Implement a turbulence model

(This page is part of the course 'CFD with OpenSource Software', by Håkan Nilsson, see **http://www.tfd.chalmers.se/~hani/kurser/OS_CFD/#YEAR_2019**)

We will here investigate how turbulence models are incorporated in the solution procedure of the solvers, and how a particular turbulence model is chosen by the user and pointed at in run-time by the code. We will then investigate how the kEpsilon, kOmegaSST and kOmegaSSTSAS models are implemented. Finally we will implement a new version of the kOmegaSST model, named kOmegaSSTF, which puts a filter to the turbulent viscosity.

Initial notes

The implementation of turbulence models was changed completely in OpenFOAM-3.0. The current descriptions are based on the new implementation (in fact OpenFOAM-v1906). If you have an old version of OpenFOAM, or if you are using FOAM-extend, you can have a look at the slides from previous years.

If you are only interested in the basic steps to copy an existing turbulence model and compile your own modified version of that model, please go to the section named "Implement your own versions of kEpsilon and kOmegaSST".

Basic usage of turbulence models

We will use the simpleFoam/pitzDaily tutorial when we examine the implementation of turbulence models. We start by going through the basic usage of turbulence models for that case.

Start by making sure that you have a fresh copy of the case in your $FOAM_RUN directory:

```
run
rm -r pitzDaily
cp -r $FOAM_TUTORIALS/incompressible/simpleFoam/pitzDaily .
cd pitzDaily
blockMesh
```

The default settings for turbulence in the constant/turbulenceProperties file/dictionary are:

```
simulationType RAS;

RAS
{
    // Tested with kEpsilon, realizableKE, kOmega, kOmegaSST, v2f,
    // ShihQuadraticKE, LienCubicKE.
    RASModel        kEpsilon;

    turbulence      on;

    printCoeffs     on;
}
```

i.e. the kEpsilon RAS model will be used, it is active, and it will print its coefficients to the terminal window (commonly forwarded to a log file).

If you like you can run the case by issuing the command simpleFoam, and investigate the results in paraFoam. You will see that there are two fields relating to turbulence: k for turbulent kinetic energy, and epsilon for turbulence dissipation. Those fields are solved using discretized equations, so there is a need for solver and scheme settings in system/{fvSolution,fvSchemes}. You can of course also see those files in the time directories, where internalField (initial conditions or solution) and boundary conditions are specified/shown. Here it should be noted that other turbulence models may be based on other variables, and it thus makes sense that it is the turbulence model that defines the variables to be used. The k and epsilon objects are thus not constructed in the createFields file of the simpleFoam solver. Instead they are constructed by the turbulence model that has been chosen at run-time. We will see this later when we look at the code of turbulence models.

Some of the initial output in the terminal window when running simpleFoam is:

```
Selecting turbulence model type RAS
Selecting RAS turbulence model kEpsilon
RAS
{
    RASModed        kEpsilon;
    turbulence      on;
    printCoeffs     on;
    Cmu             0.09;
    C1              1.44;
    C2              1.92;
    C3              0;
    sigmak          1;
    sigmaEps        1.3;
}
```

This shows that the kEpsilon model was in fact chosen for the simulation, and it also reports the model coefficients that are used in the simulation (since the RAS sub-dictionary above specified 'printCoeffs on').

Now we will figure out how the simpleFoam solver incorporates turbulence modeling in general, and the kEpsilon model in particular.

Turbulence modeling in the simpleFoam solver

The simpleFoam solver is implemented in $FOAM_SOLVERS/incompressible/simpleFoam, where the following files and directories can be found:

```
createFields.H  overSimpleFoam  porousSimpleFoam  SRFSimpleFoam
Make            pEqn.H          simpleFoam.C      UEqn.H
```

The directories overSimpleFoam, porousSimpleFoam and SRFSimpleFoam are other versions of the simpleFoam solver, which we do not consider here. The simpleFoam solver is independent of those solvers, but those additional solvers typically re-use some of the files in the simpleFoam directory. The Make directory contains compiler instructions. The main files for the simpleFoam solver are thus:

```
createFields.H pEqn.H simpleFoam.C UEqn.H
```

Here the simpleFoam.C file is the main file, in which the other files are included by #include-statements.

The turbulence modeling declarations are added to the simpleFoam solver by including a header file in the simpleFoam.C file:

```
#include "turbulentTransportModel.H"
```

That file is located in $FOAM_SRC/TurbulenceModels/incompressible/turbulentTransportModels, and it contains:

```
#include "IncompressibleTurbulenceModel.H"
#include "laminarModel.H"
#include "RASModel.H"
#include "LESModel.H"
#include "incompressible/transportModel/transportModel.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

namespace Foam
{
    namespace incompressible
    {
        typedef IncompressibleTurbulenceModel<transportModel> turbulenceModel;

        typedef laminarModel<turbulenceModel> laminarModel;
        typedef RASModel<turbulenceModel> RASModel;
        typedef LESModel<turbulenceModel> LESModel;

        template<class BasicCompressibleTurbulenceModel>
        autoPtr<BasicCompressibleTurbulenceModel> New
        (
            const volVectorField& U,
            const surfaceScalarField& phi,
            const typename BasicCompressibleTurbulenceModel::transportModel&
                transport,
            const word& propertiesName = turbulenceModel::propertiesName
        );
    }
}
```

I.e. it adds the declations of some additional classes to the simpleFoam solver: IncompressibleTurbulenceModel, laminarModel, RASModel, LESModel and transportModel. Then it uses those additional classes while defining some typeDefs (we will soon refer back to the first one, so please have a look at it) and *declaring* a templated function named New. The effect of the New function will depend on which turbulence model we have chosen, and the New function that will eventually be executed belongs to a specific turbulence model, as we will see later. However, at this point the New function is only declared and defined according to the above pieces of code. It is not executed.

The next step in simpleFoam (with respect to turbulence modeling) is to construct a pointer for the turbulence, which is done at the end of the included file createFields.H:

```
autoPtr<incompressible::turbulenceModel> turbulence
(
    incompressible::turbulenceModel::New(U, phi, laminarTransport)
);
```

I.e., the function New that is called (see that single line with the word "New" above) belongs to namespace incompressible (see the turbulentTransportModel.H file) and to the class turbulenceModel (which is a typeDef of IncompressibleTurbulenceModel<transportModel> in turbulentTransportModel.H - I asked you to look specifically at it before). The New function in turbulentTransportModel.H returns an autoPtr<BasicCompressibleTurbulenceModel>, and at the first line in the above piece of code we make sure that the template parameter BasicCompressibleTurbulenceModel in turbulentTransportModel.H should be interpreted as incompressible::turbulenceModel, where "incompressible" and "turbulenceModel" were described just above, ending up with a BasicCompressibleTurbulenceModel template parameter as IncompressibleTurbulenceModel<transportModel>, and a fact that the New function of that class will be called. We thus need to dig further into the class IncompressibleTurbulenceModel<transportModel> later (in the following section, for those who really want to, since we here have to stay at a somewhat high level in the code). The arguments that are passed to the function New in the above piece of code are U, phi, and laminarTransport. The default value is used for the fourth argument propertiesName (see the first declaration above), i.e. turbulenceModel::propertiesName. It is set in $FOAM_SRC/TurbulenceModels/turbulenceModels/turbulenceModel.C (we are explicitly naming that class in turbulenceModel::propertiesName) as:

```
const Foam::word Foam::turbulenceModel::propertiesName("turbulenceProperties");
```

Here we see that the default fourth argument for the New function is the file name of the turbulence properties dictionary, which we can thus change if we like in createFields.H (you can try that as an exercise if you like). The fields U and phi are the velocity and face flux fields that are part of the solution in simpleFoam, and those fields are constructed in createFields.H. The object laminarTransport is also constructed in createFields.H by:

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

Note that for that line to work it was necessary to add the declaration of that class in simpleFoam.C:

```
#include "singlePhaseTransportModel.H"
```

This constructs the object named laminarTransport as a singlePhaseTransportModel. That class is implemented in $FOAM_SRC/transportModels/incompressible/singlePhaseTransportModel, and it makes sure that the transportProperties dictionary is read and that the viscosity model is set (Newtonian or one of the non-Newtonian models). The laminarTransport object is supplied as one of the parameters to the function named New, since eddy-viscosity turbulence models also affect the flow through the viscosity, by computing an efficient viscosity, and the turbulence model must thus also take into account the effects of laminar viscosity and non-Newtonian behaviour on the laminar viscosity. We will see this later.

The pointer named turbulence and its member functions are then used in the simpleFoam solver (in simpleFoam.C and UEqn.H) to add the solution and effects of the turbulence model of choise, mainly:

```
turbulence->validate();
+ turbulence->divDevReff(U)
turbulence->correct();
```

We realize that the object named turbulence is a pointer, since the functions of the object are called using '->'.The operation of those member functions depend on which turbulence model we are using, which is specified when we run the case. We must thus later figure out where those functions are defined for each turbulence model.

It is worth to repeat that it is not obvious, when looking only at the files in the simpleFoam directory, how the fields for the turbulence model are constructed, and why they are read from the start time directory and written to the following time directories. Different turbulence models use different fields, so it is each turbulence model that constructs the necessary fields for that particular turbulence model, and makes sure that they are written to the time directories when the line runTime.write() in simpleFoam.C is executed.

In the next section we will dig deeper into the code to figure out what the New function is doing, and how the turbulence pointer ends up pointing at a particular turbulence model. The section can be skipped if you just want to accept that the turbulence pointer points at the turbulence model class that you have specified in the constant/turbulenceProperties dictionary. It is highly suggested that you skip the section unless you are really interested!

The function named New (only if you are really interested!)

This section is highly complicated, and not 100% clear. I also do not spend time on updating it for new versions, so there may be some small differences compared to the source code. Those who are VERY interested can go through it and get back to me with suggested improvements. Those who are NOT very interested can skip to the next section.

It was discussed in the previous section that the function named New was called when constructing the pointer named turbulence, and that it belonged to the IncompressibleTurbulenceModel<transportModel> class. We will now have a look at the implementation of that class and we find the declaration of the function New in $FOAM_SRC/TurbulenceModels/incompressible/IncompressibleTurbulenceModel/IncompressibleTurbulenceModel.H:

```
        //- Return a reference to the selected turbulence model
        static autoPtr<IncompressibleTurbulenceModel> New
        (
            const volVectorField& U,
            const surfaceScalarField& phi,
            const TransportModel& trasportModel,
            const word& propertiesName = turbulenceModel::propertiesName
        );
```

The definition of the same function is found in the correcponding IncompressibleTurbulenceModel.C file:

```
template<class TransportModel>
Foam::autoPtr<Foam::IncompressibleTurbulenceModel<TransportModel>>
Foam::IncompressibleTurbulenceModel<TransportModel>::New
(
    const volVectorField& U,
    const surfaceScalarField& phi,
    const TransportModel& transport,
    const word& propertiesName
)
{
    return autoPtr<IncompressibleTurbulenceModel>
    (
         static_cast<IncompressibleTurbulenceModel*>(
         TurbulenceModel
         <
             geometricOneField,
             geometricOneField,
             incompressibleTurbulenceModel,
             TransportModel
         >::New
         (
             geometricOneField(),
             geometricOneField(),
             U,
             phi,
             phi,
             transport,
             propertiesName
         ).ptr())
    );
}
```

We see that the function returns the output of the New function (taking 7 arguments) of the base class TurbulenceModel, which IncompressibleTurbulenceModel Inherits from. That function is defined in $FOAM_SRC/TurbulenceModels/turbulenceModels/TurbulenceModel/TurbulenceModel.C, as:

```
template
<
    class Alpha,
    class Rho,
    class BasicTurbulenceModel,
    class TransportModel
>
Foam::autoPtr
<
    Foam::TurbulenceModel<Alpha, Rho, BasicTurbulenceModel, TransportModel>
>
Foam::TurbulenceModel<Alpha, Rho, BasicTurbulenceModel, TransportModel>::New
(
    const alphaField& alpha,
    const rhoField& rho,
    const volVectorField& U,
    const surfaceScalarField& alphaRhoPhi,
    const surfaceScalarField& phi,
    const transportModel& transport,
    const word& propertiesName
)
{
    // get model name, but do not register the dictionary
    // otherwise it is registered in the database twice
    const word modelType
    (
        IOdictionary
        (
            IOobject
            (
                IOobject::groupName(propertiesName, U.group()),
                U.time().constant(),
                U.db(),
                IOobject::MUST_READ_IF_MODIFIED,
                IOobject::NO_WRITE,
                false
            )
        ).lookup("simulationType")
    );

    Info<< "Selecting turbulence model type " << modelType << endl;

    typename dictionaryConstructorTable::iterator cstrIter =
        dictionaryConstructorTablePtr_->find(modelType);

    if (!cstrIter.found())
    {
        FatalErrorInFunction
            << "Unknown TurbulenceModel type "
            << modelType << nl << nl
            << "Valid TurbulenceModel types:" << endl
            << dictionaryConstructorTablePtr_->sortedToc()
            << exit(FatalError);
    }

    return autoPtr<TurbulenceModel>
    (
        cstrIter()(alpha, rho, U, alphaRhoPhi, phi, transport, propertiesName)
    );
}
```

Following the code above, it reads the turbulenceProperties dictionary, determines the simulationType/modelType, and checks that the simulationType/modelType is one of the available types. Without describing the details related to the object cstrIter and its class (I don't know all the details, so please help me sort this out), the return statement at the end returns a pointer to the particular turbulence model type (laminar/RAS/LES) that is specified in the turbulenceProperties dictionary. In the case of the default simpleFoam/pitzDaily tutorial the turbulence model type is set to RAS. We end up in $FOAM_SRC/TurbulenceModels/turbulenceModels/RAS/RASModel/RASModel.C in the corresponding function named New:

```
template<class BasicTurbulenceModel>
Foam::autoPtr<Foam::RASModel<BasicTurbulenceModel>>
Foam::RASModel<BasicTurbulenceModel>::New
(
    const alphaField& alpha,
    const rhoField& rho,
    const volVectorField& U,
    const surfaceScalarField& alphaRhoPhi,
    const surfaceScalarField& phi,
    const transportModel& transport,
    const word& propertiesName
)
{
    // get model name, but do not register the dictionary
    // otherwise it is registered in the database twice
    const word modelType
    (
        IOdictionary
        (
            IOobject
            (
                IOobject::groupName(propertiesName, U.group()),
                U.time().constant(),
                U.db(),
                IOobject::MUST_READ_IF_MODIFIED,
                IOobject::NO_WRITE,
                false
            )
        ).subDict("RAS").lookup("RASModel")
    );

    Info<< "Selecting RAS turbulence model " << modelType << endl;

    typename dictionaryConstructorTable::iterator cstrIter =
        dictionaryConstructorTablePtr_->find(modelType);

    if (cstrIter == dictionaryConstructorTablePtr_->end())
    {
        FatalErrorInFunction
            << "Unknown RASModel type "
            << modelType << nl << nl
            << "Valid RASModel types:" << endl
            << dictionaryConstructorTablePtr_->sortedToc()
            << exit(FatalError);
    }

    return autoPtr<RASModel>
    (
        cstrIter()(alpha, rho, U, alphaRhoPhi, phi, transport, propertiesName)
    );
}
```

We see that the particular RAS turbulence model is selected based on the setting in the sub-dictionary RASModel, in the turbulenceProperties dictionary. The same procedure as before is again used to point at the particular RAS turbulence model, and we end up in the constructor of the kEpsilon model (according to the standard settings of the simpleFoam/pitzDaily tutorial), in $FOAM_SRC/TurbulenceModels/turbulenceModels/RAS/kEpsilon/kEpsilon.C, which initializes the un-named object that the object named turbulence is pointing at, and writes out the model coefficients (as we will see later). The object named turbulence can now be used to call the member functions in the kEpsilon class, such as validate(), divDevRef() and correct(), which we will have a look at soon.

The kEpsilon Model

The kEpsilon model is implemented in $FOAM_SRC/TurbulenceModels/turbulenceModels/RAS/kEpsilon. The class inheritance is given by:

```
template<class BasicTurbulenceModel>
class kEpsilon
:
 public eddyViscosity<RASModel<BasicTurbulenceModel>>
{
```

I.e., the kEpsilon class is an eddyViscosity model, with the RASModel specialization using the template parameter BasicTurbulenceModel (in this case interpreted as IncompressibleTurbulenceModel<transportModel>.

The class declares and defines some member data and member functions as well as some typeDefs and a TypeName.

The constructor is defined in kEpsilon.C:

```
template<class BasicTurbulenceModel>
kEpsilon<BasicTurbulenceModel>::kEpsilon
(
    const alphaField& alpha,
    const rhoField& rho,
    const volVectorField& U,
    const surfaceScalarField& alphaRhoPhi,
    const surfaceScalarField& phi,
    const transportModel& transport,
    const word& propertiesName,
    const word& type
)
:
    eddyViscosity<RASModel<BasicTurbulenceModel>>
    (
        type,
        alpha,
        rho,
        U,
        alphaRhoPhi,
        phi,
        transport,
        propertiesName
    ),

    Cmu_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "Cmu",
            this->coeffDict_,
            0.09
        )
    ),
    C1_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "C1",
            this->coeffDict_,
            1.44
        )
    ),
    C2_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "C2",
            this->coeffDict_,
            1.92
        )
    ),
    C3_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "C3",
            this->coeffDict_,
            0
        )
    ),
    sigmak_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "sigmak",
            this->coeffDict_,
            1.0
        )
    ),
    sigmaEps_
    (
        dimensioned<scalar>::lookupOrAddToDict
        (
            "sigmaEps",
            this->coeffDict_,
            1.3
        )
    ),

    k_
    (
        IOobject
        (
            IOobject::groupName("k", alphaRhoPhi.group()),
            this->runTime_.timeName(),
            this->mesh_,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        this->mesh_
    ),
    epsilon_
    (
        IOobject
        (
            IOobject::groupName("epsilon", alphaRhoPhi.group()),
            this->runTime_.timeName(),
            this->mesh_,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        this->mesh_
    )
{
    bound(k_, this->kMin_);
    bound(epsilon_, this->epsilonMin_);

    if (type == typeName)
    {
        this->printCoeffs(type);
    }
}
```

The constructor first uses the constructor of the base class eddyViscosity to initialize the member data inherited from that class, followed by initialization of the member data that are specific to kEpsilon, finishing with a bounding of k and epsilon and writing the model coefficients to the standard output. We see that the fields k_ and epsilon_ are constructed similar to the p field in createFields.H in the top-level solver. This will make sure that those fields are available for the turbulence model, and that they are written to disk the same way as the p field. Again, it is the turbulence model that knows which fields it needs, and here those fields are initialized according to their files in the time directory the simulation starts from.

The three functions that are called in the top-level solver simpleFoam are validate(), divDevReff(U) and correct(). We find only the correct() function in kEpsilon.C, while we have to look for the other functions in the base class(es). The correct() function is implemented as:

```
template<class BasicTurbulenceModel>
void kEpsilon<BasicTurbulenceModel>::correct()
{
 if (!this->turbulence_)
 {
 return;
 }

 // Local references
 const alphaField& alpha = this->alpha_;
 const rhoField& rho = this->rho_;
 const surfaceScalarField& alphaRhoPhi = this->alphaRhoPhi_;
 const volVectorField& U = this->U_;
 volScalarField& nut = this->nut_;
 fv::options& fvOptions(fv::options::New(this->mesh_));

 eddyViscosity<RASModel<BasicTurbulenceModel>>::correct();

 volScalarField::Internal divU
 (
 fvc::div(fvc::absolute(this->phi(), U))().v()
 );

 tmp<volTensorField> tgradU = fvc::grad(U);
 volScalarField::Internal G
 (
 this->GName(),
 nut.v()*(dev(twoSymm(tgradU().v())) && tgradU().v())
 );
 tgradU.clear();

 // Update epsilon and G at the wall
 epsilon_.boundaryFieldRef().updateCoeffs();

 // 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);
 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();
}
```

We see that it first calls the correct() function of the base class, in case there are some general things to be done. Then it discretizes and solves the epsilon_ and k_ fields, followed by relaxation and bounding of the fields. There is also a possibility to manipulate the model through the fvOptions functionality. The final line calls the function correctNut(), which is implemented in the same file and calculates the new value for the turbulent viscosity:

```
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();
}
```

The validate() function is implemented in the base class eddyViscosity ($FOAM_SRC/TurbulenceModels/turbulenceModels/eddyViscosity/eddyViscosity.C):

```
template<class BasicTurbulenceModel>
void Foam::eddyViscosity<BasicTurbulenceModel>::validate()
{
 correctNut();
}
```

I.e. it just calls the correctNut function that was discussed above. This just makes sure that the turbulent viscosity is computed correctly before starting the SIMPLE loop in simpleFoam.

The divDevReff(U) function is neither implemented in the eddyViscosity class, so we have a look in its base class linearViscousStress, located in $FOAM_SRC/TurbulenceModels/turbulenceModels/linearViscousStress. We can't find it there either, so we look in its base class, which is referred to as BasicTurbulenceModel (which we know is

IncompressibleTurbulenceModel<transportModel> in our case). We find in $FOAM_SRC/TurbulenceModels/incompressible/IncompressibleTurbulenceModel/IncompressibleTurbulenceModel.C:

```
template<class TransportModel>
Foam::tmp<Foam::fvVectorMatrix>
Foam::IncompressibleTurbulenceModel<TransportModel>::divDevReff
(
 volVectorField& U
) const
{
 return divDevRhoReff(U);
}
```

This in turn calls the divDevRhoReff(U) function, assuming that we are currently in the kEpsilon class. That function can't be found there, and neither in the eddyViscosity class, but in the linearViscousStress class ($FOAM_SRC/TurbulenceModels/turbulenceModels/linearViscousStress/linearViscousStress.C):

```
template<class BasicTurbulenceModel>
Foam::tmp<Foam::fvVectorMatrix>
Foam::linearViscousStress<BasicTurbulenceModel>::divDevRhoReff
(
 volVectorField& U
) const
{
 return
 (
 - fvc::div((this->alpha_*this->rho_*this->nuEff())*dev2(T(fvc::grad(U))))
 - fvm::laplacian(this->alpha_*this->rho_*this->nuEff(), U)
 );
}
```

We remember that for the incompressible case the density (rho_) was set to constant 1. The nuEff() function can NOT be found by the same search pattern as before: kEpsilon - eddyViscosity - linearViscousStress, but we recall that kEpsilon is a RASModel, and find the function in $FOAM_SRC/TurbulenceModels/turbulenceModels/RAS/RASModel/RASModel.H:

```
 //- Return the effective viscosity
 virtual tmp<volScalarField> nuEff() const
 {
 return tmp<volScalarField>
 (
 new volScalarField
 (
 IOobject::groupName("nuEff", this->alphaRhoPhi_.group()),
 this->nut() + this->nu()
 )
 );
 }
```

The nut() function seen above is implemented in $FOAM_SRC/TurbulenceModels/turbulenceModels/eddyViscosity/eddyViscosity.H:

```
 //- Return the turbulence viscosity
 virtual tmp<volScalarField> nut() const
 {
 return nut_;
 }
```

We can imagine that the nu() function returns the laminar kinematic viscosity (perhaps modified by non-Newtonian effects, from the singlePhaseTransportModel class, but we will not go there now), and that the nuEff() function thus returns the sum of the turbulent and laminar kinematic viscosities.

We see that only the things that are special for the kEpsilon class are implemented in that class. The rest is implemented in the base classes. If we are to implement a special version of the kEpsilon model we need to focus on the kEpsilon.H and kEpsilon.C files.

The kOmegaSST and kOmegaSSTSAS models

The kOmegaSST model is implemented in $FOAM_SRC/TurbulenceModels/turbulenceModels/RAS/kOmegaSST. If we look in those files we can't find any definition of the member functions. We realize that the kOmegaSST class inherits from another class (see *.H) that is named kOmegaSSTBase:

```
#include "kOmegaSSTBase.H"
...
template<class BasicTurbulenceModel>
class kOmegaSST
:
    public kOmegaSSTBase<eddyViscosity<RASModel<BasicTurbulenceModel>>>
{
```

The file kOmegaSSTBase.H is located in $FOAM_SRC/TurbulenceModels/turbulenceModels/Base/kOmegaSST, and we see that the files in that directory include a full description and implementation of the kOmegaSST model. The implementation can be compared with **the mathematical description** (written for version 1.6, so it is not exact). In particular we find a source term named Qsas. It is a preparation for the kOmegaSSTSAS model, and the preparation for the kOmegaSSTSAS model is also the reason to make a separate Base version of the kOmegaSST model.

The kOmegaSSTSAS model is located in $FOAM_SRC/TurbulenceModels/turbulenceModels/RAS/kOmegaSSTSAS. We see that the kOmegaSSTSAS class inherits from the kOmegaSST class:

```
#include "kOmegaSST.H"
...
template<class BasicTurbulenceModel>
class kOmegaSSTSAS
:
    public kOmegaSST<BasicTurbulenceModel>
{
```

The kOmegaSSTSAS sub-class adds some coefficients, adds a run-time selectable delta model, and remimplements the Qsas function. We will not go into the details at this point.

Implement your own versions of kEpsilon and kOmegaSST

At the end of this document we will implement a filtered version of the kOmegaSST model, named kOmegaSSTF. First we practice a bit by making our own versions of kEpsilon and kOmegaSST. In fact, there was a problem to only make a new version of kOmegaSST. That is likely to be because of the special design with the kOmegaSSTBase class. However, the problem disappeared if the kEpsilon model was first implemented. We will not investigate that problem further at the moment.

We start by copying and renaming the kEpsilon and kOmegaSST folders, files and classes:

```
foam
cp -r --parents src/TurbulenceModels/turbulenceModels/RAS/kEpsilon $WM_PROJECT_USER_DIR
cd $WM_PROJECT_USER_DIR/src/TurbulenceModels/turbulenceModels/RAS
mv kEpsilon mykEpsilon
cd mykEpsilon
mv kEpsilon.C mykEpsilon.C
mv kEpsilon.H mykEpsilon.H
sed -i s/kEpsilon/mykEpsilon/g mykEpsilon.*
foam
cp -r --parents src/TurbulenceModels/turbulenceModels/RAS/kOmegaSST $WM_PROJECT_USER_DIR
cd $WM_PROJECT_USER_DIR/src/TurbulenceModels/turbulenceModels/RAS
mv kOmegaSST mykOmegaSST
cd mykOmegaSST
mv kOmegaSST.C mykOmegaSST.C
mv kOmegaSST.H mykOmegaSST.H
```

At this point we want to change the name of the class in those files using the sed command. A problem is that there is a string 'kOmegaSSTBase' occurring in the files, so we need to use a sed command that will not modify those strings. Here is a sed command that omits lines that contain the word kOmegaSSTBase (and also gives an example of how to also omit lines with the words 'mykOmegaSST' and 'dummy'):

```
sed -Ei '/(kOmegaSSTBase|mykOmegaSST|dummy)/!s/kOmegaSST/mykOmegaSST/g' mykOmegaSST.*
```

Two important notes about the above command: 1: I experienced that it is important to have the flags in the order Ei and not iE, since otherwise the omission will not work. 2: According to comments in forums the command will not omit consecutive lines with those words. However, in this particular case it works.

Now we need to copy a file with macros that tell the compiler which instances of turbulence models to compile, as one of the available template options:

```
foam
cp --parents src/TurbulenceModels/incompressible/turbulentTransportModels/turbulentTransportModels.C $WM_PROJ
ECT_USER_DIR
cp -r --parents src/TurbulenceModels/incompressible/Make $WM_PROJECT_USER_DIR
cd $WM_PROJECT_USER_DIR/src/TurbulenceModels/incompressible/turbulentTransportModels
mv turbulentTransportModels.C myTurbulentTransportModels.C
```

Open myTurbulentTransportModels.C and make sure that the active lines are:

```
#include "turbulentTransportModels.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

// -------------------------------------------------------------------------- //
// Laminar models
// -------------------------------------------------------------------------- //

// -------------------------------------------------------------------------- //
// RAS models
// -------------------------------------------------------------------------- //

#include "mykEpsilon.H"
makeRASModel(mykEpsilon);

#include "mykOmegaSST.H"
makeRASModel(mykOmegaSST);

// -------------------------------------------------------------------------- //
// LES models
// -------------------------------------------------------------------------- //

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

Now we need to make sure that we can compile by modifying Make/files and Make/options. First go there:

```
cd $WM_PROJECT_USER_DIR/src/TurbulenceModels/incompressible
```

Then make sure that Make/files contains:

```
turbulentTransportModels/myTurbulentTransportModels.C
LIB = $(FOAM_USER_LIBBIN)/libmyIncompressibleTurbulenceModels
```

Then add two entries under EXE_INC and one entry under LIB_LIBS in Make/options to connect to the original library:

```
EXE_INC = \
-I../turbulenceModels/lnInclude \
-I$(LIB_SRC)/transportModels \
-I$(LIB_SRC)/finiteVolume/lnInclude \
-I$(LIB_SRC)/meshTools/lnInclude \
-I$(LIB_SRC)/TurbulenceModels/turbulenceModels/lnInclude \
-I$(LIB_SRC)/TurbulenceModels/incompressible/lnInclude

LIB_LIBS = \
-lincompressibleTransportModels \
-lturbulenceModels \
-lfiniteVolume \
-lmeshTools \
-lincompressibleTurbulenceModels
```

In order to compile we need to first create an lnInclude directory for the available turbulence models using the wmakeLnInclude command (this must be repeated if more turbulence models are added later), and the we can compile with wmake as usual:

```
wmakeLnInclude -u ../turbulenceModels
wmake
```

Now copy the simpleFoam/pitzDaily tutorial and add at the end of system/controlDict:

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

Run with the two new turbulence models and check the log-file that they are selected. Try also with a dummy entry and see that both the original and new turbulence models are available for simpleFoam in this particular case (which has the libs-entry in controlDict).

Implement the kOmegaSSTF model

We will in the rest of the document go through the basic steps of implementing a new kOmegaSST model, named kOmegaSSTF. An old description of how to implement the kOmegaSSTF model is found here. It sets a limit to the turbulent viscosity, which has a similar effect as the added source term in the kOmegaSSTSAS model. We will simply make a copy of our mykOmegaSST model (see previous section) and reimplement the correctNut(S2) function that computes the turbulent viscosity. We will as well see how the entire correct() function can be re-implemented (although we will not do any changes to it here).

We copy mykOmegaSST, rename directory, files and class names, and finally update the lnInclude directory:

```
cd $WM_PROJECT_USER_DIR/src/TurbulenceModels/turbulenceModels/RAS
cp -r mykOmegaSST kOmegaSSTF
mv kOmegaSSTF/mykOmegaSST.C kOmegaSSTF/kOmegaSSTF.C
mv kOmegaSSTF/mykOmegaSST.H kOmegaSSTF/kOmegaSSTF.H
sed -i s/mykOmegaSST/kOmegaSSTF/g kOmegaSSTF/kOmegaSSTF.*
cd $WM_PROJECT_USER_DIR/src/TurbulenceModels/incompressible
wmakeLnInclude -u ../turbulenceModels
```

Now add kOmegaSSTF the same way as mykOmegaSST in myTurbulentTransportModels.C (see previous section).

Before we compile we need to make the compiler aware that we have done modifications. The reason for this is that we did not modify any file that is listed in Make/files. We use the touch command to change the time-stamp of that file, so that the compiler will compile.

```
touch turbulentTransportModels/myTurbulentTransportModels.C
wmake
```

Notice that it is not shown in the output of the compilation that the kOmegaSSTF model is compiled, but it is. The compilation will take more time than expected when we only added one additonal turbulence model, and that is because all the templated alternatives mentioned in the above file must be recompiled. Right now it means that kEpsilon, kOmegaSST and kOmegaSSTF are all compiled.

Make sure that it works by running a fresh simpleFoam/pitzDaily case:

```
rm -r $FOAM_RUN/pitzDailykOmegaSSTF
cp -r $FOAM_TUTORIALS/incompressible/simpleFoam/pitzDaily $FOAM_RUN/pitzDailykOmegaSSTF
sed -i s/kEpsilon/kOmegaSSTF/g $FOAM_RUN/pitzDailykOmegaSSTF/constant/turbulenceProperties
blockMesh -case $FOAM_RUN/pitzDailykOmegaSSTF
echo 'libs ("libmyIncompressibleTurbulenceModels.so");' >> $FOAM_RUN/pitzDailykOmegaSSTF/system/controlDict
simpleFoam -noFunctionObjects -case $FOAM_RUN/pitzDailykOmegaSSTF >& $FOAM_RUN/pitzDailykOmegaSSTF/log&
```

Look at the start of that log file and read "Selecting RAS turbulence model kOmegaSSTF", which shows that the kOmegaSSTF model is used.

Now we will do some modifications to our new model. We will reimplement both the correct() and the correctNut(S2) functions. It should be noted that we do not have to reimplement the correct() function in this case, but it is done here as an example of how to do it if there are some moficiations that need to be done in that function. In the correctNut(S2) function we will introduce the limit of the turbulent viscosity, which is the idea of the kOmegaSSTF model (see the slides referred to earlier). The correct() function is inherited to the kOmegaSSTF sub-class from the kOmegaSSTBase sub-class (located in $FOAM_SRC/TurbulenceModels/turbulenceModels/Base/kOmegaSST). In order to reimplement it in the sub-class kOmegaSSTF we first need to add the declaration of the function in the .H file, and then we need to add the new definition of the function in the .C file.

In kOmegaSSTF.H, add at the end of the class declaration (after the destructor and before the curly bracket that is followed by a semicolon):

```
    //- Solve the turbulence equations and correct the turbulence viscosity
    virtual void correct();
```

In kOmegaSSTF.C, add at the end of the namespace RASModels (before the two curly brackets at the end), the entire correct() function of the kOmegaSSTBase.C file but change the template parameter name from BasicEddyViscosityModel to BasicTurbulenceModel (three instances), to adapt to the template parameter names in the rest of the kOmegaSSTF.C file. Also change the class name of the correct() function from kOmegaSSTBase to kOmegaSSTF (one instance). Add as well an Info statement at the beginning so that we can check that our new class it is in fact used. This is what the start of the correct() function in kOmegaSSTF.C should now look like (above modifications marked in yellow):

```
template<class BasicTurbulenceModel>
void kOmegaSSTF<BasicTurbulenceModel>::correct()
{
    if (!this->turbulence_)
    {
        return;
    }

    Info << "This is kOmegaSSTF" << endl;

    // Local references
    const alphaField& alpha = this->alpha_;
    const rhoField& rho = this->rho_;
    const surfaceScalarField& alphaRhoPhi = this->alphaRhoPhi_;
    const volVectorField& U = this->U_;
    volScalarField& nut = this->nut_;
    fv::options& fvOptions(fv::options::New(this->mesh_));

    BasicTurbulenceModel::correct();
    ...
```

Note in the above piece of code that you can also manipulate turbulence models using fvOptions, but we are not doing that here.

Compile (and be prepared that you will get lots of error messages):

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

Don't panic! The error messages are of the following types:

```
‘omega_’ was not declared in this scope
there are no arguments to ‘DomegaEff’ that depend on a template parameter, so a declaration of ‘DomegaEff’ mu
```

```
st be available
‘DomegaEff’ was not declared in this scope, and no declarations were found by argument-dependent lookup at th
e point of instantiatio
```

The reason we get those messages is that those objects and functions are not declared in our sub-class, but in some base-class. The current class does not know about those objects and functions, but the object belonging to the class does. We can therefore use the object belonging to the class to reach those objects and functions. This is what is done in the above code, where e.g. a local reference object alphaField is constructed as:

```
    const alphaField& alpha = this->alpha_;
```

The word "this" is a pointer to the object that belongs to our sub-class, and we here make alphaField refer to the object alpha_ of the object that belongs to our sub-class. We simply have to do the same for all the objects and functions that were mentioned in the error message. Make sure to run the following commands **only once(!!!)**:

```
cd $WM_PROJECT_USER_DIR/src/TurbulenceModels
sed -i s/"omega_"/"this->omega_"/g turbulenceModels/RAS/kOmegaSSTF/kOmegaSSTF.C
sed -i s/"alphaOmega2_"/"this->alphaOmega2_"/g turbulenceModels/RAS/kOmegaSSTF/kOmegaSSTF.C
sed -i s/"k_"/"this->k_"/g turbulenceModels/RAS/kOmegaSSTF/kOmegaSSTF.C
sed -i s/"DomegaEff(F1)"/"this->DomegaEff(F1)"/g turbulenceModels/RAS/kOmegaSSTF/kOmegaSSTF.C
sed -i s/"GbyNu(GbyNu0, F23(), S2())"/"this->GbyNu(GbyNu0, F23(), S2())"/g turbulenceModels/RAS/kOmegaSSTF/kO
megaSSTF.C
sed -i s/"Qsas(S2(), gamma, beta)"/"this->Qsas(S2(), gamma, beta)"/g turbulenceModels/RAS/kOmegaSSTF/kOmegaSS
TF.C
sed -i s/"omegaSource()"/"this->omegaSource()"/g turbulenceModels/RAS/kOmegaSSTF/kOmegaSSTF.C
sed -i s/"DkEff(F1)"/"this->DkEff(F1)"/g turbulenceModels/RAS/kOmegaSSTF/kOmegaSSTF.C
sed -i s/"Pk(G)"/"this->Pk(G)"/g turbulenceModels/RAS/kOmegaSSTF/kOmegaSSTF.C
sed -i s/"epsilonByk(F1, tgradU())"/"this->epsilonByk(F1, tgradU())"/g turbulenceModels/RAS/kOmegaSSTF/kOmega
SSTF.C
sed -i s/"kSource()"/"this->kSource()"/g turbulenceModels/RAS/kOmegaSSTF/kOmegaSSTF.C
sed -i s/"DomegaEff(F1), omega_)"/"this->DomegaEff(F1), omega_)"/g turbulenceModels/RAS/kOmegaSSTF/kOmegaSST
F.C
sed -i s/"epsilonByk(F1, tgradU()), k_)"/"this->epsilonByk(F1, tgradU()), k_)"/g turbulenceModels/RAS/kOmegaS
STF/kOmegaSSTF.C
sed -i s/"omegaInf_"/"this->omegaInf_"/g turbulenceModels/RAS/kOmegaSSTF/kOmegaSSTF.C
sed -i s/"betaStar_"/"this->betaStar_"/g turbulenceModels/RAS/kOmegaSSTF/kOmegaSSTF.C
sed -i s/"kInf_"/"this->kInf_"/g turbulenceModels/RAS/kOmegaSSTF/kOmegaSSTF.C
```

Try compiling again, and it should work:

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

Test using the simpleFoam/pitzDailykOmegaSSTF case:

```
simpleFoam -noFunctionObjects -case $FOAM_RUN/pitzDailykOmegaSSTF >& $FOAM_RUN/pitzDailykOmegaSSTF/log&
```

Look before the solution of omega at each time step, and make sure that the Info statement is written.

Now we want to change how the turbulent viscosity is corrected. We see at the end of the correct() function that the turbulent viscosity is corrected by a call to a function named correctNut(S2). That function is implemented at the beginning of the kOmegaSSTF.C file, as:

```
template<class BasicTurbulenceModel>
void kOmegaSSTF<BasicTurbulenceModel>::correctNut(const volScalarField& S2)
{
    // Correct the turbulence viscosity
     kOmegaSSTBase<eddyViscosity<RASModel<BasicTurbulenceModel>>>::correctNut
    (
         S2
     );

     // Correct the turbulence thermal diffusivity
    BasicTurbulenceModel::correctNut();
}
```

There are calls to two functions here. The second one is to a function implemented in incompressible/incompressibleTurbulenceModel.H:

```
        //- ***HGW Temporary function to be removed when the run-time selectable
        // thermal transport layer is complete
        virtual void correctNut()
        {}
```

The function does not do anything, so we can leave that call as it is.

The first function call in correctNut(S2), above, is to a function with the same name in the base class kOmegaSSTBase. It is implemented in turbulenceModels/Base/kOmegaSST/kOmegaSSTBase.C:

```
template<class BasicEddyViscosityModel>
void kOmegaSSTBase<BasicEddyViscosityModel>::correctNut
(
    const volScalarField& S2
)
{
    // Correct the turbulence viscosity
    this->nut_ = a1_*k_/max(a1_*omega_, b1_*F23()*sqrt(S2));
    this->nut_.correctBoundaryConditions();
    fv::options::New(this->mesh_).correct(this->nut_);
}
```

In order to reimplement this in the sub-class we can simply avoid calling the base class function and instead do those operations in the sub-class function. The correctNut(S2) function in kOmegaSSTF.C should then read (where we have added "this->" as discussed above):

```
template<class BasicTurbulenceModel>
void kOmegaSSTF<BasicTurbulenceModel>::correctNut(const volScalarField& S2)
{
    // Correct the turbulence viscosity
    this->nut_ = this->a1_*this->k_/max(this->a1_*this->omega_, this->b1_*this->F23()*sqrt(S2));
    this->nut_.correctBoundaryConditions();
    fv::options::New(this->mesh_).correct(this->nut_);

    // Correct the turbulence thermal diffusivity
    BasicTurbulenceModel::correctNut();
}
```

Compile and test:

```
touch turbulentTransportModels/turbulentTransportModels.C
wmake
simpleFoam -noFunctionObjects -case $FOAM_RUN/pitzDailykOmegaSSTF >& $FOAM_RUN/pitzDailykOmegaSSTF/log&
```

It runs, but the implementation is still the same as before. Now we will reimplement the first line that we added above, i.e.:

```
    this->nut_ = this->a1_*this->k_/max(this->a1_*this->omega_, this->b1_*this->F23()*sqrt(S2));
```

to (See the slides regarding the kOmegaSSTF model linked to above. The implementation below could be improved!!!):

```
    // Compute Filter
    scalar alph = 3.0; // Should be in a dictionary
    scalarField Lt = sqrt(this->k_)/(this->betaStar_*this->omega_);
    scalarField lt = alph*Foam::max(Foam::pow(this->mesh_.V().field(), 1.0/3.0),
                     (mag(this->U_)*this->runTime_.deltaT())->internalField());
    // Recalculate viscosity
    this->nut_.primitiveFieldRef() = Foam::min(Foam::pow(lt/Lt, 4.0/3.0), 1.0)*
       (this->a1_*this->k_/max(this->a1_*this->omega_, this->b1_*this->F23()*sqrt(S2)))
       ->internalField();
```

Compile and check that it goes through:

```
touch turbulentTransportModels/turbulentTransportModels.C
wmake
```

A note is that in previous years the last line started with "this->nut_.internalField() ==". In recent versions of OpenFOAM this worked to compile, but gave a zero turbulent viscosity, since internalField() returns a const-reference to the dimensioned internal field.

Run the kOmegaSSTF model with pimpleFoam for the pitzDaily case

The kOmegaSSTF model is supposed to run in unsteady mode, so that it can resolve as much as possible of the turbulence. We therefore have to switch from the steady-state simpleFoam solver to the unsteady pimpleFoam solver. For that we need to manipulate the pitzDaily case:

```
run
cd pitzDailykOmegaSSTF
foamListTimes -rm
```

Modify the files in the system directory, for use with pimpleFoam:

```
cp $FOAM_TUTORIALS/incompressible/pimpleFoam/RAS/TJunction/system/{fvSolution,fvSchemes} system
sed -i s/epsilon/omega/g system/fvSchemes
echo "wallDist { method meshWave; }" >> system/fvSchemes
sed -i s/epsilon/omega/g system/fvSolution
sed -i s/simpleFoam/pimpleFoam/g system/controlDict
sed -i s/2000/0.3/g system/controlDict;
sed -i s/"1;"/"0.0001;"/g system/controlDict
sed -i s/"writeCompression off"/"writeCompression on"/g system/controlDict
```

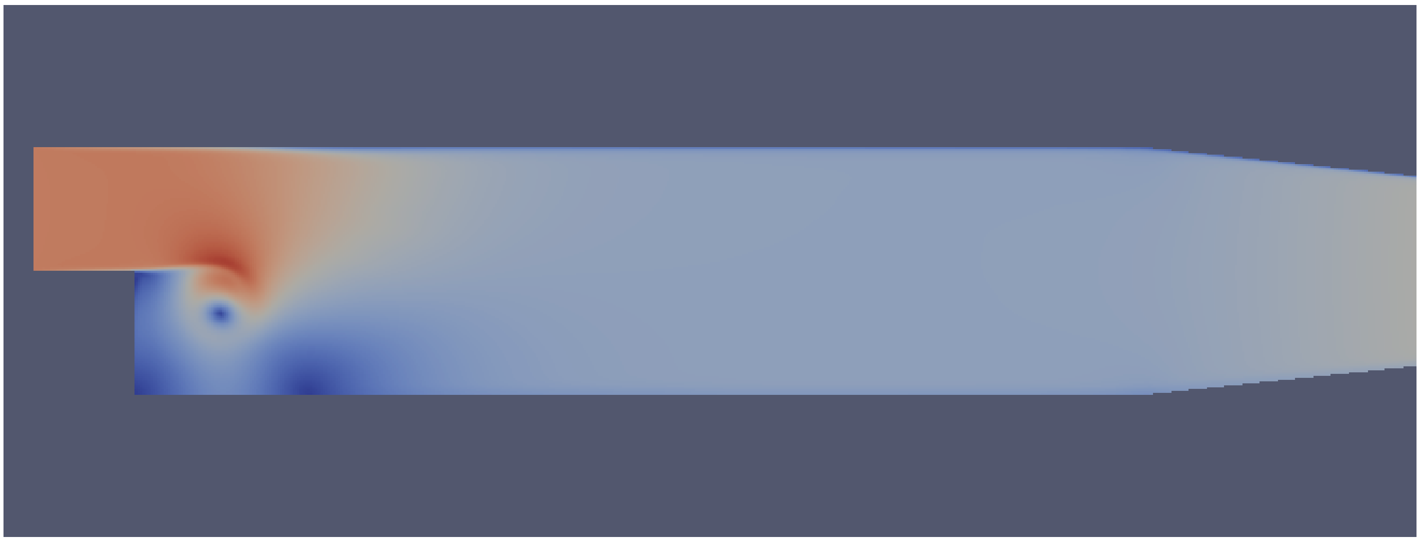
```
echo "adjustTimeStep no;" >> system/controlDict
echo "maxCo           5;" >> system/controlDict
```

Run the case and watch in paraFoam how the flow becomes highly unsteady:

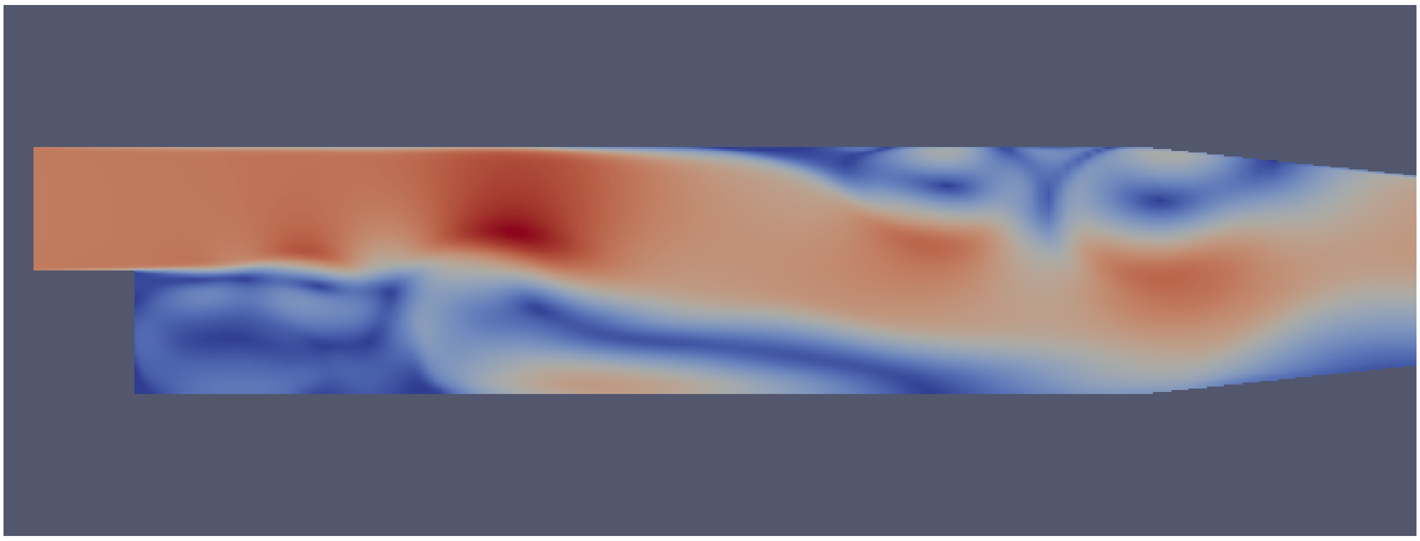
```
pimpleFoam -noFunctionObjects -case $FOAM_RUN/pitzDailykOmegaSSTF >& $FOAM_RUN/pitzDailykOmegaSSTF/log&
```

Visualize using paraFoam.

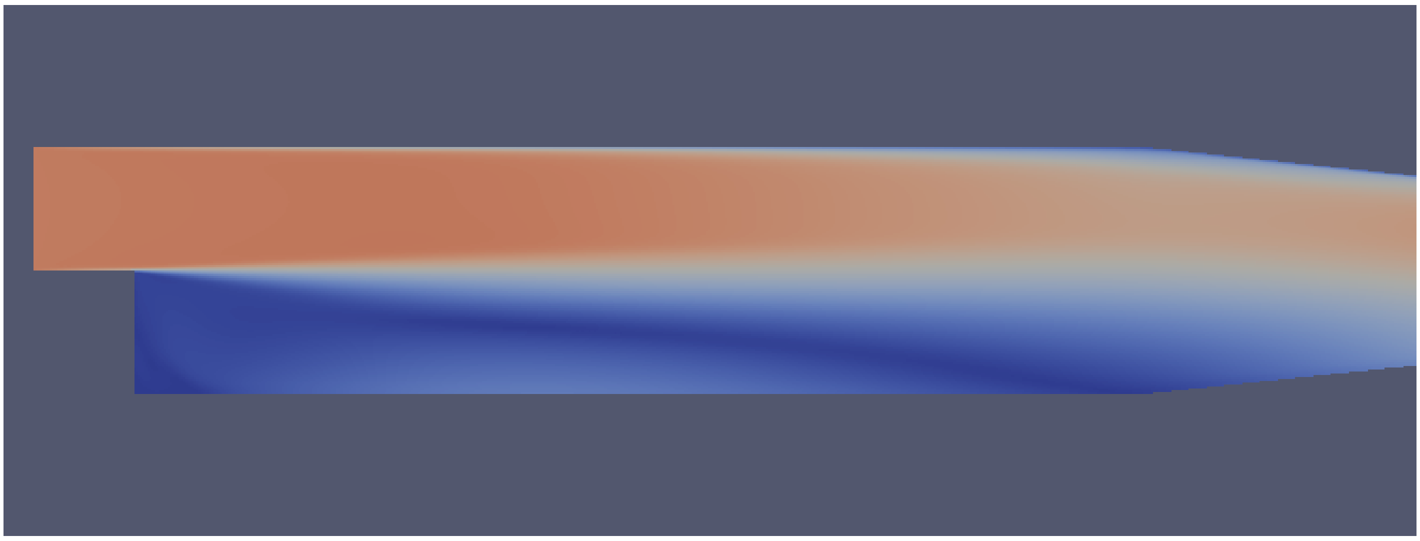
Movie:



Velocity at final time step with kOmegaSSTF:



Velocity at final time step with kOmegaSST (using the same solver and case):



License