Introduction to simulating real 3d flows – ActionCar model

Dr Gavin Tabor

28th June 2018

UNIVERSITY OF EXETER

ActionCar Case

These slides were originally developed for a teaching session for the 11th OpenFOAM Workshop held in Guimaraes, Portugal, and also used for one of the UK&RI OpenFOAM Users Meetings.

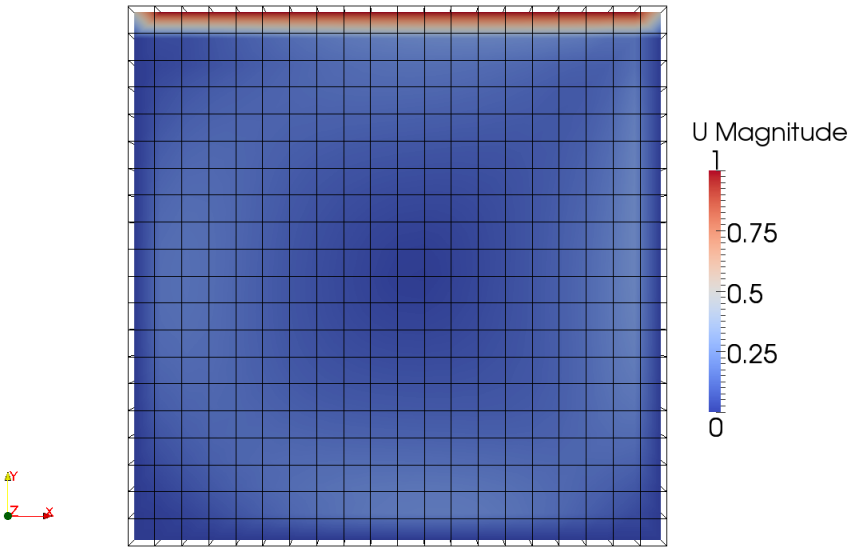
The intention is to take the reader through the process of setting up a basic automotive flow simulation using a simple CAD geometry supplied as an STL. This STL was originally developed by Patric Prosic for 3d printing (part of the RepRap project I believe).

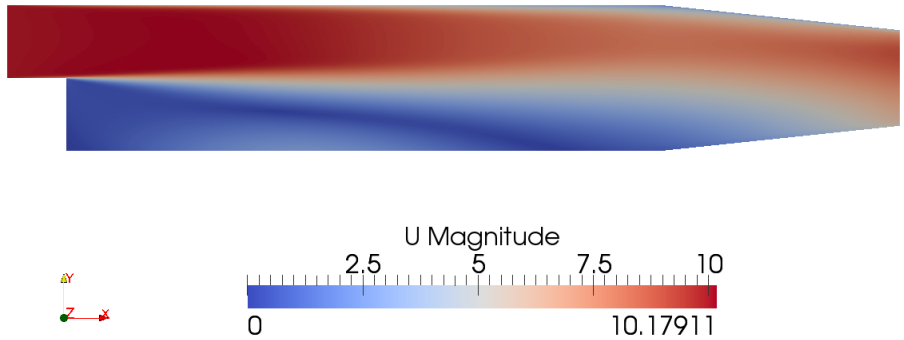
The case files are supplied as a directory `ActionCarRotatingWheels.zip`. This version should work with OFv5.

Have fun!

UNIVERSITY OF EXETER

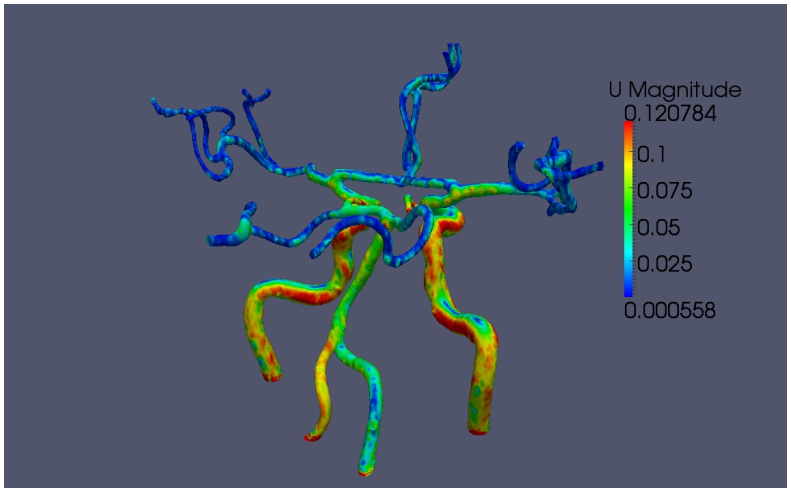
Undergraduate CFD...

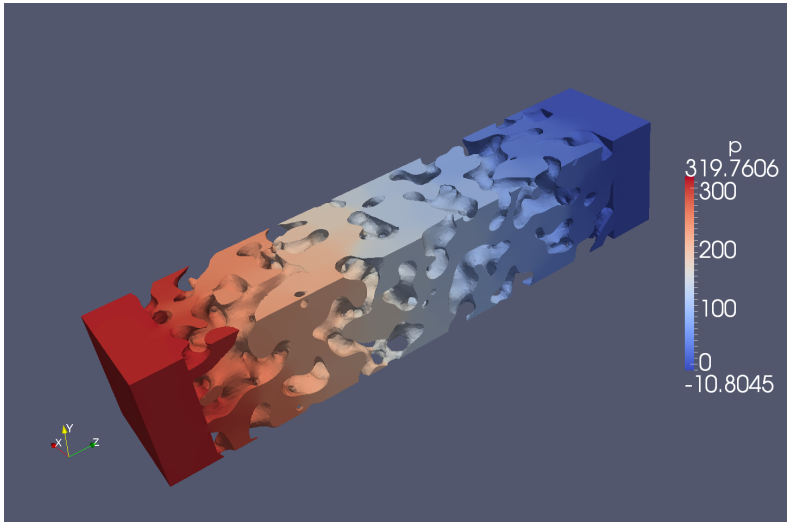


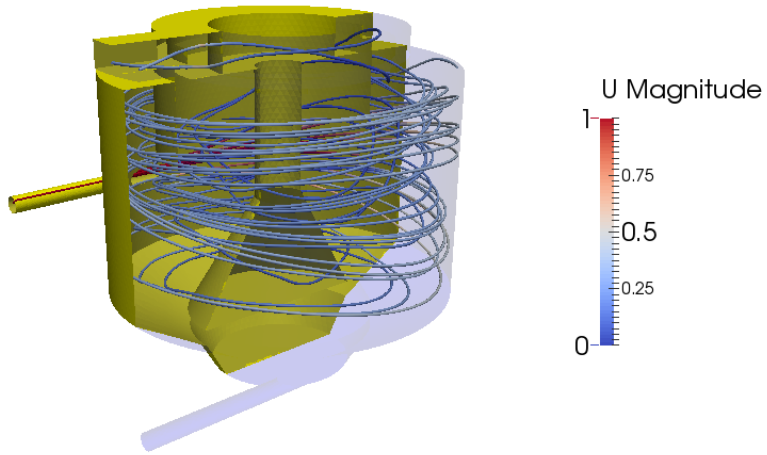


... is relatively straightforward

The real world . . .







. . . is a bit more complex!

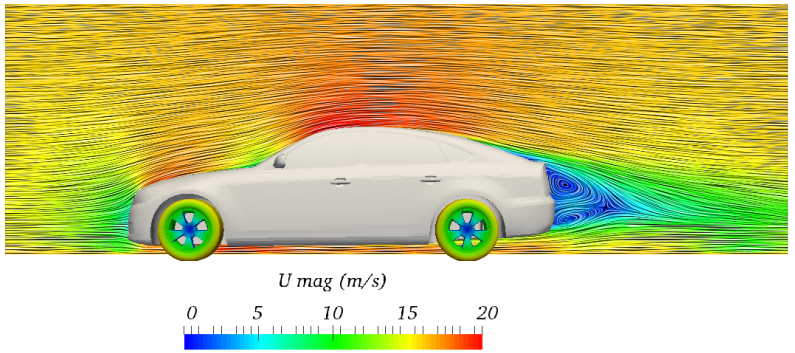
UNIVERSITY OF EXETER

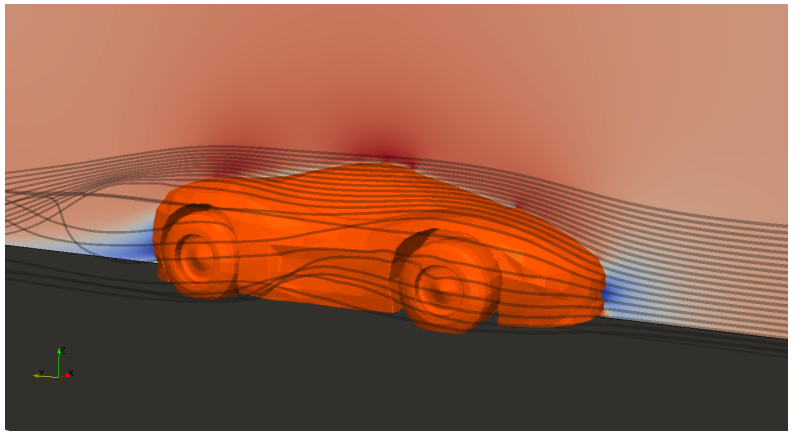
Outline of session

Aim of session – to explain some of the steps towards successful simulation of complex 3d problems :

- Meshing and mesh quality
- Turbulence modelling
- Numerical settings
- Boundary conditions

Aim to simulate a car model – relatively straightforward ActionCar model from RepRap (Patrick Prosic) – complex DrivAer case





UNIVERSITY OF EXETER

Assumptions

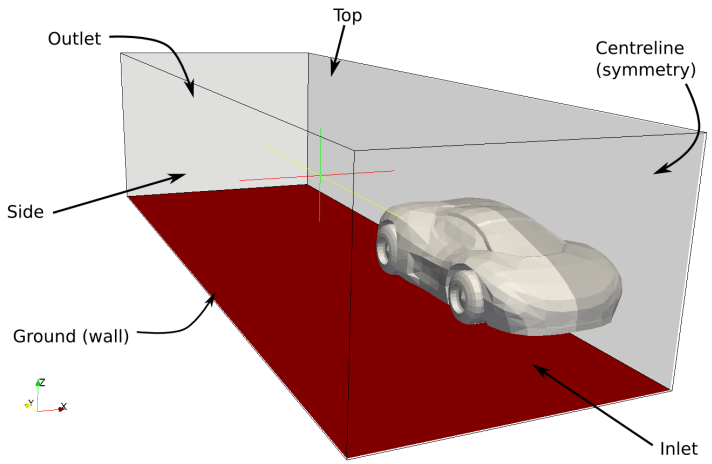
This is a beginners session, but I'm going to assume you know some stuff about OpenFOAM :

- Its command-line-driven (Linux) – I'm assuming you can run `paraFoam`, `blockMesh`/`snappyHexMesh` etc.
- Uses set of input files in a directory called *Case Directory*
- Timesteps/iteration number files in numbered subdirectories
- Input/control through *dictionaries* (keyword/value pairs) which we edit using ASCII text editors (emacs)
- `control` subdirectory contains mesh, physics
- `system` subdirectory contains numerical settings

UNIVERSITY OF EXETER

The Case

Inspect in `paraFoam` :



Y is streamwise direction;
Z height

Bounding boxes;

	X	Y	Z
Car body	-0.937	-70.1	-0.101
	41	2.95	19.9
Size	41.9	73	20
Front wheel	-0.792	-58.6	-2.08
	10.4	-45.4	11
Size	11.2	13.1	13.1
Back wheel	-0.855	-19.2	-2.2
	11.4	-4.8	12.2
Size	12.3	14.4	14.4

UNIVERSITY OF EXETER

Meshing

Meshing is probably the most significant part of any CFD process – determines the quality of the solution or even if you can get one at all!

Also incredibly time-consuming – can take weeks of effort to construct a good mesh.

Meshers can give lots of human control (CAD-based, eg Pointwise, ANSYS Mesher, gmesh) or automated cut-cell meshers (`snappyHexMesh`, ScanIP, harpoon)

Pointwise	`snappyHexMesh`
Requires detailed human interaction	Automated (but need to select entries in `snappyHexMeshDict`
Can give very high quality meshes	Tends to need more cells to mesh
Problems with highly complex geom	Robust – but not geometrically faithful

Need to learn to use various meshers – choose best for case

blockMesh

`snappyHexMesh` requires an input base mesh to truncate – usually generated using `blockMesh`. Usually rectangular but doesn't have to be (here, using graded cells)

Geometry of domain :

	X	Y	Z
Domain	-50	-100	-1.1
	20	100	50
Size	70	200	51.1

```
convertToMeters 100.0;

vertices
(
    (-0.5 -1 -0.011)
    ( 0.2 -1 -0.011)
    ( 0.2  1 -0.011)
    (-0.5  1 -0.011)
    (-0.5 -1  0.5)
    ( 0.2 -1  0.5)
    ( 0.2  1  0.5)
    (-0.5  1  0.5)
);

blocks
(
    hex (0 1 2 3 4 5 6 7) (20 40 20)
    simpleGrading (0.2 1 5.0)
);
```

snappyHexMesh settings

Set up snappyHexMesh to read in the car body and wheels :

```
geometry
{
    ActionCar.stl
    {
        type triSurfaceMesh;
        name ActionCar;
    }
    ActionCar_FrontWheel.stl
    {
        type triSurfaceMesh;
        name FrontWheel;
    }
    ActionCar_BackWheel.stl
    {
        type triSurfaceMesh;
        name BackWheel;
    }
};
```

UNIVERSITY OF EXETER

snappyHexMesh settings :2

Refinement on surfaces; at least 2 levels of refinement and up to 3 :

```
    refinementSurfaces
    {
        ActionCar
        {
            level (2 3);
        }
    }
    refinementSurfaces
    {
        FrontWheel
        {
            level (2 3);
        }
    }
    refinementSurfaces
    {
        BackWheel
        {
            level (2 3);
        }
    }
```

snappyHexMesh

Now run the codes :

```
blockMesh
snappyHexMesh
```

This generates directories 1 2 3 containing the results of

1. mesh truncation
2. snapping to surface
3. boundary layer generation

which can be quite useful. Alternatively use

```
snappyHexMesh -overwrite
```

to overwrite existing mesh info

Tips and tricks – Meshing

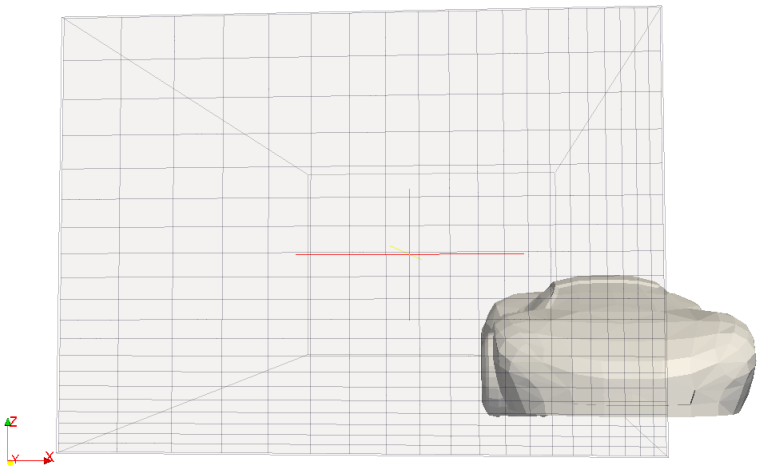
`paraview` contains a viewer for .stl files – use this to check dimensions, positioning

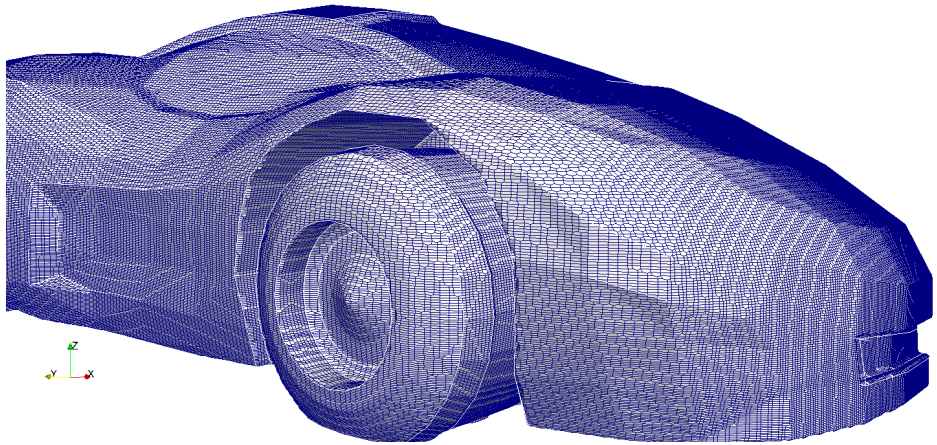
Run `surfaceFeatureExtract` to identify geometric edges in .stl files – improves accuracy :

- Adjust parameter `includedAngle` to tune how many edges are found
- Generates files in `constant/extendedFeatureEdgeMesh` – `XXXX.extendedFeatureEdgeMesh` to include in `snappyHexMeshDict`, `XXXX.obj` to visualise edges

Grade cells in `blockMeshDict` to resolve flow around car

Use `refinementBox` in `snappyHexMeshDict` to focus cell refinement in selected regions of the mesh.





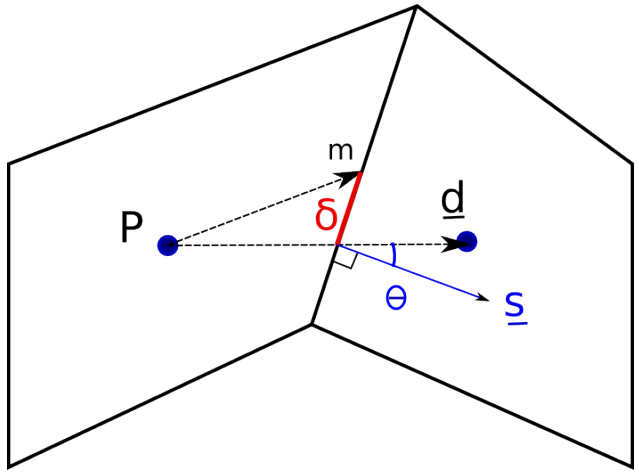
Mesh quality

Should *always* run `checkMesh` – utility checks for fatal errors (negative cell volumes etc) and mesh quality. Two main indexes :

Skewness is $\delta/|d|$

Non-orthogonality related to face angle

$$\theta = \cos^{-1} \frac{d.s}{|d||s|}$$



Both relate to numerical errors in the FVM.

Both can be compensated for (nonorthogonality correctors in PISO, `skewLinear` differencing scheme in `fvSchemes`), – but better to improve the mesh.

Mesh scaling

The car .stl was created in mm. OpenFOAM interprets this in metres, so we now have a car $73\mathrm{m}$ in length!

Could have scaled the .stl before; but easier to scale the geometry/mesh. Luckly OpenFOAM has a utility for this

```
transformPoints -scale ‘‘(0.075 0.075 0.075)’’
```

$(0.075 = 1.5/20)$.

As well as scaling, the `transformPoints` utility can also translate and rotate the mesh. It is found in

```
OpenFOAM-XXX/applications/utilities/mesh/manipulation
```

along with a number of other mesh manipulation tools.

Turbulence Modelling

Really important to get the physics correct – in this case the turbulence.

$$\mathcal{R}e = \frac{U \times L}{\nu} = \frac{10\mathrm{m/s} \times 1.5\mathrm{m}}{1.51 \times 10^{-5}\mathrm{m^2/s}} = 1 \times 10^6$$

Clearly turbulent! Various options :

Steady RANS – 2d? Cheap, inaccurate.

URANS – more accurate.

LES – Very costly, highly accurate. Must be 3d.

DES – many of the benefits of LES, rather cheaper (near-wall flow much cheaper to compute).

Turbulence modelling – cont

OpenFOAM implements (versions of) most of the main turbulence models.
OF3.0; dictionary `turbulenceProperties`, keyword `simulationType`.
Some suggestions :

Keyword	Comments
`kEpsilon`, `kOmegaSST`	Good general-purpose turbulence models
`realizableKE`	Suitable (correct) for rotating flows, strong curvature
`LRR`	Launder Reece Rodi Reynolds Stress model – anisotropic turbulence
`SpalartAlmaras`	S-A model (typically aerofoil simulations)

To check options – miss-spell model and see what it lists out!

Boundary and field types

B.C. are also a main source of errors in CFD. Need 2 pieces of info at inlet, 1 at outlet. Basic B.C types are as follows :

Type	blockMeshDict	Boundary	p	U	Other Scalar Fields
Wall	wall	wall	zeroGradient	fixedValue	zeroGradient
Inlet	patch	patch	zeroGradient	fixedValue	fixedValue
Outlet	patch	patch	fixedValue	zeroGradient	zeroGradient
Symmetry	symmetryPlane	symmetryPlane	symmetryPlane	symmetryPlane	symmetryPlane
Front/Back in 2d	empty	empty	empty	empty	empty

However there are also some specialist B.C which are essential or useful. Here we need : inlet, outlet, walls (car, wheels, ground), centreline.

Turbulent wall B.C.

OpenFOAM implements a range of law-of-the-wall type wall boundary conditions. Specify choice of wall function in `nut` variable file; need corresponding wall patches in `k`, `epsilon` and `omega`

E.g. in `nut` :

```
    car
    {
        type
nutkWallFunction;
        Cmu             0.09;
        kappa           0.41;
        E               9.8;
        value           uniform 0;
    }
```

Here

$$u^+ = \frac{1}{\kappa} \ln y^2 + E, \qquad \epsilon = C_\mu^{3/4} \frac{k^{3/2}}{l}$$

Other options include `nutRoughWallFunction`, `nutUSpaldingWallFunction` and `nutkAtmWallFunction`

Other B.C

The ground needs to move – so set `U` value `fixedValue (0 10 0)`

Wheels rotate : use `rotatingWallVelocity`

```
    backWheels
    {
        type            rotatingWallVelocity;
        origin          (2.79318 0 0);
        axis            (0 1 0);
        omega           -50.79;
        value           uniform (0 0 0)
    }
```

Outlet – use `inletOutlet` for `U`, `k`, `epsilon`, `omega`. Allows for backflow through outlet – keyword `inletValue` specifies the return value

Inlet conditions

We will need to specify k, ϵ at the inlet. CFD 'rules of thumb' apply :

$$k = \frac{3}{2}\left(I\overline{u}\right)^2, \qquad \epsilon = \frac{C_\mu^{3/4} k^{3/2}}{L}$$

for length scale L, turbulence intensity I

These are coded into b.c. `turbulentIntensityKineticEnergyInlet`, `turbulentMixingLengthDissipationRateInlet`.

It can also be useful to set initial conditions etc to these values, so worth calculating anyway! Here for $L = 1.0\mathrm{m}$ and $I = 10\%$, $k = 1.5\mathrm{m}^2/\mathrm{s}$ and $\epsilon = 0.3\mathrm{m}^2/\mathrm{s}^2$

Numerics

Specify settings for differencing schemes through `fvSchemes`, matrix inversion and general algorithm (PISO/Simple/Pimple settings) through `fvSolution`.

This is quite complex, high level stuff – if the calculation is not working, probably better to invest time in improving the mesh.

However ...

fvSchemes

`divSchemes` most impact on stability :

```
divSchemes
{
    default           none;
    div(phi,U)        bounded Gauss linearUpwindV grad(U);
    div(phi,k)        bounded Gauss upwind;
    div(phi,epsilon)  bounded Gauss upwind;
    div((nuEff*dev(T(grad(U))))) Gauss linear;
}
```

`upwind` – 1st order upwind differencing – most stable

`linear` – straightforward 2nd order differencing

`linearUpwind` – 1st/2nd order bounded scheme. "V" indicates that there are corrections to limiter based on direction of field.

Tips and tricks

. . . after Hrv Jasak! (for hex mesh)

`gradSchemes` Gauss or Gauss with limiters

`divSchemes`

- Low mesh quality – use `Upwind`
- Momentum equation – for 2nd order use `linearUpwind` (+ variants)
- Turbulence, bounded scalars – TVD/NVD schemes

`laplacianSchemes` – depends on max non-orthogonality :

- Below 60°, `Gauss linear corrected`
- Above 70°, `Gauss linear limited 0.5`

UNIVERSITY OF EXETER

fvSolution – matrices

Pressure solve is either *Conjugate Gradient* :

```
p
{
    solver          PCG;
    preconditioner  DIC;
    tolerance       1e-06;
    relTol          0;
}
```

or *Multigrid :*

```
p
{
    solver            GAMG;  // AMG for -ext branch
    tolerance         1e-7;
    relTol            0.01;
    smoother          GaussSeidel;
    nPreSweeps        0;
    nPostSweeps       2;
    cacheAgglomeration on;
    agglomerator      faceAreaPair;
    nCellsInCoarsestLevel 10;
    mergeLevels       1;
}
```

UNIVERSITY OF EXETER

Settings

Solvers work by repeating an approximate invertion over and over, until :

1. the residual falls below the solver `tolerance`, or
2. the ratio of current to initial residual drops below `relTol`, or
3. the number of iterations exceeds `maxIter`

These are *symmetric* matrix solvers – code will complain if used for asymmetric matrix. `GAMG` is fast; `PCG` is older (more robust)

For PISO, it is possible to have a separate `pFinal` setting – typically use `relTol` for $N - 1$ iterations, absolute tolerance for final pass.

Should check that `maxIter` (default 1000) is not being exceeded.

Other matrices

Other matrices typically asymmetric – need to use different solvers :

`smoothSolver` Direct solver using a smoothing solve operation (with `GAMG` for `p`)

`PBiCG` (with `PCG` for `p`)

Other settings are similar to before.

Possible to provide settings for multiple fields using text strings, e.g. :

```
"rho.*"
```

or

```
"(U|h|R|k|epsilon|omega)"
```

fvSolution – solution schemes

SIMPLE algorithm has few parameters.

```
SIMPLE
{
    nNonOrthogonalCorrectors 0;
    consistent yes;
}
```

Can also include `residualControl` settings

Also need a separate subdictionary for relaxation factors :

```
relaxationFactors
{
    fields
    {
        p               0.3;
    }
    equations
    {
        U               0.7;
        k               0.7;
        omega           0.7;
    }
}
```

PISO, PIMPLE have more options :

```
PISO
{
    nCorrectors     2;
    nNonOrthogonalCorrectors 0;
}
```

Typically look to do 2 correctors (+ 2 non-Orthogonal correctors) + use small Courant numbers (¡1.0). (Commercial codes typically use more corrector steps.)

If there is no outlet (to set $p = 0$), need to set a reference cell and value :

```
    pRefCell        0;
    pRefValue       0;
```

Starting the Simulation

If you are

1. running steady-state, or
2. only interested in long term transient behaviour

. . . then the initial conditions *should not* matter. However, *they probably do*!

Some tips :

- Choose good initial conditions; calculate likely values for k, ϵ, ω.
- Run `potentialFoam` to get a potential flow solution (for the velocity)

- If you have a coarse mesh solution, use `mapFields` to transfer to the fine mesh
- Start transient runs from a steady state solution. (Rule of thumb – for a domain of length L run for time $T = NL/u$ for solution to stabilise
- Start unstable models (eg. Reynolds Stress) from more stable ones (eg. $k - \epsilon$)

UNIVERSITY OF EXETER

Monitoring the run

OpenFOAM prints to the console information about the solution of each equation, every computational step. You should *always* monitor this, eg.

```
simpleFoam >& log &
tail -f log
```

For every equation solve, OpenFOAM prints out initial, final residuals, and number of iterations for the matrix invert. The pressure equation typically needs the most work – other equations *should* solve in 1-5 iterations.

Use the `foamLog` command to extract all possible data

Drag forces

Obvious output from the simulation – also useful to monitor convergence.

Use `functionObjects` :

- `forces` – calculates lift/drag and moments
- `forceCoeffs` – normalise by area

Include in `controlDict` dictionary – need to include appropriate S.O. library

```
functions
{
    forces
    {
        type           forces;
        functionObjectLibs ( "libforces.so" );
        outputControl   timeStep;
        timeInterval    1;
        log          yes;
        patches      (FrontWheel ActionCar BackWheel);
        rhoName      rhoInf;
        rhoInf       1.225;
        CofR         (0 -2.398 0.355);
        UName            U;
        pName            p;
    }
}
```

General tips/tricks

- Don't be afraid to experiment. If in doubt, use the defaults.
- The tutorials are complex and difficult to follow, but can be mined for information :

```
grep -r . -e ''string''
```

  is your friend here!
- Also, the source code :

```
find . -name <fileName>
```

- If something goes wrong, read the error message carefully (even miss spell keyword and see what happens!)

UNIVERSITY OF EXETER