

# Developing a solver to model the photopolymerization process

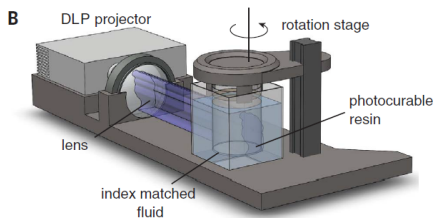
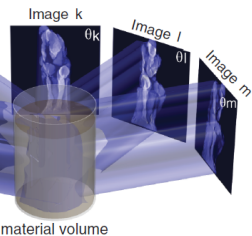
Roozbeh Salajeghe

Manufacturing Section/Construct Department,  
Technical University of Denmark (DTU),  
Kongens Lyngby, Denmark

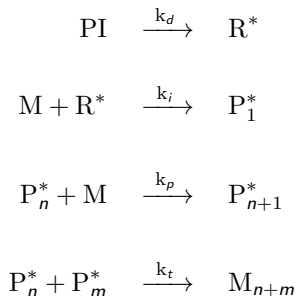
January 19, 2023



Introduction  
●○○○○○○○○○○○○○○○

Kelly et al. <sup>2</sup> [1]

<sup>2</sup>Kelly et al., Volumetric additive manufacturing via tomographic reconstruction







# Theory

The fractional free volume used in the previous equations is defined as below.

$$f = f_M \phi_M + f_p(1 - \phi_M)$$

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

$$f_p = 0.025 + \alpha_p(T - T_{gp})$$

$$\phi_M = \frac{1 - X}{1 - X + \frac{\rho_M}{\rho_P} X}$$

In which  $X$  is the monomer conversion.

$$X = \frac{[M_0] - [M]}{[M_0]}$$



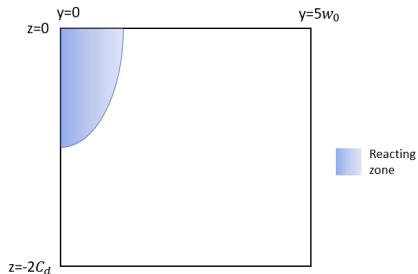
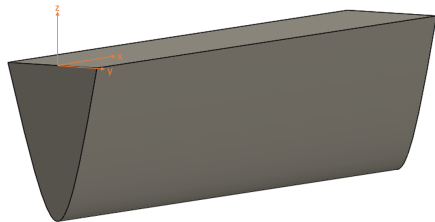






## Simulation example case

SLA simulation in Tang's thesis [3]

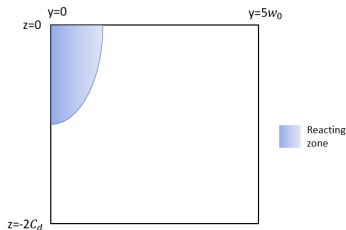


### <sup>3</sup>Tang, STEREOLITHOGRAPHY CURE PROCESS MODELING

# Theory

The intensity field is defined as

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

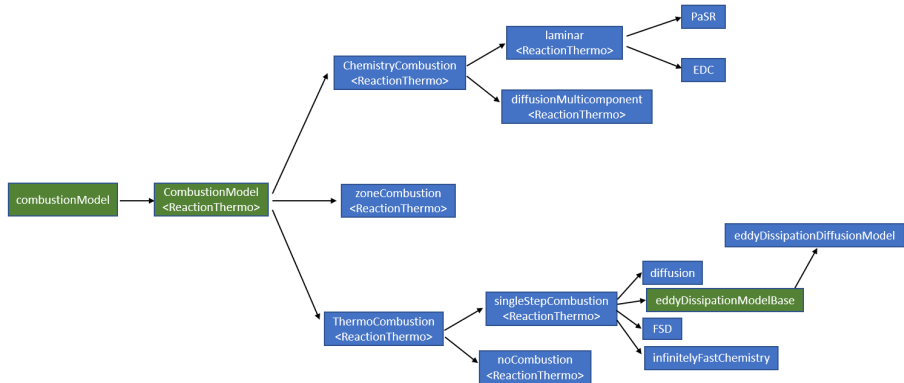


# reactingFoam Solver





# Combustion models in OF





# createFields.H

At the end of the `createFields.H` the heat generation rate is defined.

```

78 volScalarField Qdot
79 (
80     IOobject
81     (
82         "Qdot",
83         runTime.timeName(),
84         mesh,
85         IOobject::READ_IF_PRESENT,
86         IOobject::AUTO_WRITE
87     ),
88     mesh,
89     dimensionedScalar(dimEnergy/dimVolume/dimTime, Zero)
90 );

```



# laminar::correct()

```

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             }
22         }
23     }
24 }

```



# YEqn.H

Then, the value of the reaction heat generation is updated by calling `Qdot()` function on `reaction` object.

14

```
Qdot = reaction->Qdot();
```

The  $\dot{Q}$  function inside the `laminar` combustion model is shown below.

1

• • •

2

```
if (this->active())
```

3

{

4

```
tQdot.ref() = this->chemistryPtr_->Qdot();
```

5

}

6

which simply calls on the `Qdot()` function of the `chemistryPtr_`. This function is defined inside the `standardChemistryModel` class.





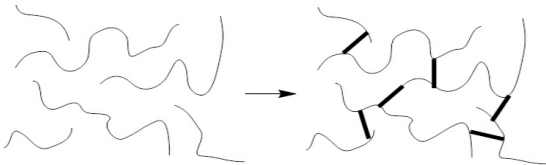
In the above piece of code, the `R()` function of the reaction pointer is called as a source term. This function, which calculates the production/consumption of each species, is shown below for the `laminar` class.

```
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 }
```



reactingFoam is not suitable for photopolymerization modeling!!

It is not feasible to specify an exact value for the molecular weight of the crosslinked molecule as its shape and size are not known.



# New solver and library



## PhotoCure class

This class will be set up from scratch. However, the header of the file should be copied from another existing predefined class.

The folder of the library is made in the user `src` directory.

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

The header is copied from the `viscosityModel` class.

```
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 content of the class should be declared between the following macro lines to prevent the multiple inclusion of the file's contents within other classes.

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

# PhotoCure.H

The first protected member of the class is a dictionary to read the resin properties from a file.

```
1  const dictionary specieProperties_;
```

Then the field variables that will later be initialized by the constructor 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_;
```

# PhotoCure.H

In the next stage, the variables that store the properties of the species are declared. Just the first one is shown here.

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

Then the properties of the mixture are declared as field 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  $\alpha = \kappa / C_p$ 
```

# PhotoCure.H

Afterwards, the public members are declared. The first public member is the constructor, which takes three arguments here.

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

And other member functions are called, one of which is shown here.

```
1 // Member Functions
2 //Calculates the Cp value of the species
3 tmp<volScalarField> calcAndGetCpi
4 (
5     const dimensionedScalar Cp0,
6     const dimensionedScalar Cp1,
7     const volScalarField& T
8 );
```



# PhotoCure.H

Afterwards, the access members are defined. These members are some functions that simply give access to the protected or private data from the outside of the class. One access function that returns the density of the mixture is shown below.

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

For this tutorial, the c++ polymorphism concept will be used. So, the functions that will be used in the derived class should be declared as virtual here. Since no definition is provided here, these functions are called pure virtual, and the class will be an abstract class. One of the pure virtual functions is shown here.

```
1 virtual tmp<volScalarField> Ki() = 0;
```



# PhotoCure.C

photoCure.C contains the definitions of the members that were previously declared in the photoCure.H file.

First, the photoCure.H file should be included.

```
1 #include "photoCure.H"
```

Next, 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    ...
```

# PhotoCure.C

In the constructor, the dictionary object is assigned to the dictionary argument that is passed as an argument. The monomer conversion and the temperature are also initialized with the constructor arguments. Other protected members are initialized by reading the dictionary.

```
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").subDict("transport")),
6  ...
```

# PhotoCure.C

In the constructor, the field variables are initialized by the monomer corresponding values. As an example, density initialization is shown here.

```
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 ),
```

# PhotoCure.C

In the block of the constructor, the `correct` function is called to update the field variables. This function will be defined later.

```
1 {  
2     correct();  
3 }
```

# PhotoCure.C

Next, the other member functions are defined. The first one is a member to calculate the specific heat capacity of each species.

```
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 }
```

# PhotoCure.C

Then, this function is used to evaluate the specific heat capacity of the mixture.

```
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



# PhotoCure.C

Density of the mixture is calculated based on the monomer volume fraction,  $Y_m$ .

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

Finally, the correct function calls the other functions to update the fields.

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

# cureReaction.H

cureReaction class is made in a similar manner as the photoCure class, except that the new class is inherited from the previous one. So, the header file of the previous class should be included at the top of the cureReaction.H file.

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

Then, the inheritance is declared.

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

# cureReaction.H

The protected variables and functions are declared similar to the previous class. At the end of this file, the access functions are defined that will give access to the protected data from outside of the class. Two of the access functions that give the value of the initiation and propagation constants are shown below.

```
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 }
```

# cureReaction.C

At the beginning of the `cureReaction.C` file, the constructor is defined. The constructor receives five arguments, three of which are passed to the constructor of the base class. After initializing a dictionary file, the other variables are initialized similar to the previous case.

```
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     ...
```

# cureReaction.C

The functions that were declared in the `cureReaction.H` file, are defined here. These functions evaluate reaction constants, monomer volume fraction, critical fractional free volume, and fractional free volume. The function that is responsible for the evaluation of the propagation constant is shown as an example here.

```
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 }
```

# cureReaction.C

At the end, the correct function is defined, which calls the other functions of the current class to update the reaction values and the correct function of the previous class to update the properties of the resin.

```
1 void Foam::cureReaction::correct()
2 {
3     photoCure::correct();
4     calcYm();
5     calcF();
6     calcKp();
7     calcKt();
8 }
```

# Compiling the library

To compile the library, the Make directory should be created, which contains the files and options files.

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

In the *files* file, it is specified which files should be compiled, and what the library should be named. So, the following lines should be added to this file.

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

# Compiling the library

In the options file, the libraries that are used and the directories from which some files are included, are specified. This file should read as below.

```
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.



# reactingCureFoam solver

When parts of the resin cure, two different fluid co-exist, which resembles a two-phase flow. In this regard, `interFoam` solver has been chosen as the framework to build the new solver.

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

Since the current solver does not rely on the VOF method and the variable  $\alpha$ , everything that is related to them should be removed from the solver. Refer to the attached files to see which lines should be omitted.

# reactingCureFoam solver

The cureReaction.H file should be included 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 ...
```

Next, the intensity field is created and initialized. For this purpose, a new file with the name createI.H is created.

# reactingCureFoam solver

Inside the createI.H file, a dictionary object is defined that is responsible to read the laser settings from a file inside the constant directory of the case.

```
1  I0dictionary laserSettings
2  (
3      I0object
4      (
5          "laserSettings",
6          runTime.constant(),
7          mesh,
8          I0object::MUST_READ_IF_MODIFIED,
9          I0object::NO_WRITE
10     )
11 );
12
13 dimensionedScalar Vs("Vs", dimVelocity, laserSettings); //laser scanning
14     velocity
15 ...
```

# reactingCureFoam solver

After reading the intensity settings, the intensity field is updated according to the equation that was shown previously.

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

# reactingCureFoam solver

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

# reactingCureFoam solver

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. After creating two dictionaries that are responsible for reading the resin's properties and reaction parameters, the concentration fields are created. Monomer concentration is shown below.

```
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 );
```

# reactingCureFoam solver

Based on the polymorphism concepts of c++, a pointer of type photoCure is created that points to an object of cureReaction class.

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

Using the access functions of the class, the variables for reaction parameters and resin properties are retrieved in other variables to make it easier to use them. One exaple is shown below.

```
1  const volScalarField& Ki = reaction -> Ki()();
```

# reactingCureFoam solver

The two files that are just created are included in the `createFields.H` file after the construction of the 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 //*****
18 #include "createI.H"
19 #include "initReact.H"
```



# reactingCureFoam solver

Since the intensity is a function of the time, it should be updated at every time step. Accordingly, a file with the name `calcI.H` is included in the main file after the time step calculation.

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

in which, the intensity is updated as below.

# reactingCureFoam solver

```
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/
9     conv).value();
10 }
```

Next, the energy and species transport equations should be solved. These equations are solved within the EEqn.H and CEqn.H files that are added after the momentum predictor step, UEqn.H

## reactingCureFoam solver

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

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

At the end of this piece of code, the reaction rates and heat generation are updated in the file with the name `updateR.H` which reads.

```

reaction -> correct();

Ri = Ki*PI*I;
Rp = Kp*pDot*M;
Rt = Kt*pDot*pDot;

Qdot = Rp*heat;

```

1  
2  
3  
4  
5  
6  
7  
8  
9  
10

- 1
- 2
- 3





## Intensity over time

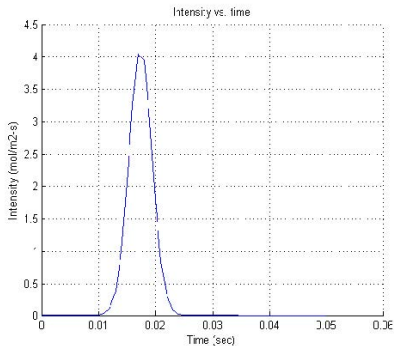
Table: laser parameters

| 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    |

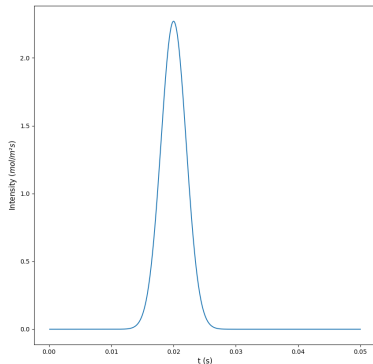
$$I = I_0 \exp \left\{ \frac{-2(V_s(t - t_0))^2 + y^2}{w_0^2} \right\} \exp(-z/D_p) \frac{\lambda(\text{nm})}{1.196 \times 10^8}$$

## Intensity over time

Tang's thesis <sup>3</sup> [3]



## Current solver



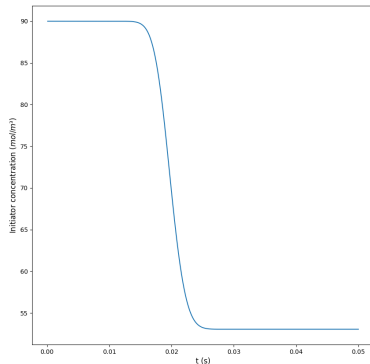
<sup>3</sup>Yanyan Tang, Stereolithography cure process modeling





| 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           |

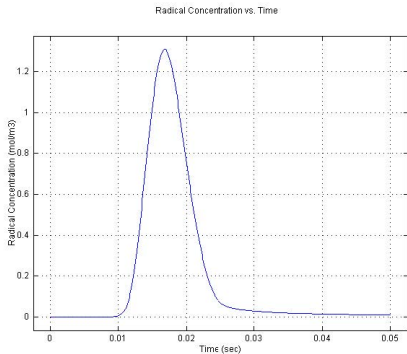
Tang's thesis <sup>3</sup> [3]



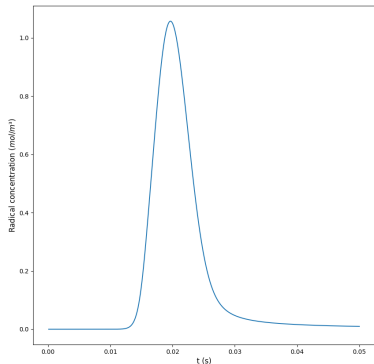
<sup>3</sup>Yanyan Tang, Stereolithography cure process modeling

## Radical concentration variation over time

Tang's thesis <sup>3</sup> [3]



## Current solver

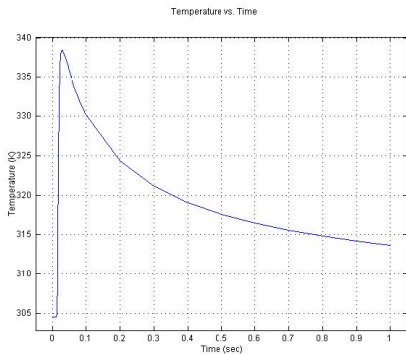


<sup>3</sup>Yanyan Tang, Stereolithography cure process modeling

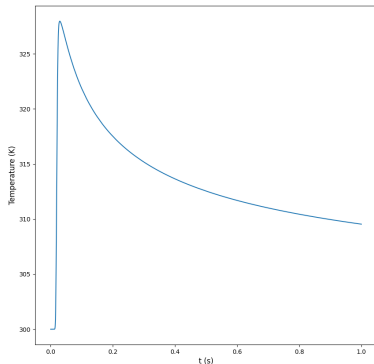


# Temperature over time

Tang's thesis <sup>3</sup> [3]



## Current solver



<sup>3</sup>Yanyan Tang, Stereolithography cure process modeling

## Bibliography



Brett E Kelly, Indrasen Bhattacharya, Hossein Heidari, Maxim Shusteff, Christopher M Spadaccini, and Hayden K Taylor.

Volumetric additive manufacturing via tomographic reconstruction.  
*Science*, 363(6431):1075–1079, 2019.



Marek Pagac, Jiri Hajnys, Quoc-Phu Ma, Lukas Jancar, Jan Jansa, Petr Stefek, and Jakub Mesicek.

A review of vat photopolymerization technology: Materials, applications, challenges, and future trends of 3d printing.  
*Polymers*, 13(4):598, 2021.



Yanyan Tang.  
*Stereolithography cure process modeling.*  
Georgia Institute of Technology, 2005.