

Cite as: Salajeghe, R.: Developing a solver to model the photopolymerization process. In Proceedings of CFD with OpenSource Software, 2022, Edited by Nilsson. H.,  
[http://dx.doi.org/10.17196/OS\\_CFD#YEAR\\_2022](http://dx.doi.org/10.17196/OS_CFD#YEAR_2022)

# CFD WITH OPENSOURCE SOFTWARE

A COURSE AT CHALMERS UNIVERSITY OF TECHNOLOGY  
TAUGHT BY HÅKAN NILSSON

---

## Developing a solver to model the photopolymerization process

---

Developed for OpenFOAM-v2112

*Author:*

Roozbeh SALAJEGHE  
Technical University of Denmark  
roosa@dtu.dk

*Peer reviewed by:*

Rafael BECKER MEIER  
Saeed SALEHI

Licensed under CC-BY-NC-SA, <https://creativecommons.org/licenses/>

Disclaimer: This is a student project work, done as part of a course where OpenFOAM and some other OpenSource software are introduced to the students. Any reader should be aware that it might not be free of errors. Still, it might be useful for someone who would like learn some details similar to the ones presented in the report and in the accompanying files. The material has gone through a review process. The role of the reviewer is to go through the tutorial and make sure that it works, that it is possible to follow, and to some extent correct the writing. The reviewer has no responsibility for the contents.

January 15, 2023

# Learning outcomes

The main requirements of a tutorial in the course is that it should teach the four points: How to use it, The theory of it, How it is implemented, and How to modify it. Therefore the list of learning outcomes is organized with those headers.

The reader will learn:

## **How to use it:**

- how to use the reactingFoam solver.

## **The theory of it:**

- the theory of free radical photopolymerization

## **How it is implemented:**

- presenting the implementation of chemical reactions in the solver

## **How to modify it:**

- how to create a new library from scratch to implement the necessary functions for the photopolymerization model
- how to modify an existing solver and tailor it for the photopolymerization process.
- how to define light intensity field for a single beam

# Prerequisites

The reader is expected to know the following in order to get maximum benefit out of this report:

- How to run standard document tutorials like damBreak tutorial.
- Fundamentals of Computational Methods for Fluid Dynamics, Book by J. H. Ferziger and M. Peric
- How to customize a solver and do top-level application programming.

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Background and motivation . . . . .	4
<b>2</b>	<b>Theory</b>	<b>5</b>
2.1	Describing the physics . . . . .	5
<b>3</b>	<b>Tutorials</b>	<b>9</b>
3.1	Explanation of a tutorial case for reactingFoam . . . . .	9
3.1.1	Geometry and boundary conditions . . . . .	9
3.1.2	System directory . . . . .	9
3.1.3	Constant directory . . . . .	10
3.1.4	0 directory . . . . .	11
3.1.5	Results . . . . .	11
<b>4</b>	<b>Implementation of the original solver and relevant libraries</b>	<b>13</b>
<b>5</b>	<b>reactingCureFoam</b>	<b>19</b>
5.1	photoPolymerization library . . . . .	19
5.1.1	photoCure class . . . . .	19
5.1.2	cureReaction class . . . . .	23
5.2	reactingCureFoam solver . . . . .	27
5.3	Setting up a tutorial case . . . . .	32
<b>A</b>	<b>Developed codes</b>	<b>43</b>
A.1	reactingCureFoam solver . . . . .	43
A.2	Constant dictionaries . . . . .	49

# Chapter 1

## Introduction

### 1.1 Background and motivation

Additive manufacturing is a developing field with lots of potentials. One of the main categories of this promising field is the UV-based additive manufacturing. In this category, UV (or near-UV) light is used to cure the resin. Resin is composed of monomers, oligomers, and a photo-initiator. When it is irradiated by UV light, a chain of reactions will start in the resin that connects the monomers together to form polymers. As a result of this process, the viscosity and density inside the liquid resin will build up, turning it into a gel or a solid.

Different UV-based additive manufacturing methods are used in the industry. In this project, the stereolithography (SLA) method is modeled. SLA is the oldest additive manufacturing method, and different theoretical and numerical models have been developed for it. Here, the framework described in Tang's PhD thesis [1] will be followed. In this PhD thesis the fluid flow is neglected. However, in some cases that the cure-induced volume shrinkage of the material is important to be considered, the fluid motion should be modeled. Additionally, fluid flow plays a pivotal role in other vat polymerization methods, such as volumetric additive manufacturing (VAM). Hence, in this tutorial, the convection terms will be kept in the equations though the velocity would be zero for the sake of comparison with the results of [1].

In OpenFOAM, there is no separate solver that solves the reactions and the relevant species transport equations inside a flowing fluid. All the current solvers capable of solving a reacting system are built on top of the combustion models. So, a user who does not have a background in combustion might find them overwhelming. Additionally, a simple modification to one of the classes that are used by the solver may be formidable due to the complicated inheritance structure that is implemented in combustion models. Here, after discussing the shortcomings of the current `reactingFoamSolver` to model the photopolymerization process, a new library and solver are developed to model this process.

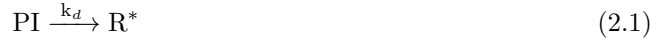
# Chapter 2

## Theory

In this chapter, the theory of the photopolymerization process is explained, and the reactions and their formulations are described.

### 2.1 Describing the physics

When exposed to UV light, photo-initiators decompose to free radicals (Eq. (2.1)). The free radicals react with monomers and turn them into monomeric radicals (Eq. (2.2)). These two chemical reactions are usually categorized within the initiation stage. Then, in a stage called propagation, monomeric radicals increase their chain length by reacting with other monomers (Eq. (2.3)) to form macroradicals until the termination stage, in which two macroradicals react with each other to form an inert product <sup>1</sup> (Eq. (2.4)). In Eq. (2.1)- (2.4), PI, R\*, M, P<sub>n</sub>\*, and M<sub>n+m</sub>, respectively, stand for photo-initiator, free radical, monomer, macroradical of length *n*, and inert polymer of length (*n* + *m*).



UV light, which initiates the photopolymerization process, can be applied to the resin in different ways. In the stereolithography (SLA) process that is investigated here, a laser scans through the surface of the resin according to a horizontal slice of a predefined geometry and cures each point to form a complete layer. A platform, on top of which the layers are made, moves downwards after the formation of the layer to let some resin come to the surface and make it ready for a new layer. Considering a laser scanning a straight line on the surface of the resin, the cured profile would be a parabolic cylinder, shown in Figure 2.1a, provided that the intensity decay in the resin follows the Beer-Lambert law, and a Gaussian beam is applied [2]. When the laser velocity is constant, all the sections that are away from the start and end point of the laser will experience similar conditions, and it suffices to model the photopolymerization in this section as a representative of all sections. Also, since the cure profile is symmetric about the xz plane, it is possible to model only half of the curing section, shown in Figure 2.1b, in which *C<sub>d</sub>* is the cure depth and *w<sub>0</sub>* is the beam radius. Note that since the cure depth is not known in advance, it should be adjusted after running some simulations. For the described section of the geometry, the intensity can be defined as

---

<sup>1</sup>When one inert product is resulted, it is called combination mechanism. Other mechanisms for termination are also possible, but for the current material which is an acrylate, combination is dominant

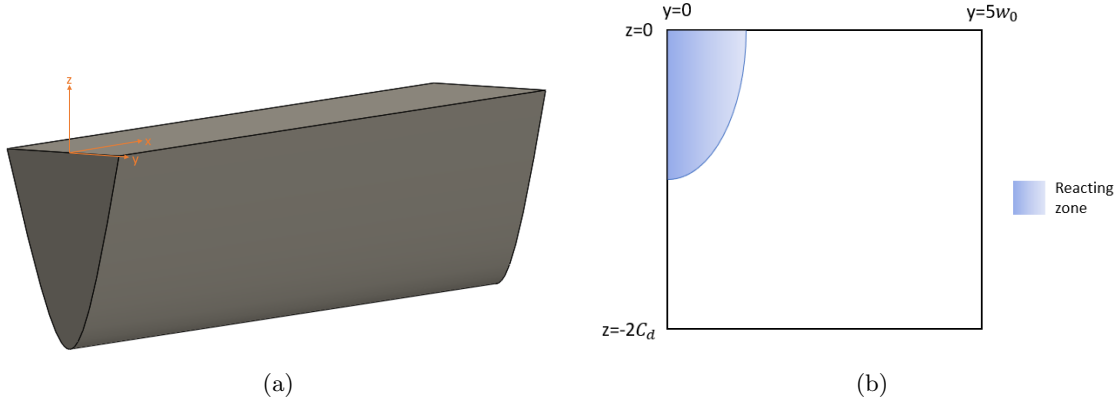


Figure 2.1: (a) cured shape in SLA process, (b) Computational domain

$$I = I_0 \exp \left\{ \frac{-2(V_s(t - t_0))^2 + y^2}{w_0^2} \right\} \exp(-z/D_p) \frac{\lambda(\text{nm})}{1.196 \times 10^8} \quad (2.5)$$

in which  $I_0$  is the maximum incident intensity at the surface of the resin,  $V_s$  is the velocity of the laser,  $w_0$  is the radius of the beam,  $D_p$  is the penetration depth of the light beam in the resin, and  $t$  is time.  $t_0$  is the initial time that can be used for adjusting the x-position of the laser relative to the modeling section.

The rate of the chemical reactions can be described as follows,

$$R_i = 2.3\phi_i\epsilon[\text{PI}]I \quad (2.6)$$

$$R_p = k_p[\text{P}^*][\text{M}] \quad (2.7)$$

$$R_t = k_t[\text{P}^*]^2 \quad (2.8)$$

In which  $R_i$ ,  $R_p$ , and  $R_t$  stand for initiation, propagation, and termination rates, respectively.  $\phi_i$  is the initiation quantum yield,  $\epsilon$  is the absorptivity, and  $k_p$  and  $k_t$  are the rate constants for propagation and termination. The square brackets around a symbol show the concentration of species; for example,  $[\text{PI}]$  stands for the concentration of the photo-initiator.

The kinetic constants of reactions are functions of both free volume and conversion degree. The free volume dependence is described as

$$k_p = \frac{k_{p0}}{1 + \exp \left[ A_p \left( \frac{1}{f} - \frac{1}{f_{cp}} \right) \right]} \quad (2.9)$$

$$k_t = \frac{k_{t0}}{1 + \left\{ R_{rd}k_p[M]/k_{t0} + \exp \left[ A_t \left( \frac{1}{f} - \frac{1}{f_{ct}} \right) \right] \right\}^{-1}} \quad (2.10)$$

and the temperature dependence, in Arrhenius format, is

$$k_{p0} = A_{Ep} e^{\frac{-E_p}{RT}} \quad (2.11)$$

$$k_{t0} = A_{Et} e^{\frac{-E_t}{RT}} \quad (2.12)$$

In the above set of equations,  $A_{Ep}$ ,  $A_{Et}$ ,  $A_t$ , and  $A_p$  are pre-exponential factors,  $E_p$  and  $E_t$  are activation energies for propagation and termination, respectively,  $R$  is the gas constant,  $f$  is the

fractional free volume,  $f_{cp}$  and  $f_{ct}$  are critical fractional free volumes for propagation and termination, respectively, and  $R_{rd}$  is a proportionality constant. The fractional free volume can be expressed as a function of the conversion degree and glass transition temperatures

$$f = f_M \phi_M + f_P (1 - \phi_M) \quad (2.13)$$

$$f_M = 0.025 + \alpha_M (T - T_{gM}) \quad (2.14)$$

$$f_P = 0.025 + \alpha_P (T - T_{gP}) \quad (2.15)$$

$$\phi_M = \frac{1 - X}{1 - X + \frac{\rho_M}{\rho_P} X} \quad (2.16)$$

In the above equations,  $\alpha$ ,  $T_g$ , and  $\rho$  stand for the volumetric coefficient of expansion, glass transition temperature, and density, respectively, for which the subscripts  $M$  and  $P$  denote monomer and polymer properties.  $f_M$  and  $f_P$  represent the fractional free volume of pure monomer and pure polymer, respectively, and the volume fraction of monomer is represented by  $\phi_M$ . Monomer conversion,  $X$ , measures the percent of the monomer molecules that have been converted,

$$X = \frac{[M_0] - [M]}{[M_0]} \quad (2.17)$$

in which,  $[M_0]$  is the initial concentration of the monomer. The critical fractional free volume that is used in equations 2.9 and 2.10 can be evaluated for both propagation and termination as

$$\frac{1}{f_c} = \frac{1}{f_c^{\text{ref}}} + \frac{E}{AR} \left( \frac{1}{T} - \frac{1}{T^{\text{ref}}} \right) \quad (2.18)$$

where  $f_c^{\text{ref}}$  is the reference fractional free volume at the reference temperature  $T^{\text{ref}}$ . The relations for specific heat capacity and density as functions of temperature are

$$C_{P,M} = 5.6 \times T(K) + 218.6 \quad (2.19)$$

$$C_{P,P} = 9.1 \times T(K) - 1535.5 \quad (2.20)$$

$$C_P = C_{P,M}(1 - X) + C_{P,P}X \quad (2.21)$$

$$\rho_P = \frac{1200}{1 + \alpha_P(T - 308)} \quad (2.22)$$

$$\rho_M = \frac{1128}{1 + \alpha_M(T - 298)} \quad (2.23)$$

$$\rho = \rho_M \phi_M + \rho_P (1 - \phi_M) \quad (2.24)$$

in which,  $\rho$  and  $C_p$  are the density and the specific heat capacity, respectively, and the subscripts  $M$  and  $P$  represent the properties of monomer and polymer, respectively. The value of other properties are mentioned in reference [1].

The reactions in a photopolymerization process are exothermic, and the temperature inside the resin increases as a result of the generated heat. It makes it necessary to also solve the energy equation to find the temperature field that will affect the properties of the resin. So, to fully model a photopolymerization process, the continuity, momentum, species transport, and the energy equations should be solved.<sup>2</sup> It should be noted that the laminar version of the momentum equation

<sup>2</sup>In this tutorial, the variation of density with temperature is ignored so as to solve the incompressible Navier-Stokes equations



is used here due to the fact that the viscosity of the resins used in photopolymerization are usually high and the flow is laminar.

$$\nabla \cdot \mathbf{U} = 0 \quad (2.25)$$

$$\frac{\partial(\rho \mathbf{U})}{\partial t} + \nabla \cdot (\rho \mathbf{U} \mathbf{U}) = -\nabla p + \nabla \cdot (\mu \nabla \mathbf{U}) + F \quad (2.26)$$

$$\frac{\partial(\rho S_i)}{\partial t} + \nabla \cdot (\rho \mathbf{U} S_i) = \nabla \cdot (D_s \nabla S_i) + R \quad (2.27)$$

$$\frac{\partial(\rho C_p T)}{\partial t} + \nabla \cdot (\rho \mathbf{U} C_p T) = \nabla \cdot (k \nabla T) + Q_R \quad (2.28)$$

Here,  $\rho$  stands for density,  $\mathbf{U}$  is the velocity vector,  $p$  is the pressure,  $\mu$  is the viscosity of the fluid,  $F$  is the body force exerted on the fluid,  $S_i$  stands for the concentration of each one of species,  $D_s$  is the diffusion coefficient of the relevant specie,  $R$  is the reaction-based production or consumption of the specie,  $h$  is the enthalpy, and  $Q_R$  stands for the reaction-based heat generation.

# Chapter 3

## Tutorials

### 3.1 Explanation of a tutorial case for reactingFoam

As described earlier, `reactingFoam` is a solver capable of solving combustion problems with chemical reactions. Below is the description of the solver in the source code.

```
1 //Description
2 //Solver for combustion with chemical reactions.
```

Since the emphasis of the current tutorial is on the chemical reactions, this topic will be discussed in more detail here. The case that has been selected for this tutorial is an example that is available in the `tutorials` directory. It is recommended that tutorial cases not be executed in their original directory. Here, it is first copied to the `run` directory.

```
cp -r $FOAM_TUTORIALS/combustion/reactingFoam/laminar/counterFlowFlame2D $FOAM_RUN
cd $FOAM_RUN/counterFlowFlame2D
```

#### 3.1.1 Geometry and boundary conditions

Figure 3.1 shows the geometry and the boundary conditions of the `counterFlowFlame2D` case. The fuel composed of methane enters the left boundary with a velocity of 0.1 m/s. Air, consisting of 23% oxygen and 77% nitrogen, enters the right boundary with the same velocity. The top and bottom boundaries are set as outlet, in which the `inletOutlet` boundary condition is specified. The boundaries perpendicular to the z-direction are set to `empty` to make the simulation two-dimensional.

#### 3.1.2 System directory

The system directory consists of `blockMeshDict`, `controlDict`, `fvSchemes`, `fvSolution`, and a file named `FOBilgerMixtureFraction`. In the `blockMeshDict` file, the setting for the two-dimensional mesh is set. At the end of the `controlDict` file, the file `FOBilgerMixtureFraction` is included inside the `functions` subdictionary. In this file, a `functionObject` with the name `BilgerMixtureFraction` is called. According to the source code of this `functionObject`, it calculates the Bilger mixture-fraction field based on the elemental composition of the mixture.

In the `fvSchemes` and `fvSolution` files, only the terms used for the chemical reactions will be explained. In the solver, the same convection scheme is used for both energy and species transport equations. This scheme is set to `limitedLinear` for the current case.

```
1 div(phi,Yi_h) Gauss limitedLinear 1;
```

The `PBiCGStab` solver with `DILU` preconditioner and the absolute tolerance of `1e-6` is used to solve the matrix equation of the mass fractions of the species.

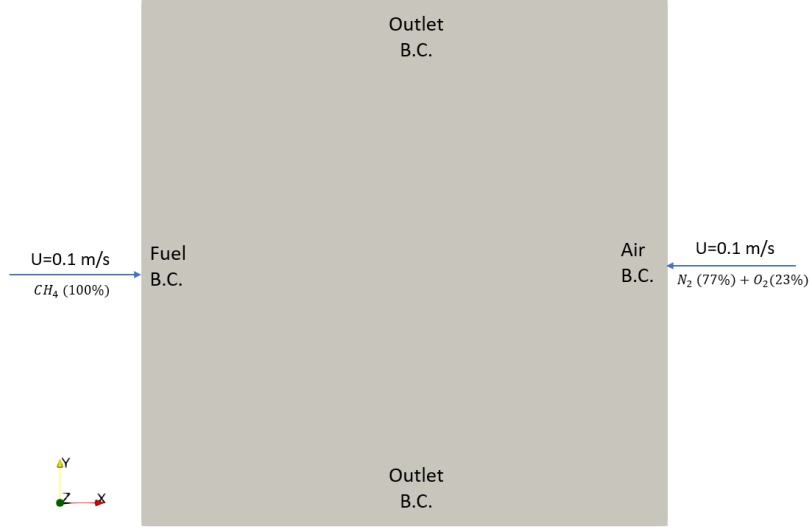


Figure 3.1: Geometry and boundaries of counterFlowFlame2D case

```

1 "(U|h|k|epsilon)"
2 {
3     solver          PBiCGStab;
4     preconditioner  DILU;
5     tolerance       1e-6;
6     relTol          0.1;
7 }
8
9 "(U|h|k|epsilon)Final"
10 {
11     $U;
12     relTol          0;
13 }
14
15 Yi
16 {
17     $hFinal;
18 }

```

### 3.1.3 Constant directory

In the constant directory, turbulence model, reactions, and thermophysical model and properties are defined. The flow regime is set to laminar in the **turbulenceProperties** file.

In the **thermophysicalProperties** file, which is already explained in previous reports of the course [3], the thermophysical settings of the mixture are set. The transport properties of the system, namely  $\nu$ ,  $\kappa$ , and  $\alpha$  are calculated according to the **sutherland** equation, which requires two constants  $A_s$  and  $T_s$  to evaluate the transport properties as a function of temperature. These two constants are defined inside the **transport** sub-dictionary for each specie. The **janaf** model is used to calculate the specific heat value ( $C_p$ ) as a function of temperature according to the following equation,

$$c_p = R((((a_4 T + a_3) T + a_2) T + a_1) T + a_0) \quad (3.1)$$

in which  $T$  is temperature, and  $a_0$  to  $a_4$  are some constants that should be given as input to the model. To evaluate the enthalpy and entropy, the model uses two other constants,  $a_5$  and  $a_6$ , that should also be specified. The **janaf** model requires all these constants to be defined at two temperature

ranges. The first set of constants, for the temperature range between `Tcommon` and `Thigh`, are defined in front of `highCpCoeffs` keyword. The second set of constants, defined in front of `lowCpCoeffs`, identifies the corresponding factors between the temperatures `Tlow` and `Tcommon`. The keywords should be defined inside the `thermodynamics` sub-dictionary for each specie. The `inertSpecie` keyword determines the species that does not take part in the reactions and is mandatory to be defined. At the end of the `thermoPhysicalProperties` file, the `foamChemistryReader` class is called to read the `reactions` and `thermo.compressibleGas` files. In the `reactions` file, the chemical reactions, the species taking part in chemical reactions, and the elements of the species are defined. In the `thermo.compressibleGas` file, the thermophysical constants of different species are specified. The keywords used in this file are compatible with the models chosen in the `thermoPhysicalProperties` file.

### 3.1.4 0 directory

In the 0 directory, the initial and boundary conditions for different equations are defined. The boundary conditions of this case are shown in Figure 3.1. Initially, the whole domain is filled with  $N_2$ , which is the inert specie. Initial and boundary conditions for each specie are defined in a file that has the name of that specie. If the file of one of the species involving in the chemical reaction is not included in the 0 directory, then the solver will use the boundary and initial conditions defined in the `Ydefault` file. The `alphat` file defines the initial and boundary conditions for the turbulent thermal diffusivity, which is redundant for the current case as the flow is laminar.

### 3.1.5 Results

Figure 3.2 shows the results of this tutorial case. In the first row of the figure, the mass fraction of  $CH_4$ , the fuel, is shown. The fuel enters from the left boundary and it burns as soon as it reaches oxygen that is entering from the right boundary. Comparing the variables shown in Figure 3.2 at  $t = 0.25$  s and  $t = 0.5$  s reveals minimal change between these two moments, implying that the problem reaches steady-state around  $t = 0.25$  s. Oxygen is also burnt instantly as it reaches the combustion zone, in which its mass fraction declines to zero as depicted in the second row of Figure 3.2. In the third and fourth rows of the figure, the mass fraction of  $CO_2$ , and the heat generation rate are shown, respectively. At  $t = 0.05$  s the combustion reactions have just started and the mass fraction of  $CO_2$  and the generated heat are quite small compared to the later times. The other two snapshots for each of these variables depict the steady-state condition.

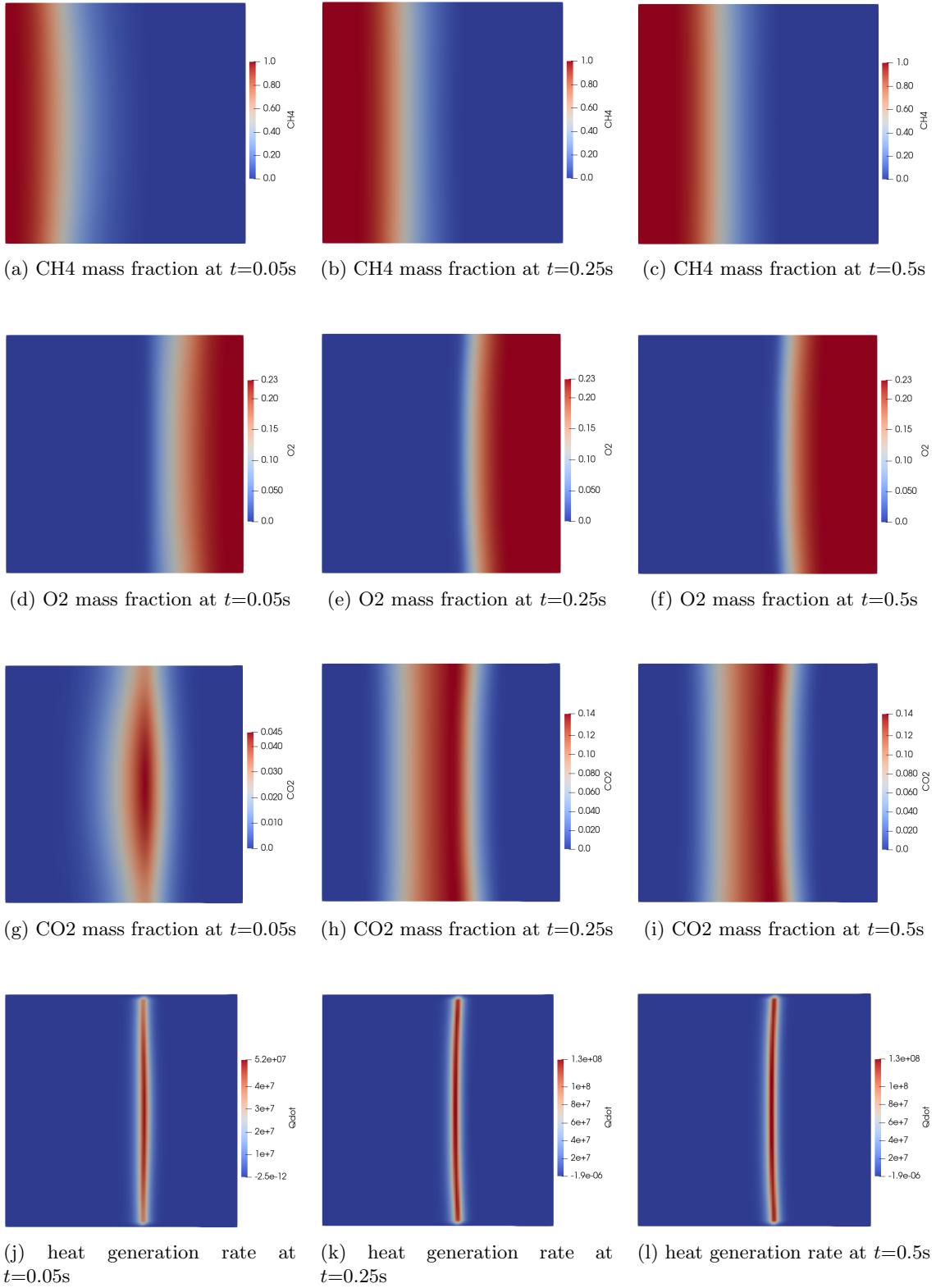


Figure 3.2: Results of the tutorial case counterFlowFlame2D.

## Chapter 4

# Implementation of the original solver and relevant libraries

Some description about the `reactingFoam` solver implementation has already been discussed in previous reports [4]. The focus of the current tutorial is on the chemical reactions, so the parts of the `reactingFoam` solver or parts of the libraries that it uses that are related to the chemical reactions are more highlighted. The other parts will be briefly pointed out.

The solver is located in the `$FOAM_SOLVERS/combustion/reactingFoam` directory. In this directory, several files and folders exist. Other than the `Make` directory that is responsible for the compilation of the `reactingFoam` solver, other folders are related to some other solvers that use some of the files of the current solver. The main file of the `reactingFoam` solver is the `reactingFoam.C`, in which some other files including the ones inside the current directory are called. In the `createFields` file, which is included in the line 66 of the main file, most variables are defined and initialized. Inside the `createFields.H` file, a pointer of the type `psiReactionThermo` is created that points to an object of a class that is specified by the user in the `thermophysicalProperties` file. This object is responsible for the changes in the thermophysical properties. A reference of that object is then saved in a variable named `thermo`.

```
1 Info<< "Reading thermophysical properties\n" << endl;  
2 autoPtr<psiReactionThermo> pThermo(psiReactionThermo::New(mesh));  
3 psiReactionThermo& thermo = pThermo();  
4 thermo.validate(args.executable(), "h", "e");
```

Another variable with the name `composition` is used to save a reference of the composition of the mixture.

```
1 basicSpecieMixture& composition = thermo.composition();  
2 PtrList<volScalarField>& Y = composition.Y();
```

To see what the function `composition()` returns, one of the derived classes of `basicThermo` should be examined to see how the function is implemented. In the `counterFlowFlame2D` tutorial explained in the previous chapter, `hePsiThermo` is specified for the type of the thermophysical model. The function `composition` is not defined inside the `hePsiThermo` class, so this function should be searched in its base class. In `heThermo`, the definition of `composition` function is given as below.

```
1 //- Return the composition of the mixture  
2 virtual typename MixtureType::basicMixtureType&  
3 composition()  
4 {  
5     return *this;  
6 }
```

It can be seen that here a pointer to the class is returned. At the beginning of the definition of the `heThermo` class, it can be seen that this class is a sub-class of the mixture type that will be specified

by the user in the `thermophysicalProperties` dictionary. Accordingly, what happens here is that the `composition` function returns an object of the `heThermo` class, which is inherited from a mixture type class, and have access to its functions.

In the next line of the `createFields` file of the `reactingFoam` solver, a variable named `Y` saves a reference of the `Y()` function that is called on the `composition` variable.

```
1 PtrList<volScalarField>& Y = composition.Y();
```

The function `Y` is defined inside the `basicMultiComponentMixture` class, and returns a pointer list of the mass fractions of the species, `Y_`.

```
1 inline Foam::PtrList<Foam::volScalarField>&
2 Foam::basicMultiComponentMixture::Y()
3 {
4     return Y_;
5 }
```

Then, after initiating the main variables of the model, namely  $\rho$ ,  $U$ , and  $p$ , the turbulence model is instantiated in the `createFields` file. Afterwards, a pointer with the name `reaction` of the type `CombustionModel` is defined.

```
1 autoPtr<CombustionModel<psiReactionThermo>> reaction
2 (
3     CombustionModel<psiReactionThermo>::New(thermo, turbulence())
4 );
```

This pointer will point to an object that will be defined during the run time based on the settings in the `combustionProperties` file. Before proceeding any further, the structure and functions of the combustion models that are relevant for a chemical reaction modeling will be discussed. In Figure 4.1, the hierarchy of the combustion models are shown. The green rectangles are abstract classes that cannot instantiate any object. However, it should be noted that a pointer of abstract classes can be created.

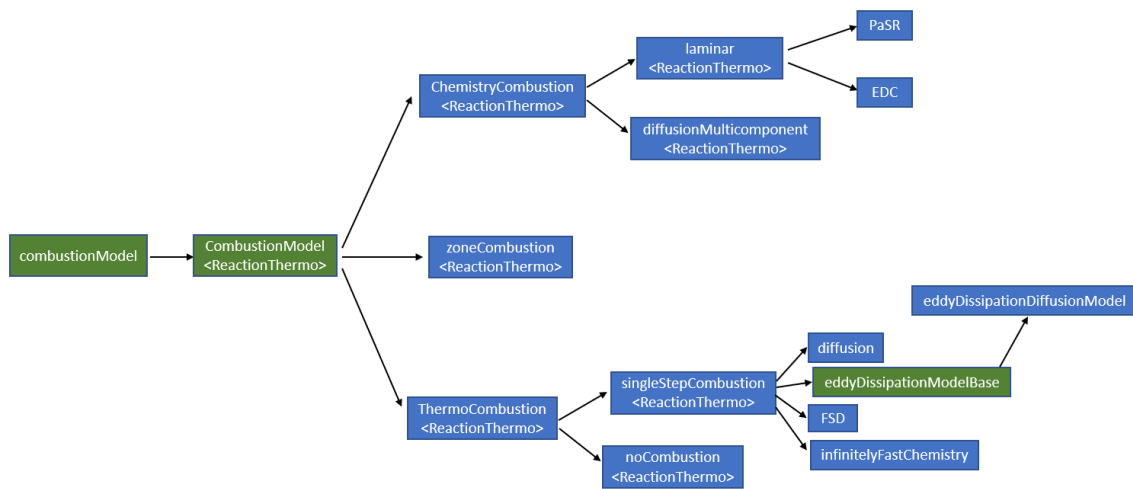


Figure 4.1: Inheritance structure of the combustion models

After some exploration through these classes, it is found out that the `laminar` combustion model is a suitable one for simulating the reactions without any extra term related to combustion physics. Herein, whenever a function of the `reaction` object is used, that function inside the `laminar` combustion model will be discussed.

At the end of the `createFields.H` file, a new field variable with the name `Qdot` is defined that stores the reaction-induced heat generation. If a file exists in the time directory, this variable will be initialized based on the file. Otherwise, it would be assigned a value of zero at this stage.

```

1 volScalarField Qdot
2 (
3     IOobject
4     (
5         "Qdot",
6         runTime.timeName(),
7         mesh,
8         IOobject::READ_IF_PRESENT,
9         IOobject::AUTO_WRITE
10    ),
11    mesh,
12    dimensionedScalar(dimEnergy/dimVolume/dimTime, Zero)
13 );

```

A general overview of the rest of the `reactingFoam.C` file has been given in reference [4]. Since this tutorial focuses on the reactions, some parts of the solver are skipped and the contents of `YEqn.H` will be explained. At the beginning of this file, a new convection scheme is defined to be used for the convection of the species in the species transport equations and the convection of enthalpy/internal energy in the energy equation.

```

1 tmp<fv::convectionScheme<scalar>> mvConvection
2 (
3     fv::convectionScheme<scalar>::New
4     (
5         mesh,
6         fields,
7         phi,
8         mesh.divScheme("div(phi,Yi_h)")
9     )
10 );

```

In the above piece of code, the divergence scheme that is defined in front of the `div(phi,Yi_h)` keyword, in the `divSchemes` dictionary inside the `fvSchemes` file, will be read and chosen as the convection scheme. In the next line, a function with the name `correct` is called. In the `laminar` combustion model this function is defined as below.

```

1 template<class ReactionThermo>
2 void Foam::combustionModels::laminar<ReactionThermo>::correct()
3 {
4     if (this->active())
5     {
6         if (integrateReactionRate_)
7         {
8             if (fv::localEulerDdt::enabled(this->mesh()))
9             {
10                 const scalarField& rDeltaT =
11                     fv::localEulerDdt::localRDeltaT(this->mesh());
12
13                 scalar maxTime;
14                 if (this->coeffs().readIfPresent("maxIntegrationTime", maxTime))
15                 {
16                     this->chemistryPtr_->solve
17                     (
18                         min(1.0/rDeltaT, maxTime)()
19                     );
20                 }
21                 else
22                 {
23                     this->chemistryPtr_->solve((1.0/rDeltaT)());
24                 }
25             }
26             else
27             {
28                 this->chemistryPtr_->solve(this->mesh().time().deltaTValue());
29             }
30 }

```



```

31     else
32     {
33         this->chemistryPtr_->calculate();
34     }
35 }
36 }

```

Here, `chemistryPtr_` is a pointer of type `BasicChemistryModel` defined in `chemistryCombustion` class, which is a base class of the current class, i.e., `laminar`. At the beginning of the above code, it checks if the `active` switch inside the `combustionProperties` is set to on or off. In case it is on, it executes the code inside the brackets. Otherwise, this function will do nothing. Inside the brackets, it is checked if the Boolean variable `integrateReactionRate_` is set to on or not. This variable, which is on by default and can be changed by the user inside `combustionProperties` file, specifies whether the reaction rate should be integrated over a time step or not. In case it is off, the `calculate` function of the `chemistryPtr_` is called. Otherwise, the `solve` function with different arguments will be called based on whether the local time stepping is active or not. Both functions `calculate()` and `solve()` are implemented inside the `StandardChemistryModel` class, a derived class of the `BasicChemistryModel`. `calculate()` is responsible for evaluating the reaction rates, and `solve()` function solves the reaction system at each time step. These functions will not be explained in more detail.

The next line in the file `YEqn.H` assigns the value of the reaction heat generation to the `Qdot` variable that was previously defined.

```

1 Qdot = reaction->Qdot();

```

Inside the `laminar` combustion model, the function `Qdot` is defined as below,

```

1 template<class ReactionThermo>
2 Foam::tmp<Foam::volScalarField>
3 Foam::combustionModels::laminar<ReactionThermo>::Qdot() const
4 {
5     ...
6
7     if (this->active())
8     {
9         tQdot.ref() = this->chemistryPtr_->Qdot();
10    }
11
12    return tQdot;
13 }

```

in which the `Qdot` function of the `chemistryPtr_` is called. This new `Qdot` function is defined inside the `standardChemistryModel` class as below.

```

1 template<class ReactionThermo, class ThermoType>
2 Foam::tmp<Foam::volScalarField>
3 Foam::StandardChemistryModel<ReactionThermo, ThermoType>::Qdot() const
4 {
5     ...
6
7     if (this->chemistry_)
8     {
9         scalarField& Qdot = tQdot.ref();
10
11         forAll(Y_, i)
12         {
13             forAll(Qdot, celli)
14             {
15                 const scalar hi = specieThermo_[i].Hc();
16                 Qdot[celli] -= hi*RR_[i][celli];
17             }
18         }
19     }
20 }

```

```

21     return tQdot;
22 }

```

First, it checks if the `chemistry` switch inside the `chemistryProperties` file is on. Then it calculates the heat of all reactions based on the following formula,

$$\dot{Q}_{\text{released}} = -\left( \sum_{\text{products}} H_f \dot{R} - \sum_{\text{reactants}} H_f \dot{R} \right) \quad (4.1)$$

in which  $H_f$  is the enthalpy of formation, and  $\dot{R}$  is the production or consumption rate of each species. In the above piece of code, `Hc()` returns the enthalpy of each species that is calculated differently based on the thermodynamic model chosen by the user. `RR_` is the signed rate of production or consumption for each specie, positive for production and negative for consumption.

Getting back to `reactingFoam` solver, the variable `Qdot` that is evaluated in the `YEqn.H` file will be later used as a source term in the energy equation inside the `EEqn.H` file. The next line inside the `YEqn.H` file creates a `volScalarField` variable with the value of zero.

```

1     volScalarField Yt(0.0*Y[0]);

```

This variable will be used later to calculate the mass fraction of the inert specie. Then, a `forAll` loop is done over all the species to solve a transport equation for each one.

```

1     forAll(Y, i)
2     {
3         if (i != inertIndex && composition.active(i))
4         {
5             volScalarField& Yi = Y[i];
6
7             fvScalarMatrix YiEqn
8             (
9                 fvm::ddt(rho, Yi)
10                + mvConvection->fvmDiv(phi, Yi)
11                - fvm::laplacian(turbulence->muEff(), Yi)
12                ==
13                reaction->R(Yi)
14                + fvOptions(rho, Yi)
15            );
16
17            YiEqn.relax();
18
19            fvOptions.constrain(YiEqn);
20
21            YiEqn.solve(mesh.solver("Yi"));
22
23            fvOptions.correct(Yi);
24
25            Yi.max(0.0);
26            Yt += Yi;
27        }
28    }

```

At the beginning of this loop, it is checked if the species is not the inert species, and is active. Then, an object of the `fvScalarMatrix` class is created in which the transport equation of each specie is discretized. It should be noted that the previously created convection scheme, `mvConvection` is used here. Two terms are used as the source terms for this equation, `fvOptions` that enables the user to add source terms through a file with the same name without modifying the solver, and `R` function that is called through the `reaction` pointer. As described previously, `reaction` is a pointer of the type `CombustionModel` that is pointing at an object created during run time, which is a derived class of `CombustionModel` class. Here, `laminar` class is assumed to be the selected one, since it is a proper combustion model for modeling chemical reactions. Accordingly, the `R` function inside `laminar` class is shown below.

```

1     template<class ReactionThermo>

```

```

2 Foam::tmp<Foam::fvScalarMatrix>
3 Foam::combustionModels::laminar<ReactionThermo>::R(volScalarField& Y) const
4 {
5     tmp<fvScalarMatrix> tSu(new fvScalarMatrix(Y, dimMass/dimTime));
6
7     fvScalarMatrix& Su = tSu.ref();
8
9     if (this->active())
10    {
11        const label specieI =
12            this->thermo().composition().species()[Y.member()];
13
14        Su += this->chemistryPtr_->RR(specieI);
15    }
16
17    return tSu;
18 }

```

In the above piece of code, it can be seen that first, a `tmp` pointer of `volScalarField` variable is created. After checking the active state of the model, the `RR` function of the `chemistryPtr_` pointer is saved in the variable that is returned at the end of the function. To understand how the `RR()` function is implemented, the `StandardChemistryModel` class should be checked. Inside this class, it can be seen that `RR()` function simply returns the `RR_` variable, which stores the production/consumption rate of each species.

```

1 template<class ReactionThermo, class ThermoType>
2 inline const Foam::DimensionedField<Foam::scalar, Foam::volMesh>&
3 Foam::StandardChemistryModel<ReactionThermo, ThermoType>::RR
4 (
5     const label i
6 ) const
7 {
8     return RR_[i];
9 }

```

The `RR_` variable, itself, is calculated inside both the `calculate()` and `solve()` functions of the `StandardChemistryModel` class. Based on the settings that the user has specified in the dictionaries, one of these functions will be called inside the `correct()` function, which was called before through the `reaction` pointer.

Getting back to the `YEqn.H` file, the transport equation of each specie is solved after it is formed and discretized with `fvScalarMatrix` object, and the mass fraction is added to the `Yt` variable. After the `forAll` loop, the mass fraction of the inert species is evaluated by subtracting the `Yt` variable from 1.

```

1 Y[inertIndex] = scalar(1) - Yt;

```

which is equivalent to the following equation.

$$Y_{\text{inertSpecie}} = 1.0 - \sum_{i=0}^{i \neq \text{inertSpecie}} Y_i \quad (4.2)$$

## Chapter 5

# reactingCureFoam

In this chapter, a new solver, called `reactingCureFoam` will be implemented to model the photopolymerization process. One of the drawbacks of the `reactingFoam` solver is that it only works based on the molar mass of the species, and not their molar concentrations. However, the concentrations of the species are required for different calculations. Whenever the concentration of a species is needed, it is evaluated by using the molar mass (molecular weight) of the species. In some applications, such as photopolymerization, when the resin cures, a monomer unit crosslinks with many others. As the final size of the polymeric chain is not determined, it is not possible to define a molar mass for it. The `reactingFoam` solver only accepts the molar mass of the species in the `thermoPhysicalProperties` dictionary, and there is no other option to work with the concentrations of the species instead of the molar mass fractions. Consequently, this solver cannot be used to model the photopolymerization process. On top of that, the reaction rates and the property variation functions related to photopolymerization cannot be modeled with one of the predefined functionalities that are already available.

It is possible to modify the libraries that are used by the `reactingFoam` solver to accommodate the necessary changes. However, modifying the libraries would be cumbersome in this case since lots of modification are needed. Accordingly, a new solver based on two new libraries is defined here.

In the three sections that follow, the general overview of the new libraries, new solver, and the preparation of a case for the solver are discussed. The libraries will be created almost from scratch and the reader will get a sense of how to build a new library.

### 5.1 photoPolymerization library

In this library, the property development functions and the reaction constants will be calculated. Both of the property variation functions and the reaction constants are dependent on the conversion degree of the photopolymerization and the temperature. The development of these libraries from scratch is discussed here. Two classes will be defined. The *photoCure* class that will mainly contain the mixture properties, and the *cureReaction* class that is inherited from the *photoCure* class and defines functions to calculate the reaction rates.

#### 5.1.1 photoCure class

First, a new folder in the `src` folder in the user directory is created.

```
cd $WM_PROJECT_USER_DIR/src
mkdir photoPolymerization
cd photoPolymerization
```

Then, a header file and a main C++ file in the OpenFOAM format are needed. These two files can be easily created with the `touch` command, and then copying the header section from other existing

classes. Here, an existing class will be copied to keep the header section. Since the contents of the class will be deleted, there is no difference between different classes. Here, `viscosityModel` class is used.

```
cp -r $FOAM_SRC/transportModels/incompressible/viscosityModels/viscosityModel ./
mv viscosityModel photoCure
cd photoCure
rm viscosityModelNew.C
mv viscosityModel.H photoCure.H
mv viscosityModel.C photoCure.C
```

The new class is named `photoCure`. After modifying the header section (renaming and adding a description), the other parts should be deleted in both files. Inside the `photoCure.H` the variables and functions will be declared. In this file, all the declarations should be included between the following lines.

```
1 #ifndef photoCure_H
2 #define photoCure_H
3
4 ... Contents of the file
5
6 #endif
```

This part of the code, prevents the multiple inclusion of a class inside a main cpp file.

First, the protected data and members of the class are declared. Hence, the keyword `protected` will be used, and the protected variables will be declared. This class, similar to other classes in OpenFOAM, should be part of the `Foam` namespace. It is responsible to read the properties of the species from a file. Accordingly, a `protected` variable of type `dictionary` is declared.

```
1 const dictionary specieProperties_;
```

The class also needs access to temperature field, conversion degree field, and the mesh. Consequently, corresponding `protected` variables are declared.

```
1 // Temperature field [K]
2 const volScalarField& T_;
3
4 // Conversion degree [dimless]
5 const volScalarField& X_;
6
7 //Storing the mesh
8 const fvMesh& mesh_;
```

In the next stage, some variables of type `dimensionedScalar` are declared to store the properties of the species. Since the number of these variable is quite high, just the declaration of the first one is shown here. The reader can find the complete file in the Appendix section. For each variable, a short description is added as a comment to make it easy to understand.

```
1 //Monomer diffusion coefficient
2 const dimensionedScalar Dm_;
```

Afterwards, the properties of the mixture are declared as `volScalarField` variables.

```
1 // * * * * * Properties of the mixture * * * * *
2
3 volScalarField rho_; // Density of mixture
4
5 volScalarField Cp_; // Specific heat of mixture
6
7 volScalarField mu_; // Viscosity of mixture
8
9 volScalarField kappa_; // Thermal conductivity of mixture
10
11 volScalarField alphas_; // Thermal diffusivity of mixture alpha = kappa/Cp
```

Subsequently, the `public` members are declared. Public members include the `constructor` of the class and the functions that update the properties of the mixture.

```

1 // Constructors
2
3 photoCure
4 (
5     const dictionary& specieProperties,
6     const volScalarField& T,
7     const volScalarField& X
8 );

```

The `constructor` gets a dictionary containing the properties of the species, the temperature field, and the conversion degree field as arguments. It will be defined in the `photoCure.C` file.

The other member functions that are declared here are also responsible for calculating and updating the properties of the mixture.

```

1 // Member Functions
2
3 //Calculates the Cp value of the species
4 tmp<volScalarField> calcAndGetCpi
5 (
6     const dimensionedScalar Cp0,
7     const dimensionedScalar Cp1,
8     const volScalarField& T
9 );
10
11 //Calculates the Cp value of the mixture
12 void calcCp();
13
14 //Calculates the viscosity of the mixture
15 void calcMu();
16
17 //Calculates the density of the mixture
18 void calcRho( const volScalarField& Ym);
19
20 //Calculates the thermal diffusivity of the mixture
21 void calcAlphat();
22
23 //Updates the properties of the mixture
24 virtual void correct();

```

The description that is provided for each function in the above piece of code describes its responsibility. The functions will be defined in the `photoCure.C` file. It should be noted that the `correct` function is declared as a `virtual` one, since a function with a similar name is going to be defined in a derived class, and the concept of polymorphism is going to be utilized.

Next, the access functions are declared and defined. These members are defined to give access to the properties of the mixture and some properties of the species such as diffusion coefficients. The declaration and definition of the first access member is shown below. The other ones are similar.

```

1 tmp<volScalarField> rho() const
2 {
3     return rho_;
4 }

```

At the end of the `photoCure.H` file, some pure virtual functions are declared. These functions will be defined in the derived class.

```

1 // * * * Pure virtual functions * * *
2
3 virtual tmp<volScalarField> Ki() = 0;
4 virtual tmp<volScalarField> Kp() = 0;
5 virtual tmp<volScalarField> Kt() = 0;
6 virtual const dimensionedScalar heat() = 0;
7 virtual const dimensionedScalar fi() = 0;

```

Based on the classes that have been used inside this file, four header files should be included. The inclusion should be done before the declaration of the **Foam** namespace.

```

1 #ifndef photoCure_H
2 #define photoCure_H
3
4 #include "dictionary.H"
5 #include "fvMesh.H"
6 #include "dimensionedScalar.H"
7 #include "volFields.H"
8
9 // * * * * *
10
11 namespace Foam

```

The contents of the **photoCure.H** file have been explained. Now, the **photoCure.C** file, which contains the definitions of the functions, will be discussed. At the beginning of this file, the **photoCure.H** file should be included.

```

1 #include "photoCure.H"

```

Then, the constructor is defined.

```

1 Foam::photoCure::photoCure
2 (
3     const dictionary& specieProperties,
4     const Foam::volScalarField& T,
5     const Foam::volScalarField& X
6 )
7 :
8     specieProperties_(specieProperties),
9     T_(T),
10    X_(X),
11    mesh_(T_.mesh()),
12    ...

```

After the colon, the variables that were previously declared, are initialized. The **specieProperties\_** dictionary is set to a **dictionary** variable that is given as an argument to the **constructor**. The variables related to the properties of the species are initialized using this dictionary, considering the structure that is defined for this dictionary to read the values.

```

1 ...
2
3 Dm_("D", dimViscosity, specieProperties_.subDict("Monomer").subDict("transport")),
4 mu_m_("mu", dimViscosity*dimDensity, specieProperties_.subDict("Monomer").subDict("transport")),
5 kappa_m_("kappa", dimPower/(dimLength*dimTemperature), specieProperties_.subDict("Monomer").
6     subDict("transport")),
7 ...

```

Afterwards, the variables of type **volScalarField** are initialized. Most of these variables are initialized with the corresponding property value of the monomer. For instance, the density of the mixture  $\rho$  is defined and initialized as below.

```

1 rho_
2 (
3     IOobject
4     (
5         "rho",
6         mesh_.time().timeName(),
7         mesh_,
8         IOobject::NO_READ,
9         IOobject::AUTO_WRITE
10    ),
11    mesh_,
12    rho_m_
13 ),

```

In the block of the `constructor`, a function that is named `correct` is called. This function will be explained soon. For now, it suffices to mention that it updates and recalculates the fields.

Afterwards, the member functions are defined. First, a function is defined to evaluate the specific heat capacity of the monomer and polymer based on the temperature. So, this function gets the temperature and two constants as temperature and calculates the specific heat of the component according to Eq. 2.19 and 2.20.

```

1 Foam::tmp<Foam::volScalarField> Foam::photoCure::calcAndGetCpi
2 (
3     const dimensionedScalar Cp0,
4     const dimensionedScalar Cp1,
5     const volScalarField& T
6 )
7 {
8     tmp<volScalarField> Cpi = Cp0 + Cp1*T;
9
10    return Cpi;
11 }

```

Using the function that just described to evaluate the specific heat of the components, the below function updates the specific heat capacity of the mixture according to Eq. 2.21.

```

1 void Foam::photoCure::calcCp()
2 {
3
4     const volScalarField& Cp_m = calcAndGetCpi(Cp0_m_, Cp1_m_, T_());
5     const volScalarField& Cp_p = calcAndGetCpi(Cp0_p_, Cp1_p_, T_());
6
7     Cp_ = Cp_m*(1 - X_) + Cp_p*X_;
8
9 }

```

The `calcMu` function, updates the viscosity of the mixture in a similar fashion. The `calcRho` function updates the density field of the mixture, according to Eq. 2.24, based on the volume fraction of the monomer that is taken as an argument. The temperature-induced change of density has been disregarded in this tutorial so that an incompressible solver can be used.

```

1 void Foam::photoCure::calcRho( const volScalarField& Ym )
2 {
3     rho_ = rho_m_*Ym + rho_p_*(1 - Ym);
4 }

```

At the end, the `correct` function is defined, which simply calls the other functions to update the fields. Note that the `calcRho` function is not called here, since it gets an argument that will be defined in the derived class. So, this function will also be defined in the derived class.

```

1 void Foam::photoCure::correct()
2 {
3     calcCp();
4     calcMu();
5     calcAlphat();
6     // The calcRho function will be called in the derived class.
7 }

```

### 5.1.2 cureReaction class

As discussed before, the `cureReaction` class that is inherited from the `photoCure` class defines the functionalities of curing reactions. In what follows, it will be explained how to declare and define this class.

Here, the `photoCure` class is copied and modified.

```

cd $WM_PROJECT_USER_DIR/src/photoPolymerization
cp -r photoCure cureReaction

```



```
cd cureReaction
mv photoCure.H cureReaction.H
mv photoCure.C cureReaction.C
```

After modifying the header section of the `cureReaction.H` file, the header file of the `photoCure` class should be included.

```
1 #ifndef cureReaction_H
2 #define cureReaction_H
3
4 #include "photoCure.H"
5
6 // * * * * *
7
8 namespace Foam
9 ...
```

In the declaration of the class, it should inherit from the `photoCure` class.

```
1 class cureReaction
2 : public photoCure
```

And then the protected data are declared. Similar to the previous case, a dictionary is responsible to read the reaction parameters from a file. Monomer concentration is also declared as a reference variable, and it will be used in some of the calculations. Volume fraction of the monomer, free volume of the mixture, and the reaction rates are the other variables that are declared as `volScalarFields` type. The other variables that are of the type `dimensionedScalar` store the constant parameters of the reactions.

In the `public` section of the file, the functions that evaluate the reaction constants and the access functions are declared. The access functions, give access to the reaction constants, heat generation rate, and quantum efficiency of the photo-initiator, as shown below. The other functions will further be explained later.

```
1 // * * * * * Access functions * * * * *
2
3 // initiation reaction constant
4 tmp<volScalarField> Ki()
5 {
6     return Ki_;
7 }
8
9 // propagation reaction constant
10 tmp<volScalarField> Kp()
11 {
12     return Kp_;
13 }
14
15 // termination reaction constant
16 tmp<volScalarField> Kt()
17 {
18     return Kt_;
19 }
20
21 // heat generation rate [j/mol]
22 const dimensionedScalar heat()
23 {
24     return heat_;
25 }
26
27 // quantum efficiency of the photoinitiator
28 const dimensionedScalar fi()
29 {
30     return fi_;
31 }
```

The `cureReaction.C` file contains the definitions of the members that were previously declared. After the inclusion of the `cureReaction.H` at the top of this file, the `constructor` is defined.

```

1 Foam::cureReaction::cureReaction
2 (
3     const dictionary& reactionParameters,
4     const dictionary& specieProperties,
5     const Foam::volScalarField& T,
6     const Foam::volScalarField& X,
7     const Foam::volScalarField& M
8 )
9 :
10     photoCure(specieProperties, T, X),
11     reactionParameters_(reactionParameters),
12     M_(M),
13     ...

```

The `constructor` receives five arguments and pass three of them to the `constructor` of its base class, `photoCure`. The other two arguments, `reactionParameters` dictionary and the conversion degree variable are stored in the corresponding variables of the class. Then, the other protected variables are initialized as before. Similar to `photoCure` class, in the block of the `constructor`, the `correct` function is called, which corrects and calculates the fields. Its definition will be shown later.

The `calcYm` function, calculates and updates the volume fraction of the monomer according to Eq. (2.16).

```

1 void Foam::cureReaction::calcYm()
2 {
3     Ym_ = (scalar(1) - X_)/(scalar(1) - X_ + X_*rho_m_/rho_p_);
4
5     //Calling the calcRho function from the base class
6     calcRho(Ym_);
7 }

```

It can be seen that at the end of this function, the `calcRho` function that was defined in the base class is called. The reason is that the density of the mixture is dependent on the monomer volume fraction that is defined and evaluated in this class. Then, the `calcF` function is defined, which evaluates the mixture fraction free volume according to Eq. (2.13)-(2.15).

```

1 void Foam::cureReaction::calcF()
2 {
3
4     volScalarField fM = 0.025 + alpha_m_*(T_ - Tg_m_);
5
6     volScalarField fP = 0.025 + alpha_p_*(T_ - Tg_p_);
7
8     f_ = fM*Ym_ + fP*(1 - Ym_);
9
10 }

```

In the next stage, the critical free volume, which will be used in the calculation of the propagation and termination reaction rates, is defined according to Eq. (2.18).

```

1 Foam::tmp<Foam::volScalarField>
2 Foam::cureReaction::calcAndGetFc
3 (
4     const dimensionedScalar E,
5     const dimensionedScalar A,
6     const dimensionedScalar TrefR,
7     const dimensionedScalar fcRefR,
8     const volScalarField& T
9 )
10 {
11     tmp<volScalarField> fc =
12         fcRefR + (E/(A*Rconst_))*(scalar(1)/T_ - TrefR);

```

```

13     fc = scalar(1)/fc;
14
15     return fc;
16 }

```

Note that in the above piece of code, `TrefR` and `fcRefR` are the reciprocals of the reference temperature and the reference critical fractional free volume that were introduced in Eq. (2.18). Then, the propagation and termination constants are evaluated as per Eq. (2.9)-(2.12).

```

1 void Foam::cureReaction::calcKp()
2 {
3
4     const volScalarField& fcp =
5         calcAndGetFc(Ep_, Ap_, TrefRp_, fcRefRp_, T_());
6
7     Kp_ = Aep_*exp(-Ep_/(Rconst_*T_));
8
9     Kp_ /= (scalar(1) + exp(Ap_*(scalar(1)/f_ - scalar(1)/fcp)));
10
11 }
12
13 void Foam::cureReaction::calcKt()
14 {
15
16     const volScalarField& fct =
17         calcAndGetFc(Et_, At_, TrefRt_, fcRefRt_, T_());
18
19     volScalarField Kt0 = Aet_*exp(-Et_/(Rconst_*T_));
20
21     Kt_ = Kt0;
22
23     Kt_ /= (scalar(1) +
24         scalar(1)/(
25             Rrd_*Kp_*M_/Kt0
26             + exp(-At_*(scalar(1)/f_ - scalar(1)/fct))
27         )
28     );
29
30 }

```

It is important to note that no function has been defined to evaluate the initiation constant, Eq. (2.2). The reason is that this parameter does not change during the simulation, and the value that was assigned to it in the `constructor` will remain constant.

```

1 ...
2 Ki_
3 (
4     IOobject
5     (
6         "Ki",
7         mesh_.time().timeName(),
8         mesh_,
9         IOobject::NO_READ,
10        IOobject::AUTO_WRITE
11    ),
12    mesh_,
13    2.3*fi_*e_ // Ri = 2.3*fi*e*PI*I
14 ),
15 ...

```

Finally, the `correct` function is defined, which calls the other functions to update the fields. It also calls the `correct` function of the base class.

```

1 void Foam::cureReaction::correct()
2 {
3     photoCure::correct();
4     calcYm();

```

```

5     calcF();
6     calcKp();
7     calcKt();
8 }

```

The new classes should be compiled. For the compilation of the current library, the make directory that contains the files and option files should be created.

```

cd $WM_PROJECT_USER_DIR/src/photoPolymerization
mkdir Make
cd Make
touch files
touch options

```

The empty files with the names `files` and `options` are created inside the `Make` directory. In the `files` file, we determine the names of the files that should be compiled and the name of the created library. Here, the `photoCure.C` and `cureReaction.C` files should be compiled. The library is named *photoPolymerization*.

```

1 photoCure/photoCure.C
2 cureReaction/cureReaction.C
3
4
5 LIB = $(FOAM_USER_LIBBIN)/libphotoPolymerization

```

In the `options` file, we specify the directories from which files are included, and the libraries that are used in our classes. For this library, the `finiteVolume` library is used.

```

1 EXE_INC = \
2   -I$(LIB_SRC)/finiteVolume/lnInclude \
3   -I$(LIB_SRC)/meshTools/lnInclude
4
5 LIB_LIBS = \
6   -lfiniteVolume

```

Finally, the library is compiled by executing the `wmake` command.

```

cd ..
wmake

```

## 5.2 reactingCureFoam solver

When a resin undergoes the curing process, parts of it that surpass the energy threshold will experience considerable property changes. So, the total system of the cured and uncured portions of the resin can be viewed as a two-phase system. In this regard, the `interFoam` solver is chosen as the base for the new solver, and the concentrations of the species and the reactions are added to this solver after some modifications. The first step is to copy the `interFoam` solver to another directory to start modifying it. To mimic the structure of the original solvers, it is copied to the `solver` directory, which was previously created in the user directory for the solvers that are modified or generated by the user.

```

cd $WM_PROJECT_USER_DIR/applications/solvers
cp -r $FOAM_SOLVERS/multiphase/interFoam/ ./
mv interFoam/ reactingCureFoam

```

At first, the other two solvers, namely `interMixingFoam` and `overInterDyMFoam` should be deleted. Then the files that are not needed by the new solver, `rhofs.H` and `alphaSuSp.H` can be removed. `interFoam` solver originally reads some files from the `VoF` folder in the `multiphase` directory. All of the lines of the code in which a file related to the *alpha*-field is included should be

deleted, and this step is not shown here. The files `interFoam.C` and `createFields.H` are attached in the Appendix section for comparison. It is important to mention that `cureReaction.H` file should be included in the `reactingCureFoam` before the main function.

```

1 ...
2 #include "fvcSmooth.H"
3 #include "cureReaction.H" // Added
4
5 // * * * * *
6
7 int main(int argc, char *argv[])
8 ...

```

In the next stage, a field for the light intensity should be created and defined. It is more desirable to define the intensity field in the boundary conditions. However, no boundary condition that can make a field that varies both in time and position. Accordingly, it should be defined in the solver itself, but it reads a dictionary in the case directory to create the intensity field. The initialization of the intensity field is done in a new file called `createI.H`. At the beginning of this file, an object of the dictionary class is instantiated to read the parameters of the laser.

```

1 IOdictionary laserSettings
2 (
3     IOobject
4     (
5         "laserSettings",
6         runTime.constant(),
7         mesh,
8         IOobject::MUST_READ_IF_MODIFIED,
9         IOobject::NO_WRITE
10    )
11 );
12
13 dimensionedScalar Vs("Vs", dimVelocity, laserSettings); //laser scanning velocity
14
15 ...

```

All the parameters are defined as `dimensionedScalar` so that they will be dimensionally consistent when the equations are solved. Then, after creating the intensity field, it is initialized according to Eq. (2.5) by using the mesh cell center vectors.

```

1
2 const volVectorField& C = mesh.C();
3 const dimensionedScalar conv("conv", dimEnergy*dimLength/dimMoles, 1.196e8); //conversion constant
4 dimensionedScalar y("y", dimLength, Foam::scalar(0));
5 dimensionedScalar z("z", dimLength, Foam::scalar(0));
6 dimensionedScalar r2("r2", dimLength*dimLength, Foam::scalar(0));
7
8
9 forAll(C, counter)
10 {
11     //const scalar& x = C[counter].component(vector::X);
12     y.value() = C[counter].component(vector::Y);
13     z.value() = C[counter].component(vector::Z);
14     r2 = Vs*Vs*(t-t0)*(t-t0) + y*y;
15     I[counter] = (I0*Foam::exp(-2*r2/(w0*w0))*Foam::exp(-mag(z)/Dp)*lambda/conv).value();
16 }

```

In a similar fashion, another file with the name `initReact.H` is created that is responsible for the initialization of the parameters of the reaction. Similar to before, two `dictionary` objects are created to read the species' properties and the reactions' parameters. All of the dictionary files are assumed to be placed in the `constant` directory of the case. Afterwards, the concentration fields of monomer (M), photo-initiator (PI), and macro-radical (pDot) are created. The initialization of the monomer field is shown below as an example.

```

1 volScalarField M

```

```

2 (
3   IOobject
4   (
5     "M",
6     runTime.timeName(),
7     mesh,
8     IOobject::MUST_READ,
9     IOobject::AUTO_WRITE
10  ),
11  mesh
12 );

```

At the next stage, the initial value of the monomer concentration is stored in the variable M0 for the evaluation of the conversion degree.

```

1 // Saving the initial concentration of Monomer, M0
2 volScalarField M0(M);

```

The conversion degree is evaluated based on the change in the monomer concentration.

```

1 // Defining conversion
2 volScalarField X
3 (
4   IOobject
5   (
6     "X",
7     runTime.timeName(),
8     mesh,
9     IOobject::NO_READ,
10    IOobject::AUTO_WRITE
11  ),
12  (M0-M)/M0
13 );

```

Using the polymorphism concepts of c++, a pointer with the name reaction of type photoCure is created that points to an object of type cureReaction.

```

1 autoPtr<photoCure> reaction
2 (
3   new cureReaction
4   (
5     reactionParameters,
6     specieProperties,
7     T,
8     X,
9     M
10  )
11 );

```

Subsequently, the reaction constants are restored in new variables, and the reaction rates are defined.

```

1 Info<< " * * * * * \n";
2 Info<< "Initializing reaction rates \n" << endl;
3
4 const volScalarField& Ki = reaction -> Ki();
5 const volScalarField& Kp = reaction -> Kp();
6 const volScalarField& Kt = reaction -> Kt();
7
8 const dimensionedScalar& fi = reaction -> fi();
9
10 //reaction rates
11 volScalarField Ri = Ki*PI*I;           // Initiation reaction rate
12 volScalarField Rp = Kp*pDot*M;        // Propagation reaction rate
13 volScalarField Rt = Kt*pDot*pDot;     // Termination reaction rate

```

In the createFields.H file, temperature field is defined as a volScalarField after the definition of the velocity field, and the new files are added after the definition of temperature field.

```

1 Info<< "Reading field T\n" << endl;
2 volScalarField T
3 (
4     IOobject
5     (
6         "T",
7         runTime.timeName(),
8         mesh,
9         IOobject::MUST_READ,
10        IOobject::AUTO_WRITE
11    ),
12    mesh
13 );
14
15
16 //***** Creating intensity and reactions *****
17 #include "createI.H"
18 #include "initReact.H"
19 //*****

```

Afterwards, the properties of the mixture are restored in new variables.

```

1 const volScalarField& rho = reaction -> rho();
2 const volScalarField& mu = reaction -> mu();
3 const volScalarField& Cp = reaction -> Cp();
4 const volScalarField& kappa = reaction -> kappa();
5 const volScalarField& Alph = reaction -> alphas();
6 const dimensionedScalar& Dm = reaction -> Dm();
7 const dimensionedScalar& Dr = reaction -> Dr();
8 const dimensionedScalar& Ds = reaction -> Ds();
9 const dimensionedScalar& heat = reaction -> heat();

```

According to Tang [1], the main source of heat generation in photopolymerization is the propagation reaction. So, the rate of heat generation can be defined as the product of enthalpy of propagation and the reaction rate of propagation as defined below.

```

1 //***** Heat generation rate *****
2 Info<< "Creating field Qdot\n" << endl;
3 volScalarField Qdot
4 (
5     IOobject
6     (
7         "Qdot",
8         runTime.timeName(),
9         mesh,
10        IOobject::NO_READ,
11        IOobject::AUTO_WRITE
12    ),
13    Rp*heat
14 );

```

As the intensity is a function of time, it should be updated in the `reactingCureFoam` file at each time step. Accordingly, a file that is named `calcI.H` is included in the solver just after the calculation of the time step.

```

1 ...
2 ++runTime;
3
4 Info<< "Time = " << runTime.timeName() << nl << endl;
5
6 //***** Updating intensity*****
7 #include "calcI.H"
8 //*****
9
10 // --- Pressure-velocity PIMPLE corrector loop
11 while (pimple.loop())
12 ...

```

and the calcI.H file contains

```

1 t.value() = runTime.timeOutputValue();
2
3 forAll(C, counter)
4 {
5     y.value() = C[counter].component(vector::Y);
6     z.value() = C[counter].component(vector::Z);
7     r2 = Vs*Vs*(t-t0)*(t-t0) + y*y;
8     I[counter] = (I0*Foam::exp(-2*r2/(w0*w0))*Foam::exp(-mag(z)/Dp)*lambda/conv).value();
9 }

```

In the main file, the equations for specie transport equations and temperature can be added after the predictor step of the momentum equation, i.e., after `UEqn.H`.

```

1 #include "UEqn.H"
2 //***** Specie transport and Temperature Eqs*****
3 #include "CEqn.H"
4 #include "EEqn.H"
5
6 #include "updateR.H"
7 //*****

```

In the `CEqn.H` file, the specie transport equations for monomer, radical, and photo-initiator are solved. As an example, the monomer transport equation is shown below. To see the complete list of equations in this file, refer to the attached files.

```

1 // ***** Monomer equation *****
2 tmp<fvScalarMatrix> tMEqn
3 (
4     fvm::ddt(rho, M) + fvm::div(rhoPhi, M)
5     - fvm::laplacian(rho*Dm, M)
6     ==
7     -rho*Rp
8 );
9 fvScalarMatrix& MEqn = tMEqn.ref();
10 MEqn.solve();

```

In the `EEqn.H` file the energy equation is solved to find the temperature field within the domain.

```

1 {
2     Info<< "Solving the energy equation\n" << endl;
3
4
5
6     tmp<fvScalarMatrix> tEEqn
7     (
8         fvm::ddt(rho, T) + fvm::div(rhoPhi, T)
9         - fvm::laplacian(Alph, T)
10        ==
11        Qdot/Cp + fvOptions(rho, T)
12    );
13
14    fvScalarMatrix& EEqn = tEEqn.ref();
15
16
17    EEqn.relax();
18
19    fvOptions.constrain(EEqn);
20
21    EEqn.solve();
22
23    fvOptions.correct(T);
24
25 }

```



After solving the energy and the transport equations, the reaction rates and the heat generation field should be updated. This is done by including the `updateR.H` file, in which the mentioned fields are updated.

```

1  reaction -> correct();
2
3  Ri = Ki*PI*I;
4  Rp = Kp*pDot*M;
5  Rt = Kt*pDot*pDot;
6
7  Qdot = Rp*heat;

```

After completing the modifications in the solver, it should be compiled. Accordingly, the new library should be added to the `options` file in the `Make` directory. The lines that should be modified are shown below. Also, note that some lines related to the two-phase flow can be deleted. Please refer to the attached files to see the whole contents of this file.

```

1  EXE_INC = \
2      ...
3      -I$(LIB_SRC)/TurbulenceModels/phaseIncompressible/lnInclude \
4      -I../photoPolymerization/lnInclude
5
6  EXE_LIBS = \
7      ...
8      -lincompressibleTurbulenceModels \
9      -L$(FOAM_USER_LIBBIN) \
10     -lphotoPolymerization

```

In the `files` file, the name of the file that should be compiled and the name of the solver that is generated are specified.

```

1  reactingCureFoam.C
2
3  EXE = $(FOAM_USER_APPBIN)/reactingCureFoam

```

Compile the solver by executing the `wmake` command.

## 5.3 Setting up a tutorial case

The case is set up based on the simulation that is done in Tang's PhD thesis [1]. As described previously in Chapter 2 and shown in Figure 3.1, a symmetric 2D simulation can represent the photopolymerization process taking place in a SLA setup, in which the resin obeys the Beer-Lambert law, and the beam is Gaussian. Figure 5.1 shows the geometry and the boundary conditions of the case. In this figure,  $B$  stands for the concentration of each one of the species or the temperature, and  $B_i$  shows the initial state of the corresponding variable. It can be seen that the left and bottom boundaries are considered far enough so that the values of the concentrations and temperature are equal to the initial field. For the sake of this tutorial, the width and height of the computational domain are set to 8 mm and 0.7 mm, respectively. Symmetry boundary condition is imposed on the left boundary and Neumann boundary condition is applied on the top one for the specie transport and energy equations. Since the fluid is stationary, the velocity for all boundaries is set to zero. Although the solution of the momentum equation is not important for this benchmark, the solver is equipped with the incompressible Navier-Stokes equations to be capable of modeling cases in which there is fluid motion such as platform-induced flows in SLA and DLP or the curing process in VAM.

The initial condition of the variables are based on the statements that are mentioned in the thesis [1]. It is mentioned that 0.2wt% photoinitiator is added to the resin. Based on the datasheet of the 2,2-dimethoxy-2-phenylacetophenone (DMPA), the photoinitiator, its initial concentration is calculated to be  $8.8 \text{ mol/m}^3$ . However, in Figure 29b of the thesis, it can be seen that the photoinitiator concentration starts from  $90 \text{ mol/m}^3$ . Accordingly, the initial photoinitiator concentration is set to  $90 \text{ mol/m}^3$ . Initial monomer concentration is set to  $1974 \text{ mol/m}^3$  based on the 99.8% mass fraction that is mentioned in the thesis, and the datasheet of Ethoxylated (4) PentaErythritol (E4PETeA).

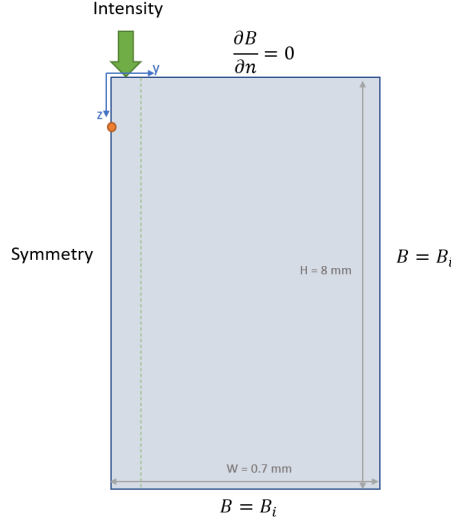


Figure 5.1: Geometry and the boundary conditions of the case prepared for the new solver.

The initial concentration of other species are initially zero. The temperature is initially set to 300 K. The initial conditions are summarized in Table 5.1.

Table 5.1: Initial values for concentrations and temperature

Variable [unit]	Initial Value
M [ $mol/m^3$ ]	1974
PI [ $mol/m^3$ ]	90
pDot [ $mol/m^3$ ]	0
T [K]	300

The parameters of the laser beam that are taken from Tang's thesis [1] are shown in Table 5.2.

Table 5.2: laser parameters

Parameter	Value	unit
$V_s$	0.0272	m/s
$\lambda$	325	nm
P	0.0288	W
$w_0$	1.1e-4	m
$D_p$	2.483e-3	m
$t_0$	0.02	s

In Table 5.3, the properties of the species are mentioned. Again, the values are taken from Tang's thesis except for the viscosity values. The viscosity of the monomer is taken from its datasheet, and the viscosity of the polymer is assigned equal to the monomer's since the flow is stagnant and the viscosity does not play a role. In the thesis, specific values are not given for the diffusion coefficients and some ranges are specified. Accordingly, some approximate values that are within the specified ranges of the thesis are used here. For density, as mentioned before, the temperature dependence has been ignored so as to use the incompressible formulations. So, the density of monomer and polymer are evaluated from Eqs. (2.22) and (2.23) at 40 °C.

The reaction parameters that are listed in the Table 5.4, except for the ones related to the critical fractional free volume variable are taken from Table 4 of Tang's thesis [1]. To calculate the critical

Table 5.3: Properties of species

Monomer			Polymer		
Property	Value	unit	Property	Value	unit
$D_m$	$1e^{-14}$	$m^2/s$	-	-	-
$\mu_m$	0.15	Pa.s	$\mu_p$	0.15	Pa.s
$\kappa_m$	0.142	W/m/k	$\kappa_p$	0.142	W/m/k
$T_{g,m}$	205.65	K	$T_{g,p}$	488.35	K
$\alpha_m$	0.00177	1/K	$\alpha_p$	0.00012	1/K
$Cp_{0,m}$	218.6	J/kg/K	$Cp_{0,p}$	-1535.5	J/kg/K
$Cp_{1,m}$	5.6	J/kg	$Cp_{1,p}$	9.1	J/kg
$\rho_m$	1099	$kg/m^3$	$\rho_p$	1199	$kg/m^3$
Macroradical			Photoinitiator		
$D_{pD}$	$1e^{-16}$	$m^2/s$	$D_{PI}$	$1e^{-14}$	$m^2/s$

fractional free volume, as per Eq. (2.18), two constants are needed,  $f_c^{ref}$  and  $T^{ref}$ . The reciprocal of these two constants are read from Figure 15 of the thesis. In table XX, **TrefRp** and **TrefRt** refers to the reciprocal of reference temperature for propagation and termination, respectively, and **fcRefRp** and **fcRefRt** refer to the reciprocal of the reference critical free volume at the reference temperature for the propagation and termination, respectively.

Table 5.4: Reaction parameters

Parameter	Value	unit
$\Phi$	0.6	1
$\epsilon$	19.9	$m^3/mol/m$
h	$2.85e^5$	J/mol
Propagation		
$A_p$	6.1	1
$A_{ep}$	28.4	$m^3/mol/s$
$E_p$	1627	J/mol
TrefRp	0.0031	1/K
fcRefRp	6	1
Macroradical		
$A_t$	6.4	1
$A_{et}$	8916	$m^3/mol/s$
$E_t$	2103	J/mol
$R_{rd}$	0.013	$m^3/mol$
TrefRt	0.0031	1/K
fcRefRt	6	1

This case can be based on any existing tutorial cases. Here, the **counterFlowFlame2D** is used. First, we go to the **run** directory and copy the tutorial case to the that directory, and rename it.

```
cd $FOAM_RUN
cp -r $FOAM_TUTORIALS/combustion/reactingFoam/laminar/counterFlowFlame2D/ ./
mv counterFlowFlame2D/ cureCase
```

In the constant directory, only the four dictionaries **g**, **laserSettings**, **reactionParameters**, and **specieProperties** are needed. For this case, gravity is deactivated. In the other dictionaries, the parameters that are shown in Tables 5.2-5.4 are shown. These dictionaries are also attached to the Appendix.

To show the performance of the new solver, different variables are plotted at point (0 0 -0.001), which is shown by a red dot in Figure 5.1. In the Tang's thesis, a similar set of graphs are depicted. However, he did not exactly determined the point at which the graphs are generated, so the results of this study show the same behavior, but they do not exactly match his results as the location of the graphs are different.

Figure 5.2 shows the intensity variation at point (0 0 -0.001) over time. As shown in Table 5.2 by  $t_0$ , the center of the laser beam reaches this point at  $t = 0.02$  depicted by a peak in Figure 5.2. Then, the Gaussian beam will continue its way in the x-direction and the intensity at the point would decay to zero. In Figure 5.3, photoinitiator concentration at the mentioned point is depicted. Before the beam reaches the point, the photoinitiator concentration is constant at the initial value. The intensity of the beam triggers a set of reactions described in Eqs. (2.1)-(2.4), in which the photoinitiator will be consumed. Accordingly, a drastic drop in its concentration can be seen in Figure 5.3. After the pass of the beam, the photoinitiator concentration is stabilized at a new level. It can also be seen that the species diffusion coefficients are so small that they do not almost have any effect on the results.

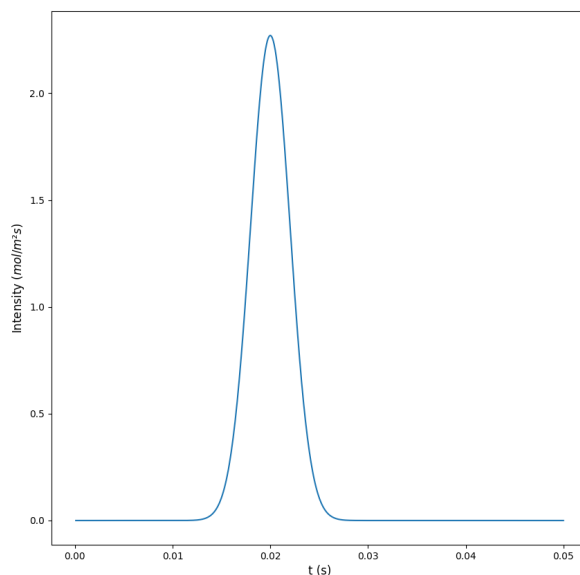


Figure 5.2: Intensity versus time at point (0,0,-0.001)

Figure 5.4 illustrates the radical concentration variation over time at the point (0 0 -0.001). Initially, the its concentration is zero in the domain. By laser arrival and the decomposition of the photoinitiators into radicals, the amount of radicals surges temporarily. The rise in the concentration of radicals will in turn boost the propagation reaction rate, Eq. (2.7), which will deplete the radicals almost as fast as they were generated. After the very sharp drop in the radical concentration, it gradually approaches its initial value of zero.

Figures 5.5 and 5.6, depicting monomer conversion and temperature variations at point (0 0 -0.001) respectively, are shown over the larger time scale of one second. Both the conversion degree and the temperature experience a sharp rise when the laser reaches the considered point. Then, the conversion degree continues to gradually increase due to the small number of radicals that are still present in the considered location. On the other hand, the temperature starts to fall soon after the disappearance of the laser beam, since the heat loss through diffusion into the surrounding environment dominates the heat generation from the slow reactions.

Figure 5.7 shows some isovalues or contours of the monomer conversion at  $t=0.4s$ . The contours

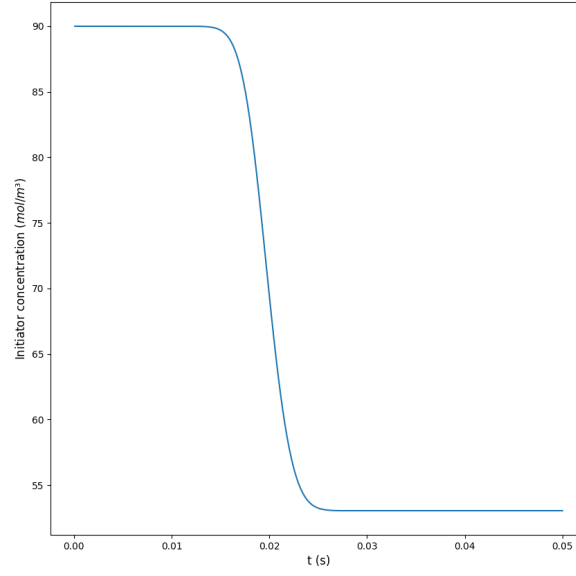


Figure 5.3: Photoinitiator concentration versus time at point (0,0,-0.001)

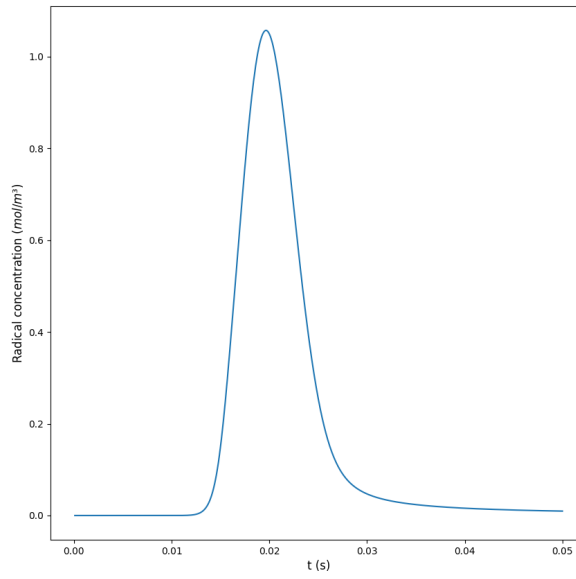


Figure 5.4: Macroradical concentration versus time at point (0,0,-0.001)

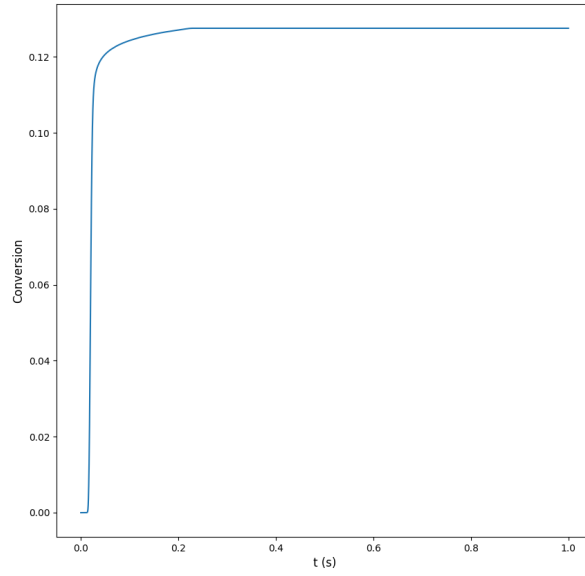


Figure 5.5: Conversion degree versus time at point (0,0,-0.001)

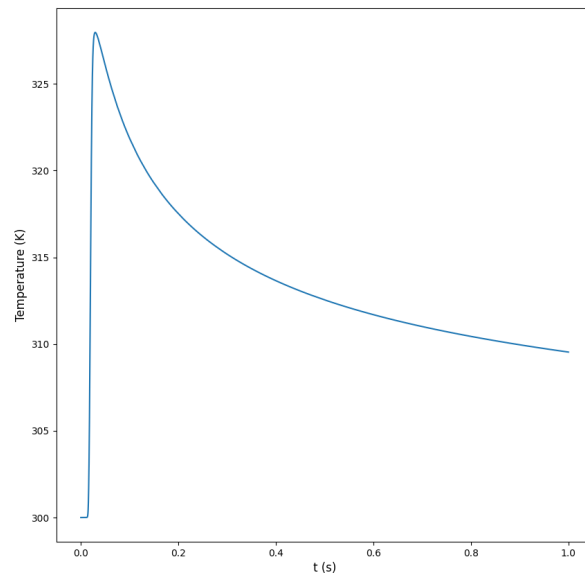
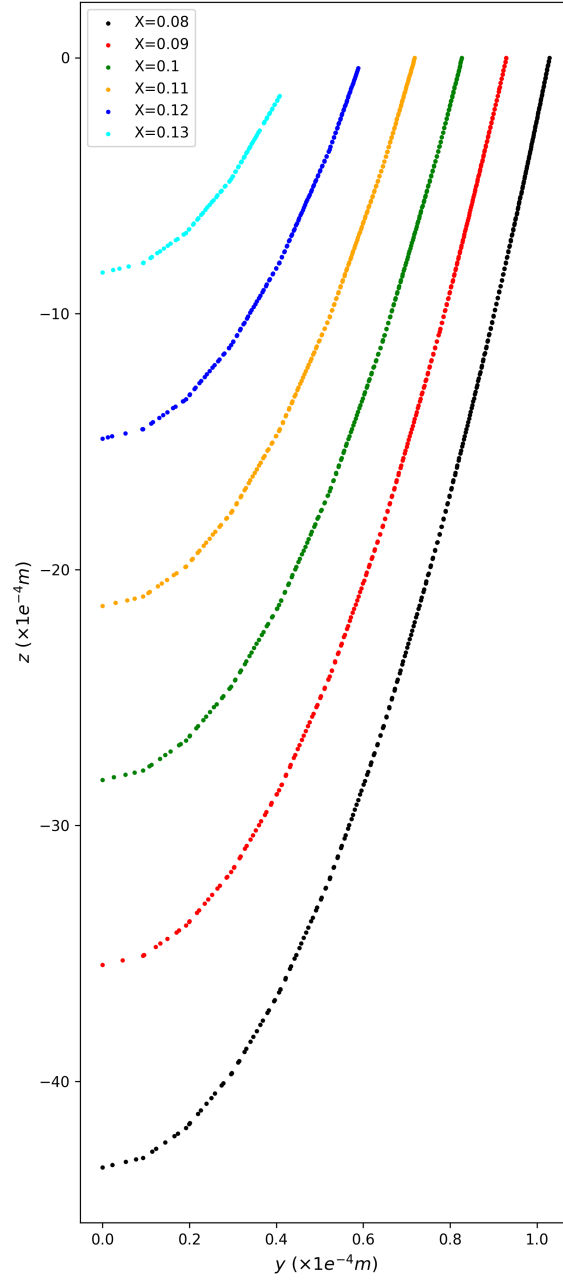


Figure 5.6: Temperature versus time at point (0,0,-0.001)

of conversion are parabolas that become deeper at lower conversion values. In the figure, it can be seen that the higher conversion contours are closer to the surface of the resin, which is expected since the top layers receive higher intensity and experience higher reaction rates.

The monomer conversion, temperature, and density fields are shown in Figure 5.8. The conversion and density strictly follow the intensity field which decays in the z-direction according to the Beer-Lambert law. The parts of the domain that have received a higher dose of light intensity would undergo more reactions to have a higher curing degree. The areas with higher curing degree will feature a higher density. Temperature field, however, does not follow the same pattern as the energy from the high-temperature areas will diffuse to the sections with lower energy. As mentioned before, the right and the bottom boundaries of the domain are assumed to be far so that they are not influenced by the reactions. Looking at the temperature field suggests that both of these boundaries (bottom boundary is not shown in this figure), should be further pushed away to have more accurate results. This task is left for the reader to do.

Figure 5.7: Contours of monomer conversion at  $t=0.4s$



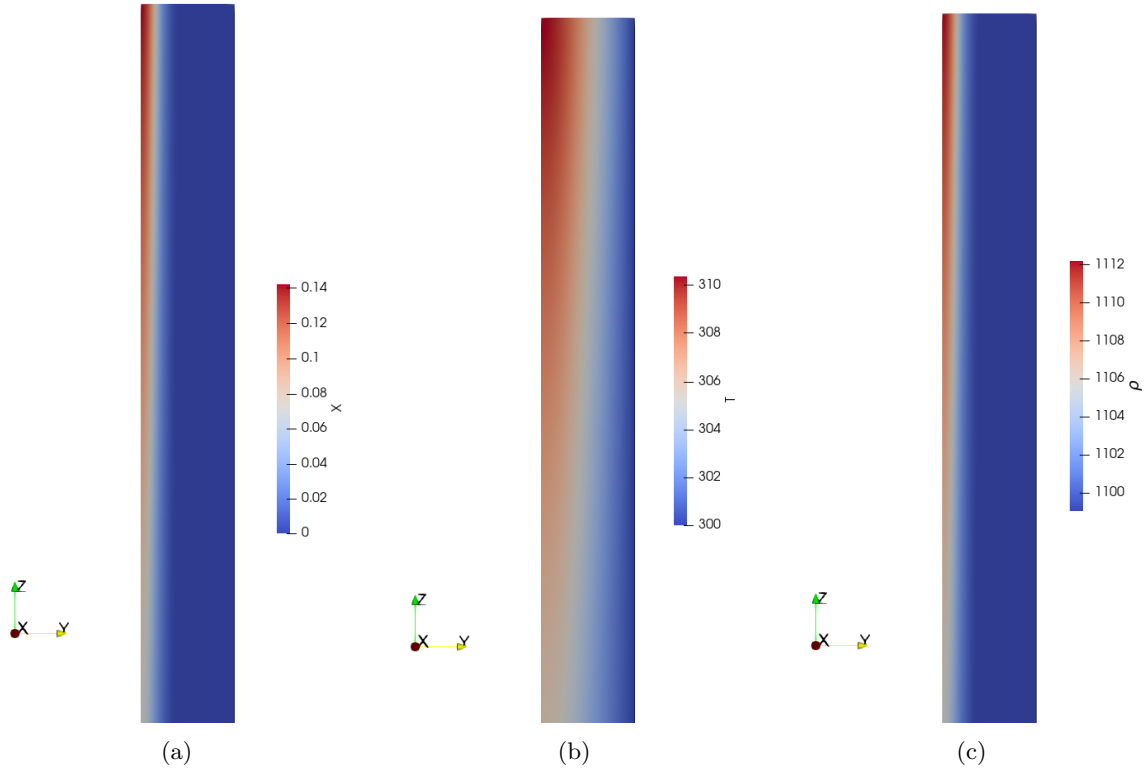


Figure 5.8: Monomer conversion (a), temperature (b), and density (c) fields at  $t=0.4s$

# Bibliography

- [1] Y. Tang, *Stereolithography cure process modeling*. Georgia Institute of Technology, 2005.
- [2] P. F. Jacobs, *Rapid prototyping & manufacturing: fundamentals of stereolithography*. Society of Manufacturing Engineers, 1992.
- [3] D. Hummel, “*Modifying buoyantPimpleFoam for the Simulation of Solid-Liquid Phase Change with Temperature-dependent Thermophysical Properties.*” In Proceedings of CFD with OpenSource Software, 2017, Edited by Nilsson H., [http://dx.doi.org/10.17196/OS\\_CFD#YEAR\\_2017](http://dx.doi.org/10.17196/OS_CFD#YEAR_2017).
- [4] S. M. Mousavi, “*Combination of reactingFoam and chtMultiRegionFoam as a first step toward creating a multiRegionReactingFoam, suitable for solid/gas phase.*” In Proceedings of CFD with OpenSource Software, 2019, Edited by Nilsson H., [http://dx.doi.org/10.17196/OS\\_CFD#YEAR\\_2019](http://dx.doi.org/10.17196/OS_CFD#YEAR_2019).

# Study questions

1. What other ways can be used to read the data form dictionaries?
2. In the libraries defined in this tutorial, **dictionary objects** are instantiated within the solver and passed as arguments to the library. How can these objects be instantiated and used inside the library itself, alternatively?
3. Why are some field variables, such as monomer conversion and temperature, declared as reference variables inside the classes of this tutorial?
4. Why is not the function `calcRho` called inside the `correct` function of the `photoCure` class?
5. Can you explain the reason of the gradual rise in the monomer conversion in Figure 5.5, when the intensity is not active in the probed point?

# Appendix A

## Developed codes

### A.1 reactingCureFoam solver

```
reactingCureFoam.C

/*-----*\
=====
\\      /  F ield      | OpenFOAM: The Open Source CFD Toolbox
\\      /  O peration   |
\\      /  A nd         | www.openfoam.com
\\      /  M anipulation |
-----\

Copyright (C) 2011-2017 OpenFOAM Foundation
Copyright (C) 2020 OpenCFD Ltd.
-----\

License
This file is part of OpenFOAM.

OpenFOAM is free software: you can redistribute it and/or modify it
under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
for more details.

You should have received a copy of the GNU General Public License
along with OpenFOAM. If not, see <http://www.gnu.org/licenses/>.

Application
interFoam

Group
grpMultiphaseSolvers

Description
Solver for two incompressible, isothermal immiscible fluids using a VOF
(volume of fluid) phase-fraction based interface capturing approach,
with optional mesh motion and mesh topology changes including adaptive
re-meshing.

/*-----*/

#include "fvCFD.H"
#include "dynamicFvMesh.H"
#include "CMULES.H"
#include "EulerDdtScheme.H"
```

```

#include "localEulerDdtScheme.H"
#include "CrankNicolsonDdtScheme.H"
#include "turbulentTransportModel.H"
#include "pimpleControl.H"
#include "fvOptions.H"
#include "CorrectPhi.H"
#include "fvcSmooth.H"
#include "cureReaction.H" // Added

// * * * * *

int main(int argc, char *argv[])
{
    argList::addNote
    (
        "Solver for photoPolymerization process"
    );

    #include "postProcess.H"

    #include "addCheckCaseOptions.H"
    #include "setRootCaseLists.H"
    #include "createTime.H"
    #include "createDynamicFvMesh.H"
    #include "initContinuityErrs.H"
    #include "createDyMControls.H"
    #include "createFields.H"
    #include "initCorrectPhi.H"
    #include "createUfIfPresent.H"

    if (!LTS)
    {
        #include "CourantNo.H"
        #include "setInitialDeltaT.H"
    }

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

    while (runTime.run())
    {
        #include "readDyMControls.H"

        if (LTS)
        {
            #include "setRDeltaT.H"
        }
        else
        {
            #include "CourantNo.H"
            #include "setDeltaT.H"
        }

        ++runTime;

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

        //***** Updating intensity*****
        #include "calcI.H"
        //*****

        // --- Pressure-velocity PIMPLE corrector loop
        while (pimple.loop())
        {
            if (pimple.firstIter() || moveMeshOuterCorrectors)
            {
                mesh.update();
            }
        }
    }
}

```

```

        if (mesh.changing())
        {

            gh = (g & mesh.C()) - ghRef;
            ghf = (g & mesh.Cf()) - ghRef;

            MRF.update();

            if (correctPhi)
            {
                // Calculate absolute flux
                // from the mapped surface velocity
                phi = mesh.Sf() & Uf();

                #include "correctPhi.H"

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

                //mixture.correct();
            }

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

        //mixture.correct();

        if (pimple.frozenFlow())
        {
            continue;
        }

        #include "UEqn.H"
        //***** Specie transport and Temperature Eqs*****
        #include "CEqn.H"
        #include "EEqn.H"

        #include "updateR.H"
        //*****

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

    }

    runTime.write();

    runTime.printExecutionTime(Info);
}

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

return 0;
}

// *****

```

## createI.H

```

Info<< "Reading laserSettings file\n" << endl;

IOdictionary laserSettings
(
    IOobject
    (
        "laserSettings",
        runTime.constant(),
        mesh,
        IOobject::MUST_READ_IF_MODIFIED,
        IOobject::NO_WRITE
    )
);

dimensionedScalar Vs("Vs", dimVelocity, laserSettings); //laser scanning velocity
dimensionedScalar lambda("lambda", dimLength, laserSettings); //wavelength
dimensionedScalar P("P", dimPower, laserSettings); // laser power
dimensionedScalar w0("w0", dimLength, laserSettings); //beam radius
dimensionedScalar Dp("Dp", dimLength, laserSettings); // Penetration depth
dimensionedScalar t0("t0", dimTime, laserSettings); // Penetration depth

dimensionedScalar IO = 2*P/(constant::mathematical::pi*w0*w0);
dimensionedScalar t("t", dimTime, runTime.timeOutputValue());

Info<< "Creating field I\n" << endl;
volScalarField I
(
    IOobject
    (
        "I",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,           //MUST_READ, //IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    mesh,
    dimensionedScalar("I", dimMoles/(dimArea*dimTime), Foam::scalar(0)),
    zeroGradientFvPatchScalarField::typeName
);

const volVectorField& C = mesh.C();
const dimensionedScalar conv("conv", dimEnergy*dimLength/dimMoles, 1.196e8); //conversion constant
dimensionedScalar y("y", dimLength, Foam::scalar(0));
dimensionedScalar z("z", dimLength, Foam::scalar(0));
dimensionedScalar r2("r2", dimLength*dimLength, Foam::scalar(0));

forAll(C, counter)
{
    //const scalar& x = C[counter].component(vector::X);
    y.value() = C[counter].component(vector::Y);
    z.value() = C[counter].component(vector::Z);
    r2 = Vs*Vs*(t-t0)*(t-t0) + y*y;
    I[counter] = (IO*Foam::exp(-2*r2/(w0*w0))*Foam::exp(-mag(z)/Dp)*lambda/conv).value();
}

```

## initReact.H

```

Info<< "Reading properties of species from specieProperties file\n" << endl;

IOdictionary specieProperties
(
    IOobject
    (
        "specieProperties",

```

```

        runTime.constant(),
        mesh,
        IOobject::MUST_READ_IF_MODIFIED,
        IOobject::NO_WRITE
    )
);

IOdictionary reactionParameters
(
    IOobject
    (
        "reactionParameters",
        runTime.constant(),
        mesh,
        IOobject::MUST_READ_IF_MODIFIED,
        IOobject::NO_WRITE
    )
);

// ***** Defining the concentrations *****
Info<< "Reading concentrations\n" << endl;

// Photo-initiator Concentration
volScalarField PI
(
    IOobject
    (
        "PI",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);

// Monomer Concentration
volScalarField M
(
    IOobject
    (
        "M",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);

// Chain radical
volScalarField pDot
(
    IOobject
    (
        "pDot",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);

```



```

// Saving the initial concentration of Monomer, MO
volScalarField MO(M);

// Defining conversion
volScalarField X
(
    IOobject
    (
        "X",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    (MO-M)/MO
);

// ***** Defining Reaction rates *****
Info<< "Initializing reaction class\n" << endl;

autoPtr<photoCure> reaction
(
    new cureReaction
    (
        reactionParameters,
        specieProperties,
        T,
        X,
        M
    )
);

Info<< " * * * * * \n";
Info<< "Initializing reaction rates \n" << endl;

const volScalarField& Ki = reaction -> Ki();
const volScalarField& Kp = reaction -> Kp();
const volScalarField& Kt = reaction -> Kt();

const dimensionedScalar& fi = reaction -> fi();

//reaction rates
volScalarField Ri = Ki*PI*I;           // Initiation reaction rate
volScalarField Rp = Kp*pDot*M;         // Propagation reaction rate
volScalarField Rt = Kt*pDot*pDot;       // Termination reaction rate

```

## CEqn.H

```

{
    Info<< "Solving the specie equations\n" << endl;

    // ***** Monomer equation *****
    tmp<fvScalarMatrix> tMEqn
    (
        fvm::ddt(M) //+ fvm::div(phi, U)
        - fvm::laplacian(Dm, M)
        ==
        -Rp
    );
    fvScalarMatrix& MEqn = tMEqn.ref();
    MEqn.solve();

    // ***** Macroradical equation *****
    tmp<fvScalarMatrix> tpDotEqn
    (
        fvm::ddt(pDot) //+ fvm::div(phi, U)

```

```

- fvm::laplacian(Dr, pDot)
==
Ri - Rt
);
fvScalarMatrix& pDotEqn = tpDotEqn.ref();
pDotEqn.solve();

// ***** photo-initiator equation *****
tmp<fvScalarMatrix> tPIEqn
(
    fvm::ddt(PI) //+ fvm::div(phi, U)
    - fvm::laplacian(Ds, PI)
    ==
    -Ri/fi
);
fvScalarMatrix& PIEqn = tPIEqn.ref();
PIEqn.solve();

// **** Updating the conversion degree ****
X = (M0-M)/M0;

// ***** Updating the reaction rates *****
Ri = 2.3*fi*e*PI*I;
Rp = kp0*pDot*M;
Rt = kt0*pDot*pDot;
}

```

## A.2 Constant dictionaries

### laserSettings

```

/*----- C++ -----*\
|=====|
| \ \ / F i e l d | OpenFOAM: The Open Source CFD Toolbox |
| \ \ / O p e r a t i o n | Version: v2112 |
| \ \ / A n d | Website: www.openfoam.com |
| \ \ / M a n i p u l a t i o n |
|=====|
\*-----*/
FoamFile
{
    version      2.0;
    format       ascii;
    class        dictionary;
    object       laserSettings;
}
// ***** //

Vs          0.0272; // m/s, Laser scanning velocity
lambda      325; // nm, wavelength
P           0.0288; // W, laser power
w0          1.1e-4; // m, laser beam radius
Dp          2.483e-3; // Beam penetration depth
t0          0; //
//yc        0; // y-pos of the center of the beam
//zc        0.025; // z-pos of the center of the beam

// ***** //

```

### reactionParameters

```

/*----- C++ -----*\
|=====|
| \ \ / F i e l d | OpenFOAM: The Open Source CFD Toolbox |

```

```

|  \ \  /  O peration   | Version:  v2112                |
|  \ \  /  A nd         | Website:  www.openfoam.com       |
|   \ \ /  M anipulation |                               |
\*-----*/
FoamFile
{
    version      2.0;
    format       ascii;
    class        dictionary;
    object        reactionParameters;
}
// *****

fi          0.6; // quantum efficiency of photo-initiator

e           19.9; // absorptivity of the photo-initiator

heat        2.85e5; // Heat of polymerization J/mol

Ap          6.1;
Aep         28.4;
Ep          1627;
TrefRp      0.0031; // reciprocal of reference temperature
fcRefRp     6;      // reciprocal of reference critical free volume

At          6.4;
Aet         8916;
Et          2103;
Rrd         0.013;
TrefRt      0.0031; // reciprocal of reference temperature
fcRefRt     6;      // reciprocal of reference critical free volume

// *****

```

### specieProperties

```

/*----- C++ -----*/
| ===== |
| \ \  /  F ield       | OpenFOAM: The Open Source CFD Toolbox |
| \ \  /  O peration   | Version:  v2112                |
|   \ \ /  A nd         | Website:  www.openfoam.com       |
|    \ \ /  M anipulation |                               |
\*-----*/
FoamFile
{
    version      2.0;
    format       ascii;
    class        dictionary;
    object        transportProperties;
}
// *****

Monomer
{
    transport
    {
        D      1e-14; //specie's diffusion coefficient
        mu      0.15; // specie's dynamic viscosity
        k       0.142; // specie's thermal conductivity
    }
}

```

```

    }

    thermodynamics
    {
        Tg      205.65; // specie's glass transition temperature
        alpha   0.00177; // specie's coefficient of thermal expansion
        Cp0     218.6; // Cp = Cp0 + Cp1 * T
        Cp1     5.6;
    }

    equationOfState
    {
        rho0    1099; // specie's density
    }
}

Polymer
{
    transport
    {
        mu      0.15;
        k       0.142;
    }

    thermodynamics
    {
        Tg      488.35;
        alpha   0.00012;
        Cp0     -1535.5;
        Cp1     9.1;
    }

    equationOfState
    {
        rho0    1199;
    }
}

Macroradical
{
    transport
    {
        D       1e-16;
    }
}

Photoinitiator
{
    transport
    {
        D       1e-14;
    }
}

// ***** //

```