Cite as: Aihara, A.: Implementation of library for acoustic sound pressure and spanwise correction. In Proceedings of CFD with OpenSource Software, 2018, Edited by Nilsson. H.,
http://dx.doi.org/10.17196/OS_CFD#YEAR_2018

CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Implementation of library for acoustic sound pressure and spanwise correction

Developed for OpenFOAM-3.0.x
Requires: AcousticAnalogy library

Author:
Aya Aihara
Uppsala University
aya.aihara@angstrom.uu.se

Peer reviewed by:
Mikko Folkersma
Mohammad Arabnejad

December 22, 2018

Learning outcomes

The reader will learn:

How to use it:

- How to use the acousticAnalogy library

The theory of it:

- The theory of the Curle's acoustic analogy and the spanwise correction

How it is implemented:

- The implementation of the AcousticAnalogyCorr library

How to modify it:

- How to modify the AcousticAnalogy library for the spanwise correction

Prerequisites

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

- Fundementals of acoustics
- It is recommended to have a look at paper [1] for understanding the method for spanwise correction

Contents

1 Theory **5**
1.1 Curle's acoustic analogy 5
1.2 Spanwise correction 6

2 AcousticAnalogy library **8**
2.1 Curle class 8
2.2 SoundObserver class 11

3 Implementation of sound pressure correction **13**
3.1 Modifications in CurleCorr.H 14
3.2 Modifications in CurleCorr.C 15
3.3 Modifications in soundObserver.H 21
3.4 Modifications in soundObserver.C 21

4 Test case **23**
4.1 Case description 23
4.2 Results 24

Preface

In this tutorial, the AcousticAnalogy library is introduced to calculate the sound pressure generated from a bluff body based on the acoustic wave equation. This library is developed by M. Heinrich and uploaded on the course website [2]. The library predicts the acoustic sound using Curle's analogy method.

When long-span bodies such as cylinder or airfoil are studied for their noise emission, it can be computationally expensive to simulate the large spatial domain which covers the whole section of the body. This tutorial extends the AcousticAnalogy library so that the sound pressure generated from the entire body surface can be obtained using the pressure field data of the computed domain based on the spanwise correction method.

Chapter 1

Theory

This chapter explains briefly the Curle's acoustic analogy, which the AcousticAnalogy library is based on and the method to correct sound pressure for the long-span body. The following section shows the expression of sound pressure p', which is obtained from the pressure and velocity fields. These flow field data are computed by the CFD solver. Since no interaction between the flow field and the sound field is assumed here, the calculation of sound is independent on the solution of the CFD simulations and thus is post-processing.

1.1 Curle's acoustic analogy

Here the fluid is assumed homogenous at rest. In order to study acoustics, we will express the pressure or the density as $p(\boldsymbol{x},t) = p_0 + p'(\boldsymbol{x},t)$, $\rho(\boldsymbol{x},t) = \rho_0 + \rho'(\boldsymbol{x},t)$, which are the summation of disturbance p', ρ' from the equilibrium state and constant values p_0, ρ_0 at rest. The wave equation for p' is derived from the equations for conservation of mass and momentum. It is expressed as

$$\frac{1}{c_0}\frac{\partial^2 p'}{\partial t^2} - \nabla^2 p' = 0 \tag{1.1}$$

where c_0 is the sound speed at rest. The propagation of the acoustic sound p' can be described by the solution for the wave equation, which can be generally obtained by applying the Gauss theorem.

The acoustic analogies, which are derived based on the wave equation, are used to predict noise in engineering applications. These analogies take different formations depending on the assumptions for derivation. The Curle's equation is one of the acoustic analogies and takes into consideration the influence of static solid boundaries upon the sound field, i.e., the Curle's analogy can be applied for the cases where a static object is placed in a fluid. It represents the disturbance of the density $\rho'(\boldsymbol{x},t)$ with integrals of the total volume V external to the solid boundaries and the surface S of the boundaries as

$$\rho'(\boldsymbol{x},t) = \frac{1}{4\pi c_0^2}\frac{\partial^2}{\partial x_i \partial x_j}\int_V \frac{T_{ij}}{r} dV(\boldsymbol{y}) - \frac{1}{4\pi c_0^2}\frac{\partial}{\partial x_i}\int_S \frac{n_j}{r}(p\delta_{ij} - \tau_{ij}) dS(\boldsymbol{y}) \tag{1.2}$$

where $r = |\mathbf{x} - \mathbf{y}|$ is the distance between the observer $\mathbf{x}$ and the sound source $\mathbf{y}$, n_j is the ourward surface normal from the fluid, T_{ij} is the Lighthill's stress tensor, which is $\rho v_i v_j + p_{ij} - c_0^2 \rho \delta_{ij}$, and τ_{ij} is $\rho v_i v_j$. The detailed derivation is described in the reference [3].

Larsson *et al.* [4] rewrites Equation (1.2) based on the formations by Brentner and Farassat [5]. The spatial derivative is converted to a temporal one and the $\frac{\partial r}{\partial x_i}$ term becomes

$$\frac{\partial r}{\partial x_i} = \frac{\partial\sqrt{(x_j - y_j)^2}}{\partial x_i} = \frac{x_i - y_i}{r} = l_i \tag{1.3}$$

where l_i is a unit vector pointing from the source location to the observer. Equation (1.2) is modified on a form where the derivatives are taken inside the integral, and the sound pressure $p'(\boldsymbol{x}, t)$ is expressed as

$$p'(\boldsymbol{x}, t) = \frac{1}{4\pi} \int_V \Big(\frac{l_i l_j}{c_0^2 r} \ddot{T}_{ij} + \frac{3 l_i l_j - \delta_{ij}}{c_0 r^2} \dot{T}_{ij} + \frac{3 l_i l_j - \delta_{ij}}{r^3} T_{ij} \Big) dV(\boldsymbol{y}) + \frac{1}{4\pi} \int_S l_i n_j \Big(\frac{\dot{p}\delta_{ij} - \dot{\tau}_{ij}}{c_0 r} + \frac{p\delta_{ij} - \tau_{ij}}{r^2} \Big) dS(\boldsymbol{y}). \quad (1.4)$$

Equation (1.4) takes the same formation as written in the code.

1.2 Spanwise correction

There are some methods which predict the total sound pressure of long-span bodies, e.g. cylinder, airfoil, plate, so on, based on the pressure radiated from a part of the span section. Their approach can be applied to extrapolate the sound pressure outside the computational domain. Here, an approach by Kato *et al.* [1] is introduced, which models the frequency characteristics of pressure to consider the phase shift in the spanwise direction. The procedure for correction is shown in Figure 1.1. The total span length of the body is L, and the length of which part intersects in the computational domain is L_s.

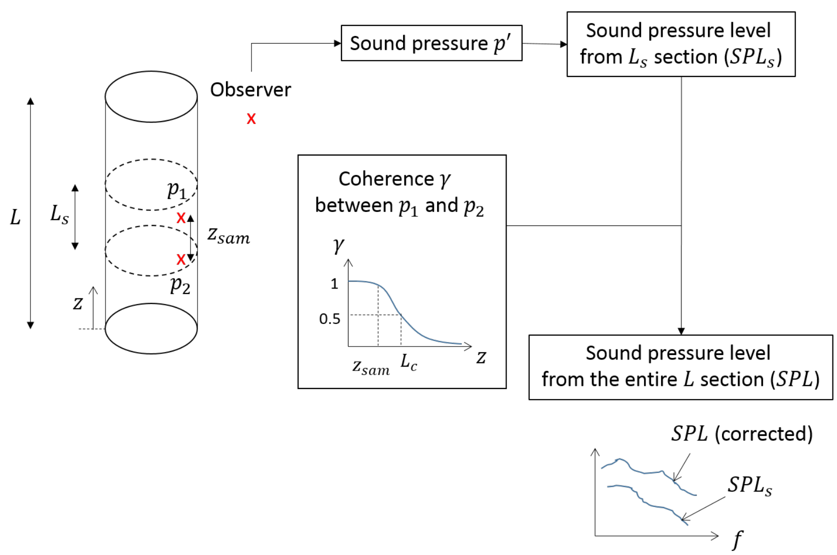

Figure 1.1: Calculation procedure for spanwise correction

The sound pressure level (SPL) is a logarithmic scale of the sound pressure expressed by $20\log_{10}(p'/p_{ref})$ dB (decibel). p_{ref} is a reference pressure that is typically the threshold of human hearing, 2×10^{-5} Pa. The SPL in the frequency domain corrected by the Kato's method, $20\log_{10}(p'_{corr}/p_{ref})$, is described as follows

$$SPL(f) \equiv \begin{cases} SPL_s(f) + 20\log(L/L_s) & (L \leq L_c(f)) \quad (1.5a) \\ SPL_s(f) + 20\log\{L_c(f)/L_s\} + 10\log\{L/L_c(f)\} & (L_s \leq L_c(f) \leq L) \quad (1.5b) \\ SPL_s(f) + 10\log(L/L_s) & (L_c(f) \leq L_s) \quad (1.5c) \end{cases}$$

where SPL_s is the value directly calculated from the source in the computed region as shown in the figure. The coherence function $\gamma(f, z)$ can be defined from coherence between surface pressure at two points apart by distance z in the spanwise direction, which are p_1 and p_2 depicted in the figure. The spanwise coherence length $L_c(f)$ is the distance z when $\gamma(f, z)$ is 0.5.

The above equations are rewritten so that the sound pressure p'_{corr} can be simply expressed using a correction coefficient r_{corr} as $p'_{corr} = r_{corr}p'$ where

$$r_{corr}(f) \equiv \begin{cases} L/L_s & (L \leq L_c(f)) & \text{(1.6a)} \\ \sqrt{LL_c}/L_s & (L_s \leq L_c(f) \leq L) & \text{(1.6b)} \\ \sqrt{L/L_s} & (L_c(f) \leq L_s) & \text{(1.6c)} \end{cases}$$

The coherence function $\gamma(f, z)$ needs to be calculated in order to find L_c at each frequency. Given signals at two locations $z = x$, y, the coherence function is generally represented as the ratio of the cross power spectral density, $W_{xy}(f)$, to the power spectral densities, $W_{xx}(f)$ and $W_{yy}(f)$.

$$\gamma(f) = \frac{|W_{xy}(f)|^2}{W_{xx}(f) \cdot W_{yy}(f)} \tag{1.7}$$

The two signals correspond to p_1 and p_2 in our case. Since $\gamma(f, z)$ is also the function of z, it is theoretically necessary to have the surface pressure at all points along the z direction in order to determine $\gamma(f, z)$ for each z. However, instead of sampling pressure at all points, $\gamma(f, z)$ is approximated in the code according to the idea by Siddon [6] for simplification. He noted that the correlation of the surface pressure can be modeled by the Gaussian function.

$$\gamma(f, z) = \exp\left(- \frac{z^2}{2l(f)^2} \right) \tag{1.8}$$

l is a constant but is dependent on frequency. For a certain frequency f, the value of $\gamma(f, z_{samp})$ can be obtained from the sampled surface pressure p_1 and p_2 which are apart by distance z_{samp}. Then l is determined, and $\gamma(f, z)$ is found as a function z. $L_c(f)$ is the value of $z = z'$ which satisfies $\gamma(f, z') = 0.5$. By doing so, the code samples the surface pressure p_1 and p_2 at only two points.

Chapter 2

AcousticAnalogy library

This chapter describes how the function object, the `AcousticAnalogy` library, which was developed by M. Heinrich [2], calculates the sound pressure based on the Curle's acoustic analogy. This library is implemented as functionObject for OpenFOAM 3.0.x, and it is intended for solving incompressible flow. A user gives the patch names of the surface, the density at rest, the sound speed, and the observer positions as inputs. Time histories of the sound pressure received at each observer are written to a file created under `postProcessing` directory.

The top-level directory `acousticFunctionObject` consists of the following files.

```
acousticFunctionObject
├── Curle
│   ├── Curle.H
│   ├── Curle.C
│   ├── CurleFunctionObject.H
│   └── CurleFunctionObject.C
├── Make
│   ├── files
│   └── options
├── soundObserver.H
└── soundObserver.C
```

The `Curle.C` file is the main source file, which describes the definition of the `Curle` class to mainly calculate the sound pressure p'. The `soundObserver.C` file defines the `SoundObserver` class, which is called inside the `Curle` class and stores both the positions of the observers and the received sound pressure.

2.1 Curle class

Some important member functions in the `Curle` class will be explained in this section. One of the member data `observers_` in the `Curle` class is declared as

```
    List<SoundObserver> observers_;
```

which is a list of the `SoundObserver` class type. As explained in the next section, the `SoundObserver` class stores data for the position of the observer and the sound pressure. Each element in the list holds the information for each observer.

The `read` member function reads the input entries, such as the sound speed, the density of fluid, information of the observers, and so on, that a user specifies in the case directory. This function also stores the observer's names and positions in `observers_`.

The `calculate` member function calculates the sound pressure p' received at each observer, which consists of the volume and the surface integrals as expressed in Equation (1.4). The `calculate` function calculates the term for the volume integral as follows.

```
428     SoundObserver& obs = observers_[obsI];
429     scalar pPrime = 0.0;
430
431     // Volume integral
432     if (cellZoneID_ != -1)
433     {
434         // List of cells in cellZoneID
435         const labelList& cells = mesh.cellZones()[cellZoneID_];
436
437         // Cell volume and cell center
438         const scalarField& V = mesh.V();
439         const vectorField& C = mesh.C();
440
441         // Loop over all cells
442         forAll(cells, i)
443         {
444             label cellI = cells[i];
445
446             // Distance to observer
447             scalar r = mag(obs.position() - C[cellI]);
448             vector l = (obs.position() - C[cellI]) / r;
449
450             // Calculate pressure fluctuation
451             pPrime += coeff * V[cellI] *
452                     (
453                         ((l*l) && d2Tijdt2[cellI]) / (cRef_ * cRef_ * r)
454                       + ((3.0 * l*l - I) && dTijdt[cellI]) / (cRef_ * r * r)
455                       + ((3.0 * l*l - I) && Tij[cellI]) / (r * r * r)
456                     );
457         }
458         reduce(pPrime, sumOp<scalar>());
459     }
```

The positions of observers are read from the list `observers_` in line 428. `pPrime` corresponds to the sound pressure p'. `coeff` is a constant, $1/4\pi$. If the equation for `pPrime` is compared with Equation (1.4), `l` is the vector $l_{i,j}$, `cRef_` is the sound speed c_0, `r` is the distance r. `Tij`, `dTijdt`, and `d2Tijdt2` are other member functions of the `Curle` class which return the Lighthill tensor T_{ij} and its first and second time derivatives, $\dot{T_{ij}}$ and $\ddot{T_{ij}}$ respectively. `Tij` is simply calculated by $\rho_0 UU^T$, and the function for `Tij` is defined as follows.

```
Foam::tmp<Foam::volTensorField> Foam::Curle::Tij() const
{
    const volVectorField& U = obr_.lookupObject<volVectorField>(UName_);

    return
    (
        rhoRef_*(U*U)
    );
}
```

`dTijdt` and `d2Tijdt2` are obtained based on the second-order backward differencing time derivative and the first-order Euler second time derivative methods, respectively. Thus the functions for `dTijdt` and `d2Tijdt2` load the velocity fields of the current and last two time steps.

The term of the surface integral is written as follows.

```
// Surface integral - loop over all patches
forAllConstIter(labelHashSet, patches_, iter)
{
    // Get patch ID
    label patchI = iter.key();

    // Surface area vector and face center at patch
    vectorField Sf = mesh.Sf().boundaryField()[patchI];
    vectorField Cf = mesh.Cf().boundaryField()[patchI];

    // Normal vector pointing towards fluid
    vectorField n = -Sf/mag(Sf);

    // Pressure field and time derivative at patch
    scalarField pp = p.boundaryField()[patchI];
    scalarField dpdtp = dpdt.boundaryField()[patchI];

    // Lighthill tensor on patch
    tensorField Tijp = Tij.boundaryField()[patchI];
    tensorField dTijdtp = dTijdt.boundaryField()[patchI];

    // Distance surface-observer
    scalarField r = mag(obs.position() - Cf);
    vectorField l = (obs.position() - Cf) / r;

    // Calculate pressure fluctuations
    pPrime += coeff * gSum
    (
        (
            (l*n)
          &&
            (
                (dpdtp*I - dTijdtp) / (cRef_*r)
              + (pp*I - Tijp) / sqr(r)
            )
        )
      * mag(Sf)
    );
}
obs.pPrime(pPrime);
```

`n` is the surface normal n_j and `mag(Sf)` is the surface area. `p` and `dpdtp` are the member functions which return the pressure and its time derivative. The functions for `p` and `dpdtp` are defined as follows.

```
Foam::tmp<Foam::volScalarField> Foam::Curle::p() const
{
    return
    (
        rhoRef_ * obr_.lookupObject<volScalarField>(pName_)
    );
```

```
}

Foam::tmp<Foam::volScalarField> Foam::Curle::dpdt() const
{
    return
    (
        rhoRef_ * Foam::fvc::ddt(obr_.lookupObject<volScalarField>(pName_))
    );
}
```

The `fvc::ddt` class returns information about the time scheme, thus `dpdt` is derived based on the time scheme specified in `fvSchemes` of the case directory. After both the volume and surface integrals are obtained, the total sound pressure `pPrime` is stored in the `obj` object in line 500.

The `writeCurle` member function writes the sound pressure each time step to both the log file and a file placed in the `postProcessing` directory as follows.

```
// File output
file(0) << obr_.time().value() << tab << setw(1) << "   ";
forAll(observers_, obsI)
{
    file(0)
        << observers_[obsI].pPrime() << "   ";
}
file(0) << endl;
```

2.2 SoundObserver class

As shown above, the main `Curle` class calculates the sound pressure received at each observer. Each sound pressure is stored in each element of the list of the `SoundObserver` class type. The `SoundObserver` class does nothing for calculations but is needed to store data for the position of the observer and the sound pressure.

The member data in the `SoundObserver` class are as follows.

```
//- Name of the sound observer
word name_;

//- Position of the sound observer
vector position_;

//- Pressure fluctuation [Pa]
scalar pPrime_;
```

The member functions are as follows.

```
//- Return name
const word& name() const
{
    return name_;
}

//- Return position of observer
const vector& position() const
{
    return position_;
}
```

```

    //- Return fluctuation pressure
    const scalar& pPrime() const
    {
        return pPrime_;
    }

    //- Set fluctuating pressure
    void pPrime(scalar pPrime);
```

A user has to give the names and positions for each observer, which are stored in `name_` and `positions_`, respectively. The sound pressure, which is p' in Equation (1.4), is stored in `pPrime_`.

Chapter 3

Implementation of sound pressure correction

The procedure to implement the spanwise correction for sound pressure is explained here. The original acousticAnalogy library calculates and writes out the sound pressure of each time step. We will modify the code so that it also calculates both the spectrum of the sound pressure (SPL_s) and the corrected spectrum (SPL) during run time. To obtain the corrected spectrum, the coherence function $\gamma(f, z)$ first needs to be found using the pressure sampled on the body surface. Then the correction coefficient $r_{corr}(f)$ is determined from $\gamma(f, z)$. The corrected spectrum SPL is calculated by multiplying $r_{corr}(f)$ to the original spectrum SPL_s.

Asummed that the `acousticFunctionObject` directory, which is uploaded on website [2], is placed under `$WM_PROJECT_USER_DIR/src`, we first go to `$WM_PROJECT_USER_DIR/src` and prepare a new directory `CurleCorr` for modification by copying and renaming files.

```
mkdir CurleCorr
cp -r acousticFunctionObject/* CurleCorr/
cd CurleCorr
mv Curle/Curle.H Curle/CurleCorr.H
mv Curle/Curle.C Curle/CurleCorr.C
mv Curle/CurleFunctionObject.H Curle/CurleCorrFunctionObject.H
mv Curle/CurleFunctionObject.C Curle/CurleCorrFunctionObject.C
```

The word `Curle` is replaced by `CurleCorr` in all files.

```
sed -i s/Curle/CurleCorr/g Curle/*
```

We rename the library as AcousticAnalogyCorr, so it should be written in `Make/files` as

```
Curle/CurleCorr.C
Curle/CurleCorrFunctionObject.C

soundObserver.C

LIB = $(FOAM_USER_LIBBIN)/libAcousticAnalogyCorr
```

and in `Make/options` as

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/fileFormats/lnInclude \
```

```
    -I$(LIB_SRC)/sampling/lnInclude \
    -I$(LIB_SRC)/randomProcesses/lnInclude

LIB_LIBS = \
    -lspecie \
    -lfiniteVolume \
    -lmeshTools \
    -lfileFormats \
    -lsampling \
    -lrandomProcesses
```

After all modifications in the following sections are completed, the code can be compiled using the command `wmake`. The new library named `libAcousticAnalogyCorr.so` will be created under `$FOAM_USER_LIBBIN`. The reader can also refer to the supplied final codes when implementing this library.

3.1 Modifications in CurleCorr.H

Two header files should be included.

```
#include "probes.H"
#include "complexFields.H"
```

The `probes` class is needed to sample the surface pressure which is then used to calculate the coherence function $\gamma(f, z)$. The `complexFields.H` file needs to be included to use the complex numbers for the Fourier analysis. To inherit the `probes` class, the top of the `CurleCorr` class declaration should be as follows.

```
class CurleCorr
:
    public functionObjectFile,
    public probes
```

The protected member data from the `probes` class which are used in this library should be added.

```
const fvMesh& mesh_;
bool loadFromFiles_;
wordReList fieldSelection_;
bool fixedLocations_;
word interpolationScheme_;
```

And the additional member data should be added as well.

```
scalar L_;
scalar Ls_;
label freqSample_;
label Nstart_;
label Naverage_;
label countStep_;
label countFFT_;
List<List<scalar>> pList_;
scalar distance_;
word fileDir_;
scalarField Coh_;
List<List<scalar>> CofftObs_;
```

L_ and Ls_ are L and L_s in Equation (1.5), respectively. For example, if freqSample_ = 1024 and Nstart_ = 3, the first $1024 \cdot 2^{3-1}$ steps are discarded. The spectra are calculated when the number of stored data reaches $1024 \cdot 2^{i+3}$ (i = 0, 1, ...). Every time the spectra are calculated, they are written out in a new file. Naverage_ is the number for averaging the power spectra, and then the averaged power spectra are used to obtain the coherence function $\gamma(f, z)$. distance_ is the distance between the locations of sampled pressure, z_{samp}. Coh_ is the value of $\gamma(f, z_{samp})$. CofftObs_ is the corrected sound pressure, p_{corr}.

The additional public member functions are declared as

```
virtual void storeSampledPressure();
virtual void calculateSpectrum();
virtual void calculateCoherence();
virtual void calculateCorrection();
virtual complexField calcFFT(const scalarList&);
```

The storeSampledPressure function stores the surface pressure p_1 and p_2 sampled by the probes class. The calculateSpectrum function calculates the spectrum of the sound pressure p' and write it to the file. The calculateCoherence function finds Coh_, i.e., the value of $\gamma(f, z_{samp})$ for each frequency f using the p_1 and p_2 data. The calculateCorrection function determines the coherence function $\gamma(f, z)$ and r_{corr} to calculate the spectrum of the corrected sound pressure p'_{corr}, which is also written to the file. The calcFFT function performs the Fourier transform. The detail of each function will be explained in the next section.

3.2 Modifications in CurleCorr.C

The following line should be included in the top.

```
#include "fft.H"
```

This header file is needed for the FFT analysis used in the calcFFT function.

The following lines should be added before the last line initialised_ = true; in the initialise function.

```
countFFT_ += Nstart_;

if ( pow(2,Nstart_) < Naverage_ )
{
    FatalErrorIn("void Foam::Curle::initialise()")
        << "Nstart is too small or Naverage is too large"
        << exit(FatalError);
}

Info << "First "<< freqSample_*pow(2,(Nstart_-1))
        << " steps will be discarded for fft." << endl;

distance_ = mag(operator[](0) -operator[](1));
Info << "Probed distance = " << distance_ << endl;

pList_.resize(2);
CofftObs_.resize(observers_.size());
```

The above lines give an error message if a user doesn't give a proper value of Nstart_ or Naverage_. distance_ is calculated here and printed out in the log file. pList_ and CofftObs_ are initialized to match their size to two and the number of the observers, respectively.

In the constructor, some lines are necessary for initialization

```
probes(name, obr, dict, loadFromFiles),
mesh_(refCast<const fvMesh>(obr)),
loadFromFiles_(loadFromFiles),
fieldSelection_(),
fixedLocations_(true),
interpolationScheme_("cell"),
L_(0),
Ls_(0),
freqSample_(1),
Nstart_(0),
Naverage_(0),
countStep_(1),
countFFT_(0),
pList_(0),
distance_(0),
fileDir_(word::null),
Coh_(0),
CofftObs_(0)
```

and one line after `read(dict);` in the `if (readFields)` statement as well.

```
probes::read(dict);
```

As for the `CurleCorr::read` function, the word `"patches"` in the `if (active_)` statement should be replaced by `"patchName"`. And the following lines should be added also in the `if (active_)` statement.

```
L_ = readScalar(dict.lookup("L"));
Ls_ = readScalar(dict.lookup("Ls"));
freqSample_ = readLabel(dict.lookup("freqSample"));
Nstart_ = readLabel(dict.lookup("Nstart"));
Naverage_ = readLabel(dict.lookup("Naverage"));
```

At the end of the `CurleCorr::read` function, the following lines are needed.

```
fileName fileSubDir = name_;

if (mesh_.name() != polyMesh::defaultRegion)
{
    fileSubDir = fileSubDir/mesh_.name();
}
fileSubDir = "postProcessing"/fileSubDir/mesh_.time().timeName();

if (Pstream::parRun())
{
    fileDir_ = mesh_.time().path()/".."/fileSubDir;
}
else
{
    fileDir_ = mesh_.time().path()/fileSubDir;
}
```

At the end of the definition of the `CurleCorr::calculate` function, one line should be inserted in the `forAll(observers_, obsI)` statement.

```
obs.storepPrime(pPrime);
```

In the `CurleCorr::write` function, the following lines should be added after the line `calculate();`.

```
probes::write();

storeSampledPressure();

if ( countStep_ == freqSample_*(pow(2,countFFT_)+pow(2,Nstart_-1)) )
{
    calculateSpectrum();
    calculateCoherence();
    calculateCorrection();
    countFFT_ += 1;
}

countStep_ += 1;
```

The first line, `probes::write();`, is not necessary if the sampled pressure does not need to be written out. The `storeSampledPressure()` function is executed all time steps. The functions, `calculateSpectrum()`, `calculateCoherence()`, and `calculateCorrection()`, are executed every time it solves the specified number of time steps, more specifically, $\texttt{freqSample\_} \cdot 2^{i+\texttt{Nstart\_}} (i = 0, 1, ...)$.

The definition of the `CurleCorr::storeSampledPressure` function should be as below.

```
void Foam::CurleCorr::storeSampledPressure()
{
    const volScalarField& p = obr_.lookupObject<volScalarField>(pName_);
    const scalarField p_sample = probes::sample( p );

    forAll(p_sample,i)
    {
        pList_[i].append(p_sample[i]);
    }
}
```

This function stores the surface pressure at two locations p_1 and p_2 that a user specifies using the function object `probes`.

The definition of the `CurleCorr::calculateSpectrum` function should be as below.

```
void Foam::CurleCorr::calculateSpectrum()
{
    Info  <<"Calculating spectrum" << endl;

    mkDir(fileDir_/mesh_.time().timeName());
    OFstream* fPtr1 = new OFstream(fileDir_/mesh_.time().timeName()/"pPrimeFFT");
    OFstream& fout1 = *fPtr1;

    fout1 << "# Frequency ";
    forAll( observers_, i)
    {
        fout1 << "pPrimeFFT_at_"<< observers_[i].name() << " ";
    }
```

```
        fout1 << endl;

        scalar deltaT = mesh_.time().deltaT().value();
        scalar N = countStep_ -freqSample_*pow(2,(Nstart_-1));

        scalarField freq(N);
        forAll( freq, i )
        {
            freq[i] = i/(deltaT*N);
        }

        forAll( observers_, i)
        {
            SubList<scalar> subpPrimeList( observers_[i].pPrimeAll(), N,
                                          freqSample_*pow(2,(Nstart_-1)));
            scalarField Cofft_obs_i = mag(calcFFT( subpPrimeList ));
            CofftObs_[i] = Cofft_obs_i;
        }

        forAll( freq, freqi)
        {
            fout1 << freq[freqi] << " ";
            forAll( observers_, i)
            {
                fout1 << CofftObs_[i][freqi] << " ";
            }
            fout1 << endl;
      }
    }
```

The original `Curle` class calculates the sound pressure in the `calculate()` function. Then this function applies the FFT to obtain the spectrum, stores both frequency and its magnitude in `CofftObs_`, and write it in a new file.

The definition of the `CurleCorr::calculateCoherence` function should be as below.

```
    void Foam::CurleCorr::calculateCoherence()
    {
        Info  <<"Calculating coherence" << endl;

        scalar N2 = ( countStep_ -freqSample_*pow(2,(Nstart_-1)) )/Naverage_;

        List<complexField> Wxx(Naverage_);
        List<complexField> Wyy(Naverage_);
        List<complexField> Wxy(Naverage_);

        forAll( Wxy, i )
        {
            List<complexField> pListFFTtemp;
            forAll( pList_, ii )
            {
                SubList<scalar> subpList( pList_[ii], N2, freqSample_*pow(2,(Nstart_-1))+N2*i);
                scalarList pListWin(N2);

                forAll( subpList, freqi )
```

```
            {
                scalar hanningi = 0.5*(1 -cos(constant::mathematical::twoPi*freqi/N2));
                pListWin[freqi] = subpList[freqi] *hanningi;
            }
            pListFFTtemp.append(calcFFT( pListWin ));
        }

        complexField Wxyi(N2);
        complexField Wxxi(N2);
        complexField Wyyi(N2);

        forAll( pListFFTtemp[0], freqi )
        {
            Wxyi[freqi] = pListFFTtemp[0][freqi].conjugate() *pListFFTtemp[1][freqi];
            Wxxi[freqi] = pListFFTtemp[0][freqi].conjugate() *pListFFTtemp[0][freqi];
            Wyyi[freqi] = pListFFTtemp[1][freqi].conjugate() *pListFFTtemp[1][freqi];
        }

        Wxy[i] = Wxyi;
        Wxx[i] = Wxxi;
        Wyy[i] = Wyyi;
    }

    Coh_.resize(N2);

    forAll( Wxy[0], freqi )
    {
        complex WWxy;
        complex WWxx;
        complex WWyy;

        forAll( Wxy, i )
        {
            WWxy += Wxy[i][freqi]/Naverage_;
            WWxx += Wxx[i][freqi]/Naverage_;
            WWyy += Wyy[i][freqi]/Naverage_;
        }

        Coh_[freqi] = magSqr(WWxy)/( mag(WWxx)*mag(WWyy) );
    }
}
```

This function first applies the FFT to each `Naverage_` segment of the sampled pressure with the Hann window. Then after averaging the power spectra of all segments, the coherence $\gamma(f, z_{samp})$ in Equation (1.7) is obtained and stored in `Coh_`.

The definition of the `CurleCorr::calculateCorrection` function should be as below.

```
void Foam::CurleCorr::calculateCorrection()
{
    Info << "Calculating correction " << endl;

    scalar deltaT = mesh_.time().deltaT().value();
    scalar N2 = ( countStep_ -freqSample_*pow(2,(Nstart_-1)) )/Naverage_;
```

```
scalarField freq(N2);
forAll( freq, i )
{
    freq[i] = i/(deltaT*N2);
}

scalar l2;
scalar Lc;
scalarField rCorr(N2);

forAll( freq, freqi )
{
    if ( Coh_[freqi] < 0.5 )
    {
        rCorr[freqi] = sqrt(L_/Ls_);
    }
    else if ( 0.5 <= Coh_[freqi] && Coh_[freqi] <= 0.999999 )
    {
        l2 = -0.5*sqr(distance_)/log(Coh_[freqi]);
        Lc = sqrt( -2 *l2 *log(0.5) );
        rCorr[freqi] = sqrt(L_*Lc)/Ls_;
        if ( Lc > L_ )
        {
            rCorr[freqi] = L_/Ls_;
        }
    }
    else
    {
        rCorr[freqi] = L_/Ls_;
    }
}

List<List<scalar>> CofftObsCorr(observers_.size());

forAll( observers_, i)
{
    CofftObsCorr[i].resize(N2);

    forAll( freq, freqi )
    {
        scalar fft0 = CofftObs_[i][freqi*Naverage_];
        CofftObsCorr[i][freqi] = fft0*rCorr[freqi];
    }
}

mkDir(fileDir_/mesh_.time().timeName());
OFstream* fPtr2 = new OFstream(fileDir_/mesh_.time().timeName()/"pPrimeFFT_corr");
OFstream& fout2 = *fPtr2;

fout2 << "# Frequency ";
forAll( observers_, i)
{
    fout2 << "pPrimeFFTcorrected_at_" << observers_[i].name() << " ";
}
```

```
        fout2 << endl;

        forAll( freq, freqi)
        {
            fout2 << freq[freqi] << " ";
            forAll( observers_, i)
            {
                fout2 << CofftObsCorr[i][freqi] << " ";
            }
            fout2 << endl;
        }
    }
```

This function calculates the correction coefficient r_{corr}, which is then used to determine the corrected spectrum of the sound pressure. The correction coefficient represented as `rCorr` in the code is found for each frequency as explained in Equation (1.6) based on the coherence. The corrected spectrum is obtained by multiplying `rCorr` to `CofftObs_` and it is printed out in the new file.

The definition of the `CurleCorr::calcFFT` function should be as below.

```
    Foam::complexField Foam::CurleCorr::calcFFT
    (
        const scalarList& tfield
    )
    {
        complexField tfftField = ReComplexField(tfield);
        labelList fftList ( 1, tfield.size() );
        complexField Cofft = fft::reverseTransform( tfftField, fftList );
        Cofft *= 2.0/pow(tfield.size(),0.5);
        Cofft[0] /= 2.0;
        Cofft.last() /= 2.0;
        return Cofft;
    }
```

This function uses the `fft` class, which needs an input of the complex field. `ReComplexField` creates a list of the complex values. The `calcFFT` function returns the result scaled by the size of input data.

3.3 Modifications in soundObserver.H

One private member data should be added

```
    List<scalar> pPrimeAll_;
```

and two public member functions as well.

```
    const List<scalar>& pPrimeAll() const
    {
        return pPrimeAll_;
    }

    void storepPrime(scalar pPrime);
```

3.4 Modifications in soundObserver.C

In the constructor, the following line should be added after `pPrime_(0.0),`.

```
pPrimeAll_(0)
```

The definition of the `SoundObserver::storepPrime` function should be as below.

```
void Foam::SoundObserver::storepPrime(scalar pPrime)
{
    pPrimeAll_.append(pPrime);
}
```

Chapter 4

Test case

This section represents an example case where the AcousticAnalogyCorr library is applied. In this test case, a circular cylinder is placed in the flow field and the sound is observed at some distance away from the cylinder. The cylinder has longer span length than the height of the computational domain. The library will calculate the spectrum of the sound pressure, p, which is generated from the span section of the computational domain. The spectrum of the corrected sound pressure, p_{corr}, generated from the entire cylinder will also be obtained.

4.1 Case description

Figure 4.1 shows the setup where the span length in the computational domain is L_s (= 0.05 m) and the total span length of the cylinder is L (= 0.5 m). The pressure is sampled at two locations on the cylinder surface, p_1 and p_2. The inlet velocity is 70.2 m/s and the cylinder diameter is 19.0 mm.

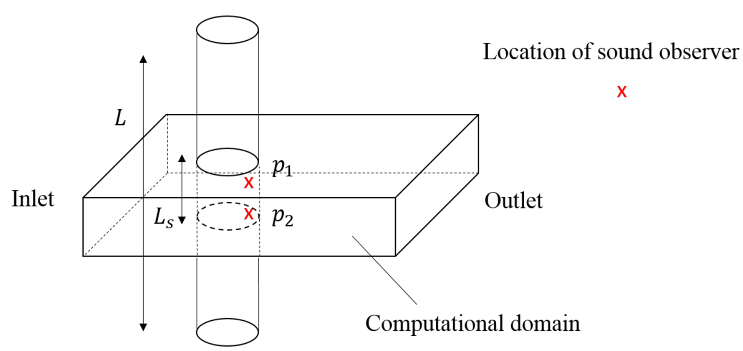

Figure 4.1: Setup of test case

The input entries in `functions` in the `controlDict` file should includes as follows.

```
CurleCorr
{
    functionObjectLibs  ( "libAcousticAnalogyCorr.so" );
    type                CurleCorr;
    outputControl       timeStep;
    outputInterval      1;
    fields              ( p );
    patchName           ( cylinder );
    fixedLocations      true;
    probeLocations
```

```
        (
            (0.0095057 0 -0.02)
            (0.0095057 0 0.02)
        );
        log                 true;
        rhoRef              1.204;
        cRef                343;
        observers
        {
            micro1 { position  (0 -2.4335 0); }
            micro2 { position  (-2.4335 0 0); }
            micro3 { position  (-2.4335 -2.4335 0); }
        }
        L                   0.5;
        Ls                  0.05;
        freqSample          1024;
        Nstart              3;
        Naverage            4;
    }
```

Two locations where the surface pressure is sampled should be specified in `probeLocations`. `cRef` is the sound speed and `rhoRef` is the density of the medium. The name and the location for sound observers should be given in `observers`. `L` and `L_s` are L and L_s, respectively. `freqSample`, `Nstart`, and `Naverage` correspond the variables mentioned early in Section 3.1. `freqSample` and `Naverage` must be a number of powers of two.

Note that the code in this library assume a constant time step, which means that `adjustableRunTime` in `controlDict` should be switched off. To make it simple, the surface pressure at only two points are chosen to sample for calculation of the coherence function. The distance between their two points should not be too close for accurate correction.

4.2 Results

Figure 4.2 shows the sound pressure spectra observed at 2.4 m away from the center of the cylinder. The red line represents the spectrum of p_{corr} obtained based on the correction coefficient r_{corr} expressed in Equation (1.6). It can be seen that the magnitude of p_{corr} are larger than that of p by correction.

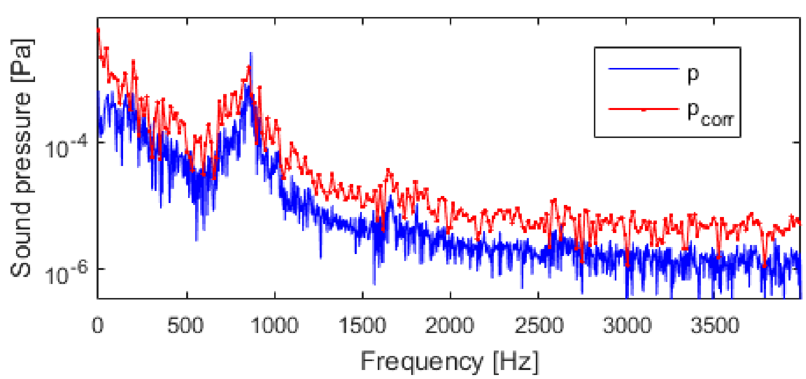

Figure 4.2: Sound pressure spectrum

Bibliography

[1] Chisachi Kato, Akiyoshi Iida, Yasushi Takano, Hajime Fujita, and Masahiro Ikegawa. Numerical prediction of aerodynamic noise radiated from low mach number turbulent wake. In *31st Aerospace Sciences Meeting*, page 145, 1993.

[2] Martin Heinrich. https://github.com/Kiiree/curleAnalogy.

[3] N Curle. The influence of solid boundaries upon aerodynamic sound. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences*, 231(1187):505–514, 1955.

[4] Johan Larsson, Lars Davidson, Magnus Olsson, and Lars-Erik Eriksson. Aeroacoustic investigation of an open cavity at low mach number. *AIAA journal*, 42(12):2462–2473, 2004.

[5] Kenneth S Brentner and Feridoun Farassat. Modeling aerodynamically generated sound of helicopter rotors. *Progress in Aerospace Sciences*, 39(2-3):83–120, 2003.

[6] Thomas E. Siddon. Surface dipole strength by crossâcorrelation method. *The Journal of the Acoustical Society of America*, 53(2):619–633, 1973.

Study questions

1. Why does the AcousticAnalogy library have the `SoundObserver` class besides the main `Curle` class?

2. What is the purpose of implementing the AcousticAnalogyCorr library?

3. What is the purpose of inheriting the `probes` class in the AcousticAnalogyCorr library?

4. If L/L_s is for example 20, what is the maximum and minimum differences of the SPL in decibel between p and p_{corr} according to the correction method implemented in the library?

5. The AcousticAnalogyCorr library creates files for the spectrum of p and p_{corr} including each of the frequency table. How many times is the difference of the frequency resolution between them?