# Tutorial 7: LES and the Taylor-Green Vortex

## Learning Outcomes:

Large Eddy simulation, properties of the Taylor-Green Vortex, Post-processing with OpenFOAM, coding in OpenFOAM and `functionObjects`.

## VII.1 Introduction

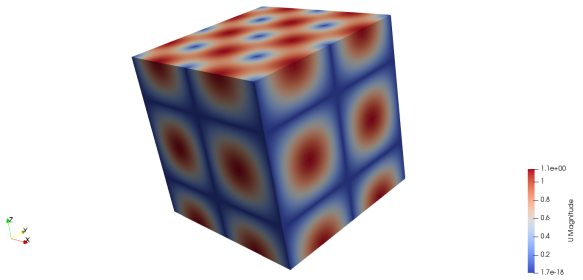


Figure 1: Initial velocity field for Taylor-Green vortexes.

This tutorial covers a number of aspects of OpenFOAM and introduces a common canonical case for turbulence simulation; that of the Taylor-Green vortex. This is a hypothetical flow problem which starts from the flow field

$$\begin{aligned} u_x &= U_0 \sin(kx)\cos(ky)\cos(kz) \\ u_y &= -U_0 \cos(kx)\sin(ky)\cos(kz) \\ u_z &= 0 \end{aligned}$$

which gives a very regular set of vortices as shown in figure 1. Obviously this cannot be set up experimentally (it is an infinite array of vortices) but computationally we can create this in a cubic box with cyclic or symmetry boundary conditions on the six faces. The vortices interact with each other and rapidly break up into turbulence, which (since there is no driving force to maintain the turbulence) decays to quiescence. This process has been extensively studied with DNS and serves as a useful test case for comparing LES codes. This tutorial sets up a standard Taylor-Green vortex which you can investigate using different LES models; it also introduces various `functionObjects` which are sections of code which can be added to OpenFOAM at runtime by inclusion into the `controlDict` dictionary, and can be used to modify or add to the behaviour of the precompiled solvers. Some of these can also be run independently through the *Command Line Interface* or CLI for postprocessing.

## VII.2 Tutorial files

The tutorial directory contains a case directory `taylorGreen` which is a fully set-up Taylor-Green vortex simulation. This will run as it stands (follow the instructions below); but there are various possible modifications eg. to compare different SGS models. The initial conditions for the flow have to be set up, and the tutorial directory also contains an OpenFOAM program `tgInitialise` which compiles to give an app to configure the velocity field. Go into this directory and type

```
wmake
```

to start the compilation. This will create an OpenFOAM app `tgInitialise` which can be used in the setup process.

Now go into the `taylorGreen` case directory and type `blockMesh`. This will create a $32 \times 32 \times 32$ cubic domain with cyclic boundary conditions for the simulation; the dimensions of the cube are $2\pi \times 2\pi \times 2\pi$. (You should check this in `blockMeshDict`). Copy the `0.orig` timestep directory to `0`, then run `tgInitialise` to set up the initial conditions. After that, run `pimpleFoam` to simulate the decay of the Taylor-Green vortex. It is worth saving the output of this to a log file, and the commands :

```
pimpleFoam > log &
tail -f log
```

will do this and at the same time write the output to the screen for inspection (the `tail` command does this).

The way the case is set up, in addition to the usual flow fields, certain additional post-processing information is being generated. You will notice that running `pimpleFoam` has generated two new directories, `dynamicCode` and `postProcessing`. The first of these will be discussed below; the second contains sampling data generated by `functionObjects` during the run. If you go into `postProcessing`, there are two sub-directories, `probes`, which contains sampled values of the pressure and velocity at the centre of the box, and `volFieldValue1`, which contains domain-averaged values of pressure and velocity; in both cases these are reported against time throughout the simulation. Details of how this is done are provided below.

More relevant than this would be the total kinetic energy in the domain; monitoring this shows the decay of the turbulence in the domain. The total kinetic energy in the simulation is given by the expression

$$E_t = \frac{1}{2}\underline{\overline{u}}^2 + k$$

where $\underline{\overline{u}}$ is the grid scale flow and $k$ the sub-grid scale kinetic energy (in SGS models which use this). This has been worked out by another `functionObject` (also described below) and output to the `log` file. If you examine this closely you will see for each timestep the line

```
Et: 0.005 0.16616
```

(the actual numbers will vary; this was just the first line I came to). This reports the time (first number) and total kinetic energy (second number). Obviously copying this all out of the log file by hand would be a pain, but there is a line of Linux magic to help here :

```
grep ‘‘Et:’’ log | awk ’print $2 $3’ > Et
```

which will do this automagically, creating an output file `Et`. If you are fluent with Linux you may recognise how this works.[1]

Finally, we may want to visualise the actual vortex 'ropes' in the flow. There are various ways of doing this, all of which involve calculating further quantities in the flow – a postprocessing task. Most often we use the following quantities :

$\frac{1}{2}|\nabla \times \underline{u}|^2$**:** the Enstrophy

$Q$**:** the 2nd invariant of the velocity gradient tensor

$\lambda_2$**:** the 2nd largest eigenvector of the sum of the square of the symmetrical and anti-symmetrical parts of the velocity gradient tensor

These quantities (and quite a few others) can be calculated after the run using the command `postProcess` as part of the Command Line Interface. If you type

```
postProcess -func enstrophy
```

then the code will read in the appropriate fields from each saved timestep in turn, calculate the enstrophy function and write it out to that timestep. You can then visualise this quantity in `paraFoam` in the usual way (select the `Contour` module with an appropriate value to display an isosurface of the enstrophy as a good way to visualise the vortex). Substitute `Q` or `lambda2` for `enstrophy` as appropriate. Meanwhile the command

```
postProcess -list
```

will list out all the postprocessing `functionObjects` that `postProcess` can access.

| [**Q.VII.1**] Read up online about enstrophy, $Q$ and $\lambda_2$ |
|---|

[1] `grep` is short for General Regular ExPression; it is a command line utility in Linux which prints out lines in a file (here, `log`) that match a particular string of characters (here `Et:`). The output then passes to `awk`, which runs a short programme on each line. The `awk` language is fairly complete and can do many things; in this case it prints out the 2nd and 3rd items in each line, which in this case happen to be the time and total kinetic energy.

## VII.3 tgInitialise

The main body of `tgInitialise` looks like this :

```
volVectorField U
(
    IOobject
    (
        "U",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);

dimensionedScalar unit("unit", dimensionSet(0,1,0,0,0,0,0),1.0);

volScalarField xx = mesh.C().component(0)/unit;
volScalarField yy = mesh.C().component(1)/unit;
volScalarField zz = mesh.C().component(2)/unit;

U.replace(0,2.0/::sqrt(3.0)*Foam::sin(xx)*Foam::cos(yy)*Foam::cos(zz));
U.replace(1,-2.0/::sqrt(3.0)*Foam::cos(xx)*Foam::sin(yy)*Foam::cos(zz));
U.replace(2,0.0);

U.write();
```

This reads in the $\underline{u}$ field (from the first available timestep directory by default). The line

```
volScalarField xx = mesh.C().component(0)/unit;
```

extracts the cell centres from the mesh (`mesh.C()`) and returns the X-component, normalised to make it dimensionless. This gives the x (and y, z) components that we need. Then `.replace()` inserts appropriate values into the $\underline{u}$ field and we write it out at the end. You may wish to change the details of this code, for instance to change the magnitude of the initial velocities!

## VII.4 Sampling and Averaging

`functionObjects` in OpenFOAM are sections of code which can be added to existing solvers to extend their functionality. There are a number of these pre-written for particular tasks, and of course ways to write your own. As we have already seen, some

can be run from the command line. Others may require more substantial data input or execute tasks during the simulation itself; these are typically activated through entries in the `functions` section of `controlDict`. Two very standard operations which can be handled in this way are sampling data from a field and averaging the values in a field. Examples of both are included here.

The first entry in `functions` uses the `probes functionObject` – this returns the cell centre value closest to the sampling point. (An alternative `functionObject`, `internalCloud`, interpolates between neighbouring cell centre values to determine a more precise value at the actual sampling point, which may be different). The actual syntax used here is

```
probes
{
    type            probes;
    libs            ("libsampling.so");
    writeControl    timeStep;

    fields
    (
        p U
    );

    probeLocations
    (
        (3.1415927 3.1415927 3.1415927)  // middle of domain 2pi^3
    );
}
```

`libs` links to specific shared object libraries which contain the actual code for the `functionObject` (here, one called `libsampling.so`). `writeControl` specifies how frequently the function is to be evaluated – here, every timestep, but you can specify other options including only calculating for saved timesteps. The entry also specifies a list of field variables to sample (`fields`) and a list of points at which to sample (`probeLocations`; more than one point can be specified).

The other runtime `functionObject` demonstrated here is `volFieldValue`. This can be used to perform a number of algebraic operations on the specified fields (or through the `regionType` setting, a subset of the geometry known as a `cellSet`). In this case the operation being carried out is a simple average of the cell values.

```
vollFieldValue1
{
   type            volFieldValue;
   libs            ("libfieldFunctionObjects.so");
   fields          (p U);
   writeControl    timeStep;
```

```
    writeFields     no; // yes | no
    regionType      all;
    operation       average;
}
```

Again; here the function is evaluated every timestep, but other settings are possible. Other possible operations include weighted and volume averages, summations, and maximum/minimum calculations.

## VII.5 Coding in `functionObjects`

If a suitable `functionObject` is not available, then it is perfectly possible to write your own. If this is a complex function then this can be coded separately to create a user defined shared object library (.so), but this is beyond the scope of the current tutorial. However, if the operation is somewhat simpler, we can use the `functionObject coded` to include some lines of code which are compiled at runtime 'on the fly' to provide a new element of functionality. Here we have

```
EtCode
{
    functionObjectLibs ("libutilityFunctionObjects.so");
    type coded;
    name EtCode;
    writeControl timeStep;
    codeExecute
    #{
        volScalarField Et
        (
            IOobject
            (
                "Et",
                mesh().time().timeName(),
                mesh(),
                IOobject::NO_READ,
                IOobject::AUTO_WRITE
            ),
            0.5*magSqr(mesh().lookupObject<volVectorField>("U"))
            +mesh().lookupObject<volScalarField>("k")
        );

        Info << "Et: "
             << mesh().time().timeName()
             << "  "
```

```
                << Et.average().value() << endl;
        #};
    }
```

The lines after `codeExecute` form a block of OpenFOAM coding which should look fairly familiar; this creates a `volScalarField Et` which will be the total kinetic energy, and sets its value using `U` and `k`. Then the `Info` statement writes out the time and the averaged value of `Et`. The whole code fragment is given a name (`EtCode` in this case) and again is evaluated every timestep.

**[Q.VII.2]** Plot the decay of $E_t$ against time. How does this compare with results from other sources (eg Nektar++)? **[Q.VII.3]** Try a different LES SGS model. This is set in `constant/turbulenceProperties` – try some of the alternative models. **[Q.VII.4]** Try increasing the resolution of the mesh from $32^3$ to $64^3$.