CFD WITH OPENSOURCE SOFTWARE

A COURSE AT CHALMERS UNIVERSITY OF TECHNOLOGY
TAUGHT BY HÅKAN NILSSON

Project work:

A boxTurb16 and dnsFoam tutorial

Developed for OpenFOAM-2.0.x

Author:
MARTIN DE MARÉ

Peer reviewed by:
OLLE PERTINEN
EHSAN YASARI

November 6th, 2011

Table of Contents

1 Introduction

This document contatins a description of the predefined OpenFOAM case `boxTurb16` in which the solver `dnsFoam` is used. It is assumed that the reader have run OpenFOAM cases before and know how to visualize the result in `paraFoam` etc.[1]

The solver `dnsFoam` implements Direct Numerical Simulation, DNS, which means that it does not make use of a subscale turbulence model but attempts to resolve all scales down to the scales where energy is dissipated as heat[2]. This puts a requirement on the spacial resolution, Δx, to be less then the Kolmogorov micro scale for length, $\eta=(\frac{(\mu/\rho)^3}{\varepsilon})^{1/4}$, where μ is the viscosity, ρ is the density and and ε is the rate of kinetic energy dissipation per unit mass. A fine spacial resolution means that the time step, Δt, which is given by the Courant number is relatively short. These constraints currently makes DNS too heavy for most applications, but in order to investigate turbulence DNS is still a useful tool.

In the first chapter after the introduction the case `boxTurb16` is run. In the subsequent chapter the functionality of `boxTurb16` is described in more detail. The focus of that chapter is to highlight the configurable parts of the code as well as to give a brief introduction to some of the key concepts in order to facilitate further exploration of the code by the reader. In the final chapter, a modification of the case `boxTurb16` is attempted. An appendix has also been included, which provides a first introduction to OpenFOAM equations.

1 If not, the OpenFOAM User Guide is a good place to start.

2 Note to the reader: There are other solvers OpenFoam that lack a subscale model, e.g. icoFoam, and in principle the same restrictions apply to these solvers as well.

2 Running boxTurb16

The case `boxTurb16` is a standard OpenFOAM tutorial which means it is straightforward to run:

Copy the directory `$FOAM_TUTORIALS/DNS/dnsFoam/boxTurb16` to an optional location in your home folder with for example

```
cp -r $FOAM_TUTORIALS/DNS/dnsFoam/boxTurb16 $FOAM_RUN
```

Enter the directory and run

```
./Allrun
```

or run in sequence:

```
blockMesh
```

which creates a mesh according to the content of the file constant/polyMesh/blockMeshDict .

```
boxTurb
```

which creates a random initial 3 dimensional flow field consistent with the current understanding of isotropic turbulence.

```
dnsFoam
```

which solves the 3 dimensional flow field with DNS methodology. In order to compensate for dissipation of the turbulence a small amount of random fluctuations are introduced in every time step.

```
enstrophy
```

A post-processing step where the enstophy is calculated from the the velocity field in each time step.

3 The Functionality of boxTurb16 in More Detail

3.1 blockMesh

`blockMesh` creates a mesh according to the content of the file `constant/polyMesh/blockMeshDict` . As is suggested in name of the case the boxturb16 case is defined as 16-by-16-by-16 box of cells. Moreover the boundary conditions are defined as cyclic in all 3 directions which is defined in `blockMeshDict` as:

```
    boundary
    (
        patch0_half0
        {
            type cyclic;
            neighbourPatch patch0_half1;
            faces
            (
                (0 3 2 1)
            );
        }
        patch0_half1
        {
            type cyclic;
            neighbourPatch patch0_half0;
            faces
            (
                (4 5 6 7)
            );
        }
        patch1_half0
        {
            type cyclic;
            neighbourPatch patch1_half1;
            faces
            (
                (0 4 7 3)
            );
    ...
```

For more information regarding `blockMesh` and `blockMeshDict` please see the OpenFOAM online documentation[3].

3 For example http://www.openfoam.com/docs/user/blockMesh.php .

3.2 boxTurb

`boxTurb` is a pre-processing utility found in `$FOAM_APP/utilities/preProcessing/boxTurb` and it generates an initial field, U, consistent with the mesh produced by `blockMesh`. Actually most of the interesting code is found in `turbGen.C` which is found in `$FOAM_SRC/randomProcesses/turbulence/`. In order to understand the functionality of `boxTurb` and `turbGen.C` let's first give some background on the theory of isotropic turbulence.

Isotropic turbulence is typically understood as a stationary process, which means that it is a random process whose statistic properties does not change when shifted in time or space. It can be shown that the coefficients for *different* wave numbers of the *discrete Fourier transform* of such a process are uncorrelated. This property makes it suitable to work in Fourier space when generating a random turbulence field since each wave number $k=[k_1, k_2, k_3]$ can be handled separately.

The field we are interested in in this case is three-dimensional, for example $U=[u_1, u_2, u_3]$. The most straightforward way to apply the Fourier transform to a three-dimensional field is to transform each component separately; $\hat{u}_1(k)$, $\hat{u}_2(k)$ and $\hat{u}_3(k)$ being the Fourier transform of u_1, u_2 u_3 respectively.

The coefficients of these three Fourier transforms can be aggregated for each wave number to form the complex vector $\hat{U}(k)=[\hat{u}_1(k), \hat{u}_2(k), \hat{u}_3(k)]$. It turns out that the components of this wave number-dependent vector are far from independent. For an incompressible field for example, the equation $\nabla \cdot U=0$ translates to $k_1 \cdot \hat{u}_1 + k_2 \cdot \hat{u}_2 + k_3 \cdot \hat{u}_3 = k \cdot \hat{U} = 0$ in Fourier space. That means that the generated vector in Fourier space should be orthogonal to the wave number vector. One way of finding a vector that is orthogonal to k is to make use of the fact that the cross-product of any vector with k fulfills this criteria, and it turns out this is exactly the trick employed in `turbGen.C` which is seen in the following code:

```
    s = K ^ s;
    s = s/(mag(s) + 1.0e-20);
    s = Ek(Ea, k0, mag(K))*s;
```

In the above code the random vector s is cross-multiplied with k, then normalized, and finally given the amplitude `Ek(Ea, k0, mag(K))`. This amplitude function is implemented in `Ek.H` (which is included in `turbGen.C` and located in the same location as `turbGen.C)` as $Ek(Ea, k_0, |k|) = Ea*(\frac{|k|}{k_0})^4 e^{-2(\frac{|k|}{k_0})^2}$ and the two variables Ea and k_0 are configurable in `constant/boxTurbDict` .

The last step of `turbGen.C` is to multiply each component in the vector, s, with a random phase before using the inverse Fourier transform to transforms the field back to the real domain as seen in

```
    fft::reverseTransform
    (
          ComplexField(cos(constant::mathematical::twoPi*rndPhases)*s,
          sin(constant::mathematical::twoPi*rndPhases)*s),
          K.nn()
     )
```

where the variable `rndPhases` holds random numbers between 0 and 1, and `K.nn()` returns the dimensions of the wave number mesh. In order to have a true random field it in Fourier space it could be argued that not only the phase but also the amplitude should be stochastic. However as seen, in this implementation only the phase is stochastic.

For more information regarding turbulence see dedicated such literature.[4]

4 For example *Turbulence in the Athmosphere* by John C. Wyngaard.

3.3 dnsFoam

It was noted earlier that energy is constantly converted to heat through the diffusion term $\mu \Delta U$. The flow will thus eventually return to rest unless something is done to keep the turbulence alive. In `dnsFoam` energy is constantly introduced in the form of a small random body force, which is generated in the following code, from `dnsFoam.C` (which is located `$FOAM_APP/solvers/DNS/dnsFoam`):

```
        force.internalField() = ReImSum
        (
            fft::reverseTransform
            (
                K/(mag(K) + 1.0e-6) ^ forceGen.newField(), K.nn()
            )
        );
```

It is again worth noting that the cross product, `^`, is used to make the generated field incompressible. `K.nn()` returns the dimensions of the wave number mesh.

The object `forceGen` in the above code is declared and initialized in `readTurbulenceProperties.H` and is of the class `UOprocess`. In the function `newfield` in `Uoprocess.C` (which is found in `$FOAM_SRC/randomProcesses/processes/`) the following code is found:

```
    forAll(UOfield, i)
    {
        if ((sqrK = magSqr(K[i])) < sqrKupper && sqrK > sqrKlower)
        {
            count++;
            UOfield[i] =
                (1.0 - Alpha*DeltaT)*UOfield[i]
              + Scale*Sigma*WeinerProcess();
        }
    }
```

In the above code it is possible to make out that energy is added randomly using a Wiener process, but only for wave numbers with a magnitude between `sqrKlower` and `sqrKupper`. `Sigma` is a measure of how much energy that is added in total to the field and `Alpha` has to do with the life time of each contribution to the force field as the contributions accumulate in time. `Sigma`, `Alpha`, `Kupper` and `Klower` are all configurable in the file `constant/turbulenceProperties`.

The definition and solving of the differential equation in dnsFoam

```
        fvVectorMatrix UEqn
        (
            fvm::ddt(U)
          + fvm::div(phi, U)
          - fvm::laplacian(nu, U)
         ==
            force
        );
        solve(UEqn == -fvc::grad(p));
```

may seem a bit confusing at first since it gives the impression that we are trying to solve a equation containing two equalities (which would make absolutely no sense mathematically). Reading the definition of `operator==` in `fvMatrix.C` however and comparing it to the implementation of the `operator-` in the same file, makes it immediately evident that the two operations are identical. Hence we are actually solving the equation given by

```
       fvVectorMatrix UEqn
        (
            fvm::ddt(U)
          + fvm::div(phi, U)
          - fvm::laplacian(nu, U)
          - force
        );
        solve(UEqn == -fvc::grad(p));
```

which makes much more sense.[5]

dnsFoam use the PISO algorithm to calculate the pressure in order to ensure incompressibility of the solution.

3.4 enstrophy

Enstrophy is a quantity related to the turbulent kinetic energy of the flow and for incompressible flows the the is given by the square of the vorticity which is exactly how it is calculated in `enstrophy.C` (which is found in `/utilities/postProcessing/velocityField/enstrophy/`) as seen in the following code:

```
        Info<< "    Calculating enstrophy" << endl;
        volScalarField enstrophy
        (
            IOobject
            (
                "enstrophy",
                runTime.timeName(),
                mesh,
                IOobject::NO_READ
            ),
            0.5*magSqr(fvc::curl(U))
        );
```

5 For more details on equation definitions in OpenFOAM see the Appendix of this document.

4 Adding Shear to boxTurb16

4.1 Objective

The objective of this modification is to illustrate how shear generated turbulence can be studied using Direct Numerical Simulation, DNS. Shear generated turbulence is not isotropic so much of the theory in the previous chapter does not apply to this kind of turbulence.

The shear in this example is obtained by replacing the cyclical boundary conditions for two of the patches with moving walls with opposing speeds. We then obtain so called Couette flow.

Secondly we want the turbulence to be shear generated. That means we should remove the added body force discussed in detail in section 2.3.

4.2 Changing Boundary Conditions

The shear in this example is obtained by replacing the cyclical boundary conditions for two of the patches with moving walls with opposing speeds. In `constant/polyMesh/blockMeshDict` `patch0_half0` and `patch0_half1` should be replaced by the highlighted part of the following code:

```
boundary
(

    movingWall_bottom
    {
        type wall;
        faces
        (
            (0 3 2 1)
        );
    }
    movingWall_top
    {
        type wall;
        faces
        (
            (4 5 6 7)
        );
    }
    patch1_half0
    {
        type cyclic;
        neighbourPatch patch1_half1;
        faces
        (
            (0 4 7 3)
            );
        }
    ...
```

In the initial condition for U, defined in `0/U`, change the boundary condition of `patch0_half0` and `patch0_half1` to the highlighted part of the below code

```
    boundaryField
    {
        movingWall_bottom
        {
            type            fixedValue;
            value           uniform (1 0 0);
        }
        movingWall_top
        {
            type            fixedValue;
            value           uniform (-1 0 0);
        }
        patch1_half0
        {
            type            cyclic;
        }
```

and make the same kind of change (replacing `patch0_half0` and `patch0_half1`) in `0/p`

```
    boundaryField
    {
        movingWall_bottom
        {
            type            zeroGradient;
        }
        movingWall_top
        {
            type            zeroGradient;
        }
        patch1_half0
        {
            type            cyclic;
        }
```

4.3 Removing the Body Force Source Term[6]

First copy the `$FOAM_APP/solvers/DNS/dnsFoam` to and optional location in your home directory using for example:

```
  cp -r $FOAM_APP/solvers/DNS/dnsFoam $FOAM_RUN
```

Edit the `dnsFoam.C` file commenting the force source in the equation definition:

```
       fvVectorMatrix UEqn
        (
            fvm::ddt(U)
          + fvm::div(phi, U)
          - fvm::laplacian(nu, U)
         // ==
         // force
        );
        solve(UEqn == -fvc::grad(p));
```

Now edit the file `Make/files` in order to get

```
        dnsFoam.C
        EXE = $(FOAM_USER_APPBIN)/dnsFoamNoSource
```

and compile by running `wmake`

4.4 Running the Modified Case

The following commands runs our modified case:

```
    blockMesh

    boxTurb

    dnsFoamNoSource

    enstrophy
```

Visualize the result in paraFoam, for example.

6 The author has been made aware that this modification gives `dnsFoam` very similar functionality to the `icoFoam`

Appendix: More on the Equation Definition in OpenFOAM

Section 2.4 of the OpenFOAM Programmers guide 2.0.0[7] begins with the following introduction to equation discretization:

> "*Equation discretisation converts the PDEs into a set of algebraic equations that are commonly expressed in matrix form as:*
>
> ***[A] [x] = [b]***
>
> *where [A] is a square matrix, [x] is the column vector of dependent variable and [b] is the source vector. The description of [x] and [b] as 'vectors' comes from matrix terminology rather than being a precise description of what they truly are: a list of values defined at locations in the geometry, i.e. a geometricField<Type>, or more specifically a volField<Type> when using FV discretisation.*
>
> ***[A] is a list of coefficients of a set of algebraic equations, and cannot be described as a geometricField<Type>. It is therefore given a class of its own: fvMatrix.*** *fvMatrix<Type> is created through discretisation of a geometric<Type>Field and therefore inherits the <Type>. It supports many of the standard algebraic matrix operations of addition +, subtraction - and multiplication *. "*

It turns out that fvMatrix holds a little more information than what is implied by the above text. Consider for example the following code from `dnsFoam.C`:

```
fvScalarMatrix pEqn
 (
     fvm::laplacian(rAU, p) == fvc::div(phi)
 );
 pEqn.solve();
```

It is evident from the above code that the `pEqn` (which is of a subclass of `fvMatrix`) must hold not only **[A]** but most also contain the right hand side, **[b]**. Actually it is also clear that the result **[x]** must also be stored within `pEqn` somewhere since the operation `pEqn.solve()` does not have an output variable in the above code.

Due to the lack of comments in the code it is hard to be absolutely confident, but a qualified guess is the following (both variables and functions were found in `fvMatrix.H`):

[b] is stored in the member variable `source_` and is accessed by a call to the function `source()`

[x] is stored in the member variable `psi_` and is accessed by a call to the function `psi()`

7 Also available at http://www.foamcfd.org/Nabla/guides/ProgrammersGuidese9.html

The Programmers Guide goes on to say:

> *“Each term in a PDE is represented individually in OpenFOAM code using the classes of static functions finiteVolumeMethod and finiteVolumeCalculus, abbreviated by a typedef to fvm and fvc respectively. fvm and fvc contain static functions, representing differential operators, e.g. ∇2 , ∇ • and ∂/∂t, that discretise geometricField<Type>s. The purpose of defining these functions within two classes, fvm and fvc, rather than one, is to distinguish:*
>
> - *functions of* ***fvm*** *that calculate implicit derivatives of and return an* ***fvMatrix<Type>***
> - *some functions of* ***fvc*** *that calculate explicit derivatives and other explicit calculations, returning a* ***geometricField<Type>****. ”*

Let's apply this new knowledge to the above example:

```
fvScalarMatrix pEqn
 (
     fvm::laplacian(rAU, p) == fvc::div(phi)
);
```

For the above code to work,we would expect there to be a definition of the `operator==` that takes a `fvMatrix<Type>` as its first input and a `geometricField<Type>` as its second input. In `fvMatrix.C` (which is located in `$FOAM_SRC/finiteVolume/fvMatrices/fvMatrix/`) we will find something similar to what we are looking for:

```
template<class Type>
Foam::tmp<Foam::fvMatrix<Type> > Foam::operator==
(
    const fvMatrix<Type>& A,
    const DimensionedField<Type, volMesh>& su
)
{
    checkMethod(A, su, "==");
    tmp<fvMatrix<Type> > tC(new fvMatrix<Type>(A));
    tC().source() += su.mesh().V()*su.field();
    return tC;
}
```

In the above code we can see that the second input `su` is added to the `source_` member of the variable `tC` which is of class `fvMatrix<Type>`, just as expected. The second input isn't of `geometricField<Type>` as we would have hoped for, but of a parent class `DimensionedField<Type, volMesh>` of `geometricField<Type>`. Let's assume for the sake of the argument that we have found the right definition.

In order to check that we have this figured out, let's also check the implementation of the `operator+` again when the first object is of class `fvMatrix<Type>` and the second is of class `geometricField<Type>`. We would then expect the `geometricField<Type>` to instead be subtracted from the `source_` member in question:

```
        template<class Type>
        Foam::tmp<Foam::fvMatrix<Type> > Foam::operator+
        (
            const fvMatrix<Type>& A,
            const DimensionedField<Type, volMesh>& su
        )
        {
            checkMethod(A, su, "+");
            tmp<fvMatrix<Type> > tC(new fvMatrix<Type>(A));
            tC().source() -= su.mesh().V()*su.field();
            return tC;
        }
```

As seen, the above code does not let us down (again with the hope that we have found the right definiton).

Adding two `fvMatrix<Type>` (or two `geometricField<Type>`) is much more intuitive, so in conclusion we would expect there to be definitions to cover four possible cases[8]

```
      fvMatrix<Type> + fvMatrix<Type>
      fvMatrix<Type> + geometricField<Type>
      geometricField<Type> + fvMatrix<Type>
      geometricField<Type> + geometricField<Type>
```

And the same goes for any operator that is allowed in these OpenFOAM equations. Thus it should now be clear how the OpenFOAM interprets an equation like the one found in `dnsFoam.C`:

```
          fvVectorMatrix UEqn
          (
              fvm::ddt(U)
            + fvm::div(phi, U)
            - fvm::laplacian(nu, U)
           ==
              force
          );
          solve(UEqn == -fvc::grad(p));
```

even though it, confusingly enough, contains not one, but two equalities!

8 The author must admit he hasn't yet found where the last two variants are implemented