Cite as: Vasudevan, S.: Coupling 3D Simulations with 1D Simulations (The Water Hammer Effect). In Proceedings of CFD with OpenSource Software, 2016, Edited by Nilsson. H.,
http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2016

CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Coupling 3D Simulations with 1D Simulations (The Water Hammer Effect)

Developed for OpenFOAM-4.x
Required: MATLAB and Simulink (R2015b), gcc-4.8.5

Author:
Sudharsan Vasudevan
sudvas@student.chalmers.se

Peer reviewed by:
Ebrahim Ghahramani
Håkan Nilsson
Olivier Petit

Licensed under CC-BY-NC-SA, https://creativecommons.org/licenses/

Disclaimer: This is a student project work, done as part of a course where OpenFOAM and some other OpenSource software are introduced to the students. Any reader should be aware that it might not be free of errors. Still, it might be useful for someone who would like learn some details similar to the ones presented in the report and in the accompanying files. The material has gone through a review process. The role of the reviewer is to go through the tutorial and make sure that it works, that it is possible to follow, and to some extent correct the writing. The reviewer has no responsibility for the contents.

January 22, 2017

Learning Outcomes

The main requirements of a tutorial is that it should teach the four points: How to use it, The theory of it, How it is implemented, and How to modify it. Therefore the list of learning outcomes is organized with those headers.

The reader will learn:

How to use it

- How to perform a 3D CFD analysis of water hammer in pipe flow using sonicLiquidFoam solver.
- How to use a coupled 1D 3D solver to analyse water hammer in pipe flow.

Theory of it

- The theory of water hammer.
- How the sonicLiquidFoam 3D CFD solver works.

How it is implemented

- How to implement time varying boundary conditions available in OpenFOAM.
- How to develop a boundary condition that interact with MATLAB/ Simulink, from scratch.

How to modify it

- How to couple a 3D CFD simulation in OpenFOAM with a 1D simulation in Simulink through the boundary conditions that interact with MATLAB/ Simulink.

Contents

1 Water Hammer **3**
- 1.1 Introduction . . . 3
- 1.2 Theoretical Background . . . 3
- 1.3 The sonicLiquidFoam solver . . . 4
- 1.4 Test Case - 3D Simulation with OpenFOAM . . . 6

2 Coupling 1D and 3D Simulations **9**
- 2.1 Coupling OpenFOAM with MATLAB/ Simulink . . . 9
  - 2.1.1 OpenFOAM - MATLAB link . . . 9
  - 2.1.2 Boundary conditions linking OpenFOAM with MATLAB . . . 11
  - 2.1.3 Solver modification . . . 17
- 2.2 Test Cases - Coupled Simulation . . . 18
  - 2.2.1 Test case - 1 . . . 18
  - 2.2.2 Test Case - 2 . . . 23

3 Conclusion **27**

Chapter 1

Water Hammer

1.1 Introduction

The pressure pulsations caused by rapid regulation of flow in a hydraulic system is known as water hammer. Water hammer is a transient phenomenon and results in problems like noise, vibration, pipe collapse etc. Both, 1-dimensional and 3-dimensional simulations are used to analyse water hammer. 1D simulations are computationally efficient. However, they have the shortcoming of not being able to resolve the flow structures in space, in detail. However a 3D CFD simulation is the apt solution for this. On the other hand, it would make the 3D simulations prohibitively computationally expensive to implement all the complex aspects of the hydraulic system. One obvious fix is to take advantage of both these simulation methodologies and develop a coupled 1-D 3-D simulation. The method for coupling a 3-D CFD simulation with a 1-D MOC (Method of Characteristics) simulation to analyse water hammer in pipe flow was established by Wang, Nilsson, Yang and Petit (2016) [1]. Quite similarly, this work is a tutorial on coupling 1D simulations in Simulink with 3D CFD simulations in OpenFOAM to analyse the effects of water hammer in pipe flow. The method of connecting MATLAB with OpenFOAM described by Palm (2012) [2] is used in this project.

1.2 Theoretical Background

The differential pressure induced in a fluid due to a sudden change in momentum causes water hammer. This transient phenomenon can be formulated using Newton's second law.

$$\frac{dp}{dt} = \rho a \frac{dv}{dt} \tag{1.1}$$

Eq. 1.1 is known as the Joukowsky equation. Here, p is the pressure, ρ is density of the fluid, v is the fluid velocity and a is the speed of sound in the fluid. The speed of sound in the fluid can further be expressed as,

$$a = \sqrt{\frac{K}{\rho}} \tag{1.2}$$

Here, K is the bulk modulus, a parameter associated with the compressibility of a fluid. The bulk modulus can in-turn be expressed as,

$$K = \rho \frac{dp}{d\rho} \tag{1.3}$$

Once the bulk modulus is obtained, the speed of sound propagation in the fluid can be estimated. This in turn can be used in the Joukowsky equation to resolve the pressure pulsations. Water hammer and resulting pressure pulsations in a pipe flow will be analysed using a simple 3D CFD simulation in the subsequent sections.

1.3 The sonicLiquidFoam solver

The sonicLiquidFoam is a 3D CFD solver in OpenFOAM and is used in this tutorial to analyse water hammer. sonicLiquidFoam is a transient solver for the laminar flow of a compressible liquid. The solver is based on the PIMPLE algorithm. An excerpt from `sonicLiquidFoam.C` is given below, followed by a brief explanation of the solution algorithm. `sonicLiquidFoam.C` is located at `$FOAM_SOLVERS/compressible/sonicFoam/sonicLiquidFoam` in OpenFOAM.

```
#include "fvCFD.H"
#include "pimpleControl.H"
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
int main(int argc, char *argv[])
{
    #include "postProcess.H"
    #include "setRootCase.H"
    #include "createTime.H"
    #include "createMesh.H"
    #include "createControl.H"
    #include "createFields.H"
    #include "initContinuityErrs.H"
    // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
    Info<< "\nStarting time loop\n" << endl;
    while (runTime.loop())
    {
        Info<< "Time = " << runTime.timeName() << nl << endl;
        #include "compressibleCourantNo.H"
        solve(fvm::ddt(rho) + fvc::div(phi));
        // --- Pressure-velocity PIMPLE corrector loop
        while (pimple.loop())
        {
            fvVectorMatrix UEqn
            (
                fvm::ddt(rho, U)
              + fvm::div(phi, U)
              - fvm::laplacian(mu, U)
            );
            solve(UEqn == -fvc::grad(p));
            // --- Pressure corrector loop
            while (pimple.correct())
            {
                volScalarField rAU("rAU", 1.0/UEqn.A());
                surfaceScalarField rhorAUf
                (
                    "rhorAUf",
                    fvc::interpolate(rho*rAU)
                );
                U = rAU*UEqn.H();
                surfaceScalarField phid
                (
                    "phid",
                    psi
                   *(
                        fvc::flux(U)
                      + rhorAUf*fvc::ddtCorr(rho, U, phi)/fvc::interpolate(rho)
```

```
47                  )
48              );
49              phi = (rho0/psi)*phid;
50              fvScalarMatrix pEqn
51              (
52                  fvm::ddt(psi, p)
53                + fvc::div(phi)
54                + fvm::div(phid, p)
55                - fvm::laplacian(rhorAUf, p)
56              );
57              pEqn.solve();
58              phi += pEqn.flux();
59              solve(fvm::ddt(rho) + fvc::div(phi));
60              #include "compressibleContinuityErrs.H"
61              U -= rAU*fvc::grad(p);
62              U.correctBoundaryConditions();
63          }
64      }
65      rho = rho0 + psi*p;
66      runTime.write();
67      Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
68          << "  ClockTime = " << runTime.elapsedClockTime() << " s"
69          << nl << endl;
70    }
71    Info<< "End\n" << endl;
72    return 0;
73 }
```

Solution algorithm:

1. Begin computation for a time step (Line 15)
2. Solve for the continuity equation (Line 19)

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho u_i}{\partial x_i} = 0 \tag{1.4}$$

3. Begin PIMPLE loop (Line 21)
4. Solve for the momentum equation (Line 29)

$$\frac{\partial \rho u_i}{\partial t} + \frac{\partial}{\partial x_j}(\rho u_i u_j) - \frac{\partial}{\partial x_j}\left(\mu \frac{\partial u_i}{\partial x_j}\right) = -\frac{\partial p}{\partial x_i} \tag{1.5}$$

5. Begin the pressure corrector loop (Line 31)
6. Correct the pressure for the number of times specified in the 'nCorrectors' criteria of the `system\fvSolution` file in the case directory.
7. Repeat steps 4 - 6 the number of times specified as 'nOuterCorrectors' in the `system/fvSolution` file or until the conditions specified in the 'residualControl' (also in the `system/fvSolution` file) are achieved, which ever is earlier.
8. Linearize the variation in density (Line 65) according to

$$\rho = \rho_0 + \psi(p - p_0) \tag{1.6}$$

where, $\psi = \frac{\partial \rho}{\partial p}$, reference pressure, p_0 and reference density, ρ_0 are constants and are specified in the `constant\thermodynamicProperties` file in the case directory.

9. Proceed to the next time step

The quantity ψ is defined as $\psi = \partial\rho/\partial p$ and can be related to the bulk modulus as

$$K = \frac{\rho}{\psi} \tag{1.7}$$

1.4 Test Case - 3D Simulation with OpenFOAM

In this section a 3D CFD simulation of a simple pipe flow will be performed to analyse water hammer.

Files required: All the files required for the entire tutorial are in the `SudharsanV` directory. Download the `SudharsanV.tgz` file to the `$HOME` directory and unpack it using the following command.

```
cd $HOME
tar xzf SudharsanV.tgz
rm SudharsanV.tgz
```

The structure of the directory is given below.

```
SudharsanV
|-- Cases
|.. |-- 1_OF_OFBC
|.. |-- 1_OF_SimulinkBC
|.. |-- Coupled_1D3D
|-- MatlabBC
|-- mySonicLiquidFoam
```

The `Cases` directory contains three test cases, where the first test case (`1_OF_OFBC`) is for the 3D CFD simulation. The `MatlabBC` directory, the `mySonicLiquidFoam` directory along with two other test cases will be required in the next chapter.

Copy the test case for the 3D CFD simulation into the run directory after sourcing OpenFOAM.

```
OF4x
cp -r $HOME/SudharsanV/Cases/1_OF_OFBC $FOAM_RUN
run
cd 1_OF_OFBC
```

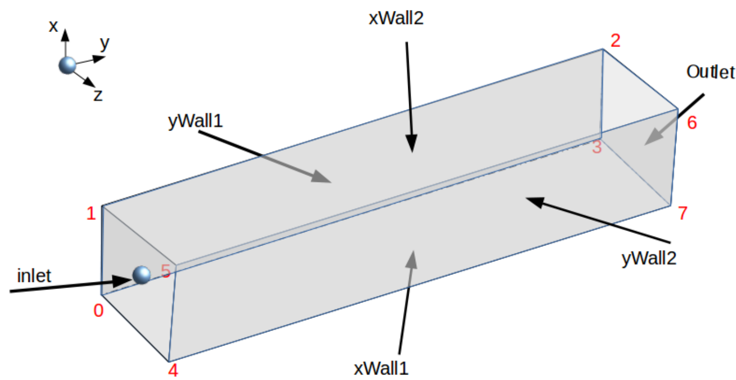

Figure 1.1: Computational Domain (not to scale)

The computational domain consists of a pipe with a square cross-section, see Fig.1.1, where also the vertex numbers and patch names are shown. The flow is along the positive y direction. The pipe is $1000m$ long and has a cross-sectional area of $1m^2$. The domain was meshed with simple hexahedral cells using the blockMesh utility in OpenFOAM. The mesh was generated with 200 cells along the y-axis and 10 cells each along the other two axes.
The boundary conditions for this test case are as follows:
The inlet to the pipe is assumed to be connected to a large reservoir. Therefore the static pressure is specified at the inlet. This value of the static pressure is the hydrostatic pressure corresponding to the location of the pipe with respect to the reservoir's free surface. The velocity at the inlet is specified as zeroGradient, a homogeneous Neumann boundary condition. At the outlet, pressure is specified as zeroGradient (homogeneous Neumann). The outlet boundary condition for velocity is specified as uniformFixedValue. This enables specification of a linear variation of velocity with time. The implementation of this boundary condition in the `0/U` file is given below.

```
outlet
{

    type uniformFixedValue;
    uniformValue tableFile;
    uniformValueCoeffs
    {
        fileName "$FOAM_CASE/time-series";
    }
}
```

Where, the linear variation of velocity is specified in the `time-series` file, placed in the case directory. The `time-series` file reads,

```
(
(0   (0 0 0))
(5  (0 1 0))
(10 (0 1 0))
(14 (0 1 0))
)
```

At the walls, boundary condition for both velocity and pressure are specified as symmetryPlane, implying a symmetric boundary condition.
Since symmetric boundary condition is used for the walls, the simulation would capture only changes along the y-axis. In other words, though the simulation is called a 3D simulation, it captures flow properties only in 1 dimension. Therefore to be more computationally efficient, use only 1 cell in x and z directions (change in the `constant/polyMesh/blockMeshDict` file) and use the 'empty' boundary condition for the walls (change in the `0/U` and `0/p` files).
Create the mesh and run the case.

```
blockMesh
checkMesh
sonicLiquidFoam >& log&
```

The following results can be obtained on successful completion of the simulation.

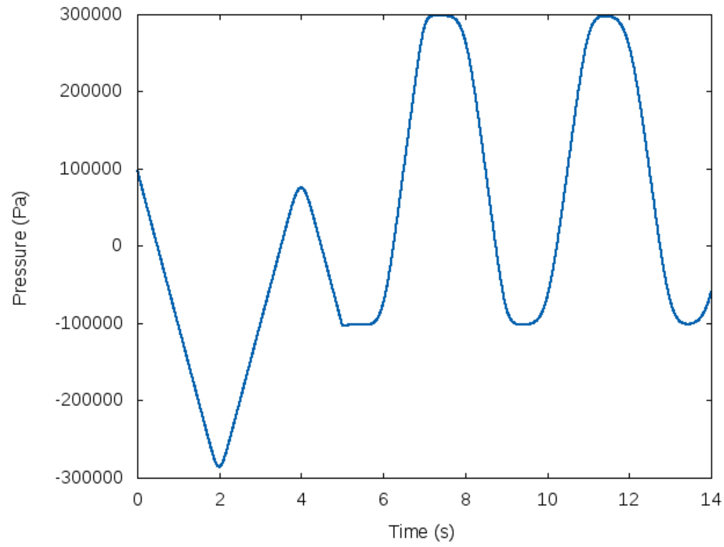

Figure 1.2: Time history of pressure at duct exit

Figure 1.2 is the pressure pulsations at the pipe exit due to water hammer.

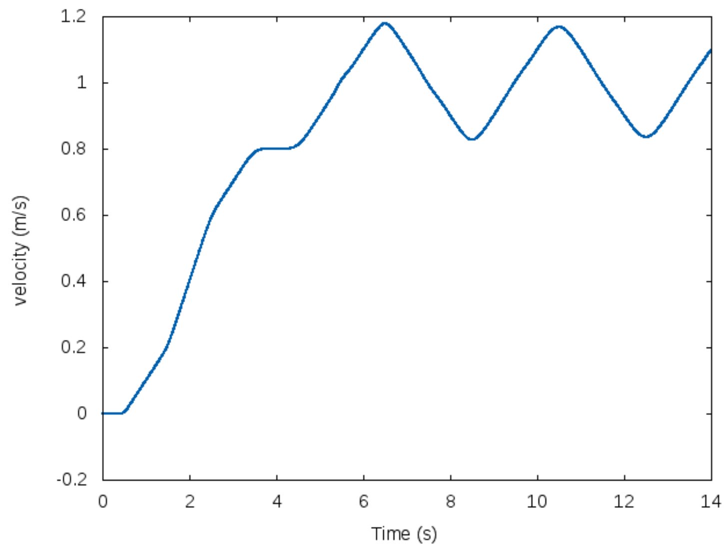

Figure 1.3: Time history of velocity at 0.5L

Figure 1.3 is the fluctuation of velocity at the middle of the pipe (500m downstream of the reservoir).

Chapter 2

Coupling 1D and 3D Simulations

In this chapter, the existing link between OpenFOAM and MATLAB developed by Palm (2012) [2] will be modified to suit current requirement and a link between OpenFOAM and Simulink will be developed. This link will be used in two test cases. The first test case will check if data is exchanged at each time step between OpenFOAM and MATLAB/ Simulink. The second test case will be a coupled 1D 3D simulation of a pipe flow.

2.1 Coupling OpenFOAM with MATLAB/ Simulink

In this section, a two-way coupling between OpenFOAM and MATLAB/ Simulink will be established through two new boundary conditions. One for velocity and one for pressure. These boundary conditions will exchange values of velocity and pressure with MATLAB/ Simulink and set the appropriate values at the patch at every time step.

Important Note: The versions of MATLAB and gcc compiler must be compatible. A simple C++ code for testing the C++ MATLAB linking is provided in the Appendix to this report. The reader is advised to check the compatibility using this simple code after sourcing OpenFOAM as the gcc version used in OpenFOAM may be different from that of the system's default version. To check the gcc version that is being used, type `which gcc` in the terminal window. Not all gcc compilers are compatible with a given version of MATLAB. Also, the library files and directories to be included in the `Make/options` file depend on the version of gcc compiler used. So one has to read through the errors during compilation and modify the instructions for the compiler i.e., `Make/options` file accordingly on a trial and error basis. In this study MATLAB R2015b and gcc-4.8.5 are used and instructions for the compiler are in accordance with these versions. One can always refer to the MathWorks website [3] for details regarding MATLAB versions and the corresponding compatible versions of gcc. The simulations for this study were performed with OpenFOAM and MATLAB installed at Chalmers University of Technology.

2.1.1 OpenFOAM - MATLAB link

The code developed by Palm (2012) [2] had to be modified to suit the current study. The relevant files are provided in the `SudharsanV/MatlabBC/externalPipe` and will be discussed in detail. The files provided here are after incorporating the changes. DO NOT make changes to any of these files. Copy the directory using the following commands

```
OF4x
cp -r $HOME/SudharsanV/MatlabBC/externalPipe $WM_PROJECT_USER_DIR/src
cd $WM_PROJECT_USER_DIR/src/externalPipe
```

The link between OpenFOAM and MATLAB was established using a class named myFirstMatlabPipe. It is to be noted that the word 'pipe' used here (and in externalPipe) is no way connected to a hydraulic pipe. The word 'pipe' here refers to the connection between MATLAB and C++. The important member function in this class is the `matlabCallScript()` function. This function obtains scalar value from MATLAB and returns a scalar value to MATLAB. However, to suit the requirements of the current study, this member function was modified to accept three input arguments and output an array with two elements. The reason for this will be clearly explain when the new boundary conditions are developed. The member function was renamed as `matlabCallScript1()` after the modifications to avoid any ambiguity with the the previous work. Note that the function declaration in `myFirstMatlabPipe.H` and function definition in `myFirstMatlabPipe.C` were modified.
The definition of `matlabCallScript1()` member function can be found in the `myFirstMatlabPipe/myFirstMatlabPipe.C` file in the `externalPipe` directory. The function definition is given below.

```
double* myFirstMatlabPipe::matlabCallScript1
(const char* matlabFilename,double inputArg1,
double inputArg2, double inputArg3) const
{
        // Increase iterator value //
        engEvalString(eMatlabPtr,"ii=ii+1;");

        // Create scalar mxArray object compatible with MATLAB and C++ //

        mxArray *inMxArray1 = mxCreateDoubleMatrix(1,1,mxREAL);
        double *inPtr1 = mxGetPr(inMxArray1);
        // Send value of inMxArray to MATLAB //
        inPtr1[0] = inputArg1;
        engPutVariable(eMatlabPtr,"inputFromCpp1",inMxArray1);

        mxArray *inMxArray2 = mxCreateDoubleMatrix(1,1,mxREAL);
        double *inPtr2 = mxGetPr(inMxArray2);
        // Send value of inMxArray to MATLAB //
        inPtr2[0] = inputArg2;
        engPutVariable(eMatlabPtr,"inputFromCpp2",inMxArray2);


        mxArray *inMxArray3 = mxCreateDoubleMatrix(1,1,mxREAL);
        double *inPtr3 = mxGetPr(inMxArray3);
        // Send value of inMxArray to MATLAB //
        inPtr3[0] = inputArg3;
        engPutVariable(eMatlabPtr,"inputFromCpp3",inMxArray3);

        // Execute MATLAB script //
        engEvalString(eMatlabPtr,matlabFilename);

        // Extract value to C++ and return //
        mxArray *outMxArray1 = engGetVariable(eMatlabPtr,"outputToCpp1");
        double *outPtr1 = mxGetPr(outMxArray1);

        engEvalString(eMatlabPtr,"t_old1=inputFromCpp1");
        return outPtr1;
};

```

In the above code, ii is a counter that increases by one, every time MATLAB is called from Open-

FOAM. Lines 10-14, 16-20 and 23-27 send values to MATLAB from OpenFOAM. Lines 33-34 receive values from MATLAB. The variable t_old1 in line 36 is used to store the time value of the previous time step, as it would be required for the 1-D 3-D coupled simulation. This will be explained in appropriate section later. Also, it is to be noted that the `engEvalString()` is a really important function. The argument provided to it is treated as a MATLAB command. Therefore `engEvalString()` is used to compile and execute MATLAB commands from C++. It is interesting to note that, the name of a MATLAB script file (*.m file) by it self is a command and hence can be executed from OpenFOAM using the `engEvalString` function. Refer to work done by Palm (2012) [2] for a detailed description of the OpenFOAM- MATLAB link.

The `Make` directory is placed in the `externalPipe` directory. The `Make/options` file was also modified to include libraries and directories corresponding to MATLAB R2015b. The `Make/options` file has the following content:

```
EXE_INC = \
          -Wl,-rpath,/chalmers/sw/sup64/matlab-2015b/bin/glnxa64 \
          -I/chalmers/sw/sup64/matlab-2015b/extern/include

LIB_LIBS = \
          -L/chalmers/sw/sup64/matlab-2015b/bin/glnxa64 \
          -leng \
          -lmx
```

If a different version of MATLAB is used, add the corresponding directories and libraries to the `Make/options` file. Compile the myFirstMatlabPipe class from the `externalPipe` directory.

```
wmake libso
```

The complete code of all the files in the `myFirstMatlabpipe` directory is provided in the Appendix to this report.

2.1.2 Boundary conditions linking OpenFOAM with MATLAB

The ultimate aim of this work is to couple a 3D CFD simulation in OpenFOAM with 1D simulation in Simulink. To achieve this the velocity and pressure values need to be exchanged at the coupling interface (boundaries) at each time step.

Boundary conditions for velocity and pressure will be developed with the following logic:

- The velocity boundary condition gets access to the velocity and pressure fields at that time step.
- A scalar value is obtained for each of these fields by averaging (pressure and y-component of velocity), which is in turn given as input to MATLAB along with current time. The current time value along with the time value at the previous time step (stored in the t_old1 variable of `myFirstMatlabPipe::matlabCallScript1()` member function) is used by Simulink for the 1-D part of the simulation.
- Values of velocity and pressure are obtained from MATLAB (after MATLAB/Simulink calculations). Note that this point and the one above provide the reason for modifying the `matlabCallScript()` member function of the `myFirstMatlabPipe` class to include more input arguments and return an array of type double as output.
- The boundary condition for velocity sets the velocity at the patch using the value obtained from MATLAB.

- The pressure boundary condition obtains the value for pressure from the velocity boundary condition and sets pressure in the boundary.

A method for developing these boundary conditions will be discussed. However, the final directories for both these boundary conditions(`timeVaryingFromMatlab` and `timeVaryingFromMatlabScalar`) are provided in the `SudharsanV/MatlabBC` directory.

Boundary condition for the vector field

In this section, a boundary condition for a vector field that interacts with MATLAB will be developed. The template for creating a boundary condition for the vector field (velocity in this case) is obtained by using the following commands.

```
cd $WM_PROJECT_USER_DIR/src
foamNewBC -f -v timeVaryingFromMatlab
```

The -f flag specifies that the boundary condition is of type fixed value. The -v flag specifies that it is for a vector field. This command creates a directory called `timeVaryingFromMatlab` with the following structure:

```
timeVaryingFromMatlab
|-- Make
|   |-- files
|   |-- options
|-- timeVaryingFromMatlabFvPatchVectorField.C
|-- timeVaryingFromMatlabFvPatchVectorField.H
```

Open the directory.

```
cd timeVaryingFromMatlab
```

Open the `timeVaryingFromMatlabFvPatchVectorField.H` file. Include the class definition of OpenFOAM-MATLAB pipe as a header file.

```
...
#include "fixedValueFvPatchFields.H"
#include "Function1.H"
#include "myFirstMatlabPipe.H" //add this line
...
```

Remove the existing private data members and member functions and add the private data given below.

```
    // Private data
        myFirstMatlabPipe mlObj;
        string matlabFile_;
        vector vel_;
        label curTimeIndex_;
        scalar t() const;
        scalar prs_;
    ...
```

Remove the declaration of the mapping functions `autoMap()` and `rmap()`. Include the two access functions as public member functions as given below :

```
    ...
    public:

        vector& vel()
        {
            return vel_;
        }
        scalar prs() const //Access function for the scalar BC
        {
                return prs_;
        }
    ...
```

Save and exit this file.

Both private and public member functions (including constructors) are defined in the `timeVaryingFromMatlabFvPatchVectorField.C` file.
Open the `timeVaryingFromMatlabFvPatchVectorField.C` file. To define the constructor functions, add the following code to the corresponding constructors just after the ':' and before the '{' (after deleting the existing code in those lines)
Constructor-1:

```
    fixedValueFvPatchVectorField(p, iF),
    matlabFile_("MyScript;"),
    vel_(0,0,0),
    curTimeIndex_(-1),
    prs_(0.0)
```

Constructor-2:

```
    fixedValueFvPatchVectorField(p, iF),
    matlabFile_("MyScript;"),//matlabFile_(dict.lookup("matlabFile")),
    vel_(0,0,0),
    curTimeIndex_(-1),
    prs_(0.0)
```

Constructor-2 is used to initialize the data members of the class when the boundary condition is set through the `0/U` file in the case directory.

Constructor-3:

```
    fixedValueFvPatchVectorField(ptf, p, iF, mapper),
    vel_(ptf.vel_),
    curTimeIndex_(-1),
    prs_(ptf.prs_)
```

Constructor-4:

```
    fixedValueFvPatchVectorField(ptf),
    vel_(ptf.vel_),
    curTimeIndex_(-1),
    prs_(ptf.prs_)
```

Constructor-5:

```
    fixedValueFvPatchVectorField(ptf, iF),
    vel_(ptf.vel_),
    curTimeIndex_(-1),
    prs_(ptf.prs_)
```

Remove the definition of the mapping functions `autoMap()` and `rmap()`. The important public member function in the class definition of this boundary condition is the `updateCoeffs()` function that actually computes the boundary condition from a matlab script. Remove the existing definition of the `updateCoeffs()` function and add the following definition.

```
1  if (updated())
2  {
3      return;
4  }
5  if (curTimeIndex_ != this->db().time().timeIndex())
6  {
7      if (t()>0.0)
8      {
9          const fvPatchField<vector>& UPatch1 =
10         patch().lookupPatchField<volVectorField, vector>("U");
11         scalar meanVelocity=gSum(UPatch1.patchInternalField()*
12         mag(patch().Sf())).y() / gSum(mag(patch().Sf()));
13         double UPatch = meanVelocity;
14         Info<<"U_out to Matlab:"<<UPatch<<endl;
15         const fvPatchField<scalar>& pPatch2 =
16         patch().lookupPatchField<volScalarField, scalar>("p");
17         scalar pPatchInternalAverage = gAverage
18         (pPatch2.patchInternalField());
19         double pPatch=pPatchInternalAverage;
20         Info<<"P_out to Matlab:"<<pPatch<<endl;
21         const char *script=matlabFile_.c_str();
22         double* output =
23         mlObj.matlabCallScript1(script,t(),UPatch,pPatch);
24         vel_.y()=output[0];
25         Info<<"U_in from Matlab"<<output[0]<<endl;
26         prs_=output[1];
27         Info<<"P_in from Matlab"<<output[1]<<endl;
28         fixedValueFvPatchField::operator==(vel_);
29     }
30     else
31     {
32         const char *script=matlabFile_.c_str();
33         double* output = mlObj.matlabCallScript1(script,t(),0,0);
34         vel_.y()=output[0];
35         prs_=output[1];
36         fixedValueFvPatchField::operator==(vel_);
37     }
38     curTimeIndex_ = this->db().time().timeIndex();
39 }
40 fixedValueFvPatchField::updateCoeffs();
```

Note that the `updateCoeffs()` function is called for both initialization of the fields in the patch (at $t = 0$) and to calculate the boundary condition at each time step. Clearly the `if (t()>0.0)` statement in Line 7 is employed for the latter. In case of initialization, two dummy values (0 in our

case) are passed to the `matlabCallScript1()` function (Line 33).
In Lines 9 to 19, the pressure and the velocity fields in the first internal cell closest to the patch at the current time is accessed and an average value of pressure and the y-component of velocity is calculated. The `matlabCallScript1()` function belonging to the `myFirstMatlabPipe` class is then called with these averaged values along with current time value and name of the MATLAB script file. Thus the data from OpenFOAM is sent to MATLAB. The values received from MATLAB are stored in the array called `output`. This two-way communication with MATLAB is implemented in Lines 22-23. The values of pressure and velocity obtained from MATLAB are stored in the data members vel_ and prs_. The scalar prs_ can be accessed by the boundary condition for the scalar field (pressure) through the `prs()` access function.
Replace the definition of the `write()` function with,

```
fvPatchVectorField::write(os);
os.writeKeyword("Velocity") << vel_ << token::END_STATEMENT << nl;
writeEntry("value", os);
```

Save and exit this file.
Once the class is defined, the `options` file in the `Make` directory is to be modified to account for the inclusion of directories and libraries related to the MATLAB link. Replace the contents of the `Make/options` file with,

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -Wl,-rpath,/chalmers/sw/sup64/matlab-2015b/bin/glnxa64 \
    -I/chalmers/sw/sup64/matlab-2015b/extern/include \
    -I$(WM_PROJECT_USER_DIR)/src/externalPipe/myFirstMatlabPipe

LIB_LIBS = \
    -lfiniteVolume \
    -lmeshTools \
    -L/chalmers/sw/unsup64/OpenFOAM/ThirdParty-4.x/platforms/linux64/gcc-4.8.5/lib64 \
    -lstdc++ \
    -L/chalmers/sw/sup64/matlab-2015b/bin/glnxa64 \
    -leng \
    -lmx \
    -L$(FOAM_USER_LIBBIN) \
    -lexternalMatlabPipes
```

Compile the boundary condition.

```
wmake libso
```

Boundary condition for the scalar field

In this section the boundary condition for the scalar (pressure in this case) is discussed. The boundary condition obtains the pressure value from the `timeVaryingFromMatlab` boundary condition and sets it in the patch. The procedure for developing the boundary condition from scratch (by defining the data members, constructor and other member functions) is similar to the one explained for `timeVaryingFromMatlab`. For brevity of the report, a detailed procedure is not provided. The directory for this boundary condition with the complete set of files is provided in the `SudharsanV` directory. Note that the template for a fixed value, scalar field boundary condition can be obtained using the following command (Do not execute the command for this tutorial).

```
foamNewBC -f -s timeVaryingFromMatlabScalar
```

The command creates a directory with the structure given below.

```
timeVaryingFromMatlabScalar/
|-- Make
|   |-- files
|   |-- options
|-- timeVaryingFromMatlabScalarFvPatchScalarField.C
|-- timeVaryingFromMatlabScalarFvPatchScalarField.H
```

Copy the directory with all the necessary files for this boundary condition to the `$WM_PROJECT_USER_DIR/src` directory.

```
cd $WM_PROJECT_USER_DIR/src
cp -r $HOME/SudharsanV/MatlabBC/timeVaryingFromMatlabScalar .
cd timeVaryingFromMatlabScalar
```

The important member function in this boundary condition is the `updateCoeffs()` function and it will be explained in detail. It is defined in `timeVaryingFromMatlabScalarFvPatchScalarField.C` file.

```
1  void Foam::timeVaryingFromMatlabScalarFvPatchScalarField::updateCoeffs()
2  {
3      if (updated())
4      {
5          return;
6      }
7      if (curTimeIndex_ != this->db().time().timeIndex())
8      {
9          const fvPatchField<vector>& UPatch1 =
10         this->patch().template lookupPatchField<volVectorField, vector>("U");
11         const timeVaryingFromMatlabFvPatchVectorField& UPatch2 =
12         refCast<const timeVaryingFromMatlabFvPatchVectorField>(UPatch1);
13         scalar pr;
14         pr = UPatch2.prs();
15         pr_=pr;
16         fixedValueFvPatchField::operator==(pr_);
17         curTimeIndex_ = this->db().time().timeIndex();
18     }
19     fixedValueFvPatchField::updateCoeffs();
20 }
```

The `refCast<>()` function defined in `$FOAM_SRC/OpenFOAM/db/typeInfo/typeInfo.H` is used here (line 11-12). This function defines a dynamic casting by passing the reference to an object of a particular class as an argument. To make it more clear, the pressure value for setting the pressure field in the boundary is required. This pressure value is to be obtained from the velocity boundary condition (`timeVaryingFromMatlab`) through its access function `prs()`. However, to access `prs()`, an object of the class `timeVaryingFromMatlabFvPatchVectorField` is required. For this, a reference to its base class (fvPatchField) is created. This is then converted to a reference to the derived class through the `refCast()` function. This type of casting is called ”downcasting” (convert from 'reference-to-base' to 'reference-to-derived'). The pressure value obtained (line 14) is used to set the scalar field in the boundary using the '==' operator (line 16).

Also, note that, the libraries and directories corresponding to the `timeVaryingFromMatlab` boundary condition are included in the `Make/options` file. The `Make/options` file as given below.

```
EXE_INC = \
    ...
    -I$(WM_PROJECT_USER_DIR)/src/externalPipe/myFirstMatlabPipe \
    -I$(WM_PROJECT_USER_DIR)/src/timeVaryingFromMatlab

LIB_LIBS = \
    ...
    -L$(FOAM_USER_LIBBIN) \
    -lexternalMatlabPipes \
    -ltimeVaryingFromMatlab
```

Compile the boundary condition.

```
wmake libso
```

The complete codes for each of the files in the `timeVaryingFromMatlabScalar` directory is provided in the Appendix to this report.

2.1.3 Solver modification

A very small adjustment is to be made to the solver. The solver is constructed in such a way that the momentum equations are solved initially followed by the equation for pressure. The `createFields.H` file, which initializes these fields, initializes pressure ahead of velocity. Since the MATLAB code initializing these fields is called from the velocity boundary condition, it would result in an error. So the order in which the fields were initialized is to be changed in `createFields.H` file. Copy the solver to the `$WM_PROJECT_USER_DIR/applications` directory, rename it and modify the `Make/files` file as given below.

```
cp -r $FOAM_SOLVERS/compressible/sonicFoam/sonicLiquidFoam \
$WM_PROJECT_USER_DIR/applications
cd $WM_PROJECT_USER_DIR/applications
mv sonicLiquidFoam mySonicLiquidFoam
cd mySonicLiquidFoam
mv sonicLiquidFoam.C mySonicLiquidFoam.C
sed -i s/sonicLiquidFoam/mySonicLiquidFoam/g Make/files
sed -i s/FOAM_APPBIN/FOAM_USER_APPBIN/g Make/files
```

Swap the initialization of pressure and velocity fields in `createFields.H` file. The `createFields.H` file should have the fields created in the order given below.

```
#include "readThermodynamicProperties.H"
#include "readTransportProperties.H"
Info<< "Reading field U\n" << endl;
volVectorField U
(
    IOobject
    (
        "U",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
```

```
        IOobject::AUTO_WRITE
    ),
    mesh
);

Info<< "Reading field p\n" << endl;
volScalarField p
(
    IOobject
    (
        "p",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);

...
```

Compile the solver.

```
wmake
```

2.2 Test Cases - Coupled Simulation

In this section, two cases will be tested. The first case is quite similar to the 3D CFD case tested earlier. The difference is, here the outlet boundary conditions will be set using MATLAB/ Simulink. This is a test case to ensure that velocity and pressure values are exchanged between OpenFOAM and MATLAB/ Simulink. The second test case is that of a coupled 1-D 3-D simulation.

2.2.1 Test case - 1

The domain is a square duct of length $1000m$ and cross-sectional area $1m^2$. Upstream of the duct is a reservoir that provides a constant hydrostatic pressure. The variation in outlet velocity with time is set using a simple Simulink signal block as shown in Fig.2.1. Note that, the velocity increases from 0 to 1 m/s in 5s and then remains constant. The pressure at the outlet was set with MATLAB. The MATLAB script is programmed to return the same value that was obtained from OpenFOAM. In a way this would be an explicit implementation of a homogeneous-Neumann boundary condition, since the value from OpenFOAM is of the first internal cell from the patch.

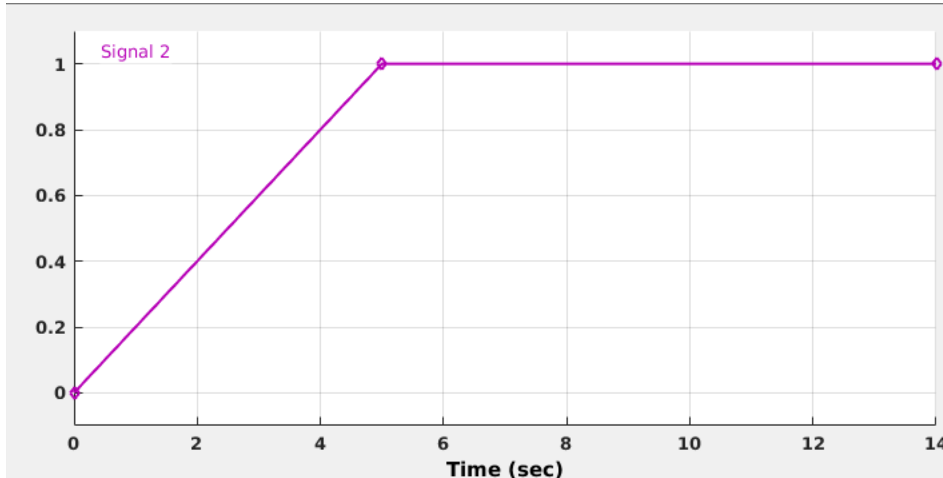

Figure 2.1: Simulink signal for variation of outlet velocity with time

Copy the case directory to the `$FOAM_RUN` directory.

```
OF4x
cp -r $HOME/SudharsanV/Cases/1_OF_SimulinkBC $FOAM_RUN
run
cd 1_OF_SimulinkBC
```

The following changes were made to files in the case directory to incorporate the new boundary conditions.
In the `0/U` file, the boundary condition at the outlet was set as given below

```
outlet
{
    type            timeVaryingFromMatlab;
}
```

Similarly, In the `0/p` file, the boundary condition at the outlet was set to,

```
outlet
{
    type            timeVaryingFromMatlabScalar;
}
```

In the `system/controlDict` file, the following code was added at the end. This is to include the shared object file of the new boundary conditions from `$FOAM_USER_LIBBIN`.

```
libs ("libtimeVaryingFromMatlab.so");
libs ("libtimeVaryingFromMatlabScalar.so");
```

Note that, the following transport and thermodynamic properties are used in this case (Refer to `constant/transportProperties`, `constant/thermodynamicProperties` files in the case directory):

1. Dynamic viscosity (μ): 0.001 $kgm^{-1}s^{-1}$
2. Reference Pressure (p_0): 101325 Pa
3. Reference density (ρ_0): 1000 kgm^{-3}

4. ψ: $1e-06\ s^2m^{-2}$

These values correspond to a speed of sound in water of 1000m/s (refer eq. 1.2 and eq. 1.7). The following MATLAB script (MyScript.m) was used in this test case.

```
1  v_in=inputFromCpp2;
2  if (ii==1)
3      outputToCpp1(1)=0;
4      outputToCpp1(2)=98100;
5      fileID = fopen('vel_in.txt','w');
6      fprintf(fileID,'%6.3f %8.4f %12.4f\n',[0 0 0]);
7      fclose(fileID);
8  else
9      if (ii==2)
10         t_start=0;
11     else
12         t_start=t_old1;
13     end
14     fileID = fopen('vel_in.txt','at');
15     wr=[inputFromCpp1 v_in inputFromCpp3];
16     fprintf(fileID,'%6.3f %8.3f %12.4f\n',wr);
17     fclose(fileID);
18
19     t_end=inputFromCpp1;
20     %Now calling simulink
21     [time,uo,out]=sim('elementary1',[t_start t_end],...
22         simset('OutputVariables', 'ty', 'FinalStateName', 'xFinal'));
23     outputToCpp1(1) = out(end); %Data to OpenFOAM
24     outputToCpp1(2) = inputFromCpp3;
25 end
```

The *if* condition in line 2 corresponds to the first time MATLAB is called from OpenFOAM. This is for initialization of velocity and pressure at the outlet patch. In lines 21-22 Simulink is called to obtain the variation of velocity with time and finally, in lines 23-24, the velocity and pressure values are sent to OpenFOAM.

Also, to ensure that the values sent from OpenFOAM is well received in MATLAB and vice-vera, "Info" statements were provided at appropriate places in the `updateCoeffs()` function of the velocity boundary condition (given below for reference). These will be reflected in the log file while running the simulation. Special key words were used to pick-up these lines using the 'grep' linux command. On the other hand in MATLAB, the incoming values are written to a file (refer to lines 5-7 and 14-17) in the above MATLAB script. These can be compared to check the communication between OpenFOAM and MATLAB.

```
void Foam::timeVaryingFromMatlabFvPatchVectorField::updateCoeffs()
{
    .
    .
    double UPatch = meanVelocity;
    Info<<"U_out to MATLAB:"<<UPatch<<endl;
    .
    double pPatch=pPatchInternalAverage;
    Info<<"P_out to MATLAB:"<<pPatch<<endl;
    .
```

```
    double* output = mlObj.matlabCallScript1(script,t(),UPatch,pPatch);
    Info<<"U_in from MATLAB"<<output[0]<<endl;
    Info<<"P_in from MATLAB"<<output[1]<<endl;
}
```

Subsequently, the key-words- 'U_out', 'P_out','U_in' and 'P_in' can be used to extract these lines using the grep command.
Since symmetric boundary condition is used for the walls, the simulation would capture only changes along the y-axis. In other words, though the simulation is called a 3D simulation, it captures flow properties only in 1 dimension. Therefore to be more computationally efficient, use only 1 cell in x and z directions (change in the `constant/polyMesh/blockMeshDict` file) and use the 'empty' boundary condition for the walls (change in the `0/U` and `0/p` files).
Create the mesh and run the case.

```
blockMesh
checkMesh
mySonicLiquidFoam >& log&
```

After successful completion of the simulation, to extract the pressure values that were given as input to MATLAB, the following linux command.

```
grep 'P_out' log > P_out
```

Compare the values of pressures in the file `P_out` with the file written from MATLAB. Similarly the velocity values can also compared with the following command.

```
grep 'U_out' log > U_out
```

To visualize the results in paraFoam, use the following:

```
paraFoam -builtin
```

The flag '-builtin' ensures that the value in time directories is used for the patch fields instead of computing it all over again.
The following results were obtained for this case.
The time history of static pressure at the exit of the pipe and that of the velocity at a position 500m upstream of the exit (ie., middle of the pipe) are plotted. Also, the results are compared with the ones obtained in the first (3D CFD) test case.

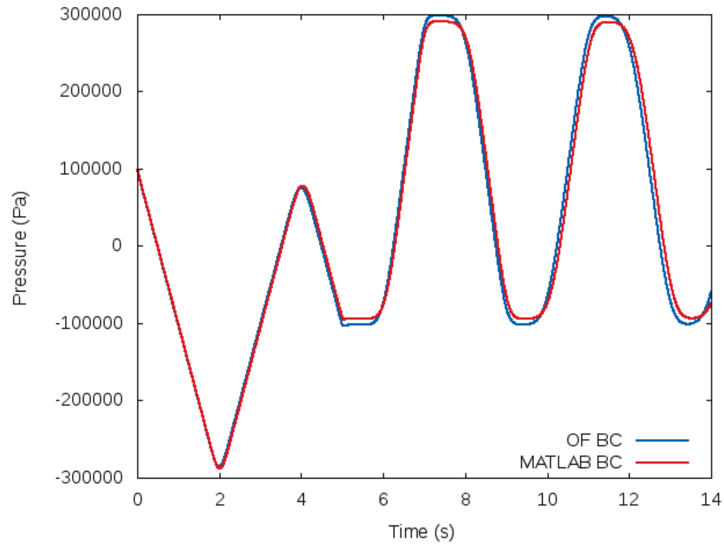

Figure 2.2: Time histories of static pressure at duct exit

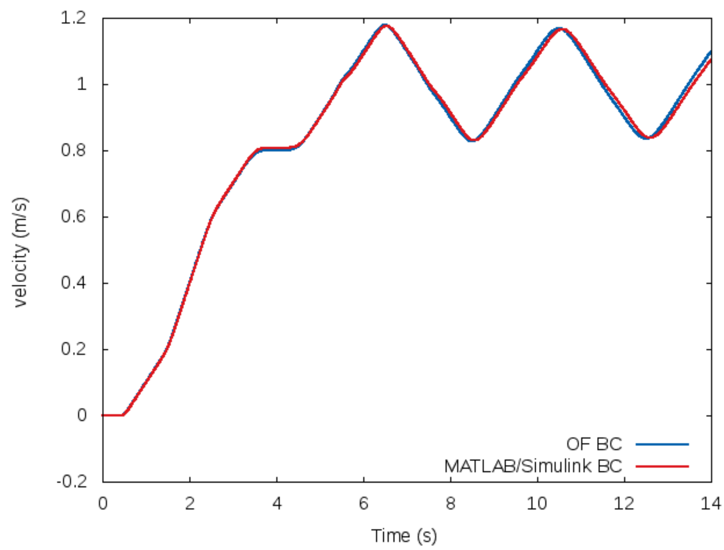

Figure 2.3: Time histories of velocity at 0.5L

The results show that pressure and velocity are exchanged at each time-step between OpenFOAM and MATLAB/Simulink.

2.2.2 Test Case - 2

In this test case a two-way coupling will be established at every time step between a 3D simulation in OpenFOAM and a 1D simulation in Simulink, to analyse water hammer in a pipe flow.
Copy the case directory.

```
OF4x
cp -r $HOME/SudharsanV/Cases/Coupled_1D3D $FOAM_RUN
run
cd Coupled_1D3D $FOAM_RUN
```

An existing Simulink tutorial case [4] on water hammer effect is chosen. The tutorial case is a part of the MATLAB Simulink installation package. This can be accessed by typing `sh_segmented_pipeline_test_rig` in the MATLAB (R2015b) command window. Note that, this command is not the same for all versions of MATLAB/ Simulink (Refer to documentation corresponding to specific version). In this case the water hammer effect is analysed on a pipe of length 25m and diameter of 0.03m. The pipe is of type 'segmented pipeline' and has 5 segments. Upstream of the pipe, is a constant pressure source. The pipe is connected to a valve at its exit. The valve is initially open and the water hammer effect is observed when the valve is controlled by a signal that shuts the valve and reopens it in a very short interval of time (Figure.2.4). However, some small modifications are done to the case to perfectly suit the requirements of the current study. The modifications are,

1. The segmented pipe is reconstructed using the basic blocks (refer to documentation on segmented pipes by MathWorks [5]). This enables the possibility of having a square cross-section.
2. The fluid is changed from 'Hydraulic' to 'Custom Hydraulic' and the parameters like density, bulk modulus and kinematic viscosity are set to match those used in the 3-D simulation and hence retain the same value (1000m/s) of speed of sound in water.
3. The frictional effects are removed in order to stay consistent with the 3-D counterpart.

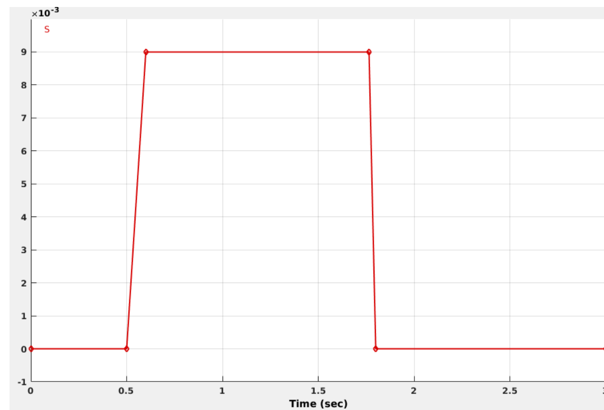

Figure 2.4: Valve control signal (positive value of signal refers to valve closing)

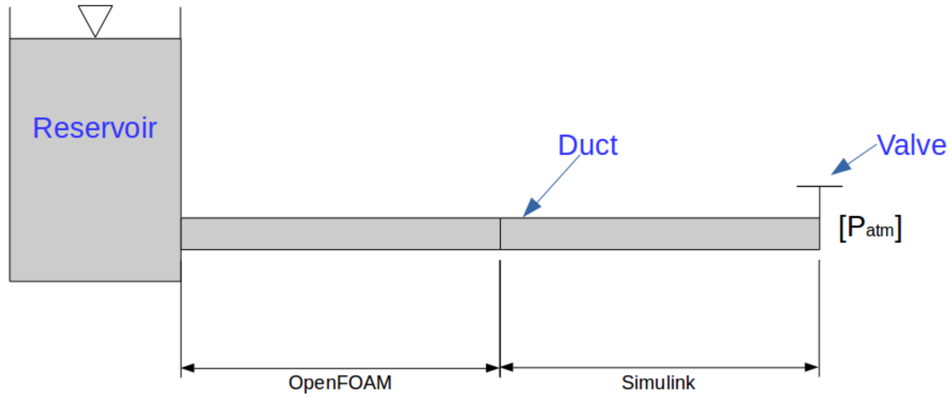

Figure 2.5: The 1D - 3D Configuration

To couple this case with OpenFOAM, the domain is divided into two parts, see Fig. 2.5. The reservoir and the pipe downstream of it (for length 10m) constitutes one part. The remaining pipe of length 15m along with the valve at its exit constitutes the other. The former will be solved in 3D using OpenFOAM and the latter in Simulink. These two simulations are coupled. This coupled domain will be simulated for a total of 3s.

The Simulink model is to be changed for the coupled configuration. Firstly, the length is changed to 15m. The pressure source is programed to accept values from OpenFOAM during each time step. The first segment and the blocks upstream of it are programmed to be initialized with flow rate (velocity) and pressure from OpenFOAM, at each time step. The pressure is probed from segment 1 and the flow rate ($Q = UA$) is probed upstream of segment 1. These velocity and pressure values are returned to OpenFOAM. Additional flow rate and pressure probes are placed to initialize the simulation each time Simulink is called from OpenFOAM.
Note: The final Simulink file, after incorporating the changes is saved as `WaterHammer.slx` and is placed in the case directory. No modification is required to any file in the case directory for this test case.
The coupling is between a 3D and a 1D domain. This certainly implies there is an interface where the exchange of data occurs. In this case it happens to be at the outlet of the 3D domain and at the inlet to the 1D domain. The outlet of the 3D domain is two-dimensional. Therefore data at this side of the interface are 2-D fields. Where as, the 1D side of this interface demands two values (one for velocity and one for pressure). Therefore the average pressure and average y-component of velocity over the area of the outlet patch in the 3D domain is computed and delivered to the 1D simulation as inputs.
The over all working of the coupled simulations is as follows:

1. At time t_n, OpenFOAM delivers velocity and pressure values at the outlet patch corresponding to time t_{n-1} to Simulink.

2. 1D computation is performed in Simulink with these values from time t_{n-1} to t_n. Simulink solves the 1D problem by numerical integration using a number of intermediate time steps between t_{n-1} and t_n.

3. Simulink returns the resulting velocity and pressure values (at t_n) to OpenFOAM.

4. 3D flow computations are performed in OpenFOAM.

Similar to the previous test case, a MATLAB script links OpenFOAM with Simulink. This script named `MyScript.m` is placed in the case directory and is given below for reference.

```
Area=0.0266*0.0266;  %Area of the square pipe
P_OF = inputFromCpp3; %Data From OpenFOAM
Pi=P_OF;
Qi=inputFromCpp2*Area; %Data From OpenFOAM
if (ii==1)  %Initialization of fields
    outputToCpp1(1)=0;
    outputToCpp1(2)=10e5;
    fileID = fopen('p_out.txt','w');
    fprintf(fileID,'%8.4f %12.3f %24.12f\n',[0 0 0]);
    fclose(fileID);
else
    if (ii==2) %First time step of the 3D simulation
        t_start=0;
        P2=P_OF; P3=P_OF; P4=P_OF; P5=P_OF;
        Q2=Qi;Q3=Qi; Q4=Qi;Q5=Qi;

    else  %From the second time step
        t_start=t_old1;
        load('IC.mat')
        P2=ic(1);P3=ic(2);P4=ic(3);P5=ic(4);
        Q2=ic(5);Q3=ic(6);Q4=ic(7);Q5=ic(8);
    end
    t_end=inputFromCpp1;
    %Now calling simulink
    [time,uo,out]=sim('WaterHammer',[t_start t_end],...
        simset('OutputVariables', 'ty', 'FinalStateName', 'xFinal'));
    outputToCpp1(1) = round((out(end,1)/Area),3); %Data to OpenFOAM
    outputToCpp1(2) = out(end,3);
    p_o=[time out(:,2) out(:,12)];
    for i=1:size(p_o,1)
        fileID = fopen('p_out.txt','at');
        fprintf(fileID,'%8.4f %12.3f %24.12f\n',p_o(i,:));
        fclose(fileID);
    end
    ic = out (end,4:11); %Write initial conditions for next time step
    filename = 'IC.mat';
    save (filename,'ic');
end
```

The inputs from OpenFOAM are stored to local MATLAB variables (Lines 2-4). The outlet patch in the 3-D part of the simulation is initialized when $ii = 1$, that is the first time MATLAB is called from OpenFOAM (Lines 5-7). From $ii = 2$, the 1-D part of the simulation computes and returns the boundary value for the outlet of the 3-D part of the simulation. Lines 25-26 call Simulink. In Lines 27-28, velocity and pressure values resulting from the 1-D computations are returned to OpenFOAM. The `fprintf` commands are used to write the pressure at the last segment to a file, which will be discussed as the results for this test case. Note that, at every 3D time step MATLAB is called from OpenFOAM. Simulink called (Lines 25-26) in the Matlab code has no memory of the values corresponding to the previous 3D time step. Hence each of the blocks in Simulink need to be initialized with the values obtained at the end of previous 3D time step. This requires storing the velocity and pressure values in the blocks at the end of each 3D time step (Lines 35-37) and the initialization of the blocks with these values at the beginning of each new 3D time step (Lines 20-21).

Since symmetric boundary condition is used for the walls (in the 3D part of the simulation), the simulation would capture only changes along the y-axis. In other words, though the simulation

is called a 3D simulation, it captures flow properties only in 1 dimension. Therefore to be more computationally efficient, use only 1 cell in x and z directions (change in the `constant/polyMesh/blockMeshDict` file) and use the 'empty' boundary condition for the walls (change in the `0/U` and `0/p` files).
Create the mesh and run the case.

```
blockMesh
checkMesh
mySonicLiquidFoam >& log&
```

The performance of the coupled simulation is compared with pure 1D simulation for this pipe flow test case by comparing time histories of the static pressure at the duct exit as shown in Figure. 2.6.

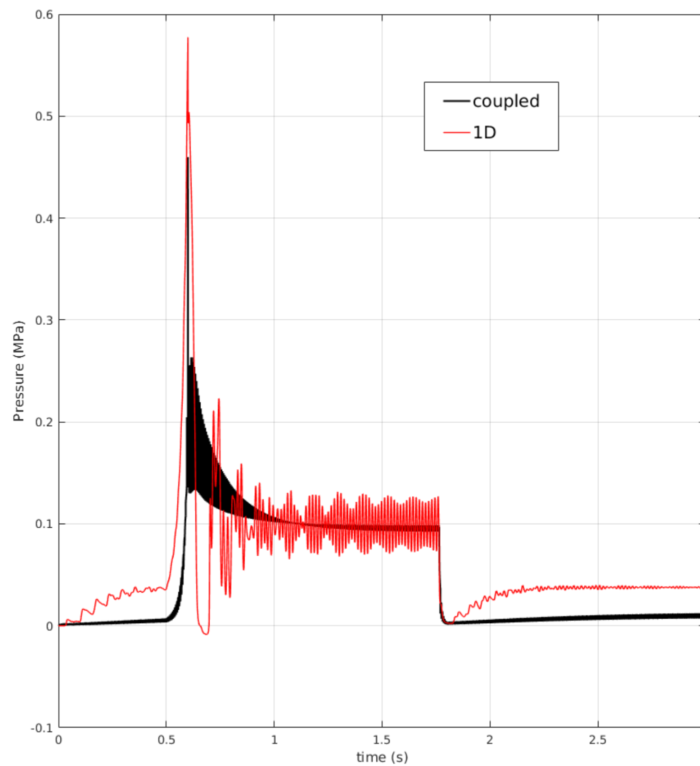

Figure 2.6: Validation of coupled 1D - 3D simulation results (Time History of Pressure)

It is evident that these results are not in perfect agreement with each other. However, the physical phenomena occurring at various important instances are similar. For example, the peak pressure can be observed in both the cases when the valve is shut. Also, the drop in pressure when the valve reopens can be observed in both these configurations.

Chapter 3

Conclusion

A two-way communication between OpenFOAM and MATLAB/ Simulink has been established in this work by means of two new boundary conditions. However, looking at the accuracy of the results, it is quite evident that there is a large scope for improvising the implementation of the coupled 1-D 3-D simulation. Further, the boundary conditions developed in this work `timeVaryingFromMatlab` and `timeVaryingFromMatlabScalar` are generic. That is they are not specific to any solver or any specific scalar or vector flow variable. So these boundary conditions can be put to test in various other applications for coupling OpenFOAM with MATLAB.

Study Questions

1. What causes water hammer?
2. State the Joukowsky equation.
3. Which algorithm is the sonicLiquidFoam solver based on?
4. What is the purpose of the engEvalString function?
5. What is the command to get the template for a fixed value scalar boundary condition?
6. Why is a single value of velocity sent from OpenFOAM to MATLAB/ Simulink at the interface of the coupled simulation (described in the tutorial)?

Bibliography

[1] C. Wang, H. Nilsson, J. Yang, O. Petit, 1D-3D coupling for hydraulic system transient simulations, Computer Physics Communications (2016), http://dx.doi.org/10.1016/j.cpc.2016.09.007

[2] Johannes Palm, Project Work: Connecting OpenFOAM with MATLAB, CFD with opensource software-2012.
http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2012/

[3] Previous Releases: System Requirements and Supported Compilers
se.mathworks.com/support/sysreq/previous_releases.html

[4] Segmented Pipeline Test Rig
https://se.mathworks.com/help/releases/R2015b/physmod/hydro/examples/segmented-pipeline-test-rig.html

[5] Segmented Pipeline in Simulink
https://se.mathworks.com/help/physmod/hydro/ref/segmentedpipeline.html

Appendix

Source Code

File: externalPipe/myFirstMatlabPipe/myFirstMatlabPipe.H

```
// Filename: myFirstMatlabPipe.H //

#ifndef myFirstMatlabPipe_H
#define myFirstMatlabPipe_H

#include "engine.h"
#include "string.h"
using namespace std;
class myFirstMatlabPipe
{
        // Type definition //

        // Private data objects //
        Engine *eMatlabPtr;

        // Number of calltimes //
        int ii;

public:
        // Constructor //
        myFirstMatlabPipe();

        // Destructor //
        virtual ~myFirstMatlabPipe();

        // Send and return a double array to a matlab script //
        virtual double* matlabCallScript1
        (const char* matlabFilename,double inputArg1,
        double inputArg2, double inputArg3) const;

        // Close the pipe to MATLAB //
        virtual void close() const;
};

#endif
```

File: externalPipe/myFirstMatlabPipe/myFirstMatlabPipe.C

```
// Filename: myFirstMatlabPipe.C //

#include<iostream>
#include<cmath>
#include "engine.h"
#include "myFirstMatlabPipe.H"
//
using namespace std;
// Constructor //
myFirstMatlabPipe::myFirstMatlabPipe()
{
         cout << "Matlab engine pointer initialized" << endl;
        // Create matlab engine pointer //
        eMatlabPtr=engOpen(NULL);
        // Create a matlab call no. iterator ii
        engEvalString(eMatlabPtr,"ii=0;");
}
// Destructor //
myFirstMatlabPipe::~myFirstMatlabPipe()
{};

double* myFirstMatlabPipe::matlabCallScript1
(const char* matlabFilename,double inputArg1,
double inputArg2, double inputArg3) const
{
        // Increase iterator value //
        engEvalString(eMatlabPtr,"ii=ii+1;");

        // Create scalar mxArray object compatible with MATLAB and C++ //

         mxArray *inMxArray1 = mxCreateDoubleMatrix(1,1,mxREAL);
         double *inPtr1 = mxGetPr(inMxArray1);
        // Send value of inMxArray to MATLAB //
        inPtr1[0] = inputArg1;
         engPutVariable(eMatlabPtr,"inputFromCpp1",inMxArray1);

        mxArray *inMxArray2 = mxCreateDoubleMatrix(1,1,mxREAL);
        double *inPtr2 = mxGetPr(inMxArray2);
        // Send value of inMxArray to MATLAB //
        inPtr2[0] = inputArg2;
        engPutVariable(eMatlabPtr,"inputFromCpp2",inMxArray2);


        mxArray *inMxArray3 = mxCreateDoubleMatrix(1,1,mxREAL);
        double *inPtr3 = mxGetPr(inMxArray3);
        // Send value of inMxArray to MATLAB //
        inPtr3[0] = inputArg3;
        engPutVariable(eMatlabPtr,"inputFromCpp3",inMxArray3);

        // Execute MATLAB script //
         engEvalString(eMatlabPtr,matlabFilename);
```

```
        // Extract value to C++ and return //
        mxArray *outMxArray1 = engGetVariable(eMatlabPtr,"outputToCpp1");
        double *outPtr1 = mxGetPr(outMxArray1);

        engEvalString(eMatlabPtr,"t_old1=inputFromCpp1");
        return outPtr1;
};

void myFirstMatlabPipe::close() const
{
        engClose(eMatlabPtr);
};
```

File: externalPipe/Make/files

```
myFirstMatlabPipe/myFirstMatlabPipe.C

LIB = $(FOAM_USER_LIBBIN)/libexternalMatlabPipes
```

File: externalPipe/Make/options

```
EXE_INC = \
          -Wl,-rpath,/chalmers/sw/sup64/matlab-2015b/bin/glnxa64 \
              -I/chalmers/sw/sup64/matlab-2015b/extern/include

LIB_LIBS = \
          -L/chalmers/sw/sup64/matlab-2015b/bin/glnxa64 \
          -leng \
          -lmx
```

File: timeVaryingFromMatlab/timeVaryingFromMatlabFvPatchVectorField.H

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2016 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.
```

```
Class
    Foam::timeVaryingFromMatlabFvPatchVectorField

Group
    grpGenericBoundaryConditions

Description
    This boundary condition provides a timeVaryingFromMatlab condition,
    calculated as:

        \f[
            Q = Q_{0} + Q_{p} + s*Q_{t}
        \f]

    where
    \vartable
        s      | single scalar value [units]
        Q_{0}  | single vector value [units]
        Q_{p}  | vector field across patch [units]
        Q_{t}  | vector function of time [units]
    \endtable

Usage
    \table
        Property   | Description                  | Req'd? | Default
        scalarData | single scalar value          | yes    |
        data       | single vector value          | yes    |
        fieldData  | vector field across patch    | yes    |
        timeVsData | vector function of time      | yes    |
        wordData   | word, eg name of data object | no     | wordDefault
    \endtable

    Example of the boundary condition specification:
    \verbatim
    <patchName>
    {
        type       timeVaryingFromMatlab;
        scalarData -1;
        data       (1 0 0);
        fieldData  uniform (3 0 0);
        timeVsData table (
                             (0 (0 0 0))
                             (1 (2 0 0))
                          );
        wordName   anotherName;
        value      uniform (4 0 0); // optional initial value
    }
    \endverbatim

SourceFiles
    timeVaryingFromMatlabFvPatchVectorField.C

\*---------------------------------------------------------------------------*/
```

```
#ifndef timeVaryingFromMatlabFvPatchVectorField_H
#define timeVaryingFromMatlabFvPatchVectorField_H

#include "fixedValueFvPatchFields.H"
#include "myFirstMatlabPipe.H"
#include "Function1.H"

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

namespace Foam
{

/*---------------------------------------------------------------------------*\
     Class timeVaryingFromMatlabFvPatchVectorField Declaration
\*---------------------------------------------------------------------------*/

class timeVaryingFromMatlabFvPatchVectorField
:
    public fixedValueFvPatchVectorField
{
    // Private data


    // Private Member Functions

        // Matla file name
        string matlabFile_;
        myFirstMatlabPipe mlObj;
            // Velocity vector
        vector vel_;
        //current time index
        label curTimeIndex_;
        //- Return current time
        scalar t() const;
        //Pressure
        scalar prs_;

public:

    //- Runtime type information
    TypeName("timeVaryingFromMatlab");

    // Constructors

        //- Construct from patch and internal field
        timeVaryingFromMatlabFvPatchVectorField
        (
            const fvPatch&,
            const DimensionedField<vector, volMesh>&
        );

        //- Construct from patch, internal field and dictionary
        timeVaryingFromMatlabFvPatchVectorField
```

```
(
    const fvPatch&,
    const DimensionedField<vector, volMesh>&,
    const dictionary&
);

//- Construct by mapping given fixedValueTypeFvPatchField
//  onto a new patch
timeVaryingFromMatlabFvPatchVectorField
(
    const timeVaryingFromMatlabFvPatchVectorField&,
    const fvPatch&,
    const DimensionedField<vector, volMesh>&,
    const fvPatchFieldMapper&
);

//- Construct as copy
timeVaryingFromMatlabFvPatchVectorField
(
    const timeVaryingFromMatlabFvPatchVectorField&
);

//- Construct and return a clone
virtual tmp<fvPatchVectorField> clone() const
{
    return tmp<fvPatchVectorField>
    (
        new timeVaryingFromMatlabFvPatchVectorField(*this)
    );
}

//- Construct as copy setting internal field reference
timeVaryingFromMatlabFvPatchVectorField
(
    const timeVaryingFromMatlabFvPatchVectorField&,
    const DimensionedField<vector, volMesh>&
);

//- Construct and return a clone setting internal field reference
virtual tmp<fvPatchVectorField> clone
(
    const DimensionedField<vector, volMesh>& iF
) const
{
    return tmp<fvPatchVectorField>
    (
        new timeVaryingFromMatlabFvPatchVectorField
        (
            *this,
            iF
        )
    );
}
```

```
    // Member functions

        // Evaluation functions

            vector& vel()
            {
                return vel_;
            }
            scalar prs() const
                {
                       return prs_;
                }
            //- Update the coefficients associated with the patch field
            virtual void updateCoeffs();


        //- Write
        virtual void write(Ostream&) const;
};


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

} // End namespace Foam

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


#endif

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

File: timeVaryingFromMatlab/timeVaryingFromMatlabFvPatchVectorField.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2016 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
```

```
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

\*---------------------------------------------------------------------------*/

#include "timeVaryingFromMatlabFvPatchVectorField.H"
#include "addToRunTimeSelectionTable.H"
#include "fvPatchFieldMapper.H"
#include "volFields.H"
#include "surfaceFields.H"

// * * * * * * * * * * * * * Private Member Functions  * * * * * * * * * * * //

Foam::scalar Foam::timeVaryingFromMatlabFvPatchVectorField::t() const
{
    return db().time().timeOutputValue();
}


// * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * //

Foam::timeVaryingFromMatlabFvPatchVectorField::
timeVaryingFromMatlabFvPatchVectorField
(
    const fvPatch& p,
    const DimensionedField<vector, volMesh>& iF
)
:
    fixedValueFvPatchVectorField(p, iF),
    matlabFile_("MyScript;"),
    vel_(0,0,0),
    curTimeIndex_(-1),
    prs_(0.0)
{

        Info<<"Using BC from Matlab"<<endl;
}


Foam::timeVaryingFromMatlabFvPatchVectorField::
timeVaryingFromMatlabFvPatchVectorField
(
    const fvPatch& p,
    const DimensionedField<vector, volMesh>& iF,
    const dictionary& dict
)
:
    fixedValueFvPatchVectorField(p, iF),
    matlabFile_("MyScript;"),//matlabFile_(dict.lookup("matlabFile")),
    vel_(0,0,0),
    curTimeIndex_(-1),
```

```
    prs_(0.0)
{


        Info<<"Using BC from Matlab"<<endl;
    fixedValueFvPatchVectorField::evaluate();

    /*
    //Initialise with the value entry if evaluation is not possible
    fvPatchVectorField::operator=
    (
        vectorField("value", dict, p.size())
    );
    */
}


Foam::timeVaryingFromMatlabFvPatchVectorField::
timeVaryingFromMatlabFvPatchVectorField
(
    const timeVaryingFromMatlabFvPatchVectorField& ptf,
    const fvPatch& p,
    const DimensionedField<vector, volMesh>& iF,
    const fvPatchFieldMapper& mapper
)
:
    fixedValueFvPatchVectorField(ptf, p, iF, mapper),
    vel_(ptf.vel_),
    curTimeIndex_(-1),
    prs_(ptf.prs_)
{}


Foam::timeVaryingFromMatlabFvPatchVectorField::
timeVaryingFromMatlabFvPatchVectorField
(
    const timeVaryingFromMatlabFvPatchVectorField& ptf
)
:
    fixedValueFvPatchVectorField(ptf),
    vel_(ptf.vel_),
    curTimeIndex_(-1),
    prs_(ptf.prs_)
{}


Foam::timeVaryingFromMatlabFvPatchVectorField::
timeVaryingFromMatlabFvPatchVectorField
(
    const timeVaryingFromMatlabFvPatchVectorField& ptf,
    const DimensionedField<vector, volMesh>& iF
)
:
```

```
    fixedValueFvPatchVectorField(ptf, iF),
    vel_(ptf.vel_),
    curTimeIndex_(-1),
    prs_(ptf.prs_)

{}


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

void Foam::timeVaryingFromMatlabFvPatchVectorField::updateCoeffs()
{
    if (updated())
    {
        return;
    }
    if (curTimeIndex_ != this->db().time().timeIndex())
    {

        if (t()>0.0)
        {
                //Read the velocity values from the patch
                const fvPatchField<vector>& UPatch1 =
                patch().lookupPatchField<volVectorField, vector>("U");
                //Compute the average velocity and convert to scalar value
                scalar meanVelocity=gSum(UPatch1.patchInternalField()*
                mag(patch().Sf())).y() / gSum(mag(patch().Sf()));
                double UPatch = meanVelocity;
                Info<<"U_out to Matlab:"<<UPatch<<endl;
                //Read the pressure values from the patch
                const fvPatchField<scalar>& pPatch2 =
                patch().lookupPatchField<volScalarField, scalar>("p");
                //Compute the average pressure and convert to scalar value
                scalar pPatchInternalAverage = gAverage(pPatch2.patchInternalField());
                double pPatch=pPatchInternalAverage;
                Info<<"P_out to Matlab:"<<pPatch<<endl;
                const char *script=matlabFile_.c_str();
                //Call Matlab
                double* output = mlObj.matlabCallScript1(script,t(),UPatch,pPatch);
                vel_.y()=output[0];
                Info<<"U_in from Matlab"<<output[0]<<endl;
                    prs_=output[1];
                Info<<"P_in from Matlab"<<output[1]<<endl;
                fixedValueFvPatchField::operator==(vel_);
        }
        else
        {
                const char *script=matlabFile_.c_str();
                double* output = mlObj.matlabCallScript1(script,t(),0,0);
                vel_.y()=output[0];
                    prs_=output[1];
                fixedValueFvPatchField::operator==(vel_);
        }
```

```
        curTimeIndex_ = this->db().time().timeIndex();
    }
    fixedValueFvPatchField::updateCoeffs();

}


void Foam::timeVaryingFromMatlabFvPatchVectorField::write
(
    Ostream& os
) const
{
    fvPatchVectorField::write(os);
    os.writeKeyword("Velocity") << vel_ << token::END_STATEMENT << nl;
    writeEntry("value", os);
}


// * * * * * * * * * * * * * * Build Macro Function  * * * * * * * * * * * * //

namespace Foam
{
    makePatchTypeField
    (
        fvPatchVectorField,
        timeVaryingFromMatlabFvPatchVectorField
    );
}

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

File: timeVaryingFromMatlab/Make/files

```
timeVaryingFromMatlabFvPatchVectorField.C

LIB = $(FOAM_USER_LIBBIN)/libtimeVaryingFromMatlab
```

File: timeVaryingFromMatlab/Make/options

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -Wl,-rpath,/chalmers/sw/sup64/matlab-2015b/bin/glnxa64 \
    -I/chalmers/sw/sup64/matlab-2015b/extern/include \
    -I$(WM_PROJECT_USER_DIR)/src/externalPipe/myFirstMatlabPipe

LIB_LIBS = \
    -lfiniteVolume \
    -lmeshTools \
    -L/chalmers/sw/unsup64/OpenFOAM/ThirdParty-4.x/platforms/linux64/gcc-4.8.5/lib64 \
    -lstdc++ \
    -L/chalmers/sw/sup64/matlab-2015b/bin/glnxa64 \
    -leng \
```

```
    -lmx \
    -L$(FOAM_USER_LIBBIN) \
    -lexternalMatlabPipes
```

timeVaryingFromMatlabScalar/timeVaryingFromMatlabScalarFvPatchScalarField.H

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2016 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

Class
    Foam::timeVaryingFromMatlabScalarFvPatchScalarField

Group
    grpGenericBoundaryConditions

Description
    This boundary condition provides a timeVaryingFromMatlabScalar condition,
    calculated as:

        \f[
            Q = Q_{0} + Q_{p} + s*Q_{t}
        \f]

    where
    \vartable
        s      | single scalar value [units]
        Q_{0}  | single scalar value [units]
        Q_{p}  | scalar field across patch [units]
        Q_{t}  | scalar function of time [units]
    \endtable

Usage
    \table
        Property   | Description                  | Req'd? | Default
```

```
        scalarData | single scalar value          | yes    |
        data       | single scalar value          | yes    |
        fieldData  | scalar field across patch    | yes    |
        timeVsData | scalar function of time      | yes    |
        wordData   | word, eg name of data object | no     | wordDefault
    \endtable

    Example of the boundary condition specification:
    \verbatim
    <patchName>
    {
        type       timeVaryingFromMatlabScalar;
        scalarData -1;
        data       1;
        fieldData  uniform 3;
        timeVsData table (
                              (0 0)
                              (1 2)
                          );
        wordName   anotherName;
        value      uniform 4; // optional initial value
    }
    \endverbatim

SourceFiles
    timeVaryingFromMatlabScalarFvPatchScalarField.C

\*---------------------------------------------------------------------------*/

#ifndef timeVaryingFromMatlabScalarFvPatchScalarField_H
#define timeVaryingFromMatlabScalarFvPatchScalarField_H

#include "fixedValueFvPatchFields.H"
#include "Function1.H"

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

namespace Foam
{

/*---------------------------------------------------------------------------*\
     Class timeVaryingFromMatlabScalarFvPatchScalarField Declaration
\*---------------------------------------------------------------------------*/

class timeVaryingFromMatlabScalarFvPatchScalarField
:
    public fixedValueFvPatchScalarField
{
    // Private data

        //pressure
        scalar pr_;
        //current time index
```

```
        label curTimeIndex_;



    // Private Member Functions

        //- Return current time
        scalar t() const;


public:

    //- Runtime type information
    TypeName("timeVaryingFromMatlabScalar");


    // Constructors

        //- Construct from patch and internal field
        timeVaryingFromMatlabScalarFvPatchScalarField
        (
            const fvPatch&,
            const DimensionedField<scalar, volMesh>&
        );

        //- Construct from patch, internal field and dictionary
        timeVaryingFromMatlabScalarFvPatchScalarField
        (
            const fvPatch&,
            const DimensionedField<scalar, volMesh>&,
            const dictionary&
        );

        //- Construct by mapping given fixedValueTypeFvPatchField
        //  onto a new patch
        timeVaryingFromMatlabScalarFvPatchScalarField
        (
            const timeVaryingFromMatlabScalarFvPatchScalarField&,
            const fvPatch&,
            const DimensionedField<scalar, volMesh>&,
            const fvPatchFieldMapper&
        );

        //- Construct as copy
        timeVaryingFromMatlabScalarFvPatchScalarField
        (
            const timeVaryingFromMatlabScalarFvPatchScalarField&
        );

        //- Construct and return a clone
        virtual tmp<fvPatchScalarField> clone() const
        {
            return tmp<fvPatchScalarField>
```

```
            (
                new timeVaryingFromMatlabScalarFvPatchScalarField(*this)
            );
        }

        //- Construct as copy setting internal field reference
        timeVaryingFromMatlabScalarFvPatchScalarField
        (
            const timeVaryingFromMatlabScalarFvPatchScalarField&,
            const DimensionedField<scalar, volMesh>&
        );

        //- Construct and return a clone setting internal field reference
        virtual tmp<fvPatchScalarField> clone
        (
            const DimensionedField<scalar, volMesh>& iF
        ) const
        {
            return tmp<fvPatchScalarField>
            (
                new timeVaryingFromMatlabScalarFvPatchScalarField
                (
                    *this,
                    iF
                )
            );
        }


    // Member functions

        // Evaluation functions
            scalar& p()
            {
                return pr_;
            }
            //- Update the coefficients associated with the patch field
            virtual void updateCoeffs();


        //- Write
        virtual void write(Ostream&) const;
};


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

} // End namespace Foam

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


#endif
```

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

timeVaryingFromMatlabScalar/timeVaryingFromMatlabScalarFvPatchScalarField.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2016 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

\*---------------------------------------------------------------------------*/

#include "timeVaryingFromMatlabScalarFvPatchScalarField.H"
#include "timeVaryingFromMatlabFvPatchVectorField.H"
#include "addToRunTimeSelectionTable.H"
#include "fvPatchFieldMapper.H"
#include "volFields.H"
#include "surfaceFields.H"

// * * * * * * * * * * * * * Private Member Functions  * * * * * * * * * * * //

Foam::scalar Foam::timeVaryingFromMatlabScalarFvPatchScalarField::t() const
{
    return db().time().timeOutputValue();
}


// * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * //

Foam::timeVaryingFromMatlabScalarFvPatchScalarField::
timeVaryingFromMatlabScalarFvPatchScalarField
(
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF
)
:
```

```
    fixedValueFvPatchScalarField(p, iF),
    pr_(0),
    curTimeIndex_(-1)
{
}


Foam::timeVaryingFromMatlabScalarFvPatchScalarField::
timeVaryingFromMatlabScalarFvPatchScalarField
(
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF,
    const dictionary& dict
)
:
    fixedValueFvPatchScalarField(p, iF),
    pr_(0),
    curTimeIndex_(-1)

{


    fixedValueFvPatchScalarField::evaluate();

    /*
    //Initialise with the value entry if evaluation is not possible
    fvPatchScalarField::operator=
    (
        scalarField("value", dict, p.size())
    );
    */
}


Foam::timeVaryingFromMatlabScalarFvPatchScalarField::
timeVaryingFromMatlabScalarFvPatchScalarField
(
    const timeVaryingFromMatlabScalarFvPatchScalarField& ptf,
    const fvPatch& p,
    const DimensionedField<scalar, volMesh>& iF,
    const fvPatchFieldMapper& mapper
)
:
    fixedValueFvPatchScalarField(ptf, p, iF, mapper),
    pr_(ptf.pr_),
    curTimeIndex_(-1)

{}


Foam::timeVaryingFromMatlabScalarFvPatchScalarField::
timeVaryingFromMatlabScalarFvPatchScalarField
(
```

```
    const timeVaryingFromMatlabScalarFvPatchScalarField& ptf
)
:
    fixedValueFvPatchScalarField(ptf),
    pr_(ptf.pr_),
    curTimeIndex_(-1)
{}


Foam::timeVaryingFromMatlabScalarFvPatchScalarField::
timeVaryingFromMatlabScalarFvPatchScalarField
(
    const timeVaryingFromMatlabScalarFvPatchScalarField& ptf,
    const DimensionedField<scalar, volMesh>& iF
)
:
    fixedValueFvPatchScalarField(ptf, iF),
    pr_(ptf.pr_),
    curTimeIndex_(-1)
{}


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


void Foam::timeVaryingFromMatlabScalarFvPatchScalarField::updateCoeffs()
{
    if (updated())
    {
        return;
    }
    if (curTimeIndex_ != this->db().time().timeIndex())
    {
        const fvPatchField<vector>& UPatch1 =
        this->patch().template lookupPatchField<volVectorField, vector>("U");
            const timeVaryingFromMatlabFvPatchVectorField& UPatch2 =
            refCast<const timeVaryingFromMatlabFvPatchVectorField>(UPatch1);
            scalar pr;
            pr = UPatch2.prs();
        pr_=pr;
        fixedValueFvPatchField::operator==(pr_);
        curTimeIndex_ = this->db().time().timeIndex();
    }
    fixedValueFvPatchField::updateCoeffs();

}


void Foam::timeVaryingFromMatlabScalarFvPatchScalarField::write
(
    Ostream& os
) const
{
```

```
    fvPatchScalarField::write(os);
    os.writeKeyword("Pressure") << pr_ << token::END_STATEMENT << nl;
    writeEntry("value", os);

}


// * * * * * * * * * * * * * * Build Macro Function  * * * * * * * * * * * * //

namespace Foam
{
    makePatchTypeField
    (
        fvPatchScalarField,
        timeVaryingFromMatlabScalarFvPatchScalarField
    );
}

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

timeVaryingFromMatlabScalar/Make/files

```
timeVaryingFromMatlabScalarFvPatchScalarField.C

LIB = $(FOAM_USER_LIBBIN)/libtimeVaryingFromMatlabScalar
```

timeVaryingFromMatlabScalar/Make/options

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -Wl,-rpath,/chalmers/sw/sup64/matlab-2015b/bin/glnxa64 \
    -I/chalmers/sw/sup64/matlab-2015b/extern/include \
    -I$(WM_PROJECT_USER_DIR)/src/externalPipe/myFirstMatlabPipe \
    -I$(WM_PROJECT_USER_DIR)/src/timeVaryingFromMatlab

LIB_LIBS = \
    -lfiniteVolume \
    -lmeshTools \
    -L/chalmers/sw/unsup64/OpenFOAM/ThirdParty-4.x/platforms/linux64/gcc-4.8.5/lib64 \
    -lstdc++ \
    -L/chalmers/sw/sup64/matlab-2015b/bin/glnxa64 \
    -leng \
    -lmx \
    -L$(FOAM_USER_LIBBIN) \
    -lexternalMatlabPipes \
    -ltimeVaryingFromMatlab
```

Test Matlab gcc compatibility using this simple code:
File Name : Mycase.C

```
#include<iostream>
#include<cmath>
#include "engine.h"
using namespace std;

double* matlabCallScript(const char* matlabFilename,double inputArg)
{

        Engine *eMatlabPtr;
        eMatlabPtr=engOpen(NULL);
        // Increase iterator value //
        //engEvalString(eMatlabPtr,"ii=ii+1;");
        // Create scalar mxArray object compatible with MATLAB and C++ //
        mxArray *inMxArray = mxCreateDoubleMatrix(1,1,mxREAL);
        double *inPtr = mxGetPr(inMxArray);
        // Send value of inMxArray to MATLAB //
        inPtr[0] = inputArg;
        engPutVariable(eMatlabPtr,"inputFromCpp",inMxArray);
        // Execute MATLAB script //
        engEvalString(eMatlabPtr,matlabFilename);
        // Extract value to C++ and return //
        mxArray *outMxArray = engGetVariable(eMatlabPtr,"outputToCpp");
        double *outPtr = mxGetPr(outMxArray);
        engClose(eMatlabPtr);
        return outPtr;
}


int main()
{
        double t = 4;
        const char *Filename = "MyScript;";
        double* Uy = matlabCallScript(Filename,t);
        cout<<Uy[0]<<','<<Uy[1]<<endl;
}
```

File Name: Makefile

```
all:
        gcc -Wall -Wl,-rpath,/chalmers/sw/sup64/matlab-2015b/bin/glnxa64 \
-I/chalmers/sw/sup64/matlab-2015b/extern/include \
-L/chalmers/sw/sup64/matlab-2015b/bin/glnxa64 -leng -lmx \
-L/chalmers/sw/unsup64/OpenFOAM/ThirdParty-4.x/platforms/linux64/gcc-4.8.5/lib64 -lstdc++ \
Mycase.C -o MyCase
```

File Name: MyScript.m

```
t1 = 5;
t2 = 200;
t3 = 300;
Vy1 = 0.5;
Vy2 = 0.75;
Vy3 = 1;
```

```
if (inputFromCpp >= 0 && inputFromCpp < t1)
    outputToCpp(1) = 0;
    outputToCpp(2) = t1;
    else if (inputFromCpp >= t1 && inputFromCpp < t2)
        outputToCpp(1) =  Vy1;
        outputToCpp(2) = t2;
        else if (inputFromCpp >= t2 && inputFromCpp < t3)
        outputToCpp(1) = Vy2;
        outputToCpp(2)=t3;
            else
                outputToCpp(1) = Vy3;
                outpuToCpp(2) = 20;
            end
        end
end
```

Compile using the `make` command
and view the output using `./MyCase`