Cite as: Ramesh Babu, A. : Implementation of Aeroacoustic Solver for weakly compressible flows.
In Proceedings of CFD with OpenSource Software, 2018, Edited by Nilsson .H.,
http://dx.doi.org/10.17196/OS_CFD#YEAR_2018

CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Developed for OpenFOAM v1806

Project work:

Implementation of Aeroacoustic Solver for weakly compressible flows

Author:
Anandh Ramesh Babu
anandhr@student.chalmers.se
Co-supervised by :
Hua-Dong Yao
huadong.yao@chalmers.se

Peer reviewed by:
Mohammad H. Arabnejad
Muye Ge

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.

December 22, 2018

Learning Outcomes

The main requirements of a tutorial are : 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.

How to use it

- The reader will learn how to use rhoPimpleAdiabaticFoam solver.
- The reader will learn how to use the implemented aeroacoustic solver for the flow past a wedge.

The theory behind it

- The reader will learn the theoretical background behind pressure-based wave equations that have to be implemented in the solver.

How it is implemented

- The reader will learn how rhoPimpleAdiabaticFoam works.

How to modify it

- The reader will learn how to implement the aeracoustic solver 'rhoPimpleAdiabaticAcousticFoam'.
- The reader will learn how to modify a case and set the necessary parameters to run aeroacoustic simulations using 'rhoPimpleAdiabaticAcousticFoam'.

Contents

1 Theory **3**
1.1 Introduction . . . 3
1.2 Governing Equations . . . 3

2 rhoPimpleAdiabaticFoam Solver **6**
2.1 rhoPimpleAdiabaticFoam.C . . . 6
2.2 createFields.H . . . 12
2.3 Make folder . . . 14

3 Implementation of Acoustic Solver **15**
3.1 Creating Fields . . . 15
3.2 acousticSolver.H . . . 18

4 Test Case **20**
4.1 system/ Folder . . . 21
4.1.1 blockMeshDict . . . 21
4.1.2 controlDict . . . 22
4.1.3 fvSchemes . . . 23
4.1.4 fvSolution . . . 23
4.2 constant/ folder . . . 23
4.2.1 thermophysicalProperties . . . 23
4.2.2 turbulenceProperties . . . 24
4.2.3 acousticSettings . . . 24
4.3 0/ Folder . . . 24
4.3.1 'p' Field . . . 25
4.3.2 'T' Field . . . 25
4.3.3 'U' Field . . . 26
4.3.4 'pAcoustic' Field . . . 27

5 Results **31**

6 Conclusion **33**

Chapter 1

Theory

1.1 Introduction

Aeroacoustics is the study of flow induced sound. Some examples of aeroacoustic noises include jet engine, wind turbines, wind musical instruments and so on. This study has been mainly practiced to reduce the noise generated from a flow field.

This field of aeroacoustics was pioneered in the 1950's by James Lighthill. The sound is created by turbulent wakes, detached boundary layers, vortex structures in the flow, flow interaction with walls and so on. There are several limitations in the computation of aeroacoustic field due to scaling difficulties, so this field has been heavily reliant on experimental methods. Lighthill developed a wave equation by rewriting the governing flow equations which included source terms from the flow equations[1]. This idea has led to further extensions such as, Curle's analogy, Ffowcs-Williams and Hawkings equations[2]. Initially, these results were computed analytically by integrating the source terms using Green's integral but with Computational Fluid Dynamics (CFD), the calculation of the source terms and the acoustic parameters was made possible.

In general, the aeroacoustic applications have two approaches namely, pressure based and density based approaches. The pressure based approach is used ideally for low mach number simulations where the the fluctuations of density are minimal. So, in such conditions, using the assumption that wave propagation is isentropic, the fluctuations of pressure are used to form a wave equation that would in turn calculate the oscillations in the acoustic regime. The density based approach is most suited for higher mach number flow where the fluctuations in density are considerable.

In current version of OpenFOAM v1806, rhoPimpleAdiabaticFoam is available which is suitable for aeroacoustic applications at low Mach number flow simulations. So this report aims at adding a pressure based solver that couples the flow field and the wave equation.

1.2 Governing Equations

In Computational Aero Acoustics (CAA), two approaches are used.

1. Direct Methods : In this method, a transient solution is calculated from the source and the propagation of the sound waves. This method uses the most exact and straightforward methodology where the compressible Navier-Stokes equations are solved directly (DNS). There are large differences in the scales between the flow variables and acoustic variables. The meshes need to be very fine and extremely small time steps must be employed which proves to be extremely expensive.

2. Hybrid Methods : In this method, using a source field in CFD analysis, the propagation is

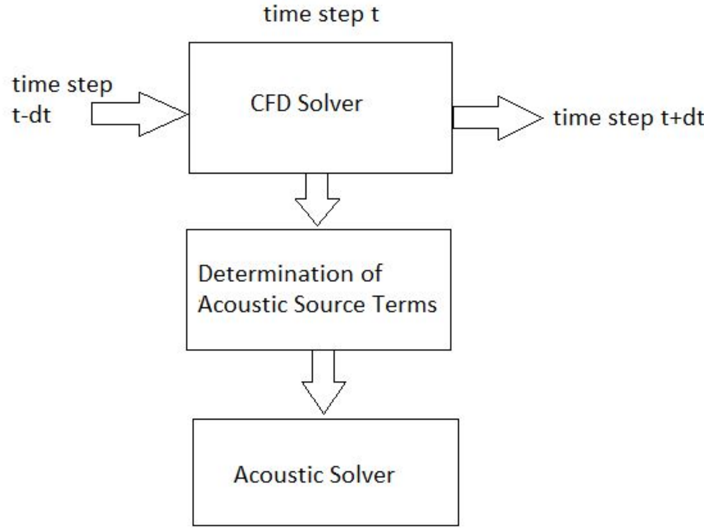

Figure 1.1: Solution Methodology

predicted using a transport method. This involves scale modeling and so the computational effort is significantly reduced and coarser meshes can be employed.

Hybrid Methods

Hybrid methods are applicable for problem with only one-way coupling between flow and acoustics. That is, the flow is independent of the acoustics. By doing this, the problem is divided into two sections, with one being the flow solution and other, the propagation of sound waves. The methodology is described in Figure 1.1.

LightHill's Acoustic Analogy

Lighthill developed analogies uncoupling the sound field from the source field. Using Navier Stokes' equations, he modeled source terms for the inhomogeneous acoustic wave equation.
Lighthill's wave equation is given by,

$$\frac{\partial^2 \rho}{\partial t^2} - a_\infty^2 \frac{\partial^2 \rho}{\partial x_i^2} = \frac{\partial^2 T_{ij}}{\partial x_{ij}} \tag{1.1}$$

where ρ is the density, a_∞ is the speed of sound and T_{ij} is the Lighthill's stress tensor and it is given by,

$$T_{ij} = \rho u_i u_j - \tau_{ij} + (p - a_\infty^2 \rho)\delta_{ij} \tag{1.2}$$

This equation is obtained by taking the time derivative of continuity equation and space derivative of momentum equation and subtracting them along with an additional term $a_\infty^2 \frac{\partial^2 \rho}{\partial x_i^2}$.
In equation 1.1, the left hand side of the equation is an ordinary wave equation and the right hand side is the acoustic source terms. To solve this equation, these source terms must be known and decoupled from the acoustic field.

Applications in Low Mach number applications

When the mach number is low, i.e in the range of 0.2 to 0.4, the flow is weakly compressible. In such cases, a transport equation for density fluctuations is not recommended as the density fluctuations

are almost negligible. So a transport equation for the acoustic pressure is created and this is coupled with the fluctuations in pressure from CFD simulations. The assumption is this method is that only the pressure perturbations are the cause of sound production. The acoustic pressure transport equation can be defined as,

$$\frac{\partial^2 p_a}{\partial t^2} - c_\infty^2 \frac{\partial^2 p_a}{\partial x^2} = -\frac{\partial^2 p'}{\partial t^2} \tag{1.3}$$

Where, p_a refers to the acoustic pressure and p' refers to the fluctuation in pressure from CFD simulations. This equation is obtained from Acoustic Perturbation Equations (APE)[3]. Eqn. 1.3, is implemented in OpenFOAM to determine the acoustic pressure fluctuations in a system which is then tested on the flow past wedge (prism).

Chapter 2

rhoPimpleAdiabaticFoam Solver

The acoustic solver is created as an extension of the in-built solver 'rhoPimpleAdiabaticFoam'. This solver finds its applications in low Mach number, weakly compressible flows that is suitable for aeroacoustic applications. The solver uses PIMPLE (PISO-SIMPLE) algorithm for time-resolved simulations. The solver uses Rhie-Chow interpolation which will be discussed later in section 2.1. The solver uses Reverse Cuthill McKee ordering scheme based on [4].
After initializing `OFv1806`, This solver is found in

```
cd $FOAM_APP/solvers/compressible/rhoPimpleAdiabaticFoam
```

The folder contains the files/folders,

- Make/ : contains 'files' and 'options' which are essential for compilation of the solver
- createFields.H : initializes all fields and objects used in the solver
- EEqn.H : Solves the energy equation
- pEqn.H : Solves the pressure equation
- UEqn.H : Solves the momentum equation
- resetBoundaries.H : Keeps standard formulation on domain boundaries to ensure compatibility with existing boundary conditions.
- rhoPimpleAdiabaticFoam.C : The main solver file that contains all the steps to be performed.

2.1 rhoPimpleAdiabaticFoam.C

A basic description of the solver is described below.

The solver file, 'rhoPimpleAdiabaticFoam.C', uses set of header files to initialize models for simulation and control as shown in Listing 2.1.

Listing 2.1: rhoPimpleAdiabaticFoam.C

```
1 #include "fvCFD.H"
2 #include "fluidThermo.H"
3 #include "turbulentFluidThermoModel.H"
4 #include "bound.H"
5 #include "pimpleControl.H"
6 #include "fvOptions.H"
7 #include "ddtScheme.H"
8 #include "fvcCorrectAlpha.H"
```

The header 'fvCFD.H' is used for include finite volume operations to be performed in the mesh. The header files that are specific to a compressible flow solver are 'fluidThermo.H, turbulentFluidThermoModel.H,' which include thermodynamic properties of the fluid that are bound to change at high speeds. With these models, the thermodynamic properties are determined. 'Bound.H' header is used to bound a scalar within the limits if it goes out. 'pimpleControl.H' is used to control and provide information on the convergence and operations of the pimple loop. 'ddtScheme.H' includes the time stepping schemes for transient simulations. 'fvcCorrectAlpha.H' is used to correct the velocity flux difference using an internal loop using correction factors.

Listing 2.2: rhoPimpleAdiabaticFoam.C

```
1   main()
2   {
3   #include "postProcess.H"
4   #include "addCheckCaseOptions.H"
5   #include "setRootCase.H"
6   #include "createTime.H"
7   #include "createMesh.H"
8   #include "createControl.H"
9   #include "createTimeControls.H"
10  #include "createFields.H"
11  #include "createFvOptions.H"
12  #include "initContinuityErrs.H"
```

From Listing 2.2, we can see that inside main(), the solver includes several classes. These classes are to add post processing functionalities, create time and mesh control, create fields (which will be discussed in createFields.H section), continuity errors and so on.

The runTime object is initialized in 'createTime.H' which is a member of class, Time. This object reads from the 'controlDict' file from the case directory and sets a start time, end time, Δt value and so on. As shown in Listing 2.3, a while loop is used to control the time stepping in the solver. Inside this loop, compressible courant number, Δt value are determined and time control is initialized. The current time value is set using 'runTime++'. In this step the current time value is incremented by the time step value. Line 9 displays the current time value in the log file.

Listing 2.3: rhoPimpleAdiabaticFoam.C

```
1   while (runTime.run())
2   {
3       #include "readTimeControls.H"
4       #include "compressibleCourantNo.H"
5       #include "setDeltaT.H"
6
7       runTime++;
8
9       Info<< "Time = " << runTime.timeName() << nl << endl;
10
11          if (pimple.nCorrPIMPLE() <= 1)
12          {
13              #include "rhoEqn.H"
14          }
15
16          // --- Pressure-velocity PIMPLE corrector loop
17          while (pimple.loop())
18          {
19              U.storePrevIter();
20              rho.storePrevIter();
21              phi.storePrevIter();
22              phiByRho.storePrevIter();
```

```
23
24              #include "UEqn.H"
25
26               // --- Pressure corrector loop
27              while (pimple.correct())
28              {
29                  #include "pEqn.H"
30               }
31
32              #include "EEqn.H"
33
34               if (pimple.turbCorr())
35              {
36                   turbulence->correct();
37               }
38          }
39
40          runTime.write();
41
42          runTime.printExecutionTime(Info);
43  }
44      Info<<"End \n"<<endl;
45
46      return 0;
47  }
```

If the number of correctors, in the PIMPLE control in fvSolution file of the case directory is less than or equal to 1, 'rhoEqn.H' is included in the code. This code is present in '$\$FOAM_SRC/$ finiteVolume/cfdTools/compressible/rhoEqn.H'. This file solves the continuity equation for density. If the number of pimple correcter is not equal to 1, this line is skipped. The code in the file, 'rhoEqn.H is given in Listing 2.4.

Listing 2.4: rhoEqn.H

```
1  {
2      solve(fvm::ddt(rho) + fvc::div(phi));
3  }
```

From Listing 2.3, after this step, the pressure-velocity coupling in this solver is done using a PIMPLE corrector loop. Using a while loop, 'pimple.loop()' starts the pimple loop when the number of corrector are 2 or higher. Inside the loop, velocity, density, flux and flux over density are stored for the previous iteration step of the loop as seen in lines 19-22 in the main solver file. Then the file 'UEqn.H' is included which solves for velocity using momentum equation. This file 'UEqn.H' is shown in Listing 2.5. Line 3 in the file is to set moving reference frame boundary conditions for the velocity field. The momentum equation solver takes into account the turbulence model specified in the 'turbulenceProperties' dictionary in the case/constant directory. The equation is relaxed implicitly and the appropriate constraints are set on the equation and solved for velocity. The code in the file, 'UEqn.H' is listed below.

Listing 2.5: UEqn.H

```
1  // Solve the Momentum equation
2
3   MRF.correctBoundaryVelocity(U);
4
5   tmp<fvVectorMatrix> tUEqn
6   (
7       fvm::ddt(rho, U) + fvm::div(phi, U)
8     + MRF.DDt(rho, U)
9     + turbulence->divDevRhoReff(U)
```

```
10    ==
11        fvOptions(rho, U)
12   );
13   fvVectorMatrix& UEqn = tUEqn.ref();
14
15   UEqn.relax();
16
17   fvOptions.constrain(UEqn);
18
19   if (pimple.momentumPredictor())
20   {
21       solve(UEqn == -fvc::grad(p));
22
23       fvOptions.correct(U);
24   }
```

After the momentum equation is solved, a pressure corrector loop is executed which solves for the pressure field as shown in Listing 2.3. As stated before, the pressure-velocity coupling is done using a PISO-SIMPLE algorithm. The velocity is computed by discretizing the velocity field from the momentum equation. This is an intermediate velocity or velocity predictor step. 'pEqn.H' is shown in Listing2.6. Using this velocity field, fields such as 'rAU' and 'HbyA' are defined. Using these, surface scalar fields 'rhorAUf' and 'rhoHbyAf' are interpolated based on the commented lines mentioned between lines 7-11. It is to be noted that the current algorithm does not include transonic options. When the working regime is non-transonic, the pressure equation is solved. Using Rhie-Chow interpolation the velocities are interpolated and using the pressure corrector equation, the pressure is solved. It is seen that the flux is updated for the pressure obtained from the current step. This forms the second part of Rhie-Chow interpolation. After the pressure field is solved, Pressure is relaxed and the velocity field is updated from the updated pressure field. Density is calculated using thermodynamic relations and finally, 'dpdt' is calculated.

Listing 2.6: pEqn.H

```
1  {
2        volScalarField rAU(1.0/UEqn.A());
3        volVectorField HbyA("HbyA", U);
4        HbyA = rAU*UEqn.H();
5
6
7        // Define coefficients and pseudo-velocities for RCM interpolation
8        // M[U] = AU - H = -grad(p)
9        // U = H/A - 1/A grad(p)
10       // H/A = U + 1/A grad(p)
11       surfaceScalarField rhorAUf
12       (
13           "rhorAUf",
14           fvc::interpolate(rho)/fvc::interpolate(UEqn.A())
15       );
16
17       surfaceVectorField rhoHbyAf
18       (
19           "rhoHbyAf",
20           fvc::interpolate(rho)*fvc::interpolate(U)
21         + rhorAUf*fvc::interpolate(fvc::grad(p))
22       );
23
24       #include "resetBoundaries.H"
25
26       if (pimple.nCorrPISO() <= 1)
27       {
28           tUEqn.clear();
29       }
30
```

```
    if (pimple.transonic())
    {
         FatalError
             << "\nTransonic option not available for " << args.executable()
             << exit(FatalError);
    }
    else
    {
        // Rhie & Chow interpolation (part 1)
        surfaceScalarField phiHbyA
        (
            "phiHbyA",
            (
                (rhoHbyAf & mesh.Sf())
              + rhorAUf*fvc::interpolate(rho)*fvc::ddtCorr(U, phiByRho)
              + fvc::interpolate(rho)
              * fvc::alphaCorr(U, phiByRho, pimple.finalInnerIter())
            )
        );

        MRF.makeRelative(fvc::interpolate(rho), phiHbyA);

        // Non-orthogonal pressure corrector loop
        while (pimple.correctNonOrthogonal())
        {
            // Pressure corrector
            fvScalarMatrix pEqn
            (
                fvm::ddt(psi, p)
              + fvc::div(phiHbyA)
              - fvm::laplacian(rhorAUf, p)
             ==
                fvOptions(psi, p, rho.name())
            );

            pEqn.solve(mesh.solver(p.select(pimple.finalInnerIter())));

            // Rhie & Chow interpolation (part 2)
            if (pimple.finalNonOrthogonalIter())
            {
                phi = phiHbyA + pEqn.flux();
            }
        }
    }

    phiByRho = phi/fvc::interpolate(rho);

    #include "rhoEqn.H"
    #include "compressibleContinuityErrs.H"

    // Explicitly relax pressure for momentum corrector
    p.relax();

    U = HbyA - rAU*fvc::grad(p);
    U.correctBoundaryConditions();
    fvOptions.correct(U);
}

rho = thermo.rho();

if (thermo.dpdt())
{
    dpdt = fvc::ddt(p);
}
```

After solving the pressure field, the solver as seen in Listing 2.3, includes the file 'EEqn.H' which

solves the temperature field in the domain. The file 'EEqn.H' is shown in Listing 2.7. This file primarily uses thermoDict to obtain case specific information. Initially, the values of 'c_p' and 'c_v' are obtained as volume scalar fields and γ is made sure to be constant throughout the domain. The solution uses isentropic relations and for this, γ must constant. Using the dictionary in the case directory, the stagnation values of pressure and temperature are taken into account. Using isentropic relations, the temperature field is determined. The compressibility factors, density are obtained in the subsequent steps.

Listing 2.7: EEqn.H

```
1  {
2      volScalarField& he = thermo.he();
3
4      const tmp<volScalarField>& tCp = thermo.Cp();
5      const tmp<volScalarField>& tCv = thermo.Cv();
6
7      const volScalarField& Cp = tCp();
8      const volScalarField& Cv = tCv();
9      const scalar gamma = max(Cp/Cv).value();
10
11     if (mag(gamma - min(Cp/Cv).value()) > VSMALL)
12     {
13         notImplemented("gamma not constant in space");
14     }
15
16     const dictionary& thermoDict = thermo.subDict("mixture");
17
18     const dictionary& eosDict = thermoDict.subDict("equationOfState");
19
20     bool local = eosDict.lookupOrDefault("local", false);
21
22     // Evolve T as:
23     //
24     // T_1 = T_0 \frac{p}{p_0}^{\frac{\gamma - 1}{\gamma}}
25
26     if (!local)
27     {
28         const scalar T0 = readScalar(eosDict.lookup("T0"));
29         const scalar p0 = readScalar(eosDict.lookup("p0"));
30
31         he = thermo.he(p, pow(p/p0, (gamma - scalar(1))/gamma)*T0);
32     }
33     else
34     {
35         const volScalarField& T0 = T.oldTime();
36         const volScalarField& p0 = p.oldTime();
37
38         he = thermo.he(p, pow(p/p0, (gamma - scalar(1))/gamma)*T0);
39     }
40
41     thermo.correct();
42
43     psi = 1.0/((Cp - Cv)*T);
44
45     rho = thermo.rho();
46     rho.relax();
47
48     rho.writeMinMax(Info);
49  }
```

The solver finally corrects turbulence and exits the PIMPLE loop as seen in Listing 2.3. The entire loop is repeated based on the number of correctors described in the dictionary. At the end of the time step, runTime.write() function prints the solver information such as residuals, number of iterations, solver, preconditioner or smoother used for solution of that variable and run-

Time.printExecutionTime(Info) prints the time taken for the execution of that time step.

2.2 createFields.H

The file, 'createFields.H' initializes all the fields and scalars that are necessary for the solver to run. For a variable to be used in the solver, it must be declared. Firstly, as seen in Listing 2.8, thermophysical properties are defined. These properties are defined as mesh dependant properties. Pressure and temperature are obtained from the function 'pThermo()' which reads the initial pressure and temperature and 'thermo.validate()' validates the thermodynamic state variables. This function is located in the class bassicThermo which is the parent class for fluidThermo.

Listing 2.8: createFields.H

```
1  Info<< "Reading thermophysical properties\n" << endl;
2
3  autoPtr<fluidThermo> pThermo
4  (
5      fluidThermo::New(mesh)
6  );
7  fluidThermo& thermo = pThermo();
8  thermo.validate(args.executable(), "h", "e");
9
10 volScalarField& p = thermo.p();
11 volScalarField& T = thermo.T();
```

In line 1 in Listing 2.9, density is defined as volScalarField. It is defined by means of an IOobject, and defined on meshes. It is read if it is present in the time directory or else, it is calculated. Density is written along with all the other variables in the time directory based on the settings in controlDict. Likewise, phi and phiByRho are defined as surfaceScalarField with appropriate object definition. The velocity is defined as volVectorField which must be read from the case directory. When other variable are not defined, the solver calculates their respective values but if velocity is not defined, the solver gives an error message.

Listing 2.9: createFields.H

```
1  volScalarField rho
2  (
3      IOobject
4      (
5          "rho",
6          runTime.timeName(),
7          mesh,
8          IOobject::READ_IF_PRESENT,
9          IOobject::AUTO_WRITE
10     ),
11     thermo.rho()
12 );
13
14 Info<< "Reading field U\n" << endl;
15 volVectorField U
16 (
17     IOobject
18     (
19         "U",
20         runTime.timeName(),
21         mesh,
22         IOobject::MUST_READ,
23         IOobject::AUTO_WRITE
24     ),
25     mesh
```

```
26 );
27
28 Info<< "Calculating face flux field phi\n" << endl;
29
30 surfaceScalarField phi
31 (
32     IOobject
33     (
34         "phi",
35         runTime.timeName(),
36         mesh,
37         IOobject::READ_IF_PRESENT,
38         IOobject::AUTO_WRITE
39     ),
40     linearInterpolate(rho)*linearInterpolate(U) & mesh.Sf()
41 );
42
43 Info<< "Calculating face flux field phiByRho\n" << endl;
44
45 surfaceScalarField phiByRho
46 (
47     IOobject
48     (
49         "phiByRho",
50         runTime.timeName(),
51         mesh,
52         IOobject::READ_IF_PRESENT,
53         IOobject::AUTO_WRITE
54     ),
55     phi/linearInterpolate(rho)
56 );
```

After this point, the turbulence model is defined as seen in Listing 2.10. A new compressible turbulence model object is created as a pointer based on the density, velocity, flux and thermodynamic properties (pressure and temperature) according to the entries defined in turbulenceProperties dictionary in the case directory.

Listing 2.10: createFields.H

```
1  Info<< "Creating turbulence model\n" << endl;
2  autoPtr<compressible::turbulenceModel> turbulence
3  (
4      compressible::turbulenceModel::New
5      (
6          rho,
7          U,
8          phi,
9          thermo
10     )
11 );
12 mesh.setFluxRequired(p.name());
```

The unsteady pressure term is defined and it is calculated in the file 'pEqn.H'. 'createMRF.H' is added if there are moving surfaces in the geometry. Finally, compressibility field 'psi' is declared as a volScalarField and it is calculated. This field is stored for 2 previous time steps. The final steps are given in Listing 2.11.

Listing 2.11: createFields.H

```
1  Info<< "Creating field dpdt\n" << endl;
2  volScalarField dpdt
```

```
3  (
4      IOobject
5      (
6          "dpdt",
7          runTime.timeName(),
8          mesh
9      ),
10     mesh,
11     dimensionedScalar(p.dimensions()/dimTime, Zero)
12 );
13
14 #include "createMRF.H"
15
16 Info<< "Creating compressibility field psi\n" << endl;
17 volScalarField psi("psi", 1.0/((thermo.Cp() - thermo.Cv())*T));
18 psi.oldTime() = 1.0/((thermo.Cp() - thermo.Cv())*T.oldTime());
19 psi.oldTime().oldTime() = 1.0/((thermo.Cp()-thermo.Cv())*T.oldTime().oldTime());
```

2.3 Make folder

This folder is responsible for compiling any codes in OpenFOAM. It contains two files namely, 'files' and 'options'. 'files' is responsible for targeting the file to be compiled and the location of the compiled file to be stored and 'options' is responsible for including appropriate links to enable compilation.
The code in 'files' is given in Listing 2.12.

Listing 2.12: Make/files

```
1 rhoPimpleAdiabaticFoam.C
2
3 EXE = $(FOAM_APPBIN)/rhoPimpleAdiabaticFoam
```

The code is 'options' is given in Listing 2.13,

Listing 2.13: Make/options

```
1  EXE_INC = \
2      -I$(LIB_SRC)/transportModels/compressible/lnInclude \
3      -I$(LIB_SRC)/thermophysicalModels/basic/lnInclude \
4      -I$(LIB_SRC)/TurbulenceModels/turbulenceModels/lnInclude \
5      -I$(LIB_SRC)/TurbulenceModels/compressible/lnInclude \
6      -I$(LIB_SRC)/finiteVolume/cfdTools \
7      -I$(LIB_SRC)/finiteVolume/lnInclude \
8      -I$(LIB_SRC)/meshTools/lnInclude \
9      -I$(LIB_SRC)/sampling/lnInclude \
10
11 EXE_LIBS = \
12     -lcompressibleTransportModels \
13     -lfluidThermophysicalModels \
14     -lspecie \
15     -lturbulenceModels \
16     -lcompressibleTurbulenceModels \
17     -lfiniteVolume \
18     -lmeshTools \
19     -lsampling \
20     -lfvOptions
```

Chapter 3

Implementation of Acoustic Solver

The acoustic solver is implemented according to equation 1.3. This equation couples fluctuating pressure from the flow field to solve for the acoustic pressure field. The operation follows the procedure as stated in Figure. 1.1. The flow field is completely solved and then pressure fluctuations are calculated followed by acoustic calculations.

In order for the solver to be implemented, it must be compiled in the user directory. The following lines are executed in the terminal window one at a time.

```
OFv1806
foam
cp -r --parents applications/solvers/compressible/rhoPimpleAdiabaticFoam $WM_PROJECT_USER_DIR
```

Then the working directory is changed to the corresponding folder.

```
ufoam
cd applications/solvers/compressible
```

Solver is renamed to something meaningful, like say 'rhoPimpleAdiabaticAcousticFoam'. The *.C file in the folder must have the same name as the folder and so it is renamed as well.

```
mv rhoPimpleAdiabaticFoam rhoPimpleAdiabaticAcousticFoam
cd rhoPimpleAdiabaticAcousticFoam
mv rhoPimpleAdiabaticFoam.C rhoPimpleAdiabaticAcousticFoam.C
sed -i s/'rhoPimpleAdiabaticFoam'/'rhoPimpleAdiabaticAcousticFoam'/g rhoPimpleAdiabaticAcousticFoam.C
```

The file, 'Make/files' is responsible for the compilation of the *.C file. So this file must be specified in the file 'files'. The compiled file must be stored in the user application bin folder.

```
sed -i s/'rhoPimpleAdiabaticFoam'/'rhoPimpleAdiabaticAcousticFoam'/g Make/files
sed -i s/'FOAM_APPBIN'/'FOAM_USER_APPBIN'/g Make/files
```

3.1 Creating Fields

For the acoustic solver, three fields, pAcoustic, pMean and pFluc are created. The first field is pAcoustic which is solved for the wave equation. pMean is the time averaged value of the pressure field which is used for the calculation of pFluc, the fluctuating value of the pressure field. pMean and pFluc are results obtained based on the CFD simulations. The code in Listing 3.1 is added to the createField.H file after the object 'dpdt' and before the line 'include createMRF.H'.

Listing 3.1: createFields.H

```
Info<< "Creating field pMean\n" << endl;
volScalarField pMean
(
    IOobject
    (
        "pMean",
        runTime.timeName(),
        mesh,
        IOobject::READ_IF_PRESENT,
        IOobject::AUTO_WRITE
    ),
    mesh,
    dimensionsScalar(p.dimensions())
);

Info<< "Creating field pFluc\n" << endl;
volScalarField pFluc
(
    IOobject
    (
        "pFluc",
        runTime.timeName(),
        mesh,
        IOobject::READ_IF_PRESENT,
        IOobject::AUTO_WRITE
    ),
    mesh,
    dimensionedScalar(p.dimensions())
);

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

From Listing 3.1, the pAcoustic file is a must read files that is read from the initial time directory for initial values and boundary conditions. pFluc and pMean are read if present but they are calculated in the solver. The fields are created like the others present in the original file, as an 'IOobject'. All the objects created are pressure fields and so the object type is 'volScalarField' and these fields are prescribed on the mesh.

Listing 3.2: createFields.H

```
Info<< "Creating field cInf\n" << endl;
volScalarField cInf
(
    IOobject
    (
        "cInf",
        runTime.timeName(),
        mesh
    ),
    mesh,
    dimensionedScalar(U.dimensions())
);
cInf = sqrt(thermo.Cp()/thermo.Cv()*(thermo.Cp()-thermo.Cv())*T);
scalar timeIndex = 1;
```

From Listing 3.2, a volume scalar field, speed of sound at freestream conditions is defined as 'cInf'. It is a calculated using thermodynamic scalar properties using the relation as in eqn. 3.1. The value of gamma is defined by eqn. 3.2 and the value of R is defined by eqn. 3.3. The values of c_p and c_v are available in the object thermo and their corresponding return functions are used to get their values. Finally, a scalar variable 'timeIndex' is created and initialized to 1. This scalar is used in the calculation of time averaged pressure field.

$$c_\infty = \sqrt{\gamma R T_\infty} \tag{3.1}$$

$$\gamma = \frac{c_p}{c_v} \tag{3.2}$$

$$R = c_p - c_v \tag{3.3}$$

Listing 3.3: createFields.H

```
IOdictionary acousticSettings
(
    IOobject
    (
        "acousticSettings",
        runTime.constant(),
        mesh,
        IOobject::MUST_READ_IF_MODIFIED,
        IOobject::NO_WRITE
    )
);

dimensionedScalar tAc
(
    "tAc",
    dimTime,
    acousticSettings.lookup("tAc")
);

dimensionedScalar nPass
(
    "nPass",
    dimless,
    acousticSettings.lookup("nPass")
);
```

In theory, the domain must be fully developed before we start time averaging the results and when the acoustic solver is solved. So an IODictionary, 'acousticSettings' is defined in which the user must define at what time the solver should start time averaging pressure field. This dictionary

is located in the constant folder of the case directory. 'tAc' is the variable that defines the time as to when the solver should start time averaging the pressure field. The variable that is defined is 'nPass' which is to specify the solver how many passes should be completed before the wave equation is solved. 'tAc' has the dimensions of time while 'nPass' is dimensionless.

3.2 acousticSolver.H

In order to simplify the implementation, a new header file containing the implementations of the solver is created and it is included in the main solver file at the right line. The header file is named 'acousticSolver.H' and it is created in the solver directory. The following line is added before runTime.write() in rhoPimpleAdiabaticAcousticFoam.C file,

```
#include "acousticSolver.H"
```

At this point in the solver, the CFD simulation is completely finished and the solver prints the information of the solvers and the solution details after which it starts the next time step. Now, the following lines are added in 'acousticSolver.H' as shown in Listing 3.3.

```
vi acousticSolver.H
```

Listing 3.4: acousticSolver.H

```
1  //acoustic solver
2  if(runTime.time()>tAc)
3  {
4   if(timeIndex == 1)
5   {
6    pMean = p;
7    pMean.storeOldTime();
8    timeIndex++;
9   }
10  else
11  {
12   Info<< "Calculating fields pMean and pFluc\n" << endl;
13   pMean = (pMean.oldTime()*(runTime.time()-runTime.deltaT())+p*runTime.deltaT())/(
        runTime.time());
14   pMean.storeOldTime();
15   if(runTime.time()>(tAc*nPass))
16   {
17    pFluc = p - pMean;
18   Info<< "Solving the wave equation for pAcoustic\n" << endl;
19   fvScalarMatrix pAcousticEqn
20   (
21     fvm::d2dt2(pAcoustic) - sqr(cInf)*fvm::laplacian(pAcoustic) + fvc::d2dt2(pFluc)
22    );
23
24   solve(pAcousticEqn);
25   }
26  }
27 }
```

At the beginning of the file, there is a conditional loop which allows the solver to proceed only when the runTime object's time value is greater that the user defined value under 'tAc'. For the first timestep after the solver enters the loop, the value of pMean is equal to p as it is the first step of averaging. Then the value is stored for the next time step and then timeIndex is incremented. In the later time steps, the else section gets executed. The time averaged value of pressure is calculated based on the formula,

$$pMean_{t=i} = \frac{(pMean_{t=i-1} \times (t - \Delta t) + p_{t=i} \times \Delta t)}{t} \tag{3.4}$$

This value is stored for each time step using 'pMean.storeOldTime()', so that it can accessed in the next time step using 'pMean.oldTime()'. The loop continues until the calculation of pMean till the second conditional statement is satisfied that is the time value must be greater than 'tAc*nPass'. This allows the solver to time average the pressure field for 'tAc*(nPass-1)' seconds. Once this condition is satisfied, the fluctuating value is calculated by taking the difference between the actual pressure value and mean pressure value. The wave equation is defined as an 'fvScalarMatrix' and it is named as 'pAcousticEqn'. This equation is defined as eqn.1.3. Here it is noted that the 'pFluc' term is a field and so 'fvc' namespace is used where as 'pAcoustic' terms are solved for and so 'fvm' namespace is used.

Finally, once the implementation of the solver is complete, the compilation[1] can be done by using,

```
wmake
```

[1] In some cases, the compiler might suggest a warning stating that timeIndex is unused. But it is to be noted that the variable is used in the calculation of pMean. It is a compiler error and can be neglected.

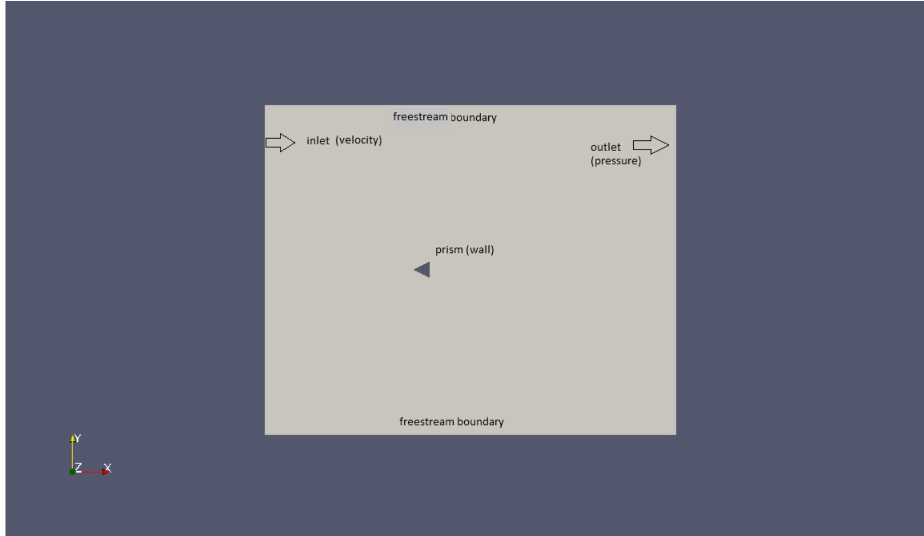

Figure 4.1: Test case domain : Prism

Chapter 4

Test Case

Once the solver is compiled, we can now use this solver on a test case to see if the implementation yields reasonable results. To do this, we are going to test this solver on a flow past a prism (wedge) at suitable conditions. Ideally, for aeroacoustic simulations, large domains and fine meshes are used. But for the simplicity and quick results, we opt a smaller domain and a relatively coarse mesh. The domain is shown in Figure. 4.1.

The prism is of height 4cm. The dimensions of the domain is 1m $\times$ 0.8m. The simulation carried out in this case is 2D. But some information on how to implement a 3D mesh is given.

The base tutorial case is in '$\$FOAM_TUTORIALS/compressible/sonicFoam/RAS/prism$'. The following lines are copied and executed line by line in the terminal window.

```
OFv1806
run
cp -r $FOAM_TUTORIALS/compressible/sonicFoam/RAS/prism .
cd prism
```

Now, we are inside the prism case folder.

4.1 system/ Folder

4.1.1 blockMeshDict

The original size of the domain is small. So first, the geometry of the case is modified. Open blockMeshDict in the system folder and modify as follow.

`vi system/blockMeshDict`

Replace the existing lines with the code given in Listing 4.1 and Listing 4.2.

Listing 4.1: system/blockMeshDict

```
1  scale    0.01;
2
3  vertices
4  (
5      (0 0 0)
6      (33 0 0)
7      (40 0 0)
8      (100 0 0)
9      (0 8 0)
10     (33 8 0)
11     (40 8 0)
12     (100 8 0)
13     (0 40 0)
14     (36 40 0)
15     (40 38 0)
16     (100 38 0)
17     (40 42 0)
18     (100 42 0)
19     (0 72 0)
20     (33 72 0)
21     (40 72 0)
22     (100 72 0)
23     (0 80 0)
24     (33 80 0)
25     (40 80 0)
26     (100 80 0)
27     (0 0 8)
28     (33 0 8)
29     (40 0 8)
30     (100 0 8)
31     (0 8 8)
32     (33 8 8)
33     (40 8 8)
34     (100 8 8)
35     (0 40 8)
36     (36 40 8)
37     (40 38 8)
38     (100 38 8)
39     (40 42 8)
40     (100 42 8)
41     (0 72 8)
42     (33 72 8)
43     (40 72 8)
44     (100 72 8)
45     (0 80 8)
46     (33 80 8)
47     (40 80 8)
48     (100 80 8)
49
50 );
```

The code in Listing 4.1 defines the position of the vertices. The values are scaled to 0.01. The positioning methods in both the geometries are the same. This code simply creates a bigger domain. The domain size is $1m \times 0.8m \times 0.08m$

Listing 4.2: system/blockMeshDict

```
blocks
(
    hex (0 1 5 4 22 23 27 26) (32 8 1) simpleGrading (0.2 1 1)
    hex (4 5 9 8 26 27 31 30) (32 32 1) simpleGrading (0.2 0.2 1)
    hex (5 6 10 9 27 28 32 31) (32 32 1) simpleGrading (1 0.2 1)
    hex (1 2 6 5 23 24 28 27) (32 8 1) simpleGrading (1 1 1)
    hex (2 3 7 6 24 25 29 28) (150 8 1) simpleGrading (7 1 1)
    hex (6 7 11 10 28 29 33 32) (150 32 1) simpleGrading (7 0.2 1)
    hex (10 11 13 12 32 33 35 34) (150 40 1) simpleGrading (7 1 1)
    hex (12 13 17 16 34 35 39 38) (150 32 1) simpleGrading (7 5 1)
    hex (16 17 21 20 38 39 43 42) (150 8 1) simpleGrading (7 1 1)
    hex (15 16 20 19 37 38 42 41) (32 8 1) simpleGrading (1 1 1)
    hex (9 12 16 15 31 34 38 37) (32 32 1) simpleGrading (1 5 1)
    hex (8 9 15 14 30 31 37 36) (32 32 1) simpleGrading (0.2 5 1)
    hex (14 15 19 18 36 37 41 40) (32 8 1) simpleGrading (0.2 1 1)
);
```

The code in Listing 4.2 is to generate blocks by connecting the vertices together. Hexahedral meshes are creates throughout the domain. 'simpleGrading' is used to provide a gradient to the mesh size for refinement. These two codes are copied to blockMeshDict in their respective sections.

4.1.2 controlDict

Once the the geometry is set, the solution control settings can be set in controlDict.
`vi system/controlDict`
In controlDict, various options such as startTime, endTime, deltaT, writeInterval and so on can be set as seen in Listing 4.3. For this simulation, the following settings were set.

Listing 4.3: system/controlDict

```
application       rhoPimpleAdiabaticAcousticFoam;

startFrom         latestTime;

startTime       0;

stopAt            endTime;

endTime           0.15;

deltaT            5e-06;

writeControl      runTime;

writeInterval     0.0001;

purgeWrite        0;

writeFormat       ascii;

writePrecision    6;

writeCompression  off;

timeFormat        general;

timePrecision     6;

runTimeModifiable true;
```

The main changes from the initial file are the endTime and the deltaT values. In this simulation, we will be looking at wake patterns and these patterns occur in a fully developed flow after 4 to 5 passes. For the flow to develop wake structures, the simulation must be run until 0.1 seconds. The deltaT value is set from 5e-07 to 5e-06. This is because, the initial case has a velocity of 300 m/s. But we are looking at velocities around mach 0.3 that is approximately less than 100 m/s. For this velocity and meshing, a deltaT of 5e-06 is reasonable and it satisfies the Courant criterion. The writeInterval can be kept the same for better visualization of the results. But, this occupies a lot of storage space so for a rough visualization, this value can be set to 0.005.

4.1.3 fvSchemes

From eqn. 1.3, we can see that two types of differential terms are used, $\frac{d^2}{dt^2}$ and $\frac{d^2}{dx^2}$. The second derivative of time and space. The discretization scheme for these terms should be specified in fvScheme file.

Listing 4.4: system/fvSchemes

```
1 d2dt2Schemes
2 {
3     default   Euler;
4 }
```

This code from Listing 4.4 is copied in the file fvSchemes after ddtSchemes section. This solves the second derivative with respect to time using Euler scheme. The second term, laplacian, is set to 'Gauss linear corrected' by default. So the solver takes this scheme for the laplacian term.

4.1.4 fvSolution

The solvers, preconditioners or smoothers are set for the solution of each variable using fvSolution. In this case, we need to set the solver type for the variable 'pAcoustic'. The code in Listing 4.5 is copied to the file at the end of the solver section.

Listing 4.5: system/fvSolution

```
1 pAcoustic
2     {
3  solver   PBiCGStab;
4  preconditioner DILU;
5  tolerance   1e-6;
6  relTol   0;
7     }
```

It is noted that, 'PBiCGStab' solver and 'DILU' preconditioner are used. These settings handle non-symmetric matrices well and so the solution is obtained in fewer iterations. Tolerance is set at 1e-6. The solution of pAcoustic is not present in any pressure velocity correction loops.

4.2 constant/ folder

4.2.1 thermophysicalProperties

This file is present in the constant directory. This contains the specifications of the gas for thermodynamic properties.

`vi constant/thermophysicalProperties`

This file contains settings about the thermodynamic relations that must be followed and specifications of the gas mixture such as weight, c_p, viscosity and Prandtl number.
The code in Listing 4.6 is pasted in the mixture section.

Listing 4.6: constant/thermophysicalProperties

```
equationOfState
  {
      p0          101325;
      T0          297.3;
  }
```

These values are calculated based on assumed boundary conditions. This definition is essential for rhoPimpleAdiabaticFoam solver as seen in Listing 2.7, lines 28 and 29.

4.2.2 turbulenceProperties

In this file, originally, kEpsilon model is used. Since, in this simulation, near wall effects are important, RNGkEpsilon model is used for better results.
`sed -i s/'kEpsilon'/'RNGkEpsilon'/g constant/turbulenceProperties`
The above line is executed in the terminal window to replace the turbulence model type.

4.2.3 acousticSettings

A dictionary must be set in the constant folder of the case directory to define additional variables which needed by the solver to run acoustic simulations. A new file is created by typing `vi constant/acousticSettings`. Two variables 'tAc' and 'nPass' are defined in the file as shown in Listing 4.7

Listing 4.7: constant/acousticSettings

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v1806                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      acousticSettings;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

tAc 0.0105;
nPass 2;

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

4.3 0/ Folder

This folder contains all the field variables and their initial values and boundary conditions are set. As stated above, this flow must be weakly compressible and so an arbitrary velocity value was chosen, say 95m/s. From the equation of state, using the stagnation value of temperature and pressure which are 297.3K and 1 atm, the temperature and pressure values were calculated using isentropic relations. In addition, pAcoustic field is created for the newly compiled solver to work.

4.3.1 'p' Field

The boundary conditions of this field remains the same as in the original tutorial file. But the value is modified to the value obtained from the compressible flow relations. The final file looks as given below. The inlet boundary condition is a 'fixedValue' boundary condition which is maintained constant throughout the simulation. 'waveTransmissive' boundary condition best suited for pressure waves which acts as a non-reflective boundary. 'zeroGradient' which is a nuemann boundary condition, used for all walls.

`vi 0/p`

Listing 4.8: 0/p

```
1  dimensions        [1 -1 -2 0 0 0 0];
2
3  internalField     uniform 96319.74;
4
5  boundaryField
6  {
7      inlet
8      {
9          type            fixedValue;
10         value           uniform 96319.74;
11     }
12
13     outlet
14     {
15         type            waveTransmissive;
16         field           p;
17         psi             thermo:psi;
18         gamma           1.4;
19         fieldInf        96319.74;
20         lInf            1;
21         value           uniform 96319.74;
22     }
23
24     bottomWall
25     {
26         type            zeroGradient;
27     }
28
29     topWall
30     {
31         type            zeroGradient;
32     }
33
34     prismWall
35     {
36         type            zeroGradient;
37     }
38
39     defaultFaces
40     {
41         type            empty;
42     }
43 }
```

4.3.2 'T' Field

Like the pressure field, the magnitude of the temperature at the initial and boundary values are modified. The code presented in Listing 4.9 is pasted in the this file. 'inletOutlet' boundary condition is a generic outflow condition which assumes the inflow value if return flow occurs.

`vi 0/T`

Listing 4.9: 0/T

```
1  dimensions        [0 0 0 1 0 0 0];
2
3  internalField     uniform 293;
4
5  boundaryField
6  {
7      inlet
8      {
9          type            fixedValue;
10         value           uniform 293;
11     }
12
13     outlet
14     {
15         type            inletOutlet;
16         inletValue      uniform 293;
17         value           uniform 293;
18     }
19
20     bottomWall
21     {
22         type            inletOutlet;
23         inletValue      uniform 293;
24         value           uniform 293;
25     }
26
27     topWall
28     {
29         type            inletOutlet;
30         inletValue      uniform 293;
31         value           uniform 293;
32     }
33
34     prismWall
35     {
36         type            zeroGradient;
37     }
38
39     defaultFaces
40     {
41         type            empty;
42     }
43 }
```

4.3.3 'U' Field

The velocity is set to 95m/s. The boundary conditions remain the same while the values of velocity, temperature, gamma and pressure are modified. 'freeStreamVelocity' boundary condition is used to set a free stream velocity on the boundary.
`vi 0/U`

Listing 4.10: 0/U

```
1  dimensions        [0 1 -1 0 0 0 0];
2
3  internalField     uniform (95 0 0);
4
5  boundaryField
6  {
7      inlet
8      {
```

```
9           type              fixedValue;
10          value             uniform (95 0 0);
11      }
12
13      outlet
14      {
15          type              inletOutlet;
16          inletValue        uniform (0 0 0);
17          value             uniform (0 0 0);
18      }
19
20      bottomWall
21      {
22          type              freestreamVelocity;
23          pInf              96319.74;
24          TInf              293;
25          UInf              (95 0 0);
26          gamma             1.4;
27          freestreamValue           uniform (95 0 0);
28      }
29
30      topWall
31      {
32          type              freestreamVelocity;
33          pInf              96319.74;
34          TInf              293;
35          UInf              (95 0 0);
36          gamma             1.4;
37          freestreamValue           uniform (95 0 0);
38      }
39
40      prismWall
41      {
42          type              noSlip;
43      }
44
45      defaultFaces
46      {
47          type              empty;
48      }
49  }
```

4.3.4 'pAcoustic' Field

This field is the result of the solution of the wave equation. The acoustic pressure field is a pressure wave and it is essential that the boundaries do not reflect them back in the domain. So all boundaries except the prism take 'waveTransmissive' boundary condition whereas prim boundary takes 'zeroGradient' boundary condition. The entire code as given in Listing 4.11 is pasted in a new file named 'pAcoustic'.

`vi 0/pAcoustic`

Listing 4.11: 0/pAcoustic

```
1   /*--------------------------------*- C++ -*----------------------------------*\
2   | =========                 |                                                 |
3   | \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
4   |  \\    /   O peration     | Version:  v1806                                 |
5   |   \\  /    A nd           | Web:      www.OpenFOAM.com                      |
6   |    \\/     M anipulation  |                                                 |
7   \*---------------------------------------------------------------------------*/
8   FoamFile
9   {
10      version     2.0;
```

```
11      format          ascii;
12      class           volScalarField;
13      object          pAcoustic;
14  }
15  // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
16
17  dimensions          [1 −1 −2 0 0 0 0];
18  internalField       uniform 0;
19
20  boundaryField
21  {
22      inlet
23      {
24          type                waveTransmissive;
25          field               pAcoustic;
26          psi                 thermo:psi;
27          gamma               1.4;
28          fieldInf            0;
29          lInf                1;
30          value               uniform 0;
31      }
32
33      outlet
34      {
35          type                waveTransmissive;
36          field               pAcoustic;
37          psi                 thermo:psi;
38          gamma               1.4;
39          fieldInf            0;
40          lInf                1;
41          value               uniform 0;
42      }
43
44      bottomWall
45      {
46          type                waveTransmissive;
47          field               pAcoustic;
48          psi                 thermo:psi;
49          gamma               1.4;
50          fieldInf            0;
51          lInf                1;
52          value               uniform 0;
53      }
54
55      topWall
56      {
57          type                waveTransmissive;
58          field               pAcoustic;
59          psi                 thermo:psi;
60          gamma               1.4;
61          fieldInf            0;
62          lInf                1;
63          value               uniform 0;
64      }
65
66      prismWall
67      {
68          type                zeroGradient;
69      }
70
71      defaultFaces
72      {
73          type                empty;
74      }
75  }
76
77  // ************************************************************************* //
```

With these variables modified, the case can be run by 'Allrun' script and the case the can be cleaned using 'Allclean' script.
For the creation of 'Allrun' script, a newfile is opened in the main case directory and the code in Listing 4.12 is pasted.

`vi Allrun`

Listing 4.12: Allrun

```
1 #!/bin/sh
2 cd ${0%/*} || exit 1                          # Run from this directory
3 . $WM_PROJECT_DIR/bin/tools/RunFunctions      # Tutorial run functions
4
5 runApplication blockMesh
6 runApplication $(getApplication)
7
8 #------------------------------------------------------------------------------
```

After pasting the code, the file is made executable by executing,

`chmod +x Allrun`

Similarily, 'Allclean' script is created.

`vi Allclean`

Listing 4.13: Allclean

```
1 #!/bin/sh
2 cd ${0%/*} || exit 1                          # Run from this directory
3 . $WM_PROJECT_DIR/bin/tools/CleanFunctions    # Tutorial clean functions
4
5 cleanCase
6
7 #------------------------------------------------------------------------------
```

`chmod +x Allclean`

The case can now be run by executing the Allrun script.

`./Allrun`

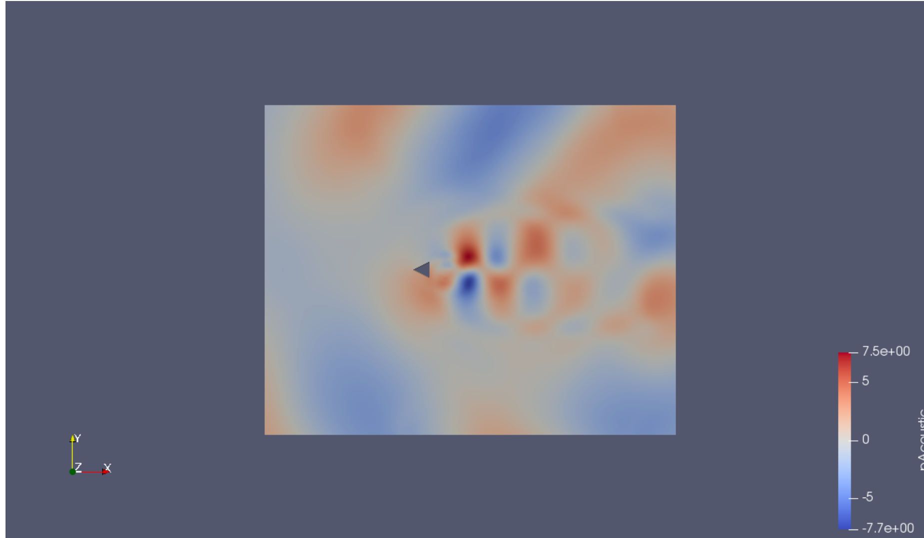

Figure 5.1: Acoustic pressure field at t=0.065s

Chapter 5

Results

The solution takes places in fairly small time steps and so it takes sometime for completion. On completion, the results can be visualized using typing 'paraFoam' in the terminal window and by loading the case. The field is set to 'pAcoustic' and simulation is started. Initially, the solution is incorrect as the flow needs to get developed. By theory at least one flow pass must be completed for the flow to be fully developed and about 4-5 flow passes for wake patterns to be generated. The solution can be skipped to 0.065 seconds by typing 649 in the frame box and we can note that the acoustic disturbances are captured due to these wake structures. Further forwarding to 0.15s (1499th frame), acoustic pressure due to wake patterns are clearly visible and they can be correlated to the instantaneous pressure values from which these acoustic pressure values are obtained from. From Figure 5.1 and Figure 5.2, it can be seen that the magnitudes of the acoustic pressure field develop over time.

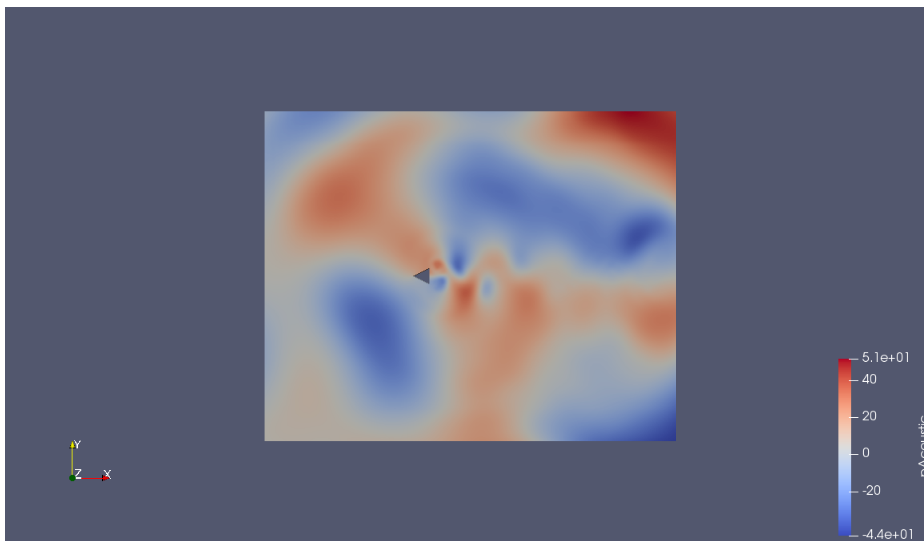

Figure 5.2: Acoustic pressure field at t=0.15s

Chapter 6

Conclusion

Thus, the report covers the contents stated in the learning outcomes: A basic theoretical background behind pressure-based wave equation, how the solver 'rhoPimpleAdiabaticFoam' has been implemented, how an aeroacoustic solver needs to be implemented as an extension to the already available 'rhoPimplAdiabaticFoam' solver and how to modify a case to make use of the newly implemented solver have been explained.

From the results in Figure 5.2, a maximum acoustic pressure of around 53 Pa is obtained which corresponds to 128dB roughly. This is calculated using the equation,

$$SPL = 20log\left(\frac{p_a}{p_{ref}}\right) \tag{6.1}$$

where SPL stands for Sound Pressure Level, p_a refers to the acoustic pressure and $p_{ref} = 2\times10^{-5}$ which refers to reference pressure.
This SPL value of 128dB is reasonable considering the geometry of the wedge and the velocity being high. It is to be noted that high sound levels occurs in the vortex region, downstream of the wedge.

Limitations and possible future work

1. The equation that is implemented is hyperbolic and so the mesh requirements are quite high. A coarse mesh has been employed in this simulation but for accurate results, a much finer mesh is required.

2. An RAS k-ϵ turbulence model is implemented and these models tend to dampen fluctuations. So it would be interesting to visualize results from LES and see the degree to which the solution is altered. But, since the equations are hyperbolic, the mesh requirement is a problem.

3. The domain is an important feature in aeroacoustic simulations. A much larger domain would be recommended to eliminate the effects of boundaries on the wedge for more accurate results.

4. The wave equation is only acceptable in weakly compressible and incompressible regimes. So for mach numbers above 0.4, the solver may yield unphysical results and it can not be used in transonic and supersonic regimes.

5. As stated above, only pressure flucatuations are considered as source terms. The effects of vorticity interferences and velocity fluctuations can be added to the solver.

Study Questions

1. What are the two methods used in CAA?
2. Explain how the hybrid method in CAA works.
3. State Lighthill's equation.
4. Why is a pressure-based wave equation used?
5. Explain the logic behind calculating the mean pressure field used in the implementation.
6. What is the purpose of the file resetBoundaries.H in the solver folder?
7. Where is the file rhoEqn.H located?

References

1. Hirschberg, Avraham, and Sjoerd W. Rienstra. "An introduction to aeroacoustics." Eindhoven university of technology (2004).
2. Uosukainen, Seppo. Foundations of acoustic analogies. VTT, 2011.
3. Siemens, P. L. M. "STAR-CCM+ Userâs Manual."
4. Knacke, T. (2013). Potential effects of Rhie Chow type interpolations in airframe noise simulations. In: Schram, C., DÃ©nos, R., Lecomte E. (ed): Accurate and efficient aeroacoustic prediction approaches for airframe noise, VKI LS 2013-03.