insert the coupling terms, and more source terms
blockMatrixTools::insertCoupling(0,3,UxpEqn,blockA,blockB);
blockMatrixTools::insertCoupling(1,3,UypEqn,blockA,blockB);
blockMatrixTools::insertCoupling(2,3,UzpEqn,blockA,blockB);

blockMatrixTools::insertCoupling(3,0,pUxEqn,blockA,blockB);
blockMatrixTools::insertCoupling(3,1,pUyEqn,blockA,blockB);
blockMatrixTools::insertCoupling(3,2,pUzEqn,blockA,blockB);
```

- The `solveBlockMatrix.H` file simply call the linear system solver, and then retrieve the solutions to the variables:

```
//- Block coupled solver call
BlockSolverPerformance<vector4> solverPerf =
BlockLduSolver<vector4>::New
(
    word("blockVar"),
    blockA,
    mesh.solver("blockVar")
)->solve(blockX, blockB);

solverPerf.print();

// Retrieve solution
blockMatrixTools::blockRetrieve(0, Ux.internalField(), blockX);
blockMatrixTools::blockRetrieve(1, Uy.internalField(), blockX);
blockMatrixTools::blockRetrieve(2, Uz.internalField(), blockX);
blockMatrixTools::blockRetrieve(3, p.internalField(), blockX);

Ux.correctBoundaryConditions();
Uy.correctBoundaryConditions();
Uz.correctBoundaryConditions();
p.correctBoundaryConditions();

U.component(vector::X) = Ux;
U.component(vector::Y) = Uy;
U.component(vector::Z) = Uz;
```

- Since we have no inter-component coupling in the block matrix, the `loopRecoverExplicit.H` file corrects the solution by a few segregated iterations:

```
#     include "readCoupledPoroFoamControl.H"

scalar iCorr = 0;
scalar initialResidual = 0;
```

```
    do
    {
      volTensorField gradU = fvc::grad(U);

          fvVectorMatrix UEqn
               (
                    fvm::laplacian(2.0*mu+lambda,U)
                ==
               -fvc::div(mu*gradU.T()+lambda*I*tr(gradU)-(mu+lambda)*gradU)
               +fvc::grad(p)
               );

          initialResidual = UEqn.solve().initialResidual();

  } while(initialResidual>convergenceTolerance && ++iCorr<nCorr);

    fvScalarMatrix pEqn2
          (
       fvm::laplacian(Dp1,p)-fvm::ddt(p)==fvc::div(fvc::ddt(Dp2,U))
          );
    pEqn2.solve();
```

And, in the end, we have an included `calculateStress.H` file to estimate the linear elastic stress field based on the converged displacement solution.

The codes in the main `coupledPoroFoam` file, which combines all these included local files look like:

```
... //- some lines are ignored here
#include "blockMatrixTools.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

int main(int argc, char *argv[])
{

#   include "setRootCase.H"
#   include "createTime.H"
#   include "createMesh.H"
#   include "createFields.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

    Info<< "\nCalculating scalar transport\n" << endl;

    for (runTime++; !runTime.end(); runTime++)
    {
        Info<< "Time = " << runTime.timeName() << nl << endl;

        dimensionedScalar deltaT = runTime.deltaT();
        volTensorField gradUold = fvc::grad(U);

           #include "prepareScalarMatrix.H"
           #include "prepareBlockMatrix.H"
           #include "solveBlockMatrix.H"
           #include "loopRecoverExplicit.H"
```

```
            #include "calculateStress.H"

        runTime.write();
    }

    Info<< "End\n" << endl;

    return(0);
}

// ************************************************************************* //
```

Though the compilation of `coupledPoroFoam` went successfully, the solver hasn't been verified by test cases yet, future works are therefore required to validate the solver.

Chapter 6

Closure

This tutorial has mainly discussed about the implementations of different solid body stress analysis in OpenFOAM. Based on what has been done, we could easily go further to implement the following topics:

- advanced plasticity with isotropic and/or kinematic hardening, cyclic plasticity, etc.
- fully coupled block matrix solver with implicit discretization of inter-component terms
- fluid structure interactions (FSI)

Moreover, when setting up more test cases, we may encounter some case-specified boundary conditions, for instance, `directionMixedDisplacement` boundary, which could have one displacement component fixed, but others free to deform. Implementing such boundary is also an interesting thing to do with OpenFOAM.

In general, we can conclude that OpenFOAM provides us a lot of potentials to implement more features of solid body stress analysis.

Bibliography

[1] Demirdžić, I., Martinović, D., 1993, *Finite Volume Method for Thermo-Elasto-Plastic Stress Analysis*. Computer Methods in Applied Mechanics and Engineering, 109, 331-349

[2] Jasak, H., Weller, H.G., 2000, *Application of the Finite Volume Method and Unstructured Meshes to Linear Elasticity*. International Journal for Numerical Methods in Engineering, Vol. 48, Issue 2, 267-287

[3] Liu, X.F., Garcia, M.H., 2007, *Numerical Investigation of Seabed Response Under Waves with Free-surface Water Flow*. International Journal of Offshore and Polar Engineering, Vol. 17, No. 2, 97-104

Appendix A. A simple non-hardening Von Mises model for `plasticStressedFoam`

October 13, 2012

1. Constitutive relation:

- elasticity + strain decomposition:

$$d\boldsymbol{\sigma} = 2\mu\left(d\boldsymbol{\varepsilon}^e\right) + \lambda \boldsymbol{I} d\boldsymbol{\varepsilon}^e, \quad d\boldsymbol{\varepsilon}^e = d\boldsymbol{\varepsilon} - d\boldsymbol{\varepsilon}^p$$

- yield surface:

$$f = \sqrt{J_2} - k$$
$$J_2 = \frac{1}{2}\mathbf{s} : \mathbf{s}$$

- Non-hardening:

$$dk = 0$$

where, J_2 is the second deviatoric stress invariant, k is the 'yield stress', and $\mathbf{s}$ is the deviatoric stress.

2. Radial return algorithm:

Inputs: $d\mathbf{U}$, $\boldsymbol{\sigma}_n$, μ, λ, k

Outputs: $\boldsymbol{\sigma}_{n+1}$, $d\boldsymbol{\varepsilon}^p$

- Elastic trial stress:

$$d\mathbf{U}^e = d\mathbf{U}$$
$$(d\boldsymbol{\sigma})_{tr} = \mu[\nabla(d\mathbf{U}^e) + \nabla(d\mathbf{U}^e)^T] + \lambda \mathbf{I} tr[\nabla(d\mathbf{U}^e)]$$
$$\boldsymbol{\sigma}_{tr} = \boldsymbol{\sigma}_n + (d\boldsymbol{\sigma})_{tr}$$

  if $f = \sqrt{J_2} - k = \sqrt{\frac{1}{2}\mathbf{s}_{tr} : \mathbf{s}_{tr}} - k < 0$, elastic trial stress is correct:

$$\boldsymbol{\sigma}_{n+1} = \boldsymbol{\sigma}_{tr}$$
$$d\boldsymbol{\varepsilon}^p = \mathbf{0}$$

  if $(f = \sqrt{J_2} - k > 0)$, do the next step.

- Plastic correction:

$$fac = \frac{\sqrt{J_2}}{k}$$
$$\boxed{\boldsymbol{\sigma}_{n+1} = iso\left(\boldsymbol{\sigma}_{tr}\right) + \frac{dev\left(\boldsymbol{\sigma}_{tr}\right)}{fac}}$$
$$d\boldsymbol{\sigma} = \boldsymbol{\sigma}_{n+1} - \boldsymbol{\sigma}_n$$

$$d\boldsymbol{\varepsilon}^e = \frac{iso\,(d\boldsymbol{\sigma})}{3\lambda + 2\mu} + \frac{dev\,(d\boldsymbol{\sigma})}{2\mu}$$

$$\boxed{d\boldsymbol{\varepsilon}^p = d\boldsymbol{\varepsilon} - d\boldsymbol{\varepsilon}^e = \frac{1}{2}\left\{\nabla(d\mathbf{U}) + [\nabla(d\mathbf{U})]^T\right\} - d\boldsymbol{\varepsilon}^e}$$

Appendix B. The `plasticPlateHole` case

October 13, 2012

- `0/dU` file

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM Extend Project: Open Source CFD        |
|  \\    /   O peration     | Version:  1.6-ext                               |
|   \\  /    A nd           | Web:      www.extend-project.de                 |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volVectorField;
    location    "0";
    object      dU;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 1 0 0 0 0 0];

internalField   uniform (0 0 0);

boundaryField
{
    left
    {
        type            symmetryPlane;
    }

    right
    {
        type            incrementTractionDisplacement;
        incrementTraction        uniform (1000 0 0);
        incrementPressure        uniform 0;
        value           uniform (0 0 0);
    }
```

```
    down
    {
        type            symmetryPlane;
    }

    up
    {
        type            incrementTractionDisplacement;
        incrementTraction       uniform (0 0 0);
        incrementPressure       uniform 0;
        value           uniform (0 0 0);
    }

    hole
    {
        type            incrementTractionDisplacement;
        incrementTraction       uniform (0 0 0);
        incrementPressure       uniform 0;
        value           uniform (0 0 0);
    }

    frontAndBack
    {
        type            empty;
    }
}

// ************************************************************************* //
```

- 0/sigma file

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM Extend Project: Open source CFD        |
|  \\    /   O peration     | Version:  1.6-ext                               |
|   \\  /    A nd           | Web:      www.extend-project.de                 |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volSymmTensorField;
    location    "0";
    object      sigma;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [1 -1 -2 0 0 0 0];

internalField   uniform (0 0 0 0 0 0);

boundaryField
{
    left
    {
        type            symmetryPlane;
    }
    right
    {
        type            zeroGradient;
    }
    down
    {
        type            symmetryPlane;
    }
    up
    {
        type            zeroGradient;
    }
    hole
    {
        type            zeroGradient;
    }
    frontAndBack
    {
        type            empty;
```

```
    }
}


// ************************************************************************** //
```

- `constant/mechanicalProperties` file

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM Extend Project: Open Source CFD        |
|  \\    /   O peration     | Version:  1.6-ext                               |
|   \\  /    A nd           | Web:      www.extend-project.de                 |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      mechanicalProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

nu              nu [0 0 0 0 0 0 0] 0.3;

E               E [1 -1 -2 0 0 0 0] 2e+11;

ky              ky [1 -1 -2 0 0 0 0] 3e+3;

planeStress     yes;

// ************************************************************************* //
```

- constant/polyMesh/blockMeshDict file

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM Extend Project: Open Source CFD        |
|  \\    /   O peration     | Version:  1.6-ext                               |
|   \\  /    A nd           | Web:      www.extend-project.de                 |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant/polyMesh";
    object      blockMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

convertToMeters 1;

vertices
(
    (0.5 0 0)
    (1 0 0)
    (2 0 0)
    (2 0.707107 0)
    (0.707107 0.707107 0)
    (0.353553 0.353553 0)
    (2 2 0)
    (0.707107 2 0)
    (0 2 0)
    (0 1 0)
    (0 0.5 0)
    (0.5 0 0.5)
    (1 0 0.5)
    (2 0 0.5)
    (2 0.707107 0.5)
    (0.707107 0.707107 0.5)
    (0.353553 0.353553 0.5)
    (2 2 0.5)
    (0.707107 2 0.5)
    (0 2 0.5)
    (0 1 0.5)
    (0 0.5 0.5)
);

blocks
```

```
(
    hex (5 4 9 10 16 15 20 21) (10 10 1) simpleGrading (1 1 1)
    hex (0 1 4 5 11 12 15 16) (10 10 1) simpleGrading (1 1 1)
    hex (1 2 3 4 12 13 14 15) (20 10 1) simpleGrading (1 1 1)
    hex (4 3 6 7 15 14 17 18) (20 20 1) simpleGrading (1 1 1)
    hex (9 4 7 8 20 15 18 19) (10 20 1) simpleGrading (1 1 1)
);

edges
(
    arc 0 5 (0.469846 0.17101 0)
    arc 5 10 (0.17101 0.469846 0)
    arc 1 4 (0.939693 0.34202 0)
    arc 4 9 (0.34202 0.939693 0)
    arc 11 16 (0.469846 0.17101 0.5)
    arc 16 21 (0.17101 0.469846 0.5)
    arc 12 15 (0.939693 0.34202 0.5)
    arc 15 20 (0.34202 0.939693 0.5)
);

patches
(
    symmetryPlane left
    (
        (8 9 20 19)
        (9 10 21 20)
    )
    patch right
    (
        (2 3 14 13)
        (3 6 17 14)
    )
    symmetryPlane down
    (
        (0 1 12 11)
        (1 2 13 12)
    )
    patch up
    (
        (7 8 19 18)
        (6 7 18 17)
    )
    patch hole
    (
        (10 5 16 21)
        (5 0 11 16)
    )
    empty frontAndBack
```

```
    (
        (10 9 4 5)
        (5 4 1 0)
        (1 4 3 2)
        (4 7 6 3)
        (4 9 8 7)
        (21 16 15 20)
        (16 11 12 15)
        (12 13 14 15)
        (15 14 17 18)
        (15 18 19 20)
    )
);

mergePatchPairs
(
);

// ************************************************************************* //
```

- system/controlDict file

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM Extend Project: Open Source CFD        |
|  \\    /   O peration     | Version:  1.6-ext                               |
|   \\  /    A nd           | Web:      www.extend-project.de                 |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      controlDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

application plasticStressedFoam;

startFrom       startTime;

startTime       0;

stopAt          endTime;

endTime         100;

deltaT          1;

writeControl    timeStep;

writeInterval   20;

purgeWrite      0;

writeFormat     ascii;

writePrecision  6;

writeCompression uncompressed;

timeFormat      general;

timePrecision   6;

runTimeModifiable yes;
```

```
libs           ("libMyBCs.so");

// ********************************************************************** //
```

- system/fvSchemes file

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM Extend Project: Open Source CFD        |
|  \\    /   O peration     | Version:  1.6-ext                               |
|   \\  /    A nd           | Web:      www.extend-project.de                 |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSchemes;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

d2dt2Schemes
{
    default     steadyState;
}

gradSchemes
{
    default         Gauss linear;
    grad(dU)         leastSquares;
}

divSchemes
{
    default         none;
    div(sigmaExp)      Gauss linear;
    div(sigmaP)        Gauss linear;
}

laplacianSchemes
{
    default         none;
    laplacian(DdU,dU) Gauss linear corrected;
}

interpolationSchemes
{
    default         linear;
}
```

```
snGradSchemes
{
    default         corrected;
}

fluxRequired
{
    default         no;
}

// ************************************************************************* //
```

- system/fvSolution file

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM Extend Project: Open Source CFD        |
|  \\    /   O peration     | Version:  1.6-ext                               |
|   \\  /    A nd           | Web:      www.extend-project.de                 |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSolution;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

solvers
{
    dU
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-06;
        relTol          0.01;
    }

}

plasticStressedFoam
{
    nCorrectors     1;
    dU              1e-06;
}

// ************************************************************************* //
```

Appendix C. The `myPoroCase` case

October 13, 2012

- `0/dU` file

```
  /*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM Extend Project: Open Source CFD        |
|  \\    /   O peration     | Version:  1.6-ext                               |
|   \\  /    A nd           | Web:      www.extend-project.de                 |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volVectorField;
    object      U;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 1 0 0 0 0 0];

internalField   uniform (0 0 0);

boundaryField
{
    movingWall
    {
        type            fixedValue;
        value           uniform (0 -1e-4 0);
    }

    fixedWalls
    {
        type            fixedValue;
        value           uniform (0 0 0);
    }

    frontAndBack
    {
```

```
        type            fixedValue;
        value           uniform (0 0 0);
    }
}

// ************************************************************************* //
```

- 0/p file

```
  /*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM Extend Project: Open Source CFD        |
|  \\    /   O peration     | Version:  1.6-ext                               |
|   \\  /    A nd           | Web:      www.extend-project.de                 |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      p;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [1 -1 -2 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    movingWall
    {
      type            fixedValue;
      value           uniform 0;
    }

    fixedWalls
    {
        type            zeroGradient;
    }

    frontAndBack
    {
        type            zeroGradient;
    }
}

// ************************************************************************* //
```

- 0/sigma file

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM Extend Project: Open source CFD        |
|  \\    /   O peration     | Version:  1.6-ext                               |
|   \\  /    A nd           | Web:      www.extend-project.de                 |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volSymmTensorField;
    location    "0";
    object      sigma;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [1 -1 -2 0 0 0 0];

internalField   uniform (0 0 0 0 0 0);

boundaryField
{
    left
    {
        type            symmetryPlane;
    }
    right
    {
        type            zeroGradient;
    }
    down
    {
        type            symmetryPlane;
    }
    up
    {
        type            zeroGradient;
    }
    hole
    {
        type            zeroGradient;
    }
    frontAndBack
    {
        type            empty;
```

```
    }
}


// ************************************************************************** //
```

- `constant/mechanicalProperties` file

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM Extend Project: Open Source CFD        |
|  \\    /   O peration     | Version:  1.6-ext                               |
|   \\  /    A nd           | Web:      www.extend-project.de                 |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      mechanicalProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

nu              nu [0 0 0 0 0 0 0] 0.3;

E               E [1 -1 -2 0 0 0 0] 1e+7;

nu              nu [0 0 0 0 0 0 0] 0.3;

ky              ky [1 -1 -2 0 0 0 0] 3e+4;

k               k [0 1 -1 0 0 0 0] 1e-3;

gamma           gamma [1 -2 -2 0 0 0 0] 1e+4;

n               n [0 0 0 0 0 0 0] 0.5;

Kprime          Kprime [1 -1 -2 0 0 0 0] 1e+9;

planeStress     no;

// ************************************************************************* //
```

- constant/polyMesh/blockMeshDict file

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM Extend Project: Open Source CFD        |
|  \\    /   O peration     | Version:  1.6-ext                               |
|   \\  /    A nd           | Web:      www.extend-project.de                 |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      blockMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

convertToMeters 0.1;

vertices
(
    (0 0 0)
    (1 0 0)
    (1 1 0)
    (0 1 0)
    (0 0 1)
    (1 0 1)
    (1 1 1)
    (0 1 1)
);

blocks
(
    hex (0 1 2 3 4 5 6 7) (20 20 20) simpleGrading (1 1 1)
);

edges
(
);

patches
(
    wall movingWall
    (
        (3 7 6 2)
    )
    wall fixedWalls
```

```
    (
        (0 4 7 3)
        (2 6 5 1)
        (1 5 4 0)
    )
    wall frontAndBack
    (
        (0 3 2 1)
        (4 5 6 7)
    )
);

mergePatchPairs
(
);

// ************************************************************************* //
```

- system/controlDict file

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM Extend Project: Open Source CFD        |
|  \\    /   O peration     | Version:  1.6-ext                               |
|   \\  /    A nd           | Web:      www.extend-project.de                 |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      controlDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

application poroPlasticStressedFoam;

startFrom       startTime;

startTime       0;

stopAt          endTime;

endTime         1;

deltaT          0.01;

writeControl    timeStep;

writeInterval   10;

purgeWrite      0;

writeFormat     ascii;

writePrecision  6;

writeCompression uncompressed;

timeFormat      general;

timePrecision   6;

runTimeModifiable yes;
```

```
// ********************************************************************** //
```

- system/fvSchemes file

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM Extend Project: Open Source CFD        |
|  \\    /   O peration     | Version:  1.6-ext                               |
|   \\  /    A nd           | Web:      www.extend-project.de                 |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      fvSchemes;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

ddtSchemes
{
    default         Euler;
}

gradSchemes
{
    default          leastSquares;
    grad(dp)         leastSquares;
    grad(dU)         leastSquares;
}

divSchemes
{
    default          none;
    div(ddt(U))      Gauss linear corrected;
    div(sigmaExp)    Gauss linear;
    div(sigmaP)      Gauss linear;
}

laplacianSchemes
{
    default         none;
    laplacian(DdU,dU) Gauss linear corrected;
    laplacian(Dp1,p)  Gauss linear corrected;
}

interpolationSchemes
{
    default         linear;
```

```
}

snGradSchemes
{
    default         none;
}

fluxRequired
{
    default         no;
    dU              yes;
}

// ************************************************************************* //
```

- system/fvSolution file

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  1.7.1                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSolution;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

solvers
{
    dU
    {
        solver          GAMG;
        tolerance       1e-06;
        relTol          0.9;
        smoother        GaussSeidel;
        cacheAgglomeration true;
        nCellsInCoarsestLevel 20;
        agglomerator    faceAreaPair;
        mergeLevels     1;
    }

    p
    {
        solver          GAMG;
        tolerance       1e-06;
        relTol          0.9;
        smoother        GaussSeidel;
        cacheAgglomeration true;
        nCellsInCoarsestLevel 20;
        agglomerator    faceAreaPair;
        mergeLevels     1;
    }
}

poroPlasticStressedFoam
{
```

```
    nCorrectors     1000;
    dU              1e-06;
    p               1e-06;
}


// ************************************************************************* //
```