Chalmers University of Technology

CFD with OpenSource software, Project

Patch deformation of a divergent-convergent nozzle

Developed for OpenFOAM-1.5-dev

Author:
Daniel Grönberg

Peer reviewed by:
Jelena Andric
Johan Pilqvist

October 30, 2010

1 Tutorial divergent-convergent nozzle

1.1 Introduction

This tutorial describes how to implement a boundary condition that deforms a patch axisymmetrical at a divergent-convergent nozzle, and also make the internal mesh follow accordingly. It is done to make a smooth transition in the expansion and compression of the nozzle. The new boundary condition can be used at any type of cylindrical geometry, and it can be used for example to optimize the flow through a nozzle. The boundary condition deforms patches according to the sinus function

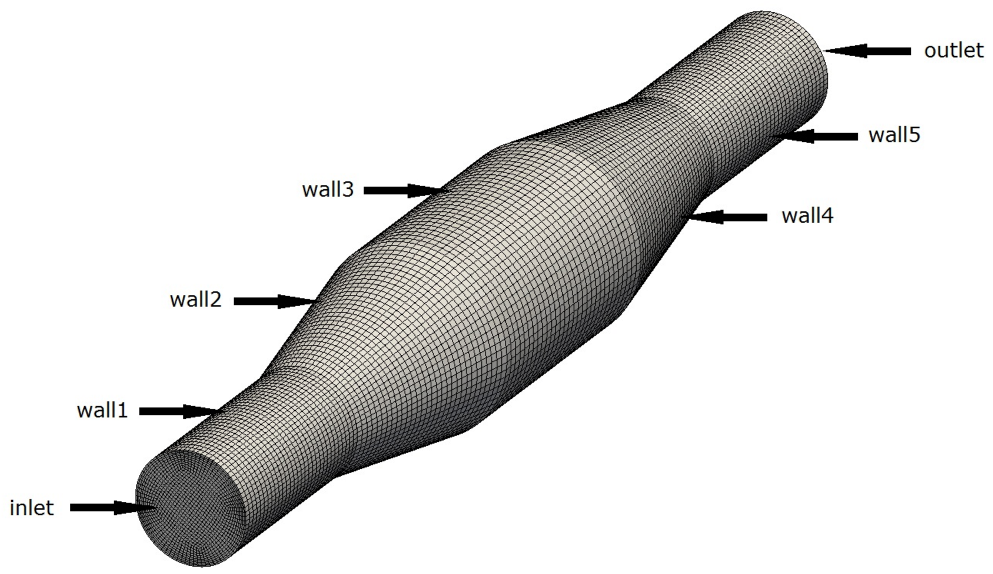


Figure 1: The mesh with the different patches at the divergent-convergent nozzle.

1.2 Pre-processing

1.2.1 Getting started

Start by sourcing the OpenFOAM-1.5-dev version and download the divergent convergent nozzle case into your `$FOAM_RUN` directory. The case can be downloaded from

`http://www.tfd.chalmers.se/~hani/kurser/OS_CFD/`

The geometry is based on the ERCOFTAC conical diffuser Case1 geometry [1][2]. The case is going to use a modified version of the class, `angularOscillatingVelocity`, in order to get the patches to deform as they should. Copy the angularOscillatingVelocity library to the run directory.

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

Rename folder and files

```
cd $FOAM_RUN
mv angularOscillatingVelocity libMySinusDeformationVelocity
cd libMySinusDeformationVelocity
mv angularOscillatingVelocityPointPatchVectorField.C \
libMySinusDeformationVelocityPointPatchVectorField.C
mv angularOscillatingVelocityPointPatchVectorField.H \
libMySinusDeformationVelocityPointPatchVectorField.H
```

Then change ”angularOscillating” to ”libMySinusDeformation” in the `.C`- and `.H` -files

```
sed -e "s/angularOscillating/libMySinusDeformation/g" \
libMySinusDeformationVelocityPointPatchVectorField.C > tmp.C
mv tmp.C libMySinusDeformationVelocityPointPatchVectorField.C
sed -e "s/angularOscillating/libMySinusDeformation/g" \
libMySinusDeformationVelocityPointPatchVectorField.H > tmp.H
mv tmp.H libMySinusDeformationVelocityPointPatchVectorField.H
```

Create the `Make` folder and in it create two files named `files` and `options`. Add the following lines into the `files` -file

```
libMySinusDeformationVelocityPointPatchVectorField.C

LIB = $(FOAM_USER_LIBBIN)/libMySinusDeformationVelocity
```

and then go into the `options` -file and add the code below

```
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 \
    -fvMotionSolver
```

move into `libMySinusDeformationVelocityPointPatchVectorField.H` -file an replace the declared variables

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

        pointField p0_;
```

with the following variables

```
        vector origin_;
        pointField p0_;

        vector axis_;
        vector direction_;
        scalar A_;
        scalar periodic_;
        scalar defTime_;
```

These variables are used to change the different settings that is used for the new boundary condition, these are further explained in section `1.3 Boundary conditions`.
Then move into `libMySinusDeformationVelocityPointPatchVectorField.C` -file and replace all the old declared variables in the Constructors with the new ones

```
libMySinusDeformationVelocityPointPatchVectorField::
libMySinusDeformationVelocityPointPatchVectorField
(
    const pointPatch& p,
    const DimensionedField<vector, pointMesh>& iF
)
:
    fixedValuePointPatchVectorField(p, iF),
    axis_(vector::zero),
    origin_(vector::zero),
    angle0_(0.0),
    amplitude_(0.0),
    omega_(0.0),
    p0_(p.localPoints())
{}
```

should be replaced with the following code

```
libMySinusDeformationVelocityPointPatchVectorField::
libMySinusDeformationVelocityPointPatchVectorField
(
    const pointPatch& p,
    const DimensionedField<vector, pointMesh>& iF
)
:
    fixedValuePointPatchVectorField(p, iF),
    origin_(vector::zero),
    p0_(p.localPoints()),
    axis_(vector::zero),
    direction_(vector::zero),
    A_(0.0),
    periodic_(0.0),
    defTime_(0.0)
{}
```

Repeat in a similar manner for the rest of the Constructors. Replace

```
libMySinusDeformationVelocityPointPatchVectorField::
libMySinusDeformationVelocityPointPatchVectorField
(
    const pointPatch& p,
    const DimensionedField<vector, pointMesh>& iF,
    const dictionary& dict
)
:
    fixedValuePointPatchVectorField(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")))
```

with the code below

```
libMySinusDeformationVelocityPointPatchVectorField::
libMySinusDeformationVelocityPointPatchVectorField
(
    const pointPatch& p,
    const DimensionedField<vector, pointMesh>& iF,
    const dictionary& dict
)
:
    fixedValuePointPatchVectorField(p, iF, dict),
    origin_(dict.lookup("origin")),
    axis_(dict.lookup("axis")),
    direction_(dict.lookup("direction")),
    A_(readScalar(dict.lookup("A"))),
    periodic_(readScalar(dict.lookup("periodic"))),
    defTime_(readScalar(dict.lookup("defTime")))
```

The last two Constructors are ordered in a similar way with the old variables, replace the old code in the two last constructors

```
    axis_(ptf.axis_),
    origin_(ptf.origin_),
    angle0_(ptf.angle0_),
    amplitude_(ptf.amplitude_),
    omega_(ptf.omega_),
    p0_(ptf.p0_)
```

with the following new piece of code

```
    origin_(ptf.origin_),
    p0_(ptf.p0_),
    axis_(ptf.axis_),
    direction_(ptf.direction_),
    A_(ptf.A_),
    periodic_(ptf.periodic_),
    defTime_(ptf.defTime_)
```

All the old variables should now have been replaced with the new ones. Go to the Member functions and replace the following

```
void libMySinusDeformationVelocityPointPatchVectorField::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);
}
```

with this piece of code

```
void libMySinusDeformationVelocityPointPatchVectorField::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("direction")
        << direction_ << token::END_STATEMENT << nl;
    os.writeKeyword("A")
        << A_ << token::END_STATEMENT << nl;
    os.writeKeyword("periodic")
        << periodic_ << token::END_STATEMENT << nl;
    os.writeKeyword("defTime")
        << defTime_ << token::END_STATEMENT << nl;
    p0_.writeEntry("p0", os);
    writeEntry("value", os);
}
```

Now the file is ready for the implementation of the axisymmetric patch deformation.

1.2.2 Implementation of patch deformation

To deform a patch axisymmetrically in three dimensions, the easiest way is to use a cylindrical coordinate system. OpenFOAM uses Cartesian coordinates, therefore a conversion to cylindrical coordinates has to be done. This conversion can be done by yourself or with the help of the class `cylidricalCS`. This tutorial will go through how to convert it using the class `cylidricalCS`. The deformation is going to be done in the radial direction with the displacement `dr`. The sinus function is chosen since it has a shape of the desired deformation that makes the transitions smooth at the expansion and compression regions of the nozzle.

$$dr = A\sin(\frac{2\pi}{L}z - \pi) \tag{1}$$

This function is displayed in Figure 2. The amplitude, `A`, is a user specified value of how much the

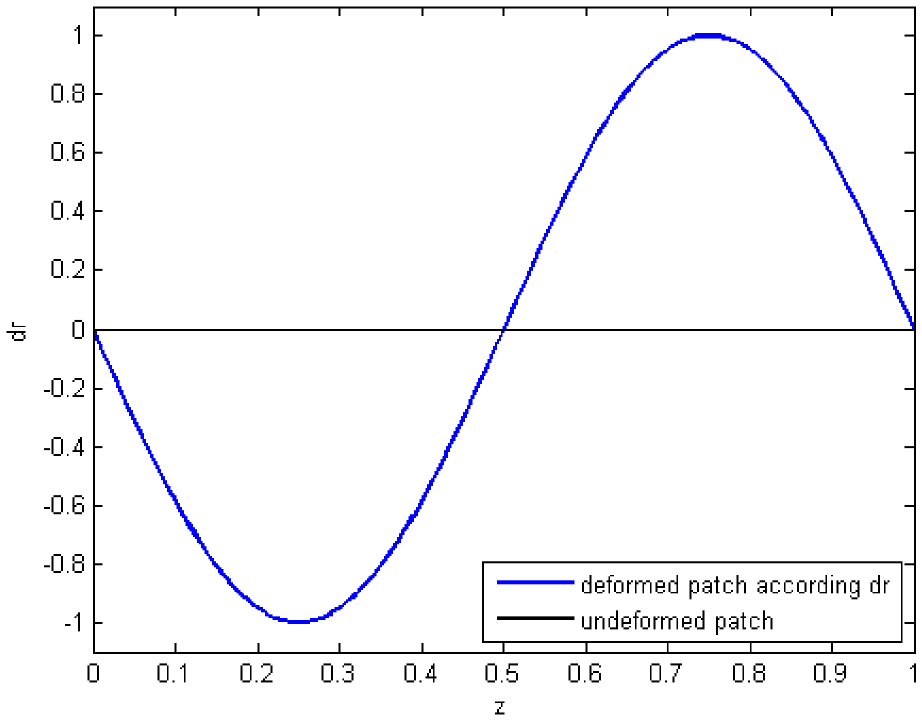


Figure 2: Patch deformation function

patch should deform. `L`, is the length of the patch along the axis in the flow direction. In this case it happens to be along the `z` axis. This is specified so that the patch is deformed according to one period of the function. The function has an angular displacement of the half of a period, so that the function is passing through the origin of the coordinate system.
Move into the `libMySinusDeformationVelocityPointPatchVectorField.C` -file, and start the modifications by adding

```
#include "cylindricalCS.H"
#include "mathematicalConstants.H"
```

at the beginning of the file, in order to use cylindricalCS and mathematical constants, such as π.
Then go to the Member functions section.

```
void libMySinusDeformationVelocityPointPatchVectorField::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()
    );

    fixedValuePointPatchVectorField::updateCoeffs();
}
```

The patch is deformed in this section. So this should be edited according to the following.

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

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

    // Locates the boundaries at the patch
    boundBox bb(p0_, true);

    vectorField p0Rel = p0_ - origin_;
    vector dr;
    vectorField sd=p0Rel;
    vectorField p0Relc=p0Rel;
    scalar L;
    // Defines the patch length
    if ( axis_[0] == 1 || axis_[0] == -1)
    {
        L=bb.max()[0]-bb.min()[0];
    }
    if ( axis_[1] == 1 || axis_[1] == -1)
    {
        L=bb.max()[1]-bb.min()[1];
    }
    if ( axis_[2] == 1 || axis_[2] == -1)
    {
        L=bb.max()[2]-bb.min()[2];
    }

    // Implementing the class cylindricalCS
    cylindricalCS ccs("ccs",origin_,axis_,direction_);

    // Conversion Cartesian coordinates to cylindrical coordinates at the patch
    p0Relc=ccs.localVector(p0Rel);

    forAll(p0_,iter)
    {
        // Deformation according a sinus function in cylindrical coordinates
        dr = vector(A_*sin(((2*mathematicalConstant::pi)/L)*p0Relc[iter][2]-
             mathematicalConstant::pi),p0Relc[iter][1], 0);

        // Conversion from deformation in cylindrical to Cartesian coordinates
        sd[iter] = ccs.globalVector(dr);
    };
    scalar multipl = 1;
    // For periodic b.c.
    if ( periodic_ == 1 )
    {
        // Revese motion for periodic b.c.
        if ((int)floor(t.value()/defTime_)% 2  != 0) multipl = -1;
    }
    // No motion
    else if ((periodic_ == 0) && (t.value()> defTime_)) multipl = 0;
    vectorField::operator=
    (
        sd *multipl / defTime_
    );

    fixedValuePointPatchVectorField::updateCoeffs();
}
```

This field in the `updateCoeffs()` section will update the mesh for every time-step gradually according to the specified deformation function until deformation time is reached. The patch deformation is defined to deform the patch in the chosen flow direction, that is along either `x`, `y` or `z` axis. The origin of the deformation should always be specified at the start of the patch along the specified axis. For example if the patch is starting at the `z`-value 0, where the flow direction is the same as the `z`-axis, then the origin and axis should be specified as `(0 0 0)` and `(0 0 1)` respectively. It should also be mentioned that the deformation will always act in the normal direction of the patch. The variable `p0_` is a point field, which will be given all point coordinates of the patch.

```
    cylindricalCS ccs("ccs",origin_,axis_,direction_);
```

The line above is a constructor where the object ccs of the class cylindricalCS is made with the arguments in the brackets to define how the cylindrical coordinate system should be constructed. A deeper investigation of how cylindricalCS works will be described in section 2 cylindricalCS. The origin_ and axis_ are stated as above. The direction_ is however, an input of how the angles should be calculated in the cylindricalCS class. With the example above, where the axis is (0 0 1), the direction should be specified as (1 1 0). cylindricalCS uses the direction_ vector to calculate the radial and angular directions. The cylindrical coordinate system will begin at the angle of 0 degrees on the positive x-axis and move 180 degrees to the negative x-axis when y is larger than 0. When y is smaller than 0, the angles will go from 0 to -180 degrees instead. The radial direction will always originate at the flow direction axis. The conversion from Cartesian to cylindrical coordinates are done at the points of the patch with ccs.localVector.

```
    p0Relc=ccs.localVector(p0Rel);
```

Here the p0Rel is a vectorfield with the stored locations of the points at the patch relative to the specified origin in Cartesian coordinates. The p0Relc is the same as p0Rel but in cylindrical coordinates. The deformation takes place in the radial direction according to the specified sinus function.

```
        dr = vector(A_*sin(((2*mathematicalConstant::pi)/L)*p0Relc[iter][2]-
             mathematicalConstant::pi),p0Relc[iter][1], 0);
```

The deformation will originate from the patch and deform the patch by moving its points by the distance dr according to the function. Variable L is the length of the patch at the flow direction axis. This makes the patch deform exactly as one full period of the specified function. In order for OpenFOAM to deform the patch, the deformation has to be done in Cartesian coordinates. ccs.globalVector will take care of this.

```
        sd[iter] = ccs.globalVector(dr);
```

Here the deformation in cylindrical coordinates will be converted to Cartesian coordinates for every point at the patch by using the class cylindricalCS again. The following piece of code was originally implemented by Helgason, Eysteinn [3], and it offers an opportunity to choose if the patch deformation should be reversed, or, if the deformation should stop, when deformation time is reached. The t.value() is the current time-step.

```
    scalar multipl = 1;
    // For periodic b.c.
    if ( periodic_ == 1 )
    {
        // Revese motion for periodic b.c.
        if ((int)floor(t.value()/defTime_)% 2  != 0) multipl = -1;
    }
    // No motion
    else if ((periodic_ == 0) && (t.value()> defTime_)) multipl = 0;
    vectorField::operator=
    (
        sd *multipl / defTime_
    );
```

If the defTime_ is set to 0.2 and periodic_ is set to 1, then the patch deforms as usual until time 0.2 is reached. Then between 0.2 and 0.4 it is reversed, i.e it goes back to its original position. At 0.4 - 0.6 the patch deforms as usual again, and this is repeated over and over. The `vectorField::operator=` is calculating the deformation by using velocities. Therefore the deformation distance, `sd`, should be divided by the deformation time, `defTime_`, to get the deformation velocities of the points in the patch. The boundary condition is now ready to be compiled. Compile with the following command while standing in the `libMySinusDeformationVelocity` directory.

```
wclean
wmake libso
```

1.3 Boundary conditions

In order to deform the patch as wanted, the new boundary condition has to be set at the patch in `pointMotionU` -file, which determines the velocity of each point as the patch deforms [4]. This file can be found in the `0` directory in the case folder. In this tutorial the boundary condition is set as below for the wall in the expansion of the nozzle. The values can be changed to make the patch deform as wanted.

```
    wall2
    {
        type libMySinusDeformationVelocity;
        origin (0 0 0);
        axis (0 0 1);
        direction (1 1 0);
        A 0.014;
        periodic 0;
        defTime 0.1;
        value uniform (0 0 0);
    }
```

The `origin`, `axis` and `direction` are as explained earlier in section `1.2.2 Implementation of patch deformation`. The amplitude of the sinus function can be controlled with `A`. This variable has a limitation in magnitude, and this magnitude will be different for different cross-section sizes. This is because of a limitation on how much the cell volumes can decrease, and the difficulty for the cell volumes to adapt into the small throat that is created due to the deformation. The sign of the amplitude determines the direction of the function normal to the patch. The `periodic` option can be either `0` or `1`. The `defTime` will control at which time the deformation will be completed. So if `defTime` is `0.1`, the deformation at the patch will start from `0` and deform gradually until time `0.1`, where the patch will have the shape as the specified sinus function. If `periodic` has a value of `1` the deformation will be reversed at time `0.1` and the shape of the patch will return to its original state at time `0.2`. Then the same procedure is repeated, hence a periodic motion is created. The same boundary condition is set for wall4, but with a reversed sign of the amplitude, to get the smooth transition even when the pipe is compressed. The rest of the patches in the `pointMotionU` -file is set to zero, since they are not going to be deformed. In the velocity file, `U`, the inlet velocity is set to 1 m/s and the outlet to velocity gradient of 0. wall1, wall3 and wall5 are set to 0 m/s and act as fixed walls. In the pressure file, `p`, the outlet pressure is set to 0 Pa, and the rest of the patches are set to 0 pressure gradient.

1.4 Initiate solver settings

The last step before the case can be solved, is to edit the `controlDict` -file to look like the following

```
/*--------------------------------*- 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      controlDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

application     icoFoamAutoMotion;

startFrom       startTime;

startTime       0;

stopAt          endTime;

endTime         1;

deltaT          4e-03;

writeControl    timeStep;

writeInterval   4;

purgeWrite      0;

writeFormat     binary;

writePrecision  6;

writeCompression uncompressed;

timeFormat      general;

timePrecision   6;

runTimeModifiable yes;

adjustTimeStep  no;

maxCo           0.2;

libs ("libMySinusDeformationVelocity.so");


// ************************************************************************* //
```

The solver that is going to be used is a modified `icoFoam` solver that can do dynamic meshes, and it is called `icoDyMFoam`. In order to use this, the application should be set to `icoFoamAutoMotion;`. At the last line, the path to the new boundary condition is added in order for the solver to recognize and use it. The `icoDyMFoam` solver also needs an additional file in the `constant` directory called `dynamicMeshDict`, which looks like

```
/*--------------------------------*- 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;

solver velocityLaplacian;

diffusivity  uniform;

// ************************************************************************* //
```

This file basically specifies how the internal mesh should move when the boundary is moving when the icoDyMFoam solver is used. The `dynamicMotionSolverFvMesh` is a mesh-manipulation model that is used for dynamic meshes where the topology of the mesh remains the same. The solver determines which equation that should be solved to create the motion of the mesh. Here it is set to `velocityLaplacian`, which solves the Laplacian of the diffusivity and cell motion velocity [4]. The `velocityLaplacian` uses the `pointMotionU` file to do this. The `diffusivity` is set to `uniform`, which will make the internal mesh deform uniformly when the boundary is moving.

1.5 Running the code

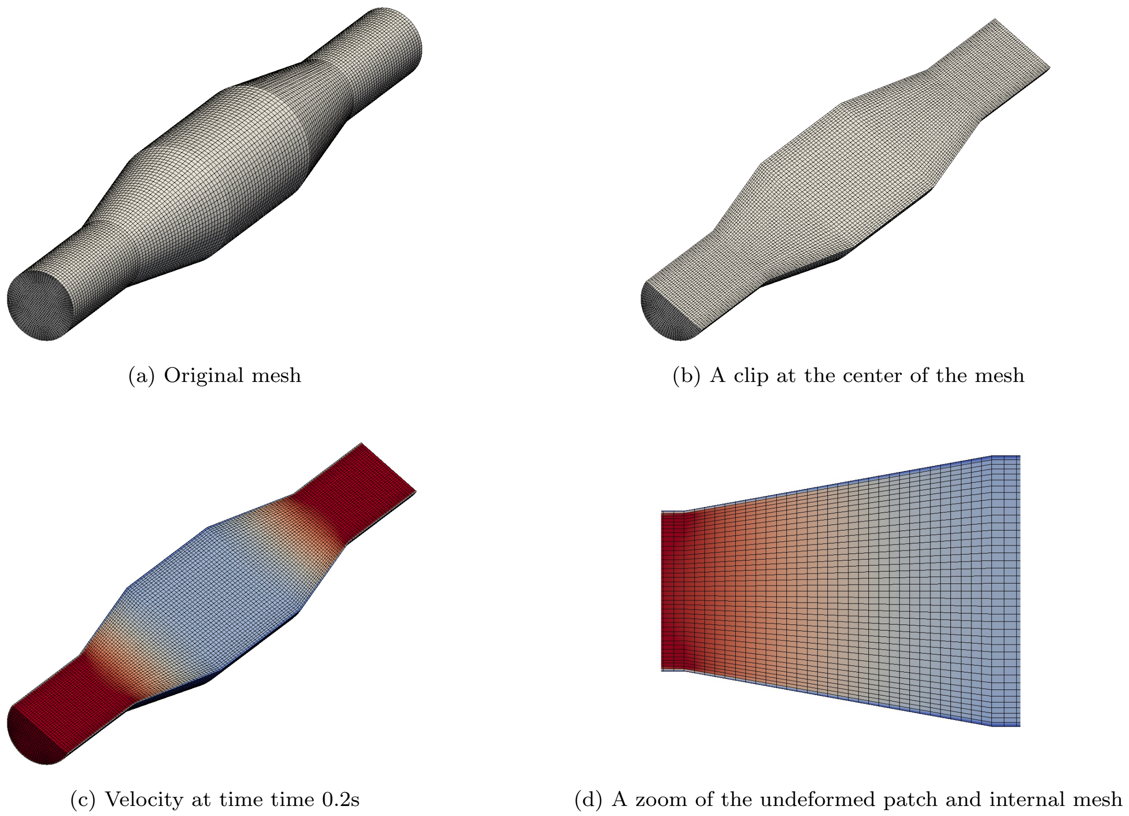
Mesh the geometry and solve the case

```
cd $FOAM_RUN/dcn
blockMesh
icoDyMFoam
```

1.6 Post-processing

Three calculations are going to be presented, one without any deformation the second with a deformation of the expansion and compression walls at the amplitude of 0.014 and the last is performed with the amplitude of 0.04. The deformation time was set to 0.1s and the deformation as non-periodic. Since the result that shows how the fluid behaves is not the main purpose of this tutorial, the velocity pictures were taken after just 0.2s. The velocity figures is presented only to show how the post-processing can be done, therefore no further flow analysis has been done.

1.6.1 Pre-deformation


(a) Original mesh

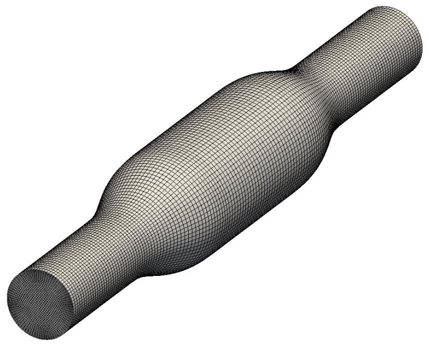
(b) A clip at the center of the mesh

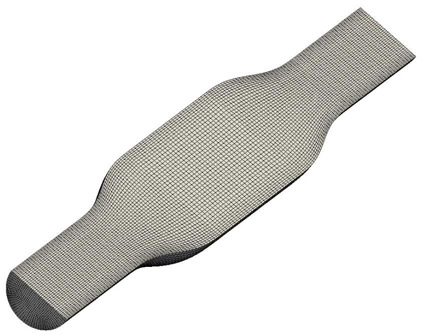
(c) Velocity at time time 0.2s

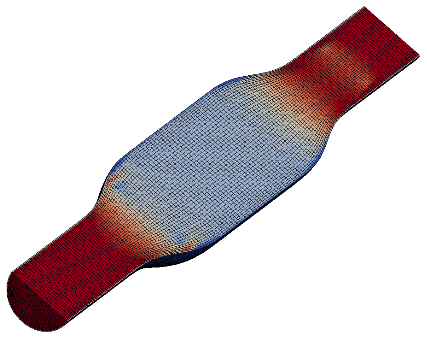
(d) A zoom of the undeformed patch and internal mesh

Figure 3: Undeformed mesh

1.6.2 Deformation with amplitude of 0.014 at time 0.2s


(a) Deformed mesh


(b) A clip at the center of the mesh


(c) Velocity at time time 0.2s

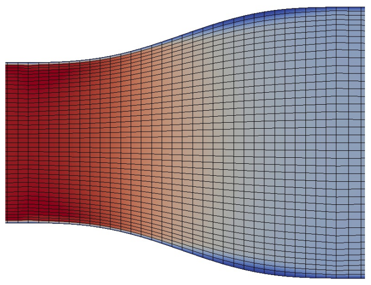
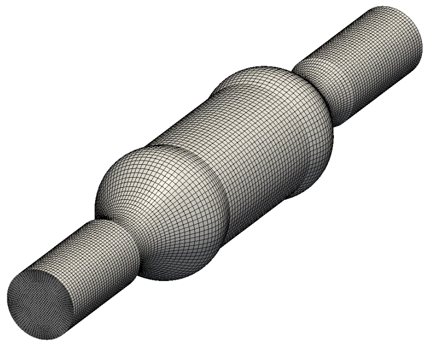
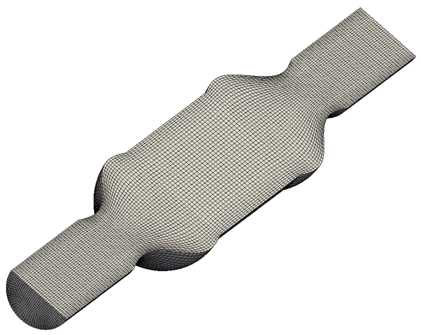
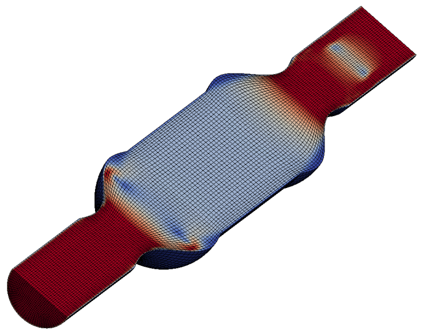

(d) A zoom of the deformed patch and internal mesh

Figure 4: Deformation with an amplitude of 0.014

1.6.3 Deformation with amplitude of 0.04 at time 0.2s


(a) Deformed mesh


(b) A clip at the center of the mesh


(c) Velocity at time time 0.2s

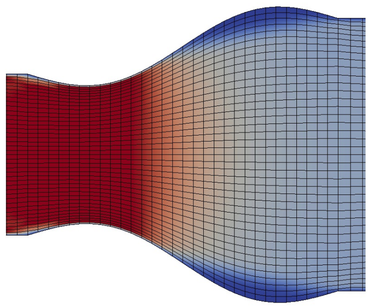

(d) A zoom of the deformed patch and internal mesh

Figure 5: Deformation with an amplitude of 0.04

2 cylindricalCS

This section is dedicated to explain how `cylindricalCS` works, and how it can be implemented in general. First, lets have a look at the constructors in `cylindricalCS.H`

```
    // Constructors

        //- Construct null
        cylindricalCS();

        //- Construct from components
        cylindricalCS
        (
            const word& name,
            const point& origin,
            const vector& axis,
            const vector& direction
        );

        //- Construct from origin and rotation angles
        cylindricalCS
        (
            const word& name,
            const point& origin,
            const coordinateRotation& cr
        );

        //- Construct from dictionary
        cylindricalCS(const word& name, const dictionary& dict);
```

These constructors will define how the cylindrical coordinate system should be built. So if one writes for example:

```
cylindricalCS ccs("ccs",(0 0 0),(0 0 1),(1 1 0);
```

the cylindrical coordinate system will be constructed from an origin and two axes. How the components are stored in `cylindricalCS` is shown in the table below

Input	Type		Stored
"ccs"	word	⇒	name
(0 0 0)	point	⇒	origin
(0 0 1)	vector	⇒	axis
(1 1 0)	vector	⇒	direction

The coordinate system can also be constructed from a dictionary or an origin and rotation. `cylindricalCS` can convert a vector or a vector field, as presented in `1.2.2 Implementation of patch deformation`.

```
p0Relc=ccs.localVector(p0Rel);
```

Here the `p0Rel` is a vector field that is converted to a vector field, `p0Relc`, in cylindrical coordinates. At the top of the `cylindricalCS.H`, the `coordinateSystem.H` -file is included. The `coordinateSystem` class is generally used to rotate coordinate systems. `cylindricalCS` uses it to convert vectors and vector fields from global Cartesian vector to components in local coordinate system. Since the global coordinate system is in Cartesian coordinates, the local coordinate system is going to be converted into cylindrical coordinates. This is done in the following part of the file.

```
        //- Convert from global Cartesian vector to components in
        //  local coordinate system
        vector localVector(const vector& global) const
        {
            return globalToLocal(global, false);
        }

        //- Convert from global Cartesian vector to components in
        //  local coordinate system
        tmp<vectorField> localVector(const vectorField& global) const
        {
            return globalToLocal(global, false);
        }
```

The `ccs.localVector` is written in order to be returned as a `globalToLocal` vector or vector field in the local coordinate system that will become the cylindrical coordinate system. To convert cylindrical vectors or vector fields components to Cartesian coordinate system components, `ccs.globalVector` has to be specified to the vector or vector field in your code. the `coordinateSystem.H` will then convert from local to global coordinate system components, and return a `localToGlobal` vector or vector field. This is displayed below

```
 //- Convert from vector components in local coordinate system
        //  to global Cartesian vector
        vector globalVector(const vector& local) const
        {
            return localToGlobal(local, false);
        }

        //- Convert from vector components in local coordinate system
        //  to global Cartesian vector
        tmp<vectorField> globalVector(const vectorField& local) const
        {
            return localToGlobal(local, false);
        }
```

After this is done the `cylindricalCS` will use these new vectors or vector fields and calculate the components into the wanted coordinate system. This is performed in the Member functions section in the `cylindricalCS.C` -file. The first part shown below will convert from Cartesian to cylindrical coordinates.

```
Foam::vector Foam::cylindricalCS::globalToLocal
(
    const vector& global,
    bool translate
) const
{
    const vector lc =
        coordinateSystem::globalToLocal(global, translate);

    return vector
    (
        sqrt(sqr(lc.x()) + sqr(lc.y())),
        atan2(lc.y(),lc.x())*180.0/mathematicalConstant::pi,
        lc.z()
    );
}
Foam::tmp<Foam::vectorField> Foam::cylindricalCS::globalToLocal
(
    const vectorField& global,
    bool translate
) const
{
    const vectorField lc =
        coordinateSystem::globalToLocal(global, translate);

    tmp<vectorField> tresult(new vectorField(lc.size()));
    vectorField& result = tresult();

    result.replace
    (
        vector::X,
        sqrt(sqr(lc.component(vector::X)) + sqr(lc.component(vector::Y)))
    );

    result.replace
    (
        vector::Y,
        atan2(lc.component(vector::Y), lc.component(vector::X))*
        180.0/mathematicalConstant::pi
    );

    result.replace(vector::Z, lc.component(vector::Z));

    return tresult;
}
```

The `globalToLocal` from `coordinateSystem.H` vector will be placed in vector `lc`. First, the Pythagoras' theorem is used at the local x and y components in order to calculate the radial components. Then arctan will be used at the same components to create the angular direction converted from radians to degrees. The same procedure is done on a vector field. The new vector components are now created and returned into the `globalToLocal` vector that is linked to `cylindricalCS.H`. The conversion from cylindrical to Cartesian components is done in the following part

```
Foam::vector Foam::cylindricalCS::localToGlobal
(
    const vector& local,
    bool translate
) const
{
    scalar theta =
        local.y()*mathematicalConstant::pi/180.0;

    return coordinateSystem::localToGlobal
    (
        vector(local.x()*cos(theta), local.x()*sin(theta), local.z()),
        translate
    );
}
```

```
Foam::tmp<Foam::vectorField> Foam::cylindricalCS::localToGlobal
(
    const vectorField& local,
    bool translate
) const
{
    scalarField theta =
        local.component(vector::Y)*mathematicalConstant::pi/180.0;

    vectorField lc(local.size());
    lc.replace(vector::X, local.component(vector::X)*cos(theta));
    lc.replace(vector::Y, local.component(vector::X)*sin(theta));
    lc.replace(vector::Z, local.component(vector::Z));

    return coordinateSystem::localToGlobal(lc, translate);
}
```

The local `y` component is converted from degrees to radians. Then the Cylindrical components will be converted to components in Cartesian coordinates. As before, the same procedure is done for vector fields. To sum up the previous explanations transformations are presented in the table below.

Input	Type	Coordinatesystem		Type	Coordinatesystem
localVector	vector	Cartesian	$\Rightarrow$	vector	Cylindrical
localVector	vectorField	Cartesian	$\Rightarrow$	vectorField	Cylindrical
globalVector	vector	Cylindrical	$\Rightarrow$	vector	Cartesian
globalVector	vectorField	Cylindrical	$\Rightarrow$	vectorField	Cartesian

3 References

[1] H. Nilsson, M. Page, M. Beaudoin, B. Gschaider and H. Jasak. The openFOAM Turbomachinery Working Group, and Conclusions from the Turbomachinery Session of the Third OpenFOAM Workshop. 24th IAHR Symposium on Hydraulic Machinery and Systems, October 27-31, 2008, Foz Do Iguassu, Brazil.
`http://www.tfd.chalmers.se/~hani/pdf_files/IAHR2008turboWG.pdf`

[2] O. Bounous. Studies of the ERCOFTAC Conical Diffuser with OpenFOAM. Research Report 2008:05, Applied Mechanics, Chalmers University of Technology, Sweden, 2008. Presented at the Third OpenFOAM Workshop in Milano, July 9-11, 2008.
`http://www.tfd.chalmers.se/~hani/pdf_files/OmarReport_Complete.pdf`

[3] Helgason, Eysteinn 2008: ”Point-wise deformation of mesh patches”
`http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2008/EysteinnHelgason/EH_PatchDeformReport.pdf`

[4] Moradnia, Pirooz 2007: ”A tutorial on how to use Dynamic Mesh solver IcoDyMFoam”
`http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2007/PiroozMoradnia/OpenFOAM-rapport.pdf`