December 21, 2017

Implementation of cavitation models into the multiphaseEulerFoam solver

Surya Kaundinya Oruganti

Laboratories Mechanical Des Fluides Et D'acoustique
Ecole Central de Lyon
and
Volvo Group
Lyon, France

December 21, 2017

Outline

Overview of multiphase solvers

1. VOF method
   - applicable for segregated flows where sharp interfaces between phases exits.
   - Base solver : interFoam
2. Eulerian multifluid method
   - applicable for inter-dispersed flows with high volume fraction of both the phases.
   - Base solver : twoPhaseEulerFoam
3. Eulerian -Lagrangian method
   - This approach is used for dispersed flows where the dispersed phase volume fraction is very small.
   - Base solver : sprayFoam
4. Hybrid VOF - multifluid method
   - This approach is used for modelling both segregated and dispersed flows.
   - Base solver : multiphaseEulerFoam

VOF method

1. The flow is represented as a fluid mixture, i.e both phases have same flow velocity. Therefore it solves for a single momentum equation with an additional surface tension force (F_s) acting at interface separating the phases.

$$\frac{\partial \rho U}{\partial t} + \nabla.(\rho U U) = -\nabla(p) + \nabla.(\mu(\nabla(U) + \nabla(U)^T)) + \rho g + F_s$$

$$F_s = \sigma \kappa(x) n, \quad n = \nabla\alpha / mag(\nabla\alpha), \quad \kappa(x) = \nabla.n$$

2. The interface sharpening method of Weller is used to solve the phase fraction equation. An additional interface compression term is added to the equation to compress the volume fraction field and maintain a sharp interface.

$$\frac{\partial \alpha}{\partial t} + \nabla.(\alpha U) + \nabla.(\alpha(1-\alpha)U_r) = Sp.\alpha + Su$$

$$U_r = C_\alpha \frac{\nabla\alpha}{mag(\nabla\alpha)}$$

Eulerian multifluid method

1. Each phase is represented by separate set of flow equations.
2. The two phases are coupled by the mass, momentum and energy transfer terms between the phases. For simplicity, only the drag forces between the phases is considered for this study.

$$\frac{\partial \alpha_k}{\partial t} + \nabla.(\alpha_k.u_k) = 0$$

$$\frac{\partial(\rho_k \alpha_k u_k)}{\partial t} + (\rho_k \alpha_k u_k.\nabla)(u_k) = -\alpha_k \nabla p + \nabla.(\mu_k \alpha_k \nabla u_k) + F_g + F_D$$

3. Different drag force models have been implemented in OpenFOAM. The general formulation of the drag force is given as,

$$F_D = C_d(U_r)$$

Hybrid VOF-Eulerian multifluid method

1. Basically it is a Eulerian multifluid solver for n-phases with separate flow equations for each phase.

$$\frac{\partial(\rho_k \alpha_k u_k)}{\partial t} + (\rho_k \alpha_k u_k . \nabla)(u_k) = -\alpha_k \nabla p + \nabla . (\mu_k \alpha_k \nabla u_k) + F_g + F_D k + F_S k$$

2. Additionaly, for selected phase pairs, an interface tracking approach can be switched on to solve the phase transport equation similar to VOF method.

$$\frac{\partial \alpha_k}{\partial t} + u_k . \nabla(\alpha_k) + \nabla . (\alpha_k (1 - \alpha_k) u_r) = 0$$

$$u_r = C_\alpha \frac{\nabla \alpha}{mag(\nabla \alpha)}$$

3. C_α is used as a binary switch to turn on or off the interface sharpening.
4. $C_\alpha = 0$ results in solution of purely dispersive flow according to the multi-fluid model.
5. $C_\alpha = 1$, results in solution of segregated flow using the VOF approach.

Overview

1. The main solver related files are listed below:

```
CourantNo.H
CreateFields.H
multiphaseEulerFoam.C
pEqn.H
UEqns.H
TEqns.H
```

2. Additionaly a transport model for the multiphase(generic n-phase) has been defined using *multiphaseSystem* library. The main files in the library are lister below:

```
multiphaseSystem.H  multiphaseSystem.C
phaseModel.H  phaseModel.C
diameterModels
```

3. The different source terms defining the interaction between the phases like drag force or the heat transfer are defined using the *interfacialModels* library.

```
dragModels
heatTransferModels
```

multiphaseEulerFoam.C

1. The main files included in the solver are :

```
#include "multiphaseSystem.H"
#include "phaseModel.H"
#include "dragModel.H"
```

2. The main solver algorithm is shown below :

```
while (pimple.loop())
{
    // Solves the turbulence equation and corrects the viscosity
    turbulence->correct();
    // Solves the phase transport equations of each phase
    fluid.solve();
    // Calculates the mixture density
    rho = fluid.rho();
    // Momentum Predictor step for each phase
    #include "UEqns.H"
    // --- Pressure corrector loop
    while (pimple.correct())
    {
        #include "pEqn.H"
    }
}
```

3. Where *fluid* is an object of the multiphaseSystem class and is defined in createFields.H file.

```
multiphaseSystem fluid(U, phi);
```

dambreak4phase

- For easy understanding of the solver, a dambreak case with four different phases i.e. air, oil, water and mercury is presented as shown below:

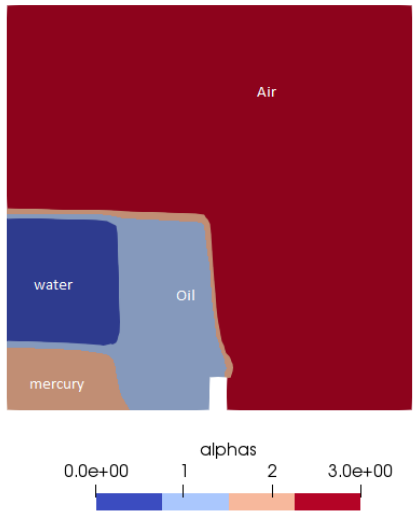

Figure: Initial condition

transportProperties

- The transport properties can be divided in two sections. The first section defines the individual phase properties as shown below :

```
phases
(
    water
    {
        nu          1e-06;
        kappa       1e-06;
        Cp          4195;
        rho         1000;
    }
    oil
    {
        nu          1e-06;
        kappa       1e-06;
        Cp          4195;
        rho         500;
    }
    mercury
    {
        nu          1.125e-07;
        kappa       1e-06;
        Cp          4195;
        rho         13529;
    }
    air
    {
        nu          1.48e-05;
        kappa       2.63e-2;
        Cp          1007;
        rho         1;
    }
);
```

transportProperties contd...

- In the second part the interface properties like drag force and phase pairs where interface tracking are required is defined.

```
interfaceCompression
(
    (air water)     1
    (air oil)       1
    (air mercury)   1
);
drag
(
    (air water)
    {
        type SchillerNaumann;
        residualPhaseFraction 1e-3;
        residualSlip 1e-3;
    }
    (air oil)
    {
        type blended;
        ....
        ....
    }
)
```

- The interfaceCompression is the switch (C_α)which defines whether for a given phase pair we use VOF or not.
- Here as the interface between air(atmosphere) and all other fluids is well defined. So we use interface capturing only for interface between air and other fluids.
- Also a different drag models can be chosen for different phase interface.

Result

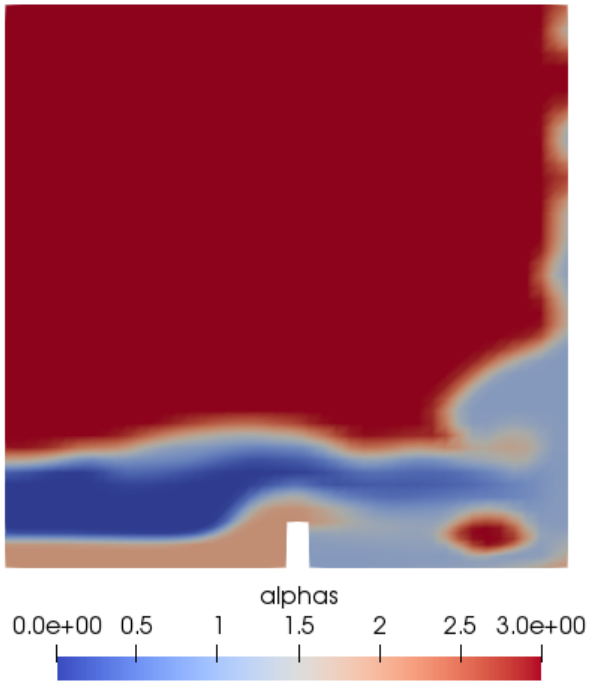

Figure: t = 0.5s

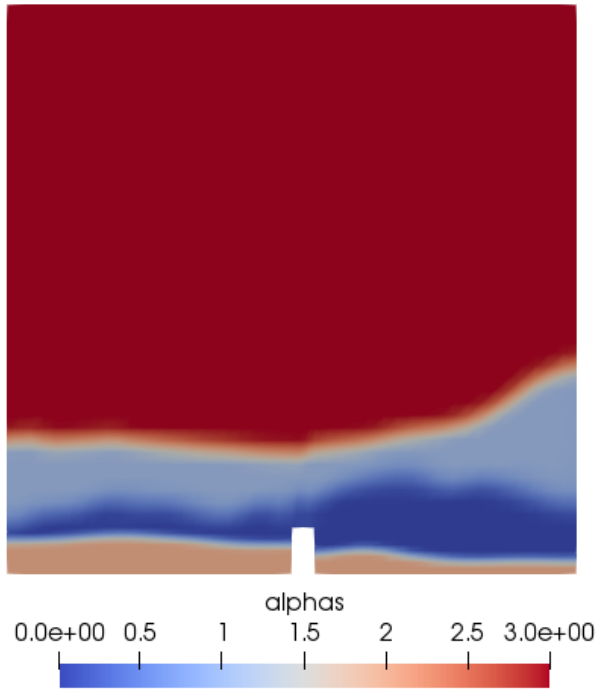

Figure: t = 3s

Interesting points

1. how the solver is extended to consider more than 2 phases?
2. how the interface properties defined for phase pairs implemented into the solver?
3. how the interface tracking is coupled to the phase transport equation only for selected phase pairs?
4. based on above answers, how can the solver be extended for cavitating flows?

multiphaseSystem class

1. The class is defined as :

```
Incompressible multi-phase mixture with built in solution for the
phase fractions with interface compression for interface-capturing.
```

2. The main files included in multiphaseSystem.H are :

```
#include "incompressible/transportModel/transportModel.H"
#include "IOdictionary.H"
#include "phaseModel.H"
#include "dragModel.H"
#include "HashPtrTable.H"
```

3. The phaseModel class is used to read in the user-defined fluid properites of individual phases.

```
bool Foam::phaseModel::read(const dictionary& phaseDict)
{
    phaseDict_ = phaseDict;
    {
      phaseDict_.lookup("nu") >> nu_.value();
      phaseDict_.lookup("kappa") >> kappa_.value();
      phaseDict_.lookup("Cp") >> Cp_.value();
      phaseDict_.lookup("rho") >> rho_.value();
      return true;
    }
}
```

interfacePair and HashTable

1. A subclass called *interphasePair* defined in multiphaseSystem.H. This class uses a hash function called symmHash() to read and store the interface properties defined for each individual interface between different phases.

```
class interfacePair : public Pair<word>
{
    class symmHash: public Hash<interfacePair>
    {
    public: symmHash()
        {}
        label operator()(const interfacePair& key) const
        {
         return word::hash()(key.first()) + word::hash()(key.second());
        }
    };
}
```

2. Then any specific property for a given phase pair is later on retrieved using a hash table. How the C_α values are stored in a hash table called $cAlphas$ is shown below:

```
typedef HashTable<scalar, interfacePair, interfacePair::symmHash> scalarCoeffSymmTable;
scalarCoeffSymmTable cAlphas_;
```

Interface compression for selected pairs

1. The *solveAlphas()* function checks the cAlpha value defined for a given phase pair. If cAlpha is defined then it adds the compression flux term . So by default cAlpha is 0.

```
surfaceScalarField phir(phase1.phi() - phase2.phi());

scalarCoeffSymmTable::const_iterator cAlpha
(
    cAlphas_.find(interfacePair(phase1, phase2))
);

if (cAlpha != cAlphas_.end())
{
    surfaceScalarField phic
    (
        (mag(phi_) + mag(phir))/mesh_.magSf()
    );

    phir += min(cAlpha()*phic, max(phic))*nHatf(phase1, phase2);
}
```

MULES

1. The *solveAlphas()* function uses MULES to solve the phase transport equations.
2. MULES is an iterative implementation of the Flux Corrected Transport technique (FCT) , used to guarantee boundedness in the solution of hyperbolic problems.
3. It computes a corrected Flux between an high and a low order scheme solution with a weighting factor λ

$$F_C = F_L + \lambda(F_H - F_L)$$

4. where the weighting factor is calculated so that the value of the net flux in a cell , must be neither greater than a local maximum nor lesser than a local minimum.

MULES in solveAlphas

1. First the corrected flux for each phase is calculated using MULES::limit function.

```
MULES::limit
(
    1.0/mesh_.time().deltaT().value(),
    geometricOneField(),
    phase1,
    phi_,
    alphaPhiCorr,
    zeroField(), zeroField(),
    1, 0,
    true
);
```

2. Then the MULES :: explicit function is used to solve for the phase fraction.

```
MULES::explicitSolve
(
    geometricOneField(),
    phase1,
    alphaPhi,
    zeroField(), zeroField()
);
```

Cavitation Models

1. The liquid-vapor mass transfer (evaporation and condensation) is governed by the vapor transport equation:

$$\frac{\partial \alpha}{\partial t} + \nabla.(\alpha U) = m^+ + m^-$$

2. where m^+, m^- are the vaporization and condensation rates between two phases.They are derived differently by different models.
3. For the current study Kunz model is used. Here the mass transfer across a liquid-vapor interface is assumed to be in dynamic equilibrium. The condensation and vaporzation rates are given by :

$$m^+ = \frac{C_p \rho_v \alpha_l min(0, p - p_v)}{0.5 \rho_l U_\infty^2 t_\infty} \quad m^- = \frac{C_d \rho_v \alpha_l^2 (1 - \alpha_l)}{t_\infty}$$

multiphaseChangeEulerFoam - new solver

1. Copy the multiphaseEulerFoam solver to project directory.

```
cp -r $FOAM_SOLVER/mulitphase/mulitphaseEulerFoam $WM_PROJECT_USER_DIR/applications/multiphaseChangeEule
cd $WM_PROJECT_USER_DIR/applications/multiphaseChangeEulerFoam
```

2. Recompile the multphaseSystem and interfacialModels libraries by making following changes.

```
vi multiphaseSystem/Make/files
LIB = $(FOAM_USER_LIBBIN)/libusermultiphaseSystem

vi interfacialModels/Make/files
LIB = $(FOAM_USER_LIBBIN)/libusercompressibleMultiphaseEulerianInterfacialModels

vi interfacialModels/Make/options
LIB_LIBS = \
    -L$(WM_PROJECT_USER_DIR)/platforms/linux64GccDPInt32Opt/lib/ \
    -lusermultiphaseSystem
```

3. Recompile the solver by making following changes.

```
vi Make/files
EXE = $(FOAM_USER_APPBIN)/libusermultiphaseSystem

vi Make/options
LIB_LIBS = \
    -L$(WM_PROJECT_USER_DIR)/platforms/linux64GccDPInt32Opt/lib/ \
    -lusermultiphaseSystem
    -lusercompressibleMultiphaseEulerianInterfacialModels
```

How to extend the solver for cavitating flows

1. Adding "phaseChangeModels" directory to interfacialModels library
2. Defining the "phaseChangeModel" class, which is the base class for cavitation models.
3. Adding the "Kunz" cavitation model as a derived class with specific implementation of the mass transfer source terms.
4. Adding the mass transfer source terms to phase transport equation solved by solveAlphas() function in multiphaseSystem class.

Adding phaseChangeModels to interfacialModels library

1. Go to the interfacialModels directory and add following listed directories.

```
./phaseChangeModels
  ./phaseChangeModel
      phaseChangeModel.H
      phaseChangeModel.C
      newPhaseChangeModel.C
  ./Kunz
      Kunz.H
      Kunz.C
```

2. Recompile interfacialModels library making following changes:

```
vi Make/files
phaseChangeModels/phaseChangeModel/phaseChangeModel.C
phaseChangeModels/phaseChangeModel/newPhaseChangeModel.C
phaseChangeModels/Kunz/Kunz.C
```

phaseChangeModel - base class

- Constructor of base class

```
Foam::phaseChangeModel::phaseChangeModel
 (
    const word& type,
    const dictionary& interfaceDict,
    const phaseModel& phase1,
    const phaseModel& phase2
 )
 :
   interfaceDict_(interfaceDict),
   phase1_(phase1),
   phase2_(phase2),
   phaseChangeModelCoeffs_(interfaceDict.optionalSubDict(type + "Coeffs")),
   pSat_("pSat", dimPressure, interfaceDict.lookup("pSat"))
   {}
```

- Member functions

```
//- Return the mass condensation and vaporisation rates as a
//  coefficient to multiply (phase2) for the condensation rate
//  and a coefficient to multiply  phase1 for the vaporisation rate
virtual Pair<tmp<volScalarField>> mDotAlphal() const = 0;
```

runTimeSelection mechanism

- Allows easy definition and manupilation of multiple derived classes from a base class.
- declareRunTimeSelectionTable - invoked in phaseChangeModel.H

```
declareRunTimeSelectionTable
(
    autoPtr,
    phaseChangeModel,
    dictionary,
    (
       const dictionary& interfaceDict,
       const phaseModel& phase1,
       const phaseModel& phase2
    ),
    (interfaceDict, phase1, phase2)
);
```

- defineRunTimeSelectionTable - invoked in phaseChangeModel.C

```
namespace Foam
 {
   defineTypeNameAndDebug(phaseChangeModel, 0);
   defineRunTimeSelectionTable(phaseChangeModel, dictionary);
 }
```

- addToRunTimeSelectionTable - invoked in Kunz.C

```
addToRunTimeSelectionTable(phaseChangeModel, Kunz, dictionary);
```

- New - "Selector" function added to newPhaseChangeModel.C

Kunz cavitation model

- Reading the model coefficients from user defined sub-dictionary phaseChangeModelCoeffs.

```
UInf_("UInf", dimVelocity, phaseChangeModelCoeffs_.lookup("UInf") ),
tInf_("tInf", dimTime, phaseChangeModelCoeffs_.lookup("tInf") ),
Cc_("Cc", dimless, phaseChangeModelCoeffs_.lookup("Cc") ),
Cv_("Cv", dimless, phaseChangeModelCoeffs_.lookup("Cv") ),
```

- Calculation of vaporization and condensation rate coefficients used in member functions.

```
mcCoeff_(Cc_*phase2_.rho()/tInf_),
mvCoeff_(Cv_*phase2_.rho()/(0.5*phase2_.rho()*sqr(UInf_)*tInf_))
```

- Definition of the mass transfer source terms.

```
Foam::Pair<Foam::tmp<Foam::volScalarField>>
Foam::phaseChangeModels::Kunz::mDotAlphal() const
{
  const volScalarField& p = phase1_.db().lookupObject<volScalarField>("p");
  volScalarField limitedAlpha1(min(max(phase1_, 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_)
   );
}
```

Accessing phaseChangeModels in multiphaseystem class

- Include phaseChangeModel.H file in multiphaseSystem.H.

  `#include phaseChangeModel.H`

- Similar to cAlphas, hash tables are used to process the data from the phaseChangeModels in multiphaseSystem.

```
// Hash table for the phaseChange models for different phase pairs
typedef HashPtrTable<phaseChangeModel, interfacePair, interfacePair::symmHash>
    pcTable;

pcTable pcModels_;

//- Return the table of phase change models
const pcTable& pcModels() const
{
 return pcModels_;
}
// Hash table for the condensation and evaporation rate coefficients for different phase pairs
typedef HashPtrTable<volScalarField, interfacePair, interfacePair::symmHash>
    cFields;

typedef HashPtrTable<volScalarField, interfacePair, interfacePair::symmHash>
    vFields;

//- Return the table of cCoeffs and vCoeffs for all phase change models
autoPtr<cFields> cCoeffs() const;
autoPtr<vFields> vCoeffs() const;
```

Implementing the source terms in phase transport equation

Let us revisit the formulation of the phase transport equation with source terms.

$$\frac{\partial \alpha}{\partial t} + \nabla.(\alpha U) + \nabla.(\alpha(1-\alpha)U_r) = Sp.\alpha + Su$$

For the liquid phase,

$$Sp.\alpha_l + Su = \alpha_l.m^+ + \alpha_v.m^-$$

$$Sp = m^+$$

$$Su = \alpha_v.m^-$$

On the other hand, for the vapor phase transport equation, the source terms will be negative of the vaporization and condensation rates defined for liquid phase.

$$Sp.\alpha_v + Su = -\alpha_l.m^+ - \alpha_v.m^-$$

So for the vapor phase,

$$Sp = -m^-$$

$$Su = -\alpha_c.m^+$$

Implementing the source terms in phase transport equation

- Here the impelemntation of the Su term for a given phase using pcSu function is shown.

```
tmp<volScalarField> pcSu
(
    const phaseModel& phase,
    const cFields& cCoeffs,
    const vFields& vCoeffs
) const;
```

- The function iterates over the phase model table and the condensation and vaporisation coefficient tables to search if the phase is either evaporating or condensating.

```
pcTable::const_iterator pcIter = pcModels_.begin();
cFields::const_iterator cIter = cCoeffs.begin();
vFields::const_iterator vIter = vCoeffs.begin();
for
(
    ;
    pcIter != pcModels_.end() && cIter != cCoeffs.end() &&     vIter != vCoeffs.end();
    ++pcIter, ++cIter,++vIter
)
```

Implementing the source terms in phase transport equation (2)

- Here in the implementation of code the first phase in the phase pair should always be the liquid and the second phase is the vapor phase.

```
const phaseModel *phasePtr = nullptr;
if
(
    &phase == &pcIter()->phase1()
 || &phase == &pcIter()->phase2()
)
{
  if(&phase == &pcIter()->phase1())
  {
    phasePtr = &pcIter()->phase2();
    const volScalarField alpha = *phasePtr;
    const volScalarField Su = *cIter();
    tpcSu.ref() = Su*alpha;
  }

  if(&phase == &pcIter()->phase2())
  {
    phasePtr = &pcIter()->phase1();
    const volScalarField alpha = *phasePtr;
    const volScalarField Su = *vIter();
    tpcSu.ref()= -Su*alpha;
  }
}
```

- The Sp term is also implemented on similar lines by pcSp function.

Implementing the source terms in phase transport equation (3)

- Finally the Su and Sp terms are added to the MULES functions used in solveAlphas().

```
const volScalarField Sp(pcSp(phase, cCoeffs(), vCoeffs()));
const volScalarField Su(pcSu(phase, cCoeffs(), vCoeffs()));

MULES::limit
(
    1.0/mesh_.time().deltaT().value(),
    geometricOneField(),
    phase,
    phi_,
    alphaPhiCorr,
    Sp(),
    Su(),
    1,
    0,
    true
);
MULES::explicitSolve
(
    geometricOneField(),
    phase,
    alphaPhi,
    Sp(),
    Su()
);
```

throttle - Model

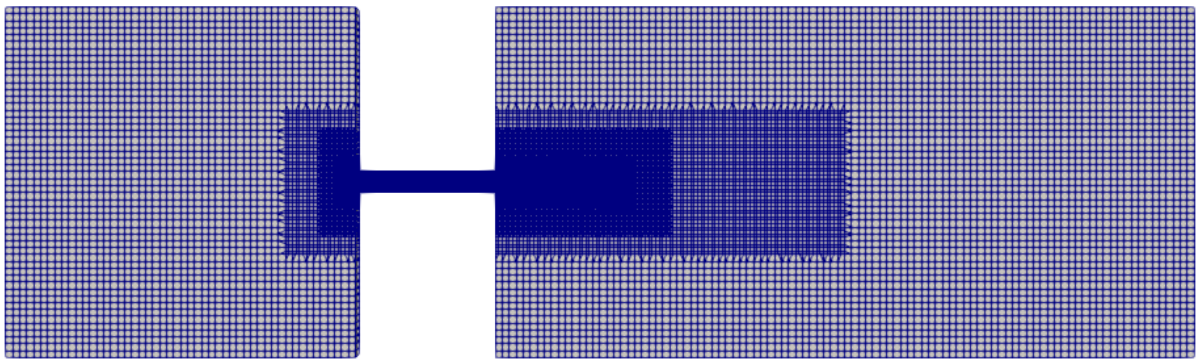

Figure: Mesh with local refinement

The test case uses utilities like refineMesh and topoSet have been used to have finer mesh in regions close to nozzle.

transportProperties and cavitation model

```
phases
(
   water
   {
     nu          3.8e-06;
     kappa       1e-06;
     Cp          4195;
     rho         820;
     diameterModel constant;
     constantCoeffs
     {
        d               1e-3;
     }
   }
  vapor
  {
    nu          4.52e-05;
    kappa       1e-06;
    Cp          4195;
    rho         0.12;
    diameterModel constant;
    constantCoeffs
    {
       d               1e-5;
    }
  }
);
```

```
  sigmas
   (
    (water vapor)       0.03
   );

  interfaceCompression
  (
    (water vapor)       1
  );

phaseChange
  (
    (water vapor)
    {
       type Kunz;
       pSat 2288;
       KunzCoeffs
        {
          UInf             20.0;
          tInf             0.005;
          Cc               1000;
          Cv               1000;
        }
    }
  );
```

Boundary conditions

a) alpha.water

```
internalField   uniform 1;
boundaryField
{
    inlet
    {
     type            fixedValue;
     value           uniform 1;
    }
    outlet
    {
     type            fixedValue;
     value           uniform 1;
    }
    walls
    {
    type            zeroGradient;
    }
    frontback
    {
    type            empty;
    }
}
```

b) alpha.vapor

```
internalField   uniform 0;
boundaryField
{
    inlet
    {
     type            calculated;
     value           uniform 0;
    }
    outlet
    {
     type            calculated;
     value           uniform 0;
    }
    walls
    {
     type            calculated;
     value           uniform 0;
    }
    frontback
    {
    type            empty;
    }
}
```

Boundary conditions (2)

c) prgh

```
internalField   uniform 50e5;
boundaryField
{
  inlet
  {
    type            totalPressure;
    p0              uniform 50e5;
  }
  outlet
  {
   type             fixedValue;
   value            uniform 15e5;
  }
 walls
 {
   type             zeroGradient;
 }
 frontback
 {
   type             empty;
 }
}
```

d) U.water and U.vapor

```
internalField   uniform (0 0 0);
boundaryField
{
inlet
{
    type            zeroGradient;
    value           uniform (0 0 0);
}
outlet
{
    type            zeroGradient;
    value           uniform (0 0 0);
}
walls
{
    type            noSlip;
}
frontback
{
    type            empty;
}
}
```

Results