Cite as: Ramesh Babu, A. : Implementation of Aeroacoustic Solver for weakly compressible flows.
In Proceedings of CFD with OpenSource Software, 2018,
http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_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

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.

November 26, 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 an aeroacoustic solver for flow past a wedge.

The theory behind it

- The reader will learn the theoretical background behind pressure-based wave equations that has 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 . . . 15

3 Implementation of Acoustic Solver **17**
- 3.1 Creating Fields . . . 17
- 3.2 acousticSolver.H . . . 19

4 Test Case **21**
- 4.1 system/ Folder . . . 21
  - 4.1.1 blockMeshDict . . . 21
  - 4.1.2 controlDict . . . 23
  - 4.1.3 fvSchemes . . . 24
  - 4.1.4 fvSolution . . . 24
- 4.2 constant/ folder . . . 25
  - 4.2.1 thermophysicalProperties . . . 25
  - 4.2.2 turbulenceProperties . . . 26
- 4.3 0/ Folder . . . 26
  - 4.3.1 'p' Field . . . 26
  - 4.3.2 'T' Field . . . 27
  - 4.3.3 'U' Field . . . 28
  - 4.3.4 'pMean' Field . . . 29
  - 4.3.5 'pAcoustic' Field . . . 29

5 Results **32**

6 Conclusion **34**

7 Study Questions **35**

8 References **36**

9 Appendix **37**

Chapter 1

Theory

1.1 Introduction

Aeroacoustics is the study of flow induced sound. Some aeroacoustic noises include jet engine, wind turbines, wind musical instruments. 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 and vortex structures in the flow and flow interaction with walls and so on. There are several limitations in the computation of aeroacoustic field due to scaling difficulties and so it 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. This idea has led to further extensions such as, Curle's analogy, Ffowcs-Williams and Hawkings equations. 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 flows are most suited for higher mach number flow where the fluctuations in density are considerable.

In current version of OpenFOAM 'OFv1806', rhoPimpleAdiabaticFoam is available which is suitable for aeroacoustic applications at low Mach number flow simulations. So this report aims at adding an pressure based solver that couples the flow field and 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 but the computational requirement is very high where the meshes are very fine.

2. Hybrid Methods : In this method, using a source field in CFD analysis, the propagation is predicted using a transport method. This involves scale modeling and so the computational effort is significantly reduced and coarser meshes can be employed.

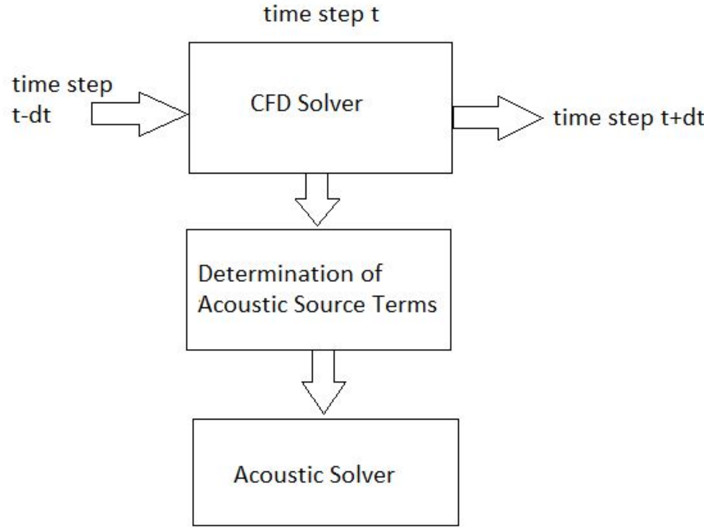

Figure 1.1: Solution Methodology

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 fundamental 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 T_{ij} is the Lighthill's stress tensor and 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 0.3 to 0.4, the flow is called 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 for a 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 flow past wedge (prism).

Chapter 2

rhoPimpleAdiabaticFoam Solver

The acoustic solver is created as an extension of the in-built solver 'rhoPimpleAdiabaticFoam'. This solver is finds its applications in low Mach number, weakly compressible flows that is suitable for aeroacoustic applications. The solver uses PIMPLE (PISO-SIMPLE) solutions for time-resolved simulations. The interpolation type is RCM. 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 overview of the working of the solver is described below.

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

Listing 2.1: rhoPimpleAdiabaticFoam.C

```
#include "fvCFD.H"
#include "fluidThermo.H"
#include "turbulentFluidThermoModel.H"
#include "bound.H"
#include "pimpleControl.H"
#include "fvOptions.H"
#include "ddtScheme.H"
#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

```
#include "postProcess.H"
#include "addCheckCaseOptions.H"
#include "setRootCase.H"
#include "createTime.H"
#include "createMesh.H"
#include "createControl.H"
#include "createTimeControls.H"
#include "createFields.H"
#include "createFvOptions.H"
#include "initContinuityErrs.H"
```

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. 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

```
while (runTime.run())
{
    #include "readTimeControls.H"
    #include "compressibleCourantNo.H"
    #include "setDeltaT.H"

    runTime++;

    Info<< "Time = " << runTime.timeName() << nl << endl;

         if (pimple.nCorrPIMPLE() <= 1)
        {
            #include "rhoEqn.H"
         }

       // --- Pressure-velocity PIMPLE corrector loop
         while (pimple.loop())
```

```
        {
            U.storePrevIter();
            rho.storePrevIter();
            phi.storePrevIter();
            phiByRho.storePrevIter();

            #include "UEqn.H"

             // --- Pressure corrector loop
            while (pimple.correct())
            {
                #include "pEqn.H"
             }

            #include "EEqn.H"

             if (pimple.turbCorr())
             {
                 turbulence->correct();
             }
         }

         runTime.write();

         runTime.printExecutionTime(Info);
 }
```

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 listed below.

Listing 2.4: rhoEqn.H

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

The pressure-velocity coupling in this solver is done using a PIMPLE corrector loop. Using a while loop, 'pimple.loop()' starts the pimple loop which 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. 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

```
// Solve the Momentum equation

 MRF.correctBoundaryVelocity(U);

 tmp<fvVectorMatrix> tUEqn
 (
      fvm::ddt(rho, U) + fvm::div(phi, U)
    + MRF.DDt(rho, U)
    + turbulence->divDevRhoReff(U)
  ==
      fvOptions(rho, U)
 );
 fvVectorMatrix& UEqn = tUEqn.ref();

 UEqn.relax();

 fvOptions.constrain(UEqn);

 if (pimple.momentumPredictor())
 {
      solve(UEqn == -fvc::grad(p));

      fvOptions.correct(U);
 }
```

After the momentum equation is solved, a pressure corrector loop is executed which solves for the pressure field. 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. 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

```
{
      volScalarField rAU(1.0/UEqn.A());
      volVectorField HbyA("HbyA", U);
      HbyA = rAU*UEqn.H();


      // Define coefficients and pseudo-velocities for RCM interpolation
      // M[U] = AU - H = -grad(p)
      // U = H/A - 1/A grad(p)
      // H/A = U + 1/A grad(p)
```

```
surfaceScalarField rhorAUf
(
    "rhorAUf",
    fvc::interpolate(rho)/fvc::interpolate(UEqn.A())
);

surfaceVectorField rhoHbyAf
(
    "rhoHbyAf",
    fvc::interpolate(rho)*fvc::interpolate(U)
  + rhorAUf*fvc::interpolate(fvc::grad(p))
);

#include "resetBoundaries.H"

if (pimple.nCorrPISO() <= 1)
{
    tUEqn.clear();
}

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 **??** includes the file 'EEqn.H' which solves the temperature field in the domain. 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 for this. 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

```
{
    volScalarField& he = thermo.he();

    const tmp<volScalarField>& tCp = thermo.Cp();
    const tmp<volScalarField>& tCv = thermo.Cv();

    const volScalarField& Cp = tCp();
    const volScalarField& Cv = tCv();
    const scalar gamma = max(Cp/Cv).value();

    if (mag(gamma - min(Cp/Cv).value()) > VSMALL)
```

```
    {
        notImplemented("gamma not constant in space");
    }

    const dictionary& thermoDict = thermo.subDict("mixture");

    const dictionary& eosDict = thermoDict.subDict("equationOfState");

    bool local = eosDict.lookupOrDefault("local", false);

    // Evolve T as:
    //
    // T_1 = T_0 \frac{p}{p_0}^{\frac{\gamma - 1}{\gamma}}

    if (!local)
    {
        const scalar T0 = readScalar(eosDict.lookup("T0"));
        const scalar p0 = readScalar(eosDict.lookup("p0"));

        he = thermo.he(p, pow(p/p0, (gamma - scalar(1))/gamma)*T0);
    }
    else
    {
        const volScalarField& T0 = T.oldTime();
        const volScalarField& p0 = p.oldTime();

        he = thermo.he(p, pow(p/p0, (gamma - scalar(1))/gamma)*T0);
    }

    thermo.correct();

    psi = 1.0/((Cp - Cv)*T);

    rho = thermo.rho();
    rho.relax();

    rho.writeMinMax(Info);
}
```

The solver finally corrects turbulence and exits the PIMPLE loop. 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 runTime.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, thermophysical properties are defined. These properties are defined as mesh dependant properties. Pressure and temperature are read from the function 'pThermo()' which reads the initial pressure and temperature and 'thermo.validate()' validates the thermodynamic state variables.

Listing 2.8: createFields.H

```
Info<< "Reading thermophysical properties\n" << endl;

autoPtr<fluidThermo> pThermo
(
    fluidThermo::New(mesh)
);
fluidThermo& thermo = pThermo();
thermo.validate(args.executable(), "h", "e");

volScalarField& p = thermo.p();
volScalarField& T = thermo.T();
```

Density is defined as volume scalar field. 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 after each time step. Likewise, phi and phiByRho are defined as surface scalar field with appropriate object definition. The velocity is defined as volume vector field 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

```
volScalarField rho
(
    IOobject
    (
        "rho",
        runTime.timeName(),
        mesh,
        IOobject::READ_IF_PRESENT,
        IOobject::AUTO_WRITE
    ),
    thermo.rho()
);

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

Info<< "Calculating face flux field phi\n" << endl;
```

```
surfaceScalarField phi
(
    IOobject
    (
        "phi",
        runTime.timeName(),
        mesh,
        IOobject::READ_IF_PRESENT,
        IOobject::AUTO_WRITE
    ),
    linearInterpolate(rho)*linearInterpolate(U) & mesh.Sf()
);

Info<< "Calculating face flux field phiByRho\n" << endl;

surfaceScalarField phiByRho
(
    IOobject
    (
        "phiByRho",
        runTime.timeName(),
        mesh,
        IOobject::READ_IF_PRESENT,
        IOobject::AUTO_WRITE
    ),
    phi/linearInterpolate(rho)
);
```

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

Listing 2.10: createFields.H

```
Info<< "Creating turbulence model\n" << endl;
autoPtr<compressible::turbulenceModel> turbulence
(
    compressible::turbulenceModel::New
    (
        rho,
        U,
        phi,
        thermo
    )
);
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 volume scalar field and calculated. This field is stored for 2 previous time steps.

Listing 2.11: createFields.H

```
Info<< "Creating field dpdt\n" << endl;
volScalarField dpdt
(
    IOobject
    (
        "dpdt",
        runTime.timeName(),
        mesh
    ),
    mesh,
    dimensionedScalar(p.dimensions()/dimTime, Zero)
);

#include "createMRF.H"

Info<< "Creating compressibility field psi\n" << endl;
volScalarField psi("psi", 1.0/((thermo.Cp() - thermo.Cv())*T));
psi.oldTime() = 1.0/((thermo.Cp() - thermo.Cv())*T.oldTime());
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 pasted below.

Listing 2.12: Make/files

```
rhoPimpleAdiabaticFoam.C

EXE = $(FOAM_APPBIN)/rhoPimpleAdiabaticFoam
```

The code is 'options' is pasted below,

Listing 2.13: Make/options

```
EXE_INC = \
    -I$(LIB_SRC)/transportModels/compressible/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/basic/lnInclude \
    -I$(LIB_SRC)/TurbulenceModels/turbulenceModels/lnInclude \
    -I$(LIB_SRC)/TurbulenceModels/compressible/lnInclude \
    -I$(LIB_SRC)/finiteVolume/cfdTools \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/sampling/lnInclude \

EXE_LIBS = \
```

```
    -lcompressibleTransportModels \
    -lfluidThermophysicalModels \
    -lspecie \
    -lturbulenceModels \
    -lcompressibleTurbulenceModels \
    -lfiniteVolume \
    -lmeshTools \
    -lsampling \
    -lfvOptions
```

Chapter 3

Implementation of Acoustic Solver

The acoustic solver is implemented as per eqn. 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 fig. 1.1. The flow field is completely solved and then pressure fluctuations are calculated followed by acoustic calculations. As stated below, the hybrid method is followed where there is a one-way coupling between flow field and acoustic field.

Inorder for the solver to be implemented, it must be compiled in the user directory. The following lines are executed 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 meaning, like say 'rhoPimpleAdiabaticAcousticFoam'. All the files in that folder are renamed correspondingly including the *.C file as shown below.

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

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

```
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 following code is added to the createField.H file after the object 'dpdt' and before the line 'include createMRF.H'.

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

The pMean and pAcoustic files are must read files that are read from the initial time directory for initial values and boundary conditions. pFluc is read if present but it is 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.

```
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;
```

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.

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

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

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

3.2 acousticSolver.H

Inorder to simplify the implementation, a new header file contains 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'.

```
vi acousticSolver.H
```

```
timeIndex++;
Info<< "Calculating fields pMean and pFluc\n" << endl;
pMean = (pMean.oldTime()*(timeIndex-1)+p)/timeIndex;
pMean.storeOldTime();
pFluc = p - pMean;
Info<< "Solving the wave equation for pAcoustic\n" << endl;
fvScalarMatrix pAcousticEqn
(
  fvm::d2dt2(pAcoustic) - sqr(cInf)*fvm::laplacian(pAcoustic) + fvc::d2dt2(pFluc)
  );

solve(pAcousticEqn);
```

At the beginning of the header file, timeIndex value is incremented every timestep. The time averaged value of pressure is calculated based on the formula,

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

This value is stored for each time step as old time step 'pMean.storeOldTime()', so that it can accessed in the next time step using 'pMean.oldTime()'. Once the pMean value is calculated, 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 per 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 solver is setup, 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.

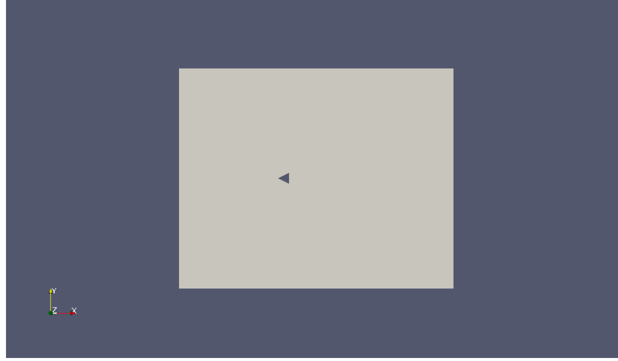

Figure 4.1: Test case domain : Prism

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 fig. 4.1.

The prism is of height 4cm. The dimensions of the domain is 1m × 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 available in '$\$FOAM_TUTORIALS/compressible/sonicFoam/RAS/prism$'. The following lines are copied and executed line by line.

```
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 below.

```
scale    0.01;

vertices
(
    (0 0 0)
    (33 0 0)
    (40 0 0)
    (100 0 0)
    (0 8 0)
    (33 8 0)
    (40 8 0)
    (100 8 0)
    (0 40 0)
    (36 40 0)
    (40 38 0)
    (100 38 0)
    (40 42 0)
    (100 42 0)
    (0 72 0)
    (33 72 0)
    (40 72 0)
    (100 72 0)
    (0 80 0)
    (33 80 0)
    (40 80 0)
    (100 80 0)
    (0 0 8)
    (33 0 8)
    (40 0 8)
    (100 0 8)
    (0 8 8)
    (33 8 8)
    (40 8 8)
    (100 8 8)
    (0 40 8)
    (36 40 8)
    (40 38 8)
    (100 38 8)
    (40 42 8)
    (100 42 8)
    (0 72 8)
    (33 72 8)
    (40 72 8)
    (100 72 8)
    (0 80 8)
    (33 80 8)
    (40 80 8)
    (100 80 8)

);
```

The first piece of code 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 $1m \times 0.8m \times 0.08m$

```
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 second piece of code 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. For this simulation, the following settings were set.

```
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 inital file are the endTime and deltaT values. In this simulation, we will be looking at wake patterns and wake patterns occur in a fully developed flow after 4 to 5 passes. for the flow to develop wake structures, the simulation must be run uptil 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. The criterion for this Courant number. For this velocity and meshing, a deltaT of 5e-06 is reasonable. The writeInterval is kept the same to visualize better 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.

```
d2dt2Schemes
{
    default              Euler;
}
```

This code 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 following code is copied to the file at the end of the solver section.

```
pAcoustic
   {
        solver            PBiCGStab;
        preconditioner    DILU;
        tolerance         1e-6;
        relTol            0;
   }
```

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 spcifications of the gas mixture such as weight, c_p, viscosity and Prandtl number.
The following code is pasted in the mixture section.

```
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.

4.3 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 which is 297.3K, the temperature and pressure values were calculated using isentropic relations. In addition, pAcoustic and pMean fields are 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' is a nuemann boundary condition.
`vi 0/p`

```
dimensions      [1 -1 -2 0 0 0 0];

internalField   uniform 96319.74;

boundaryField
{
    inlet
    {
        type            fixedValue;
        value           uniform 96319.74;
    }

    outlet
    {
        type            waveTransmissive;
        field           p;
        psi             thermo:psi;
        gamma           1.4;
        fieldInf        96319.74;
        lInf            1;
        value           uniform 96319.74;
    }

    bottomWall
    {
        type            zeroGradient;
    }

    topWall
```

```
    {
        type            zeroGradient;
    }

    prismWall
    {
        type            zeroGradient;
    }

    defaultFaces
    {
        type            empty;
    }
}
```

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 below 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`

```
dimensions      [0 0 0 1 0 0 0];

internalField   uniform 293;

boundaryField
{
    inlet
    {
        type            fixedValue;
        value           uniform 293;
    }

    outlet
    {
        type            inletOutlet;
        inletValue      uniform 293;
        value           uniform 293;
    }

    bottomWall
    {
        type            inletOutlet;
        inletValue      uniform 293;
        value           uniform 293;
    }

    topWall
    {
        type            inletOutlet;
        inletValue      uniform 293;
        value           uniform 293;
```

```
    }

    prismWall
    {
        type            zeroGradient;
    }

    defaultFaces
    {
        type            empty;
    }
}
```

4.3.3 'U' Field

The velocity is set at 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`

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

internalField   uniform (95 0 0);

boundaryField
{
    inlet
    {
        type            fixedValue;
        value           uniform (95 0 0);
    }

    outlet
    {
        type            inletOutlet;
        inletValue      uniform (0 0 0);
        value           uniform (0 0 0);
    }

    bottomWall
    {
        type            freestreamVelocity;
        pInf            96319.74;
        TInf            293;
        UInf            (95 0 0);
        gamma           1.4;
        freestreamValue           uniform (95 0 0);
    }

    topWall
    {
        type            freestreamVelocity;
```

```
        pInf                96319.74;
        TInf                293;
        UInf                (95 0 0);
        gamma               1.4;
        freestreamValue             uniform (95 0 0);
    }

    prismWall
    {
        type                noSlip;
    }

    defaultFaces
    {
        type                empty;
    }
}
```

4.3.4 'pMean' Field

The mean field of pressure at the inital time step is the same as the pressure field itself. So the field 'pMean' is created by just copying the pressure field into a new file.
`cp 0/p 0/pMean`

4.3.5 '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 is pasted in a new file named 'pAcoustic'.
`vi 0/pAcoustic`

```
/*--------------------------------*- 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       volScalarField;
    object      pAcoustic;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

```
dimensions        [1 -1 -2 0 0 0 0];
internalField     uniform 0;

boundaryField
{
    inlet
    {
        type            waveTransmissive;
        field           pAcoustic;
        psi             thermo:psi;
        gamma           1.4;
        fieldInf        0;
        lInf            1;
        value           uniform 0;
    }

    outlet
    {
        type            waveTransmissive;
        field           pAcoustic;
        psi             thermo:psi;
        gamma           1.4;
        fieldInf        0;
        lInf            1;
        value           uniform 0;
    }

    bottomWall
    {
        type            waveTransmissive;
        field           pAcoustic;
        psi             thermo:psi;
        gamma           1.4;
        fieldInf        0;
        lInf            1;
        value           uniform 0;
    }

    topWall
    {
        type            waveTransmissive;
        field           pAcoustic;
        psi             thermo:psi;
        gamma           1.4;
        fieldInf        0;
        lInf            1;
        value           uniform 0;
    }

    prismWall
    {
        type            zeroGradient;
```

```
    }

    defaultFaces
    {
        type            empty;
    }
}

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

With these variables modified, the case can be run creating '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 following piece of code is pasted. `vi Allrun`

```
#!/bin/sh
cd ${0%/*} || exit 1                        # Run from this directory
. $WM_PROJECT_DIR/bin/tools/RunFunctions    # Tutorial run functions

runApplication blockMesh
runApplication $(getApplication)

#------------------------------------------------------------------------------
```

After pasting the code, the file is made executable by executing,
`chmod +x Allrun`

Similarily, 'Allclean' script is created.
`vi Allclean`

```
#!/bin/sh
cd ${0%/*} || exit 1                        # Run from this directory
. $WM_PROJECT_DIR/bin/tools/CleanFunctions  # Tutorial clean functions

cleanCase

#------------------------------------------------------------------------------
```

`chmod +x Allclean`
The case can now be run by executing the Allrun script.
`./Allrun`

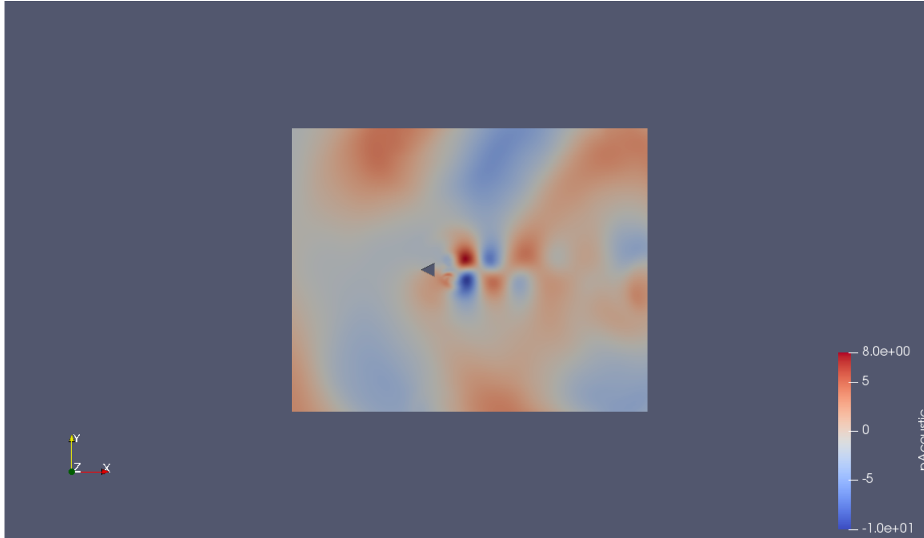

Figure 5.1: Acoustic pressure field at t=0.065s

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 atleast 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.

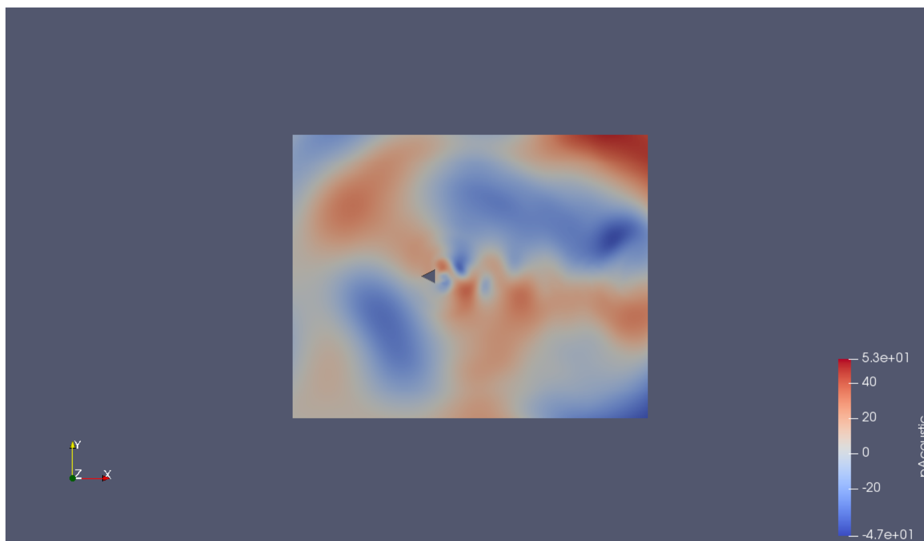

Figure 5.2: Acoustic pressure field at t=0.15s

Chapter 6

Conclusion

Thus, an acoustic solver was implemented as an extension to the already available 'rhoPimplAdiabaticFoam' solver. Though the results seem physically acceptable, they are not validated. 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. For future work, the effects of vorticity interferences and velocity fluctuations can be added to the solver.

Chapter 7

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. Explain the logic behind calculating the mean pressure field used in the implementation.
5. What is the purpose of the file resetBoundaries.H in the solver folder?
6. Where is the file rhoEqn.H located?

Chapter 8

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."

Chapter 9

Appendix

The appendix contains codes of the modified solver,'rhoPimpleAdiabaticAcousticFoam' and the test case,'prism'.

Solver Files

rhoPimpleAdiabaticAcousticFoam.C

```
#include "fvCFD.H"
#include "fluidThermo.H"
#include "turbulentFluidThermoModel.H"
#include "bound.H"
#include "pimpleControl.H"
#include "fvOptions.H"
#include "fvcAverage.H"
#include "ddtScheme.H"
#include "fvcCorrectAlpha.H"

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

int main(int argc, char *argv[])
{
    #include "postProcess.H"

    #include "addCheckCaseOptions.H"
    #include "setRootCase.H"
    #include "createTime.H"
    #include "createMesh.H"
    #include "createControl.H"
    #include "createTimeControls.H"

    #include "createFields.H"
    #include "createFvOptions.H"
    #include "initContinuityErrs.H"

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

    Info<< "\nStarting time loop\n" << endl;
```

```
        while (runTime.run())
    {
        #include "readTimeControls.H"
        #include "compressibleCourantNo.H"
        #include "setDeltaT.H"

        runTime++;

        Info<< "Time = " << runTime.timeName() << nl << endl;

        if (pimple.nCorrPIMPLE() <= 1)
        {
            #include "rhoEqn.H"
        }

        // --- Pressure-velocity PIMPLE corrector loop
        while (pimple.loop())
        {
            U.storePrevIter();
            rho.storePrevIter();
            phi.storePrevIter();
            phiByRho.storePrevIter();

            #include "UEqn.H"

            // --- Pressure corrector loop
            while (pimple.correct())
            {
                #include "pEqn.H"
            }

            #include "EEqn.H"

            if (pimple.turbCorr())
            {
                turbulence->correct();
            }
        }

        #include "acousticSolver.H"
        runTime.write();

        runTime.printExecutionTime(Info);
    }

    Info<< "End\n" << endl;

    return 0;
}
```

UEqn.H

```
// Solve the Momentum equation

MRF.correctBoundaryVelocity(U);

tmp<fvVectorMatrix> tUEqn
(
    fvm::ddt(rho, U) + fvm::div(phi, U)
  + MRF.DDt(rho, U)
  + turbulence->divDevRhoReff(U)
 ==
    fvOptions(rho, U)
);
fvVectorMatrix& UEqn = tUEqn.ref();

UEqn.relax();

fvOptions.constrain(UEqn);

if (pimple.momentumPredictor())
{
    solve(UEqn == -fvc::grad(p));

    fvOptions.correct(U);
}
```

pEqn.H

```
{
    volScalarField rAU(1.0/UEqn.A());
    volVectorField HbyA("HbyA", U);
    HbyA = rAU*UEqn.H();


    // Define coefficients and pseudo-velocities for RCM interpolation
    // M[U] = AU - H = -grad(p)
    // U = H/A - 1/A grad(p)
    // H/A = U + 1/A grad(p)
    surfaceScalarField rhorAUf
    (
        "rhorAUf",
        fvc::interpolate(rho)/fvc::interpolate(UEqn.A())
    );

    surfaceVectorField rhoHbyAf
    (
        "rhoHbyAf",
        fvc::interpolate(rho)*fvc::interpolate(U)
      + rhorAUf*fvc::interpolate(fvc::grad(p))
    );
```

```
#include "resetBoundaries.H"

if (pimple.nCorrPISO() <= 1)
{
    tUEqn.clear();
}

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);
}
```

EEqn.H

```
{
    volScalarField& he = thermo.he();

    const tmp<volScalarField>& tCp = thermo.Cp();
    const tmp<volScalarField>& tCv = thermo.Cv();

    const volScalarField& Cp = tCp();
    const volScalarField& Cv = tCv();
    const scalar gamma = max(Cp/Cv).value();

    if (mag(gamma - min(Cp/Cv).value()) > VSMALL)
    {
        notImplemented("gamma not constant in space");
    }

    const dictionary& thermoDict = thermo.subDict("mixture");

    const dictionary& eosDict = thermoDict.subDict("equationOfState");

    bool local = eosDict.lookupOrDefault("local", false);

    // Evolve T as:
    //
    // T_1 = T_0 \frac{p}{p_0}^{\frac{\gamma - 1}{\gamma}}

    if (!local)
    {
        const scalar T0 = readScalar(eosDict.lookup("T0"));
        const scalar p0 = readScalar(eosDict.lookup("p0"));

        he = thermo.he(p, pow(p/p0, (gamma - scalar(1))/gamma)*T0);
```

```
    }
    else
    {
        const volScalarField& T0 = T.oldTime();
        const volScalarField& p0 = p.oldTime();

        he = thermo.he(p, pow(p/p0, (gamma - scalar(1))/gamma)*T0);
    }

    thermo.correct();

    psi = 1.0/((Cp - Cv)*T);

    rho = thermo.rho();
    rho.relax();

    rho.writeMinMax(Info);
}
```

resetBoundaries.H

```
{
    // Keep standard formulation on domain boundaries to ensure compatibility
    // with existing boundary conditions
    const Foam::FieldField<Foam::fvsPatchField, scalar> rhorAUf_orig
    (
        fvc::interpolate(rho.boundaryField()*rAU.boundaryField())
    );

    const Foam::FieldField<Foam::fvsPatchField, vector> rhoHbyA_orig
    (
        fvc::interpolate(rho.boundaryField()*HbyA.boundaryField())
    );

    surfaceScalarField::Boundary& rhorAUfbf = rhorAUf.boundaryFieldRef();
    surfaceVectorField::Boundary& rhoHbyAfbf = rhoHbyAf.boundaryFieldRef();

    forAll(U.boundaryField(), patchi)
    {
        if (!U.boundaryField()[patchi].coupled())
        {
            rhorAUfbf[patchi] = rhorAUf_orig[patchi];
            rhoHbyAfbf[patchi] = rhoHbyA_orig[patchi];
        }
    }
}
```

acousticSolver.H

```
timeIndex++;
Info<< "Calculating fields pMean and pFluc\n" << endl;
```

```
pMean = (pMean.oldTime()*(timeIndex-1)+p)/timeIndex;
pMean.storeOldTime();
pFluc = p - pMean;
Info<< "Solving the wave equation for pAcoustic\n" << endl;
fvScalarMatrix pAcousticEqn
(
  fvm::d2dt2(pAcoustic) - sqr(cInf)*fvm::laplacian(pAcoustic) + fvc::d2dt2(pFluc)
  );

solve(pAcousticEqn);
```

Make/files

```
rhoPimpleAdiabaticAcousticFoam.C

EXE = $(FOAM_USER_APPBIN)/rhoPimpleAdiabaticAcousticFoam
```

Make/options

```
EXE_INC = \
    -I$(LIB_SRC)/transportModels/compressible/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/basic/lnInclude \
    -I$(LIB_SRC)/TurbulenceModels/turbulenceModels/lnInclude \
    -I$(LIB_SRC)/TurbulenceModels/compressible/lnInclude \
    -I$(LIB_SRC)/finiteVolume/cfdTools \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/sampling/lnInclude \

EXE_LIBS = \
    -lcompressibleTransportModels \
    -lfluidThermophysicalModels \
    -lspecie \
    -lturbulenceModels \
    -lcompressibleTurbulenceModels \
    -lfiniteVolume \
    -lmeshTools \
    -lsampling \
    -lfvOptions
```

Test Case files

system/blockMeshDict

```
scale    0.01;

vertices
(
    (0 0 0)
```

```
    (33 0 0)
    (40 0 0)
    (100 0 0)
    (0 8 0)
    (33 8 0)
    (40 8 0)
    (100 8 0)
    (0 40 0)
    (36 40 0)
    (40 38 0)
    (100 38 0)
    (40 42 0)
    (100 42 0)
    (0 72 0)
    (33 72 0)
    (40 72 0)
    (100 72 0)
    (0 80 0)
    (33 80 0)
    (40 80 0)
    (100 80 0)
    (0 0 8)
    (33 0 8)
    (40 0 8)
    (100 0 8)
    (0 8 8)
    (33 8 8)
    (40 8 8)
    (100 8 8)
    (0 40 8)
    (36 40 8)
    (40 38 8)
    (100 38 8)
    (40 42 8)
    (100 42 8)
    (0 72 8)
    (33 72 8)
    (40 72 8)
    (100 72 8)
    (0 80 8)
    (33 80 8)
    (40 80 8)
    (100 80 8)

);

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)
);

edges
(
);

boundary
(
    inlet
    {
        type patch;
        faces
        (
            (0 22 26 4)
            (4 26 30 8)
            (8 30 36 14)
            (14 36 40 18)
        );
    }
    outlet
    {
        type patch;
        faces
        (
            (3 7 29 25)
            (7 11 33 29)
            (11 13 35 33)
            (13 17 39 35)
            (17 21 43 39)
        );
    }
    bottomWall
    {
        type patch;
        faces
        (
            (0 1 23 22)
            (1 2 24 23)
            (2 3 25 24)
        );
    }
    topWall
    {
        type patch;
        faces
```

```
        (
            (18 40 41 19)
            (19 41 42 20)
            (20 42 43 21)
        );
    }
    prismWall
    {
        type wall;
        faces
        (
            (12 10 32 34)
            (12 34 31 9)
            (9 31 32 10)
        );
    }
);

mergePatchPairs
(
);
```

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;
```

system/fvSolution

```
solvers
{
    "rho.*"
    {
        solver          diagonal;
    }

    "p.*"
    {
        solver          smoothSolver;
        smoother        symGaussSeidel;
        tolerance       1e-06;
        relTol          0;
    }

    "(U|e|R).*"
    {
        $p;
        tolerance       1e-05;
    }

    "(k|epsilon).*"
    {
        $p;
        tolerance       1e-08;
    }
    pAcoustic
   {
        solver          PBiCGStab;
        preconditioner  DILU;
        tolerance       1e-6;
        relTol          0;
        maxIter         5000;
   }
}

PIMPLE
{
    nOuterCorrectors 2;
    nCorrectors       1;
    nNonOrthogonalCorrectors 0;
}
```

system/fvSchemes

```
ddtSchemes
{
    default          Euler;
}

d2dt2Schemes
{
    default              Euler;
}
gradSchemes
{
    default          Gauss linear;
}

divSchemes
{
    default          none;
    div(phi,U)       Gauss limitedLinearV 1;
    div(phi,e)       Gauss limitedLinear 1;
    div(phid,p)      Gauss limitedLinear 1;
    div(phi,K)       Gauss limitedLinear 1;
    div(phiv,p)      Gauss limitedLinear 1;
    div(phi,k)       Gauss upwind;
    div(phi,epsilon) Gauss upwind;
    div(((rho*nuEff)*dev2(T(grad(U))))) Gauss linear;
}

laplacianSchemes
{
    default          Gauss linear corrected;
}

interpolationSchemes
{
    default          linear;
}

snGradSchemes
{
    default          corrected;
}
```

constant/turbulenceProperties

```
simulationType  RAS;

RAS
{
```

```
    RASModel            RNGkEpsilon;

    turbulence          on;

    printCoeffs         on;
}
```

constant/thermophysicalProperties

```
thermoType
{
    type                hePsiThermo;
    mixture             pureMixture;
    transport           const;
    thermo              hConst;
    equationOfState     perfectGas;
    specie              specie;
    energy              sensibleInternalEnergy;
}

mixture
{
    equationOfState
    {
      p0              101325;
      T0              297.3;
    }
    specie
    {
        molWeight           28.9;
    }
    thermodynamics
    {
        Cp                  1005;
        Hf                  0;
    }
    transport
    {
        mu                  1.8e-05;
        Pr                  0.7;
    }
}
```

0/p

```
dimensions          [1 -1 -2 0 0 0 0];

internalField       uniform 96319.74;
```

```
boundaryField
{
    inlet
    {
        type                fixedValue;
        value               uniform 96319.74;
    }

    outlet
    {
        type                waveTransmissive;
        field               p;
        psi                 thermo:psi;
        gamma               1.4;
        fieldInf            96319.74;
        lInf                1;
        value               uniform 96319.74;
    }

    bottomWall
    {
        type                zeroGradient;
    }

    topWall
    {
        type                zeroGradient;
    }

    prismWall
    {
        type                zeroGradient;
    }

    defaultFaces
    {
        type                empty;
    }
}
```

0/U

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

internalField     uniform (95 0 0);

boundaryField
{
    inlet
    {
        type                fixedValue;
```

```
        value               uniform (95 0 0);
    }

    outlet
    {
        type                inletOutlet;
        inletValue          uniform (0 0 0);
        value               uniform (0 0 0);
    }

    bottomWall
    {
        type                freestreamVelocity;
        pInf                96319.74;
        TInf                293;
        UInf                (95 0 0);
        gamma               1.4;
        freestreamValue             uniform (95 0 0);
    }

    topWall
    {
        type                freestreamVelocity;
        pInf                96319.74;
        TInf                293;
        UInf                (95 0 0);
        gamma               1.4;
        freestreamValue             uniform (95 0 0);
    }

    prismWall
    {
        type                noSlip;
    }

    defaultFaces
    {
        type                empty;
    }
}
```

0/T

```
dimensions          [0 0 0 1 0 0 0];

internalField       uniform 293;

boundaryField
{
    inlet
    {
```

```
        type                fixedValue;
        value               uniform 293;
    }

    outlet
    {
        type                inletOutlet;
        inletValue          uniform 293;
        value               uniform 293;
    }

    bottomWall
    {
        type                inletOutlet;
        inletValue          uniform 293;
        value               uniform 293;
    }

    topWall
    {
        type                inletOutlet;
        inletValue          uniform 293;
        value               uniform 293;
    }

    prismWall
    {
        type                zeroGradient;
    }

    defaultFaces
    {
        type                empty;
    }
}
```

0/pMean

```
dimensions          [1 -1 -2 0 0 0 0];

internalField       uniform 96319.74;

boundaryField
{
    inlet
    {
        type                fixedValue;
        value               uniform 96319.74;
    }

    outlet
```

```
    {
        type            waveTransmissive;
        field           p;
        psi             thermo:psi;
        gamma           1.3;
        fieldInf        96319.74;
        lInf            1;
        value           uniform 96319.74;
    }

    bottomWall
    {
        type            zeroGradient;
    }

    topWall
    {
        type            zeroGradient;
    }

    prismWall
    {
        type            zeroGradient;
    }

    defaultFaces
    {
        type            empty;
    }
}
```

0/pAcoustic

```
dimensions      [1 -1 -2 0 0 0 0];
internalField   uniform 0;

boundaryField
{
    inlet
    {
        type            waveTransmissive;
        field           pAcoustic;
        psi             thermo:psi;
        gamma           1.4;
        fieldInf        0;
        lInf            1;
        value           uniform 0;
    }

    outlet
    {
        type            waveTransmissive;
```

```
        field           pAcoustic;
        psi             thermo:psi;
        gamma           1.4;
        fieldInf        0;
        lInf            1;
        value           uniform 0;
    }

    bottomWall
    {
        type            waveTransmissive;
        field           pAcoustic;
        psi             thermo:psi;
        gamma           1.4;
        fieldInf        0;
        lInf            1;
        value           uniform 0;
    }

    topWall
    {
        type            waveTransmissive;
        field           pAcoustic;
        psi             thermo:psi;
        gamma           1.4;
        fieldInf        0;
        lInf            1;
        value           uniform 0;
    }

    prismWall
    {
        type            zeroGradient;
    }

    defaultFaces
    {
        type            empty;
    }
}
```

0/k

```
dimensions      [0 2 -2 0 0 0 0];

internalField   uniform 1000;

boundaryField
{
    inlet
    {
```

```
        type            fixedValue;
        value           uniform 1000;
    }
    outlet
    {
        type            inletOutlet;
        inletValue      uniform 1000;
        value           uniform 1000;
    }
    bottomWall
    {
        type            inletOutlet;
        inletValue      uniform 1000;
        value           uniform 1000;
    }
    topWall
    {
        type            inletOutlet;
        inletValue      uniform 1000;
        value           uniform 1000;
    }
    prismWall
    {
        type            kqRWallFunction;
        value           uniform 1000;
    }
    defaultFaces
    {
        type            empty;
    }
}
```

0/epsilon

```
dimensions      [0 2 -3 0 0 0 0];

internalField   uniform 266000;

boundaryField
{
    inlet
    {
        type            fixedValue;
        value           uniform 266000;
    }
    outlet
    {
        type            inletOutlet;
        inletValue      uniform 266000;
        value           uniform 266000;
    }
```

```
    bottomWall
    {
        type            inletOutlet;
        inletValue      uniform 266000;
        value           uniform 266000;
    }
    topWall
    {
        type            inletOutlet;
        inletValue      uniform 266000;
        value           uniform 266000;
    }
    prismWall
    {
        type            epsilonWallFunction;
        value           uniform 266000;
    }
    defaultFaces
    {
        type            empty;
    }
}
```

0/alphat

```

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

internalField   uniform 0;

boundaryField
{
    inlet
    {
        type            calculated;
        value           uniform 0;
    }
    outlet
    {
        type            calculated;
        value           uniform 0;
    }
    bottomWall
    {
        type            calculated;
        value           uniform 0;
    }
    topWall
    {
        type            calculated;
        value           uniform 0;
    }
```

```
    prismWall
    {
        type            compressible::alphatWallFunction;
        value           uniform 0;
    }
    defaultFaces
    {
        type            empty;
    }
}
```

0/nut

```
dimensions      [0 2 -1 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    inlet
    {
        type            calculated;
        value           uniform 0;
    }
    outlet
    {
        type            calculated;
        value           uniform 0;
    }
    bottomWall
    {
        type            calculated;
        value           uniform 0;
    }
    topWall
    {
        type            calculated;
        value           uniform 0;
    }
    prismWall
    {
        type            nutkWallFunction;
        value           uniform 0;
    }
    defaultFaces
    {
        type            empty;
    }
}
```