LUNDS UNIVERSITET
Lunds Tekniska Högskola

Project work for the PhD course in OpenFOAM

A tutorial on how to use Dynamic Mesh solver **IcoDyMFOAM**

Performed by: Pirooz Moradnia
Contact: pirooz.moradnia@forbrf.lth.se

Spring 2008, Göteborg-Sweden

Introduction:

IcoDyMFoam is a _transient solver_ for _incompressible_, _laminar flow_ of _Newtonian fluids_ with _moving mesh_. In this document we will describe how to use this solver for different applications and get to know different input variables to choose from.

Basics of IcoDyMFoam:

IcoDyMFoam performs all moving or topologically changing actions to change the mesh. The actions to be taken are specified in the **dynamicMeshDict** file of the relevant case.

The key point in adapting the mesh movement and manipulation for a specific case is that one may need to write a customized library in which the type of mesh manipulation and relevant parameters are stored. This library will then be called by the mesh moving system in the **OpenFOAM**.

Another requirement is to check if all necessary functions resulting in the fulfillment of the desired action already exist in the "**libtopoChangerFvMesh**" (located in the path:
/lib/OpenFOAM/libtopoChangerFvMesh.so) or one should modify it.

In the simplest case where one only needs to move a simple mesh, it is enough to use the automatic mesh motion solver. Where the appropriate dynamic mesh is used and the motion of the boundary points is specified. The motion solver will then solve the mesh motion equation which can be controlled to achieve appropriate mesh grading, distortion control, etc.

For some cases the appropriate mesh motion classes are already available in the **OpenFOAM** distribution. Otherwise if for a special case there is no exactly matching pre-defined mesh class, then the user should think about the correct class of motion and topological variation to perform the right job.

All lower-level functionality, including the actual execution of topological changes, mesh mapping, data transfer etc. are done automatically by the classes already present in the code; thus one should not worry about mesh ordering, cell numbering etc. in any new mesh class.

Steps in generating a case with moving mesh:

1. The first step is to generate a mesh in the same way as one does in other solvers, namely with **blockMeshDict** dictionary. The key point is however, in order to have mesh movement in any direction, the boundary type for the moving and changing cells in the relevant direction should be set to "patch". This way, we ensure that there is no fixed wall in the specified boundaries, which means we can assign a moving mesh boundary to that.

2. The second step is to specify moving-mesh boundary conditions which allow the cells to move in the desired direction. In the boundary-conditions input files, the "slip" boundary types should be selected for them.

   In all **IcoDyMFoam** cases one should add a "**dynamicMeshDict**" file in the "**constant**" folder.

In this file the user determines the mesh-manipulation dictionaries and classes, as well as solvers, diffusivities and, if necessary, the coefficients required in each case. An example of how the dynamicMeshDict file should look like is shown below:

```
/*---------------------------------------------------------------------------------------------------*\
| =========                |                                                                         |
| \\      /  F ield        | OpenFOAM: The Open Source CFD Toolbox                                   |
|  \\    /   O peration    | Version:  1.4                                                           |
|   \\  /    A nd          | Web:      http://www.openfoam.org                                       |
|    \\/     M anipulation |                                                                         |
\*---------------------------------------------------------------------------------------------------*/

FoamFile
{
    Version         2.0;
    format          ascii;

    root            "";
    case            "";
    instance        "";
    local           "";

    class           dictionary;
    object          motionProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
dynamicFvMesh           dynamicMotionSolverFvMesh;
motionSolverLibs        ("libfvMotionSolvers.so");
solver                  displacementLaplacian;
diffusivity             uniform;
// ************************************************************************* //
```

We will now have a look at different phrases and relevant options in this file.

1- Selecting the right mesh-manipulation approach:

We have a variety of selections for the mesh-manipulation approaches to choose from. At the time of writing this document, the general types used in the official version (**OpenFOAM-1.4.1**) are:

1.1 Automatic mesh motion: is used when the topology of the mesh does not change, but instead only the spacing between (almost all) nodes changes by stretching or squeezing. It is the default selection in the **dynamicMeshDict** file and the corresponding library selected for that is "**libDynamicFvMesh.so**" (located in the path:**/lib/OpenFOAM/ /libDynamicFvMesh.so**).

1.2 Topological changes in the mesh: is used when the topology of the mesh changes during simulation. In this case the user should specify the name of the corresponding dynamic library, "**libtopoChangerFvMesh.so**" (located in the path **/lib/OpenFOAM/libtopoChangerFvMesh.so**).

The necessary changes in the mesh will be identified by special mesh-modifying commands. Mesh-modifiers comprise a set of data which are used specifically for performing certain topological changes on the mesh, by modifying points, faces or cells (or zones in general). The different mesh modifiers in this case are:

- **Boundary attach or detach**
- **Layer addition or removal**
- **Sliding interfaces**

This way one can basically perform all mesh manipulation operations and change the mesh in any arbitrary way during the simulations. The use of modifiers will be described later.

Mesh-manipulation models:

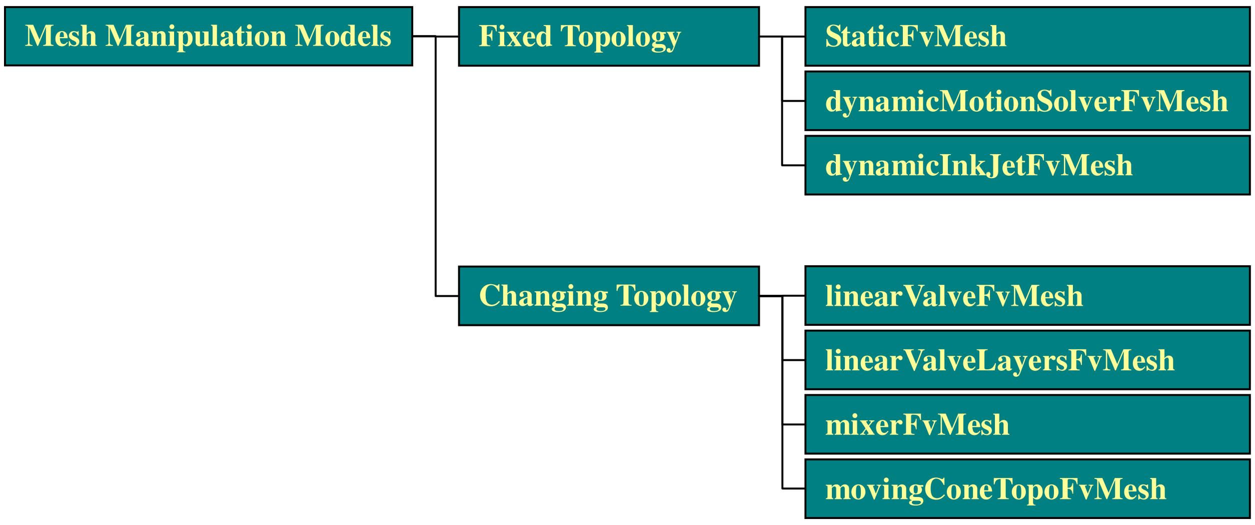
After selecting a mesh-manipulation approach, it is time to specify the appropriate class to change the mesh. Let us identify the different options in each approach. There are currently three mesh-manipulation types in automatic mesh motion solver approach (located in the path: **src/dynamicFvMesh**):

1.1.1 staticFvMesh
1.1.2 dynamicMotionSolverFvMesh:
1.1.3 dynamicInkJetFvMesh

While in the topological mesh changer (located in the path: **scr/topoChangerFvMesh**), the available options are:

1.2.1 linearValveFvMesh
1.2.2 linearValveLayersFvMesh
1.2.3 mixerFvMesh
1.2.4 movingConeTopoFvMesh

The following sketch would be a good illustration for different mesh manipulation models:



It is now good to have a brief description about how each type works in each case.

1.1 dynamicFvMesh:

Here, as mentioned, the topology of the mesh remains constant and the only change in geometry, if any, will be done by stretching or squeezing the cells and node positions. Now let us have a look at each class to see how it works:

1.1.1 staticFvMesh:

As its name suggests, there will be no dynamic changes in the mesh and is the same as the case with no mesh motion. The type requires no additional phrases; e.x. solver type, mesh diffusivity model, coefficients, etc.

1.1.2 dynamicMotionSolverFvMesh:

Unlike the previous case, here we have a mesh manipulation type which solves the cell movement equations, its name suggests. It can be used in cases were the resolution is not changing too much during the mesh motion; i.e. relatively small changes in mesh occur, so that cell density changes do not affect the results during the computations. This type of mesh motion approach is the simplest type of mesh motion solver and in its dictionary one should specify the name of appropriate "**solver**" and the type of "**diffusivity**" model. This way the equations of cell motion will be solved according to the selected solver specifications and with help of mesh diffusivity models, which will be discussed in more details later.

1.1.3 dynamicInkJetFvMesh:

It is a part of **dynamicFvMesh**, which means that in the **dynamicMeshDict** file, the solver and diffusivity model should still exist in the input file. It doesn't change the topology of the mesh and thus should again be used in cases where the changes in cell density do not affect the results and corresponding precision during the simulations. In this case the mesh is moving based on a harmonic motion around a reference plane along the x-axis, so the user should specify a sub-dictionary, named "**dynamicInkJetFvMeshCoeffs**", comprising three coefficients as below:

– amplitude	(the amplitude of the motion)
– frequency	(the frequency of the motion)
– refPlaneX	(the x-component of the reference plane)

1.2 topoChangerFvMesh:

In this class, as mentioned previously, the topology of the mesh does not remain constant. For this to take place correctly, we need mesh modifier input files to describe what kind of mesh manipulation action we aim to undertake. It is generally used in cases where either the original topology can not be kept at all or the precision of the solution would be affected by keeping the original mesh settings during the simulations. The class **polyTopoChanger** will search for the **meshModifiers** input file and if it exists, the necessary data are extracted from it. Otherwise the data are read from the **dynamicMeshDict** file or similar. We now have a look at the different classes of this category.

1.2.1 linearValveFvMesh:

This type is specially designed to make it possible to use sliding meshes at the interface of two pieces of mesh in relative linear motion. It is originally adopted from the "**mixerSlider**" case (described later in this text) and is modified to be especially used the interface between the inner and outer sides of the locus of a valve moving in an internal combustion engine. This means that a virtual interface will be made between the inner and outer parts of the cylinder, which is swept by the valve and over which a jump happens between the mesh nodes on both sides. This is done first by decoupling the nodes at the two sides and then coupling them at the end of each time step.

First the points, faces and cells in the interface are identified. The cut points are then set to empty zones and then the inner and outer zones are read. The cells on both sides are detached and the mesh motion equations are solved and the change is performed. Finally the interface is attached.

This way the mesh inside the cylinder (the moving part) gets stretched or squeezed, depending on the problem, while the mesh in the outer part remains fixed. In this case the user should specify certain variables and assign values to them in a sub-dictionary "slider" in the **dynamicFvMesh** dictionary. The inner and outer parts of the sliding mesh will be identified in the sub-dictionary "slider" as below:

innerSliderName	inside;	(the inner side of the interface)
outerSliderName	outside;	(the outer side of the interface)

Also, there is yet another sub-dictionary called "**linearValveFvMeshCoeffs**", in which the user selects the motion solver type for mesh handling. Other input variables are the same as the **dynamicFvMesh** case.

1.2.2 linearValveLayersFvMesh:

This case is very similar to the **linearValveFvMesh** case, with the extra feature of layer addition and removal in the valve- and piston-swept regions instead of pure stretching or squeezing of nodes and cells. The basic input variables needed are again the same as those mentioned above for the **linearValveFvMesh** case, together with those added in an extra sub-dictionary called "**layer**" here:

layerPatchName	patch;	(type of boundary to be handled)
minThickness		(minimum thickness of the layers)
maxThickness		(maximum thickness of the layers)

Also in the main body of the motion dictionary the velocity of the piston should be specified under:

```
pistonVelocty
```

1.2.3 mixerFvMesh:

This case is designed to deal with the sliding interface located between the rotor and casing meshes in a simple mixer. Here we need an interface between two parts in relative rotational motion. The mesh under inspection here is the one used in **mixerVessel2D** case in **MRFSimpleFoam** directory (**MRF** refers to **Multi-Reference-Frame**). In this case there is a relative motion between the two mesh parts, which requires a sliding interface between the two regions, as in the two previous cases. Now we have a look at the different files required to use this class:

dynamicMeshDict: Here we need to specify information about the libraries, besides a sub-dictionary to specify the relevant coefficients, in more details, coordinate system, rotational velocity of the rotor, and the two sides of the sliding surface. This is shown below:

```
dynamicFvMeshLib      "libtopoChangerFvMesh.so";    (name of the library)
dynamicFvMesh         mixerFvMesh;                  (name of the class)

mixerFvMeshCoeffs                                   (coefficients sub-dictionary)
{
   coordinateSystem                         (info about the coordinate system)
   {
      type                          (type of the coordinate system)
      origin                        (coordinate system origin)
      axis                          (axis of rotation)
      direction                     (direction of rotation)
  }

  rpm                                       (rotational speed)

   slider                                   (determines the sliding interface)
   {
      inside                        (name of inner interface side)
      outside                       (name of outer interface side)
   }
}
```

There is yet another important dictionary named **MRFZones**, which is also located in “**constant**” folder. It determines the moving parts, or in other words, the mesh parts with reference frames other than the one used for the main mesh. It is written in the following way:

```
1                                   (number of parts with new reference frames)
(
   rotor                            (name of the part)
   {
      patches               (reference frame info (dimensioned))
      origin                (reference frame origin)
      axis                  (direction of rotation axis)
      omega                 (rotational frequency (degree/sec))
   }
)
```

Once this dictionary is read, the following zones are automatically generated:

```
cutFaceZone
cutPointZone
masterFaceZone
slaveFaceZone
masterPatch
slavePatch
```

These are used by the **slidingInterface** class, which in turn gives the relative motion between the two sides of the sliding interface.

1.2.4 movingConeTopoFvMesh:

This is a special version modified to be used for simple cases similar to the **movingCone** case in the original **icoDyMFoam**, the difference being mesh movement models used. Here the mesh moves based on a harmonic sinusoidal model.

In the original case, as one can notice, the mesh movement is performed automatically, which means that the topology of the mesh remains the same and only the points move according to the solution model and diffusivity.

In the present case, the topology does not remain the same, but instead, the number of cell layers in the moving region changes. This gives a better resolution control during the simulation, because the minimum and maximum allowable cell layer thicknesses are specified. This provides that, specially in the cases with large mesh moving distances, the problem with high mesh stretching (poor resolution) and/or squeezing (unnecessary computational power used) in different regions is solved.

In this case, the mesh motion is first performed simply by the original squeezing / stretching approach until either the minimum or the maximum cell layer thicknesses reach a critical (specified) value. In this case, depending on the region and criterion, a new cell layer is added (to the expanding region) or an old cell layer is removed (from the compressing region) and the new cell sizes are adjusted; this is all done while the simulations continue, so that the solution will remain continuous, independent of the number or the size of the cell layers.

Here the user should make changes to the **dynamicMeshDict** file and also provide an extra dictionary file in the “constant” directory of the case, called “**meshModifiers**”. This file contains extra information about the moving boundaries. Let us now have a look at the necessary keywords in each dictionary and describe them.

dynamicMeshDict:

In this file, first the name of the correct library and class should be specified; that is:

```
dynamicFvMeshLibs     1("libtopoChangerFvMesh.so");
dynamicFvMesh         movingConeTopoFvMesh;
```

Secondly, a sub-dictionary containing necessary coefficients for the class should be added. The Coefficients should be specified for the pre-defined keywords to specify moving and fixed boundaries and characteristics, besides the minimum and maximum cell layer thicknesses in each region (refer to **movingConeTopoFvMesh.C** file). The sub-dictionary keywords to be specified are:

```
movingConeTopoFvMeshCoeffs
{
   motionVelAmplitude            (velocity amplitude of motion, vector)
   motionVelPeriod               (period of motion, scalar)

   leftEdge                      (left-most side of the domain, scalar)
   leftObstacleEdge              (left side of the moving object, scalar)
   rightObstacleEdge             (righ tside of the moving object, scalar)

   left                          (the region on the left hand side of the object)
   {
      minThickness        (minimum cell thickness in the left region, scalar)
      maxThickness        (maximum cell thickness in the left region, scalar)
   }

   right                         (the region on the right hand side of the object)
   {
      minThickness        (minimum cell thickness in the right region, scalar)
      maxThickness        (maximum cell thickness in the right region, scalar)
   }
}
```

meshModifiers:

Here the user should provide information about the number of regions and certain action to be taken on them for the specific mesh-manipulation method. Also one can select if the specified action should be active or not in the ongoing part of the simulations. The script below is a piece of the dictionary file for the **movingConeTopoFvMesh** case, which might be useful to help understanding the whole idea:

```
2                                      (number of regions)
(
right                                  (right region)
{
   type layerAdditionRemoval;   (name (ID) of the action to be taken)
   faceZoneName rightExtrusionFaces;          (name of the zone)
   minLayerThickness            (minimum layer thickness on right side, scalar)
   maxLayerThickness            (maximum layer thickness on right side, scalar)
   oldLayerThickness            (thickness of the layer before extrusion)
   active on;                   (extrusion will be performed in this region)
}
      left                                    (left region)
{
   type layerAdditionRemoval;   (name (ID) of the action to be taken)
   faceZoneName leftExtrusionFaces;           (name of the zone)
   minLayerThickness            (minimum layer thickness on left side, scalar)
   maxLayerThickness            (maximum layer thickness on left side, scalar)
   oldLayerThickness            (thickness of the layer before extrusion)
   active on;                   (extrusion will be performed in this region)
}
)
```

This is basically how the parameters should be specified in this class, which in turn performs the mesh-motion action in the desired way.

2. Selecting the right Solver:

Moving points in a grid requires models and corresponding mesh motion equations to be solved. In the official version of OpenFOAM-1.4.1 there are a number of available solvers (located in the path: **src/fvMotionSolver/fvMotionSolvers**) working based on the following models:

2.1 displacementLaplacian
2.2 velocityLaplacian
2.3 SBRStress

It might be beneficial to note that the Laplacian solvers have a "component" version as well, which is just similar to the general cases, with the difference being the directional solution in the latter case (for example only x-component, etc.). We now have a look at what each of these solvers refer to and how they work:

2.1 displacementLaplacian

In this case, the equations of cell motion are solved based on the Laplacian of the diffusivity and the cell displacement; thus the diffusivity model should be read from the **dynamicMeshDict** and an extra file named "**pointDisplacement**" in the starting time folder should be available. This means that one should specify the (dimensioned) final displacement of mesh components as well as the mesh displacement of the internal field, if applicable. The aforementioned file should include the following definitions:

dimensions	(determines the displacement dimensions)
internalField	(the displacement of the internal field mesh)
boundaryField	(the displacement of the boundaries mesh)

In addition, one can also specify another file called "**cellDisplacement**" which is very similar to **pointDisplacement file**, with the difference being the class types: the former is a **pointVectorField** class, while the latter is a **volVectorField** type. Notice that the existence of this file is optional and for simplicity can be skipped. The solver evaluates a Laplace equation for the diffusivity and the cell displacement in each time step, which gives the appropriate displacement of the cells and corresponding points.

2.2 velocityLaplacian

This case is very similar to the previous solver, with the difference being the equation to be solved, which is the Laplacian of the diffusivity and the cell motion velovity; thus a "**pointMotionU**" file should be read here, which determines the velocity at which each single boundary is moving. Another file which might or might not be present is the **cellMotionU** file, which is basically the same as the **pointMotionU**, but with a **volVectorField** class instead of the **pointVectorField** class in the previous case. The set of input variables are the same as those in the displacementLaplacian case, but care should be taken in determining the dimensions, bearing in mind that here the code deals with boundary velocities, instead of final motions.

2.3 SBRStress

It is a displacement model, solving Laplacian of diffusivity and the cellDisplacement. It also takes into account the solid body rotation term in calculations. The necessary files to be read after each time step calculation are the **pointDisplacement** from the constant directory and the **points** file in the polyMesh directory. Again the **cellDisplacement** file can also be provided but is not necessary.

3. diffusivity models:

Diffusivity models determine how the points should be moved after solving the equation of cell motion for each time step. The diffusivity models (available in the path: **src/fvMotionSolver/motionDiffusivity**) implemented in the official OpenFOAM-1.4.1 version fall into two main categories:

3.1. Quality-based methods: In which the diffusion field is a function of cell quality measures. The models in this category are:

3.1.1. uniform
3.1.2. directional
3.1.3. motionDirectional
3.1.4. inverseDistance

3.2. Distance-based methods: Which are used together with the quality-based models and in which the diffusion field will be a function of cell center distance "*l*" to the nearest selected boundary. These models are used with "inverseDistance" method above and are:

3.2.1. Linear
3.2.2. Quadratic
3.2.3. Exponential

Mesh modifications are performed after each time step, based on the information provided. Consider the initial simple grid below:

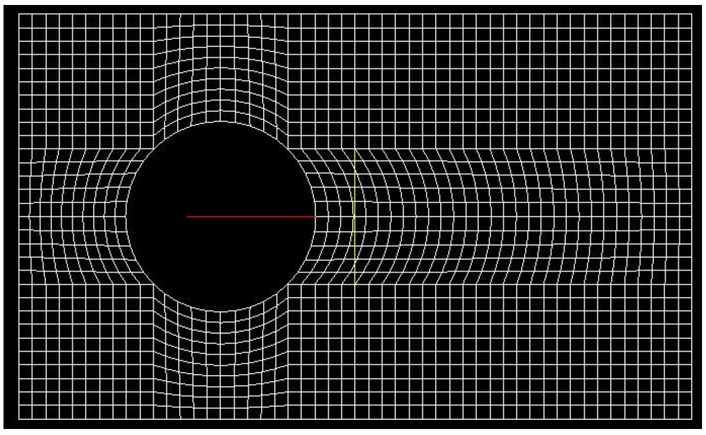


Figure 1: Initial mesh

We will now have a brief look at each model and manipulator to see how their different variables affect the mesh motion:

3.1.1. uniform

No specific data is needed here except the name of the diffusivity model. The mesh manipulation will be done uniformly for all moving boundaries; that is to say, all cells in each region get stretched or squeezed with the same ratio. On the other hand, the different parts of the mesh are handled uniformly, depending on their distance from the moving faces. The figure below shows the result of mesh motion using this model:

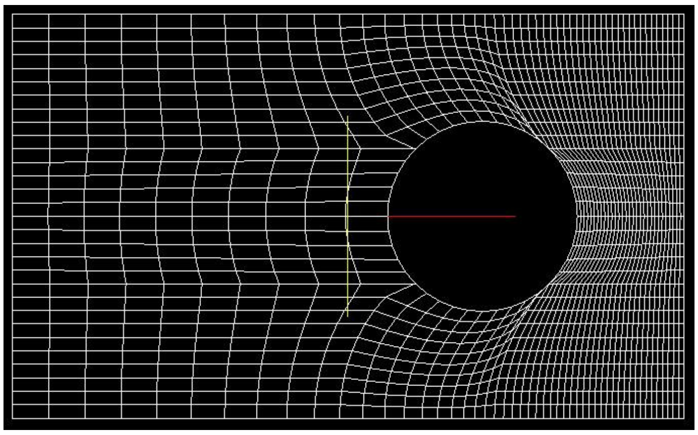


Figure 2: Uniform diffusivity

3.1.2. directional

Here the mesh stretching or squeezing will be done proportional to the direction of the motion. In this case we need to specify two scalar coefficients for the model to work. One can use a third mixed coefficient as a combination of the two, but it is not necessary. Let us now have a look at how these two numbers affect the mesh motion and then explain about their roles:

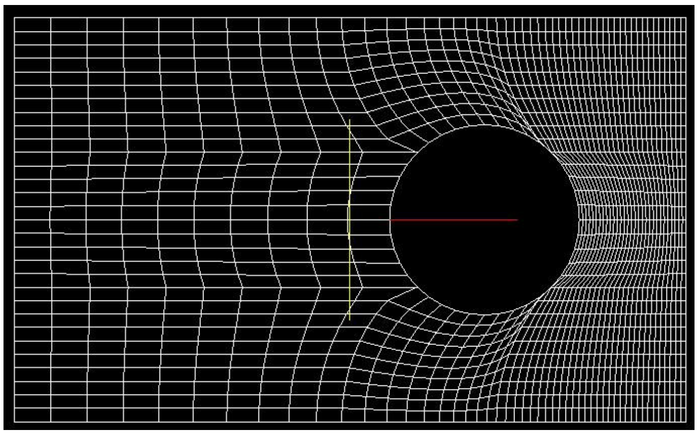


Figure 3: Directional diffusivity with coefficients 0.1 & 0.1

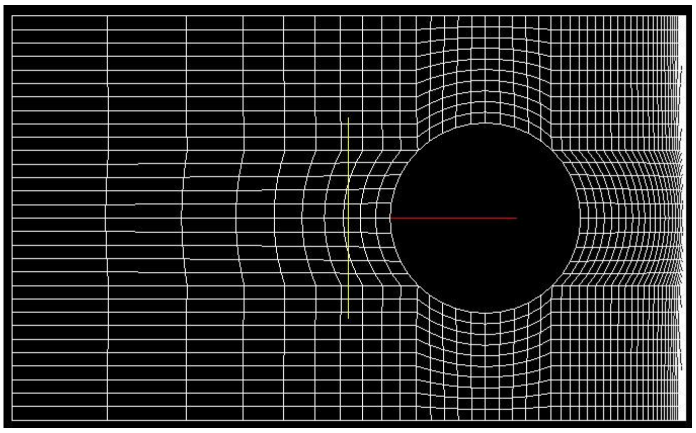


Figure 4: Directional diffusivity with coefficients 0.1 & 10

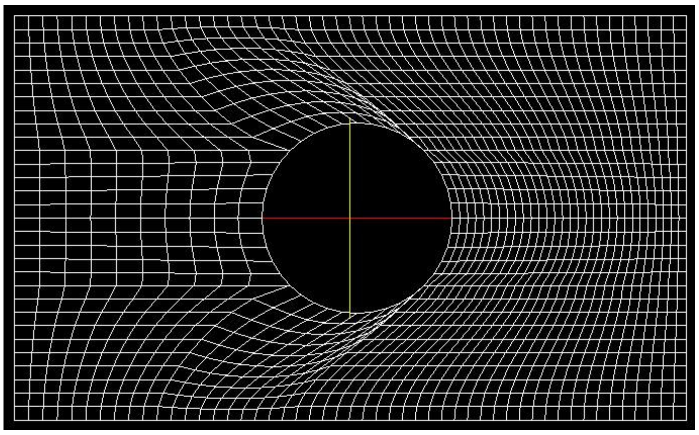


Figure 5: Directional diffusivity with coefficients 10 & 0.1

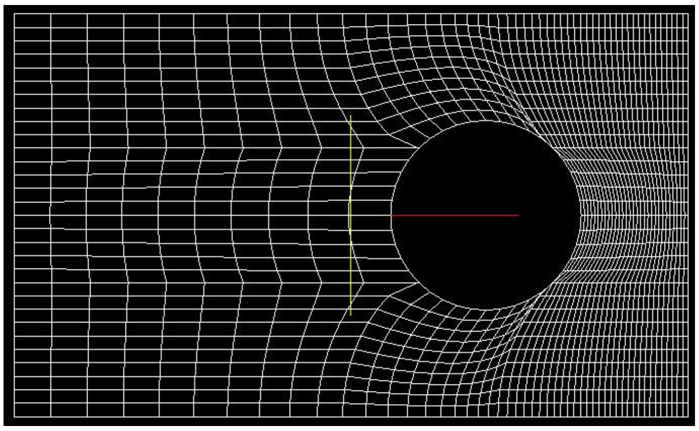


Figure 6: Directional diffusivity with coefficients 10 & 10

As it is obvious, the first number determines the **mean cell non-orthogonality**; while the second number determines the **mean cell skewness**.

3.1.3. motionDirectional

Here again we should specify the same two coefficients for the model to work. The difference between this model and the directional diffusivity model is that here the mesh manipulation is done by prioritizing the moving body and adjusting the cells in a way that is more appropriate for the moving body; while in the other case, the mesh manipulation is done by considering the slipping boundaries.

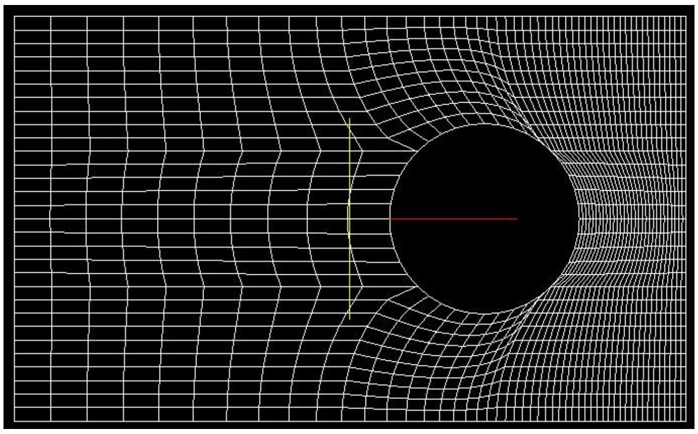


Figure 7: motionDirectional diffusivity with coefficients 0.1 & 0.1

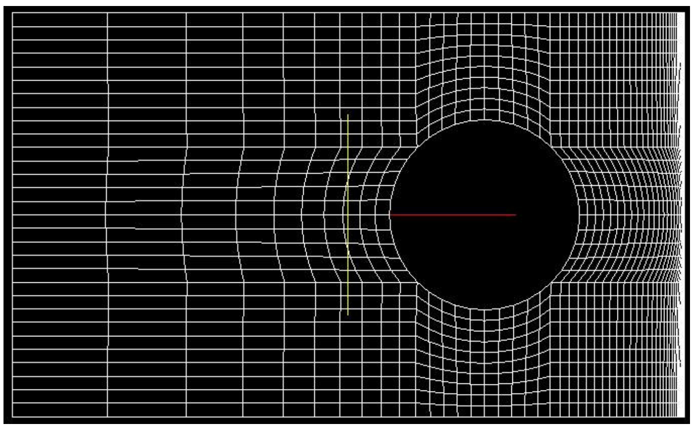


Figure 8: motionDirectional diffusivity with coefficients 0.1 & 10

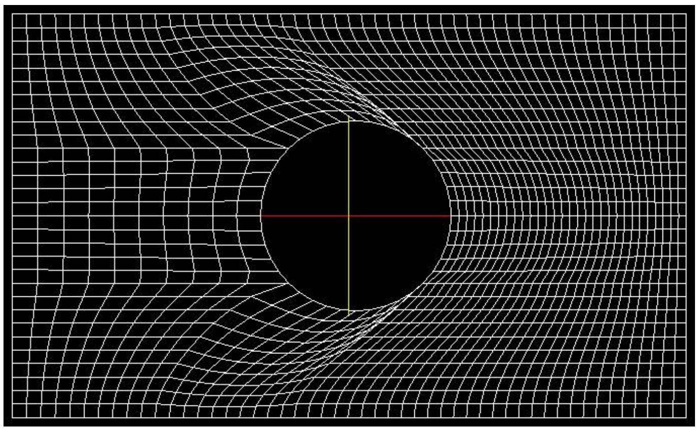


Figure 9: motionDirectional diffusivity with coefficients 10 & 0.1

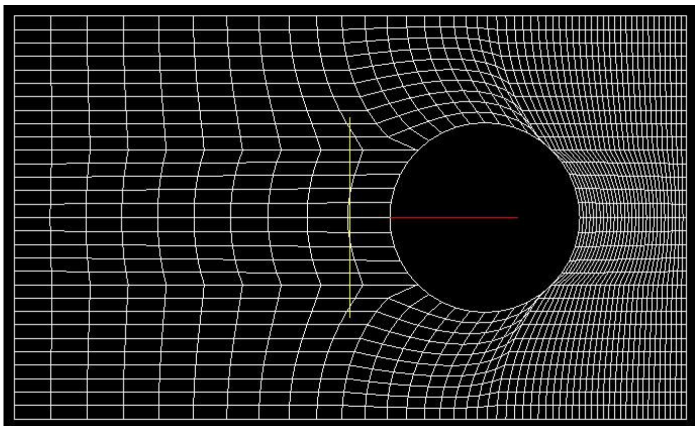


Figure 10: motionDirectional diffusivity with coefficients 10 & 10

3.1.4. inverseDistance

In this case the user specifies one or more boundaries and the diffusivity of the field is based on the inverse of the distance from that boundary. To clarify this, let's have a look at the figure below:

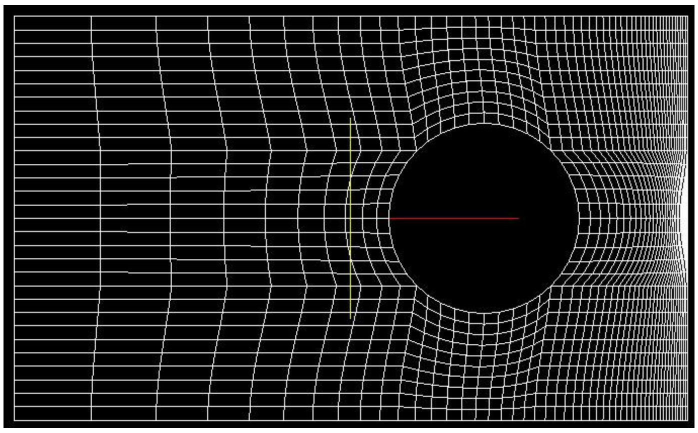


Figure 11: diffusivity with inverse distance from the circular body

The methods discussed above were the so called "**Quality-based methods**". Let us now have a look at the other category, i.e. "**Distance-based methods**":

3.2.1. Linear

In this case the diffusivity field is based linearly on the inverse of the cell center distance to the nearest boundary, that is: ($1/l$). The corresponding figure for this model is figure 11 above.

3.2.2. Quadratic

The same as 3.2.1, with the only difference being a quadratic relation instead of a linear one: ($1/l^2$). Looking at figure below one can easily distinguish between this model and the linear one:

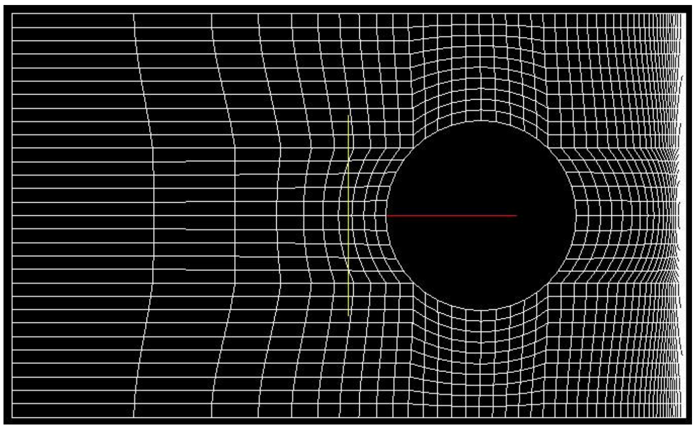


Figure 12: diffusivity with quadratic inverse distance from the circular body

3.2.3. Exponential

Finally, in this method the field diffusivity is based on the exponential of the inverse of cell-center distance to the selected boundaries: ($1/\exp(-l)$). The figure below shows the use of this method with different exponents selected:

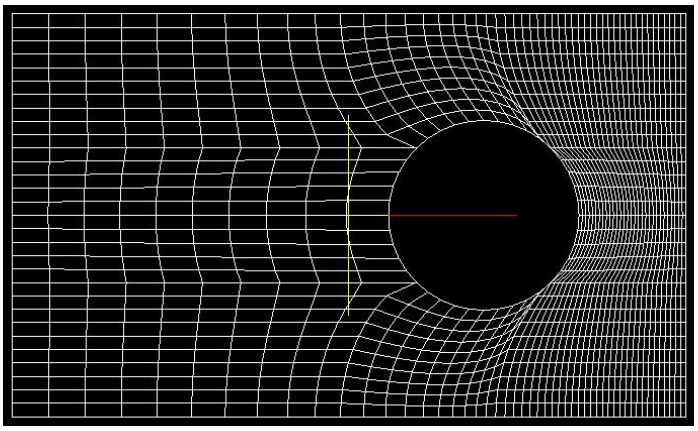


Figure 13: diffusivity with inverse distance from the circular body, exponent 0.1

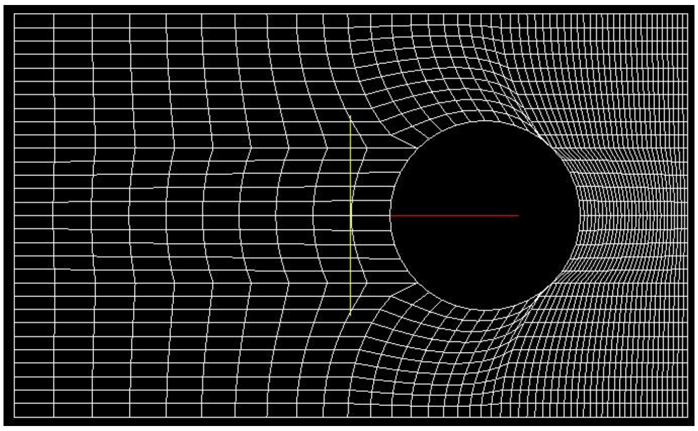


Figure 14: diffusivity with inverse distance from the circular body, exponent 100

References:

- http://openfoam.cfd-online.com/
- http://openfoamwiki.net
- http://powerlab.fsb.hr/ped/kturbo/OpenFOAM/
- http://www.opencfd.co.uk/openfoam/
- http://www.tfd.chalmers.se/~hani/kurser/OF_phD_2007/