# [OpenFOAM source code digging] interFoam



2016年4月5日

Japanese original link

Posted on: November 11, 2012

Translation Date: April 5, 2016

We recommend viewing with a Chrome browser.

------ OpenFOAM source code digging series ------

OpenFOAM Source Code Digging Go to Table of Contents

---------- ----------

----- Basic grammar handling OpenFOAM source code ---- -

Go to syntax basics covering OpenFOAM source code

---------- ----------

### ingress

Let's take a look at interFoam's code

### version used

OpenFOAM 2.1.1

### Volume of Fluid (VOF) 법

interFoam is a multiphase fluid (two different phases) solver by Volume Of Fluid (VOF) method. In the VOF method, the volume fraction (0 to 1) of each phase is used to distinguish the phases. When there are two phases, if the volume fraction of the first phase is $\alpha_1$, the volume fraction of the second phase is calculated as $1-\alpha_1$.

The behavior of the volume fraction of the phase is expressed by the transport equation.

$$D\alpha_1/Dt = 0$$

The momentum equation solves the same two equations as for single-phase, but finds the density and viscosity from each of the two fluids.

$$\rho = \alpha_1 \rho_1 + (1 - \alpha_1) \rho_2$$
$$\mu = \alpha_1 \mu_1 + (1 - \alpha_1) \mu_2$$

Surface tension is added to the momentum equation.

$$f_s = \sigma \kappa n \text{ (surface tension)}$$
$$n = \nabla\alpha_1/|\nabla\alpha_1| \text{ (normal vector of the boundary surface)}$$
$$\kappa = \nabla \cdot n \text{ (curvature of the boundary surface)}$$

### Interface Compression

In interFoam, it produces released as follows: a volume fraction of the transport equation .

$$D\alpha_1/Dt + \nabla \cdot (\alpha_1 (1 - \alpha_1) U_r) = 0$$

Ur is the relative velocity of the phase (U1-U2), called "compression velocity". The fluid flux on the side corresponding to Ur is calculated as follows .

$$\Phi_r = n_f \min (C_a | \Phi | / | S |, \max (| \Phi | / | S |))$$

Ca is an integer indicating the degree of compression. This corresponds to cAlpha designated by fvSolution.

To discretize the compression velocity, a scheme called "interfaceCompression" is used.

fvSchemes

```
divSchemes

{

...

div(phi,alpha) Gauss vanLeer;

div(phirb,alpha) Gauss interfaceCompression;

}
```

### Interface Compression

Let's look at the code.

interFoam.C

```
    while (runTime.run())
    {
        #include "readTimeControls.H"
        #include "CourantNo.H"
        #include "alphaCourantNo.H"
        #include "setDeltaT.H"

        runTime++;

        Info<< "Time = " << runTime.timeName() << nl << endl;

        twoPhaseProperties.correct();

        #include "alphaEqnSubCycle.H"

        // --- Pressure-velocity PIMPLE corrector loop
        while (pimple.loop())
        {
            #include "UEqn.H"

            // --- Pressure corrector loop
            while (pimple.correct())
            {
                #include "pEqn.H"
            }

            if (pimple.turbCorr())
            {
                turbulence->correct();
            }
        }

        runTime.write();

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

### twoPhaseMixture

First, twoPhaseProperties.correct() is being called. twoPhaseProperties is defined in createFields.H as follows .

```
twoPhaseMixture twoPhaseProperties(U, phi);
```

correct() of twoPhaseMixture is as follows.

$FOAM_SRC/transportModels/incompressible/incompressibleTwoPhaseMixture/twoPhaseMixture.H

```
virtual void correct()

{

calcNu ();

}
```

calcNu() looks like this: (twoPhaseMixture.C).

```
//- Calculate and return the laminar viscosity
void Foam::twoPhaseMixture::calcNu()
{
    nuModel1_->correct();
    nuModel2_->correct();

    const volScalarField limitedAlpha1
    (
        "limitedAlpha1",
        min(max(alpha1_, scalar(0)), scalar(1))
    );

    // Average kinematic viscosity calculated from dynamic viscosity
    nu_ = mu()/(limitedAlpha1*rho1_ + (scalar(1) - limitedAlpha1)*rho2_);
}
```

The kinematic viscosity coefficient is calculated.

### alphaEqnSubCycle

Next, take a look at the following line :

```
#include "alphaEqnSubCycle.H"
```

alphaEqnSubCycle.H

```
label nAlphaCorr(readLabel(pimple.dict().lookup("nAlphaCorr")));

label nAlphaSubCycles(readLabel(pimple.dict().lookup("nAlphaSubCycles")));

if (nAlphaSubCycles > 1)
{
    dimensionedScalar totalDeltaT = runTime.deltaT();
    surfaceScalarField rhoPhiSum(0.0*rhoPhi);

    for
    (
        subCycle<volScalarField> alphaSubCycle(alpha1, nAlphaSubCycles);
        !(++alphaSubCycle).end();
    )
    {
        #include "alphaEqn.H"
        rhoPhiSum += (runTime.deltaT()/totalDeltaT)*rhoPhi;
    }

    rhoPhi = rhoPhiSum;
}
else
{
    #include "alphaEqn.H"
}

interface.correct();

rho == alpha1*rho1 + (scalar(1) - alpha1)*rho2;
```

If nAlphaSubCycle (specified in fvSolution) is 1 or more, enter the subcycle and enter "alphaEqn.H". Otherwise, proceed without subcycles .

### subCycle

Let's follow the subCycle.

$FOAM_SRC/OpenFOAM/algorithms/subCycle/subCycle.H

```
        subCycle(GeometricField& gf, const label nSubCycles)
        :
            subCycleField<GeometricField>(gf),
            subCycleTime(const_cast<Time&>(gf.time()), nSubCycles)
        {}
```

subCycleField holds the received field.

$FOAM_SRC/OpenFOAM/db/Time/subCycleTime.C

```
Foam::subCycleTime::subCycleTime(Time& t, const label nSubCycles)
:
    time_(t),
    nSubCycles_(nSubCycles),
    subCycleIndex_(0)
{
    time_.subCycle(nSubCycles_);
}
```

Finally, it is executed until Time.

Time.C

```
Foam::TimeState Foam::Time::subCycle(const label nSubCycles)
{
    subCycling_ = true;
    prevTimeState_.set(new TimeState(*this));

    setTime(*this - deltaT(), (timeIndex() - 1)*nSubCycles);
    deltaT_ /= nSubCycles;
    deltaT0_ /= nSubCycles;
    deltaTSave_ = deltaT0_;

    return prevTimeState();
}


void Foam::Time::endSubCycle()
{
    if (subCycling_)
    {
        subCycling_ = false;
        TimeState::operator=(prevTimeState());
        prevTimeState_.clear();
    }
}
```

For sub-cycles, work is done, such as a finer division of the time. At the end of the subcycle, it is returned to its original state. That is, it can be seen that when dealing with normal time within a subcycle, it is treated as time within a subcycle .

### alphaEqn

Now, let's go to alphaEqn.H.

alphaEqn.H

```
{
    word alphaScheme("div(phi,alpha)");
    word alpharScheme("div(phirb,alpha)");

    surfaceScalarField phic(mag(phi/mesh.magSf()));
    phic = min(interface.cAlpha()*phic, max(phic));
    surfaceScalarField phir(phic*interface.nHatf());

    for (int aCorr=0; aCorr<nAlphaCorr; aCorr++)

{
         surfaceScalarField phiAlpha
        (
            fvc::flux
            (
                phi,
                alpha1,
                alphaScheme
            )
          + fvc::flux
            (
                -fvc::flux(-phir, scalar(1) - alpha1, alpharScheme),
                alpha1,
                alpharScheme
            )
        );

        MULES::explicitSolve(alpha1, phi, phiAlpha, 1, 0);

        rhoPhi = phiAlpha*(rho1 - rho2) + phi*rho2;
}

Info<< "Phase-1 volume fraction = "
    << alpha1.weightedAverage(mesh.Vsc()).value()
    << "  Min(alpha1) = " << min(alpha1).value()
    << "  Max(alpha1) = " << max(alpha1).value()

 << endl;

}
```

Here, the transport equation of alpha1 is solved. aphir is the fluid flux at the compression velocity. I'm going into a loop and solving an equation, which should be nAlphaCorr > 1 (specified by fvSolution). I created phiAlpha and solved it with MULES::explicitSolve(). The first term in phiAlpha is the fluid movement term, and the second term is the compression velocity term. In the second term, alpharScheme("div(phirb,alpha)") is used. This means that we need to specify "interfaceCompression" (fvSchemes) .

### interfaceCompression

The interfaceCompression scheme is as follows.

$FOAM_SRC/transportModels/interfaceProperties/interfaceCompression/interfaceCompression.H

```
class interfaceCompressionLimiter
{

public:

    interfaceCompressionLimiter(Istream&)
    {}

    scalar limiter
    (
        const scalar cdWeight,
        const scalar faceFlux,
        const scalar phiP,
        const scalar phiN,
        const vector&,
        const scalar
    ) const
    {
        // Quadratic compression scheme
        //return min(max(4*min(phiP*(1 - phiP), phiN*(1 - phiN)), 0), 1);

        // Quartic compression scheme
        return
            min(max(
            1 - max(sqr(1 - 4*phiP*(1 - phiP)), sqr(1 - 4*phiN*(1 - phiN))),
            0), 1);
    }
};
```

I don't quite understand, so let's move on to the next one .

**MULES**

MULES is described in MULES.H as follows.

```
MULES: Multidimensional universal limiter with explicit solution.

Solve a convective-only transport equation using an explicit universal
multi-dimensional limiter.

Parameters are the variable to solve, the normal convective flux and the
actual explicit flux of the variable which is also used to return limited
flux used in the bounded-solution.
```

.... Let's move on to the next one.

$FOAM_SRC/finiteVolume/fvMatrices/solvers/MULES/MULES.H

```
    surfaceScalarField muEff
    (
        "muEff",
        twoPhaseProperties.muf()
      + fvc::interpolate(rho*turbulence->nut())
    );

    fvVectorMatrix UEqn
    (
        fvm::ddt(rho, U)
      + fvm::div(rhoPhi, U)
      - fvm::laplacian(muEff, U)
      - (fvc::grad(U) & fvc::grad(muEff))
    //- fvc::div(muEff*(fvc::interpolate(dev(fvc::grad(U))) & mesh.Sf()))
    );

    UEqn.relax();

    if (pimple.momentumPredictor())
    {
        solve
        (
            UEqn
         ==
            fvc::reconstruct
            (
                (
                    fvc::interpolate(interface.sigmaK())*fvc::snGrad(alpha1)
                  - ghf*fvc::snGrad(rho)
                  - fvc::snGrad(p_rgh)
                ) * mesh.magSf()
            )
        );
    }
```

twoPhaseProperties.muf() is calculated as follows.

twoPhaseMixture.C

```
Foam::tmp<Foam::surfaceScalarField> Foam::twoPhaseMixture::muf() const
{
    const surfaceScalarField alpha1f
    (
        min(max(fvc::interpolate(alpha1_), scalar(0)), scalar(1))
    );

    return tmp
    (
        new surfaceScalarField
        (
            "muf",
            alpha1f*rho1_*fvc::interpolate(nuModel1_->nu())
          + (scalar(1) - alpha1f)*rho2_*fvc::interpolate(nuModel2_->nu())
        )
    );
}
```

### surface tension

The right side of the momentum equation is complexly expressed because surface tension is added. The term below is the surface tension .

```
fvc::interpolate(interface.sigmaK())*fvc::snGrad(alpha1)
```

The interface is defined as createFields.H .

```
// Construct interface from alpha1 distribution

interfaceProperties interface(alpha1, U, twoPhaseProperties);
```

$FOAM_SRC/transportModels/interfaceProperties/interfaceProerties.H

```
tmp<volScalarField> sigmaK() const

{

return sigma_*K_;

}
```

sigmaK() returns $\sigma\kappa$ . $\kappa$ is calculated as follows.

interfaceProperties.C

```
void Foam::interfaceProperties::calculateK()
{
    const fvMesh& mesh = alpha1_.mesh();
    const surfaceVectorField& Sf = mesh.Sf();

    // Cell gradient of alpha
    const volVectorField gradAlpha(fvc::grad(alpha1_));

    // Interpolated face-gradient of alpha
    surfaceVectorField gradAlphaf(fvc::interpolate(gradAlpha));

    //gradAlphaf -=
    //    (mesh.Sf()/mesh.magSf())
    //   *(fvc::snGrad(alpha1_) - (mesh.Sf() & gradAlphaf)/mesh.magSf());

    // Face unit interface normal
    surfaceVectorField nHatfv(gradAlphaf/(mag(gradAlphaf) + deltaN_));
    correctContactAngle(nHatfv.boundaryField(), gradAlphaf.boundaryField());

    // Face unit interface normal flux
    nHatf_ = nHatfv & Sf;

    // Simple expression for curvature
    K_ = -fvc::div(nHatf_);

    // Complex expression for curvature.
    // Correction is formally zero but numerically non-zero.
    /*
    volVectorField nHat(gradAlpha/(mag(gradAlpha) + deltaN_));
    forAll(nHat.boundaryField(), patchi)
    {
        nHat.boundaryField()[patchi] = nHatfv.boundaryField()[patchi];
    }

    K_ = -fvc::div(nHatf_) + (nHat & fvc::grad(nHatfv) & nHat);
    */
}
```

### gravity

The term of gravity is given below .

```
- ghf*fvc::snGrad(rho)
```

ghf is defined in createFields.H as follows .

```
surfaceScalarField ghf("ghf", g & mesh.Cf());
```

The dot product of the gravitational acceleration vector and the center position vector of the plane is shown. That is, the product of the height in the direction of gravity and the acceleration due to gravity is calculated.

### pEqn.H

Let's go to pEqn.H.

pEqn.H

```
{
    volScalarField rAU(1.0/UEqn.A());
    surfaceScalarField rAUf(fvc::interpolate(rAU));

    U = rAU*UEqn.H();
    surfaceScalarField phiU
    (
        "phiU",
        (fvc::interpolate(U) & mesh.Sf())
      + fvc::ddtPhiCorr(rAU, rho, U, phi)
    );

    adjustPhi(phiU, U, p_rgh);

    phi = phiU +
    (
        fvc::interpolate(interface.sigmaK())*fvc::snGrad(alpha1)
      - ghf*fvc::snGrad(rho)
    )*rAUf*mesh.magSf();

    while (pimple.correctNonOrthogonal())
    {
        fvScalarMatrix p_rghEqn
        (
            fvm::laplacian(rAUf, p_rgh) == fvc::div(phi)
        );

        p_rghEqn.setReference (pRefCell, getRefCellValue (p_rgh, pRefCell));

        p_rghEqn.solve (mesh.solver (p_rgh.select (pimple.finalInnerIter ())));

        if (pimple.finalNonOrthogonalIter ())
        {
            phi - = p_rghEqn.flux ();
        }
    }

    U + = rAU * fvc :: reconstruct ((phi - phiU) / rAUf);
    U.correctBoundaryConditions ();

    #include "continuityErrs.H"

    p == p_rgh + rho * gh;

    if (p_rgh.needReference ())
    {
        p + = dimensionedScalar
        (
            "p",
            p.dimensions (),
            pRefValue - getRefCellValue (p, pRefCell)
        );
        p_rgh = p - rho * gh;
    }
}
```

The surface tension is also taken into account here.

Pressure and release the p_rgh, which is calculated as follows .

```
p_rgh = p - rgh * gh;
```

gh is in createFields.H .

```
volScalarField gh("gh", g * mesh.C());
```

Same as ghf. Here we use the center of the cell. That is p_rgh is p- ρ means a gh.

### references

H. Rusche, Computational fluid dynamics of dispersed two-phase flows at high phase fractions, PhD Thesis, Imperial College, 2002

OpenFOAM Source Code Digging Go to Table of Contents



**Other posts in the ' OpenFOAM > Source Code Digging ' category**

- **[OpenFOAM source code digging] interFoam (0)**
- **[OpenFOAM source code digging] Boundary condition (0)**
- **[OpenFOAM source code digging] Wall function (0)**
- **[OpenFOAM source code analysis] autoPtr and tmp (0)**
- **[OpenFOAM source code digging] Porous media (0)**
- **[OpenFOAM source code analysis] Added transport characteristics (0)**