Cite as: Ghosh, D.: Implementation of the FWH aero-acoustic analogy for sector analysis of an axi-symmetric turbomachine. In Proceedings of CFD with OpenSource Software, 2021, Edited by Nilsson. H., http://dx.doi.org/10.17196/OS_CFD#YEAR_2021

CFD WITH OPENSOURCE SOFTWARE

A COURSE AT CHALMERS UNIVERSITY OF TECHNOLOGY
TAUGHT BY HÅKAN NILSSON

Implementation of the FWH aero-acoustic analogy for sector analysis of an axi-symmetric turbomachine

---

Developed for OpenFOAM-v2006

Author:
DEBARSHEE GHOSH
Chalmers University of Technology
ghoshd@chalmers.com

Peer reviewed by:
Dr. Niklas ANDERSSON
Dr. Saeed SALEHI
Pengxu ZOU

February 9, 2022

Learning outcomes

This tutorial aims to address the following four questions: How to use it, The theory of it, How it is implemented, and How to modify it.

The reader will learn:

How to use it:

- A tutorial illustrating the implementation of the aero-acoustic library for a single sector of an axi-symmetric turbomachine and adapting the results to obtain the aero-acoustic pressure waves for the full annulus.

The theory of it:

- The theory behind Ffowcs-Williams and Hawkings (FWH) analogy.
- The theory behind Farassat 1A (F1A) Formulation.

How it is implemented:

- An external aero-acoustic library will be used as the starting point which already implements the Farassat 1A formulation to compute the FWH analogy.
- SRFPimpleFoam solver will be used along with cyclic boundary conditions to simulate a single sector of the axi-symmetric turbomachine.

How to modify it:

- Copying the single sector FWH surface results and adapting the cell centre vectors and surface area vectors to obtain the results for the remaining sectors.

Prerequisites

The reader is expected to know the following in order to get maximum benefit out of this report:

- How to compile a library in OpenFOAM
- Fundamentals of Computational Methods for Fluid Dynamics
- How to customise a solver and do top-level application programming in OpenFOAM

Contents

Nomenclature

Acronyms

BPF	Blade Passing Frequency
CAA	Computational Aero Acoustics
DNS	Direct Numerical Simulation
F1A	Farassat 1A Formulation
FWH	Ffowcs Williams and Hawkings
LES	Large Eddy Simulation
SPL	Sound Power Level
UML	Unified Modelling Language

English symbols

$\Box^2$	Wave operator	
δ_{ij}	Kronecker Delta	
ρ	Density of fluid	kg/m^3
τ	Source time	s
$\vec{n}$	Unit outward normal vector to source surface, with components n_i	m
$\vec{r}$	Distance between observer and source with components r_i	m
$\vec{u}$	Local fluid velocity with components u_i	m/s
$\vec{v}$	Velocity of source surface with components v_i	m/s
$\vec{x}$	Source position vector with components x_i	m
$\vec{y}$	Observer position vector with components y_i	m
c_o	Speed of propagation of acoustic wave	m/s
f	Acoustic wave frequency	Hz
$H(f)$	Heaviside function	
L	Eddy length scale	m
M	Local Mach number vector of source, with components M_i	
p	Pressure	Pa
P_{ij}	Compressive stress tensor	Pa
T_{ij}	LightHill stress tensor	Pa
U_c	Turbulent eddy convection speed	m/s

Greek symbols

λ	Acoustics Wavelength	m

Superscripts

’	Time varying component of fluid property

Subscripts

L	Loading Noise Component
n	Component of vector in direction normal to source surface
o	Fluid variable in quiescent medium

r	Component of vector in radiation direction
ref	Reference
ret	Quantity evaluated at retarded time
rms	Root mean square
T	Thickness Noise Component

Chapter 1

Introduction

Noise has increasingly become a topic of concern for several industrial applications. Noise in general is undesirable and can adversely affect the quality of our life. Flow noise generated by fans, vehicles, wind turbines, and propulsion systems are major contributors to this unwanted sound. Initial interest in understanding the noise generated by flows, revolved around the jet engine development during World War 2 to develop less noisy engines to avoid detection by the enemy. Today, noise generated by most industrial equipment is of significance. One such interesting area of application is in the electric vehicle development. As electric vehicles no longer have a internal combustion engine the the largest source of sound is the fan used in the cooling pack and therefore understanding and effectively reducing the sound produced by this low pressure axial fan is of great importance to produce an overall noiseless vehicle.

The study of the noise generated by air flows interacting with surrounding bodies is termed as aero-acoustics. Aero-acoustics in general is a computationally expensive field, as it involves very large and fine meshes. This is a result of having to simulate the source of the sound and then propagate the sound all the way down to the position of the far-field observer. While computational resources and power is increasing at all times, it is still important to find alternatives to simulating the entire region from the source of the sound to the far-field observer. The first understanding of how sound waves are generated by a turbulent flow, was provided in 1952. Sir James Lighthill published [1] his theory of aerodynamic sound and the subject of aeroacoustics was born. This theory, which is known as Lighthill's Acoustic Analogy, provides the basis for our understanding of sound generation by flow.

Lighthill's analogy addresses sound generation by a region of high speed turbulent flow in a stationary fluid. Lighthill determines the equations that describe the generation of sound waves that propagate to the acoustic far field, as distinct from defining the fluid motion in the turbulent flow. Solution to these equations lead to the Curle's theorem [2] and Ffowcs Williams and Hawking (FWH) theorem [3]. The Curle and FWH are used to predict far-field noise experienced by the observer by only solving for the source of the sound. This largely reduces the computationally expenses as this removes the necessity of meshing and solving the far field regions as well. The use of these analogies can be avoided by solving the entire flow field using Large Eddy Simulation (LES) or Direct Numerical Simulation (DNS) simulations however as expected these are extremely expensive and most often do not offer the benefit over using these analogies which are computationally significantly cheaper.

This report explores the methodology to compute the acoustic pressure waves generated by single sector of an axi-symmetric turbomachine using the FWH analogy and adapting the results to obtain the aero-acoustic pressure waves for the full annulus. This is done to eliminate the full annulus and therefore reduced computational effort significantly

Chapter 2

Theory

This chapter introduces the concepts, equations and terminologies required to understand the OpenFOAM implementation of the Ffowcs Williams and Hawkings (FWH) analogy using the Farassat 1A formulation (F1A). The F1A Formulation yields the time varying pressure perturbations generated by the turbulence created by a moving but rigid solid surface.

2.1 Introduction

Aero-acoustics is the study of noise generated by air flows. There are several sources of noise in engineering systems such as rotor noise, boundary layer noise, fan noise and air frame noise. Sound waves are essentially small perturbations in pressure that propagate through the fluid medium.

These sound waves are generated by turbulent eddies convected by the mean flow coming in contact with a solid body, which generates a rapid pressure change on the surface of the solid body. These rapid pressure changes propagate through the medium as sound waves. The frequencies of the fluctuations which results from this interaction are determined by the eddy size (L) and its convection velocity (U_c) and are calculated according to Eq. (2.1). The size of the eddies are usually in the same order of magnitude as the smallest dimension of the mean flow. The sound waves generated at this frequency will correspondingly have a wavelength (λ), calculated according to Eq. (2.2). In Eq. (2.2) c_o is the speed at which the the sound waves propagate through the medium. For a sound wave propagating through air, c_o is considered to be 343ms^{-1}.

$$f = \frac{U_c}{L} \tag{2.1}$$

$$\lambda = \frac{Lc_\mathrm{o}}{U_c} \tag{2.2}$$

The pressure at any point in the flow is a function of both the position and time and is given as the sum of the ambient pressure (p_o) and a time varying perturbation ($p^{'}(t)$). The time varying pressure perturbation is calculated according to Eq. (2.3). The human ear's sensitivity is logarithmic and is measured using a decibel scale, referred to as the sound pressure level (SPL) and is calculated according to Eq. (2.4), in terms of the root mean square of the fluctuating pressure time history (p_rms) and a reference pressure (p_ref). For almost all airborne applications the standard $p_\mathrm{ref} = 20\mu$Pa. Subsequently, p_rms is the time average of the square of the fluctuating pressure and calculated according to Eq. (2.5)

$$p^{'}(t) = p(t) - p_\mathrm{o} \tag{2.3}$$

$$\mathrm{SPL} = 20\log_{10}(\frac{p_{\mathrm{rms}}}{p_{\mathrm{ref}}}) \tag{2.4}$$

$$p_{rms} = \sqrt{\frac{1}{2T}\int_{-T}^{T}(p(t) - p_{\mathrm{o}})dt} \tag{2.5}$$

Aero-acoustic solvers aim to find this time varying pressure perturbation ($p^{'}(t)$), to subsequently calculate the SPL, frequency and amplitude of the acoustic waves experienced by the human ear.

2.2 Governing Equations

There are two main approaches in Computational Aero Acoustics (CAA).

1. **Direct Approach**: A transient solution is obtained by solving the compressible Navier-Stokes equations directly using Direct Numerical Simulation (DNS) or Large Eddy Simulation (LES) to obtain the far field pressure perturbations experienced by an observer. There are large differences in the scales between the flow variables and acoustic variables. Therefore the meshes need to be very fine and extremely small time steps must be employed. This approach is very computationally expensive. Additionally, the entire domain all the way till the far field observer needs to be meshed and simulated.

2. **Hybrid Approach**: Hybrid methods assume one-way coupling between the flow and acoustics. That is, the flow is independent of the acoustics. This assumption is valid for most low-mach and super-sonic applications but fails to be true in the hyper-sonic regime or regimes of large density variations. This allows the problem to be divided into two sections, with one being the flow solution and other, the propagation of sound waves. Therefore only the source of sound needs to be simulated and different analogies can then be implemented to compute the acoustic waves propagated to the far field.

2.2.1 Ffowcs Williams and Hawkings Analogy

The FWH equation is an exact rearrangement of the continuity equation and the Navier-Stokes equations into the form of an inhomogeneous wave equation with two surface source terms (monopole and dipole) and a volume source term (quadrupole). The purpose of a FWH surface is to provide a far field solution to the wave equation given an accurate numerical calculations on a surface which bounds the source region. The most useful applications of the FWH analogy is in the calculation of the acoustic far field from detailed numerical simulations of a flow within a limited region containing the source region. Recent advances in computational methods have enabled the accurate calculation of many time varying flows. But the computational domain is limited by the size of the computer, and usually cannot be extended to the acoustic far field. It is assumed that the CFD calculations accurately capture the pressure fluctuations, so that the FWH surface may be arbitrarily located within the numerical domain. This is important because the numerical calculations at the edges of the computational domain may be adversely influenced by numerical boundary conditions, so the FWH surface is usually placed inside the numerical domain in a region where there is confidence in the calculations.

The FWH analogy computes the far field acoustic pressure perturbation p' at any point outside the region of turbulence as function of time according to Eq. (2.6). Source term 1, term 2 and term 3 on the right hand side of Eq. (2.6) refer to the quadrupole, dipole and monopole terms respectively. The three source terms in the FWH equation each has its physical interpretation. The thickness noise (monopole source) is determined completely by the geometry and kinematics of the body. The loading noise (dipole source) is generated by the force that acts on the fluid as a result of the presence of the body. The quadrupole source term accounts for nonlinear effects (e.g., nonlinear wave propagation; variations in the local sound speed; and noise generated by shocks, vorticity and turbulence in the flow field) [4]. The three source terms are inter-dependent but the separation in their physical meaning allows for some flexibility depending upon the physics of the problem. For example, for a low speed flow the quadrupole source term can be neglected, similarly in the rotor plane only the thickness term (monopole source) is dominant and the other two source terms may be neglected. The main disadvantage of the traditional FWH approach is that to predict noise produced by a body operating in the transonic regime, the quadrupole source terms has to be included. The quadrupole source term is a voulme source term and therfore a volume integration has to be performed over the entire source region, which is very computationally expensive.

$$\Box^2 p^{'}(\vec{x},t) = \frac{\partial^2}{\partial x_i \partial x_j} T_{ij} H(f) - \frac{\partial}{\partial x_i}[P_{ij}\hat{n_j} + \rho u_i(u_n - v_n)\delta(f)] + \frac{\partial}{\partial t}[\rho_o v_n + \rho(u_n - v_n)\delta(f)] \tag{2.6}$$

$$T_{ij} = \rho u_i u_j + (p' - c_o^2 \rho')\delta_{ij} - \tau_{ij} \tag{2.7}$$

$$\Box^2 = \frac{1}{c^2}\frac{\partial^2}{\partial t^2} - \nabla^2 \tag{2.8}$$

2.2.2 Farassat 1A Formulation

The Farassat 1A Formulation [3] [5] provides an integral representation of the FWH equation, which does not take into consideration the quadrupole term (the volume source term) in Eq. (2.6). This assumption is valid when the flow is not in the transonic regime.

In Figure 2.1 Ω is the volume containing the moving surface. $\partial\Omega$ is the bounding FWH surface. $\vec{y}$ is the position of the far-field observer. $\vec{x}$ the position of the source at the point of integration on surface element. $\vec{r}$ is the distance between the observer and the source along the direction of radiation and is calculated according to Eq. (2.9). $\vec{n}$ is the outward facing normal to the FWH surface at the given point of integration. $\vec{v}$ is the velocity with which the solid body moves. Additionally all terms placed in square brackets are evaluated at retarded time that is with respect to the source Eq. (2.10), where τ is the source time and t is the observer time. M is the local Mach number vector of the source. The subscripts n, r depict the components of the respective vectors in the FWH surface normal direction and the radiation direction respectively. The summation of the p_T and the p_L term yields the total pressure fluctuation terms as a function of time.

Implementing these equations in a CFD code involves the computing p_T and the p_L terms for each mesh face constituting the FWH surface and summing them over to obtain the total contribution of the FWH surface and then finally repeating the same procedure for all FWH surfaces.

$$\vec{r} = \vec{x} - \vec{y} \tag{2.9}$$

$$\tau = t - r/c \tag{2.10}$$

$$p'(\vec{x},t) = p'_T(\vec{x},t) + p'_L(\vec{x},t) \tag{2.11}$$

$$U_i = \left(1 - \frac{\rho}{\rho_o}\right) v_i + \frac{\rho u_i}{\rho_o} \tag{2.12}$$

$$L_i = P_{ij}\vec{n_j} + \rho u_i(u_n + v_n) \tag{2.13}$$

$$4\pi p'_T(\vec{x},t) = \int_{f=0} \left[\frac{\rho_0(\dot{U}_n + U_{\dot{n}})}{r(1-M_r)^2}\right]_{ret} + \left[\frac{\rho_0 U_n(r\dot{M}_r + c_0(M_r - M^2))}{r^2(1-M_r)^3}\right]_{ret} d\Omega \tag{2.14}$$

$$4\pi c_o p'_L(\vec{x},t) = \int_{f=0} \left[\frac{\dot{L}_r}{r(1-M_r)^2}\right]_{ret} + c_0\left[\frac{L_r - L_M}{r^2(1-M_r)^2}\right]_{ret} + \left[\frac{L_r(r\dot{M}_r + c_o(M_r - M^2))}{r^2(1-M_r)^3}\right]_{ret} d\Omega \tag{2.15}$$

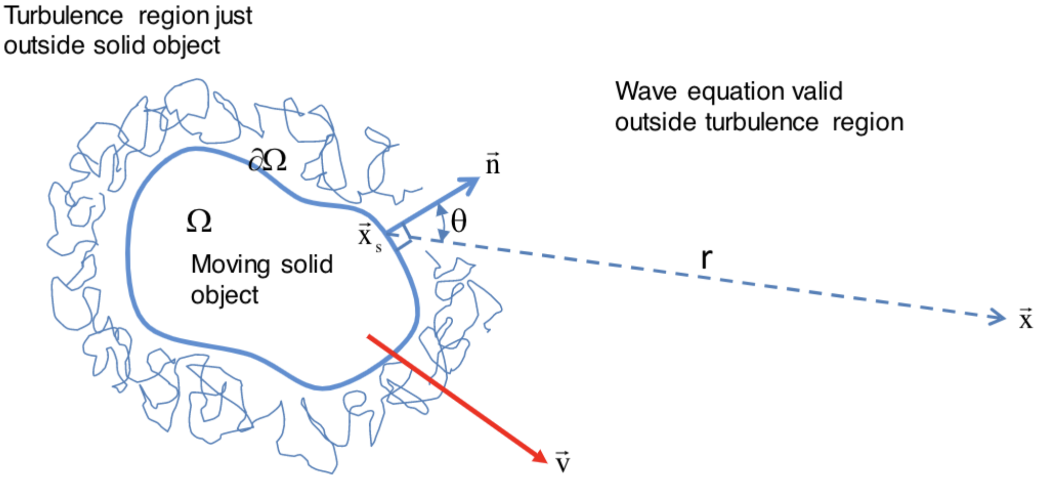

Figure 2.1: Turbulence and acoustic domain

Chapter 3

Aero-acoustic OpenFOAM Library

This chapter aims to explain the implementation of the Farassat 1A formulation for the FWH analogy in OpenFOAM. The existing aero-acoustic [6] library will be described here.

The library is available in github libAcoustics. The user will first need to be have a github account to download the files. The current library has three far-field prediction methods, namely the Curle, FWH and CFD-BEM coupling analogy. The library is also available for several ESI and Extend versions of OpenFOAM. All modules of the library have included in them a `wmakeAll.sh` file which can simply be run in the terminal window using the `\.wmakeAll.sh` command once OpenFOAM is sourced in the terminal. This document however deals only with the setup and modification of the FWH analogy using the F1A formulation.

3.1 File Structure

This OpenFOAM library is a function object which aims at implementing the FWH analogy to obtain the acoustic pressure fluctuations generated by FWH surfaces, i.e, surfaces that bound the source of the sound. This is done by finding the contribution of each face centre constituting the FWH surface to the thickness and loading term as described in Eq 2.14 and Eq 2.15 respectively. Subsequently the contribution of all the FWH surfaces are summed up to find the acoustic pressure experienced positioned in the acoustic far field.

There are several steps involved in obtaining the pressure fluctuation as a function of time for an observer positioned in the acoustic far-field region and then subsequently obtain the SPL, frequency and amplitude of the acoustic waves are obtained by using a fourier transform on the fluctuating pressure data. These steps are broken down into 5 different .C files in the existing function object.

1. `AcousticAnalogy.C` : This file reads the case setup dictionaries and identifies the various parameters required to setup the F1A formulation. It collects the following information from the case files : observer positions, speed of propagation of sound, the definition of FWH surfaces and far field density and fourier transform frequency. It also sets up the files and directories in which the output results will be stored.
2. `FfowcsWilliamsHawkings.C` : Initialises all variable collected by AcousticAnalogy.C and additionally defines functions to sample the surface pressure, surface density and surface velocity for any sampled FWH surface.
3. `fwhFormulation.C` : This file sets up all the intermediate geometric variables required for the calculation of F1A formulation. It mainly calculates the observer position with respect to each face centre constituting the FWH surfaces defined in the case dictionary.
4. `Farassat1AFormulation.C` : This file calculates the pressure fluctuation by all the FWH surfaces according to Eq. (2.14) and Eq. (2.15).

5. `SoundObserver.C` : This file performs the fourier transform according to Eq. to obtain the SPL and frequency of the acoustic pressure waves from the pressure fluctuation as a function of time

This library uses inheritance and friend classes extensively. A friend class can access private and protected members of another class in which it is declared as friend. The process of a child or sub-class taking on the functionality of a parent or super-class is referred to as inheritance. The Unified Modelling Language (UML) diagram as shown in Figure 3.1 illustrates the class inheritance and relationships to other classes in addition to the class attributes and methods. The attributes and methods of each class are listed in the top and bottom box respectively for each class. Inheritance between a sub-class and super-class is symbolised with a straight connected line with a closed hollow arrowhead pointing towards the super-class. Similarly, a friend class is illustrated by a straight solid arrow pointing towards the friend class. As illustrated in Figure 3.1, the `FfowcsWilliamsHawkings` class inherits from the `AcousticAnalogy` class while `Farassat1AFormulation` class inherits from the `fwhFormulation` class. Additionally, both `fwhFormulation` and `Farassat1AFormulation` are friend classes of the `FfowcsWilliamsHawkings` class. All attributes and methods of a class have different access levels depending on the access modifier or visibility. The different access levels are public (+), private (-), protected (#). Some of the attributes and methods in each of the classes has been listed in Figure 3.1 using different visibility options.

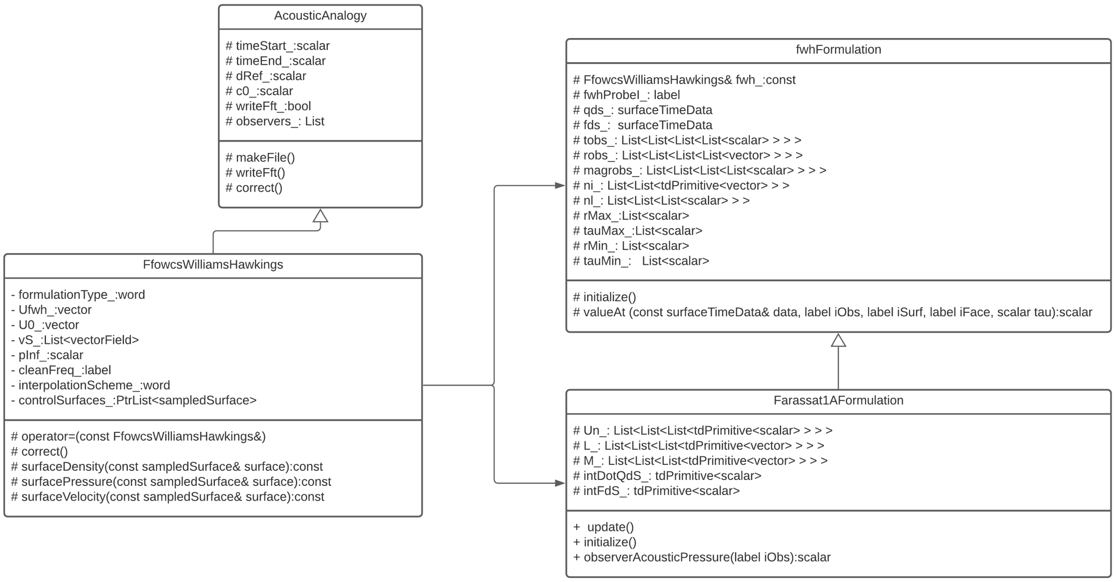

Figure 3.1: UML Diagram

3.2 AcousticAnalogy.C

This file reads data required for the the computation of the F1A formulation from the case setup dictionaries and creates folders and files required to save the output data. This file contains three main functions: the `makeFile`, `writeFft` and `read` functions.

This file extracts the following data from the case setup directories:

1. Simulation start time (`timeStart_`)
2. Simulation end time (`timeEnd_`)
3. Speed of propagation of acoustic waves (`c0_`)
4. Far field density (`rhoInf`)
5. List of observers (`observers_`)
6. Reference dimension (`dRef_`)
7. Reference pressure (`pRef_`)
8. Boolean declaring whether or not to perform a fourier transform to calculate the SPL and frequency data (`writeFft`)

The simulation start time and end time are declared in the `controlDict` dictionary. A separate `fwhControl` dictionary is introduced in which `writeFft`, `c0`, `dRef`, `rhoInf` have to be declared. A sub-dictionary called the observers dictionary declares the list of observer is included under `fwhControl` dictionary. It also declares the position of the observers in the cartesian coordinate system, the reference pressure (`pRef_`) required to calculate the SPL. `dRef_` is used to normalise the results when a 2D simulation is carried out. `dRef` should be set as the depth of the domain when carrying out 2D simulations and to -1 for 3D simulations. `observers_` is a list of observers declared in the case setup dictionary.

`Lines 18 - 65` creates two constructors and a destructor of the `AcousticAnalogy` class. It takes three inputs, depending upon the inputs one of the two constructors is selected.

Acoustic Analogy Constructors and Destructors

```
18 Foam::functionObjects::AcousticAnalogy::AcousticAnalogy
19 (
20     const word& name,
21     const Time& runTime,
22     const dictionary& dict
23 )
24 :
25     forces
26     (
27         name,
28         runTime,
29         dict
30     ),
31     analogyOutPtr_(nullptr),
32     timeStart_(-1.0),
33     timeEnd_(-1.0),
34     writeFft_(true),
35     c0_(343.0),
36     dRef_(-1.0),
37     observers_(0)
38 {
39 }
40
41 Foam::functionObjects::AcousticAnalogy::AcousticAnalogy
42 (
43     const word& name,
44     const objectRegistry& obr,
```

```
    const dictionary& dict
)
:
    forces
    (
        name,
        obr,
        dict
    ),
    analogyOutPtr_(nullptr),
    timeStart_(-1.0),
    timeEnd_(-1.0),
    writeFft_(true),
    c0_(343.0),
    dRef_(-1.0),
    observers_(0)
{
}

Foam::functionObjects::AcousticAnalogy::~AcousticAnalogy()
{}
```

The `makeFile()` function belonging to the `AcousticAnalogy` class is used to create the output directory. A top-level folder called `acoustic data` is created at the same level as the `system`, `control` and `0` directory. Within the `acoustic data` directory a file is created for every observer listed in the observer sub-dictionary mentioned in the fwhControl dictionary, and named according to the analogy used and the observer name. It then writes the headers for these files. The file contains two columns, namely `pFluct` and `Time`.

Acoustic Analogy `makeFile()|`

```
void Foam::functionObjects::AcousticAnalogy::makeFile()
{
    if (Pstream::master())
    {
        if(analogyOutPtr_.valid())
        {
            return;
        }
    }

    fileName ResultsDir;

    if (Pstream::master() && Pstream::parRun())
    {
        ResultsDir = obr_.time().rootPath() + "/" + obr_.time().caseName().path()  + "/acousticData";
        mkDir(ResultsDir);
    }
    else if (!Pstream::parRun())
    {
        ResultsDir = obr_.time().rootPath() + "/" + obr_.time().caseName() + "/acousticData";
        mkDir(ResultsDir);
    }
    else
    {
    }

    // File update
    if (Pstream::master() || !Pstream::parRun())
    {

        analogyOutPtr_.set
        (
            new OFstream
            (
                ResultsDir + "/" + (name() + "-time.dat")
            )
```

```
103         );
104 
105         analogyOutPtr_() << "Time" << " ";
106         forAll(observers_, iObserver)
107         {
108             analogyOutPtr_() << observers_[iObserver].name() << "_pFluct ";
109         }
110         analogyOutPtr_() << endl;
111     }
112 }
```

The `writeFft()` function belonging to the `AcousticAnalogy` class is used to write the SPL, pressure fluctuation and frequency of the acoustic waves experienced by the observers. `Lines 114 - 126` defines the location of the results directory `acoustic data` which as mentioned previously lies in the same level as the `system`, `control` and `0` directory. `Lines 127 - 165` first calculate the simulation time `tau`, at which the fourier transform needs to be calculated and then for each observer the fourier transform is carried out which yields the SPL and frequency and is saved under their respective files.

Acoustic Analogy writeFft() function

```
114 void Foam::functionObjects::AcousticAnalogy::writeFft()
115 {
116     fileName ResultsDir;
117 
118     if (Pstream::master() && Pstream::parRun())
119     {
120         ResultsDir = obr_.time().rootPath() + "/" + obr_.time().caseName().path()  + "/acousticData";
121     }
122     else if (!Pstream::parRun())
123     {
124         ResultsDir = obr_.time().rootPath() + "/" + obr_.time().caseName() + "/acousticData";
125     }
126 
127     if (Pstream::master() || !Pstream::parRun())
128     {
129         const fvMesh& mesh = refCast<const fvMesh>(obr_);
130 
131         scalar  tau;
132         if (mesh.time().startTime().value() > timeStart_)
133         {
134             tau = (mesh.time().value() - mesh.time().startTime().value());
135         }
136         else
137         {
138             tau = (mesh.time().value() - timeStart_);
139         }
140 
141         forAll(observers_, iObserver)
142         {
143             SoundObserver& obs = observers_[iObserver];
144             autoPtr<List<List<scalar> > > obsFftPtr (obs.fft(tau));
145 
146             List<List<scalar> >& obsFft = obsFftPtr();
147 
148             if (obsFft[0].size() > 0)
149             {
150                 Log << "Executing fft for obs: " << obs.name() << endl;
151                 fileName fftFile = ResultsDir + "/fft-" + name() + "-" + obs.name() + ".dat";
152 
153                 OFstream fftStream (fftFile);
154                 fftStream << "Freq p\' spl" << endl;
155 
156                 forAll(obsFft[0], k)
157                 {
158                     fftStream << obsFft[0][k] << " " << obsFft[1][k] << " " << obsFft[2][k] << endl;
159                 }
160 
```

```
                fftStream.flush();
            }
        }
    }
}
```

The `read()` function belonging to the `AcousticAnalogy` class is used to read all the inputs provided in the case dictionary. The case dictionary needs to contain the `timeStart_`, `timeEnd_` which refer to the simulation start time and end time. These are mentioned in the `controlDict` dictionary. All of the above mentioned inputs are read by the `read()` function. Following which the `makeFile()` function belonging to the `AcousticAnalogy` class is called upon to create the files and dictionaries to store the output.

Acoustic Analogy read() function

```
bool Foam::functionObjects::AcousticAnalogy::read(const dictionary& dict)
{
    if (!forces::read(dict))
    {
        return false;
    }
    Info << "Reading analogy settings" << endl;

    dict.lookup("timeStart") >> timeStart_;

    dict.lookup("timeEnd") >> timeEnd_;

    dict.lookup("writeFft") >> writeFft_;

    dict.lookup("c0") >> c0_;

    dict.lookup("dRef") >> dRef_;

    dict.lookup("rhoInf") >> rhoRef_;

    //read observers
    {
        const dictionary& obsDict = dict.subDict("observers");
        wordList obsNames = obsDict.toc();
        forAll (obsNames, obsI)
        {
            word oname = obsNames[obsI];
            vector opos (vector::zero);
            obsDict.subDict(oname).lookup("position") >> opos;
            scalar pref = 2.0e-5;
            obsDict.subDict(oname).lookup("pRef") >> pref;
            label fftFreq = 1024;
            obsDict.subDict(oname).lookup("fftFreq") >> fftFreq;

            observers_.append
            (
                SoundObserver
                (
                    oname,
                    opos,
                    pref,
                    fftFreq
                )
            );
        }
    }

    this->makeFile();

    return true;
```

3.3 FfowcsWilliamsHawkings.C

This file initialises all variable collected by `AcousticAnalogy.C` and additionally defines functions to sample the surface pressure, surface density and surface velocity for any sampled FWH surface. `FfowcsWilliamsHawkings.C` inherits from the `AcousticAnalogy` class and has as friend class both `fwhFormulation` class and `Farassat1AFormulation` class. The main purpose of this file is to define the functions for obtaining the pressure, velocity and density on the sampled surface. `Lines 135 - 162` initialise the variable `vS` which is used to store the velocity at the face centres of the sampled FWH surfaces. It then checks if the formulation type declared in the case setup dictionary is correct, if not it throws an error.

FfowcsWilliamsHawkings::initialize()

```
135 // * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * //
136
137 void Foam::functionObjects::FfowcsWilliamsHawkings::initialize()
138 {
139
140     vS_.resize(controlSurfaces_.size());
141     forAll(controlSurfaces_, iSurf)
142     {
143         vS_[iSurf].resize(controlSurfaces_[iSurf].Cf().size());
144         vS_[iSurf] = vector::zero;
145     }
146
147     //Allocate pointer to FWH formulation
148     if (formulationType_ == "Farassat1AFormulation")
149     {
150             fwhFormulationPtr_.set
151             (
152                 new Farassat1AFormulation(*this)
153             );
154     }
155     else
156     {
157         Info << "Wrong formulation type: " << formulationType_ << endl
158         << "Please, select: " << endl
159          << "1) Farassat1AFormulation " << endl;
160     }
161
162 }
```

The sub-class`FfowcsWilliamsHawkings` inherits from the `AcousticAnalogy`, the super-class and therefore has access to all the objects from the `AcousticAnalogy` class. All the variables declared in the dictionary in the case setup and read by the read() function in the `AcousticAnalogy` class is made available to the `FfowcsWilliamsHawkings` class. These variables are then stored under respective variable names as seen in `lines 171-178`. Additionally the list of FWH surfaces are stored in `controlSurfaces_`.

FfowcsWilliamsHawkings::read()

```
164 bool Foam::functionObjects::FfowcsWilliamsHawkings::read(const dictionary& dict)
165 {
166     if (!AcousticAnalogy::read(dict))
167     {
168         return false;
169     }
170
171     dict.lookup("formulationType") >> formulationType_;
172     dict.lookup("Blades") >> Blades_;
173     dict.lookup("Ufwh") >> Ufwh_;
174     dict.lookup("U0") >> U0_;
175     dict.lookup("pInf") >> pInf_;
176     dict.lookup("interpolationScheme") >> interpolationScheme_;
177     dict.lookup("cleanFreq") >> cleanFreq_;
```

```
178
179     const fvMesh& mesh = refCast<const fvMesh>(obr_);
180     PtrList<sampledSurface> newList
181     (
182         dict.lookup("surfaces"),
183         sampledSurface::iNew(mesh)
184     );
```

`Lines 285 - 322` create three function objects under the `FfowcsWilliamsHawkings` class which compute the surface pressure, density, velocity for a given sampled surface. These function objects will be used later to sample the aforementioned quantities on the FWH control surfaces.

FfowcsWilliamsHawkings::surfaceDensity/surfaceVelocity/surfacePressure

```
285
286 Foam::tmp<Foam::scalarField> Foam::functionObjects::FfowcsWilliamsHawkings::surfaceDensity(const sampledSurface& surface) const
287 {
288     tmp<Field<scalar> > rhoSampled
289     (
290         sampleOrInterpolate<scalar>(this->rho()(), surface)
291     );
292
293     return rhoSampled;
294 }
295
296 Foam::tmp<Foam::vectorField> Foam::functionObjects::FfowcsWilliamsHawkings::surfaceVelocity(const sampledSurface& surface) const
297 {
298     const volVectorField& U = obr_.lookupObject<volVectorField>("U");
299
300     tmp<Field<vector> > USampled;
301
302     USampled = sampleOrInterpolate<vector>(U , surface);
303
304     return USampled;
305 }
306
307 Foam::tmp<Foam::scalarField> Foam::functionObjects::FfowcsWilliamsHawkings::surfacePressure(const sampledSurface& surface) const
308 {
309     tmp<Field<scalar> > pSampled;
310     const volScalarField& p = obr_.lookupObject<volScalarField>(pName_);
311
312     pSampled = sampleOrInterpolate<scalar>(p , surface);
313
314     if (p.dimensions() != dimPressure)
315     {
316         pSampled.ref() *= rhoRef_;
317     }
318
319     //Info << pSampled() << endl;
320
321     return pSampled;
322 }
```

3.4 fwhFormulation.C

This file computes the distance between each face centre of each of the FWH control surfaces to every observer and then subsequently calculates the time required by a pressure fluctuation generated from each of those face centres to reach the observer. The `fwhFormulation` constructor defined in between `lines 4 - 20`, takes as input a reference to the `FfocwsWilliamsHawkings` class. As mentioned in section 3.3, `fwhFormulation` class is a friend class to `FfowcsWilliamsHawkings` class and therefore will have access to all private and protected member functions of the `FfowcsWilliamsHawkings` class.

fwhFormulation

```
 1 #include "FfowcsWilliamsHawkings.H"
 2 #include "fwhFormulation.H"
 3
 4 Foam::functionObjects::fwhFormulation::fwhFormulation(const FfowcsWilliamsHawkings& fwh)
 5 :
 6     fwh_(fwh),
 7     qds_(0),
 8     fds_(0),
 9     tobs_(0),
10     robs_(0),
11     magrobs_(0),
12     ni_(0),
13     nl_(0),
14     rMax_(0),
15     tauMax_(0),
16     tauMin_(0)
17 {
18     this->initialize();
19 }
```

`Lines 23 - 47` resize all required variables to have a size equal to the number of observers times the number of control surfaces times the number of faces per control surface. As an example, assume there are two observers, three FWH control surfaces and the three FWH control surfaces have 100,200, and 300 faces respectively. In such a case all required variables will be sized to 2 * 3 *100/200/300 respectively.

`Lines 23 - 36` resize the variables `qds` and `fds` to the aforementioned sizes. Similarly, `lines 36 - 47` resize `tobs` to the aforementioned sizes. The object `tobs` refers to `t` in Eq 2.10 and stores the time required by the acoustic pressure wave to travel from the face centre to the observer. `tauMax` and `rMax` store the maximum time and maximum distance between the source nd the observer for every FWH control surface. Both `tauMax` and `rMax` are initialised to a size equal to the number of observers in `lines 48 - 49`. `Lines 55 - 94` initialise objects `robs` and `magrobs`, which respectively store the $\vec{r}$ and the magnitude of $\vec{r}$ defined in Eq 2.9. These variables will later be used to calculate the p'_T and p'_L terms respectively for all face centres constituting the FWH control surfaces.

fwhformulation::initialize()

```
21 void Foam::functionObjects::fwhFormulation::initialize()
22 {
23     //allocate qds_, fds_ and vds_
24     qds_.resize(fwh_.observers_.size());
25     fds_.resize(fwh_.observers_.size());
26     forAll(fwh_.observers_, iObs)
27     {
28         qds_[iObs].resize(fwh_.controlSurfaces_.size());
29         fds_[iObs].resize(fwh_.controlSurfaces_.size());
30         forAll(fwh_.controlSurfaces_, iSurf)
31         {
32             qds_[iObs][iSurf].resize(fwh_.controlSurfaces_[iSurf].Cf().size());
33             fds_[iObs][iSurf].resize(fwh_.controlSurfaces_[iSurf].Cf().size());
34         }
```

```
35     }
36 
37     //allocate tobs
38     tobs_.resize(fwh_.observers_.size());
39     forAll(fwh_.observers_, iObs)
40     {
41         tobs_[iObs].resize(fwh_.controlSurfaces_.size());
42         forAll(fwh_.controlSurfaces_, iSurf)
43         {
44             tobs_[iObs][iSurf].resize(fwh_.controlSurfaces_[iSurf].Cf().size());
45         }
46     }
47 
48     tauMax_.resize(fwh_.observers_.size(), 0.0);
49     rMax_.resize(fwh_.observers_.size(), 0.0);
50 
51     //allocate robs
52     robs_.resize(fwh_.observers_.size());
53     magrobs_.resize(fwh_.observers_.size());
```

`Line 55` of `fwhFormulation` class initiates a `forAll` loop which is executed for each observer. In `line 57` the value of `rMax` for each observer is initially set to zero. In `line 58` a reference `obs` of the `SoundObserver` class is created. `Lines 59 - 60` resize `robs` and `magrobs` to the number of observer times the number of FWH control surfaces.

`Line 61` initiates a `forAll` loop which is executed for each control surface. `Lines 63 - 64` resize `robs` and `magrobs` to the number of observer times the number of FWH control surfaces times the number of face centres constituting the FWH control surface. `Cf()` returns the face centres for each given FWH control surface and `Cf().size()` returns the number of face centres for a given surface. `Line 66` initiates a `forAll` loop which is executed for each control face centre on the current FWH control surface.

`Line 68` computes the distance between the current face centre of a FWH control surface and an observer by taking the vector difference between the source position (`Cf`)and the observer position (`obs.position`) according to Eq 2.9, and stores it in the object `robs`, which has three componets, namely in the `x`, `y` and `z` direction. These components can be addressed using `r[0]`, `r[1]` and `r[2]` respectively. Subsequently, this value is assigned to a temporary vector `r` as seen in `line 69`.

`U0` and `c0` refer to the background velocity of at the observer if any and the speed of propagation of acoustic pressure waves, which in most cases is the speed of sound itself. `Lines 70 - 94` compute the magnitude of the $\vec{r}$ calculated according to Eq 2.9

fwhformulation::initialize()

```
55     forAll(fwh_.observers_, iObs)
56     {
57         rMax_[iObs] = 0.0;
58         const SoundObserver& obs = fwh_.observers_[iObs];
59         robs_[iObs].resize(fwh_.controlSurfaces_.size());
60         magrobs_[iObs].resize(fwh_.controlSurfaces_.size());
61         forAll(fwh_.controlSurfaces_, iSurf)
62         {
63             robs_[iObs][iSurf].resize(fwh_.controlSurfaces_[iSurf].Cf().size());
64             magrobs_[iObs][iSurf].resize(fwh_.controlSurfaces_[iSurf].Cf().size());
65             const vectorField& Cf = fwh_.controlSurfaces_[iSurf].Cf();
66             forAll(Cf, i)
67             {
68                 robs_[iObs][iSurf][i] = obs.position() - Cf[i];
69                 vector r = robs_[iObs][iSurf][i];
70                 scalar R_ = sqrt
71                     (
72                     sqr(r[0])
73                     +
```

```
 74                     ( 1 - sqr(mag(fwh_.U0_)/fwh_.c0_))
 75                     *
 76                     ( sqr(r[1]) + sqr(r[2]) )
 77                     );
 78 
 79         magrobs_[iObs][iSurf][i] =
 80             (
 81             -(mag(fwh_.U0_)/fwh_.c0_) * r[0] + R_
 82             ) / (1 - sqr(mag(fwh_.U0_)/fwh_.c0_));
 83 
 84 
 85                 if (magrobs_[iObs][iSurf][i] > rMax_[iObs])
 86                 {
 87                     rMax_[iObs] = magrobs_[iObs][iSurf][i];
 88                 }
 89             }
 90         }
 91         reduce(rMax_[iObs], maxOp<scalar>());
 92         tauMax_[iObs] = rMax_[iObs] / fwh_.c0_;
 93         reduce(tauMax_[iObs], maxOp<scalar>());
 94     }
```

`Lines 96 - 127` aim to calculate the direction of the normal of each face associated to a FWH control surface. `Line 97` declares an object `Cs` which is a list of vectors containing the size of the control surfaces declared. As done previously `lines 98 - 99` resize `ni` and `nl` to the number of FWH control surfaces. `Line  Sf()` returns the face area vector. `Line 100` initiates a `forAll` to iterate over the number of control surfaces. For each control surface two vector fields `Sf` and `Cf` are declared in `Lines 102 - 103` to store the face area vectors and the face centre vector. In `Lines 104-105` `ni` and `nl` are resized to the number of faces constituting that particular face.

`Lines 107-108` introduce a scalar `surfSize` which is used to store the number of faces per control surface and the `reduce` operation is used to sum across processors. `magSf` is used to store the magnitude of the face area vectors, which is initialised to zero at `line 111`. In `line 112` a `forAll` is initialised to iterate over the number of face centres per control surface. On line `line 114`, `magSf` is assigned the value of the magnitude of the face area vector assigned to that face. On `line 115`, `ni` for each face is assigned the direction vector of the face area vector. This achieved by dividing the face area vector by its magnitude. `Lines 116 -  123` introduces an if condition, where if the inner product between face centre direction vector and the face area direction vector is positive, `nl` is assigned the value one and minus one if it is negative.

fwhformulation::initialize()

```
 96     //calculate normals
 97     List<vector> Cs(fwh_.controlSurfaces_.size());
 98     ni_.resize(fwh_.controlSurfaces_.size());
 99     nl_.resize(fwh_.controlSurfaces_.size());
100     forAll(ni_, iSurf)
101     {
102         const vectorField& Sf = fwh_.controlSurfaces_[iSurf].Sf();
103         const vectorField& Cf = fwh_.controlSurfaces_[iSurf].Cf();
104         ni_[iSurf].resize(Cf.size());
105         nl_[iSurf].resize(Cf.size());
106         Cs[iSurf] = gSum(Cf);
107         scalar surfSize = scalar(Cf.size());
108         reduce (surfSize, sumOp<scalar>());
109         Cs[iSurf] /= surfSize;
110 
111         scalar magSf = 0.0;
112         forAll(ni_[iSurf], iFace)
113         {
114             magSf = mag(Sf[iFace]);
115             ni_[iSurf].value(iFace) = Sf[iFace]/magSf;
116             if ( ((Cf[iFace] - Cs[iSurf]) & ni_[iSurf].value(iFace)) > 0 )
117             {
```

```
                nl_[iSurf][iFace] = 1.0;
            }
            else
            {
                nl_[iSurf][iFace] = -1.0;
            }
            ni_[iSurf].value(iFace) *= nl_[iSurf][iFace];
        }
    }
}
```

3.5 Farassat1AFormulation.C

`Lines 34 - 37` define a constructor for `Farassat1AFormulation.C` which takes as input, a reference object to the `FfocwsWlliamsHawkings` class. Lines 6-12 include the declaration of the variables that are required in the calculation of the Farassat 1A formulation. All of them are initially set to zero. `Lines 46 - 48` call on the initialise function, which is in turn defined between `lines 57 - 79`. The initialise function sets up the size of the three variables L_r, M_r, U_n using the resize command. Initially the member objects are set to have a size equal to the number of observers as seen in `lines 59 - 64`. Every observer will have a certain contribution from each control surface ($\partial\Omega$ in Figure 2.1) and therefore an additional dimension is added to include the number of control surfaces. The contribution of each control surface is calculated by summing up the contribution of every face constituting that control surface. Therefore an additional dimension of the number of face centres constituting each control surface is included for each member object. `Cf()` returns a surface field vector containing face centres and `Cf.size()` returns the number of face centres.

Farassat 1A Formulation

```
34 Foam::functionObjects::Farassat1AFormulation::Farassat1AFormulation
35 (
36     const FfowcsWilliamsHawkings& fwh
37 )
38 :
39     fwhFormulation(fwh),
40     Un_(0),
41     Lr_(0),
42     Mr_(0),
43 
44     intDotQdS_(0.0, fwh_.obr_.time().value()),
45     intFdS_(0.0, fwh_.obr_.time().value())
46 {
47     this->initialize();
48 }
49 
50 // * * * * * * * * * * * * * * * * * Destructor  * * * * * * * * * * * * * * * //
51 
52 Foam::functionObjects::Farassat1AFormulation::~Farassat1AFormulation()
53 {}
54 
55 
56 // * * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * //
57 void Foam::functionObjects::Farassat1AFormulation::initialize()
58 {
59     intFdS_.resize(fwh_.observers_.size());
60     intDotQdS_.resize(fwh_.observers_.size());
61 
62     Lr_.resize(fwh_.observers_.size());
63     Mr_.resize(fwh_.observers_.size());
64     Un_.resize(fwh_.observers_.size());
65 
66     forAll(Lr_, iObs)
67     {
68         Lr_[iObs].resize(fwh_.controlSurfaces_.size());
69         Mr_[iObs].resize(fwh_.controlSurfaces_.size());
70         Un_[iObs].resize(fwh_.controlSurfaces_.size());
71 
72         forAll(Lr_[iObs], iSurf)
73         {
74             Lr_[iObs][iSurf].resize(fwh_.controlSurfaces_[iSurf].Cf().size());
75             Mr_[iObs][iSurf].resize(fwh_.controlSurfaces_[iSurf].Cf().size());
76             Un_[iObs][iSurf].resize(fwh_.controlSurfaces_[iSurf].Cf().size());
77         }
78     }
79 }
```

`Line 81` defines a new member function belonging to the `Farassat1AFormulation` class which takes as input, the number of observers as iObs. Lines 86-109 define the initial value of all the terms required in Eq. (2.14) and Eq. (2.15).

Farassat 1A Formulation

```
 81 Foam::scalar Foam::functionObjects::Farassat1AFormulation::observerAcousticPressure(label iObs)
 82 {
 83     scalar ct = fwh_.obr_.time().value();
 84 
 85         //Farassat 1A
 86     vector L (vector::zero);
 87     scalar lr (0.0);
 88     scalar lM (0.0);
 89     scalar dotlr (0.0);
 90     vector r  (vector::zero);
 91     vector rh (vector::zero);
 92     vector n  (vector::zero);
 93     scalar dS (0.0);
 94     scalar magr(0.0);
 95     vector M  (vector::zero);
 96     scalar magM (0.0);
 97     scalar Mr (0.0);
 98     scalar dotMr (0.0);
 99     tensor Pf (tensor::zero);
100     scalar OneByOneMr(0.0);
101     scalar OneByOneMrSq(0.0);
102 
103     scalar fpart1 (0.0);
104     scalar fpart2 (0.0);
105     scalar fpart3 (0.0);
106     vector U(vector::zero);
107     scalar Un(0.0);
108     scalar dotUn(0.0);
109     vector dotn(vector::zero);
```

In `line 120 Sf()` is used to return the face area vectors for the sampled FWH control surface. `Line 121 - 123` samples the surface pressure, density and velocity for the sampled FWH control surface. Subsequently for each of the sampled FWH control surface the, $\vec{r}$ is calculated for each face constituting the control surface. $\vec{r}$ is the distance between the source face centre and the observer position. The definition of `robs` is in `fwhFormulation.C`. M is the mach number at the face centre and is calculated using the velocity at the face centre `vS` and the speed of sound propagation c0. `Line 141` corresponds to Eq. (2.12), `Line 143` corresponds to Eq. (2.13). `Lines 145 - 150` corresponds to obtaining the inner products along both the control surface normal ($\vec{n}$) and along direction of radiation ($\vec{r}$). `Lines 154 - 156` obtain the L_r, M_r, U_n at each face centre.

Farassat 1A Formulation

```
111     forAll(fwh_.controlSurfaces_, iSurf)
112     {
113         const sampledSurface& surf = fwh_.controlSurfaces_[iSurf];
114         if (surf.interpolate())
115         {
116             Info<< "WARNING: Interpolation for surface " << surf.name() << " is on, turn it off"
117                 << endl;
118         }
119 
120         const vectorField& Sf = surf.Sf();
121         vectorField uS (fwh_.surfaceVelocity(surf)());
122         scalarField rhoS (fwh_.surfaceDensity(surf)());
123         scalarField pS (fwh_.surfacePressure(surf)() - fwh_.pInf_);
124 
125         //Farassat 1A formulation
126         forAll(Sf, iFace)
127         {
```

```
128
129             //For observe No iObs
130             {
131                 r = robs_[iObs][iSurf][iFace];
132                 magr = magrobs_[iObs][iSurf][iFace];
133                 rh = r / magr;
134                 dS = mag(Sf[iFace]);
135                 n = ni_[iSurf].value(iFace);
136
137                 M = fwh_.vS_[iSurf][iFace] / fwh_.c0_;
138                 Mr = M & rh;
139                 magM = mag(M);
140
141                 U = (1.0 - rhoS[iFace] / fwh_.rhoRef_) * fwh_.vS_[iSurf][iFace]
142                     + rhoS[iFace] * uS[iFace] / fwh_.rhoRef_;
143         Pf = pS[iFace]*tensor::I + rhoS[iFace]*uS[iFace]*(uS[iFace] - fwh_.vS_[iSurf][iFace]);
144
145                 L = Pf & n;
146                 lM = L & M;
147         lr = L & rh;
148                 Mr = M & rh;
149                 Un = U & n;
150
151                 OneByOneMr = 1.0 / (1.0 - Mr);
152                 OneByOneMrSq = OneByOneMr*OneByOneMr;
153
154                 Un_[iObs][iSurf].value(iFace) = Un;
155                 Lr_[iObs][iSurf].value(iFace) = lr;
156                 Mr_[iObs][iSurf].value(iFace) = Mr;
```

`Lines 158-161` compute the time derivative of L_r, M_r, U_n. The dot operator is defined in the `FfowcsWilliamsHawkings.C` file. `Line 163` first creates a column in which the observer time `tobs` is added and then computes p'_T according to Eq. (2.14) between `lines 174 - 177`. Similarly `lines 130 - 133` computes the three terms on the right hand side in Eq. (2.15). Finally the contribution of all the faces constituting the face centres are summed in `lines 195 - 213`

Farassat 1A Formulation

```
157
158                 dotlr = Lr_[iObs][iSurf].dot(ct, iFace);
159                 dotMr = Mr_[iObs][iSurf].dot(ct, iFace);
160                 dotUn = Un_[iObs][iSurf].dot(ct, iFace);
161                 dotn = ni_[iSurf].dot(ct, iFace);
162
163                 qds_[iObs][iSurf][iFace].first().append(tobs_[iObs][iSurf][iFace]);
164                 qds_[iObs][iSurf][iFace].second().append
165                 (
166                     (
167                         fwh_.rhoRef_ * (dotUn + (U & dotn)) * OneByOneMrSq / magr
168                         +
169                         fwh_.rhoRef_ * Un * (magr * dotMr + fwh_.c0_ * (Mr - magM*magM)) *
170                         OneByOneMrSq * OneByOneMr / magr / magr
171                     )*dS
172                 );
173
174                 fpart1 = dotlr * (dS / magr / fwh_.c0_) * OneByOneMrSq;
175                 fpart2 = (lr - lM) * (dS / magr / magr) * OneByOneMrSq;
176                 fpart3 = lr * (dS / magr / magr / fwh_.c0_) * OneByOneMrSq * OneByOneMr *
177                         (magr * dotMr + fwh_.c0_ * Mr - fwh_.c0_ * magM * magM);
178
179                 fds_[iObs][iSurf][iFace].first().append(tobs_[iObs][iSurf][iFace]);
180                 fds_[iObs][iSurf][iFace].second().append
181                 (
182                         fpart1 + fpart2 + fpart3
183                 );
184
185             }//observer
```

```
        } //For Sf
    } // for controlSurfaces_

    scalar ct1 = ct+fwh_.obr_.time().deltaT().value()*1.0e-6;//slightly increase time to get inside of time step

    scalar retv = 0.0;
    intDotQdS_.value(iObs) = 0.0;
    intFdS_.value(iObs)    = 0.0;
    //calculate acoustic pressure, zero if source didn't reached observer
    forAll(fwh_.controlSurfaces_, iSurf)
    {
        forAll(qds_[iObs][iSurf], iFace)
        {
            retv = valueAt(qds_, iObs, iSurf, iFace, ct1);

            intDotQdS_.value(iObs) += retv;
            retv = valueAt(fds_, iObs, iSurf, iFace, ct1);
            intFdS_.value(iObs) += retv;
        }
    }

    reduce (intDotQdS_.value(iObs), sumOp<scalar>());
```

3.6 SoundObserver.C

This file contains the function for calculating the fourier transform. The details of how the fourier transform is performed is not looked at in this report. Instead, we just look at the input and output. It requires as input the fluctuating pressure component as a function of time. The fluctuating pressure and time is obtained from the F1A formulation file and is used as input into the fft function as `p_` and `tau`. The first output from the fft function is Frequency in Hz, the second output is the Pressure amplitude in Pa and the third output is the Sound Pressure Level (SPL) in dB.

SoundObserver

```
Foam::autoPtr<Foam::List<Foam::List<Foam::scalar> > > Foam::SoundObserver::fft(scalar tau) const
{

    List<List<scalar> > fft_res(3);
    forAll (fft_res, i)
    {
        fft_res[i].resize(0);
    }

    if ( (p_.size() > 0) && (p_.size() % fftFreq_ == 0) )
    {
        FoamFftwDriver fftw (p_, tau);

        autoPtr<Pair<List<scalar> > > pfft = fftw.simpleScalarForwardTransform();

        fft_res[0].resize(pfft().first().size());
        fft_res[1].resize(pfft().first().size());
        fft_res[2].resize(pfft().first().size());

        forAll (pfft().first(), k)
        {
            fft_res[0][k] = pfft().first()[k]; //Frequency, Hz
            fft_res[1][k] = pfft().second()[k]; //pressure amplitude, Pa
            if (fft_res[1][k] > SMALL)
            {
                fft_res[2][k] = 20*log10(fft_res[1][k] / pref_); //SPL, dB
            }
            else
            {
                fft_res[2][k] = 0.0;
```

```
            }
        }
    }

    return autoPtr<List<List<scalar> > >
    (
        new List<List<scalar> >
        (
            fft_res
        )
    );
}

//
//END-OF-FILE
//
```

Chapter 4

Modification of Library

This section will focus on adapting the existing library, to obtain the result for an entire 360 degree sector from the results obtained by simulating a single sector of a turbomachine.

The current library produces $p'(\vec{x}, t)$, SPL, frequency and amplitude of the acoustic waves for every observer. However, this library does not accommodate the simulation of a single sector of an axi-symmetric model and then subsequently adjusting the results for a full annulus.

Consider Figure 4.1, where the blue surface is a FWH control surface, which bounds the acoustic source and lies within the simulated domain, and the grey surface is a copy of the blue surface obtained by rotating the blue surface by an angle equal to 360°/ number of blades. Here, let us assume that there are four blades and therefore the grey surface is obtained by rotating the blue surface by 90°. The grey surface is not a part of the generated mesh but instead is just used to calculate the acoustic pressure wave, by using the results from the simulated FWH control surface, in this case the blue surface.

In Figure 4.1, the red coloured face on the grey control surface is obtained by rotating the red coloured face on the blue surface by 90°. Both the red faces will have the same L_i and U_i, as defined in Eq 2.13 and Eq 2.12. Similarly the green faces will have the same L_i and U_i. However, the distance and orientation of the red or green faces to the observer will be different and therefore will have different contribution to the thickness and loading acoustic pressure wave term, to the observer placed in the acoustic far-field. Each FWH control surface will be copied and rotated by the number of sectors required to form a full annulus. As an example, if the turbo-machine contains 4 blades, each control surface will be copied and rotated 4 times, each time by an angle of 90 degrees. Therefore to obtain the contribution of the copied sectors, an additional dimension of integration needs to be added. Apart from integrating over the number of control surfaces and the number of faces constituting the control surfaces, the p'_T and p'_L contribution from the copied sectors need to be accounted for.

The position of the copied sector is not identical to the original sector and therefore needs to be accounted for. This is done by manipulating the face centre values and surface normal vectors for each of the copied sectors. An additional input indicating the number of sectors to complete the full annulus now needs to be included in the case dictionary files. In the below code, this is referred to as `Blades`, which is member function of the `FfowcsWilliamsHawking` class. It can be seen in `lines 29 - 31` an additional dimension for the number of blades is now included.

In `lines 75` it can be seen that angle by which the control surface is to be rotated by is calculated simply as the full annulus angle (360°) divided by the number of blades and incremented in a step wise manner to the total number of blades. Once the rotation angle is obtained both the face

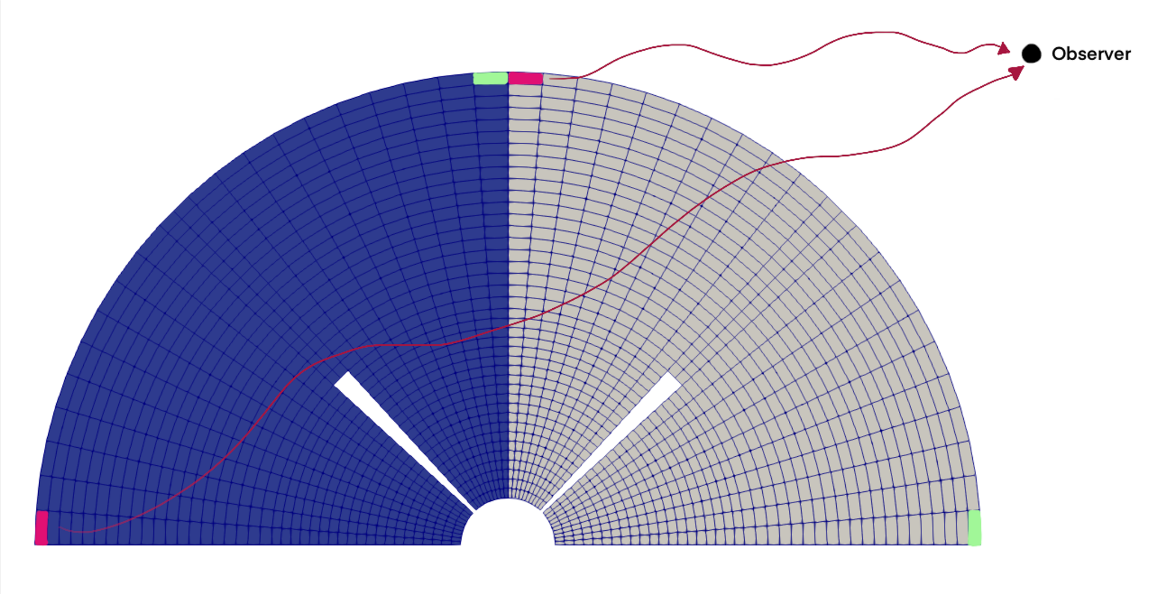

Figure 4.1: FWH Surface rotated around z-axis

centre values and the surface normal values are transformed using a rotation matrix as shown in Eq. 4. x, y, z' are the rotated coordinates about the z-axis and x, y, z are the original coordinates. The transformed coordinates now represent the face centre and surface normal vectors of the rotated surface. And therefore the p'_T and p'_L will reflect the contribution due to the rotated sector. These transformations are done everywhere face centre values or surface normal vectors are used.

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} cos(\theta) & sin(\theta) & 0 \\ -sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{4.1}$$

These changes are incorporated in the `fwhFormulation.C` and the `Farassa1AFormulation.C` file. It is also important to note that the above mentioned rotation matrix is applicable only for rotation about the z-axis and therefore the current library works only if the turbo-machine is rotating about the z-axis. In `lines 21-42` it can be seen that in addition to `qds` and `fds` being sized to the number of observers, control surfaces and number of faces per control surface an additional dimension of number of blades has been included.

fwhFormulationModified

```
21
22 void Foam::functionObjects::fwhFormulation::initialize()
23 {
24    //allocate qds_, fds_ and vds_
25     qds_.resize(fwh_.observers_.size());
26     fds_.resize(fwh_.observers_.size());
27     forAll(fwh_.observers_, iObs)
28     {
29       qds_[iObs].resize(fwh_.Blades_);
30       fds_[iObs].resize(fwh_.Blades_);
31       for(int iBl = 0; iBl<fwh_.Blades_; ++iBl)
32       {
33         qds_[iObs][iBl].resize(fwh_.controlSurfaces_.size());
34         fds_[iObs][iBl].resize(fwh_.controlSurfaces_.size());
35
36         forAll(fwh_.controlSurfaces_, iSurf)
37           {
```

```
    qds_[iObs][iBl][iSurf].resize(fwh_.controlSurfaces_[iSurf].Cf().size());
    fds_[iObs][iBl][iSurf].resize(fwh_.controlSurfaces_[iSurf].Cf().size());
     }
   }
  }
```

The additional dimension of number of blades is similarly added to both `robs` and `magrobs` as seen in `lines 64 - 73`. In `lines 70 - 75` the angle by which the control surface has to be rotated is calculated according to the number of blades. As an example, if there are 6 blades, for every iteration of the loop the control surface will be increased by 60 degrees. Once the angle by which the rotation matrix needs to be adjusted is decided, the rotation matrix is computed according to Eq. 4 as seen in `lines 88 - 92`. The new coordinates of the rotated coordinates are now used to calculate `robs/`,i.e, the position of the observer as seen in `line 94`, which is subsequently used to compute the magnitude of the distance between the observer and source as seen in `line 95 - 100`.

fwhFormulationModified

```
64    forAll(fwh_.observers_, iObs)
65    {
66        rMax_[iObs] = 0.0;
67        const SoundObserver& obs = fwh_.observers_[iObs];
68    robs_[iObs].resize(fwh_.Blades_);
69    magrobs_[iObs].resize(fwh_.Blades_);
70    for(int iBl = 0; iBl<fwh_.Blades_; ++iBl)
71    {
72      robs_[iObs][iBl].resize(fwh_.controlSurfaces_.size());
73      magrobs_[iObs][iBl].resize(fwh_.controlSurfaces_.size());
74
75      const double theta = ((360*iBl)/(fwh_.Blades_))*(constant::mathematical::pi)/180;
76
77      forAll(fwh_.controlSurfaces_, iSurf)
78        {
79          robs_[iObs][iBl][iSurf].resize(fwh_.controlSurfaces_[iSurf].Cf().size());
80          magrobs_[iObs][iBl][iSurf].resize(fwh_.controlSurfaces_[iSurf].Cf().size());
81
82          const vectorField& Cf = fwh_.controlSurfaces_[iSurf].Cf();
83          vector q_ = vector::zero;
84          forAll(Cf, i)
85        {
86          // Only rotation about z-axis is accounted
87          // It will give wrong results for rotation about any other axis
88          vector tmpCf = Cf[i];
89          q_.x() = (Foam::cos(theta)*tmpCf.x()) + (Foam::sin(theta)*tmpCf.y());
90          q_.y() = (Foam::cos(theta)*tmpCf.y()) - (Foam::sin(theta)*tmpCf.x());
91          q_.z() = tmpCf.z();
92
93          robs_[iObs][iBl][iSurf][i] = obs.position() - q_;
94          vector r = robs_[iObs][iBl][iSurf][i];
95          scalar R_ = sqrt
96                  (
97             sqr(r[0])
98                  +
99                  ( 1 - sqr(mag(fwh_.U0_)/fwh_.c0_))
100                 *
```

In the `Farassat1AFormulation.c` file, an additional dimension of integration has been added in a mnner very similar to the `fwhFormulation.C` file. No changes are required in the other files.

Farassat1AFormulationModified

```
// * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * //
void Foam::functionObjects::Farassat1AFormulation::initialize()
{
  intFdS_.resize(fwh_.observers_.size());
  intDotQdS_.resize(fwh_.observers_.size());
  L_.resize(fwh_.observers_.size());
  M_.resize(fwh_.observers_.size());
```

```
 Un_.resize(fwh_.observers_.size());

 forAll(fwh_.observers_, iObs)
  {
    L_[iObs].resize(fwh_.Blades_);
  M_[iObs].resize(fwh_.Blades_);
  Un_[iObs].resize(fwh_.Blades_);
   for(int iBl = 0; iBl<fwh_.Blades_; ++iBl)
  {
   L_[iObs][iBl].resize(fwh_.controlSurfaces_.size());
   M_[iObs][iBl].resize(fwh_.controlSurfaces_.size());
   Un_[iObs][iBl].resize(fwh_.controlSurfaces_.size());

   forAll(fwh_.controlSurfaces_, iSurf)
    {
     L_[iObs][iBl][iSurf].resize(fwh_.controlSurfaces_[iSurf].Cf().size());
     M_[iObs][iBl][iSurf].resize(fwh_.controlSurfaces_[iSurf].Cf().size());
     Un_[iObs][iBl][iSurf].resize(fwh_.controlSurfaces_[iSurf].Cf().size());
    }
  }
  }
}
```

Farassat1AFormulationModified

```
  scalar ct1 = ct+fwh_.obr_.time().deltaT().value()*1.0e-6;//slightly increase time to get inside of time step

  scalar retv = 0.0;
  intDotQdS_.value(iObs) = 0.0;
  intFdS_.value(iObs)    = 0.0;
  //calculate acoustic pressure, zero if source has not yet reached observer
  for(int iBl = 0; iBl<fwh_.Blades_; ++iBl)
   {
  forAll(fwh_.controlSurfaces_, iSurf)
   {
    forAll(qds_[iObs][iBl][iSurf], iFace)
     {
    retv = valueAt(qds_, iObs,iBl ,iSurf, iFace, ct1);

    intDotQdS_.value(iObs) += retv;
    retv = valueAt(fds_, iObs, iBl, iSurf, iFace, ct1);
    intFdS_.value(iObs) += retv;
     }
   }
   }

  reduce (intDotQdS_.value(iObs), sumOp<scalar>());
```

The full `Farassat1AFormulationModified.C` and `fwhFormulationModified.C` have been included in the 6.1

Chapter 5

Test Case

This chapter aims to explain the implementation of the modified acoustic library and the setup of a case using the SRFPimpleFoam Solver. SRFPimpleFoam is a transient incompressible solver which allows for simulations of rotating domains. The SRF solver rotates the frame of reference to simulate rotating conditions. No mixing planes are used and only a single rotating domain is simulated.

5.1 CFD Domain

The Mixer tutorial is used as the base case to define the CFD domain and mesh. The tutorial can be found in the below mentioned location:

```
cd $FOAM_TUTORIALS/incompressible/SRFSimpleFoam/mixer/
```

In Figure 5.1,the Blade is the source region and the Inlet, Outlet and OuterWall will be used as the bounding FWH Surface. The mesh is obtained using the `blockMesh` utility. The mesh created is axi-symmetric and uses the cyclic boundary for the periodic faces as shown in Figure 5.2. The domain rotates about the z axis and the inlet is at the origin at 2500rpm. The extent of the domain 0.1 m in all three directions.

5.2 Case Setup Files

The rotor2D case listed under `$FOAM_TUTORIALS\SRFPimpleFoam\rotor2D` will be used as the starting point for setting up the boundary conditions. Only significant changes between the case setups will be listed in this chapter.

SRFSimpleFoam is a steady state solver and the mixer case listed in the tutorials is steady state. However, we need pressure as a function of time and therefore need a transient case. This is achieved by increasing the rotational speed of the mixer to 2500 rpm in the SRF properties file in `constant/` dictionary. The `controlDict`, `fvSchemes`, `fvSolutions` files have been modified to include the effects of solving a transient problem. These files can be found in the Chapter 6.1

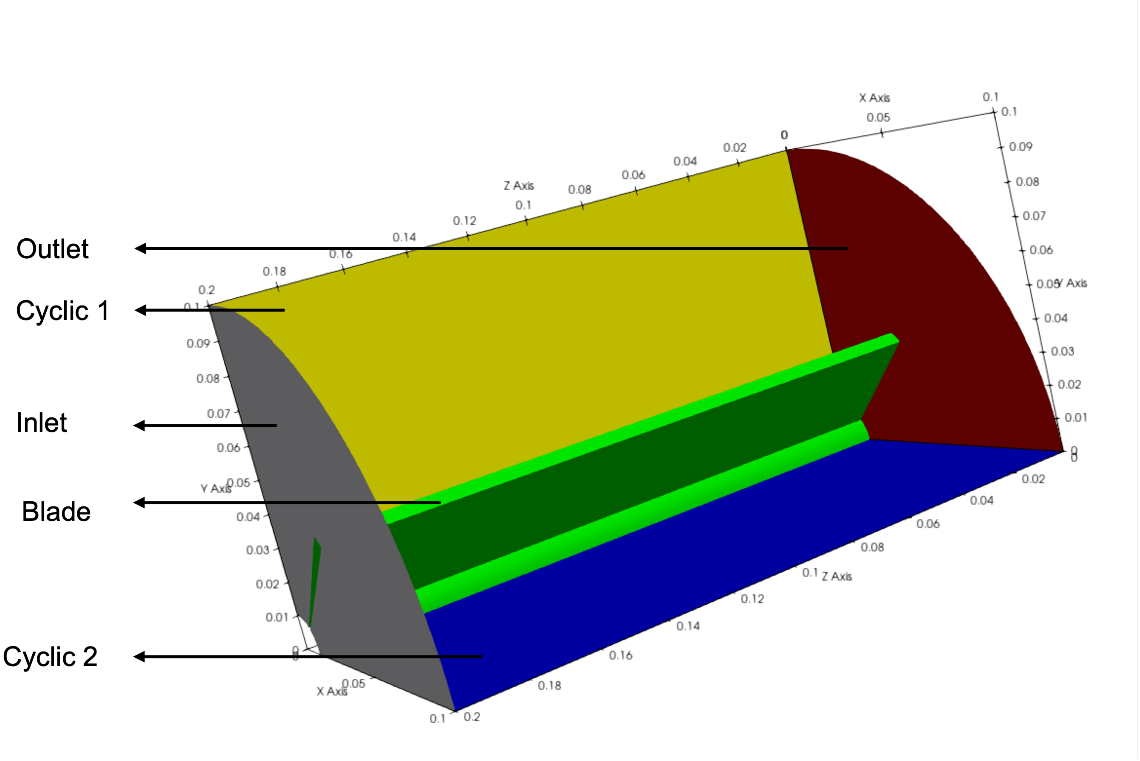

Figure 5.1: CFD Domain and Boundaries

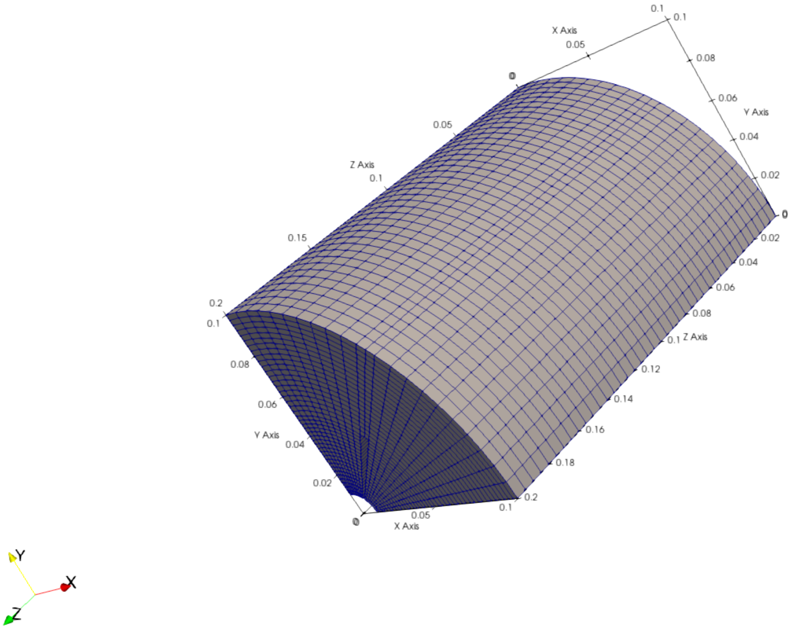

Figure 5.2: Mesh

SRF Properties

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2006                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      SRFProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

SRFModel        rpm;

origin          (0 0 0);
axis            (0 0 1);

rpmCoeffs
{
    rpm         2500;
}


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

In the `fwhControl` dictionary calls for a common settings file. The details will of this file will be discussed later. In the `fwhControl` dictionary, the patches declared refer to the fwh control surfaces. In this case three control surfaces namely, inlet, outlet and the outerWall patches which bound the mixer blade have been selected as the fwh control surfaces. `U0` refers to any background velocity. It is usually set to zero, other than for cases in which there is a constant velocity near the far field observer. `cleanFreq` does not affect the solution in anyway, it just clean out the unnecessary data every 100 iterations in this case. `Blades` refers to the number of sectors required to complete the full annulus. The surfaces sub-dictionary is used to define the aforementioned patches or fwh control surfaces. Any number of patches can be declared.

fwhControl

```
FwhFarassat1A
{
        type                        FfowcsWilliamsHawkings;
            #include                    "commonSettings";
            patches                 ("inlet" "outlet" "outerWall");
            interpolationScheme         cell;
            formulationType             Farassat1AFormulation;
            U0                          (.0 .0 .0);
            cleanFreq                   100;
    Blades              4;
            Ufwh                    (.0 .0 .0);

        surfaces
        (

                inlet
                {
                    type            patch;
                    patches         ("inlet");
                    interpolate     false;
                }
                outlet
                {
```

```
                    type            patch;
                    patches         ("outlet");
                    interpolate     false;
                }
                outerWall
                {
                    type            patch;
                    patches         ("outerWall");
                    interpolate     false;
                }

        );
}
```

The commonSettings file located in the `system` directory, declares the sampling start time (`timeStart`), sampling end time (`timeEnd`), propagation speed for sound waves (`c0`), backgorund velocity at observer (`U0`), reference dimension (`dRef`), to be set at -1 for a 3d simulation and the domain depth for a 2d simulation, density at observer (`rho`), this should be set to `rho` for a compressible simulation and to `rhoInf` for an incompressible simulation. A sub-dictionary including the position of the observers is then included. The position is mentioned in the cartesian coordinates along with the reference pressure, which is used to calculate the SPL experience by the observer and lastly the fourier transform frequency needs to be mentioned

commonSettings

```
    libs ("libAcoustics.so");

    log                 true;

    writeFft            true;


    timeStart           0;

    timeEnd             1;

    c0                  340;

    U0                  (0 0 0);

    dRef                -1;

    pName               p;

    pInf                101325;

    rho                 rhoInf;

    rhoInf              1.2;

    CofR                (0 0 0);

    observers
    {
        Observer-A
        {
            position    (0 5 5);
            pRef        2.0e-5;
            fftFreq     1024;
        }

        Observer-B
        {
            position    (0 2 3);
            pRef        2.0e-5;
            fftFreq     1024;
        }
```

```
    }
```

5.3 Running the Simulation

The acoustic library can be compiled and the simulation can be run at once by running the `Allrun` script in the `run/FWH/` directory using the following command `./Allrun &`.

5.4 Output

The results provided in this section account for three observers, namely Observer A, Observer B and Observer C at locations (0,1,1), (0,2,2) and (0,3,3). Observer A being the closest and Observer C being the furthest from the source. Once the simulation is complete, an additional folder `acousticData` is created within the `FWH` folder. The `acousticData` folder contains 4 files can be found. One file per observer containing the SPL, Frequency and p' amplitude at the following location:

```
run/FWH/acousticData/
```

A small extract of the file for Observer A can be found below. Similar files exist for Observer B and Observer C. The three columns refer to the SPL, Frequency and p' amplitude respectively. Figure 5.3 illustrates the SPL in the frequency domain.

The peaks observed in Figure 5.3 refer to the blade passing frequency (BPF). BPF is the increase in SPL as the blade rotates past the observer. BPF is calculated according to Eq. 5.4. For this case the BPF is at 170Hz, which is where the first peak is observed. It is also observed that the peaks observed for Observer A is greater than for Observer B which is in turn greater than Observer C. This is as expected as Observer A is closest to the source.

$$BPF = (RPM * NumberofBlades/60) \tag{5.1}$$

fft-FwhFarassat1A-Observer-A.dat

```
2431.64 0.154301 77.7468
2436.52 0.150407 77.5248
2441.41 0.144386 77.1699
2446.29 0.137655 76.7552
2451.17 0.132017 76.392
2456.05 0.128959 76.1884
2460.94 0.128892 76.1839
2465.82 0.13092 76.3195
2470.7 0.133319 76.4772
2475.59 0.134384 76.5464
2480.47 0.133013 76.4573
2485.35 0.128974 76.1894
2490.23 0.122944 75.7736
2495.12 0.116354 75.295
2500 0.110977 74.8841
2504.88 0.108234 74.6666
```

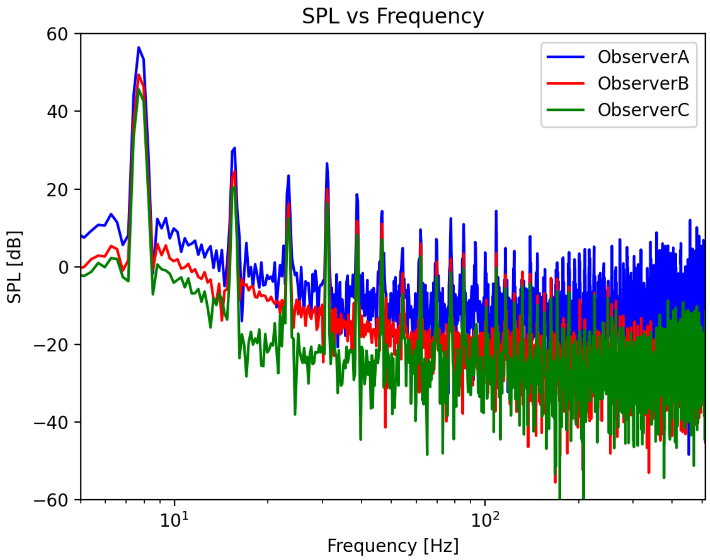

Figure 5.3: Sound Pressure Level vs Frequency

Another file containing the pressure as a function time is created at the below mentioned location.

```
run/FWH/acousticData/$Fwh-Farassat1A-time.dat$
```

A small extract of these files are provided here. The first column corresponds to the time and the remaining columns correspond to pressure fluctuation for each observer respectively. Figure 5.4 plots the pressure fluctuation data as a function of time. Again it is observed here that Observer A experiences higher pressure fluctuations compared to Observer B and Observer C.

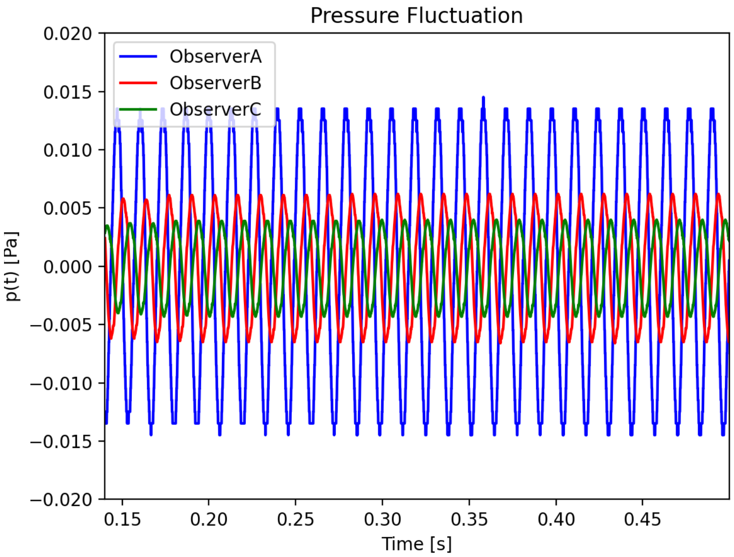

Figure 5.4: Pressure fluctuation as a function of time

Fwh-Farassat1A-time.dat

```
0.05 -258.149 -62.6054 -27.5089
0.0501 -258.15 -62.6052 -27.5087
0.0502 -258.15 -62.605 -27.5086
0.0503 -258.151 -62.6049 -27.5085
0.0504 -258.151 -62.6047 -27.5084
0.0505 -258.151 -62.6046 -27.5083
0.0506 -258.152 -62.6045 -27.5082
0.0507 -258.152 -62.6044 -27.508
0.0508 -258.152 -62.6043 -27.5079
0.0509 -258.152 -62.6042 -27.5079
0.051 -258.152 -62.6041 -27.5079
0.0511 -258.152 -62.6041 -27.5078
0.0512 -258.152 -62.6042 -27.5077
0.0513 -258.152 -62.6042 -27.5075
0.0514 -258.152 -62.6042 -27.5074
0.0515 -258.152 -62.6043 -27.5073
```

Bibliography

[1] L. M. James, “On sound generated aerodynamically i. general theory,” *Proc. R. Soc. Lond. A Math. Phys. Sci*, vol. 211, pp. 564–587, 1952.

[2] N. Curle, “The influence of solid boundaries upon aerodynamic sound,” *Proc. R. Soc. Lond. A Math. Phys. Sci*, vol. 231, pp. 505–514, 1955.

[3] J. Ffowcs Williams and D. Hawkings, “Sound generated by turbulence and surfaces in arbitrary motion,” *Philosophical Transactions of the Royal Society*, vol. A264, no. 1151, pp. 321–342, 1969.

[4] K. S. Brentner and F. Farassat, “An analytical comparison of the acoustic analogy and kirchhoff formulation for moving surfaces,” *AIAA Journal*, vol. 36, no. 8, pp. 1379–1386, 1998.

[5] F. Farassat and M. Myers, “Extension of kirchhoff’s formula to radiation from moving surfaces,” *Journal of Sound and Vibration*, vol. 123, no. 3, pp. 451–461, 1999.

[6] A. Epikhin, I. Evdokimov, M. Kraposhin, M. Kalugin, and S. Strijhak, “Development of a dynamic library for computational aeroacoustics applications using the openfoam open source package,” *Procedia Computer ScienceVolume*, vol. 66, pp. 150–157, 2015.

Study questions

1. How do you adjust the library to accommodate a compressible solution ?
2. How do you adjust the library to accommodate 2d simulations ?
3. How do you adjust the library to accommodate background velocity at the observer ?
4. Which terms are ignored whilst computing the Farassat 1A formulation ?
5. What is the difference between direct and hybrid CAA approaches ?

Chapter 6

The first appendix

6.1 First Section

6.1.1 fwhFormulationModified.C

fwhFormulationModified

```
#include "FfowcsWilliamsHawkings.H"
#include "fwhFormulation.H"

Foam::functionObjects::fwhFormulation::fwhFormulation(const FfowcsWilliamsHawkings& fwh)
:
    fwh_(fwh),
    fwhProbeI_(0),
    qds_(0),
    fds_(0),
    tobs_(0),
    robs_(0),
    magrobs_(0),
    ni_(0),
    nl_(0),
    rMax_(0),
    tauMax_(0),
    tauMin_(0)
{
    this->initialize();
}

void Foam::functionObjects::fwhFormulation::initialize()
{
   //allocate qds_, fds_ and vds_
    qds_.resize(fwh_.observers_.size());
    fds_.resize(fwh_.observers_.size());
    forAll(fwh_.observers_, iObs)
    {
      qds_[iObs].resize(fwh_.Blades_);
      fds_[iObs].resize(fwh_.Blades_);
      for(int iBl = 0; iBl<fwh_.Blades_; ++iBl)
      {
        qds_[iObs][iBl].resize(fwh_.controlSurfaces_.size());
        fds_[iObs][iBl].resize(fwh_.controlSurfaces_.size());

        forAll(fwh_.controlSurfaces_, iSurf)
          {
        qds_[iObs][iBl][iSurf].resize(fwh_.controlSurfaces_[iSurf].Cf().size());
        fds_[iObs][iBl][iSurf].resize(fwh_.controlSurfaces_[iSurf].Cf().size());
          }
      }
    }
```

```
    //allocate tobs
    tobs_.resize(fwh_.observers_.size());
    forAll(fwh_.observers_, iObs)
    {
      tobs_[iObs].resize(fwh_.Blades_);
      for(int iBl = 0; iBl<fwh_.Blades_; ++iBl)
    {
      tobs_[iObs][iBl].resize(fwh_.controlSurfaces_.size());
      forAll(fwh_.controlSurfaces_, iSurf)
        {
          tobs_[iObs][iBl][iSurf].resize(fwh_.controlSurfaces_[iSurf].Cf().size());
        }
    }
    }
    tauMax_.resize(fwh_.observers_.size(), 0.0);
    rMax_.resize(fwh_.observers_.size(), 0.0);

    //allocate robs
    robs_.resize(fwh_.observers_.size());
    magrobs_.resize(fwh_.observers_.size());

    forAll(fwh_.observers_, iObs)
    {
        rMax_[iObs] = 0.0;
        const SoundObserver& obs = fwh_.observers_[iObs];
    robs_[iObs].resize(fwh_.Blades_);
    magrobs_[iObs].resize(fwh_.Blades_);
    for(int iBl = 0; iBl<fwh_.Blades_; ++iBl)
    {
      robs_[iObs][iBl].resize(fwh_.controlSurfaces_.size());
      magrobs_[iObs][iBl].resize(fwh_.controlSurfaces_.size());

      const double theta = ((360*iBl)/(fwh_.Blades_))*(constant::mathematical::pi)/180;

      forAll(fwh_.controlSurfaces_, iSurf)
        {
          robs_[iObs][iBl][iSurf].resize(fwh_.controlSurfaces_[iSurf].Cf().size());
          magrobs_[iObs][iBl][iSurf].resize(fwh_.controlSurfaces_[iSurf].Cf().size());

          const vectorField& Cf = fwh_.controlSurfaces_[iSurf].Cf();
          vector q_ = vector::zero;
          forAll(Cf, i)
        {
          // Only rotation about z-axis is accounted
          // It will give wrong results for rotation about any other axis
          vector tmpCf = Cf[i];
          q_.x() = (Foam::cos(theta)*tmpCf.x()) + (Foam::sin(theta)*tmpCf.y());
          q_.y() = (Foam::cos(theta)*tmpCf.y()) - (Foam::sin(theta)*tmpCf.x());
          q_.z() = tmpCf.z();

          robs_[iObs][iBl][iSurf][i] = obs.position() - q_;
          vector r = robs_[iObs][iBl][iSurf][i];
          scalar R_ = sqrt
                    (
                sqr(r[0])
                    +
                    ( 1 - sqr(mag(fwh_.U0_)/fwh_.c0_))
                    *
                    ( sqr(r[1]) + sqr(r[2]) )
                    );

            magrobs_[iObs][iBl][iSurf][i] =
            (
            -(mag(fwh_.U0_)/fwh_.c0_) * r[0] + R_
            ) / (1 - sqr(mag(fwh_.U0_)/fwh_.c0_));


          if (magrobs_[iObs][iBl][iSurf][i] > rMax_[iObs])
```

```
            {
              rMax_[iObs] = magrobs_[iObs][iBl][iSurf][i];
            }
        }
        }
        }

        reduce(rMax_[iObs], maxOp<scalar>());
        tauMax_[iObs] = rMax_[iObs] / fwh_.c0_;
    }
    //calculate normals
    List<vector> Cs(fwh_.controlSurfaces_.size());
    ni_.resize(fwh_.Blades_);
    nl_.resize(fwh_.Blades_);
    for(int iBl = 0; iBl<fwh_.Blades_; ++iBl)
      {
    const double theta = ((360*iBl)/(fwh_.Blades_))*(constant::mathematical::pi)/180;
    ni_[iBl].resize(fwh_.controlSurfaces_.size());
    nl_[iBl].resize(fwh_.controlSurfaces_.size());

    forAll(fwh_.controlSurfaces_, iSurf)
      {
        const vectorField& Sf = fwh_.controlSurfaces_[iSurf].Sf();
        const vectorField& Cf = fwh_.controlSurfaces_[iSurf].Cf();
        ni_[iBl][iSurf].resize(Cf.size());
        nl_[iBl][iSurf].resize(Cf.size());

        Cs[iSurf] = gSum(Cf);
        scalar surfSize = scalar(Cf.size());
        reduce (surfSize, sumOp<scalar>());
        Cs[iSurf] /= surfSize;

        scalar magSf = 0.0;
        vector q_ = vector::zero;
        forAll(Cf, iFace)
          {
        vector tmpSf = Sf[iFace];
        q_.x() = (Foam::cos(theta)*tmpSf.x()) + (Foam::sin(theta)*tmpSf.y());
        q_.y() = (Foam::cos(theta)*tmpSf.y()) - (Foam::sin(theta)*tmpSf.x());
        q_.z() = tmpSf.z();

        magSf = mag(q_);
        ni_[iBl][iSurf].value(iFace) = q_/magSf;
        if ( ((Cf[iFace] - Cs[iSurf]) & ni_[iBl][iSurf].value(iFace)) > 0 )
          {
            nl_[iBl][iSurf][iFace] = 1.0;
          }
        else
          {
            nl_[iBl][iSurf][iFace] = -1.0;
          }
        ni_[iBl][iSurf].value(iFace) *= nl_[iBl][iSurf][iFace];
          }
      }
      }
}

Foam::functionObjects::fwhFormulation::~fwhFormulation()
{
}

Foam::scalar Foam::functionObjects::fwhFormulation::observerAcousticPressure(label iObs)
{
    return 0.0;
}

void Foam::functionObjects::fwhFormulation::clearExpiredData()
{
```

```
    scalar ct   = fwh_.obr_.time().value();// - fwh_.obr_.time().deltaT().value()*1.0e-6;
    reduce(ct, minOp<scalar>());

    fwhProbeI_++;
    if ( mag(fwhProbeI_ % fwh_.cleanFreq_) > VSMALL  )
    {

    }
    else
    {
        fwhProbeI_ = 0;
        scalar expiredTime = 0.0;
        label  expiredIndex= -1;
        label  newsize     = 0;
        forAll(qds_, iObs)
        {
            for(int iBl = 0; iBl<fwh_.Blades_; ++iBl)
            {
                forAll(qds_[iObs][iBl], iSurf)
                {
          forAll(qds_[iObs][iBl][iSurf], iFace)
            {
              if (tauMin_.size())
            {
              expiredTime = ct - (tauMax_[iObs] - tauMin_[iObs]);
            }
              else
            {
              expiredTime = ct - tauMax_[iObs];
            }
              const pointTimeData& qdsOldPointData = qds_[iObs][iBl][iSurf][iFace];
              expiredIndex= findExpiredIndex(qdsOldPointData, expiredTime);

                    // -1 - if nothing found, from 0 to (size-1) for indices to remove
              if (expiredIndex > -1)
            {
              newsize = qdsOldPointData.first().size() - (expiredIndex + 1);

                        //clean qds
              pointTimeData newPointData;

              newPointData.first().resize(newsize);
              newPointData.second().resize(newsize);
              for(label iTime=expiredIndex+1; iTime<qdsOldPointData.first().size(); iTime++)
                {
                  newPointData.first() [iTime-(expiredIndex+1)] = qdsOldPointData.first()[iTime];
                  newPointData.second()[iTime-(expiredIndex+1)] = qdsOldPointData.second()[iTime];
                }
              qds_[iObs][iBl][iSurf][iFace].first().operator=(newPointData.first());
              qds_[iObs][iBl][iSurf][iFace].second().operator=(newPointData.second());

              //clean fds
              const pointTimeData& fdsOldPointData = fds_[iObs][iBl][iSurf][iFace];
              for(label iTime=expiredIndex+1; iTime<fdsOldPointData.first().size(); iTime++)
                {
                  newPointData.first()[iTime-(expiredIndex+1)] = fdsOldPointData.first()[iTime];
                  newPointData.second()[iTime-(expiredIndex+1)] = fdsOldPointData.second()[iTime];
                }

              fds_[iObs][iBl][iSurf][iFace].first().operator=(newPointData.first());
              fds_[iObs][iBl][iSurf][iFace].second().operator=(newPointData.second());
            }
            }
        }
        }
    }
    }
}
```

```

void Foam::functionObjects::fwhFormulation::update()
{
    scalar ct   = fwh_.obr_.time().value();

    if (mag(fwh_.Ufwh_) > SMALL )
    {
        forAll(fwh_.observers_, iObs)
        {
            rMax_[iObs] = 0.0;
            tauMax_[iObs] = 0.0;
            if (rMin_.size())
            {
                rMin_[iObs] = GREAT;
            }
       for(int iBl = 0; iBl<fwh_.Blades_; ++iBl)
          {
        const double theta = ((360*iBl)/(fwh_.Blades_))*(constant::mathematical::pi)/180;
        forAll(fwh_.controlSurfaces_, iSurf)
          {
            const vectorField& Cf = fwh_.controlSurfaces_[iSurf].Cf();
            vector q_ = vector::zero;
            forAll(Cf, i)
              {
            vector tmpCf = Cf[i];
            q_.x() = (Foam::cos(theta)*tmpCf.x()) + (Foam::sin(theta)*tmpCf.y());
            q_.y() = (Foam::cos(theta)*tmpCf.y()) - (Foam::sin(theta)*tmpCf.x());
            q_.z() = tmpCf.z();

            robs_[iObs][iBl][iSurf][i] = fwh_.observers_[iObs].position() - q_;
            magrobs_[iObs][iBl][iSurf][i] = mag(robs_[iObs][iBl][iSurf][i]);
            if (magrobs_[iObs][iBl][iSurf][i] > rMax_[iObs])
              {
                rMax_[iObs] = magrobs_[iObs][iBl][iSurf][i];
              }

            if (rMin_.size() && (magrobs_[iObs][iBl][iSurf][i] < rMin_[iObs]))
              {
                rMin_[iObs] = magrobs_[iObs][iBl][iSurf][i];
              }
              }
          }
          }
            reduce(rMax_[iObs], maxOp<scalar>());
            tauMax_[iObs] = rMax_[iObs] / fwh_.c0_;

            if (tauMin_.size())
            {
                reduce(rMin_[iObs], minOp<scalar>());
                tauMin_[iObs] = rMin_[iObs] / fwh_.c0_;
            }
        }
        for(int iBl = 0; iBl<fwh_.Blades_; ++iBl)
      {
```

6.1.2 controlDict

controlDict

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2006                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      controlDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

application     SRFPimpleFoam;

startFrom       startTime;

startTime       0;

stopAt          endTime;

endTime         0.3;

deltaT          1e-04;

writeControl    timeStep;

writeInterval   100;

purgeWrite      0;

writeFormat     ascii;

writePrecision  6;

writeCompression off;

timeFormat      general;

timePrecision   6;

runTimeModifiable true;

adjustTimeStep  no;

maxCo           1;

functions
{
        #include "fwhControl"
}
// ************************************************************************* //
```

6.1.3 fvSchemes

fvSchemes

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2006                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSchemes;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

ddtSchemes
{
    default         Euler;
}

gradSchemes
{
    default         Gauss linear;
    limited         cellLimited Gauss linear 1;
}

divSchemes
{
    default         none;
    div(phi,Urel)   Gauss limitedLinearV 1;
    div(phi,k)      Gauss limitedLinear 1;
    div(phi,epsilon) Gauss limitedLinear 1;
    div((nuEff*dev2(T(grad(Urel))))) Gauss linear;
}

laplacianSchemes
{
    default         Gauss linear corrected;
}

interpolationSchemes
{
    default         linear;
}

snGradSchemes
{
    default         corrected;
}

wallDist
{
    method meshWave;
}


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

6.1.4 fvSolution

fvSolution

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2006                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSolution;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

solvers
{
    p
    {
        solver          GAMG;
        tolerance       1e-08;
        relTol          0.05;
        smoother        GaussSeidel;
        nCellsInCoarsestLevel 20;
    }

    pFinal
    {
        $p;
        relTol          0;
    }

    "Urel.*"
    {
        solver          smoothSolver;
        smoother        GaussSeidel;
        nSweeps         2;
        tolerance       1e-07;
        relTol          0.1;
    }

    "k.*"
    {
        solver          smoothSolver;
        smoother        GaussSeidel;
        nSweeps         2;
        tolerance       1e-07;
        relTol          0.1;
    }

    "epsilon.*"
    {
        solver          smoothSolver;
        smoother        GaussSeidel;
        nSweeps         2;
        tolerance       1e-07;
        relTol          0.1;
    }
}

PIMPLE
{
```

```
    nOuterCorrectors 1;
    nCorrectors     2;
    nNonOrthogonalCorrectors 0;
    pRefCell        0;
    pRefValue       0;
}

relaxationFactors
{
    equations
    {
        "Urel.*"        1;
        "k.*"           1;
        "epsilon.*"     1;
    }
}

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