CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Project work:

Description of reactingParcelFilmFoam

Developed for OpenFOAM-2.2.x

Author:
Emil Ljungskog

Peer reviewed by:
Reza Gooya
Olivier Petit

January 24, 2014

Contents

1 Introduction

Multiphase flows with sprays and surface films are ubiquitous with applications in many parts of the industry, for example rain on a car window or spray painting. Such flows can be simulated in OpenFOAM using `reactingParcelFilmFoam`, which according to the source code is a

> "Transient PIMPLE solver for compressible, laminar or turbulent flow with reacting Lagrangian parcels, and surface film modeling."

Furthermore, the solver has the ability to handle chemical reactions and combustion; features that will not be discussed here.

This text will treat `reactingParcelFilmFoam`, starting with a brief review of some of the theory of multiphase flows. Following the theory is a description of the implementation of the solver, including the classes handling the parcels and the surface film, after which a tutorial on how to run a case is presented. The text ends with a description on how to modify the `LiquidEvaporationBoil` boiling model.

2 Theory

This section will give a brief introduction to the theory of Lagrangian particle tracking, and mass and heat exchange. The descriptions will be far from exhaustive, why the interested reader is urged to seek knowledge in these matters elsewhere, for example [1] and [2].

2.1 Lagrangian particle tracking

Simulations of two-phase flows with a continuous and a dispersed phase are often carried out using Lagrangian particle tracking, in which the continuous phase is treated in a Eulerian frame of reference while the particles in the dispersed phase are tracked individually in a Lagrangian ditto. The influence from the dispersed phase on the continuous ditto is then introduced as source terms in the flow equations as [1]

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho u_i}{\partial x_i} = S_C \tag{2.1}$$

$$\frac{\partial \rho u_i}{\partial t} + \frac{\partial \rho u_i u_j}{\partial x_j} = -\frac{\partial p}{\partial x_i} - \frac{\partial \tau_{ij}}{\partial x_j} + S_{i,p} \tag{2.2}$$

where S_C is a source term describing the mass transfer between the phases, while $S_{i,p}$ describes the momentum exchange. Note that the particles in the dispersed phase are treated as point sources in this formulation, which is why the mass fraction is omitted. The equations for the continuous phase are solved in the same way as for single phase flow, while the particle trajectories are integrated from the force balance

$$m_p \frac{du_i}{dt} = F_i \tag{2.3}$$

where m_p denotes the particle mass, u_i its velocity and F_i the forces acting on it. For homogeneous spherical particles, we have $m_p = \rho_p \frac{d_p^3 \pi}{6}$, where ρ_p is the particle density and d_p the diameter of the particle.
Depending on the flow conditions and the desired solution accuracy, different forces can be included. Examples of such forces are drag, buoyancy and pressure force.

The coupling between the phases can be treated differently depending on the nature of the flow. For example, this project will cover two-way coupling, in which it is assumed that both phases affect each other, but no interactions (collisions, etc.) between the particles in the dispersed phase are taken into account.

2.2 Mass and heat exchange

The dispersed phase might exchange not only momentum, but also mass and heat with the carrying phase. Such phenomena will be briefly discussed here.

2.2.1 Boiling

Vaporization of droplets due to boiling can be divided into two phenomena; flash vaporization and vaporization due to heat transfer. The former is a rapid process that occurs for liquids at super heated conditions, such as pre-heated gasoline injected into a direct injected gasoline engine. The latter is a process driven by the heat transfer from the surrounding media to the droplet. The model used in the solver discussed here is developed for the fuel vaporization process in GDI engines, and can be found in [4], to which the interested reader is referred for details.

2.2.2 Species transport

As the two phases exchanges mass, the composition of the carrier phase will change, which is described by the species transport equation [3]

$$\frac{\partial \rho Y_k}{\partial t} + \frac{\partial \rho u_i Y_k}{\partial x_i} = \frac{\partial}{\partial x_i}\left(\rho D_k \frac{\partial Y_k}{\partial x_i}\right) + \dot{\omega}_i. \tag{2.4}$$

Here Y_k is the mass fraction of species k, ρ the bulk density, D_k the species diffusion coefficient and $\dot{\omega}_i$ a source term describing the generation or destruction of species.

2.2.3 Energy transport

As for the mass transfer, the energy exchange between phases will affect the carrier phase. This phenomenon is governed by the energy equation [3]

$$\frac{\partial \rho h}{\partial t} + \frac{\partial \rho u_i h}{\partial x_i} = \frac{\partial}{\partial x_i}\left(\frac{\mu}{\sigma_h}\frac{\partial h}{\partial x_i} + \mu\left(\frac{1}{Sc_k} - \frac{1}{\sigma_h}\right)\sum_{k=1}^{N} h_k \frac{\partial Y_k}{\partial x_i}\right) + \frac{\partial p}{\partial t} + S_{rad} \tag{2.5}$$

where h is the enthalpy, μ the dynamic viscosity, $\sigma_h = \frac{c_p \mu}{k}$ the Prandtl number, $Sc_k = \frac{\mu}{\rho D_k}$ the Schmidt number, and S_{rad} a source term for the radiative heat transfer.

3 Implementation of `reactingParcelFilmFoam`

As previously mentioned, `reactingParcelFilmFoam` is a transient solver for compressible flows with reacting Lagrangian particles and surface films. It uses the PIMPLE algorithm, which is a blend of the SIMPLE and PISO algorithms, to solve the pressure-velocity coupling of the fluid phase.
If we examine the source code, located in `$WM_PROJECT_DIR/applications/solvers/lagrangian/reactingParcelFilmFoam/reactingParcelFilmFoam.C`, we find that the continuity equation (2.1) (note that it is named `rhoEqn` in the code) is solved once outside the PIMPLE loop, while the equations for momentum (2.2), species (2.4) and energy (2.5) is solved inside the main loop. The PIMPLE pressure correction equation (see eg. [3]) is then solved in a pressure corrector loop inside the main loop together with the continuity equation.

Before the PIMPLE loop, the parcels and the film are updated, which is done by calling their respective `evolve()` functions. It is defined in the `regionModel` class that is inherited by all region models, to which the Lagrangian tracking and surface film belong. The source code for this class can be found in `$WM_PROJECT_DIR/src/regionModels/regionModel/regionModel/regionModel.C`.
As can be seen in listing 1, the actual updating is done by the three functions `preEvolveRegion()`, `evolveRegion()`, and `postEvolveRegion()`, that are defined in the classes for the different region models.

```
void Foam::regionModels::regionModel::evolve()
{
    if (active_)
    {
        Info<< "\nEvolving " << modelName_ << " for region "
            << regionMesh().name() << endl;

        //read();
        preEvolveRegion();
        evolveRegion();
        postEvolveRegion();

        // Provide some feedback
        if (infoOutput_)
        {
            Info<< incrIndent;
            info();
            Info<< endl << decrIndent;
        }
    }
}
```

Listing 1: The `evolve()` function of the `regionModel` class.

The solver can also handle combustion and chemical reactions; features that will not be discussed here. However, if the user wish to neglect these phenomena, they can be defined as inactive in their respective control dictionaries (`combustionProperties` and `chemistryProperties`, both located in the `constant` directory of the case). If so, they are completely switched off, so that no computational power is wasted.

3.1 The `basicReactingCloud` class

The cloud of particles is handled by the `basicReactingCloud` class, which is described in the source code as a

> "Cloud class to introduce reacting parcels"

It has support for various physical phenomena, such as heat transfer, radiation, phase change, chemical reactions and modeling of turbulent dispersion. Which of these that shall be included in the simulation is specified by the user in `reactingCloud1Properties`, located in the case's `constant` directory. An excerpt of the alternatives can be seen in table 1 on page 9.

The class definition is found in `$WM_PROJECT_DIR/src/lagrangian/intermediate/clouds/derived/basicReactingCloud/basicReactingCloud.H`, which contains the single type definition found in listing 2. As can be seen, a `basicReactingCloud` object

```
typedef ReactingCloud
<
    ThermoCloud
    <
        KinematicCloud
        <
            Cloud
            <
                basicReactingParcel
            >
        >
    >
> basicReactingCloud;
```

Listing 2: Definition of `basicReactingCloud`.

is built layer-by-layer from templated classes, with a `Cloud` of `basicReactingParcel` as a foundation, where `basicReactingParcel` is an extension of the `particle` class to group particles into parcels.

3.1.1 Evolution of the cloud

The `ThermoCloud` class, defined in `$WM_PROJECT_DIR/src/lagrangian/intermediate/clouds/Templates/ThermoCloud/ThermoCloud.H`, defines its own `evolve()` function, which consists of a call to `solve()` in `KinematicCloud`, found in listing 3 on the next page. The source code for `KinematicCloud` can be found in `$WM_PROJECT_DIR/src/lagrangian/intermediate/clouds/Templates/KinematicCloud/KinematicCloud.H`.

It can be seen that steady-state solutions are treated slightly different from transient ones, due to an extra save of the previous state of the cloud and different treatment of the source terms in the case of coupling. The main work is done by `preEvolve()`, which caches fields and updates the volume fraction, and `evolveCloud()`, which is responsible for the actual evolution of the cloud. After this, some information is printed by the `info()` function, `postEvolve()` is called to do some minor cleanup and advance the solution to the next iteration, and the state is restored for the steady-state case.

Examining the code for `evolveCloud()` in listing 4 on page 8, we find that it starts by resetting the source terms in case of a coupled simulation. After that, a discrimination is once again made between transient and steady-state solutions. In the case of a transient simulation, the size of the cloud is determined, after which particles are injected from the surface film. If this changes the size of the cloud, the volume fraction is recomputed and the new cloud size stored. The main injection is then made at the injectors, after which the movement of the cloud is computed using the `motion()` function, which consists of a call to `move()` and an update of the volume fractions.
For the case of a steady-state simulation, film injection is disregarded, while injections from the injectors is kept. The cloud is then moved by the aforementioned `move()` function.

The `move()` function called here resides in the abstract `Cloud` class, which is defined in `$WM_PROJECT_DIR/src/lagrangian/basic/Cloud/Cloud.H`. It is a rather low-level function that handles the motion of parcels between processors in the case of a parallel simulation. To do the physical movement of the parcels, it calls another `move()` in the `KinematicParcel` class, in which the actual integration of the force balance is carried out. The `KinematicParcel` source code resides in `$WM_PROJECT_DIR/src/lagrangian/intermediate/parcels/Templates/KinematicParcel/KinematicParcel.H`.

```
template<class CloudType>
template<class TrackData>
void Foam::KinematicCloud<CloudType>::solve(TrackData& td)
{
    if (solution_.steadyState())
    {
        td.cloud().storeState();

        td.cloud().preEvolve();

        evolveCloud(td);

        if (solution_.coupled())
        {
            td.cloud().relaxSources(td.cloud().cloudCopy());
        }
    }
    else
    {
        td.cloud().preEvolve();

        evolveCloud(td);

        if (solution_.coupled())
        {
            td.cloud().scaleSources();
        }
    }

    td.cloud().info();

    td.cloud().postEvolve();

    if (solution_.steadyState())
    {
        td.cloud().restoreState();
    }
}
```

Listing 3: The `solve()` function of the `KinematicCloud` class.

```
template<class CloudType>
template<class TrackData>
void Foam::KinematicCloud<CloudType>::evolveCloud(TrackData& td)
{
    if (solution_.coupled())
    {
        td.cloud().resetSourceTerms();
    }

    if (solution_.transient())
    {
        label preInjectionSize = this->size();

        this->surfaceFilm().inject(td);

        // Update the cellOccupancy if the size of the cloud has changed
        // during the injection.
        if (preInjectionSize != this->size())
        {
            updateCellOccupancy();

            preInjectionSize = this->size();
        }

        injectors_.inject(td);


        // Assume that motion will update the cellOccupancy as necessary
        // before it is required.
        td.cloud().motion(td);
    }
    else
    {
//        this->surfaceFilm().injectSteadyState(td);

        injectors_.injectSteadyState(td, solution_.trackTime());

        td.part() = TrackData::tpLinearTrack;
        CloudType::move(td,  solution_.trackTime());
    }
}
```

Listing 4: The `evolveCloud()` function of the `KinematicCloud` class.

Table 1: Excerpt of sub-models for `basicReactingCloud`.

Model	Possible choices	Description
particleForces	BrownianMotion	Brownian motion
	SRF	Rotating reference frame force
	SaffmanMeiLiftForce	Saffman lift force, spherical particles
	TomiyamaLift	Tomiyama lift force
	gravity	Gravity
	nonInertialFrame	Non-inertial frame force
	nonSphereDrag	Drag force for non-spherical particles
	paramagnetic	Magnetic force
	pressureGradient	Pressure gradient force
	sphereDrag	Drag force for spherical particles
	virtualMass	Virtual mass force
dispersionModel	none	Neglect turbulent dispersion
	gradientDispersion	Perturb velocity in direction of $-\nabla k$
	stochasticDispersionRAS	Discrete Random Walk model
heatTransferModel	none	No heat transfer
	RanzMarshall	Ranz-Marshall correlation
phaseChangeModel	none	No phase change
	liquidEvaporation	Evaporation only
	liquidEvaporationBoil	Evaporation and boiling

3.2 The `surfaceFilmModel` class

The `surfaceFilmModel` class has been defined in `$WM_PROJECT_DIR/src/regionModels/surfaceFilmModels/surfaceFilmModel/surfaceFilmModel.H`, where it is described as a

> "Base class for surface film models"

It is a virtual class and is thereby not complete, but can be used as a base class for fully implemented classes. In the case of `surfaceFilmModel`, there are two such classes; `kinematicSingleLayer` and `thermoSingleLayer`. The latter is an extension of the first to take thermodynamic effects, such as vaporization, into account and will be discussed below.

Parameters and definitions of sub-models for the `surfaceFilmModel` class has to be provided by the user in the dictionary `constant/surfaceFilmProperties`. An excerpt of some sub-models for `thermoSingleLayer` can be found in table 2.

Table 2: Excerpt of sub-models for `thermoSingleLayer`.

Model	Possible choices	Description
thermoModel	constant	Constant thermodynamic properties
	singleComponent	Computed thermodynamic properties
forces	surfaceShear	Surface shear force
	thermocapillary	Thermocapillary (Marangoni) force
	contactAngle	Film contact angle force
injectionModels	curvatureSeparation	Film separation due to wall curvature
	drippingInjection	Droplet injection due to dripping
phaseChangeModel	none	No phase change
	standardPhaseChange	Evaporation model, includes boiling

3.2.1 The `thermoSingleLayer` class

The `thermoSingleLayer` class is capable of handling surface films consisting of a single component with constant or varying thermodynamic properties. It can also handle phase change, injection of droplets from the film into the cloud, heat transfer with both wall and fluid, radiation, as well as hydrophilic and hydrophobic surfaces. The class definition can be found in `$WM_PROJECT_DIR/src/regionModels/surfaceFilmModels/surfaceFilmModel/surfaceFilmModel.H`.

As previously established, the call to `film.evolve()` in the solver will result in a call to `evolveRegion()`, where the majority of the work is done. The function in question can

be seen in listing 5 on the following page. The function starts by updating the indicator function that gives the film coverage, after which surface velocities and temperatures are retrieved. When updating the surface velocities, a quadratic velocity profile is assumed. The source terms are then updated for the sub models and the continuity equation is solved before the main loop is entered.
Inside the main loop, the pressure sources are updated before the momentum and energy equations are solved. A thickness correction equation is then solved in an inner loop, which makes the procedure resemble the SIMPLE algorithm. After the loop, the `deltaRho_` helper field and temperature are updated before the source terms are reset to prepare for the next time step.

```
void thermoSingleLayer::evolveRegion()
{
    if (debug)
    {
        Info<< "thermoSingleLayer::evolveRegion()" << endl;
    }

    // Update film coverage indicator
    correctAlpha();

    // Update film wall and surface velocities
    updateSurfaceVelocities();

    // Update film wall and surface temperatures
    updateSurfaceTemperatures();

    // Update sub-models to provide updated source contributions
    updateSubmodels();

    // Solve continuity for deltaRho_
    solveContinuity();

    for (int oCorr=1; oCorr<=nOuterCorr_; oCorr++)
    {
        // Explicit pressure source contribution
        tmp<volScalarField> tpu(this->pu());

        // Implicit pressure source coefficient
        tmp<volScalarField> tpp(this->pp());

        // Solve for momentum for U_
        tmp<fvVectorMatrix> UEqn = solveMomentum(tpu(), tpp());

        // Solve energy for hs_ - also updates thermo
        solveEnergy();

        // Film thickness correction loop
        for (int corr=1; corr<=nCorr_; corr++)
        {
            // Solve thickness for delta_
            solveThickness(tpu(), tpp(), UEqn());
        }
    }

    // Update deltaRho_ with new delta_
    deltaRho_ == delta_*rho_;

    // Update temperature using latest hs_
    T_ == T(hs_);

    // Reset source terms for next time integration
    resetPrimaryRegionSourceTerms();
}
```

Listing 5: `evolveRegion()` from `thermoSingleLayer`.

4 Run a case

In order to get to know the solver even better, we will use it on a case where a spray of water is injected into a flow of hot air in a circular diffusor.

4.1 Case description

The geometry of the diffusor can be seen in figure 1, while boundary conditions are found in table 3.
Hot air flows through the diffusor, in which a spray of water droplets is injected into the air stream. The droplets will then evaporate and thereby affect the composition and motion of the fluid.

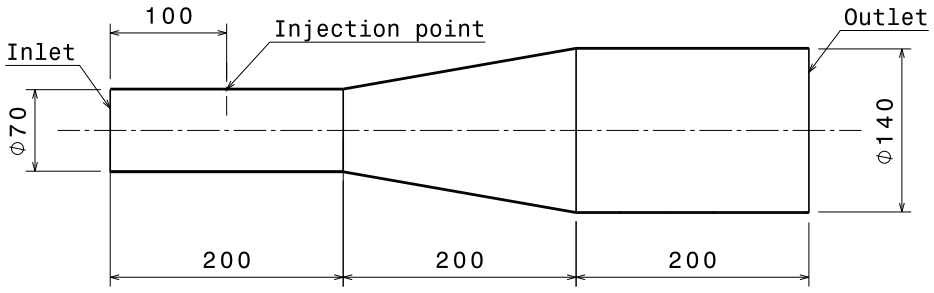

Figure 1: Geometry

Table 3: Boundary conditions.

	Parameter	Value
Inlet	Velocity	30 m/s
	Temperature	700 K
	Turbulent kinetic energy, k	13.5 m^2/s^2
	Specific dissipation rate, ω	734.85 s^{-1}
	H_2O mass fraction	0.05
	O_2 mass fraction	0.1
Outlet	Pressure, p	101 300 Pa
Injection	Mass flow	2.5×10^{-3} kg/s
	Droplet diameter	5×10^{-5} m
	Droplet Temperature	300 K
	Position (x, y, z)	(0.1, 0.034, 0)m

4.2 Solver setup

In order to perform the simulations, we will need to set up the solver properly. To simplify this process, we will use an existing tutorial case as a starting point and complement it with some files from other tutorial cases. We will also have to make quite substantial changes ourselves.

4.2.1 Retrieving files

Initialize the OpenFOAM environment and copy the `cylinder` tutorial to the run directory:

```
OF22x
run
cp -r $FOAM_TUTORIALS/lagrangian/reactingParcelFilmFoam/cylinder evaporatingSpray
cd evaporatingSpray
```

The tutorial we just copied treats a laminar case, why we will need to get some additional files for the turbulence modeling. These are copied from the `pitzDaily` tutorial for the `simpleFoam` solver:

```
cp $FOAM_TUTORIALS/incompressible/simpleFoam/pitzDaily/constant/RASProperties \
        constant/
cp $FOAM_TUTORIALS/incompressible/simpleFoam/pitzDaily/0/{epsilon,k} 0.org/
```

The mesh will be constructed by `blockMesh`, which needs a `blockMeshDict`. This dictionary will be constructed from a `m4`-script, which defines the geometry and meshing parameters. Download `blockMeshDict.m4` from `http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2013/` and put it in `constant/polyMesh`.

Since we will run the simulation in parallel, we need a `decomposeParDict`, which we get from the `interFoam/damBreak` tutorial.

```
cp $FOAM_TUTORIALS/multiphase/interFoam/laminar/damBreak/system/decomposeParDict \
        system/
```

4.2.2 Set boundary conditions

To set the boundary conditions for the species, open `0.org/N2` in an editor and change

```
boundaryField
{
    "(sides|frontAndBack)"
    {
            type         zeroGradient;
    }

    region0_to_wallFilmRegion_wallFilmFaces
    {
            type         zeroGradient;
    }
}
```

to

```
boundaryField
{
    inlet
    {
        type         fixedValue;
        value        uniform 0.79;
    }

    outlet
    {
        type         zeroGradient;
    }

    region0_to_wallFilmRegion_wallFilmFaces
    {
            type         zeroGradient;
    }
}
```

Copy this change to the other species and set the correct values:

```
cd 0.org
cp N2 H2O; cp N2 O2
sed -i s/0.79/0.85/g N2
sed -i s/0.79/0.05/g H2O
sed -i s/0.79/0.1/g O2
sed -i s/N2/O2/g O2
sed -i s/N2/H2O/g H2O
cd ..
```

Velocity

Edit `0.org/U` in a similar way as for the species, so that we get

```
internalField   uniform (30 0 0);

boundaryField
{
    inlet
    {
        type         fixedValue;
        value        uniform (30 0 0);
    }

    outlet
    {
        type         pressureInletOutletVelocity;
        value        uniform (30 0 0);
    }

    region0_to_wallFilmRegion_wallFilmFaces
    {
            type         fixedValue;
        value        uniform (0 0 0);
    }
}
```

Pressure

As for the species and velocity, edit `0.org/p` to get

```
boundaryField
{
    inlet
    {
        type         fixedFluxPressure;
    }

    outlet
    {
        type         fixedValue;
        value        $internalField;
    }

    region0_to_wallFilmRegion_wallFilmFaces
    {
```

```
        type         zeroGradient;
    }
}
```

Since `reactingParcelFilmFoam` doesn't solve for the actual pressure, but for the pressure minus the hydrostatic pressure, `0.org/p_rgh` has to be edited as well:

```
boundaryField
{
    inlet
    {
        type         calculated;
        value        $internalField;
    }

    outlet
    {
        type         calculated;
        value        $internalField;
    }

    region0_to_wallFilmRegion_wallFilmFaces
    {
        type         calculated;
        value        $internalField;
    }
}
```

Temperature

Finally, set the temperature boundary conditions by editing `0.org/T` to contain the following:

```
internalField   uniform 700;

boundaryField
{
    inlet
    {
        type         fixedValue;
        value        uniform 700;
    }

    outlet
    {
        type         zeroGradient;
```

```
    }

    region0_to_wallFilmRegion_wallFilmFaces
    {
        type         zeroGradient;
    }
}
```

4.2.3 Set up turbulence model

We will use the $k-\omega$ SST model, which means that some of the files we retrieved previously have to be edited. Firstly, make sure that the correct model is used:

```
sed -i s/laminar/RASModel/g constant/turbulenceProperties
sed -i s/kEpsilon/kOmegaSST/g constant/RASProperties
sed -i s/epsilon/omega/g system/fvS*
```

Furthermore, add the following to `system/fvSolution` at line 55:

```
"(k|omega)Final"
{
    $UFinal;
}
```

Since the files we copied from the `pitzDaily` tutorial uses the $k-\varepsilon$ model, we have to change `epsilon` to `omega`, and set proper boundary conditions for our case:

```
cd 0.org/
mv epsilon omega
sed -i /lowerWall/,+4d {k,omega}
sed -i /frontAndBack/,+3d {k,omega}
sed -i s/upperWall/wall/g {k,omega}
sed -i s/'0 2 -3 0 0 0 0'/'0 0 -1 0 0 0 0'/g omega
sed -i s/0.375/13.5/g k
sed -i s/14.855/734.85/g omega
sed -i 's/omegaWallFunction/compressible::&/g' omega
```

The last line makes sure that the compressible wall function for ω is used. This is only necessary for `omega`, since the solver makes the substitution automatically for the other wall functions.

4.2.4 Set spray and film properties

Change `constant/reactingCloud1Properties` according to appendix A. This will set the injection properties in accordance with table 3, include the desired models, and set appropriate values for the model parameters. It will also set the start time of the injection to 0.1 s, so that the flow is developed before the particles are injected.
The properties for the surface film has to be edited as well. To do so, change the `thermoModel` entry from `singleComponent` to `constant` in `constant/surfaceFilmProperties`, and add the following inside the `thermoSingleLayer` block:

```
constantThermoCoeffs
{
    rho0        rho0 [1 -3 0 0 0] 1000;
    mu0         mu0 [1 -1 -1 0 0] 1e-3;
    sigma0      sigma0 [1 0 -2 0 0] 0.07;
    Cp0         Cp0 [0 2 -2 -1 0] 4187;
    kappa0      kappa0 [1 1 -3 -1 0] 0.58;
}
```

4.2.5 Set up domain decomposition

Edit `decomposeParDict` so that the domain is split in three parts along the x-axis, and make a copy for the wall film region:

```
sed -i s/'( 2 2 1 )'/'( 3 1 1 )'/ system/decomposeParDict
cp system/decomposeParDict system/wallFilmRegion
```

4.2.6 Set time step and write interval

The settings for the time step and data output are residing in `system/controlDict`. We will not use a fixed time step, but rather let the solver make a suitable choice based on a limiting Courant number. To do this, set

```
adjustTimeStep  yes;

maxCo           1;
```

Also set `endTime` to a suitable value; 0.13 might be a good choice. Furthermore, choose `writeInterval` so that a decent amount of data is obtained; 0.01 is probably reasonable.

4.3 Run the simulation

When all files are properly set up, start by creating a `0/`-directory, and a mesh from the previously retrieved `m4`-script

```
cp -r 0.org 0
m4 constant/polyMesh/blockMeshDict.m4 > constant/polyMesh/blockMeshDict
blockMesh
```

Construct the proper sets in the mesh, and create the wall film mesh in the `0`-directory:

```
topoSet
extrudeToRegionMesh -overwrite
```

Decompose both the fluid domain and the film region:

```
decomposePar
decomposePar -region wallFilmRegion
```

To run the simulation in parallel using MPI, execute

```
mpirun -np 3 reactingParcelFilmFoam -parallel >& log.reactingParcelFilmFoam
```

The contents of the log file can be followed continuously with

```
tailf log.reactingParcelFilmFoam
```

which will print the last ten lines of the log file to stdout and update as new lines are added to the file.

4.4 Post-processing

4.4.1 Reconstruct the domain

The domain was previously decomposed in order to run the simulation in parallel, which means that the it has to be reconstructed before the data can be visualized. This is done using `reconstructPar` on both the internal mesh and the wall film region as

```
reconstructPar -newTimes
reconstructPar -newTimes -region wallFilmRegion
```

4.4.2 Convert data to VTK format

Since `paraFoam` cannot handle Lagrangian fields in a good way, the data has to be converted to the VTK file format using `foamToVTK` as

```
foamToVTK -noZero
foamToVTK -noZero -region wallFilmRegion
```

The `-noZero` flag has to be used since Paraview seems unable to find the particles if the `0` directory is included.

4.4.3 Visualize data in Paraview

Start Paraview by issuing the command

```
paraview
```

and read the data using the GUI (Ctrl+O). All VTK files resides in the `VTK/` folder; the most interesting ones are probably `evaporatingSpray_..vtk` (fluid data), `lagrangian/reactingCloud1/reactingCloud1_..vtk` (spray data) and `wallFilmRegion/region0_to_wallFilmRegion_wallFilmFaces/region0_to_wallFilmRegion_wallFilmFaces_..vtk` (wall film data).

The spray can be visualized using spherical glyphs of suitable radius, while the wall film is seen by simply color the `region0_to_wallFilmRegion_wallFilmFaces` surface by one of the variables. However, in order for the inside of the domain where the spray resides to be visible, the opacity of the wall film region has to be lowered.

5 Change the boiling model

We will now make a small change in the boiling model used in the `LiquidEvaporationBoil` model. This means that we will have to recompile parts of the code in order for our changes to take effect.

5.1 Copy the necessary files

Copy the intermediate lagrangian library to the working directory and change the name of the evaporation model:

```
cd $WM_PROJECT_DIR
cp -r --parents src/lagrangian/intermediate/ $WM_PROJECT_USER_DIR
cd $WM_PROJECT_USER_DIR/src/lagrangian/intermediate/submodels/Reacting/\
        PhaseChangeModel
mv LiquidEvaporationBoil myLiquidEvaporationBoil
cd myLiquidEvaporationBoil
mv LiquidEvaporationBoil.C myLiquidEvaporationBoil.C
mv LiquidEvaporationBoil.H myLiquidEvaporationBoil.H
sed -i 's/LiquidEvaporationBoil/my&/g' *
cd $WM_PROJECT_USER_DIR/src/lagrangian/intermediate/
```

5.2 Edit build files

In order for the compiler to find our new model, some files has to be edited. Starting with `Make/files`, change the last line to

```
LIB = $(FOAM_USER_LIBBIN)/libmyLagrangianIntermediate
```

Since the compilation procedure of these libraries are slightly involved, a less obvious file has to be edited as well:

```
sed -i 's/LiquidEvaporationBoil/my&/g' \
        parcels/include/makeReactingParcelPhaseChangeModels.H
```

5.3 Edit the code

The article upon which the boiling model is based [4] uses the Nusselt number to calculate the vaporization rate, but the implementation in `LiquidEvaporationBoil.C` uses the Sherwood number divided by two instead. To fix this, add the following function

to `myLiquidEvaporationBoil.C` at line 63, just after the function that computes the Sherwood number:

```
template<class CloudType>
Foam::scalar Foam::LiquidEvaporationBoil<CloudType>::Nu
(
    const scalar Re,
    const scalar Pr
) const
{
    return 2.0 + 0.6*Foam::sqrt(Re)*cbrt(Pr);
}
```

Also add the following lines at line 255, in the `calculate` function:

```
// Prandtl number
const scalar Pr = Cpc*nu*rhos/kappac;

// Nusselt number
const scalar Nu = this->Nu(Re, Pr);
```

Finally, change the computation of the intermediate model variable B to comply with [4]. This is done by changing line 296 to

```
const scalar B = 2*pi*kappac/Cpc*d*Nu;
```

and remove the comment on line 294 that says `// NOTE: using Sherwood number instead of Nusselt number`.

Since we have defined a new function, we must also declare it. This is done by adding

```
//- Nusselt number as a function of Reynolds and Prandtl numbers
scalar Nu(const scalar Re, const scalar Pr) const;
```

to `myLiquidEvaporationBoil.H` at line 84.

5.4 Compile the library

Go to the `intermediate` directory and compile the library:

```
cd $WM_PROJECT_USER_DIR/src/lagrangian/intermediate/
wmake libso
```

5.5 Update case files and run

In order to use the edited model, we must tell the solver to use it and where to find it. To do the former, edit the `phaseChangeModel` in `constant/reactingCloud1Properties`:

```
run
cd evaporatingSpray
sed -i 's/LiquidEvaporationBoil/my&/g' constant/reactingCloud1Properties
```

The latter is done by adding

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

to `system/controlDict`. To proceed and run a simulation, follow the instructions in section 4.

References

[1] C. T. Crowe. *Multiphase flows with droplets and particles*. Boca Raton, FL: CRC Press, 2012. ISBN: 9781439840504.

[2] D. Schroeder. *An introduction to thermal physics*. San Francisco, CA: Addison Wesley, 2000. ISBN: 0-321-27779-1.

[3] H. Versteeg and W. Malalasekera. *An Introduction to Computational Fluid Dynamics: The Finite Volume Method (2nd Edition)*. Prentice Hall, 2007. ISBN: 9780131274983.

[4] B. Zuo, A. Gomes, and C. Rutland. "Studies of Superheated Fuel Spray Structures and Vaporization in GDI Engines". In: *Eleventh International Multidimensional Engine Modeling User's Group Meeting*. 2000.

A reactingCloud1Properties

```
/*---------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.2.0                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      reactingCloud1Properties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

solution
{
    active          true;
    coupled         yes;
    transient       yes;
    cellValueSourceCorrection no;

    sourceTerms
    {
        schemes
        {
            rho             explicit 1;
            U               explicit 1;
            Yi              explicit 1;
            h               explicit 1;
            radiation       explicit 1;
        }
    }

    interpolationSchemes
    {
        rho             cell;
        U               cellPoint;
```

```
        thermo:mu       cell;
        T               cell;
        Cp              cell;
        p               cell;
        kappa           cell;
    }

    integrationSchemes
    {
        U               Euler;
        T               analytical;
    }
}

constantProperties
{
    rho0            1000;
    T0              300;
    Cp0             4187;

    youngsModulus   1e9;
    poissonsRatio   0.35;

    epsilon0        1;
    f0              0.5;
    Pr              0.7;
    Tvap            273;
    Tbp             373;

    constantVolume  false;
}

subModels
{
    particleForces
    {
        sphereDrag;
        gravity;
    }

    injectionModels
    {
        model1
```

```
    {

            type            coneInjection;
    SOI             0.1;
    duration        1;
    positionAxis
    (
        ((0.1 0.034 0) (0 -1 0))
    );

    massTotal       0.0025;
    parcelsPerInjector 1e6;
    //parcelsPerSecond 500;
    parcelBasisType mass;
    flowRateProfile constant 0.1;
    Umag            constant 25.0;
    thetaInner      constant 25;
    thetaOuter      constant 45;

    sizeDistribution
    {
        type         fixedValue;
        fixedValueDistribution
        {
        value        50e-6;
        }
    }

    }
}

dispersionModel stochasticDispersionRAS;

patchInteractionModel standardWallInteraction;

heatTransferModel RanzMarshall;

compositionModel singlePhaseMixture;

phaseChangeModel liquidEvaporationBoil;

surfaceFilmModel thermoSurfaceFilm;
```

```
    radiation       off;

    standardWallInteractionCoeffs
    {
        type            rebound;
    }

    RanzMarshallCoeffs
    {
        BirdCorrection  true;
    }

    singlePhaseMixtureCoeffs
    {
        phases
        (
            liquid
            {
                H2O          1;
            }
        );
    }

    liquidEvaporationBoilCoeffs
    {
        enthalpyTransfer latentHeat;

        activeLiquids    ( H2O );
    }

    thermoSurfaceFilmCoeffs
    {
        interactionType absorb;
    }
}


cloudFunctions
{}


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