Tutorial

Solve Cavitating flow around a 2D hydrofoil using a user modified version of interPhaseChangeFoam

NaiXian LU

Department of Shipping and Marine Technology, Chalmers University of Technology,

SE 412 96 Gothenburg, Sweden

Email: naixian.lu@chalmers.se

In this tutorial we shall solve a problem of cavitating flow around a 2 dimensional NACA hydrofoil using a user modified version of a standard OpenFOAM-1.5 multiphase solver interPhaseChangeFoam. Flow modeling is achieved by Large Eddy Simulation (LES). The phase interface is taken care of using a Volume of Fluid (VOF) approach where the transport equation for the volume fraction γ, is incorporated into the filtered equations of continuity and momentum. Cavitation is modeled by certain mass transfer models modeling the mass transfer source term appearing in the continuity equation. The author does not intend to go into details about the mechanism of different mass transfer models and the comparison between these models falls out of the scope of this tutorial.

1. Governing Equations and Implementations

1.1 Flow modeling

By applying low-pass filtering to Navier-Stokes Equations, using a pre-defined filter kernel function $G = G(x,\Delta)$, the LES equations are derived as:

$$\begin{cases} \nabla \cdot \bar{v} = m_1 \\ \partial_t(\bar{v}) + \nabla \cdot \overline{(v \otimes v)} = -\dfrac{\nabla \bar{p}}{\rho} + \nabla \cdot (\bar{S} - B) + m_2 \end{cases}$$

where the over-bar denotes the low-pass filtered dependent variables. $\bar{S} = 2\nu\bar{D}$ is the filtered viscous stress tensor, $\bar{D}$ is the filtered rate-of-strain-tensor $D = \frac{1}{2}(\nabla v + \nabla v^T)$. $B = (\overline{v \otimes v} - \bar{v} \otimes \bar{v})$ is the subgrid stress tensor, representing the influence of the small, unresolved eddy scales on the larger, resolved flow scales. The commutation error terms, $m_1 = \nabla \cdot \bar{v} - \overline{\nabla \cdot v}$ and $m_2 = \nabla \cdot (\overline{v \otimes v}) - \overline{\nabla \cdot (v \otimes v)} + \nabla \bar{p} - \overline{\nabla p} - \nabla \cdot \bar{S} + \overline{\nabla \cdot S}$ are expected to be significantly smaller than the subgrid terms therefore these terms will be set to zero. Based on Boussinesq hypothesis, a subgrid viscosity ν_k is considered and the resulting term in the LES equations becomes $B = -2\nu_k\bar{D}$ so that the whole viscous term can be described as a function of the effective viscosity (summation of the molecular viscosity and subgrid viscosity) and rate-of-strain tensor.

1.2 Cavitation modeling

In the VOF approach, the physical properties of the fluid are scaled by a volume fraction, *γ=volume liquid/total volume*, which is the liquid volume equation fraction, with γ=1 corresponding to pure water. It is defined as

$$\gamma = \lim_{\delta \to 0} \frac{\delta V_l}{\delta V_l + \delta V_v}$$

and is used to scale the physical properties of vapour and liquid as

$$\begin{cases} \rho = \gamma\rho_l + (1-\gamma)\rho_v \\ \mu = \gamma\mu_l + (1-\gamma)\mu_v \end{cases}$$

Using the VOF approach, a transport equation for the volume fraction needs to be incorporated into the filtered equations of continuity and momentum. Besides, when the flow starts to cavitate, the governing equations become no longer divergence free and are evolved as:

$$\begin{cases} \partial_t\gamma + \nabla \cdot \bar{v}\gamma = \dfrac{\dot{m}}{\rho_l} \\ \nabla \cdot \bar{v} = S_p \\ \partial_t(\rho\bar{v}) + \nabla \cdot (\rho\bar{v} \otimes \bar{v}) = -\nabla\bar{p} + \nabla \cdot \rho(\bar{S} - B) \end{cases}$$

where $S_p = (\rho_l^{-1} - \rho_v^{-1})\dot{m}$, the mass transfer rate $\dot{m}$ is to be modeled by mass transfer models.

The implemented mass transfer models in interPhaseChangeFoam are Merkle, Kunz and SchnerrSauer mass transfer model. The governing equations are respectively:

Merkle mass transfer model:

$$\begin{cases} \dot{m}^+ = \left(C_v / \frac{1}{2} U_\infty{}^2 t_\infty \right) / \rho_l \cdot \gamma \min\left[0, \bar{p} - p_v\right] \\ \dot{m}^- = \left(C_c / \frac{1}{2} U_\infty{}^2 t_\infty \right) \rho_v / \rho_l \cdot \gamma [1-\gamma] \max\left[0, \bar{p} - p_v\right] \end{cases}$$

Kunz' mass transfer model:

$$\begin{cases} \dot{m}^+ = \left(C_v / \frac{1}{2} U_\infty{}^2 t_\infty \right) \rho_v / \rho_l \cdot \gamma \min\left[0, \bar{p} - p_v\right] \\ \dot{m}^- = \left(C_c / t_\infty \right) \rho_v \cdot \gamma^2 [1-\gamma] \end{cases}$$

Kunz mass transfer model is based on the work by Merkle et al. , with a modification that corresponds to the behavior of a fluid near the transition point. The final form of the model can be considered as based on fairly intuitive, ad hoc arguments. The destruction of liquid, or creation of vapour, $\dot{m}^+$, is modeled to be proportional to the amount by which the pressure is below the vapour pressure and the destruction of vapour $\dot{m}^-$ is based on a third order polynomial function of the volume fraction. The specific mass transfer rate is computed as $\dot{m} = \dot{m}^+ + \dot{m}^-$, $\bar{p}$ is the filtered pressure, p_v is the vaporization pressure and C_{prod}, C_{dest}, U_∞ and t_∞ are empirical constants based on the mean flow.

SchnerrSauer mass transfer model:

$$\dot{m} = -3\rho_v \sqrt[3]{n_0 \frac{4}{3}\pi\left(\alpha^2 - \alpha^3\left(1 - \rho_v / \rho_l\right)\right)} sign\left(p_v - p\right) \sqrt{\frac{2}{3}\frac{|p_v - p|}{\rho_l}}$$

Sauer's model is based on bubble dynamics and the amount of vapour in a control volume is calculated from the number of nesting vapour bubbles and an average radius of these bubbles. In the above equation n_0 stands for the number density of micro bubbles and α=*volume vapour/total volume* is the vapour volume fraction defined as

$$\alpha = \lim_{\delta \to 0} \frac{\delta V_v}{\delta V_v + \delta V_l}$$

1.3 Implementations

First we are going to take a look at the inplementation at the original solver interPhaseChangeFoam under \$WM_PROJECT_DIR/applications/solvers/multiphase/*interPhaseChangeFoam/*.

In the VOF method, we solve the momentum equation and continuity equation for the two phase mixture. The momentum equation takes the form:

$$\partial_t(\rho U) + \nabla \cdot (\rho U U) - \nabla \cdot \mu \nabla U - \rho g = -\nabla p - F_s$$

where F_s is the surface tension force which takes place only at the free surfaces. The surface tension is computed as

$$F_s = \sigma \kappa(x) n$$

where n is a unit vector normal to the interface that can be calculated from

$$n = \frac{\nabla \gamma}{|\nabla \gamma|}$$

and κ is the curvature of the interfae that can be calculated from

$$\kappa(x) = \nabla \cdot n$$

The volume fraction is solved by a separate transport equation with an extra artificial compression term to perform necessary compression of the surface. It takes the form:

$$\partial_t \gamma + \nabla \cdot (\gamma U) + \nabla \cdot \left(\gamma(1-\gamma)U_\gamma\right) = \frac{\dot{m}^+ + \dot{m}^-}{\rho_l}$$

where U_γ is a velocity field suitable to compress the interface. This artificial term is active only in the interface region due to the term γ(1-γ).

Different LES models are implemented respectively under *$FOAM_SRC/turbulenceModels/LES/incompressible.*

The original source code of interPhaseChangeFoam.C looks:

```
/******************interPhaseChangeFoam.C******************/
Info<< "\nStarting time loop\n" << endl;

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

        runTime++;

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

        twoPhaseProperties->correct();

        #include "gammaEqnSubCycle.H"

        turbulence->correct();

        // --- Outer-corrector loop
        for (int oCorr=0; oCorr<nOuterCorr; oCorr++)
        {
            #include "UEqn.H"
```

```
        // --- PISO loop
        for (int corr=0; corr<nCorr; corr++)
        {
            #include "pEqn.H"
        }

        #include "continuityErrs.H"
    }

    runTime.write();

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

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

  return(0);
}
```

When the time iteration loop starts, the solver first calculates the correction of PISO loop around the γ phase. Then the courant number is calculated and based on the courant number value, the new time step size is adjusted.

In the gammaEqnSubCycle.H, the solver looks for the number of correctors to the loop around the gamma equation and the number to sub-cycles. Then, two phase properties are calculated in gammaEqn.H (included in gammaEqnSubCycle.H) for the prescribed number of sub-cycles. The algorithm to solve for gamma equation uses a technique called interfaceCompression to resolve some of the fundamental problems of the traditional VOF interface compression methods. Finally the density of the mixture is calculated using the weighted averaged of the gamma field.

```
/******************gammaEqnSubCycle.H******************/
surfaceScalarField rhoPhi
(
  IOobject
  (
    "rhoPhi",
    runTime.timeName(),
    mesh
  ),
  mesh,
  dimensionedScalar("0", dimensionSet(1, 0, -1, 0, 0), 0)
);

{
```

```
    label nGammaCorr
    (
        readLabel(piso.lookup("nGammaCorr"))
    );

    label nGammaSubCycles
    (
        readLabel(piso.lookup("nGammaSubCycles"))
    );

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

    volScalarField divU = fvc::div(phi);

    dimensionedScalar totalDeltaT = runTime.deltaT();

    if (nGammaSubCycles > 1)
    {
        for
        (
            subCycle<volScalarField> gammaSubCycle(gamma, nGammaSubCycles);
            !(++gammaSubCycle).end();
        )
        {
#           include "gammaEqn.H"
        }
    }
    else
    {
#       include "gammaEqn.H"
    }

    if (nOuterCorr == 1)
    {
        interface.correct();
    }

    rho == gamma*rho1 + (scalar(1) - gamma)*rho2;
}

/*****************gammaEqn.H*****************/
{
```

```
word gammaScheme("div(phi,gamma)");
word gammarScheme("div(phirb,gamma)");

surfaceScalarField phir("phir", phic*interface.nHatf());

for (int gCorr=0; gCorr<nGammaCorr; gCorr++)
{
    surfaceScalarField phiGamma =
        fvc::flux
        (
            phi,
            gamma,
            gammaScheme
        )
      + fvc::flux
        (
            -fvc::flux(-phir, scalar(1) - gamma, gammarScheme),
            gamma,
            gammarScheme
        );

    Pair<tmp<volScalarField> > vDotAlphal =
        twoPhaseProperties->vDotAlphal();
    const volScalarField& vDotcAlphal = vDotAlphal[0]();
    const volScalarField& vDotvAlphal = vDotAlphal[1]();

    volScalarField Sp
    (
        IOobject
        (
            "Sp",
            runTime.timeName(),
            mesh
        ),
        vDotvAlphal - vDotcAlphal
    );

    volScalarField Su
    (
        IOobject
        (
            "Su",
            runTime.timeName(),
```

```
            mesh
        ),
        divU*gamma
      + vDotcAlphal
    );

    MULES::implicitSolve(oneField(), gamma, phi, phiGamma, Sp, Su, 1, 0);

    rhoPhi +=
        (runTime.deltaT()/totalDeltaT)
      *(phiGamma*(rho1 - rho2) + phi*rho2);
  }

  Info<< "Liquid phase volume fraction = "
      << gamma.weightedAverage(mesh.V()).value()
      << "  Min(gamma) = " << min(gamma).value()
      << "  Max(gamma) = " << max(gamma).value()
      << endl;
}
```

MULES:implicitSolve(oneField(),gamma, phi, phiGamma, Sp, Su, 1,0) asks for:

gamma: actual value of γ to be solved

phi: normal convective flux

phiGamma: $=\gamma(1-\gamma)U\gamma$

Sp: implicit source term

Su: divergence term

1,0: maximum and minimum value of γ

In the UEqn.H the discretized momentum equation is solved to compute an intermediate velocity field. The first part of the UEqn.H is the left hand side of the momentum equation. Term fvm::Sp(fvc::ddt(rho)+fvc::div(rhoPhi),U) takes the form of Sp(a,b), it takes a scalar linearized field for the first argument and the field which are solved for as the second. It is used to create a larger diagonal term (to aid the solver) so it only can be used if the linearization of the source term has a negative dependency on the variable being solved for. In the second part the algorithm solves the LHS of the momentum equation to be equal to the gravity and surface tension forces.

```
/*****************UEqn.H*****************/
surfaceScalarField muf =
      twoPhaseProperties->muf()
    + fvc::interpolate(rho*turbulence->nuSgs());

  fvVectorMatrix UEqn
```

```
    (
        fvm::ddt(rho, U)
      + fvm::div(rhoPhi, U)
      - fvm::Sp(fvc::ddt(rho) + fvc::div(rhoPhi), U)
      - fvm::laplacian(muf, U)
      - (fvc::grad(U) & fvc::grad(muf))
    //- fvc::div(muf*(fvc::interpolate(dev2(fvc::grad(U))) & mesh.Sf()))
    );

    UEqn.relax();

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

In the pEqn.H, the mass fluxes at the cells faces are calculated and the pressure equation is thereafter solved, then the mass fluxes at the cell faces are corrected by the prescribed number of inner PISO loops. After performing the momentum corrector step on the basis of the new pressure field, the continuity error is computed. Repeat from solving the momentum equation for the prescribed number of times of the outer corrector loop.

```
/**********************pEqn.H**********************/
{
    volScalarField rUA = 1.0/UEqn.A();
    surfaceScalarField rUAf = fvc::interpolate(rUA);

    U = rUA*UEqn.H();

    surfaceScalarField phiU
    (
        "phiU",
```

```
    (fvc::interpolate(U) & mesh.Sf())
  + fvc::ddtPhiCorr(rUA, rho, U, phi)
);

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

adjustPhi(phi, U, pd);

Pair<tmp<volScalarField> > vDotP = twoPhaseProperties->vDotP();
const volScalarField& vDotcP = vDotP[0]();
const volScalarField& vDotvP = vDotP[1]();

for(int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
{
    fvScalarMatrix pdEqn
    (
        fvc::div(phi) - fvm::laplacian(rUAf, pd)
      + (vDotvP - vDotcP)*(rho*gh - pSat) + fvm::Sp(vDotvP - vDotcP, pd)
    );

    pdEqn.setReference(pdRefCell, pdRefValue);

    if (corr == nCorr-1 && nonOrth == nNonOrthCorr)
    {
        pdEqn.solve(mesh.solver(pd.name() + "Final"));
    }
    else
    {
        pdEqn.solve(mesh.solver(pd.name()));
    }

    if (nonOrth == nNonOrthCorr)
    {
        phi += pdEqn.flux();
    }
}

p = pd + rho*gh;
```

```
    U += rUA*fvc::reconstruct((phi - phiU)/rUAf);
    U.correctBoundaryConditions();
}
```

All the variables used in the mass transfer models are defined in

$FOAM_APP/solvers/multiphase/interPhaseChangeFoam/phaseChangeTwoPhaseMixtures/PhaseChangeTwoPhaseMixture.H. According to the definition, in the following mass transfer models implementations, mDotAlphal() returns the mass condensation and vaporization rates as a coefficient to multiply (1-alphal) for the condensation rate and a coefficient to multiply alphal for the vaporization rate; while mDotP() returns the mass condensation and vaporization rates as an explicit term for the condensation rate and a coefficient to multiply (p-pSat) for the vaporization rate.

Mass transfer models are implemented under

$FOAM_APP/solvers/multiphase/interPhaseChangeFoam/phaseChangeTwoPhaseMixtures/<Model>/<Model>.C.

```
/* * * * * * * * * * * * * * * * * Merkle.C * * * * * * * * * * * * * * */
// * * * * * * * * * * * * * * * Constructors * * * * * * * * * * * * * //
Foam::phaseChangeTwoPhaseMixtures::Merkle::Merkle
(
    const volVectorField& U,
    const surfaceScalarField& phi,
    const word& alpha1Name
)
:
    phaseChangeTwoPhaseMixture(typeName, U, phi, alpha1Name),

    UInf_(phaseChangeTwoPhaseMixtureCoeffs_.lookup("UInf")),
    tInf_(phaseChangeTwoPhaseMixtureCoeffs_.lookup("tInf")),
    Cc_(phaseChangeTwoPhaseMixtureCoeffs_.lookup("Cc")),
    Cv_(phaseChangeTwoPhaseMixtureCoeffs_.lookup("Cv")),

    p0_("0", pSat().dimensions(), 0.0),
    mcCoeff_(Cc_/(0.5*sqr(UInf_)*tInf_)),
    mvCoeff_(Cv_*rho1()/(0.5*sqr(UInf_)*tInf_*rho2()))
{
    correct();
}
// * * * * * * * * * * * * * * Member Functions * * * * * * * * * * * * * //
Foam::Pair<Foam::tmp<Foam::volScalarField> >
Foam::phaseChangeTwoPhaseMixtures::Merkle::mDotAlphal() const
{
    const volScalarField& p = alpha1_.db().lookupObject<volScalarField>("p");
```

```
    return Pair<tmp<volScalarField> >
    (
        mcCoeff_*max(p - pSat(), p0_),
        mvCoeff_*min(p - pSat(), p0_)
    );
}

Foam::Pair<Foam::tmp<Foam::volScalarField> >
Foam::phaseChangeTwoPhaseMixtures::Merkle::mDotP() const
{
    const volScalarField& p = alpha1_.db().lookupObject<volScalarField>("p");
    volScalarField limitedAlpha1 = min(max(alpha1_, scalar(0)), scalar(1));

    return Pair<tmp<volScalarField> >
    (
        mcCoeff_*(1.0 - limitedAlpha1)*pos(p - pSat()),
        (-mvCoeff_)*limitedAlpha1*neg(p - pSat())
    );
}
```

/ * * * * * * * * * * * * * * * * Kunz.C * * * * * * * * * * * * * * * /

```
// * * * * * * * * * * * * * * * Constructors * * * * * * * * * * * * * * //
Foam::phaseChangeTwoPhaseMixtures::Kunz::Kunz
(
    const volVectorField& U,
    const surfaceScalarField& phi,
    const word& alpha1Name
)
:
    phaseChangeTwoPhaseMixture(typeName, U, phi, alpha1Name),

    UInf_(phaseChangeTwoPhaseMixtureCoeffs_.lookup("UInf")),
    tInf_(phaseChangeTwoPhaseMixtureCoeffs_.lookup("tInf")),
    Cc_(phaseChangeTwoPhaseMixtureCoeffs_.lookup("Cc")),
    Cv_(phaseChangeTwoPhaseMixtureCoeffs_.lookup("Cv")),

    p0_("0", pSat().dimensions(), 0.0),
    mcCoeff_(Cc_*rho2()/tInf_),
    mvCoeff_(Cv_*rho2()/(0.5*rho1()*sqr(UInf_)*tInf_))
{
    correct();
```

```
}
// * * * * * * * * * * * * * * Member Functions * * * * * * * * * * * * * * //
Foam::Pair<Foam::tmp<Foam::volScalarField> >
Foam::phaseChangeTwoPhaseMixtures::Kunz::mDotAlphal() const
{
    const volScalarField& p = alpha1_.db().lookupObject<volScalarField>("p");
    volScalarField limitedAlpha1 = min(max(alpha1_, scalar(0)), scalar(1));

    return Pair<tmp<volScalarField> >
    (
        mcCoeff_*sqr(limitedAlpha1)
       *max(p - pSat(), p0_)/max(p - pSat(), 0.01*pSat()),

        mvCoeff_*min(p - pSat(), p0_)
    );
}

Foam::Pair<Foam::tmp<Foam::volScalarField> >
Foam::phaseChangeTwoPhaseMixtures::Kunz::mDotP() const
{
    const volScalarField& p = alpha1_.db().lookupObject<volScalarField>("p");
    volScalarField limitedAlpha1 = min(max(alpha1_, scalar(0)), scalar(1));

    return Pair<tmp<volScalarField> >
    (
        mcCoeff_*sqr(limitedAlpha1)*(1.0 - limitedAlpha1)
       *pos(p - pSat())/max(p - pSat(), 0.01*pSat()),

        (-mvCoeff_)*limitedAlpha1*neg(p - pSat())
    );
}
```

Note that different from the theoretical governing equations of the Merkle and Kunz mass transfer models, the implementations use 0.01*pSat instead of a value 0 to trigger the condensation and vaporization constants to be active.

```
/ * * * * * * * * * * * * * * SchnerrSauer.C* * * * * * * * * * * * * * /
// * * * * * * * * * * * * * * * Constructors * * * * * * * * * * * * * * //
Foam::phaseChangeTwoPhaseMixtures::SchnerrSauer::SchnerrSauer
(
    const volVectorField& U,
    const surfaceScalarField& phi,
    const word& alpha1Name
)
```

```
:
    phaseChangeTwoPhaseMixture(typeName, U, phi, alpha1Name),

    n_(phaseChangeTwoPhaseMixtureCoeffs_.lookup("n")),
    dNuc_(phaseChangeTwoPhaseMixtureCoeffs_.lookup("dNuc")),
    Cc_(phaseChangeTwoPhaseMixtureCoeffs_.lookup("Cc")),
    Cv_(phaseChangeTwoPhaseMixtureCoeffs_.lookup("Cv")),

    p0_("0", pSat().dimensions(), 0.0)
{
    correct();
}

// * * * * * * * * * * * * * * Member Functions * * * * * * * * * * * * * * //
Foam::tmp<Foam::volScalarField>
Foam::phaseChangeTwoPhaseMixtures::SchnerrSauer::rRb
(
    const volScalarField& limitedAlpha1
) const
{
    return pow
    (
        ((4*mathematicalConstant::pi*n_)/3)
       *limitedAlpha1/(1.0 + alphaNuc() - limitedAlpha1),
        1.0/3.0
    );
}

Foam::dimensionedScalar
Foam::phaseChangeTwoPhaseMixtures::SchnerrSauer::alphaNuc() const
{
    dimensionedScalar Vnuc = n_*mathematicalConstant::pi*pow3(dNuc_)/6;
    return Vnuc/(1 + Vnuc);
}


Foam::tmp<Foam::volScalarField>
Foam::phaseChangeTwoPhaseMixtures::SchnerrSauer::pCoeff
(
    const volScalarField& p
) const
{
    volScalarField limitedAlpha1 = min(max(alpha1_, scalar(0)), scalar(1));
```

```
    volScalarField rho =
        (limitedAlpha1*rho1() + (scalar(1) - limitedAlpha1)*rho2());

    return
        (3*rho1()*rho2())*sqrt(2/(3*rho1()))
       *rRb(limitedAlpha1)/(rho*sqrt(mag(p - pSat()) + 0.01*pSat()));
}


Foam::Pair<Foam::tmp<Foam::volScalarField> >
Foam::phaseChangeTwoPhaseMixtures::SchnerrSauer::mDotAlphal() const
{
    const volScalarField& p = alpha1_.db().lookupObject<volScalarField>("p");
    volScalarField limitedAlpha1 = min(max(alpha1_, scalar(0)), scalar(1));

    volScalarField pCoeff = this->pCoeff(p);

    return Pair<tmp<volScalarField> >
    (
        Cc_*limitedAlpha1*pCoeff*max(p - pSat(), p0_),

        Cv_*(1.0 + alphaNuc() - limitedAlpha1)*pCoeff*min(p - pSat(), p0_)
    );
}

Foam::Pair<Foam::tmp<Foam::volScalarField> >
Foam::phaseChangeTwoPhaseMixtures::SchnerrSauer::mDotP() const
{
    const volScalarField& p = alpha1_.db().lookupObject<volScalarField>("p");
    volScalarField limitedAlpha1 = min(max(alpha1_, scalar(0)), scalar(1));

    volScalarField apCoeff = limitedAlpha1*pCoeff(p);

    return Pair<tmp<volScalarField> >
    (
        Cc_*(1.0 - limitedAlpha1)*pos(p - pSat())*apCoeff,

        (-Cv_)*(1.0 + alphaNuc() - limitedAlpha1)*neg(p - pSat())*apCoeff
    );
}
```

1.4 Modification to the original code

In order to create a user modified version of the solver, it is suggested that the user copy the original solver to $WM_PROJECT_USER_DIR/applications/solvers meanwhile rename it to ooodlesInterPhaseChange for further modifications which are described below.

First, a slight modification was suggested to better prove the near wall behavior of the code by modifying the wall viscosity. The law implemented is Spalding.

$$y^+ = u^+ + \frac{1}{e^{\kappa B}}\left\{e^{\kappa u^+} - \left(1 + \kappa u^+ + \frac{\kappa^2 u^{+2}}{2} + \frac{\kappa^3 u^{+3}}{6}\right)\right\}$$

where kappa and B are model constants.

This is done in a user created header file wallViscosity.H put under the main solver directory which is executed before solving the momentum equation.

The other modification was made to the Kunz mass transfer model implementation. In the source code ooodlesInterPhaseChange/phaseChangeTwoPhaseMixtures/Kunz.C, the mass condensation rate and vaporization rate coefficient are computed as the following:

```
mDotAlphal()
        return Pair<tmp<volScalarField> >
        (
         mcCoeff_*sqr(limitedAlpha1)
        *max(p - pSat(),p0_)/max(p - pSat(), 0.001*mag(pSat())),
        //*max(p - pSat(), p0_)/max(p - pSat(), 0.01*pSat()),
        mvCoeff_*min(p - pSat(), p0_)
        );
mDotP()
        return Pair<tmp<volScalarField> >
        (
         mcCoeff_*sqr(limitedAlpha1)*(1.0 - limitedAlpha1)
         *pos(p - pSat())/max(p - pSat(),0.001*mag(pSat())),
        //*pos(p - pSat())/max(p - pSat(), 0.01*pSat()),
        (-mvCoeff_)*limitedAlpha1*neg(p - pSat())
         );
```

The condensation and vaporization coefficients, mcCoeff and mvCoeff, are implemented in such a way that when the local pressure is lower than the vaporization pressure pSat, the condensation rate becomes active, while the pressure is lower than pSat, the vaporization rate is triggered. The modification is made due to the fact that using a 0 value for the reference pressure will require the pSat to be negative. Therefore the magnitude of the vaporization is taken instead of the value.

1.5 Compile the code

Copy the source code to the user's working directory

cp ooodlesInterPhaseChange.tar $WM_PROJECT_USER_DIR/application/solvers

tar xvf ooodlesInterPhaseChange.tar

Modify the *Make/files* to write the executable in $FOAM_USER_APPBIN

EXE = $(FOAM_USER_APPBIN)/ooodlesInterPhaseChange

Compile the code

wclean

rm -r Make/linux*

wmake

2. A test case

A test case *naca15_test_case* is provided to the users. Copy the case to the working directory

cp naca15_test_case.tar $WM_PROJECT_USER_DIR/run

tar xvf naca15_test_case.tar

As usual the case contains three folders *0/*, *constant/* and *system/*.

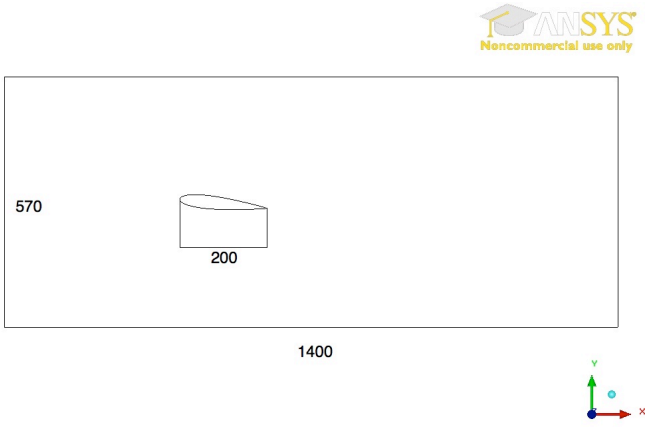
2.1 Mesh description

With the chord length c=200mm. the NACA0015 profile is rotated 6° around the center of gravity and set in the domain of 1400 x 570 mm, extending 2 chord lengths ahead of the leading edge, ending 4 chord lengths behind the trailing edge (in relation to 0° angle of attack) and with a vertical extent reflecting the size of the cavitation tunnel. The grid is a C-grid type consisted of 53,478 cells generated by ANSYS ICEMCFD with one cell in the z direction saved in a Fluent mesh format. To use a converter to convert the mesh into FOAM mesh one should have the fluent.msh file under the case directory.

fluentMeshToFoam fluent.msh –scale 0.001

A scale factor of 0.001 is applied since the mesh is generated in mm in ICEMCFD and OpenFOAM uses the standard SI unit meter.

Figure 1 and 2 show respectively the view of the whole domain and a close-up view around the foil.



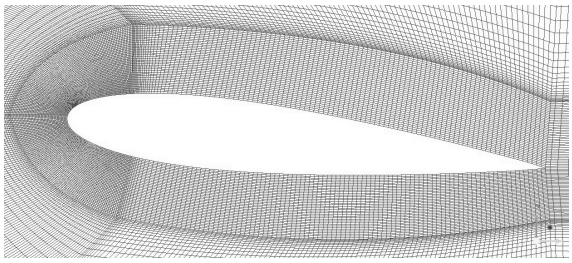


Figure 1 Computational domain

Figure 2 Grid around the wing

2.2 Boundary and initial conditions

Boundary conditions are defined in the */constant/polyMesh/boundary* file after the convertion is done. There are five boundary entries in the the *boundary* file. Proper type are chosen in order to set a fixed velocity of 6m/s at the inlet and fixed pressure of 0 Pa at the outlet. A fixed very close to one is given as the initial condition for gamma for numerical reasons. Top and bottom boundaries are set as symmetryPlane with zeroGradient for all the quantities to mimic no flux through the interface, front and back boundaries to be empty by default.

2.3 Fluid properties

In the *constant* directory locate three property files, *environmentalProperties transportProperties* and *LESProperties*.

The environmentalProperties dictionary specifies the gravity acceleration vector, in this case it is neglected therefore the value is set to be (0,0,0).

The *transportProperites* file specifies the material properties for each fluid, separated into two subdictionaries *phase1* (water, corresponding to γ=1) and *phase2* (vapour, corresponding to γ=0). Densities are specified under the keyword *rho* while kinematic viscosity is under keyword *nu*. The viscosity parameters for the other models are specified with reference to the tutorial *Breaking of a dam*. The mass transfer model is specified under the keyword *phaseChangeTwoPhaseMixture* and the model coefficients are determined in the subdictionary *<model>Coeffs*. In this test case the Kunz mass transfer model is selected. The mass transfer model constants are chosen under the principle of as strong as possible but since these are practical parameters they should always be checked and tuned to better couple the pressure field with the vapor field. The vapourization pressure is defined in the *cavitation* subdictionary under the keyword *pSat*, *startN* controls the number of time steps ran before the cavitation source is turned on, while *rampN* controls the number of time steps during which the cavitation source is added into the continuity equation to assure numerical stability. By specifying *yes* under the keyword *restart*, the simulation of caviatating flow can be continued with the values of *startN* and *rampN* being neglected (which means the source term is already activated).

The *LESProperties* determines the subgrid model under the keyword *LESModel* and the model coefficients are specified in the *<model>Coeffs* subdictionaries. Note that in this test case the *LESModel* entry is specified to be *laminar*, which implies that the subgrid model employed here is an implicit LES model, considering the action of the subgrid scale is equivalent to a strictly dissipative action, and letting the leading order truncation error in the discretization of the fluxes emulate the energy dissipation.

2.4 Time step control

Courant number has a significant impact on the reliability and stability of the unstable flow simulation. Recommended by OpenFOAM, the upper limit of the *Co* should be around 0.2. Therefore a detlaT of 2e-05 seconds is specified in the */system/controDict* file. Note that the cavitating flow calculation should always be started from a converged wetted flow (field. When the cavitation is toggled on, simulation needs to be restarted from the latest resolution.

2.5 Discretisation schemes

The free surface treatment in OpenFOAM does not account for the effects of turbulence. All free surface simulations can be viewed as a direct numerical simulation (DNS) of fluid flow. Therefore there is a high requirement for the mesh resolution of the interface.

The solver uses the multidimensional universal limiter for explicit solution (MULES) method, to maintain boundedness of the phase fraction independent of underlying numerical scheme, mesh structure, etc.. The choice of schemes for convection is therefore not restricted to those that are strongly stable or bounded, e.g. upwind differencing.

2.6 Solution and algorithm control

The /system/*fvSolution* file controls the equation solvers, tolerance and algorithms. In this case it contains two subdictionaries: *solvers* and *PISO*.

In the first subdictionary, *solver*, each linear-solver used for each discretised equation is specified. This is done by specifying the solver of each variable being solved, in this case which are: *pcorr* (pressure corrector), *pd*, *pdFinal* (the final value of pressure after the correction), *U*, *k*, *B*, *nuTilda*, and *gamma*. The variable name is followed by the solver name and a dictionary containing the parameters that the solver uses. The pressure variables and turbulence quantities are solved by solver Preconditioned (bi-) conjugate gradient (*PCG*/*PBiCG*), velocity is solved by a solver using a smoother (*smoothSolver*), and volume fraction gamma by *MULESImplicit* which is described already in section 2.5. Solver tolerance, *tolerance*, and ratio of current to initial residuals, *relTol*, are specified afterwards. The solver stops if either of the tolerance falls below the specified value. Preconditioning of matrices in the conjugate gradient solvers is specified by the keyword *preconditioner*. In our test case the Geometric-algebraic multi-grid (*GAMG*) method and Diagonal incomplete-LU (DILU) is used. The principle of GAMG is to generate a quick

solution on a mesh with a small number of cells, map this solution onto a finer mesh, use it as an initial guess to obtain an accurate solution on the fine mesh. It starts with the mesh specified by the user and coarsens/refines the mesh in stages. The user is required to specify an approximate mesh size at the most coarse level in terms of the number of cells *nCellsInCoarsetestLevel*. The agglomeration of cells is performed by the algorithm specified by the *agglomerator* keyword and *faceAreaPair* method is used as recommended. Agglomeration can be cached by the *cacheAgglomeration* switch. Smoothing is specified by the *smoother*, generally *GaussSeidel* is the most reliable option, but for bad matrices *DIG* offers better convergence. The number of sweeps used by the smoother at different levels of mesh density are specified by the *nPreSweeps*, *nPostSweeps* and *nFinestSweeps*. The *nPreSweeps* entry is used as the algorithm is coarsening the mesh, is *nPostSweeps* used as the algorithm is refining, and *nFinestSweeps* is used when the solution is at its finest level. The *mergeLevels* keyword controls the speed at which coarsening or refinement levels is performed. It is often best to do so only at one level at a time by setting the level of 1.

In the second subdictionary, *PISO*, which is an algorithm of iterative procedures for solving equations for velocity and pressure of transient problems, the number of correctors is specified by keyword *nCorrectors*. To account for mesh non-orthogonality, the number of non-orthogonal correctors is specified by the *nNonOrthogonalCorrectors* keyword. A mesh is orthogonal if, for each face within it, the face normal is parallel to the vector between the centres of the cells that the face connects, e.g. a mesh of hexahedral cells whose faces are aligned with a Cartesian coordinate system. *nGammaCorr* specifies the number of correctors to the loop around the gamma equation. Of particular interest are the *nGammaSubCycles* and *cGamma*. *nGammaSubCycles* represents the number of sub-cycles within the gamma equation; sub-cycles are additional solutions to an equation within a given time step. It is used to enable the solution to be stable without reducing the time step and vastly increasing the solution time. Here we specify 4 sub-cycles, which means that the gamma equation is solved in 4 × quarter length time steps within each actual time step. The *cGamma* keyword is a factor that controls the compression of the interface. In this case it is chosen to be enhanced compression by a value of 2.

2.7 Running the code

Go to the case directory *$FOAM_RUN/naca15_test_case* and run the command:

ooodlesInterPhaseChange &> log

The simulation of wetted flow should be long enough to get the stabilized pressure distribution. To be able to judge if the solution is stabilized or not, one can add some probes in the domain or plot lift/drag coefficient with time sequence. This can be realized by adding functions in the *system/controlDict* file.

2.8 Post-processing

The post-processing is done by Fieldview. Firstly the FOAM data needs to be converted to a readable format for Fieldview by

foamToFieldview9

Different computational surfaces and plots can be created in Fieldview to visualize the flow field. In the following section a 2D plot of the pressure coefficient along the hydrofoil is examined, and the computational surface of the volume fraction gamma is created to illustrate the development of the cavity by the transient data control.

2.8.1 Wetted Flow

Figure 3 illustrates the pressure coefficient distribution along the foil of the wetted flow. The pressure coefficient is defined as

$$C_p = \frac{p - p_{ref}}{\frac{1}{2}\rho U_\infty^2}$$

The lower side of the closed curve represents the suction side and the upper part of the curve represents the pressure side of the foil.

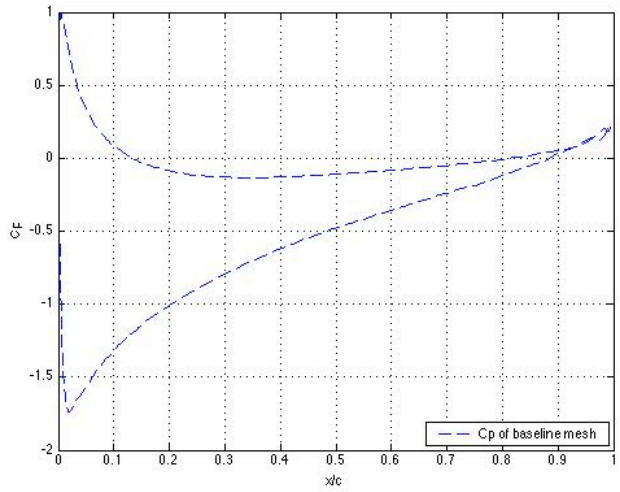


Figure 3 Cp distribution of wetted flow

2.8.2 Cavitating Flow

The next step is to determine at which cavitation number the cavitating flow simulation should be carried out.

By definition, cavitation number: $\sigma = \dfrac{P_{ref} - p_v}{\frac{1}{2}\rho U_\infty^2}$

where P_{ref} is the reference pressure, p_v is the vaporization pressure, U_∞ is the reference velocity and ρ is the density of the liquid. In our test case, the reference pressure takes the value of outlet pressure 0 Pa and reference velocity is the inlet velocity 6 m/s. Usually in experiments, to achieve the desired cavitation number, the pressure is decreased gradually but in computations vaporization pressure is tuned to match the chosen cavitation number, in our test case σ=1 which is reached by setting pSat to be -18000 Pa.

Many interesting features of cavitating flow are able to be captured by the computations, such as re-entrant jets and periodic shedding. A sequence of instantaneous vapour volume fraction illustrating the periodic shedding of cavitation is presented in *Figure 4*. A sheet cavity starts at the leading edge, whereby the cavity is transported along the surface of the foil and when it exceeds a certain size, it becomes unstable and sheds periodically. This process is controlled by a re-entrant jet which is forming downstream of the cavity and travels in the opposite direction to the outer flow towards the leading edge. The re-entrant jet cuts the cavity resulting in a shedding of cloud cavitation which is transported with the free stream towards the trailing edge.

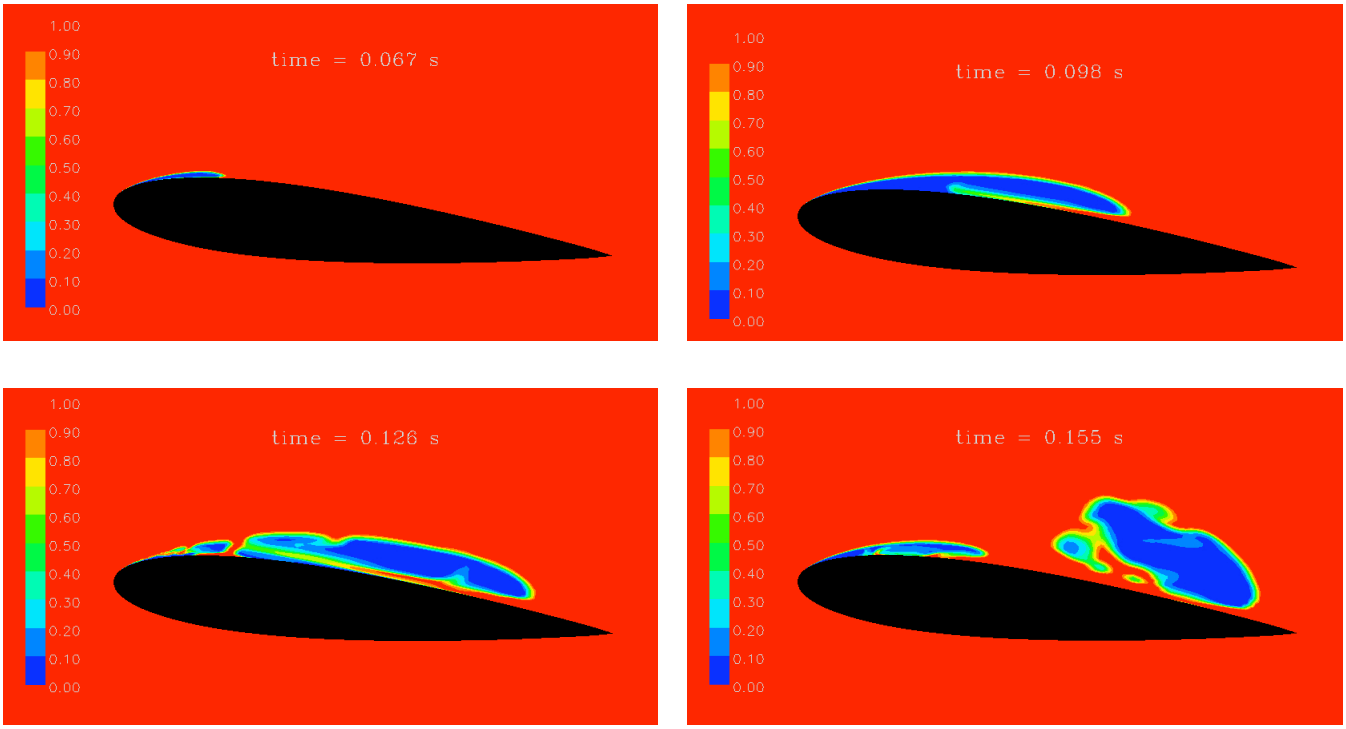


Figure 4 Development of cavities by FIELDVIEW