CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Project work:

An Unsteady-Periodic Flow generated by a Oscillating Moving Mesh

---

Developed for OpenFOAM-2.1.x

Author:
Ardalan Javadi

Peer reviewed by:
Unknown
Håkan Nilsson

October 21, 2012

Chapter 1

point-wise deformation of mesh patches

1.1 Introduction

This report is given to introduce zero-net-mass-flux (synthetic) jet that is an unsteady periodic flow of with transitional intrinsic. The synthetic jet is introduced by Smith & Glezer(1991) . A typical jet actuator comprises a cavity with an oscillating diaphragm and a slot or orifice from which the jet issues. A train of vortex rings is generated due to the time-periodic motion of the diaphragm imparting a finite momentum into the surrounding fluid with zero-net-mass-flux. However, a unique feature of synthetic jets is that they are formed entirely from the working fluid of the flow system in which they are deployed and, thus, can transfer linear momentum to the flow system without net mass injection across the flow boundary.

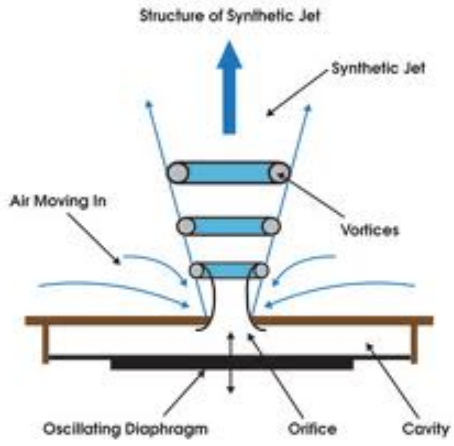

Figure 1.1: Schematic of a synthetic jet.

1.2 Pre-processing

The point-wise deformation of mesh patches gives possibilities to changes shapes of an object, for example in optimisation purposes. It also gives possibility for active flow control calculations by pre-defined movement of patches that affect the flow while running a simulation.
This tutorial describes how to build a library that introduces new mesh boundary conditions which give the user possibility to deform patches of a mesh according to a particular polynomial-periodic function. With small changes this method can easily be used with other functions, either by hard coding it into the library or by including some equation parser in the library. Finally the point-wise deformation will be shown in action by deforming the diphragm of a synthetic jet. Active flow control will also be introduced and implemented by allowing for periodic changes of patches where the patch returns to its original position within a specified time limit.

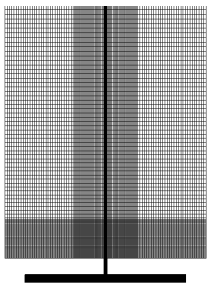

Figure 1.2: Schematic of a synthetic jet.

1.2.1 Getting started

The easiest way to start is to find a library that gives the most similar behaviour to what point-wise deformation is expected to have. The library chosen here rotates patches around a defined axis by defing velocity of each node. It can be found in:

```
cp -r  $FOAM_SRC/fvMotionSolver/pointPatchFields/derived/angularOscillatingVelocity/
 $FOAM_RUN/libMyPolynomVelocity
cd $FOAM_RUN/libMyPolynomVelocity
```

and also copy the `Make` folder:

```
cp -r $FOAM_SRC/fvMotionSolver/Make $FOAM_RUN/libMyPolynomVelocity
```

Clean up:

```
cd $FOAM_RUN/libMyPolynomVelocity
wclean
rm -r Make/linux*
```

It is recommended to rename files and folders in order to not get them mixed up with the original library. Here the folder will be renamed `libMyPolynomVelocity` and the new library will be named `libMyPolynomVelocityPointPatchVectorField.`

```
sed -e 's/angularOscillating/libMyPolynom/g' \ angularOscillatingVelocityPointPatchVectorField.C >
 libMyPolynomVelocityPointPatchVectorField.C
sed -e 's/angularOscillating/libMyPolynom/g' \ angularOscillatingVelocityPointPatchVectorField.H >
 libMyPolynomVelocityPointPatchVectorField.H
```

```
1  libMyPolynomVelociyPointPatchVectorField.C
2
3   LIB = $(FOAM_USER_LIBBIN)/libMyPolynomVelocity
```

To be able to compile the library it is also necessary to edit the files and options files inside the `Make` folder. The `Make/files` should only include the following:

The `Make/options` should include the following:

```
1  EXE_INC = \
2  -I$FOAM_SRC/triSurface/lnInclude\
3  -I$FOAM_SRC/meshTools /lnInclude\
4  -I$FOAM_SRC/dynamicMesh/lnInclude\
5  -I$FOAM_SRC/finiteVolume/lnInclude\
6  -I$FOAM_SRC/fvMotionSolver/lnInclude\
7
8  LIB_LIBS = \
9  -ltriSurface\
10  -lmeshTools \
11  -ldynamicMesh \
12  -lfiniteVolume
```

1.2.2 Boundary and initial conditions

We note that one line must be added compared to the file we copied. The reason is that OpenFOAM implicitly includes files from the current library from which we copied the files. Those include-files are no longer in the current library. Now the library can be compiled from the `libMyPolynomVelocity` folder `(\$FOAM RUN/libMyPolynomVelocity)` by typing:

```
wmake libso
```

OpenFOAM needs to be instructed to use our `libMyPolynomVelocity` library. That is done by adding `libs ("libMyPolynomVelocity.so");` at the bottom of `system/controlDict` and creating a boundary field of type `libMyPolynomVelocity` in the `0/pointMotionU` file. An example of a `pointMotionU` file where our mesh boundary condition is applied to the body patch is shown here:

```
.
.
34 body
35 {
36     type libFvMotionSolver;
37     axis (0 0 1);
38     origin (1.5e-3 1.5e-3 0);
39     angle0 0;
40     amplitude 0.5;
41     omega 2094;
42     value uniform (0 0 0);
43 }
.
.
```

The moving boundary in `pointMotionU` file, for more details see movingCone tutorial

Note that this corresponds to the original entries for `angularOscillatingVelocity` boundary conditions but we now use the new type name. In order to only deform the mesh without doing any

flow calculations the `moveDynamicMesh` utility can be used. It is necessary to add a `dynamicMeshDict` file in the constant folder. An example of a dynamicMeshDict is shown below:

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      motionProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dynamicFvMesh dynamicMotionSolverFvMesh;

motionSolverLibs ("libfvMotionSolvers.so");
//solver displacementSBRStress;
//solver displacementLaplacian;
//solver displacementComponentLaplacian
//solver displacementInterpolation
//solver velocityComponentLaplacian x;
solver velocityLaplacian;

diffusivity  uniform;
//diffusivity  directional (10 .10 1);
//diffusivity  motionDirectional (.10 10 1);
//diffusivity  inverseDistance 1 (body);
//diffusivity  file motionDiffusivity;
//diffusivity  quadratic inverseDistance 1 (body);
//diffusivity  exponential 0.1 inverseDistance 1 (body);
```

1.2.3 The original library

At the moment, our new library is an exact copy of the `angularOscillatingVelocity` library. We will now take a closer look at the library to learn how to modify it. In order to implement a new patch deformation it is neccessary to modify the lines where the input variables, that are read from the `0/pointMotionU` dictionary, are initialised. That is done in `libMyPolynomVelocityPointPatchVectorField.H`, no other modification to the `.H` file is necessary. In this case, the boundary condition will need an axis, origin, base angle, amplitude and frequency.

```
52   public      fixedValuePointPatchField<vector>
53 {
54      // Private data
55
56         vector axis;
57         vector origin;
58         scalar angle0;
59         scalar amplitude;
60         scalar omega;
61
62         pointField p0_;
```

The `libMyPolynomVelocityPointPatchVectorField.C` has four different constructors which all give values to the variables initialised in the .H

le. One of them looks up the values in the 0/pointMotionU, the other give possibilities for initialisation with other methods. These constructors need to be modified to include the input

variables defined in the `.H` file. The second constructor, the one that reads from the dictionary, is shown here:

```
libMyPolynomVelocityPointPatchVectorField::
libMyPolynomVelocityPointPatchVectorField
(
    const pointPatch& p,
    const DimensionedField<vector, pointMesh>& iF,
    const dictionary& dict
)
:
    fixedValuePointPatchField<vector>(p, iF, dict),
    axis_(dict.lookup("axis")),
    origin_(dict.lookup("origin")),
    angle0_(readScalar(dict.lookup("angle0"))),
    amplitude_(readScalar(dict.lookup("amplitude"))),
    omega_(readScalar(dict.lookup("omega")))
{
    if (!dict.found("value"))
    {
        updateCoeffs();
    }

    if (dict.found("p0"))
    {
        p0_ = vectorField("p0", dict , p.size());
    }
    else
    {
        p0_ = p.localPoints();
    }
}
```

We recognize the entries in the `0/pointMotionU` file. The updateCoeffs method under Member Functions is where the calculations for the deformation take place. The "=" operator must be rede ned to include the velocity of the nodes on the deformed patch. The deformation is defined as velocity of nodes at each time step in a particular direction.

```
void angularOscillatingVelocityPointPatchVectorField::updateCoeffs()
{
    if (this->updated())
    {
        return;
    }

    const polyMesh& mesh = this->dimensionedInternalField().mesh()();
    const Time& t = mesh.time();
    const pointPatch& p = this->patch();

    scalar angle = angle0_ + amplitude_*sin(omega_*t.value());
    vector axisHat = axis_/mag(axis_);
    vectorField p0Rel(p0_ - origin_);

    vectorField::operator=
    (
```

```
        (
            p0_
          + p0Rel*(cos(angle) - 1)
          + (axisHat ^ p0Rel*sin(angle))
          + (axisHat & p0Rel)*(1 - cos(angle))*axisHat
          - p.localPoints()
        )/t.deltaTValue()
    );

    fixedValuePointPatchField<vector>::updateCoeffs();
}
```

The write function outputs information regarding the deformation and its control variables:

```
void angularOscillatingVelocityPointPatchVectorField::write
(
    Ostream& os
) const
{
    pointPatchField<vector>::write(os);
    os.writeKeyword("axis")
        << axis_ << token::END_STATEMENT << nl;
    os.writeKeyword("origin")
        << origin_ << token::END_STATEMENT << nl;
    os.writeKeyword("angle0")
        << angle0_ << token::END_STATEMENT << nl;
    os.writeKeyword("amplitude")
        << amplitude_ << token::END_STATEMENT << nl;
    os.writeKeyword("omega")
        << omega_ << token::END_STATEMENT << nl;
    p0_.writeEntry("p0", os);
    writeEntry("value", os);
}
```

1.2.4 The unsteady-periodic patch

Here a patch deformation will be implemented according to a polynomial which constants are given in `0/pointMotionU`. The polynomial here will be second order in both x and y but can easily be changed for another function. The polynomial has the form: $f(x,y) = [X2 \times x^2 + X1 \times x + Y2 \times y^2 + Y1 \times y + Cconst] \times sin(\omega \times t)$ $\omega = 2\pi \times 4 \times defTime$

To be able to describe a surface in any direction by only x and y a new coordinate system is set up that will be used for the polynomial. The `xAxis` and `yAxis` denote the transformation from the fixed coordinate system. The origin denotes the origin of the new coordinate system. `defTime` controls how long time the deformation should take and periodic is set to 1 if the deformation should move back to original position after deformation and then repeat (active flow control). These input values are declared in `libMyPolynomVelocityPointPatchVectorField.H`:

```
    public fixedValuePointPatchField<vector>
{
    // Private data

        vector origin_;
        pointField p0_;
```

```
        scalar X2_;
        scalar X1_;
        scalar Y2_;
        scalar Y1_;
        scalar Cconst_;
        vector xAxis_;
        vector yAxis_;
        scalar periodic_;
        scalar defTime_;
```

The input variables are initialised in the four constructors in `libMyPolynomVelocityPointPatchVectorField.C`. Below is the initialization of the input variables in the first constructor shown. All four constructors should be modified accordingly.

```
libMyPolynomVelocityPointPatchVectorField::
libMyPolynomVelocityPointPatchVectorField
(
    const pointPatch& p,
    const DimensionedField<vector, pointMesh>& iF
)
:
    fixedValuePointPatchField<vector>(p, iF),
    origin_(vector::zero),
    p0_(p.localPoints()),
    X2_(0.0),
    X1_(0.0),
    Y2_(0.0),
    Y1_(0.0),
    Cconst_(0.0),
    xAxis_(vector::zero),
    yAxis_(vector::zero),
    periodic_(0.0),
    defTime_(0.0)
{}
```

Next is where the deformation calculations take place. First the points on the patch relative to the coordinate system of the polynomial are found. These points are then rotated from the fixed coordinate system, $(x; y; z)$, into the coordinate system of the polynomial,$(X; Y; Z)$. The rotation is done using the following definition of Euler angles, where line of nodes N is the intersection between the two coordinate systems xy and XY planes. α is the angle between the $x - axis$ and the line of nodes, β is the angle between the $z - axis$ and the $Z - axis$ and γ is the angle between the line of nodes and the $X - axis$. The rotational matrix is then given as:

$$\underline{\hat{p}} = \underline{p}\mathbf{R}$$

$$= \begin{bmatrix} x & y & z \end{bmatrix} \begin{bmatrix} cos\alpha & -sin\alpha & 0 \\ sin\alpha & cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\beta & -sin\beta \\ 0 & sin\beta & cos\beta \end{bmatrix} \begin{bmatrix} cos\gamma & -sin\gamma & 0 \\ sin\gamma & cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where the leftmost matrix represents a rotation around the $z - axis$ of the original reference frame. The middle matrix represents a rotation around an intermediate $x - axis$ which is the line of nodes, N, and the rightmost matrix represents a rotation around the axis Z of the final reference frame. Carrying out the matrix multiplication gives:

$$\mathbf{R} = \begin{bmatrix} cos\alpha cos\gamma - sin\alpha cos\beta sin\gamma & cos\alpha sin\gamma - sin\alpha cos\beta cos\gamma & sin\beta sin\alpha \\ sin\alpha cos\gamma - cos\alpha cos\beta sin\gamma & sin\alpha sin\gamma - cos\alpha cos\beta cos\gamma & -sin\beta cos\alpha \\ sin\beta sin\gamma & sin\beta cos\gamma & cos\beta \end{bmatrix}$$

In line 178 below the rotation matrix $\mathbf{R}$ is created. The for loop in line 183 rotates the points and creates the plane and rotates it then back to the original coordinate system. The scalar multipl

is used to control the direction of deformation and if the time, `t.value()`, is larger than the defined deformation time, `defTime`, for non periodic deformation it becomes zero. Finally in line 196 the "=" operator is redefined in units of velocity.

```
void libMyPolynomVelocityPointPatchVectorField::updateCoeffs()
{
    if (this->updated())
    {
        return;
    }

    const polyMesh& mesh = this->dimensionedInternalField().mesh()();
    const Time& t = mesh.time();
    const pointPatch& p = this->patch();

    vectorField p0Rel = p0_ - origin_;
    vector zAxis = xAxis_ ^ yAxis_;
    vector xAxisOrg = vector(1, 0, 0);//Original axis used for reference
    vector yAxisOrg = vector(0, 1, 0);
    vector zAxisOrg = vector(0, 0, 1);

    // Euler angles start
    vector Nline = (xAxisOrg ^ yAxisOrg) ^ (xAxis_ ^ yAxis_);
    scalar alpha = acos(xAxisOrg & Nline);///(mag(xAxisOrg)*mag(Nline)));
    scalar beta = acos(zAxisOrg & zAxis);///(mag(zAxisOrg)*mag(zAxis)));
    scalar gamma = acos(Nline & xAxis_);///(mag(Nline)*mag(xAxis_)));
    scalar Rrot1(cos(alpha)*cos(gamma)-sin(alpha)*cos(beta)*sin(gamma));
    scalar Rrot2(-cos(alpha)*sin(gamma)-sin(alpha)*cos(beta)*cos(gamma));
    scalar Rrot3(sin(beta)*sin(alpha));
    scalar Rrot4(sin(alpha)*cos(gamma)+cos(alpha)*cos(beta)*sin(gamma));
    scalar Rrot5(-sin(alpha)*sin(gamma)+cos(alpha)*cos(beta)*cos(gamma));
    scalar Rrot6(-sin(beta)*cos(alpha));
    scalar Rrot7(sin(beta)*sin(gamma));
    scalar Rrot8(sin(beta)*cos(gamma));
    scalar Rrot9(cos(beta));
    // Rotation matrix created
    tensor Rrot(Rrot1, Rrot2, Rrot3, Rrot4, Rrot5, Rrot6, Rrot7, Rrot8, Rrot9);
    tensor RrotInv = inv(Rrot);
    vector p0rot;
    vectorField sd=p0Rel;
    forAll(p0_,iter)
    {
        p0rot = p0Rel[iter] & Rrot; // p relative to new origin rotated
        // Plane from x and y values calculated and inserted into z values
        p0rot = vector(0, 0, X2_*p0rot[0]*p0rot[0]+X1_*p0rot[0]+Y2_*p0rot[1]*p0rot[1]+Y1_*p0rot[1]+Cco
        sd[iter] = p0rot & RrotInv; // Plane rotated back to original position
    };
    scalar multipl = 1;
    if ( periodic_ == 1 ) // For periodic b.c.
    {
        if ((int)floor(t.value()/defTime_)% 2  != 0) multipl = -1; // Revese motion for periodic b.c.
    }
    else if ((periodic_ == 0) && (t.value()> defTime_)) multipl = 0; // No motion
    vectorField::operator=
```

```
    (
        sd *multipl / defTime_
    );

    fixedValuePointPatchField<vector>::updateCoeffs();
}
```

1.2.5 The unsteady periodic motion of the diaphragm

The following case files can be found on the course homepage. A mesh with squared cylinder will be used as an example, see fig 2. The cylinder has the following dimensions: length 20 cm, width 30 cm and height 10 cm. It is fixed to the walls, in the z-direction, by the ends. This mesh is very coarse and will not give correct results but is used in illustrative purpose to show possibilities that patch deformation give. Deformation will be done on top and bottom and then periodic deformation will be added to the top and effects on the flow be compared. The inlet velocity is $Ux = 1m/s$, from the left, and slip conditions on tunnel walls and no slip condition on cylinder. Boundary conditions set in `0/pointMotionU` are:

```
FoamFile
{
    version 2.0;
    format ascii;
    class pointVectorField;
    object pointMotionU;
}

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

dimensions      [0 1 -1 0 0 0 0];

internalField   uniform (0 0 0);

boundaryField
{
    inlet
    {
        type libMyPolynomVelocity;
        origin (0 -0.006 0.0025);
        value uniform (0 0 0);
        X2 -7;
        X1 0;
        Y2 0;
        Y1 0;
        Cconst 0.0028;
        xAxis (1 0 0);
        yAxis (0 0 1);
        periodic 1;
        defTime 0.1;

    }
    outlet
    {
        type            fixedValue;
```

```
        value           uniform (0 0 0);
    }

    wall
    {
        type            fixedValue;
        value           uniform (0 0 0);
    }
    frontAndBack
    {
        type            fixedValue;
        value           uniform (0 0 0);
    }

}
```

The boundary conditions show that the deformation is suppose to take 0.2 s and the shape of the patch should follow the function $f(x; y) = -0.7x^2 + 0.0028$ with the origin in (0 0.006 0.0025) which is the center of the upper patch and they should return to origin and repeat. One period then takes 0.1 s. Figure 4 shows the effect the periodic movement of the diaphragm.

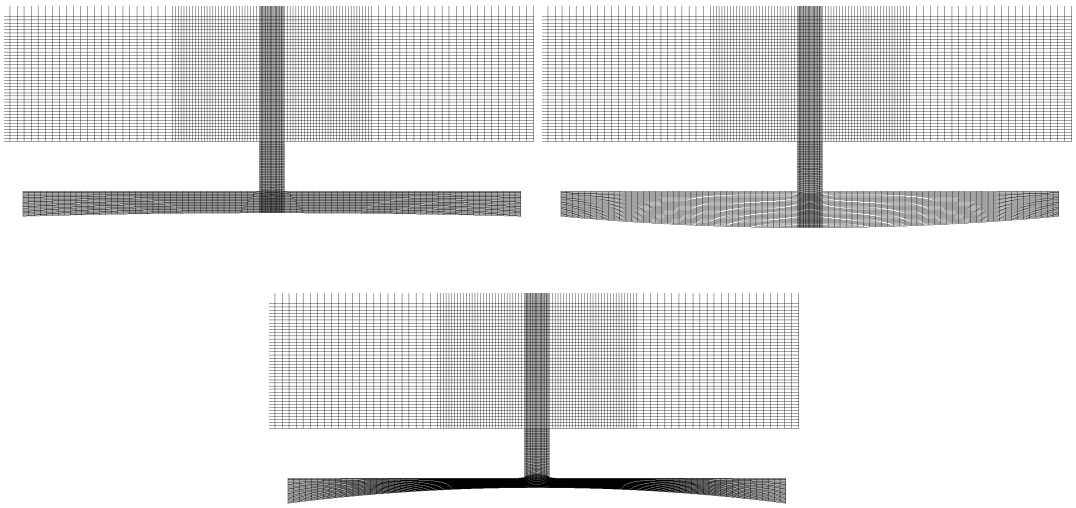

Figure 1.3: The unsteady periodic motion of the diaphragm.