CFD with OpenSource software
A course at Chalmers University of Technology
Taught by Håkan Nilsson

Project work:

Implementation of a Temperature Dependent Viscosity Model in OpenFOAM

Developed for OpenFOAM-2.1.0

Author:
Mostafa Payandeh

Peer reviewed by:
Johannes.Palm
Jelena Andric

November, 2012

1 Introduction

The scope of this tutorial is to look at viscosity model in OpenFOAM deeper. In addition, a simple modification will be done on a solver to include temperature as well as on a strain-rate dependent viscosity model class to make a temperature/shear-rate dependent viscosity model.

1.1 What is viscosity?

Viscosity is the ability of fluid to transfer momentum by virtue of diffusion. Dynamic viscosity (**μ**) is a property of fluid and describes the relation between stain rate and stress. Therefore, mathematically it is formulated by the following equation:

$$\tau = \mu \frac{\partial u_i}{\partial x_i}$$

Moreover, fluid can be a Newtonian in which viscosity will be constant or it can be non-Newtonian fluid that is called dependent viscosity. Also kinematic viscosity (**ν**) is calculated by dividing the dynamic viscosity by density of fluid. The rheology behaviour of non-Newtonian fluid in *material processing* is divided into four main categories [1]:

- Thixotropic behaviour (Time dependence): describes the relation of viscosity and time. Non-Newtonian behaviour in form of thixotropic in the fluid mostly related to internal structure of fluid and relaxation time. For example in slurry of metal (molten matrix with solid particle) if the relaxation time increases the agglomeration rate will increase. Therefore, more force will be needed to move the larger particles and viscosity increase.
- Pseudoplastic behaviour (Shear thinning): in which increasing in the shear rate the viscosity dropped. This behaviour break down the particles bridge and particles become more round shape.
- Dilatant behaviour (Shear thickening): in which by increasing in the shear rate the viscosity increased.
- Rheopectic: Viscosity increases with stress over time.

Modelling of viscosity in non-Newtonian fluid depends on what dependency they have. In addition, some criteria come into the model, which can satisfy physical and mathematical criteria. Therefore, a universal model for all non-Newtonian behaviour would not be realistic.

For example, Ostwald de Waele (power law) model can be described simply the dependency of viscosity on shear rate by including shear rate in viscosity definition[2]:

$$\mu = k(\frac{\partial u_i}{\partial x_i})^{n-1}$$

In this model **k** as flow index and shear, exponent **n** are two model parameters. Both **k** and **n** are mostly measured by means of rheometry. The value of **n** can vary between zero and one, be equal to one or become more than one that describe pseudoplastic behaviour, Newtonian behaviour and dilatant behaviour respectively. Value of **k** depends on material stiffness. From

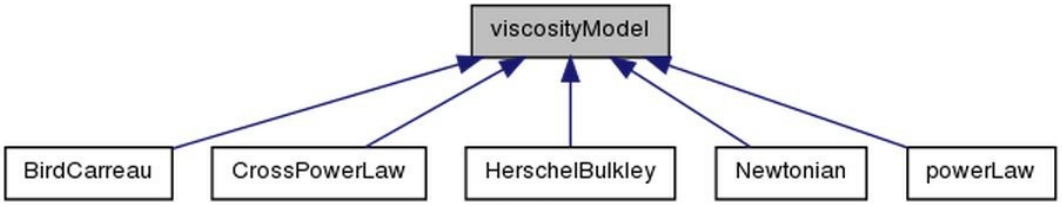
physical and mathematical point of view, it is important that the value of shear rate is constrained between upper and lower limit and also value of **n** stays above zero!!!

OpenFOAM is distributed with a library of transport models that contains different viscosity models (constant or dependent). The type of models implemented in this library are those found in rheology to describe fluids on internal structure, shear rate or temperature with a complex structure, e.g. polymers, food.

2 ViscosityModel class

The **`viscosityModel`**[1] is an abstract base class for incompressible viscosity models. In object-oriented programming, an abstract class has no implementation, only an interface. Thus, it can only be used polymorphically. *Polymorphism* enables us to program in the general way instead of to program in the particular case. Moreover, polymorphism gives the opportunely to write programs that process objects of classes that are part of the same class hierarchy as if they were all objects of the hierarchy's base class [3].

Therefore, the classes that inherited from the **`viscosityModel`** class must be available. In OpenFOAM 2.1, this class has five derived classes as five implanted models. All of these classes describe viscosity as a function of shear-rate except Newtonian model. Then, when **`nu()`** function of each class is called, the viscosity is calculated based on the class. It is also very easy to add the new derived class that describes the viscosity in different manner.



2.1 Strain Rate in OpenFOAM

Generally, T as tensor can separate to symmetric and skew (antisymmetric) tensors. The symmetric part of velocity gradient, **symm(T)** is the rate of deformation and antisymmetric part, **skew(T)** is the velocity tensor. In OpenFOAM, U is the velocity field. Therefore, strain rate can be formulated as:

$$Strain\ Rate = \sqrt{2 * \ symm(gard(U)) : symm(gard(U))}$$

where

$$symm\big(grad(U)\big) = \ grad(U) + \ grad(U).T$$

[1] **For this tutorial, courier new font style is used to distinguish the classes, objects, member functions and member data in OpenFOAM from descriptive parts.**

strainRate() is a member function in **vicosityModel** class (**viscosityModel.C** line 58-61 to return the value of the strain rate when is called from derived classes (e.g. **powerLaw**).

```
00058 Foam::tmp<Foam::volScalarField> Foam::viscosityModel::strainRate() const
00059 {
00060     return sqrt(2.0)*mag(symm(fvc::grad(U_)));
00061 }
```

2.2 powerlaw class in OpenFoam

The **powerLaw** class is derived class from the base class **viscosityModel** class. The **powerLaw** class has the function **nu()** and function **correct()** that return a value for the laminar viscosity and corrects the laminar viscosity respectively.

```
00100         //- Return the laminar viscosity
00101         tmp<volScalarField> nu() const
00102     {
00103             return nu_;
00104         }
00105
00106         //- Correct the laminar viscosity
00107         void correct()
00108         {
00109             nu_ = calcNu();
00110         }
```

By looking at line 109 in **powerLaw.H** file, the value of laminar viscosity is corrected by calling function **calcNu()**. This private member function of **powerLaw** class, calculates viscosity based on power law formulation. The piece of code (line 51 to 69) of **poweLaw.C** file calculates the viscosity by selecting the maximum value of the **nuMin** and minimum value of **nuMax** and power law equation. **VSMALL** is a constant with a very small positive value. Therefore, the **max (VSMALL, other values)** guarantees a nonzero positive value to be used in a function that will fail otherwise.

```
00050 Foam::tmp<Foam::volScalarField>
00051 Foam::viscosityModels::powerLaw::calcNu() const
00052 {
00053     return max
00054     (
00055         nuMin_,
00056         min
00057         (
00058             nuMax_,
00059             k_*pow
00060             (
00061                 max
00062                 (
00063                     dimensionedScalar("one", dimTime, 1.0)*strainRate(),
00064                     dimensionedScalar("VSMALL", dimless, VSMALL)
00065                 ),
00066                 n_.value() - scalar(1.0)
00067             )
00068         )
00069     );
```

Moreover, the member function **read** in **powerLaw** class, reads the constant values **k, n, nuMin** and **nuMax** from the **transportProperties** file.

```
0106 bool Foam::viscosityModels::powerLaw::read
00107 (
00108     const dictionary& viscosityProperties
00109 )
00110 {
00111     viscosityModel::read(viscosityProperties);
00112
00113     powerLawCoeffs_ = viscosityProperties.subDict(typeName + "Coeffs");
00114
00115     powerLawCoeffs_.lookup("k") >> k_;
00116     powerLawCoeffs_.lookup("n") >> n_;
00117     powerLawCoeffs_.lookup("nuMin") >> nuMin_;
00118     powerLawCoeffs_.lookup("nuMax") >> nuMax_;
00119
00120     return true;
```

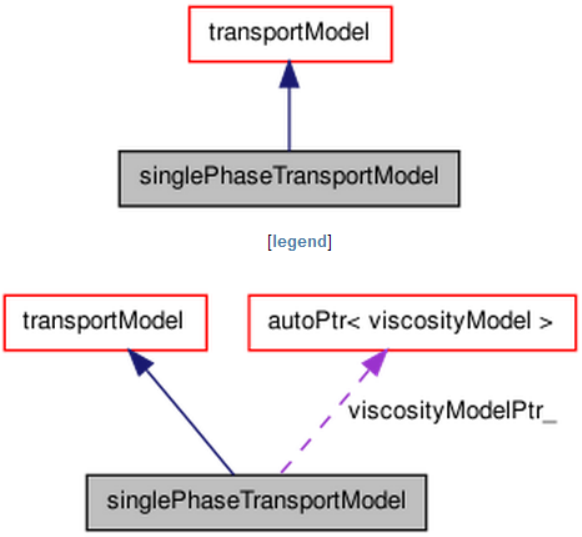
In addition, the **nu** will be written in the file with the same name for each time step.

```
00089     nu_
00090     (
00091         IOobject
00092         (
00093             name,
00094             U_.time().timeName(),
00095             U_.db(),
00096             IOobject::NO_READ,
00097             IOobject::AUTO_WRITE
00098         ),
00099         calcNu()
00100     )
```

3 TransportModel Class

TransportModel class is a base-class for all transport models used by the incompressible turbulence models. The **TransportModel** class has different derived classes to manage multiphase or single phase condition. Public member functions **nu()** return the laminar viscosity, function **correct(),** make a correction on the laminar viscosity and function **read**, read transportProperties dictionary.

Different derived class are available for **transportModel** class e.g. **twoPhaseMixture**, **singlePhaseTransportModel**, etc. For looking deeper how the **transportModel** and **viscosityModel** class collaborate, the **singlePhaseTransportModel** class will be discussed shortly.



3.1 singlePhaseTransportModel class

`singlePhaseTransportModel` class is a derived class simple single-phase transport model based on **`viscosityModel`** class. **`viscosityModelPtr_`** as an object of **`viscosityModel`** class declare in header file of **`singlePhaseTranspot`** class line 61.

```
00061         autoPtr<viscosityModel> viscosityModelPtr_;
```

The member function **`nu()`** is defined in the C file (line 52-55) and it returns the value of the calculated viscosity. By looking at **`nu()`** function, the **`viscosityModelPtr_`** is the private member data which is a pointer. So when we want to call the function **`nu()`**from **`viscosityModel`** class on this object, we use **`->`**.

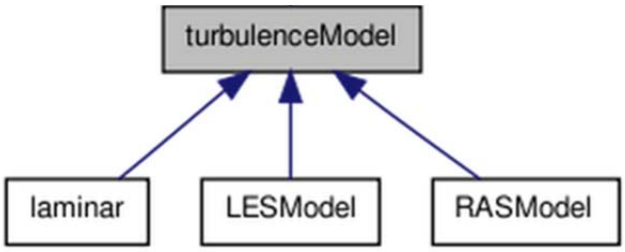
```
00052 Foam::tmp<Foam::volScalarField> Foam::singlePhaseTransportModel::nu() const
00053 {
00054     return viscosityModelPtr_->nu();
00055 }
```

Also, the member function **`correct()`** of this class, corrects the laminar viscosity in the same way.

```
00058 void Foam::singlePhaseTransportModel::correct()
00059 {
00060     viscosityModelPtr_->correct();
00061 }
```

4 turbulenceModel

`turbulanceModel` is the abstract base class for incompressible turbulence models and there are three classes derived from it:



- **`Laminar`** class is a model for laminar flow.
- **`RasModel`** class is a large library of RAS (Reynolds-Averaged Simulation) turbulence model. Also,
- **`LESModel`** class is a library based on Large Eddy Simulation (LES)models.

At header file of **`turbulenceModel`** class (line 164 to174), the access function to incompressible transport model is defined. Class **`turbulanceModel`** has a protected member data **`transportModel`**_, which is an object of the **`transportModel`** class. So function nu acts on **`transportModel`**_ object and return the laminar viscosity nu as a volume scalar field value.

```
00164          //- Access function to incompressible transport model
00165          inline transportModel& transport() const
00166      {
00167              return transportModel_;
00168          }
00169
00170          //- Return the laminar viscosity
00171          inline tmp<volScalarField> nu() const
00172      {
00173              return transportModel_.nu();
00174          }
```

5 Viscosity and Solvers

In this part, the two solvers that use the viscosity library will be described.

5.1 nonNewtonianIcoFoam Solver

The **nonNewtonianIcoFoam** is a standard transient solver in OpenFOAM for incompressible, laminar flow of non-Newtonian fluids, based on PISO method. The difference between this solver and **icoFoam** is the non-Newtonian viscosity of fluid compared to Newtonian viscosity in **icoFoam**. In addition, this solver does not use turbulence model library because of laminar flow condition is assumed.

To use non-Newtonian model, the access of **nonNewtonianIcoFoam** solver to **transportModel** class as a base-class for all transport models such as **viscosityModel** class to calculate viscosity value, is essential. Therefore, **fluid** as an object belong to **singlephaseTransportModel** class is defined in **createField.H** (line 34) for **nonNewtonianIcoFoam** solver to access to all member function and member data of this class.

```
00034     singlePhaseTransportModel fluid(U, phi);
```

By looking at line 57 to 66 of **nonNewtonian.C** file, the member function **correct()**is called first to correct the value of viscosity. Then, member function **nu()** from the selected viscosity class, is called and the momentum part of the equation is built up.

```
00057         fluid.correct();
00058
00059         fvVectorMatrix UEqn
00060         (
00061             fvm::ddt(U)
00062           + fvm::div(phi, U)
00063           - fvm::laplacian(fluid.nu(), U)
00064         );
00065
00066         solve(UEqn == -fvc::grad(p));
```

5.2 SimpleFoam Solver

The **simpleFoam** solver based on SIMPLE algorithm is the OpenFOAM standard solver for incompressible fluid and turbulent flow under steady-state condition. The abbreviation SIMPLE stands for Semi-Implicit Method for Pressure-Linked Equations. This solver contains three header files and one main file. To access to **transpotModel** class to calculate viscosity the **laminarTransport** object of the **singlePhaseTransportModel** class is created in **CreateField.H** file of this solver (line 36).

```
00036     singlePhaseTransportModel laminarTransport(U, phi);
00037
```

In order to access to the **`turbulenceModel`** class, **`turbulence`** as an object of **`RASModel`** class is created .

```
00038     autoPtr<incompressible::RASModel> turbulence
00039     (
00040         incompressible::RASModel::New(U, phi, laminarTransport)
00041     );
```

The momentum corrector inside **`UEqn.H`** file contains the operator **`"=="`** which represents mathematical equality between two sides of an equation. Also, by this operation, the code automatically rearranges the equation in such way that all the implicit terms go into the matrix, and all the explicit terms contribute to the source vector. Turbulence models are treated with the **`turbulence`** pointer by calling the **`divDevReff`** member function in **`laminar`** class, derived from **`RASModel`** class.

```
00003     tmp<fvVectorMatrix> UEqn
00004     (
00005         fvm::div(phi, U)
00006       + turbulence->divDevReff(U)
00007       ==
00008         sources(U)
00009     );
```

In this tutorial, laminar flow is the selected model for using RAS model and turbulence condition is off. **`divDevReff(volVectorField& U)`** as a member function has a volVectorField argument and returns the source term for the momentum equation. (**`laminar.C`**, line 170 to 177)

```
00170 tmp<fvVectorMatrix> laminar::divDevReff(volVectorField& U) const
00171 {
00172     return
00173     (
00174       - fvm::laplacian(nuEff(), U)
00175       - fvc::div(nuEff()*dev(T(fvc::grad(U))))
00176     );
00177 }
```

The **`nuEff()`** as a virtual function returns the effective viscosity and is implanted in **`LESModel`**, **`RASModel`** and **`Laminar`** classes. It is consist of two functions; **`nut()`**as member function in the laminar class to calculate the turbulence viscosity (zero for the laminar flow) and the **`nu()`**as a function in **`viscosityModel`** class to include laminar viscosity. As discussed before, **`transportModel`** and **`turbulenceModel`** has an access function to incompressible transport model to return the laminar viscosity

```
00228         virtual tmp<volScalarField> nuEff() const
00229     {
00230             return tmp<volScalarField>
00231             (
00232                 new volScalarField("nuEff", nut() + nu())
00233             );
00234         }
```

.

6 Temperature-dependent viscosity

The dependency of viscosity has been discussed by introducing different classes. Including temperature into the case, influence the material viscosity during the test. This means that the new class for viscosity is essential. In this section, by using **`powerLaw`** class as well as the **`simpleFoam`** solver, a new temperature viscosity model and a new solver to calculate the temperature will be implemented in the OpenFOAM.

The dependency of viscosity obeys the following equations:

$$\mu = k(\frac{\partial u_i}{\partial x_i})^{n-1}$$

$$k = k_0 - m_k(T - T_0)$$

in which $\boldsymbol{k_0}$ and $\boldsymbol{m_k}$ are initial value of viscosity and temperature dependency coefficient. So, modification of the **`powerLaw`** class into new **`tempdeppowerLaw`** class based on this formulation will be done. For simplicity constant density is assumed. Moreover, the energy equation will be added into the **`simpleFoam`** solver to create a new solver, **`tempsimpleFoam,`** using steady state condition:

$$\mathrm{u}.\nabla T = \nabla.(\alpha \nabla \mathrm{T})$$

in which $\boldsymbol{\alpha}$ is thermal diffusivity to calculate the temperature distribution. Also, assuming constant thermal conductivity and heat capacity, the value of thermal diffusivity will be remain constant.

Note: "blue lines" are command lines and "red lines" must insert to the corresponding file.

6.1 Include T in powerLaw model

Open new terminal window

```
OF21x
run
cp -r $FOAM_SRC/transportModels/incompressible/viscosityModels/powerLaw  tempdeppowerLaw
cd tempdeppowerLaw/
rm powerLaw.dep
mkdir Make
cd Make
cp  $FOAM_SRC/transportModels/incompressible/Make/files files
cp  $FOAM_SRC/transportModels/incompressible/Make/options options
```

Make/file directory (replace)

```
tempdeppowerLaw.C
LIB = $(FOAM_USER_LIBBIN)/libusertempdeppowerLaw
```

Make/options (replace)

```
EXE_INC = \
    -I$(LIB_SRC)/transportModels/incompressible/lnInclude/ \
    -I$(LIB_SRC)/finiteVolume/lnInclude
LIB_LIBS = \
-lfiniteVolume
```

---
powerLaw.C (replace or add)

```
// * * * * * * * * * * * * * Private Member Functions * * * * * * * * * * * * * //

Foam::tmp<Foam::volScalarField>
Foam::viscosityModels::powerLaw::calcNu() const
{
const volScalarField& T= U_.mesh().lookupObject<volScalarField>("T");

        return max
    (
        nuMin_,
        min
        (
            nuMax_,
            (k_-kslope_*(T-Tbase_))*pow
            (
                max
                (
                    dimensionedScalar("one", dimTime, 1.0)*strainRate(),
                    dimensionedScalar("VSMALL", dimless, VSMALL)
                ),
              n_.value() - scalar(1.0)
            )
        )
    );
}

// * * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * * //
Foam::viscosityModels::powerLaw::powerLaw
(
    const word& name,
    const dictionary& viscosityProperties,
    const volVectorField& U,
    const surfaceScalarField& phi
)
:
    viscosityModel(name, viscosityProperties, U, phi),
```

```
    powerLawCoeffs_(viscosityProperties.subDict(typeName + "Coeffs")),
    k_(powerLawCoeffs_.lookup("k")),
    n_(powerLawCoeffs_.lookup("n")),
    kslope_(powerLawCoeffs_.lookup("kslope")),
    Tbase_(powerLawCoeffs_.lookup("Tbase")),
    nuMin_(powerLawCoeffs_.lookup("nuMin")),
    nuMax_(powerLawCoeffs_.lookup("nuMax")),
    nu_
    (
        IOobject
        (
            name,
            U_.time().timeName(),
            U_.db(),
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),
        calcNu()
    )



// * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * * //

bool Foam::viscosityModels::powerLaw::read
(
    const dictionary& viscosityProperties
)
{
    viscosityModel::read(viscosityProperties);

    powerLawCoeffs_ = viscosityProperties.subDict(typeName + "Coeffs");

    powerLawCoeffs_.lookup("k") >> k_;
    powerLawCoeffs_.lookup("n") >> n_;
    powerLawCoeffs_.lookup("kslope") >> kslope_;
    powerLawCoeffs_.lookup("Tbase") >> Tbase_;
    powerLawCoeffs_.lookup("nuMin") >> nuMin_;
    powerLawCoeffs_.lookup("nuMax") >> nuMax_;

    return true;
}
```

---
powerLaw.H (add)

```
/*---------------------------------------------------------------------------*\
                        Class powerLaw Declaration
\*---------------------------------------------------------------------------*/
class powerLaw
:
    public viscosityModel
{
    // Private data

        dictionary powerLawCoeffs_;

        dimensionedScalar k_;
        dimensionedScalar n_;
        dimensionedScalar kslope_;
        dimensionedScalar Tbase_;
        dimensionedScalar nuMin_;
        dimensionedScalar nuMax_;

        volScalarField nu_;
```

```
cd ..
mv powerLaw.H tempdeppowerLaw.H
mv powerLaw.C tempdeppowerLaw.C
sed -i s/powerLaw/tempdeppowerLaw/g tempdeppowerLaw.C
sed -i s/powerLaw/tempdeppowerLaw/g tempdeppowerLaw.H
wmake libso
```

6.2 Include temperature in simpleFoam

```
run
cp -r $FOAM_APP/solvers/incompressible/simpleFoam temperatureSimpleFoam
cd temperatureSimpleFoam/
wclean
rm -r SRFSimpleFoam/
rm -r porousSimpleFoam/
rm -r MRFSimpleFoam/
mv simpleFoam.C tempSimpleFoam.C
cp UEqn.H TEqn.H
```

Add or replace following red lines to corresponding file:

---
tempSimpleFoam.C (add)

```
// --- Pressure-velocity SIMPLE corrector
        {
            #include "UEqn.H"
            #include "pEqn.H"
            #include "TEqn.H"
        }
```

TEqn.H (replace)

```
tmp<fvScalarMatrix> TEqn
(
    fvm::div(phi, T)
  - fvm::laplacian(TempD_, T)    );

TEqn().relax();
solve(TEqn());
```

---
createField.H (add)

```
Info<< "Reading field T\n" << endl;
volScalarField T
(
    IOobject
    (
        "T",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);

#include "createPhi.H"

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

singlePhaseTransportModel laminarTransport(U, phi);

// diffusivity [TempD]
dimensionedScalar TempD_(laminarTransport.lookup("TempD"));
```

Make/file (replace)
---

tempSimpleFoam.C

EXE = $(FOAM_USER_APPBIN)/tempSimpleFoam

Make/options (add)

EXE_LIBS = \
-L$(FOAM_USER_LIBBIN) \

compile by command

wmake

Now we can run a case by using tempSimpleFoam solver and temdeppowerLaw model for viscosity calculation.

7 Run a Case

First option: Download the case from course homepage and run the case by using tempSimpleFoam –noFunctionObjects command

Second option is to follow tutorial:

```
run
cp -r $FOAM_TUTORIALS/incompressible/simpleFoam/pitzDaily Extrutionmouldcase
cd Extrutionmouldcase
```

---
constant/ polyMesh/ blockMeshDict

```
convertToMeters 1;

vertices
(
    (0 0 -0.05)
    (0.5 0 -0.05)
    (1 0 -0.05)
    (0 1 -0.05)
    (0.5 1 -0.05)
    (1 1 -0.05)
    (0 3 -0.05)
    (0.5 3 -0.05)
    (3 3 -0.05)
    (0 4 -0.05)
    (0.5 4 -0.05)
    (3 4 -0.05)
    (0 0 0.05)
    (0.5 0 0.05)
    (1 0 0.05)
    (0 1 0.05)
    (0.5 1 0.05)
    (1 1 0.05)
    (0 3 0.05)
    (0.5 3 0.05)
    (3 3 0.05)
    (0 4 0.05)
    (0.5 4 0.05)
    (3 4 0.05)

);

blocks
(
    hex (0 1 4 3 12 13 16 15) (10 20 1) simpleGrading (1 1 1)
    hex (1 2 5 4 13 14 17 16) (10 20 1) simpleGrading (1 1 1)
    hex (3 4 7 6 15 16 19 18) (10 40 1) simpleGrading (1 1 1)
    hex (6 7 10 9 18 19 22 21) (10 20 1) simpleGrading (1 1 1)
    hex (7 8 11 10 19 20 23 22) (60 20 1) simpleGrading (1 1 1)
);

edges
(
);

boundary
(
    inlet
    {
        type patch;
        faces
        (
            (0 1 13 12)
            (1 2 14 13)
        );
    }
    outlet
    {
        type patch;
        faces
        (
            (8 11 23 20)
        );
    }
    Right
    {
        type symmetryPlane;
        faces
        (
            (0 12 15 3)
            (3 15 18 6)
            (6 18 21 9)
        );
    }
    wall
    {
        type patch;
        faces
        (
            (2 5 17 14)
            (5 4 16 17)
            (7 19 16 4)
            (10 22 23 11)
            (7 8 20 19)
            (10 9 21 22)
        );
    }
    frontAndBack
    {
        type empty;
        faces
        (
            (0 3 4 1)
            (4 5 2 1)
            (4 3 6 7)
            (7 6 9 10)
            (10 11 8 7)
            (12 13 16 15)
            (16 13 14 17)
            (15 16 19 18)
            (21 18 19 22)
            (22 19 20 23)
        );
    }
);

mergePatchPairs
(
);
```

Use blockMesh command to create the mesh.

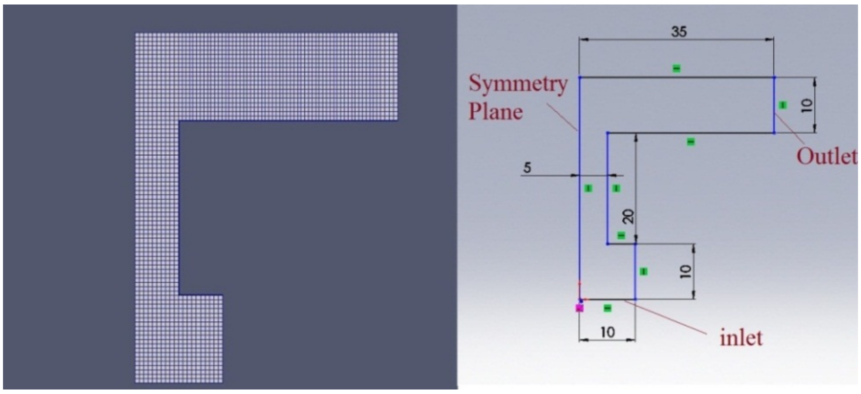


Figure 1- The geometry and dimensions of mould

To change the boundary conditions, type the following commands:

```
cd 0
cp U T
rm k nut epsilon nuTilda
```

and change the corresponding files as:

p	T	U

dimensions [0 2 -2 0 0 0 0];
 	
internalField uniform 0;
 	
boundaryField
{
inlet
{
type zeroGradient;
}
 	
outlet
{
type fixedValue;
value uniform 0;
}
 	
Right
{
type symmetryPlane;
}
 	
wall
{
type zeroGradient;
}
frontAndBack
{
type empty;
}
}
	FoamFile
	{
	version 2.0;
	format ascii;
	class volScalarField;
	object T;
	}
	// * //
	
	dimensions [0 0 0 1 0 0 0];
	
	internalField uniform 400;
	
	boundaryField
	{
	inlet
	{
	type fixedValue;
	value uniform 400;
	}
	
	outlet
	{
	type zeroGradient;
	}
	Right
	{
	type symmetryPlane;
	}
	
	wall
	{
	type fixedValue;
	value uniform 200;
	}
	frontAndBack
	{
	type empty;
	}
	}
		dimensions [0 1 -1 0 0 0 0];
		
		internalField uniform (0 0 0);
		
		boundaryField
		{
		inlet
		{
		type fixedValue;
		value uniform (0 0.000001 0);
		}
		
		outlet
		{
		type zeroGradient;
		}
		Right
		{
		type symmetryPlane;
		}
		
		wall
		{
		type fixedValue;
		value uniform (0 0 0);
		}
		frontAndBack
		{
		type empty;
		}
		}

---
constant/RASPropertie(modify).

```
RASModel        laminar;
turbulence      off;
printCoeffs     off;
```

---
constant/transportProperties (add)

```
transportModel  tempdeppowerLaw;

nu              nu [ 0 2 -1 0 0 0 0 ] 1;


tempdeppowerLawCoeffs
{
    k           k [0 2 -1 0 0 0 0] 2800;
    n           n [0 0 0 0 0 0 0] 0.64;
    kslope      kslope [0 2 -1 -1 0 0 0] 0.5;
    Tbase       Tbase [0 0 0 1 0 0 0] 300;
    nuMin       nuMin [0 2 -1 0 0 0 0] .1;
    nuMax       nuMax [0 2 -1 0 0 0 0] 10000000;
  }

TempD       TempD  [0 2 -1 0 0 0 0] 1e-8;
```

Note: kslope dimension must be $m^2/s.K$ in order to be dimension consistent.
TempD is thermal diffusivity.

---
system /controlDict (*modify*).

```
application     tempSimpleFoam;
writeInterval   2;
```

and finally add in the end of the file

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

---
system/ fvSchemes

```
in divSchemes

   div(phi,T) Gauss linear ;


in laplacianSchemes

    laplacian(TempD,T) Gauss linear corrected;
```

---
system/fvSolution (add)

```
T
   {
      solver          PBiCG;
      preconditioner  DILU;
      tolerance       1e-05;
      relTol          0.1;
   }
```

Also the relaxation factor for T and the residual control must be set to e-3 and 0.7 respectively.

Add the following line for *residualControl*

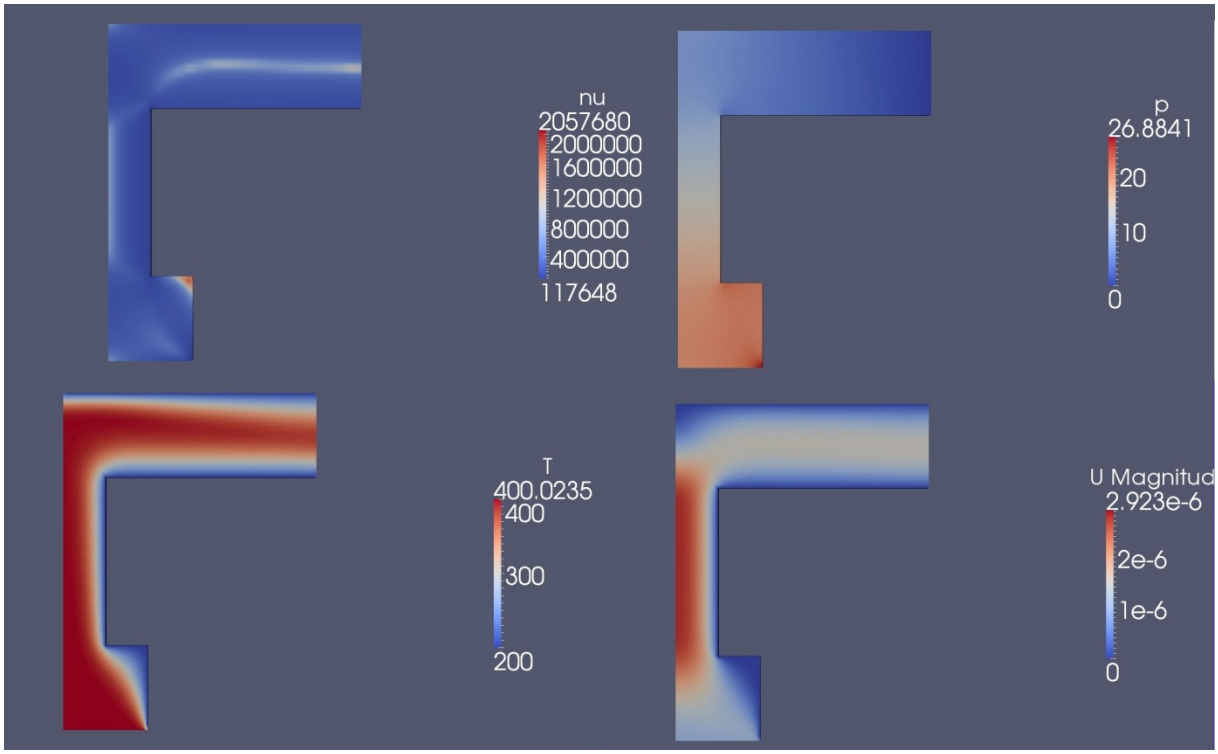
```
T               1e-2;
```

and for the *equation* in *relaxationFactors*

```
T               0.7;
```

```
cd ..
tempSimpleFoam -noFunctionObjects
```

Use paraFoam to check the result.



References

1.Fan, Z., *Semisolid metal processing.* International Materials Reviews, 2002. **47**(2): p. 49-85.
2.Atkinson, H.V., *Modelling the semisolid processing of metallic alloys.* Progress in Materials Science, 2005. **50**(3): p. 341-412.
3.Deitel, P. and H. Deitel, *C++ How to Program*2011: Prentice Hall.