Cite as: Shang, Y.: A Two-equation SGS model tutorial. Proceedings of CFD with OpenSource Software, 2017, Edited by Nilsson. H., http://dx.doi.org/10.17196/OS_CFD#YEAR_2017

CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

A Two-equation SGS model tutorial

Developed for OpenFOAM-v1706+

Author:
Yeru Shang
University of Sussex
yeru.shang@sussex.ac.uk
yeru_shang@hotmail.com

Peer reviewed by:
Luofeng Huang
Mohammad Arabnejad

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.

December 20, 2017

Learning outcomes

The reader will learn:

How to use it:

- How to use the Two-equation Sub-grid Scale (SGS) model for Large Eddy Simulation (LES).

The theory of it:

- The theory of the Eddy Viscosity turbulence model.

How it is implemented:

- How to implement the new Two-equation SGS model.

How to modify it:

- How to modify the implemented Two-equation SGS model.

Prerequisites

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

- An Introduction to Computational Fluid Dynamics, The Finite Volume Method, Book by H. K. Versteeg and W. Malalasekera;
- Basic theory of turbulence model;
- Have basic knowledge of running pitzDaily tutorial case.

Contents

Chapter 1

Introduction

The main aim of this tutorial is to describe how to implement a new Two-equation Sub-grid Scale (SGS) model. The new model is claimed by its developer to be self-adaptive and be able to model turbulence on any arbitrary mesh density, i.e. from Reynolds-averaged Navier-Stokes (RANS) to Large Eddy Simulation (LES) and even to Direct Numerical Simulation (DNS) [3]. The tutorial also demonstrates how to implement wall damping function as a warm-up exercise.

The main content is listed below:

1. This tutorial starts with a theoretical background of turbulence modelling, including different methods of turbulence modelling, the commonly used eddy viscosity model concept and the Two-equations SGS model being implemented. A tour of the templated turbulence model library in OpenFOAM 1706 will also be given with focus on most related existing models in order for the reader to familiarise themselves with these codes.

2. The Chapter 3 will be dedicated to show how exactly the damping function and the new model are being implemented by a step by step guide.

3. The Chapter 4 will prepare the pitzDaily test case to show the result of this implemented wall damping function and the new model by using ParaView.

4. The tutorial will end with conclusion and further development.

Chapter 2

Background

2.1 Turbulence Modelling

Computational Fluid Dynamics (CFD) has a indispensable role in engineering application and research activities. It can simulate real flow conditions, heat transfers and other phenomena with a reasonable accuracy but much lower cost than conducting an experiment. However CFD still has many bottlenecks, one among which is its capability to model turbulent flow, the most common flow type in the real engineering situation.

Turbulence, the three-dimensional, random and complex state of a fluid with wide range of length scales, is one of the most challenging problems in fluid dynamics, yet having great significance in practical engineering applications. Consequently, numerous scientists have invested a great deal of effort in the observation, description and understanding of turbulent flows. It was found out that by applying the conservation of mass, momentum and energy, governing equation of fluid flow can be derived. If using Newtonian model for viscous stresses, the governing equation will lead to Navier-Stokes Equations (NSEs) [4]. If NSEs is solved on a spatial grid that is fine enough to solve the Kolmogorov length scale with time step sizes that sufficiently small to resolve the fastest fluctuation, all flow characteristics can be captured, including mean flow and turbulence. This method is known as Direct Numerical Simulation (DNS). But the computational cost of DNS is prohibitably high and is not used in real engineering applications.

Due to the limitation of computing resources, the attention of early CFD research was on the mean flow and modeling the effect of turbulence on the mean flow. This lead to a method called Reynolds-averaged Navier-Stokes (RANS). This method conducts a time or ensemble average on NSEs and the extra term created due the averaging process is modelled by so-called RANS turbulence models. RANS has been widely applied in industrial flow computations due to its modest computing resources requirement and reasonable accuracy. However, it is widely recognised that RANS fails to provide satisfactory accuracy in flow with separation, reattachment and noise, etc. It is mainly due to RANS represents all turbulent energy by modelling, whereas turbulence plays an dominant role in such flow conditions. Here comes the Large Eddy Simulation (LES), it can offer a solution to balance the high computational cost of DNS and the low accuracy of RANS. LES spatially filters the NSEs and directly resolves the governing equations for large eddies (larger than filter size[1]) and their turbulent energy, while it uses a Sub-grid Scale (SGS) model to simulate the effect of small eddies (smaller than filter size) on the mean flow and large eddies. Computing resource requirement of LES is greater than RANS but is much less than DNS. Thanks to the rapid increase of computing power, LES has started to be applied on complex geometries.

Recently, hybrid turbulence models which combine RANS and LES characteristics in various way have attracted lots of research attentions. Detached Eddy Simulation (DES) is one of the examples. DES uses RANS formula to solve the flow field close to the wall whereas use LES to solve large (detached) eddies away from the wall. Some other hybrid models solve eddy viscosity

[1] For implicit LES, filter size is mesh size.

by both RANS and LES and blend them according to certain parameters, normally being mesh size. Scale-adaptive Simulation (SAS) model, such as k-ω SST SAS model invited by Menter and co-workers [1], adds an extra production term in the ω equation which will increase the production of ω when detecting the unsteadiness, thereby decrease the turbulence viscosity in order to provide RANS with LES content in unsteady regions without any blend factor.

The Two-equation SGS model to be implemented in the tutorial is also claimed to be self adapting. But unlike k-ω SST SAS model which is based on RANS model and resolve more turbulence if mesh resolution allows to do so, this model is based on SGS model and can resolve as much energy as possible for any mesh resolution, i.e. it changes characters across RANS, LES and even DNS depending on flow situation and mesh density [3].

2.2 Eddy Viscosity Model

By far, most of the turbulence models implemented in OpenFOAM are based on eddy viscosity theory, such as Spalart-Allmaras, k-ϵ series and k-ω series RANS models, and Smagorinsky and k-equation SGS model for LES, including the one being implemented. Therefore, it is crucial to understand eddy viscosity theory before starting any implementation or modification of this kind of turbulence models.

By applying a spatial filter of uniform width, Δ, on incompressible, Newtonian flows with constant thermo-physical properties, one can obtain the governing equations for the Large Eddy Simulation of such a flow as

$$\nabla \cdot \bar{\boldsymbol{U}} = 0 \tag{2.1a}$$

$$\frac{\partial \bar{\boldsymbol{U}}}{\partial t} + \nabla \cdot (\bar{\boldsymbol{U}}\bar{\boldsymbol{U}}) = -\nabla \bar{p} + \nabla \cdot \nu(\nabla \bar{\boldsymbol{U}} + \nabla \bar{\boldsymbol{U}}^T) - \nabla \cdot \tau \tag{2.1b}$$

Where overbar indicates spatial filtering process, $\bar{U}$ is filtered instantaneous velocity (time-averaged velocity in RANS), $\overline{p}$ is filtered instantaneous pressure (time-averaged velocity in RANS) divided by the constant density, ν is kinematic viscosity, τ is the SGS stress tensor (Reynolds stress in RANS) which has to be modelled to close the system.

It is well known that in the Newton's law of viscosity for incompressible flow

$$\tau_{Newtonian} = 2\nu S = \nu(\nabla \boldsymbol{U} + \nabla \boldsymbol{U}^T) \tag{2.2}$$

where S is the rate of deformation of fluid elements. It has been found that the turbulent stresses increase as the mean rate of deformation increases. Boussinesq hypothesis proposed that the Reynolds stress in RANS is proportional to the mean rates of deformation. In SGS model, this theory is interpreted as SGS stresses are proportional to the instantaneous rates of deformation, i.e.

$$\tau = -2\nu_t \overline{S} + \frac{1}{3} tr(\tau)\boldsymbol{I} = -\nu_t(\nabla \overline{\boldsymbol{U}} + \nabla \overline{\boldsymbol{U}}^T) + \frac{1}{3} tr(\tau)\boldsymbol{I} \tag{2.3}$$

where ν_t is the SGS eddy viscosity (or turbulence viscosity in RANS) and I is Kronecker Delta.

On dimensional grounds, it is assumable that ν_t can be expressed as a product of a SGS velocity scale, ϑ, and a SGS length scale, ℓ, as

$$\nu_t = C\vartheta\ell \tag{2.4}$$

where C is a dimensionless constant.

Therefore, the turbulence model based on Eddy Viscosity theory is to find appropriate equations for ϑ and ℓ by either algebraic relation or transport equations, and to use them to obtain ν_t in order to close the filtered NSEs.

2.3 Inagi Wall Damping Function

The wall damping function which will be used as a warm-up exercise is proposed by Inagi et al. [2]. It is only applicable to SGS models that contain k-equation. As can be seen in later section that the standard k-equation SGS model evaluates kinematic eddy viscosity via

$$\nu_t = C_k \Delta \sqrt{k_{sgs}} \tag{2.5}$$

while wall damping function introduce a parameter F_{wY} into the equation above, i.e.

$$\nu_t = F_{wY} C_k \Delta \sqrt{k_{sgs}} \tag{2.6}$$

where

$$F_{wY} = \frac{1}{1 + \Delta \sqrt{2|S_{ij}|^2}/C_T \sqrt{k_{sgs}}} \tag{2.7}$$

and $C_T = 10.0$.

2.4 Two-equation SGS model

The new Two-equation SGS model proposed by Perot and Gadebusch [3] reads

$$\frac{\partial k_{sgs}}{\partial t} + \nabla \cdot (k_{sgs} \bar{\boldsymbol{U}}) = \nabla \cdot \left[\left(\nu + \frac{\nu_t}{\sigma_k} \right) \nabla k_{sgs} \right] + \alpha P - \varepsilon_{sgs} \tag{2.8a}$$

$$\frac{\partial \varepsilon_{sgs}}{\partial t} + \nabla \cdot (\varepsilon_{sgs} \bar{\boldsymbol{U}}) = \nabla \cdot \left[\left(\nu + \frac{\nu_t}{\sigma_\varepsilon} \right) \nabla \varepsilon_{sgs} \right] + \frac{\varepsilon_{sgs}}{k_{sgs}} \left[C_{\varepsilon 1} P - C_{\varepsilon 2} \varepsilon_{sgs} \right] \tag{2.8b}$$

where k_{sgs} is the SGS turbulent kinetic energy, ε_{sgs} is the SGS turbulent kinetic energy dissipation rate, $P = \nu_t (\nabla \overline{\boldsymbol{U}} + \nabla \overline{\boldsymbol{U}}^T) \nabla \overline{\boldsymbol{U}}$ is the production of SGS turbulent kinetic energy. The parameter $C_{\varepsilon 2} = (11/6) f + (25/Re_T) f^2$, where $Re_T = k_{sgs}^2 / \nu \varepsilon_{sgs}$ is the local turbulent Reynolds number and function $f = (Re_T/30[\sqrt{1 + 60/Re_T} - 1]$. The constants in this model are

$$C_{\varepsilon 1} = 1.55; \quad \sigma_k = 1.0; \quad \sigma_\varepsilon = 1.2 \tag{2.9}$$

The energy transfer (backscatter) variable, α, is

$$\alpha = 1.5 \left\{ 1 - C^* \left(\frac{k_{sgs}}{k_{sgs} + k_r} \right)^2 \left[\left(\frac{\Delta x_i}{\sqrt{k_r}} \frac{\partial \sqrt{k_r}}{\partial x_i} \right)^2 + 0.11 \right]^{-1} \right\} \tag{2.10}$$

Where the empirically determined constant $C^* = 0.28$, and

$$\frac{\left(\Delta x_i \frac{\partial \sqrt{k_r}}{\partial x_i} \right)^2}{k_r} = \frac{\left\{ \left(\Delta x \frac{\partial \sqrt{k_r}}{\partial x} \right)^2 + \left(\Delta y \frac{\partial \sqrt{k_r}}{\partial y} \right)^2 + \left(\Delta z \frac{\partial \sqrt{k_r}}{\partial z} \right)^2 \right\}}{k_r} \tag{2.11}$$

is a dimensionless measure of the gradient of resolved turbulent kinetic energy.

The eddy viscosity is then evaluated as

$$\nu_t = C_\mu \frac{k_{sgs}^2}{\varepsilon_{sgs}} \left(\frac{k_{sgs}}{k_{sgs} + k_r} \right) \tag{2.12}$$

where $C_\mu = 0.18$ and $k_r = 0.5 \times U'^2$ is the resolved turbulent kinetic energy.
Finally, the SGS stresses are constructed as

$$\tau = -\alpha \nu_t (\nabla \overline{\boldsymbol{U}} + \nabla \overline{\boldsymbol{U}}^T) + \frac{2}{3} k_{sgs} \boldsymbol{I} \tag{2.13}$$

2.5 Turbulence Model Library

Turbulence model library in OpenFOAM 1706 is a templated library, located at `$FOAM_SRC/TurbulenceModels`. The templated Turbulence model class contains many sub-classes. How is the specific turbulence model selected during the run time is well documented in the lecture of Prof. Nilsson [2]. This section will take reader to a tour from the perspective of specific turbulence model.

A good practice of implementing new piece of code in OpenFOAM is to firstly find class with a similar function and then modify the functionality based on it. The model being implemented is a SGS model but belongs to k-ε series. A detailed look at the available SGS model at `$FOAM_SRC/TurbulenceModels/turbulenceModels/LES` shows the there is no k-ε based SGS model, the directory contains only the k-equation SGS model. Whereas k-ε series model only appears in `$FOAM_SRC/TurbulenceModels/turbulenceModels/RAS` which contains RANS models. Therefore, current model will be a combination of k-equation SGS model and k-ε RANS model.

2.5.1 The Standard k-ε Model in OpenFOAM 1706

The standard k-ε RANS model implemented in OpenFOAM 1706 reads

$$\frac{\partial k}{\partial t} + \nabla \cdot (k\bar{\boldsymbol{U}}) = \nabla \cdot \left[\left(\nu + \frac{\nu_t}{\sigma_k} \right) \nabla k \right] + P - \varepsilon \tag{2.14a}$$

$$\frac{\partial \varepsilon}{\partial t} + \nabla \cdot (\varepsilon\bar{\boldsymbol{U}}) = \nabla \cdot \left[\left(\nu + \frac{\nu_t}{\sigma_\varepsilon} \right) \nabla \varepsilon \right] + \frac{\varepsilon}{k} \left[C_{\varepsilon 1} P - C_{\varepsilon 2} \varepsilon \right] \tag{2.14b}$$

$$\nu_t = C_\mu \frac{\sqrt{k}}{\varepsilon} \tag{2.14c}$$

where k is the turbulent kinetic energy, ε is its dissipation rate. The constants of this model are

$$C_\mu = 0.09; \quad C_{\varepsilon 1} = 1.44; \quad C_{\varepsilon 2} = 1.92; \quad \sigma_k = 1.0; \quad \sigma_\varepsilon = 1.3 \tag{2.15}$$

This model is implemented in the directory `$FOAM_SRC/TurbulenceModels/turbulenceModels/RAS/kEpsilon` by two files, i.e. `kEpsilon.C` and `kEpsilon.H`. The H file declares the class, any member data and functions, and the C file contains the detailed implementation of the model as (line 254 to line 295)

```
// Dissipation equation
tmp<fvScalarMatrix> epsEqn
(
    fvm::ddt(alpha, rho, epsilon_)
  + fvm::div(alphaRhoPhi, epsilon_)
  - fvm::laplacian(alpha*rho*DepsilonEff(), epsilon_)
 ==
    C1_*alpha()*rho()*G*epsilon_()/k_()
  - fvm::SuSp(((2.0/3.0)*C1_ - C3_)*alpha()*rho()*divU, epsilon_)
  - fvm::Sp(C2_*alpha()*rho()*epsilon_()/k_(), epsilon_)
  + epsilonSource()
  + fvOptions(alpha, rho, epsilon_)
);

epsEqn.ref().relax();
fvOptions.constrain(epsEqn.ref());
epsEqn.ref().boundaryManipulate(epsilon_.boundaryFieldRef());
solve(epsEqn);
```

[2] Link: https://pingpong.chalmers.se/public/courseId/8331/lang-en/publicPage.do?item=3855255

```
fvOptions.correct(epsilon_);
bound(epsilon_, this->epsilonMin_);

// Turbulent kinetic energy equation
tmp<fvScalarMatrix> kEqn
(
    fvm::ddt(alpha, rho, k_)
  + fvm::div(alphaRhoPhi, k_)
  - fvm::laplacian(alpha*rho*DkEff(), k_)
 ==
    alpha()*rho()*G
  - fvm::SuSp((2.0/3.0)*alpha()*rho()*divU, k_)
  - fvm::Sp(alpha()*rho()*epsilon_()/k_(), k_)
  + kSource()
  + fvOptions(alpha, rho, k_)
);

kEqn.ref().relax();
fvOptions.constrain(kEqn.ref());
solve(kEqn);
fvOptions.correct(k_);
bound(k_, this->kMin_);

correctNut();
```

And the effective diffusivity for k and ε are calculated in H file as (line 161 to line 185)

```
    //- Return the effective diffusivity for k
    tmp<volScalarField> DkEff() const
    {
        return tmp<volScalarField>
        (
            new volScalarField
            (
                "DkEff",
                (this->nut_/sigmak_ + this->nu())
            )
        );
    }

    //- Return the effective diffusivity for epsilon
    tmp<volScalarField> DepsilonEff() const
    {
        return tmp<volScalarField>
        (
            new volScalarField
            (
                "DepsilonEff",
                (this->nut_/sigmaEps_ + this->nu())
            )
        );
    }
```

Then the turbulent kinematic viscosity, ν_t, is evaluated in C file as

```
template<class BasicTurbulenceModel>
void kEpsilon<BasicTurbulenceModel>::correctNut()
{
    this->nut_ = Cmu_*sqr(k_)/epsilon_;
    this->nut_.correctBoundaryConditions();
    fv::options::New(this->mesh_).correct(this->nut_);

    BasicTurbulenceModel::correctNut();
}
```

2.5.2 The Standard k-equation SGS Model in OpenFOAM 1706

The standard k-equation SGS model implemented in OpenFOAM 1706 reads

$$\frac{\partial k_{sgs}}{\partial t} + \nabla \cdot (k_{sgs}\bar{\boldsymbol{U}}) = \nabla \cdot \left[\left(\frac{\nu + \nu_t}{\sigma_k}\right)\nabla k_{sgs}\right] + P - \varepsilon_{sgs} \tag{2.16a}$$

$$\nu_t = C_k \Delta \sqrt{k_{sgs}} \tag{2.16b}$$

where k is the SGS turbulent kinetic energy, $\varepsilon_{sgs} = C_e k^{3/2}/\Delta$ is its dissipation rate. The constant of this model are

$$C_k = 0.094; \quad C_\varepsilon = 1.048; \quad \sigma_k = 1.0; \tag{2.17}$$

This model is implemented in the directory
`$FOAM_SRC/TurbulenceModels/turbulenceModels/LES/kEqn` by two files,
i.e. `kEqn.C` and `kEqn.H`. The H file declares the class, any member data and functions, the C file contains the detailed implementation of the model as (line 186 to line 205)

```
tmp<fvScalarMatrix> kEqn
(
    fvm::ddt(alpha, rho, k_)
  + fvm::div(alphaRhoPhi, k_)
  - fvm::laplacian(alpha*rho*DkEff(), k_)
 ==
    alpha*rho*G
  - fvm::SuSp((2.0/3.0)*alpha*rho*divU, k_)
  - fvm::Sp(this->Ce_*alpha*rho*sqrt(k_)/this->delta(), k_)
  + kSource()
  + fvOptions(alpha, rho, k_)
);

kEqn.ref().relax();
fvOptions.constrain(kEqn.ref());
solve(kEqn);
fvOptions.correct(k_);
bound(k_, this->kMin_);

correctNut();
```

And the effective diffusivity for k_{sgs} is calculated in H file as (line 152 to line 159)

```
//- Return the effective diffusivity for k
tmp<volScalarField> DkEff() const
{
    return tmp<volScalarField>
```

```
            (
                new volScalarField("DkEff", this->nut_ + this->nu())
            );
        }
```

Then the turbulent kinematic viscosity, ν_t, is evaluated in C file as

```
template<class BasicTurbulenceModel>
void kEqn<BasicTurbulenceModel>::correctNut()
{
    this->nut_ = Ck_*sqrt(k_)*this->delta();
    this->nut_.correctBoundaryConditions();
    fv::options::New(this->mesh_).correct(this->nut_);

    BasicTurbulenceModel::correctNut();
}
```

It is observed that the k equations for k-equation SGS and k-ε RANS model are almost the same, except for the evaluation of ε. Therefore, although RANS and LES have fundamentally different mathematical theory, the unclosed governing equations solved on a computer are identical![3] This, theoretically, indicates there could be an universal model which can solve turbulent energy over whole spectrum, i.e. across RANS, LES and DNS.

[3]subject to the spatial filtering operation is implicit for LES.

Chapter 3

Model Implementation

3.1 Warm-up Exercise

The implementation of Inagi wall damping function will be treated as a warm-up exercise before implementing the new SGS model.

As the Turbulence model classes are templated in OpenFOAM 1706, one has to copy the entire Turbulence model directory into the user directory rather than only coping certain existing models as for previous version, such as OpenFOAM 2.3.0. So some preparatory work is needed as below:

```
of+ // reader may use OF1706+
mkdir -p $FOAM_RUN // make sure having user and run directory
foam
cp -r --parents src/TurbulenceModels $WM_PROJECT_USER_DIR
cd $WM_PROJECT_USER_DIR/src/TurbulenceModels
```

Find all the Make directories, change the location of compiled files to relevant user directory, and compile:

```
find . -name Make
sed -i s/FOAM_LIBBIN/FOAM_USER_LIBBIN/g ./*/Make/files
./Allwmake
```

Make sure the following three new shared-object files in
`$WM_PROJECT_USER_DIR/platforms/linux64GccDPInt32Opt/lib`:

```
libcompressibleTurbulenceModels.so
libincompressibleTurbulenceModels.so
libturbulenceModels.so
```

Copy the kEqn SGS model and change the class name to kEqnInagi and conduct all necessary process to compile:

```
cd turbulenceModels/LES
cp -r kEqn kEqnInagi
cd kEqnInagi
mv kEqn.H kEqnInagi.H
mv kEqn.C kEqnInagi.C
sed -i 's/kEqn/kEqnInagi/g *
sed -i 's/"OpenFoam Foundation"/"Your Name"/g *
```

Then open `turbulentTransportModels.C`:

```
cd $WM_PROJECT_USER_DIR/src/TurbulenceModels
vi incompressible/turbulentTransportModels/turbulentTransportModels.C
```

add the following lines:

```
#include "kEqnInagi.H"
makeLESModel(kEqnInagi);
```

under the lines for kEqn model:

```
#include "kEqn.H"
makeLESModel(kEqn);
```

save and close the file, then update lnInclude directory and compile:

```
wmakeLnInclude ../../../turbulenceModels
./../../../Allwmake
```

Then open `kEqnInagi.C` by:

```
vi turbulenceModels/LES/kEqnInagi/kEqnInagi.C
```

and replace

```
    this->nut_ = Ck_*sqrt(k_)*this->delta();
```

with

```
    dimensionedScalar verySmall
    (
            "verySmall",
            dimensionSet (0, 1, -1, 0, 0, 0, 0),
            VSMALL
    );
    this->nut_ = 10*Ck_*k_*this->delta()/
    (10.0*sqrt(k_) + this->delta()*sqrt(2*magSqr(symm(fvc::grad(this->U_)))) + verySmall);
```

Then save and close the file, update `turbulentTransportModels.C` and recompile:

```
touch incompressible/turbulentTransportModels/turbulentTransportModels.C
./../../../Allwmake
```

So far the Inagi wall damping function has been implemented, its preliminary result will be shown together with the new SGS model in later Chapter.

3.2 Preparatory Work

Same as the warm-up exercise, some preparatory work is needed before implementing the model: Copy the kEqn SGS model and change the class name to kEpsilonSAS and conduct all necessary process to compile:

```
cd $WM_PROJECT_USER_DIR/src/TurbulenceModels
cd turbulenceModels/LES
cp -r kEqn kEpsilonSAS
cd kEpsilonSAS
mv kEqn.H kEpsilonSAS.H
mv kEqn.C kEpsilonSAS.C
sed -i 's/kEqn/kEpsilonSAS/g *
sed -i 's/"OpenFoam Foundation"/"Your Name"/g *
```

Then open `turbulentTransportModels.C` via:

```
cd $WM_PROJECT_USER_DIR/src/TurbulenceModels
vi incompressible/turbulentTransportModels/turbulentTransportModels.C
```

add following lines:

```
#include "kEpsilonSAS.H"
makeLESModel(kEpsilonSAS);
```

under lines for kEqn model:

```
#include "kEqn.H"
makeLESModel(kEqn);
```

save and close the file, then update lnInclude directory and compile:

```
wmakeLnInclude ../../../turbulenceModels
./../../../Allwmake
```

Following the preparatory work, the new k-ε SGS model can now be implemented by adding the functionality from k-ε RANS model to the existing k-equation SGS model with extra modifications.

It is a good practice to modify the code by adopting the step-by-step approach, starting from constants and existing functionality, then gradually reach the complicated functions. Every major step will be tested on cases before moving to next step. It is the methodology adopted when implementing the new model. However, it will be lengthy if all the steps are shown here. Therefore only the differences between kEqn SGS model and the final implementation of k-ε SGS model are provided.

3.3 Detailed Implementation

3.3.1 H file

Under `// Fields` list:

add

```
volScalarField epsilon_;
volVectorField UMean_;
volScalarField kR_;
volScalarField dimlessGradkR_;
volScalarField alfa_;
volScalarField nutByAlfa_;
```

under

```
 volScalarField k_;
```

Under `// Model constants` list:

replace

```
dimensionedScalar Ck_;
```

with

```
dimensionedScalar Cnu_;
dimensionedScalar Ce1_;
dimensionedScalar sigmaK_;
dimensionedScalar sigmaEps_;
```

Under `// Protected Member Functions` list:

add

```
virtual tmp<fvScalarMatrix> epsilonSource() const;
```

under

```
virtual tmp<fvScalarMatrix> kSource() const;
```

Under `// Member Functions` list:

replace

```
//- Return sub-grid dissipation rate
virtual tmp<volScalarField> epsilon() const;

//- Return the effective diffusivity for k
tmp<volScalarField> DkEff() const
{
    return tmp<volScalarField>
    (
        new volScalarField("DkEff", this->nut_ + this->nu())
    );
}
```

with

```
//- Return sub-grid dissipation rate
virtual tmp<volScalarField> epsilon() const
{
    return epsilon_; // added
}

//- Return the effective diffusivity for k
tmp<volScalarField> DkEff() const
{
    return tmp<volScalarField>
    (
        new volScalarField("DkEff", (nutByAlfa_ + this->nu())/sigmaK_)
    );
}

//- Return the effective diffusivity for epsilon
tmp<volScalarField> DepsilonEff() const
{
    return tmp<volScalarField>
    (
        new volScalarField("DepsilonEff", (this->nutByAlfa_ +
        this->nu())/sigmaEps_)
    );
}
```

3.3.2 C file

Under `// Protected Member Functions` list:

within the function of

```
template<class BasicTurbulenceModel>
void kEpsilonSAS<BasicTurbulenceModel>::correctNut()
```

replace

```
    this->nut_ = Ck_*sqrt(k_)*this->delta();
```

with

```
    kR_ = 0.5*magSqr(UMean_ - this->U_);
    volVectorField gradSqrtkR = fvc::grad(sqrt(kR_));

    surfaceVectorField fV = this->mesh_.Sf();
    volScalarField surfaceSumX = 0.5*fvc::surfaceSum(mag(fV.component(0)));
    volScalarField surfaceSumY = 0.5*fvc::surfaceSum(mag(fV.component(1)));
    volScalarField surfaceSumZ = 0.5*fvc::surfaceSum(mag(fV.component(2)));

    volScalarField cv
    (
            IOobject
            (
                    "cv",
                    this->runTime_.timeName(),
                    this->mesh_,
                    IOobject::NO_READ,
                    IOobject::AUTO_WRITE
            ),
            this->mesh_,
            dimensionedScalar("zero",dimVolume,0.0)
    );

    cv.ref() = this->mesh_.V();

    dimensionedScalar surfaceMin
    (
            "surfaceMin",
            dimensionSet (0, 2, 0, 0, 0, 0, 0),
            VSMALL
    );

    dimlessGradkR_ =
    (
     sqr(cv / (surfaceSumX + surfaceMin) * gradSqrtkR.component(0)) +
     sqr(cv / (surfaceSumY + surfaceMin) * gradSqrtkR.component(1)) +
     sqr(cv / (surfaceSumZ + surfaceMin) * gradSqrtkR.component(2))
    )
    /
    (kR_ + this->kMin_);

    alfa_ =
    1.5*
       (
         1.0 -
         0.28 * sqr(k_ / (k_ + kR_ + this->kMin_)) /
         (dimlessGradkR_ + 0.11)
       );
```

```
    // Calculate sgs nut
    nutByAlfa_ = Cnu_*sqr(k_)/(epsilon_+this->epsilonMin_)*
    (k_/(k_+kR_+this->kMin_));
    this->nut_ = alfa_*nutByAlfa_;
```

also add

```
template<class BasicTurbulenceModel>
tmp<fvScalarMatrix> kEpsilonSAS<BasicTurbulenceModel>::epsilonSource() const
{
    return tmp<fvScalarMatrix>
    (
        new fvScalarMatrix
        (
            epsilon_,
            dimVolume*this->rho_.dimensions()*epsilon_.dimensions()
            /dimTime
        )
    );
}
```

under

```
template<class BasicTurbulenceModel>
tmp<fvScalarMatrix> kEqn<BasicTurbulenceModel>::kSource() const
{
    return tmp<fvScalarMatrix>
    (
        new fvScalarMatrix
        (
            k_,
            dimVolume*this->rho_.dimensions()*k_.dimensions()
            /dimTime
        )
    );
}
```

Within `Constructors`:

replace

```
    Ck_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "Ck",
            this->coeffDict_,
            0.094
        )
    )
```

with

```
    epsilon_
    (
        IOobject
```

```
        (
            IOobject::groupName("epsilon", this->U_.group()),
            this->runTime_.timeName(),
            this->mesh_,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        this->mesh_
    ),

    UMean_
    (
        IOobject
        (
            IOobject::groupName("UMean", this->U_.group()),
            this->runTime_.timeName(),
            this->mesh_,
            IOobject::MUST_READ,
            IOobject::NO_WRITE
        ),
        this->mesh_
    ),

    kR_
    (
        IOobject
        (
            IOobject::groupName("kR", this->U_.group()),
            this->runTime_.timeName(),
            this->mesh_,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        this->mesh_,
        dimensionedScalar("kR",this->k_.dimensions(),SMALL)
    ),

    dimlessGradkR_
    (
            IOobject
            (
                    "dimlessGradkR",
                    this->runTime_.timeName(),
                    this->mesh_,
                    IOobject::NO_READ,
                    IOobject::AUTO_WRITE
            ),
            this->mesh_,
            dimensionedScalar("dimlessGradkR",dimless,SMALL)
    ),

    alfa_
    (
        IOobject
```

```
        (
            IOobject::groupName("alfa", this->U_.group()),
            this->runTime_.timeName(),
            this->mesh_,
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),
        this->mesh_,
        dimensionedScalar("alfa",dimless,SMALL)
    ),

    nutByAlfa_
    (
        IOobject
        (
            IOobject::groupName("nutByAlfa", this->U_.group()),
            this->runTime_.timeName(),
            this->mesh_,
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),
        this->mesh_,
        dimensionedScalar("nutByAlfa",this->nut_.dimensions(),SMALL)
    ),

    Cnu_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "Cnu",
            this->coeffDict_,
            0.18
        )
    ),

    Ce1_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "Ce1",
            this->coeffDict_,
            1.55
        )
    ),

    sigmaK_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "sigmaK",
            this->coeffDict_,
            1.0
        )
    ),
```

```
    sigmaEps_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "sigmaEps",
            this->coeffDict_,
            1.2
        )
    )
```

Also add

```
    bound(epsilon_, this->epsilonMin_);
```

under

```
    bound(k_, this->kMin_);
```

Within the function of

```
template<class BasicTurbulenceModel>
bool kEqn<BasicTurbulenceModel>::read()
```

replace

```
        Ck_.readIfPresent(this->coeffDict());
```

with

```
        Cnu_.readIfPresent(this->coeffDict());
        Ce1_.readIfPresent(this->coeffDict());
        sigmaK_.readIfPresent(this->coeffDict());
        sigmaEps_.readIfPresent(this->coeffDict());
```

Remove the function below:

```
template<class BasicTurbulenceModel>
tmp<volScalarField> kEpsilonSAS<BasicTurbulenceModel>::epsilon() const
{
    return tmp<volScalarField>
    (
        new volScalarField
        (
            IOobject
            (
                IOobject::groupName("epsilon", this->U_.group()),
                this->runTime_.timeName(),
                this->mesh_,
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
            this->Ce_*k()*sqrt(k())/this->delta()
        )
    );
}
```

Within the function of:

```
template<class BasicTurbulenceModel>
void kEpsilonSAS<BasicTurbulenceModel>::correct()
```

add

```
    Info << "This is kEpsilonSAS" << endl;
```

before

```
    // Local references
```

Also add

```
    // Calculate Ce2 dynamically
    tmp<volScalarField> ReT(sqr(k_)/(this->nu()*(epsilon_+this->epsilonMin_)));
    tmp<volScalarField> f(sqrt(sqr(ReT()/30)+ReT()/15)-ReT()/30);
    volScalarField Ce2(11/6*f()+25/(ReT()+SMALL)*sqr(f()));
    ReT.clear();
    f.clear();

    volScalarField epsilon(epsilon_ + this->epsilonMin_);
    volScalarField k(k_ + this->kMin_); // added

    // Dissipation equation // added
    tmp<fvScalarMatrix> epsEqn
    (
        fvm::ddt(alpha, rho, epsilon_)
      + fvm::div(alphaRhoPhi, epsilon_)
      - fvm::laplacian(alpha*rho*DepsilonEff(), epsilon_)
     ==
        Ce1_*alpha*rho*G*epsilon_/k
      - fvm::SuSp(((2.0/3.0)*Ce1_)*alpha*rho*divU, epsilon_)
      - fvm::Sp(Ce2*alpha*rho*epsilon_/k, epsilon_)
      + epsilonSource()
      + fvOptions(alpha, rho, epsilon_)
    );

    epsEqn.ref().relax();
    fvOptions.constrain(epsEqn.ref());
    //epsEqn.ref().boundaryManipulate(epsilon_.boundaryFieldRef());
    solve(epsEqn);
    fvOptions.correct(epsilon_);
    bound(epsilon_, this->epsilonMin_);
```

before

```
    tmp<fvScalarMatrix> kEqn
```

Within the function of

```
    tmp<fvScalarMatrix> kEqn
```

replace

```
        alpha*rho*G
```

with

```
        alpha*rho*G*alfa_
```

also replace

```
- fvm::Sp(this->Ce_*alpha*rho*sqrt(k_)/this->delta(), k_)
```

with

```
- fvm::Sp(alpha*rho*epsilon_/k, k_)
```

Then recompile:

```
cd $WM_PROJECT_USER_DIR/src/TurbulenceModels
touch incompressible/turbulentTransportModels/turbulentTransportModels.C
./Allwmake
```

Chapter 4

Tutorial Setup

This chapter covers the necessary setup needed to get the pitzDaily case running with the Inagi wall damping function and the new turbulence model. The original case details can be found in following directories respectively:

`$FOAM_TUTORIALS/incompressible/pisoFoam/LES/pitzDaily`

As mentioned in Prerequisites, the readers are presumed to have some knowledge on pitzDaily tutorial cases. So the basic introduction of the case, such as geometry will not be presented here.

4.1 k-equation with Inagi wall damping function

Copy the pitzDaily tutorial to the run directory:

```
run
rm -r pitzDaily
cp -r $FOAM_TUTORIALS/incompressible/pisoFoam/LES/pitzDaily \
$FOAM_RUN/pitzDailyKInagi
cd $FOAM_RUN/pitzDailyKInagi
```

The file structure of the pitzDaily case is similar to other OpenFOAM tutorials which contain normal `/0`, `/constant` and `/system` directories. As usual, in `/system` directory, one can find `controlDict` for write and time control, `blockMeshDict` for mesh setup, `fvSchemes` for discretisation method, and `fvSolution` for solver control. In `/constant` directory, one can find `transportProperties` for viscosity and `turbulenceProperties` for turbulence models. As the Wall damping function is hard-coded into the SGS model `kEqnInagi`, so the damping function can be applied by directly replacing the SGS model in `/constant/turbulenceProperties`, i.e.:
replace

```
    LESModel        dynamicKEqn;
```

with

```
    LESModel        kEqInagi;
```

Then, just type following command to run the case:

```
blockMesh
pisoFoam >& log&
```

4.2 Two-equation SGS Model

4.2.1 Getting Started

Same as the case setup for Inagi wall damping function, the pitzDaily tutorial should be copied to the run directory first:

```
run
rm -r pitzDaily
cp -r $FOAM_TUTORIALS/incompressible/pisoFoam/LES/pitzDaily \
$FOAM_RUN/pitzDailyKESAS
cd $FOAM_RUN/pitzDailyKESAS
```

4.2.2 Changes in 0 directory

As this case is only for demonstration, so extra parameters, such as `nuTilda` and `s`, which are not solved in current turbulence model should be removed. However, the `U`, `p`, `nuSgs` and `k` stay the same as in existing tutorial.

The new SGS model will solve ε-equation which requires `epsilon` dictionary to be provided in `/0` directory as an initial condition. So copy `/0/k` to `/0/epsilon`, then change `object` from `k` to `epsilon`, and make remaining `epsilon` file as:

```
dimensions      [0 2 -3 0 0 0 0];

internalField   uniform 79e-5;

boundaryField
{
    inlet
    {
        type            fixedValue;
        value           uniform 79e-5;
    }

    outlet
    {
        type            zeroGradient;
    }

    upperWall
    {
        type            fixedValue;
        value           uniform 0;
    }

    lowerWall
    {
        type            fixedValue;
        value           uniform 0;
    }

    frontAndBack
    {
        type            empty;
    }
}
```

`$FOAM_TUTORIALS` has `pitzDaily` case for k-equation SGS model and k-ε RANS model. The initial k value for current case stays the same as the tutorial case for k-equation SGS model. It is found out that the `k` value in `/0` directory for the SGS model is much smaller than the case for RANS model. So the ratio of ε between the RANS and LES cases is kept as the same ratio of `k` between existing RANS and LES cases.

4.2.3 Changes in system directory

In `/system/controlDict`:
comment out everything except `fieldAverage1`. Then within the `fieldAverage1`
only keep `mean on;` for `U`, and turn others off.

In `/system/fvSchemes`:
add

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

under

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

In `/system/fvSolution`:
replace

```
"(U|k|B|nuTilda|s)"
```

with

```
"(U|k|epsilon|B|nuTilda|s)"
```

4.2.4 Changes in constant directory

Keep the `/constant/transportProperties` unchanged.

In `/constant/turbulenceProperties`:
replace

```
LESModel        dynamicKEqn;
```

with

```
LESModel        kEpsilonSAS;
```

also replace

```
delta        cubeRootVol;
```

to

```
delta        vanDriest;
```

4.2.5 Running the code

Just type following command to run the case:

```
blockMesh
pisoFoam >& log&
```

4.3 Post-processing in ParaView

It is reported that the new model can lead to divergence, whereas removing the energy backscatter term in k-equation can improve the stability. The real cause for divergence is subject to further investigation. The result shown in the test cases is based on the code which is without the energy backscatter term in k-equation.

A preliminary comparison between k-equation with Inagi wall damping function, k-ε SGS model, k-equation SGS model with `cubeRootVol` Δ and conventional k-ε RANS model is conducted. pitzDaily tutorial case located in `$FOAM_TUTORIALS` is used as the test case.

The inlet velocity are set to be $U = 10m/s$ with turbulence intensity of 2%, 1% and 1% at x, y and z direction respectively. Initial condition for pressure, p, eddy viscosity, ν_t, k, and ε value are kept as same as in the original tutorial cases. The result are analysed at $t = 0.3s$.

Fig. 4.1 shows the instantaneous velocity magnitude for all SGS models, with the top one being the k-equation with Inagi wall damping function, middle one being the new model, and the bottom one being the k-equation SGS model with `cubeRootVol` Δ. The figure indicates that the new model can produce a "LES-like" velocity profile as k-equation SGS model does. In addition, the Inagi wall damping function shows a different velocity profile from k-equation SGS model with `cubeRootVol` Δ, indicating the modification works.

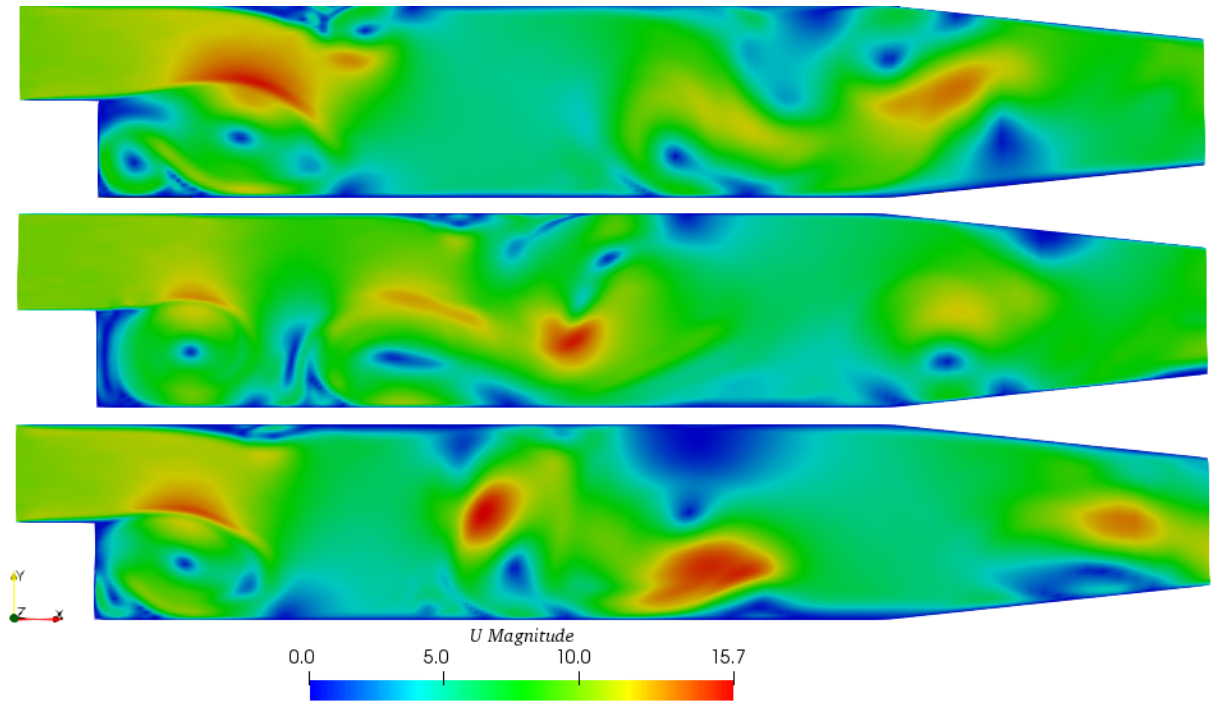

Figure 4.1: Instantaneous velocity magnitude for k-equation with Inagi wall damping function (top), k-ε SGS model (middle), and k-equation SGS model with normal Δ (bottom).

Fig. 4.2 shows the time-averaged velocity magnitude for the three SGS models plus k-ε RANS model. It is found that the time-averaged velocity profile of the three SGS models have some similarities, but different from conventional k-ε RANS model. However the RANS model cannot predict the unsteadiness for the current case.

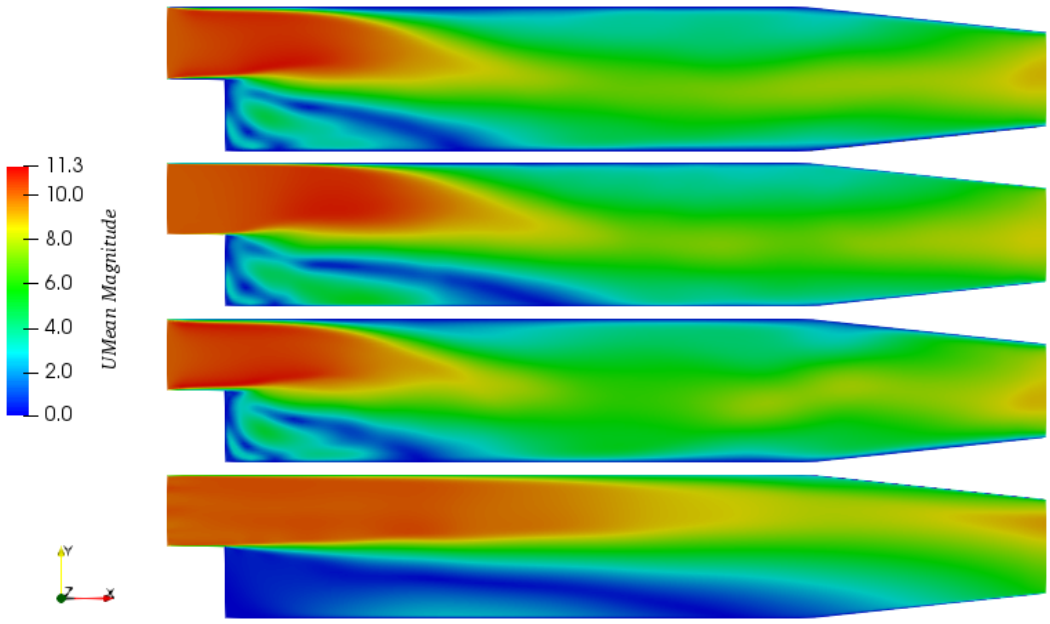

Figure 4.2: Time-averaged velocity magnitude for k-equation SGS model with Inagi wall damping function (top), k-ε SGS model (second from top), k-equation SGS model with normal Δ (second from bottom) and k-ε RANS model (bottom).

Chapter 5

Conclusion and Future Work

A wall damping function and new k-ε based Two-equation SGS model have been implemented and preliminary tested on 2D pitzDaily case. However the preliminary test shows that the new model can lead to divergence, the stability can be improved by removing the energy backscatter term from k-equation. The real cause for divergence is subject to further investigation. After removing the backscatter term, the wall dumping function and the new model are compared with conventional k-equation SGS model and conventional k-ε RANS model on the pitzDaily case . The results indicate that same as the conventional k-equation model, the new model can resolve turbulence and produce an unsteady solution. In addition the Inagi wall damping function also shows effects on the result indicating the implementation is working. As expected, the conventional k-ε RANS model is not able to capture the unsteadiness.

In terms of the future work, it is important to find out the cause of divergence. Then corresponding improvement will be done followed by extensive tests on channel flow, backward facing step and other more complicated scenarios.

Bibliography

[1] Lars Davidson. "Evaluation of the SST-SAS model: channel flow, asymmetric diffuser and axi-symmetric hill". In: (2006).

[2] Masahide Inagaki, Tsuguo Kondoh, and Yasutaka Nagano. "A mixed-time-scale SGS model with fixed model-parameters for practical LES". In: 127 (Jan. 2005).

[3] J. Blair Perot and Jason Gadebusch. "A self-adapting turbulence model for flow simulation at any mesh resolution". In: *Physics of Fluids* 19.11 (2007), p. 115105. DOI: 10.1063/1.2780195. eprint: https://doi.org/10.1063/1.2780195. URL: https://doi.org/10.1063/1.2780195.

[4] H.K. Versteeg and W. Malalasekera. *An Introduction to Computational Fluid Dynamics: The Finite Volume Method*. Pearson Education Limited, 2007. ISBN: 9780131274983. URL: https://books.google.se/books?id=RvBZ-UMpGzIC.

Study questions

1. How to find available turbulence models in OpenFOAM?
2. What is eddy viscosity turbulence model?
3. What's the preferred way of developing new model in OpenFOAM? Use the Two-equation SGS model as an example.
4. How to use the Two-equation SGS model implemented in this tutorial?
5. How to modify the Two-equation SGS model implemented in this tutorial?