Chalmers, Spring 2009
OpenFOAM

Tutorial: Point-wise deformation of mesh patches

Eysteinn Helgason

1 Introduction

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 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 sides of a cube. 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. The icoDyMFoam solver is used as it deforms the mesh while simultaneously running flow simulation.

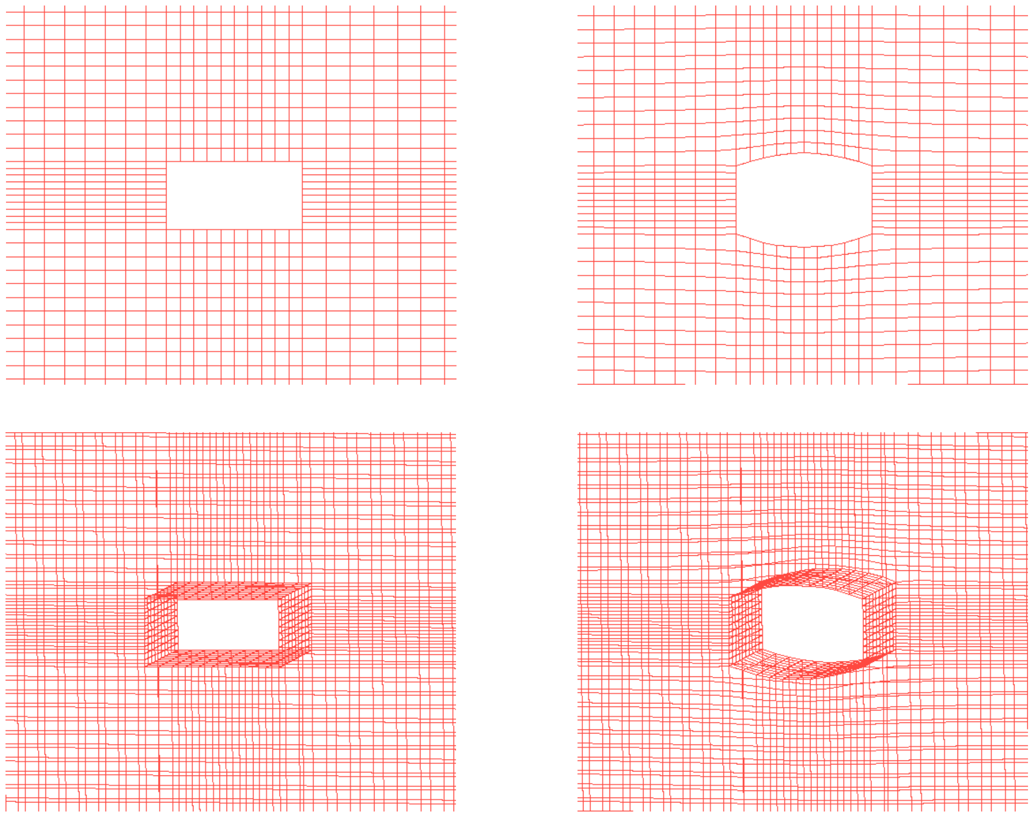


Figure 1: The figures on the left show a mesh with out deformation. The figures on the right show an example of point-wise deformation.

2 Point-wise deformation

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 defining velocity of each node. It can be found in:

```
$FOAM_SRC/fvMotionSolver/pointPatchFields/derived/angularOscillatingVelocity/
```

Let's begin by copying it to our working directory:

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

and also copy the `Make` folder:

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

Clean up:

```
cd $FOAM_RUN/angularOscillatingVelocity
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`.

```
cd $FOAM_RUN
mv angularOscillatingVelocity libMyPolynomVelocity
cd libMyPolynomVelocity
mv angularOscillatingVelocityPointPatchVectorField.C \
  libMyPolynomVelocityPointPatchVectorField.C
mv angularOscillatingVelocityPointPatchVectorField.H \
  libMyPolynomVelocityPointPatchVectorField.H
```

Then it is necessary to edit the `.C` and `.H` files and change all instances of `angularOscillating` to `libMyPolynom`.

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

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:

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

Note the addition `_USER` in line 3, this places the library in the user library directory and makes it impossible for the user to overwrite any original OpenFOAM libraries.

The `Make/options` should include the following:

```
EXE_INC = \
    -I$FOAM_SRC/triSurface/lnInclude \
    -I$FOAM_SRC/meshTools/lnInclude \
    -I$FOAM_SRC/dynamicMesh/lnInclude \
    -I$FOAM_SRC/finiteVolume/lnInclude \
    -I$FOAM_SRC/fvMotionSolver/lnInclude

LIB_LIBS = \
    -ltriSurface \
    -lmeshTools \
    -ldynamicMesh \
    -lfiniteVolume
```

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:

```
/*---------------------------------------------------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  1.5                                   |
|   \\  /    A nd           | Web:      http://www.openfoam.org               |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/

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

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

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

internalField   uniform (0 0 0);

boundaryField
{
    fixedSurroundings
    {
        type            fixedValue;
```

```
        value           uniform (0 0 0);
    }
    movingSurroundings
    {
        type            slip;
    }
    body
    {
        type libMyPolynomVelocity;
        axis (0 0 1);
        origin (1.5e-3 1.5e-3 0);
        angle0 0;
        amplitude 0.5;
        omega 2094;
        value uniform (0 0 0);
    }
}
```

Note that this corresponds to the original entries for `angularOscillatingVelocity` boundary conditions but we now use the new type name.
The solver that will be used is called `icoDyMFoam`, which is a transient solver for incompressible, laminar flow of Newtonian fluids with moving mesh. In order to only deform the mesh without doing any flow calculations the `moveMesh` utility can be used. For `icoDyMFoam` or `moveMesh` to work correctly it is necessary to add a `dynamicMeshDict` file in the `constant` folder. An example of a `dynamicMeshDict` is shown below:

```
/*---------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  1.5                                   |
|   \\  /    A nd           | Web:      http://www.OpenFOAM.org               |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      motionProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dynamicFvMesh dynamicMotionSolverFvMesh;

motionSolverLibs ("libfvMotionSolvers.so");
solver velocityLaplacian;

diffusivity  uniform;
```

2.2 A closer look at the 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.

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

        vector axis_;
        vector origin_;
        scalar angle0_;
        scalar amplitude_;
        scalar omega_;

        pointField p0_;
```

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

```
// * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * //

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.deltaT().value()
    );

    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);
}
```

2.3 Polynomial patch deformation with periodic motion.

The following changes have already been implemented into the library that can be found on the course homepage.
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:

$$z = \mathrm{X2} \cdot x^2 + \mathrm{X1} \cdot x + \mathrm{Y2} \cdot y^2 + \mathrm{Y1} \cdot y + \mathrm{Cconst}$$

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

$$\begin{aligned}
\underline{\hat{p}} &= \underline{p}\mathbf{R} \\
&= [x, y, z] \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\beta & -\sin\beta \\ 0 & \sin\beta & \cos\beta \end{bmatrix} \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix}
\end{aligned}$$

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.

```
146 void libMyPolynomVelocityPointPatchVectorField::updateCoeffs()
147 {
148     if (this->updated())
149     {
150         return;
151     }
152 
153     const polyMesh& mesh = this->dimensionedInternalField().mesh()();
154     const Time& t = mesh.time();
155     const pointPatch& p = this->patch();
156 
157     vectorField p0Rel = p0_ - origin_; // Points relative to new origin
158     vector zAxis = xAxis_ ^ yAxis_;
159     vector xAxisOrg = vector(1, 0, 0);//Original axis used for reference
160     vector yAxisOrg = vector(0, 1, 0);
161     vector zAxisOrg = vector(0, 0, 1);
162 
163     // Euler angles start
164     vector Nline = (xAxisOrg ^ yAxisOrg) ^ (xAxis_ ^ yAxis_);
165     scalar alpha = acos(xAxisOrg & Nline);///(mag(xAxisOrg)*mag(Nline)));
166     scalar beta = acos(zAxisOrg & zAxis);///(mag(zAxisOrg)*mag(zAxis)));
167     scalar gamma = acos(Nline & xAxis_);///(mag(Nline)*mag(xAxis_)));
168     scalar Rrot1(cos(alpha)*cos(gamma)-sin(alpha)*cos(beta)*sin(gamma));
169     scalar Rrot2(-cos(alpha)*sin(gamma)-sin(alpha)*cos(beta)*cos(gamma));
170     scalar Rrot3(sin(beta)*sin(alpha));
171     scalar Rrot4(sin(alpha)*cos(gamma)+cos(alpha)*cos(beta)*sin(gamma));
172     scalar Rrot5(-sin(alpha)*sin(gamma)+cos(alpha)*cos(beta)*cos(gamma));
173     scalar Rrot6(-sin(beta)*cos(alpha));
174     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;
    vector sb = vector(0.5, 0, 0);
    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]+Cconst_);
        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();
}
```

3 Deformation of a square cylinder

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 $U_x = 1\,\mathrm{m/s}$, from the left, and slip conditions on tunnel walls and no slip condition on cylinder. Boundary conditions set in `0/pointMotionU` are:

```
cubeY
   {
        type libMyPolynomVelocity;
        origin (0.7 0.8 0.15);
        value uniform (0 0 0);
        X2 -2;
        X1 0;
        Y2 0;
        Y1 0;
        Cconst 0.02;
        xAxis (1 0 0);
        yAxis (0 0 1);
        periodic 1;
        defTime 0.2;
   }
```

The boundary conditions show that the deformation is suppose to take 0.2 s and the shape of the patch should follow the function $f(x, y) = -2x^2 + 0.02$ with the origin in (0.7 0.6 0.15) which is the center of the upper patch and they should return to origin and repeat. One period then takes 0.4 s. The same boundary conditions are set for the `YMinus` patch of the cylinder except that `periodic` is set to 0. The solver used is `icoDyMFoam`. Figure 3 shows the flow for the original mesh, without any deformation, and then for deformed mesh without periodic patch deformation. Figure 4 shows the effect the periodic movement of the patch on top of the cylinder has on the flow.

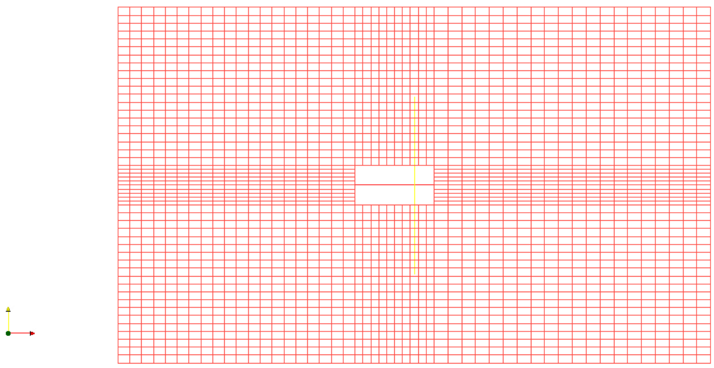
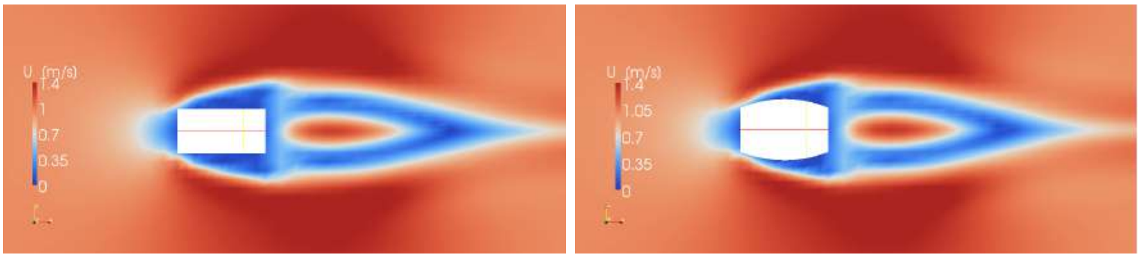
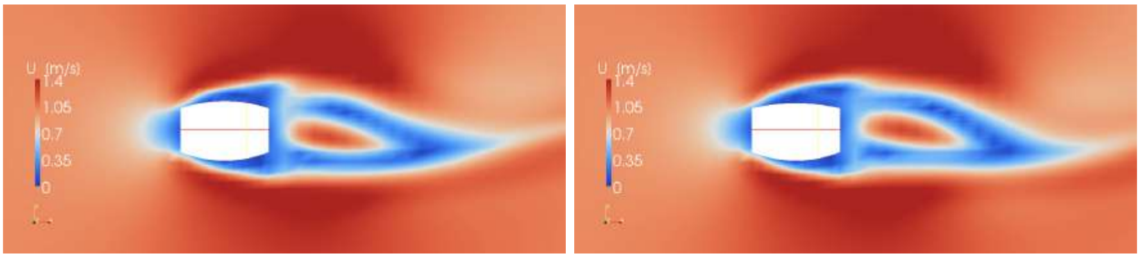


Figure 2: The original mesh, dimensions of the cylinder are length: 20 cm, width 30 cm and height 10 cm.

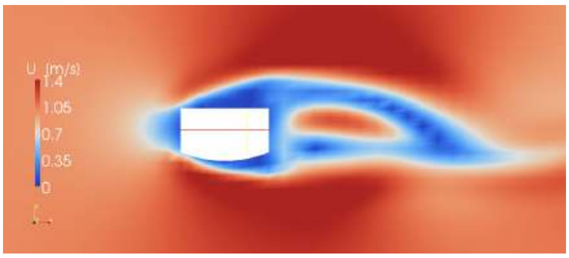


(a) (b)

Figure 3: The inlet velocity is $U_x = 1\,\mathrm{m/s}$ from left. The images show U[m/s] after 2 s for: (a) the original mesh without any deformation, (b) deformed mesh but no periodic boundary.



(a) (b)



(c)

Figure 4: The periodic movement of the patch on top of the cylinder affects the flow. The inlet velocity is $U_x = 1\,\mathrm{m/s}$ and the figures are at (a) t = 1.76 s, (b) t = 1.88 s, (c) t= 2.00 s.