Division of Combustion
Department of Applied Mechanics
Chalmers University of Technology

CFD with OpenSource software, project

Dynamic mesh refinement in *dieselFoam*

Developed for OpenFOAM-1.6.x
Requires: dieselFoam

Author:
Anne Kösters

Peer reviewed by:
Anton Persson
Jelena Andric

November 3, 2010

Contents

Chapter 1

Dynamic mesh refinement in *dieselFoam*

1.1 Introduction

The idea of this tutorial is to implement a dynamic mesh refinement in the *dieselFoam* solver. The results of simulations of several spray properties are grid dependent. A grid that gives satisfied results with acceptable computation time is needed. For spray simulations the cell size is getting important if a droplet/ parcel or vapor phase occurs in a cell, since in these cells the gas and liquid phase are interacting. A dynamic mesh refinement would allow to start with a relatively coarse mesh what will be refined during the simulation in regions where smaller cells are needed. Hence the computation time would be reduced and the results are less depdendent on the initial cell size. Figure 1.1 shows the results of vapor and liquid penetration using two different meshes, while vapor penetration is given by the continiously increasing curve. As can be seen the results differ using different grid sizes.

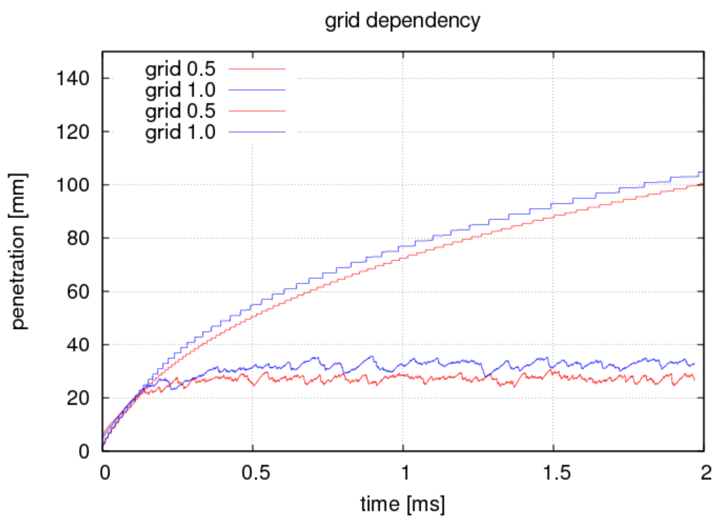

Figure 1.1: Results of vapor and liquid penetration of a Diesel spray using two different grids

1.2 Solvers and Libraries

The solvers and libraries used in this tutorial for comparisons or modifications are:

- *interDyMFoam*
- *dieselFoam*
- *dynamicFvMesh*
- *dieselEngineFoam*

1.3 Mesh refinement in the *interDyMFoam* solver

The *interDyMFoam* solver in OpenFOAM-1.6.x has a mesh refinement implemetation at the phase interface between two different phases. All cells where the phase interface exist are refined. This solver is used as an example how to modify the *dieselFoam* solver. Since the mesh refinement in the *dieselFoam* solver should take place if a droplet or vapor occurs in a cell, the volume fraction of the evaporated fuel will be used to define the cells that should be refined. The mesh refinement in the *interDyMFoam* is achieved using the *dynamicFvMesh* library and is initialized with a function called *update()* (*mesh.update()*), compare the source code of the *interDyMFoam* solver below.

```
#include "fvCFD.H"
#include "dynamicFvMesh.H"
#include "MULES.H"
#include "subCycle.H"
#include "interfaceProperties.H"
#include "twoPhaseMixture.H"
#include "turbulenceModel.H"

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

int main(int argc, char *argv[])
{
    #include "setRootCase.H"
    #include "createTime.H"
    #include "createDynamicFvMesh.H"
    #include "readGravitationalAcceleration.H"
    #include "readPISOControls.H"
    #include "initContinuityErrs.H"
    #include "createFields.H"
    #include "readTimeControls.H"
    #include "correctPhi.H"
    #include "CourantNo.H"
    #include "setInitialDeltaT.H"

    // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
    Info<< "\nStarting time loop\n" << endl;

    while (runTime.run())
    {
        #include "readControls.H"
        #include "CourantNo.H"
```

```
// Make the fluxes absolute
fvc::makeAbsolute(phi, U);

#include "setDeltaT.H"

runTime++;

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

scalar timeBeforeMeshUpdate = runTime.elapsedCpuTime();

// Do any mesh changes
mesh.update();

if (mesh.changing())
{
    Info<< "Execution time for mesh.update() = "
        << runTime.elapsedCpuTime() - timeBeforeMeshUpdate
        << " s" << endl;
}

if (mesh.changing() && correctPhi)
{
    #include "correctPhi.H"
}

// Make the fluxes relative to the mesh motion
fvc::makeRelative(phi, U);

if (mesh.changing() && checkMeshCourantNo)
{
    #include "meshCourantNo.H"
}

twoPhaseProperties.correct();

#include "alphaEqnSubCycle.H"

#include "UEqn.H"

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

turbulence->correct();

runTime.write();

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

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

    return 0;
}
```

The function called *mesh.update()* is defined in the dynamicFvMesh library and can be found in the file
/OpenFOAM/OpenFOAM-1.6.x/src/dynamicFvMesh/dynamicRefineFvMesh/dynamicRefineFvMesh.C.save.

This function defines the mesh refinement and is shown in the appendix. The main steps in this function can be summarized as follows:

- read the dynamicMeshDict (located in the *constant* folder of the case)
- interpolation to get the point values of the field on which the refinement should happen
- definition of all points that are within the refinement range
- definition of cells that should be refined
- if there are cells that should be refined, do the refinement
- definition of points (cells) that will be unrefined
- if there are cells that should be unrefined, do the unrefinement
- return *hasChanged* to the code after the refinement is done

The dictionary that defines some values that are needed for the refinement must be located in the *constant* folder of a case. The dicitionary is called *dynamicMeshDict* and reads:

```
dynamicFvMesh   dynamicRefineFvMesh;

dynamicRefineFvMeshCoeffs
{
    refineInterval  1;
    field           alpha1;
    lowerRefineLevel 0.001;
    upperRefineLevel 0.999;
    unrefineLevel   10;
    nBufferLayers   1;
    maxRefinement   2;
    maxCells        200000;
    correctFluxes
    (
        (
            phi
            U
        )
    );
    dumpLevel       true;
}
```

The field that is used to define the cells that should be refined is *alpha1*. The refine interval has to be greater or equal to one. The lower and upper refine level set the range for the refinement. After some modification with the point values of the field gamma, the values depending on the field gamma have to be in the range of the refine level, otherwise the cell is defined to be not refined. The entry nBufferLayers stands for the number of buffer layers that will be extended. Maximal refinement means the number of refinements that are allowed for one cell. Also the fluxes that need to be corrected after the refinement are defined.

1.4 Mesh refinement in the *dieselFoam* solver: *dieselDyMFoam*

1.4.1 General set-up to get a new solver

First step is to copy the *dieselFoam* in the user directory and modify the solver name and due to this the file *Make/files* and the name of the .C file. Also be sure you save the new solver in the user library.

1.4.2 dieselDyMFoam.C

To achieve a mesh refinement in *dieselFoam* a more detailed look on the solver has to be done. Where and how can the function *mesh.update()* be added and are there any modifications needed in the solver? The *dieselEngineFoam* is very similar to the *dieselFoam* solver, just a moving mesh is applied. So actually the mesh is changed during the simulation. The same should happen in the *dieselFoam* solver, or actually in the new solver called *dieselDyMFoam*, but cells will be refined and not moved. In the *dieselEngineFoam* solver the changes to the mesh happen before the spray is evolved (*dieselspray.evolve()*), hence the *mesh.update()* will be placed before the function *dieselSpray.evolve()*. We also need to make the fluxes absolute as it is done in the *interDyMFoam* solver, so that they can be divided due to the cell refinement. Also the file *createMesh.H* needs to be interchanged with *createDynamicMesh.H* and *dynamicFvMesh.H* has to be included as well. The new file *dieselDyMFoam.C* in the solver *dieselDyMFoam* reads:

```
#include "fvCFD.H"
#include "dynamicFvMesh.H"
#include "hCombustionThermo.H"
#include "turbulenceModel.H"
#include "spray.H"
#include "psiChemistryModel.H"
#include "chemistrySolver.H"
#include "multivariateScheme.H"
#include "IFstream.H"
#include "OFstream.H"
#include "Switch.H"

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

int main(int argc, char *argv[])
{
    #include "setRootCase.H"
    #include "createTime.H"
    #include "createDynamicFvMesh.H"$
    #include "createFields.H"
    #include "readGravitationalAcceleration.H"
    #include "readCombustionProperties.H"
    #include "createSpray.H"
    #include "initContinuityErrs.H"
    #include "readTimeControls.H"
    #include "compressibleCourantNo.H"
    #include "setInitialDeltaT.H"

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

    Info<< "\nStarting time loop\n" << endl;

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


//*********************************************************
//**********************************************************
// Make the fluxes absolute
fvc::makeAbsolute(phi, U);
//**********************************************************
//*********************************************************


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

      Info<< "Evolving Spray" << endl;



//*********************************************************
//*********************************************************
//start mesh refinement

scalar timeBeforeMeshUpdate = runTime.elapsedCpuTime();

// Do any mesh changes
mesh.update();

if (mesh.changing())
{
Info<< "Execution time for mesh.update() = "
<< runTime.elapsedCpuTime() - timeBeforeMeshUpdate
<< " s" << endl;
}

//end mesh refinement
//*********************************************************
//*********************************************************



        dieselSpray.evolve();
        Info<< "Solving chemistry" << endl;

        chemistry.solve
        (
            runTime.value() - runTime.deltaT().value(),
            runTime.deltaT().value()
        );

        // turbulent time scale
        {
            volScalarField tk =
                Cmix*sqrt(turbulence->muEff()/rho/turbulence->epsilon());
            volScalarField tc = chemistry.tc();

            // Chalmers PaSR model
            kappa = (runTime.deltaT() + tc)/(runTime.deltaT()+tc+tk);
        }

        chemistrySh = kappa*chemistry.Sh()();

        #include "rhoEqn.H"
        #include "UEqn.H"

        for (label ocorr=1; ocorr <= nOuterCorr; ocorr++)
        {
            #include "YEqn.H"
            #include "hsEqn.H"

            // --- PISO loop
            for (int corr=1; corr<=nCorr; corr++)
            {
                #include "pEqn.H"
            }
        }

        turbulence->correct();

        #include "spraySummary.H"

        rho = thermo.rho();

        if (runTime.write())
        {
            chemistry.dQ()().write();
        }

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

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

```
    return 0;
}
```

1.4.3 createFields.H

Also the *createFields.H* file needs to be modified. The original file can be found in the *dieselEngineFoam* solver. This file must be copied to the new solver and a new field called Ytf needs to be created. Ytf stands for the volume fraction of evaporated fuel. This volume fraction is used to define the cells where the mesh should be refined. The definition of *pcorrTypes* is taken from the *interDyMFoam* solver and somehow defines a pressure correction at the boundaries.

1.4.4 YEqn.H

The file *YEqn.H* needs to be copied from the dieselEngineFoam solver and modified. In the YEqn.H file the transport equation for the species is defined. We need to add the transport equation for the evaporated fuel.

1.4.5 Make/options

Finally also the file *Make/options* need to be modified. The second line needs to be changed to have the right path to the dieselEngineFoam solver directory:

```
 -I$(LIB_SRC)/../applications/solvers/combustion/dieselEngineFoam \
```

Also the new libraries that are needed for the mesh refinement process need to be added. See the new libraries below:

```
EXE_INC = \

    [...]

    -I$(LIB_SRC)/dynamicMesh/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/dynamicFvMesh/lnInclude \

    [...]

EXE_LIBS = \

    [...]

    -lmeshTools \
    -ldynamicFvMesh \
    -ltopoChangerFvMesh \

    [...]
```

Chapter 2

Tutorial, step by step

2.1 Solver *dieselDyMFoam*

A new solver will be defined, based on the dieselFoam solver. So the dieselFoam solver is copied in the user directory and renamed to the new solver name dieselDyMFoam. Also the name of the solver in all the files is changed.

```
sol

cp -r combustion/dieselFoam/ $WM_PROJECT_USER_DIR/applications/solvers/

cd $WM_PROJECT_USER_DIR/applications/solvers

mv dieselFoam dieselDyMFoam

cd dieselDyMFoam

mv dieselFoam.C dieselDyMFoam.C

sed -i s/dieselFoam/dieselDyMFoam/g  dieselDyMFoam.C

sed -i s/dieselFoam/dieselDyMFoam/g Make/files
```

To set the library path right, do following change in Make/files:
change

```
EXE = $(FOAM_APPBIN)/dieselDyMFoam
```

to

```
EXE = $(FOAM_USER_APPBIN)/dieselDyMFoam
```

And also change in Make/options:

```
    -I../dieselEngineFoam \
```

to

```
   -I$(LIB_SRC)/../applications/solvers/combustion/dieselEngineFoam \
```

Try to compile the solver.

```
   wclean
   wmake
```

Now the solver will be modified to achieve the mesh refinement during simulation. First the function *YEqn.H* will be copied from the *dieselEngineFoam* solver:

```
 cp $FOAM_SOLVERS/combustion/dieselEngineFoam/YEqn.H .
```

A new field that defines the source term of the volume fraction of the evaporated volume is needed. Also the transport equation of the evaporated volume fraction *Ytf* is needed. This is done by adding following lines in the end of the *YEqn.H* file:

```
volScalarField YtfSource
(
    IOobject
    (
        "kappa",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    mesh,
    dimensionedScalar("zero",dieselSpray.evaporationSource(0)().dimensions() , 0.0)
);

forAll(composition.Y(), i)
{
    YtfSource += dieselSpray.evaporationSource(i);
}

solve
(
    fvm::ddt(rho, Ytf)
  + mvConvection->fvmDiv(phi, Ytf)
  - fvm::laplacian(turbulence->muEff(), Ytf)
  ==
    YtfSource,
    mesh.solver("Yi")
);
```

Since we need a new field that defines the volume fraction of the evaporated fuel the file *createFields.H*, located in the *dieselEngineFoam* solver, needs to be copied and modified in our new solver by doing:

```
cp $FOAM_SOLVERS/combustion/dieselEngineFoam/createFields.H .
```

Modify the file by adding following lines in the end then:

```
volScalarField Ytf
(
    IOobject
    (
        "Ytf",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);

wordList pcorrTypes
    (
        p.boundaryField().size(),
        zeroGradientFvPatchScalarField::typeName
    );

    for (label i=0; i<p.boundaryField().size(); i++)
    {
        if (p.boundaryField()[i].fixesValue())
        {
            pcorrTypes[i] = fixedValueFvPatchScalarField::typeName;
        }
    }


    label pRefCell = 0;
    scalar pRefValue = 0.0;
    setRefCell(p, mesh.solutionDict().subDict("PISO"), pRefCell, pRefValue);
```

Finally the .C file of our new solver needs to be modified. We need to do several modifications that are shown below:
1. change the line

```
    #include "createMesh.H"
```

to

```
    #include "createDynamicFvMesh.H"
```

2. add in the beginning of the file following line:

```
#include "dynamicFvMesh.H"
```

3. before *runTime++;* following lines need to be added:

```
 //*************************************
// Make the fluxes absolute
fvc::makeAbsolute(phi, U);
 //*************************************
```

4. before the line *dieselSpray.evolve();* the function doing the mesh refinement has to be called by adding:

```
 //********************************************************
 //start mesh refinement
 //********************************************************

scalar timeBeforeMeshUpdate = runTime.elapsedCpuTime();

// Do any mesh changes
mesh.update();


if (mesh.changing())
{
Info<< "Execution time for mesh.update() = "
<< runTime.elapsedCpuTime() - timeBeforeMeshUpdate
<< " s" << endl;
}
//********************************************************
//end mesh refinement
//********************************************************
```

The last step is to change the *Make/options*. Following lines need to be added:
1. add in EXE INC following three lines:

```
 -I$(LIB_SRC)/dynamicMesh/lnInclude \
 -I$(LIB_SRC)/meshTools/lnInclude \
 -I$(LIB_SRC)/dynamicFvMesh/lnInclude \
```

2. add in EXE LIBS following lines:

```
-lmeshTools \
-ldynamicFvMesh \
-ltopoChangerFvMesh \
```

Finally the new solver can be compiled by doing:

```
wclean
wmake
```

2.2 Setting up the case

We also need to set up a case where the new solver can be applied. First we enter our *run* directory by typing:

```
run
```

The following lines copy the aachenBomb case from the tutorials and add the *dynamicMeshDict* in the constant folder of the case.

```
cp -r $FOAM_TUTORIALS/combustion/dieselFoam/aachenBomb/ .
cd aachenBomb
cp $FOAM_TUTORIALS/multiphase/interDyMFoam/ras/damBreakWithObstacle/constant/ \
dynamicMeshDict constant/
```

Now we need to apply our field that we want to do the refinement on in the *dynamicMeshDict*. In constant/dynamicMeshDict change

```
alpha1
```

to

```
Ytf
```

We also need to add the field Ytf in the 0 directory. This can easily be done by doing:

```
cp  0/Ydefault 0/Ytf
```

and also change *Tdefault* to *Ytf* in the file *Ytf*.
In system/fvSolution following lines need to be added in the PISO loop :

```
 pRefPoint      (0 0 0);
 pRefValue      0;
```

So the PISO loop in the end of the file will read:

```
PISO
{
    nCorrectors     2;
    nNonOrthogonalCorrectors 0;
    pRefPoint      (0 0 0);
    pRefValue      0;
}
```

Now all changes that are necessary to get the case running are done. Last step is to run *blockMesh*, after that the case can be started with the command *dieselDyM-Foam*. But spray simulations are very sensitive to different set-ups and also the spray model implemented in OpenFOAM is not always very robust. A crash during the simulation can happen and the set-ups need to be modified to avoid it. Suggestions are to modify the *blockMeshDict*, so that the initial cell size is good to use cell refinement. Also the Courant number, the time step and the *unrefineLevel* in the *dynamicMeshDict* are a good start to modify the case. These modifications are connected to spray modeling in general and will not be explained in more detail.

Chapter 3

Results

3.1 Mesh

Figure 3.1-3.3 show the mesh at several time steps after starting the injection of the fuel. As can be seen the grid refines in a certain region during the simulation. Figure 3.4-3.6 show the mesh at different times after start of injection, colored by the field Ytf. This field also traces back to the shape of the spray. As can be seen the refinement happens in the cells where the field occurs.

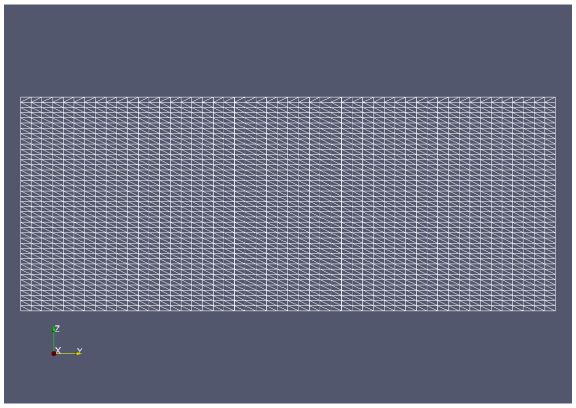

Figure 3.1: Mesh t=0 ms

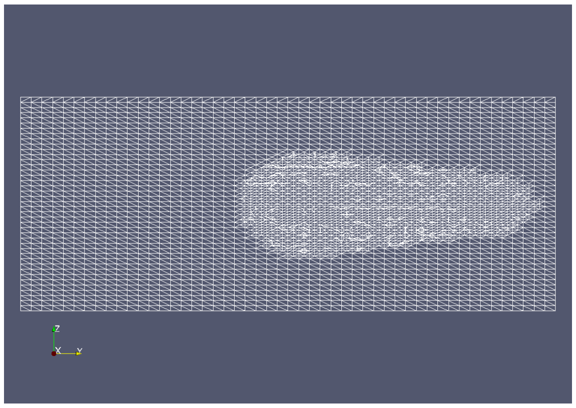

Figure 3.2: Mesh t=1 ms

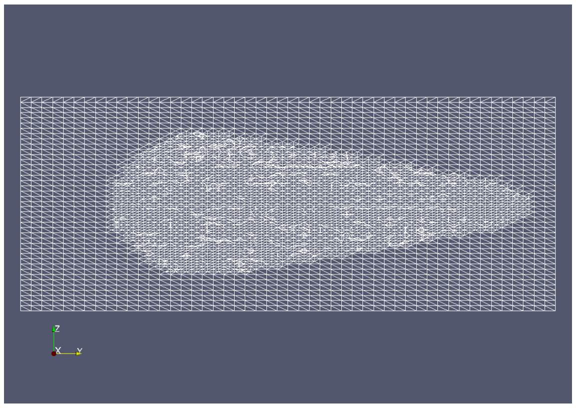

Figure 3.3: Mesh t=2 ms

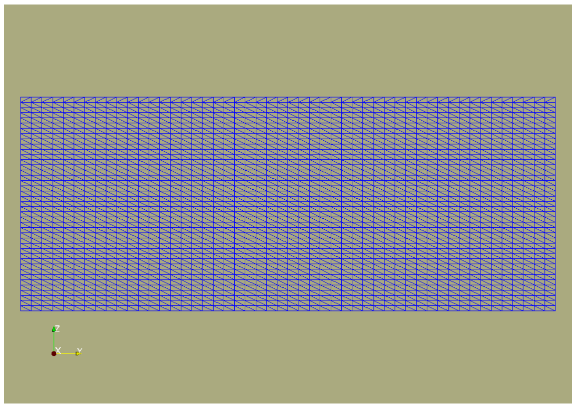

Figure 3.4: Mesh t=0 ms

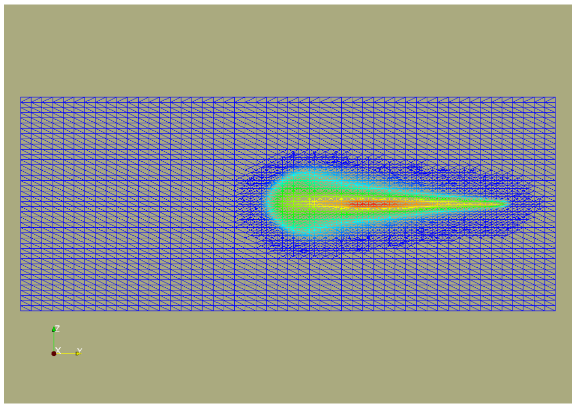

Figure 3.5: Mesh t=1 ms

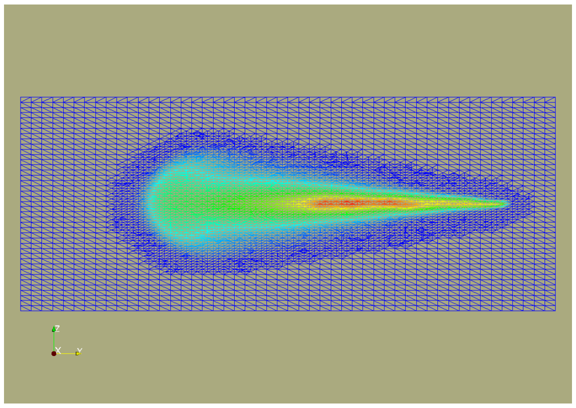

Figure 3.6: Mesh t=2 ms

3.2 Vapor and liquid penetration

Figure 3.7 shows the results of vapor and liquid penetration using different grids (0.5x0.5x1.0mm, 1.0x1.0x2.0mm and a grid starting with 1.0x1.0x2.0mm and refined to 0.5x0.5x1.0mm), while vapor penetration is given by the continiously increasing curve. With the dynamic mesh refinement the liquid penetration is calculated the same as with the finer grid, but the computation time is lower. The vapor penetration differs for all three grids. The vapor penetration is more sensitive due to several set-ups of the several spray submodels, especially the turbulence models and their constant.

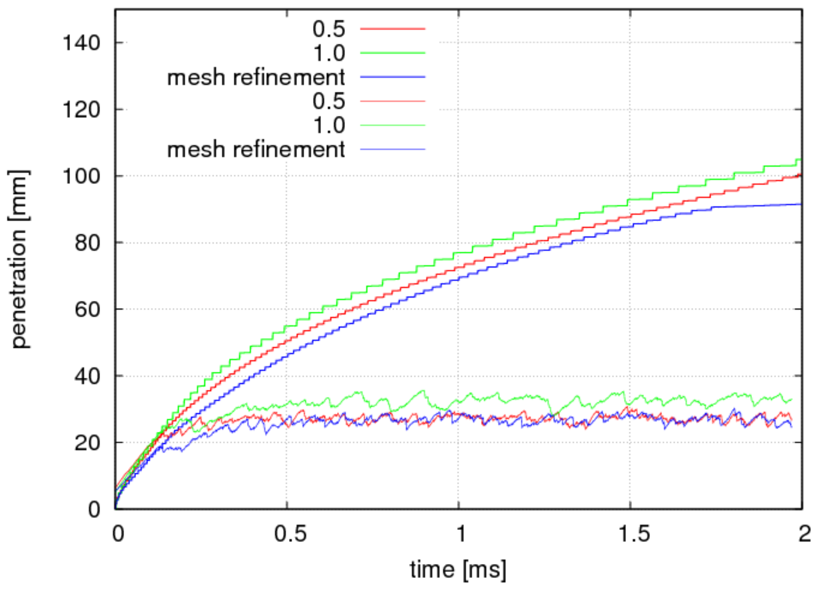

Figure 3.7: Vapor and liquid penetration

3.3 Sensitivity studies

3.3.1 Initial cell size

Since the results of simulations of sprays are grid size dependent, the final cell size after the mesh refinement should be constant. The advantage of dynamic mesh refinement is that the initial cell size can be changed. Figure 3.8 shows the results for vapor and liquid penetration using different initial cell sizes by keeping the final cell size after mesh refinement constant with around 0.5x0.5x1.0 mm. The results of vapor and liquid penetration are not depending between the grids that were tested here.
Figure 3.9 and 3.10 show the grid of two different simulations at t=1 ms. The first figure (Fig. 3.9) shows the grid with an initial cell size of around 1.0x1.0x2.0mm and a minimum cell size in the refined region of around 0.5x0.5x1.0mm. Figure 3.10 shows the grid with an initial cell size of around 4.0x4.0x8.0mm and a minimum cell size in the refined region of around 0.5x0.5x1.0mm. As can be seen the cell number of the mesh with an initial cell size of 4.0x4.0x8.0mm is less in the region around the spray, but at it was shown in figure 3.8 the results are the same.

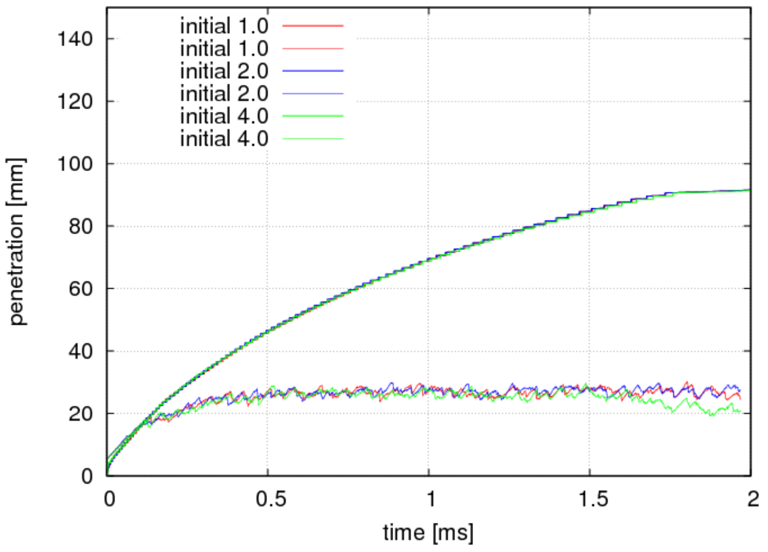

Figure 3.8: Vapor and liquid penetration, different initial grid sizes

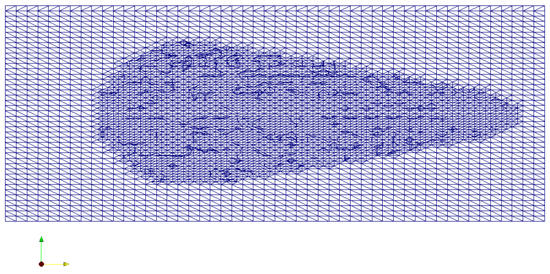

Figure 3.9: Initial cell size = 1.0x1.0x2.0 mm, t=1 ms

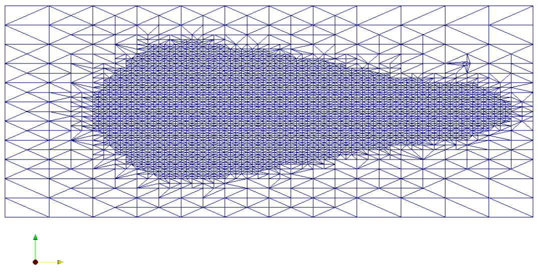

Figure 3.10: Initial cell size = 4.0x4.0x8.0 mm, t=1 ms

3.3.2 Refinement range of Ytf

Figure 3.11 shows the results of vapor and liquid penetration using different lower limits for the refinement range of Ytf. The results are independent up to an lower limit of Ytf=0.001. If the limit is further increased (Ytf=0.01) the results of the liquid penetration start to differ slightly. The vapor penetration is less sensitive on the lower refinement range limit of Ytf.

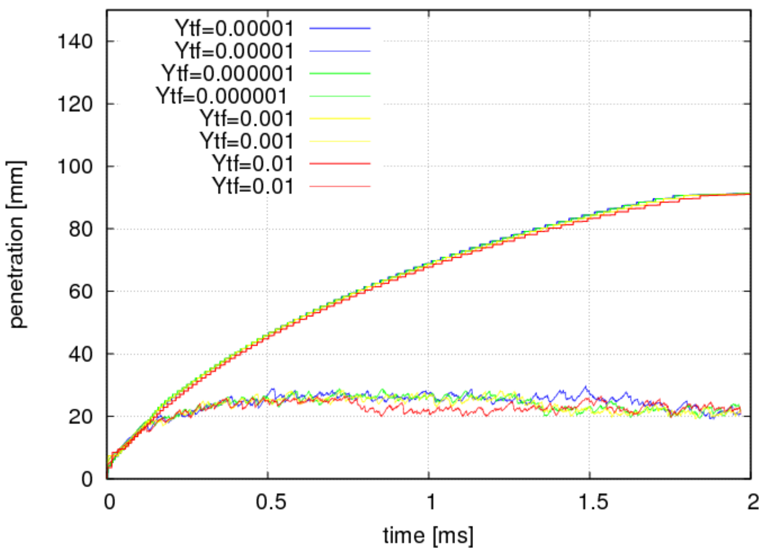

Figure 3.11: Vapor and liquid penetration, different refinement range of Ytf

3.3.3 CPU time

The CPU time for the different grids are shown in table 3.1.

Table 3.1: Comparison different CPU times

mesh	CPU time [s]
0.5	25583
initial 1.0	14135
initial 2.0	8588
initial 4.0	8468

As can be seen the CPU time is decreasing more than 65% (for initial cell size of around 4.0x4.0x8.0mm) using dynamic mesh refinement. To increase the initial cell size from 2.0x2.0x4.0mm to 4.0x4.0x8.0mm does not influence the CPU time very much anymore, so an initial cell size of 2.0x2.0x4.0mm could be a good choice for further simulations.

Appendix A

A.1 *dynamicRefineFvMesh.C.save*

```
bool dynamicRefineFvMesh::update()
{
    // Re-read dictionary. Choosen since usually -small so trivial amount
    // of time compared to actual refinement. Also very useful to be able
    // to modify on-the-fly.
    dictionary refineDict
    (
        IOdictionary
        (
            IOobject
            (
                "dynamicMeshDict",
                time().constant(),
                *this,
                IOobject::MUST_READ,
                IOobject::NO_WRITE,
                false
            )
        ).subDict(typeName + "Coeffs")
    );

    label refineInterval = readLabel(refineDict.lookup("refineInterval"));

    if (refineInterval == 0)
    {
        return false;
    }
    else if (refineInterval < 0)
    {
        FatalErrorIn("dynamicRefineFvMesh::update()")
            << "Illegal refineInterval " << refineInterval << nl
            << "The refineInterval setting in the dynamicMeshDict should"
            << " be >= 1." << nl
            << exit(FatalError);
    }



    bool hasChanged = false;
```

```
// Note: cannot refine at time 0 since no V0 present since mesh not
//       moved yet.

if (time().timeIndex() > 0 && time().timeIndex() % refineInterval == 0)
{
    label maxCells = readLabel(refineDict.lookup("maxCells"));

    if (maxCells <= 0)
    {
        FatalErrorIn("dynamicRefineFvMesh::update()")
            << "Illegal maximum number of cells " << maxCells << nl
            << "The maxCells setting in the dynamicMeshDict should"
            << " be > 0." << nl
            << exit(FatalError);
    }

    label maxRefinement = readLabel(refineDict.lookup("maxRefinement"));

    if (maxRefinement <= 0)
    {
        FatalErrorIn("dynamicRefineFvMesh::update()")
            << "Illegal maximum refinement level " << maxRefinement << nl
            << "The maxCells setting in the dynamicMeshDict should"
            << " be > 0." << nl
            << exit(FatalError);
    }

    const volScalarField& gamma = lookupObject<volScalarField>
    (
        refineDict.lookup("field")
    );

    const scalar minLevel = readScalar(refineDict.lookup("minLevel"));
    const scalar maxLevel = readScalar(refineDict.lookup("maxLevel"));
    const label nBufferLayers =
        readLabel(refineDict.lookup("nBufferLayers"));

    // Points marked for refinement
    PackedList<1> refinePoint(nPoints(), 0);

    {
        // Do naive interpolation to get point values
        // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

        scalarField pointGamma(nPoints(), 0.0);
        labelList nPointGamma(nPoints(), 0);

        forAll(pointCells(), pointI)
        {
            const labelList& pCells = pointCells()[pointI];

            forAll(pCells, i)
```

```
            {
                pointGamma[pointI] += gamma[pCells[i]];
                nPointGamma[pointI]++;
            }
        }

        syncTools::syncPointList
        (
            *this,
            pointGamma,
            plusEqOp<scalar>(), // combine op
            0.0,                // null value
            false               // no separation
        );
        syncTools::syncPointList
        (
            *this,
            nPointGamma,
            plusEqOp<label>(),  // combine op
            0,                  // null value
            false               // no separation
        );

        forAll(pointGamma, pointI)
        {
            pointGamma[pointI] /= nPointGamma[pointI];
        }

        // Mark all points with gamma witin refine range
        markRefinePoints
        (
            pointGamma,
            minLevel,
            maxLevel,
            refinePoint
        );
    }



    {
        labelList cellsToRefine
        (
            selectRefineCells
            (
                maxCells,
                maxRefinement,
                refinePoint
            )
        );

        label nCellsToRefine = returnReduce
        (
            cellsToRefine.size(), sumOp<label>()
        );

        if (nCellsToRefine > 0)
        {
            // Refine/update mesh and map fields
            autoPtr<mapPolyMesh> map = refine(cellsToRefine);

            // Update the interpolated field such that newly created points
            // don't get unrefined.
            const labelList& pointMap = map().pointMap();
            const labelList& reversePointMap = map().reversePointMap();

            // Map refinePoint. Set new points to a refine level
            {
                PackedList<1> newRefinePoint(pointMap.size());

                forAll(pointMap, pointI)
                {
                    label oldPointI = pointMap[pointI];

                    if (oldPointI < 0)
                    {
                        newRefinePoint.set(pointI, 1);
                    }
                    else if (reversePointMap[oldPointI] != pointI)
                    {
                        newRefinePoint.set(pointI, 1);
                    }
                    else
                    {
                        newRefinePoint.set(pointI, refinePoint.get(pointI));
                    }
                }
                refinePoint.transfer(newRefinePoint);
            }


            hasChanged = true;
        }
    }

    // Extend with a buffer layer to prevent neighbouring points being
    // unrefined.
    for (label i = 0; i < nBufferLayers; i++)
    {
        extendMarkedPoints(refinePoint);
    }

    {
        // Select unrefineable points that are not marked in refinePoint
        labelList pointsToUnrefine
        (
            selectUnrefinePoints
            (
                    refinePoint
                )
            );

            label nSplitPoints = returnReduce
            (
                pointsToUnrefine.size(),
                sumOp<label>()
            );

            if (nSplitPoints > 0)
            {
                // Refine/update mesh
                unrefine(pointsToUnrefine);

                hasChanged = true;
            }
        }


        if ((nRefinementIterations_ % 10) == 0)
        {
            // Compact refinement history occassionally (how often?).
            // Unrefinement causes holes in the refinementHistory.
            const_cast<refinementHistory&>(meshCutter().history()).compact();
        }
        nRefinementIterations_++;
    }

    mesh.changing(hasChanged);

    return hasChanged;
}
```