CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Description of matrix discretization with focus on the Gauss laplacian discretization operator and how to create a modified version

Developed for OpenFOAM-1906

Author:
Jesper Roland Kjærgaard Qwist
Technical University of Denmark
jrkp@mek.dtu.dk

Peer reviewed by:
Mohammed Khanouki
Roberto Mosca

January 9, 2020

Learning outcomes

The reader will learn:

How to use it:

- How to specify the Gauss Laplacian discretization scheme in `system/fvSchemes`.

- How to find the available for scheme options for the interpolation of the diffusion coefficient to cell face centres and explicit non orthogonal mesh correction.

The theory of it:

- The theoretical background for the Gauss Laplacian discretization scheme, both for a scalar diffusivity and a tensorial diffusivity.

- The theoretical background for the modified Gauss Laplacian discretization scheme with the Ghost Fluid Method (GFM) [1], which improves the dynamic pressure calculation at the interface between two fluids in an incompressible two-phase flow.

How it is implemented:

- How the standard Gauss Laplacian discretization scheme is implemented for both scalar and tensor diffusion.

- How the base class for Laplacian schemes work.

- How a keyword, used in `system/fvSchemes`, is defined in the code and how this links to the run-time selection mechanism.

How to modify it:

- How to make a copy of the Gauss Laplacian scheme in the user directory.

- How remove the template functionality in the Gauss Laplacian discretization class and limit the class to scalar diffusion and scalar dependent variable.

- How to implement the GFM method in the Gauss Laplacian discretization scheme.

- How a matrix is handled in OpenFOAM.

Prerequisites

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

- How to run standard document tutorials like damBreak tutorial.

- How to find source files with the find command.

- How to customize a solver and do top-level application programming.

- Fundamentals of Computational Methods for Fluid Dynamics, Book by J. H. Ferziger and M. Peric

It is also recommended that the reader knows the following in order to get maximum benefit out of this report:

- How OpenFOAM discretize equations from Ph.D. theses: Error Analysis and Estimation for the Finite Volume Method with Applications to Fluid Flows, 1996, H. Jasak.

- Basic knowledge of two-phase flows from Ph.D. thesis: Computational fluid dynamics of dispersed two-phase flows at high phase fractions, 2002, H. Rusche.

- Basic knowledge of C++.

- Know what macro is.

Contents

Chapter 1

Introduction

I was inspired to make this tutorial after reading about a method, which improves the way Open-FOAM treats the dynamic pressure close to the free surface [1]. The OpenFOAM solver `interIsoFoam` solves an incompressible two-phase flow problem, where the two phases are condensed into a single momentum equation.

The momentum equation for each of the two phases in a single computational cell is condensed into a single momentum equation by introducing the Volume Of Fluid method (VOF). In this method a volume fraction for each cell defines the fraction of the cell volume which is occupied by phase 1, α_1. The volume fraction for phase 2 is defined as $\alpha_2 = 1 - \alpha_1$, such that the total volume fraction is 1. This works very well when we are not at an interface between two phases with a large density difference.

The dynamic pressure [1] is defined as

$$p = p_{total} - \rho \mathbf{g} \bullet \mathbf{x} \qquad (1.1)$$

where ρ is the fluid density, $\mathbf{g}$ is the gravitational vector and $\mathbf{x}$ is the position in space with respect to origin of the coordinate system. It is observed that the definition of the dynamic pressure depends on the density field. The density field in a two-phase flow is by nature discontinuous at the interface between the two phases, as long as the free surface remains well defined. The discontinuity is not captured in `interIsoFoam`, because the dynamic pressure gradient and dynamic pressure laplacian operators performs a continuous linear interpolation of the dynamic pressure field across the free surface. Thereby it indirectly applies a continuous linear interpolation of the density field, since Gauss' theorem is used for the discretization, and here we need to interpolate the dynamic pressure from cell-centres to face-centres. The interpolation inconsistency leads to unphysical behavior at the fluid interface for two-phase flows with a high density ratio as mentioned by Vukcevic et al. [1].

In this tutorial we will only work with the Gauss Laplacian operator used in the dynamic pressure equation, which has been derived from the continuity equation and momentum equation. The final test case is a still water surface.

In this work I need to refer to many different files. I have only specified the names of the files to avoid that the report gets too messy. The full paths can be found by opening a new terminal window and using the command

```
find $FOAM_SRC -name <Filename>
```

where you need to replace `<Filename>` with the name of the relevant file, that you are searching for. All the modified are not part of the source code and should be found in the files related to this report.

Chapter 2

Gauss scheme

This chapter describes the standard Gauss discretization scheme used to discretize the laplacian operator in OpenFOAM.

2.1 How to use it

The specification of schemes is performed in the dictionary file `system/fvSchemes` in all case setups. The laplacian schemes is specified in the sub-dictionary `laplacianSchemes` of `system/fvSchemes`. The sub-dictionary is created by

```
laplacianSchemes
{
    // Specify schemes here
}
```

Now lets say you want to specify the scheme for

$$\nabla \bullet (\gamma \nabla \phi) \tag{2.1}$$

which is part of the governing equation you want to solve. In your OpenFOAM application you have specified the term as

```
fvm::laplacian(gamma,phi)
```

where the diffusion coefficient γ corresponds to `gamma` and the dependent variable ϕ corresponds to `phi`. The required entry in sub-dictionary `laplacianSchemes` is in this case

```
laplacian(gamma,phi)
```

The dictionary now reads

```
laplacianSchemes
{
    laplacian(gamma,phi)
}
```

However we have not defined any discretization yet. We will continue with the scheme options in next section.

2.1.1 Find available scheme options

Now we need to specify discretization scheme. This is quite simple, since the only valid choice is `Gauss`, which tells the code to use Gauss' theorem in the discretization practice. We can figure out what options we have by inserting a scheme name, which does not exist, for example

```
laplacianSchemes
{
    laplacian(gamma,phi) giveMeOptions;
}
```

When executing a case using this scheme, the code will return an error message saying

```
--> FOAM FATAL IO ERROR:
Unknown laplacian scheme giveMeOptions

Valid laplacian schemes are :
1(Gauss)
```

This tells us that there is `1` laplacian scheme and the name is `Gauss`. We replace `giveMeOptions` with `Gauss`, so now we have

```
laplacianSchemes
{
    laplacian(gamma,phi) Gauss;
}
```

We can now try to run our case again, which causes OpenFOAM to return another error message that says

```
--> FOAM FATAL IO ERROR:
discretization scheme not specified

Valid schemes are :

58
(
CoBlended
Gamma
Gamma01
LUST
MUSCL
MUSCL01
Minmod
OSPRE
QUICK
SFCD
SuperBee
UMIST
biLinearFit
blended
cellCoBlended
clippedLinear
cubic
cubicUpwindFit
```

```
deferredCorrection
downwind
filteredLinear
filteredLinear2
filteredLinear3
fixedBlended
harmonic
interfaceCompression
limitWith
limitedCubic
limitedCubic01
limitedGamma
limitedLimitedCubic
limitedLimitedLinear
limitedLinear
limitedLinear01
limitedMUSCL
limitedVanLeer
limiterBlended
linear
linearFit
linearPureUpwindFit
linearUpwind
localBlended
localMax
localMin
midPoint
outletStabilised
pointLinear
quadraticFit
quadraticLinearFit
quadraticLinearUpwindFit
quadraticUpwindFit
reverseLinear
skewCorrected
upwind
vanAlbada
vanLeer
vanLeer01
weighted
)
```

OpenFOAM requires a keyword that describes how to interpolate the coefficient `gamma` from cell-centres to cell-faces. Later when we will look at the implementation, I will explain how to see this from the source code, but for now we will choose the scheme `linear` and continue. Our dictionary now reads

```
laplacianSchemes
{
    laplacian(gamma,phi) Gauss linear;
}
```

When we execute our case again, we get yet another error message that says

```
--> FOAM FATAL IO ERROR:
```

```
discretization scheme not specified

Valid schemes are :
7(corrected faceCorrected limited linearFit orthogonal quadraticFit uncorrected)
```

Again it is not obvious to us what the scheme should be used for. This options tells the code how we want to treat the non-orthogonal correction term $\mathbf{k}\bullet(\nabla\mathbf{U})_f$, which can be seen from Equation (3.17). The vector $\mathbf{k}$ comes from a split of the surface normal area vector in an orthogonal component and a non-orthogonal component. The split can be performed in different ways, one of which is described in Section 2.2.1.

2.2 Theory

The theoretical section presents the theoretical background for the Gauss Laplacian discretization scheme presented so that it fits with the code implementation. The diffusion term from Equation (2.1) is the starting point, and I will consider scalar and tensor diffusion types. The code implementation of the Gauss Laplacian discretization is found in the source files

```
$FOAM_SRC/finiteVolume/finiteVolume/laplacianSchemes/gaussLaplacianScheme
/gaussLaplacianScheme.C
$FOAM_SRC/finiteVolume/finiteVolume/laplacianSchemes/gaussLaplacianScheme
/gaussLaplacianScheme.H
$FOAM_SRC/finiteVolume/finiteVolume/laplacianSchemes/gaussLaplacianScheme
/gaussLaplacianSchemes.C
```

2.2.1 Gauss discretization of Laplacian with scalar diffusion coefficient

The theory behind the laplacian operator of a scalar field ϕ with a scalar diffusion coefficient γ is presented in this section. Gauss's theorem is used to convert a volume integral to a surface integral as given by

$$\int_{V_P} \nabla\bullet(\gamma\nabla\phi)\,\mathrm{d}V = \int_{S_P}\mathrm{d}S\bullet(\gamma\nabla\phi) \tag{2.2}$$

where V_P is the cell volume and S_P is the cell surface.
The divergence operator $\nabla\bullet()$ of a function $f(x,y,z)$ in a Cartesian coordinate system is given by

$$\nabla\bullet(f(x,y,z)) = \frac{\partial f(x,y,z)}{\partial x} + \frac{\partial f(x,y,z)}{\partial y} + \frac{\partial f(x,y,z)}{\partial z} \tag{2.3}$$

The gradient operator $\nabla()$ of a function $f(x,y,z)$ in a Cartesian coordinate system is given by

$$\nabla(f(x,y,z)) = \begin{bmatrix} \frac{\partial f(x,y,z)}{\partial x} \\ \frac{\partial f(x,y,z)}{\partial y} \\ \frac{\partial f(x,y,z)}{\partial z} \end{bmatrix} \tag{2.4}$$

The integral over the cell surface can be evaluated as a summation over the cell faces, which are plane surfaces. The surface integral of each cell face is evaluated using the midpoint rule. This means that we in the sum over all cell faces use the values at the face centroid. This gives

$$\int_{S_P}\mathrm{d}S\bullet(\gamma\nabla\phi) = \sum_{f=1}^{n_f}\mathbf{S}_f\bullet(\gamma\nabla\phi)_f \tag{2.5}$$

where $\mathbf{S}_f$ is the face normal vector with magnitude equal to the face area. The common name is the surface area vector. n_f is the number of faces of the cell. Subscript $_f$ indicates that the variable

should be defined at the cell face centre. The equation can be rearranged so that the interpolated diffusion coefficient γ_f is separated from $\mathbf{S}_f \bullet (\nabla \phi)_f$, since it is just a scalar.

$$\sum_{f=1}^{n_f} \mathbf{S}_f \bullet (\gamma \nabla \phi)_f = \sum_{f=1}^{n_f} \gamma_f \mathbf{S}_f \bullet (\nabla \phi)_f \tag{2.6}$$

The term $\mathbf{S}_f \bullet (\nabla \phi)_f$ can be rewritten to

$$\sum_{f=1}^{n_f} \mathbf{S}_f \bullet (\gamma \nabla \phi)_f = \sum_{f=1}^{n_f} \gamma_f |\mathbf{S}_f| \mathbf{n}_f \bullet (\nabla \phi)_f \tag{2.7}$$

where $\mathbf{n}_f = \frac{\mathbf{S}_f}{|\mathbf{S}_f|}$ is the face normal vector. $|\mathbf{S}_f|$ is the face area. $\mathbf{n}_f \bullet (\nabla \phi)_f$ is the surface normal gradient at the face. In OpenFOAM $\gamma_f |\mathbf{S}_f|$ corresponds to `gammaMagSf` in the file `gaussLaplacianScheme.C`. The surface normal gradient at the face $\mathbf{n}_f \bullet (\nabla \phi)_f$ is treated according to the mesh complexity. Figure 2.1 illustrates three cases with increasing mesh complexity.
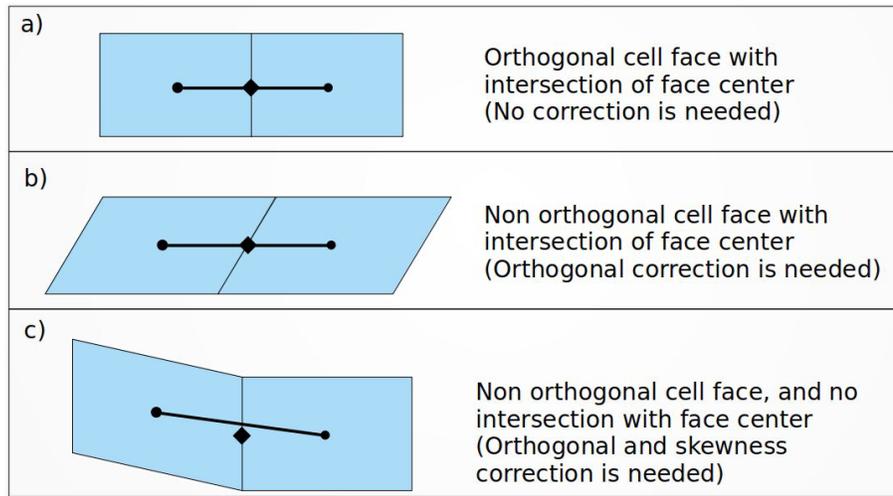
Figure 2.1: Three mesh configurations presented in increasing complexity

Orthogonal meshes

The simplest case is an orthogonal mesh, where the line between owner and neighbour cell centre intersects the face centre, as presented in Figure 2.1. An orthogonal cell only has right angles between the cell faces. Each face is shared by a pair of cells, where the cell with the lowest index is defined as the face owner cell and the other cell the face neighbour cell. In this case the surface normal gradient can be calculated as the difference between the nodal values over the distance

$$\mathbf{n}_f \bullet (\nabla \phi)_f = \underbrace{|\mathbf{n}_f|}_{=1} \frac{\phi_N - \phi_P}{|\mathbf{d}|} \tag{2.8}$$

where the magnitude of the normal $|\mathbf{n}_f| = 1$, and $|\mathbf{d}|$ is the magnitude of the vector from owner cell-centre P to neighbour cell-centre N. Equation (2.7) reduces to

$$\int_{V_P} \nabla \bullet (\gamma \nabla \phi) \, \mathrm{d}V = \sum_{f=1}^{n_f} \gamma_f |\mathbf{S}_f| \frac{\phi_N - \phi_P}{|\mathbf{d}|} \tag{2.9}$$

which is the final discretization of the laplacian term for an orthogonal mesh.

The discretization of the laplacian term can be separated in an uncorrected discretization and a corrected discretization. The uncorrected part of the discretization includes the implicit discretization and contributions from boundary conditions. The corrected part of the discretization includes the explicit discretization, which is used to correct the implicit discretization, for example due to mesh non orthogonolity. The member function `fvmLaplacianUncorrected` takes care of the uncorrected discretization, where the implicit discretization for an orthogonal mesh is given by Equation (2.9). The function declaration is given by

gaussLaplacianScheme.H
```
118        static tmp<fvMatrix<Type>> fvmLaplacianUncorrected
119        (
120            const surfaceScalarField& gammaMagSf,
121            const surfaceScalarField& deltaCoeffs,
122            const GeometricField<Type, fvPatchField, volMesh>&
123        );
```

The function is static, so it applies to all objects, when it is called from the class. The function needs `gammaMagSf` $= \gamma_f |\mathbf{S}_f|$, `deltaCoeffs` $= \frac{|\mathbf{n}_f|}{|\mathbf{d}|} = \frac{1}{|\mathbf{d}|}$ and the dependent variable ϕ, which is a volume field. The implementation is described by the following pieces of code. First

gaussLaplacianScheme.C
```
46   template<class Type, class GType>
47   tmp<fvMatrix<Type>>
48   gaussLaplacianScheme<Type, GType>::fvmLaplacianUncorrected
49   (
50       const surfaceScalarField& gammaMagSf,
51       const surfaceScalarField& deltaCoeffs,
52       const GeometricField<Type, fvPatchField, volMesh>& vf
53   )
54   {
```

tells the code which function we are defining and its input and output. `vf` is short for "Volume Field" and denotes the dependent variable ϕ. Next section

gaussLaplacianScheme.C
```
55        tmp<fvMatrix<Type>> tfvm
56        (
57            new fvMatrix<Type>
58            (
59                vf,
60                deltaCoeffs.dimensions()*gammaMagSf.dimensions()*vf.dimensions()
61            )
62        );
63        fvMatrix<Type>& fvm = tfvm.ref();
```

creates a new matrix in the memory managed by the class `tmp` and a reference to the matrix is saved. Next section is

gaussLaplacianScheme.C
```
65        fvm.upper() = deltaCoeffs.primitiveField()*gammaMagSf.primitiveField();
66        fvm.negSumDiag();
67
```

Line 65 sets all the off-diagonal coefficient above the matrix diagonal, and line 66 sets the diagonal coefficients from the off diagonal coefficients. The details of exactly how those line work according to the code will not be described. In stead we will have a look at how the matrix assembly can be made more transparent and link it to the theory in the section about matrix assembly, which is found in Section 2.4.

The next piece of code treats the boundary faces.

```
                     _____ gaussLaplacianScheme.C _____
68      forAll(vf.boundaryField(), patchi)
69      {
70          const fvPatchField<Type>& pvf = vf.boundaryField()[patchi];
71          const fvsPatchScalarField& pGamma = gammaMagSf.boundaryField()[patchi];
72          const fvsPatchScalarField& pDeltaCoeffs =
73              deltaCoeffs.boundaryField()[patchi];
74
75          if (pvf.coupled())
76          {
77              fvm.internalCoeffs()[patchi] =
78                  pGamma*pvf.gradientInternalCoeffs(pDeltaCoeffs);
79              fvm.boundaryCoeffs()[patchi] =
80                  -pGamma*pvf.gradientBoundaryCoeffs(pDeltaCoeffs);
81          }
82          else
83          {
84              fvm.internalCoeffs()[patchi] = pGamma*pvf.gradientInternalCoeffs();
85              fvm.boundaryCoeffs()[patchi] = -pGamma*pvf.gradientBoundaryCoeffs();
86          }
87      }
88
89      return tfvm;
90  }
```

Line 68-87 loops through the faces at the boundary patches and sets the matrix and source contribution from the boundary conditions, that we have specified. Line 75-81 is used if the patch is coupled, for example a cyclic inlet and outlet condition. Line 83-86 is used for the boundary patches, where we only have cells on one side of the face, hence this is the boundary faces of our computational domain. The matrix assembly which takes place in this function is described in more detail in Section 2.4, where it also related to theory.

Non orthogonal meshes

In Figure 2.1 case b) the mesh is non orthogonal, because the cell face is not orthogonal to the vector $\mathbf{d}$ from P to N, however it still intersect with the face centre, so we still get the value at the face centre, when we interpolate. The surface normal gradient $\mathbf{n}_f \bullet (\nabla \phi)_f$ is separated in an orthogonal part, which is treated implicitly, and a non orthogonal part, which is treated explicitly. To account for the non-orthogonality a separation of the normal vector is performed according to

$$\mathbf{n}_f = \boldsymbol{\Delta} + \mathbf{k} \tag{2.10}$$

which separates the normal vector $\mathbf{n}_f$ in an orthogonal projection $\boldsymbol{\Delta}$ and a non orthogonal projection $\mathbf{k}$. Figure 2.2 illustrates the separation of the normal vector using the over relaxed approach, which is the one used as default in OpenFOAM.
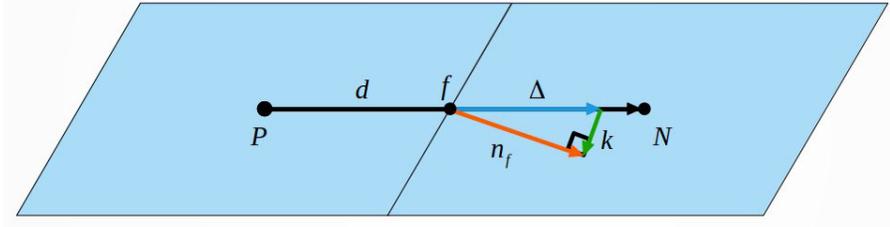
Figure 2.2: Illustration of the over relaxed approach to separate the normal in a face orthogonal and a face non orthogonal component.

The surface normal gradient $\mathbf{n}_f \bullet (\nabla \phi)_f$ is now found by

$$\mathbf{n}_f \bullet (\nabla \phi)_f = \mathbf{\Delta} \bullet (\nabla \phi)_f + \mathbf{k} \bullet (\nabla \phi)_f \tag{2.11}$$

where the implicit orthogonal term $\mathbf{\Delta} \bullet (\nabla \phi)_f$ is given by

$$\mathbf{\Delta} \bullet (\nabla \phi)_f = |\mathbf{\Delta}| \frac{\phi_N - \phi_P}{|\mathbf{d}|} \tag{2.12}$$

and this expands the expression for the surface normal gradient to

$$\mathbf{n}_f \bullet (\nabla \phi)_f = |\mathbf{n}_f| \frac{\phi_N - \phi_P}{|\mathbf{d}|} = \underbrace{|\mathbf{\Delta}| \frac{\phi_N - \phi_P}{|\mathbf{d}|}}_{Orthogonal} + \underbrace{\mathbf{k} \bullet (\nabla \phi)_f}_{Non\ orthogonal} \tag{2.13}$$

where the non orthogonality is accounted for explicitly. The explicit evaluation of the gradient term $(\nabla \phi)_f$ in the non orthogonal term is calculated from a linear interpolation of the gradient

$$(\nabla \phi)_f = f_x (\nabla \phi)_P + (1 - f_x)(\nabla \phi)_N \tag{2.14}$$

where the interpolation factor f_x is defined as

$$f_x = \frac{\mathbf{fN}}{\mathbf{d}} \tag{2.15}$$

The vector $\mathbf{fN}$ is the vector from the face-centre to the neighbour cell-centre. The explicit evaluation of the cell-centred gradients $(\nabla \phi)_P$ and $(\nabla \phi)_N$ is calculated with Gauss' theorem

$$(\nabla \phi)_P = \frac{1}{V_P} \sum_{f=1}^{n_f} \mathbf{S}_f \phi_f \tag{2.16}$$

$$(\nabla \phi)_N = \frac{1}{V_N} \sum_{f=1}^{n_f} \mathbf{S}_f \phi_f \tag{2.17}$$

where V_P is the volume of cell P and V_N is the volume of cell N. The interpolation of ϕ is given by

$$\phi_f = f_x \phi_P + (1 - f_x)\phi_N \tag{2.18}$$

The discretization of the laplacian operator now reads

$$\int_{V_P} \nabla \bullet (\gamma \nabla \phi)\ \mathrm{d}V = \sum_{f=1}^{n_f} \gamma_f |\mathbf{S}_f| \left(|\mathbf{\Delta}| \frac{\phi_N - \phi_P}{|\mathbf{d}|} + \mathbf{k} \bullet (\nabla \phi)_f \right) \tag{2.19}$$

The calculation of $\mathbf{\Delta}$ and $\mathbf{k}$ can be performed in various ways as seen from H. Jasak's PhD thesis [2, p. 84-86]. The implementation in OpenFOAM is slightly different from the presentation given by H. Jasak [2, p. 84-86], since OpenFOAM separates the normal vector into an orthogonal and an

non orthogonal vectorial projection, instead of separating the surface area vector.

In OpenFOAM it has been decided that the vectors $\mathbf{n}_f$ and $\mathbf{k}$ should be orthogonal to each other. This way of determining the vectorial projections is called the **Over-relaxed approach** according to H. Jasak [2, p. 85]. However remember that we use the surface normal vector instead of the surface area vector. $\boldsymbol{\Delta}$ is given by

$$\boldsymbol{\Delta} = \frac{\mathbf{d}}{\mathbf{d} \bullet \mathbf{n_f}} |\mathbf{n}_f|^2 \tag{2.20}$$

which is exactly the same expression presented by H. Jasak [2, (3.32)] after substituting $\mathbf{S}$ with $\mathbf{n}_f$, and remembering that $|\mathbf{n}_f|^2 = 1$, so the expression for the orthogonal projection reduces to

$$\boldsymbol{\Delta} = \frac{\mathbf{d}}{\mathbf{d} \bullet \mathbf{n_f}} \tag{2.21}$$

The implementation in OpenFOAM further splits the vector $\boldsymbol{\Delta}$ in

$$\boldsymbol{\Delta} = \underbrace{\mathbf{d}}_{\texttt{delta}} \underbrace{\frac{1}{\mathbf{d} \bullet \mathbf{n_f}}}_{\texttt{nonOrthDeltaCoeffs}} \tag{2.22}$$

and applies a stabilising condition for the product $\mathbf{d} \bullet \mathbf{n_f}$ to avoid zero division in the fraction. We finally arrive at the implemented expression given by

$$\boldsymbol{\Delta} = \underbrace{\mathbf{d}}_{\texttt{delta}} \underbrace{\frac{1}{max\left(\mathbf{d} \bullet \mathbf{n_f}, 0.05|\mathbf{d}|\right)}}_{\texttt{nonOrthDeltaCoeffs}} \tag{2.23}$$

The implicit as well as the explicit discretization of the laplacian term is handled by the function `fvmLaplacian` in the file `gaussLaplacianSchemes.C` line 37-86. The function `fvmLaplacian` calls the function `fvmLaplacianUncorrected`, which performs the implicit discretization, at line 52-57. The code reads

```
―――――――――――――――― gaussLaplacianSchemes.C ――――――――――――――――
52    tmp<fvMatrix<Type>> tfvm = fvmLaplacianUncorrected            \
53    (                                                            \
54        gammaMagSf,                                              \
55        this->tsnGradScheme_().deltaCoeffs(vf),                  \
56        vf                                                       \
57    );
```

The orthogonal vector $\boldsymbol{\Delta}$ enters the implicit discretization through the second function input `this->tsnGradScheme_().deltaCoeffs(vf)`, which corresponds to $\frac{\boldsymbol{\Delta}}{\mathbf{d}}$. I will now explain how I figured this out from inspecting the code implementation.

At line 55 the code calls the function `deltaCoeffs(vf)` from the private data member `tsnGradScheme_`, which is a surface normal gradient scheme according to the base class declaration `laplacianScheme.H`, where line 82 reads

```
tmp<snGradScheme<Type>> tsnGradScheme_;
```

The class `snGradScheme` is the abstract base class for surface normal gradient schemes, and the function `deltaCoeffs(vf)` is defined as pure virtual function in the `snGradScheme` class declaration, as seen by

```
―――――――――――――――――――――― snGradScheme.H ――――――――――――――――――――――
152        //- Return the interpolation weighting factors for the given field
153        virtual tmp<surfaceScalarField> deltaCoeffs
154        (
155            const GeometricField<Type, fvPatchField, volMesh>&
156        ) const = 0;
```

Hence, all surface normal gradient schemes derived from this base class must define this function. To find an actual implementation of the function `deltaCoeffs(vf)`, we need to look at one of the surface normal gradient schemes derived from the base class `snGradScheme`. The available surface normal gradient schemes are found in source code in the folder

`$FOAM_SRC/finiteVolume/finiteVolume/snGradSchemes`

The surface normal gradient scheme called `correctedSnGrad` is used when the keyword `corrected` is used as the last keyword when defining a laplacian scheme in `system/fvSchemes`. In the class declaration of `correctedSnGrad`, the declaration and definition of the function `deltaCoeffs(vf)` is found in the same file and is given by

```
————————————————— correctedSnGrad.H —————————————————
98          //- Return the interpolation weighting factors for the given field
99          virtual tmp<surfaceScalarField> deltaCoeffs
100         (
101             const GeometricField<Type, fvPatchField, volMesh>&
102         ) const
103         {
104             return this->mesh().nonOrthDeltaCoeffs();
105         }
```

The function returns `this->mesh().nonOrthDeltaCoeffs();`, which is a call to the function `nonOrthDeltaCoeffs()` available from the constant reference to the mesh object `mesh()`. The function `nonOrthDeltaCoeffs()` is in fact declared and defined in the class `surfaceInterpolation`, that the mesh class `fvMesh` inherits from, and therefore the mesh has access to this function. The function `nonOrthDeltaCoeffs()` is defined in `surfaceInterpolation.C` by the code

```
————————————————— surfaceInterpolation.C —————————————————
100  const Foam::surfaceScalarField&
101  Foam::surfaceInterpolation::nonOrthDeltaCoeffs() const
102  {
103      if (!nonOrthDeltaCoeffs_)
104      {
105          makeNonOrthDeltaCoeffs();
106      }
107
108      return (*nonOrthDeltaCoeffs_);
109  }
```

This function ensures that `nonOrthDeltaCoeffs_` is only evaluated, if it is not defined, and it returns a pointer to the private data member `nonOrthDeltaCoeffs_`. If the data member `nonOrthDeltaCoeffs_` is not defined , it is calculated by the function `makeNonOrthDeltaCoeffs()`, which is also defined in `surfaceInterpolation.C`. The code reads

```
————————————————— surfaceInterpolation.C —————————————————
254  void Foam::surfaceInterpolation::makeNonOrthDeltaCoeffs() const
255  {
256      if (debug)
257      {
258          Pout<< "surfaceInterpolation::makeNonOrthDeltaCoeffs() : "
259              << "Constructing differencing factors array for face gradient"
260              << endl;
261      }
262
263      // Force the construction of the weighting factors
264      // needed to make sure deltaCoeffs are calculated for parallel runs.
265      weights();
```

```
266
267        nonOrthDeltaCoeffs_ = new surfaceScalarField
268        (
269            IOobject
270            (
271                "nonOrthDeltaCoeffs",
272                mesh_.pointsInstance(),
273                mesh_,
274                IOobject::NO_READ,
275                IOobject::NO_WRITE,
276                false // Do not register
277            ),
278            mesh_,
279            dimless/dimLength
280        );
281        surfaceScalarField& nonOrthDeltaCoeffs = *nonOrthDeltaCoeffs_;
282        nonOrthDeltaCoeffs.setOriented();
283
284
285        // Set local references to mesh data
286        const volVectorField& C = mesh_.C();
287        const labelUList& owner = mesh_.owner();
288        const labelUList& neighbour = mesh_.neighbour();
289        const surfaceVectorField& Sf = mesh_.Sf();
290        const surfaceScalarField& magSf = mesh_.magSf();
291
292        forAll(owner, facei)
293        {
294            vector delta = C[neighbour[facei]] - C[owner[facei]];
295            vector unitArea = Sf[facei]/magSf[facei];
296
297            // Standard cell-centre distance form
298            //NonOrthDeltaCoeffs[facei] = (unitArea & delta)/magSqr(delta);
299
300            // Slightly under-relaxed form
301            //NonOrthDeltaCoeffs[facei] = 1.0/mag(delta);
302
303            // More under-relaxed form
304            //NonOrthDeltaCoeffs[facei] = 1.0/(mag(unitArea & delta) + VSMALL);
305
306            // Stabilised form for bad meshes
307            nonOrthDeltaCoeffs[facei] = 1.0/max(unitArea & delta, 0.05*mag(delta));
308        }
309
310        surfaceScalarField::Boundary& nonOrthDeltaCoeffsBf =
311            nonOrthDeltaCoeffs.boundaryFieldRef();
312
313        forAll(nonOrthDeltaCoeffsBf, patchi)
314        {
315            fvsPatchScalarField& patchDeltaCoeffs = nonOrthDeltaCoeffsBf[patchi];
316
317            const fvPatch& p = patchDeltaCoeffs.patch();
318
319            const vectorField patchDeltas(mesh_.boundary()[patchi].delta());
```

16

```
320
321            forAll(p, patchFacei)
322            {
323                vector unitArea =
324                    Sf.boundaryField()[patchi][patchFacei]
325                    /magSf.boundaryField()[patchi][patchFacei];
326
327                const vector& delta = patchDeltas[patchFacei];
328
329                patchDeltaCoeffs[patchFacei] =
330                    1.0/max(unitArea & delta, 0.05*mag(delta));
331            }
332        }
333    }
```

The term `nonOrthDeltaCoeffs` from Equation (2.23) is implemented at line 307 above, where `unitArea` is $\mathbf{n}_f$ and `delta` is $\mathbf{d}$. The discretization from Equation (2.19) can be rewritten to

$$\sum_{f=1}^{n_f} \gamma_f |\mathbf{S}_f| \left(|\boldsymbol{\Delta}| \frac{\phi_N - \phi_P}{|\mathbf{d}|} + \mathbf{k} \bullet (\nabla\phi)_f \right) =$$

$$\sum_{f=1}^{n_f} \gamma_f |\mathbf{S}_f| \left(\left| \frac{\mathbf{d}}{max\left(\mathbf{d} \bullet \mathbf{n_f}, 0.05|\mathbf{d}|\right)} \right| \frac{\phi_N - \phi_P}{|\mathbf{d}|} + \mathbf{k} \bullet (\nabla\phi)_f \right) =$$

$$\sum_{f=1}^{n_f} \gamma_f |\mathbf{S}_f| \left(\frac{|\mathbf{d}|}{|\mathbf{d}|} \left| \frac{1}{max\left(\mathbf{d} \bullet \mathbf{n_f}, 0.05|\mathbf{d}|\right)} \right| (\phi_N - \phi_P) + \mathbf{k} \bullet (\nabla\phi)_f \right) =$$

$$\sum_{f=1}^{n_f} \gamma_f |\mathbf{S}_f| \left(\left| \frac{1}{max\left(\mathbf{d} \bullet \mathbf{n_f}, 0.05|\mathbf{d}|\right)} \right| (\phi_N - \phi_P) + \mathbf{k} \bullet (\nabla\phi)_f \right)$$

where we now can identify `nonOrthDeltaCoeffs` as the term

$$\texttt{nonOrthDeltaCoeffs} = \frac{\boldsymbol{\Delta}}{\mathbf{d}} = \frac{1}{max\left(\mathbf{d} \bullet \mathbf{n_f}, 0.05|\mathbf{d}|\right)}$$

The expression for $\boldsymbol{\Delta}$ is also used to compute $\mathbf{k}$ at line 375 in the function `makeNonOrthCorrectionVectors()` in `surfaceInterpolation.C`. The code reads

```
                            ———— surfaceInterpolation.C ————
375    corrVecs[facei] = unitArea - delta*NonOrthDeltaCoeffs[facei];
```

This corresponds exactly to

$$\mathbf{k} = \mathbf{n_f} - \boldsymbol{\Delta} = \mathbf{n_f} - \mathbf{d}\frac{\mathbf{d}}{max\left(\mathbf{n_f} \bullet \mathbf{d}, 0.05\mathbf{d}\right)} \tag{2.24}$$

In OpenFOAM, the non orthogonal correction in Equation (2.19) is performed in the function `fvmLaplacian` specialised to scalar diffusion found in `gaussLaplacianSchemes.C` and the first part of the code performing the uncorrected discretization with `fvmLaplacianUncorrected` is given by

```
                            ———— gaussLaplacianSchemes.C ————
37    template<>                                                              \
38    Foam::tmp<Foam::fvMatrix<Foam::Type>>                                   \
39    Foam::fv::gaussLaplacianScheme<Foam::Type, Foam::scalar>::fvmLaplacian  \
40    (                                                                       \
41        const GeometricField<scalar, fvsPatchField, surfaceMesh>& gamma,    \
42        const GeometricField<Type, fvPatchField, volMesh>& vf               \
```

```
43  )                                                                          \
44  {                                                                          \
45      const fvMesh& mesh = this->mesh();                                     \
46                                                                             \
47      GeometricField<scalar, fvsPatchField, surfaceMesh> gammaMagSf          \
48      (                                                                      \
49          gamma*mesh.magSf()                                                 \
50      );                                                                     \
51                                                                             \
52      tmp<fvMatrix<Type>> tfvm = fvmLaplacianUncorrected                     \
53      (                                                                      \
54          gammaMagSf,                                                        \
55          this->tsnGradScheme_().deltaCoeffs(vf),                            \
56          vf                                                                 \
57      );                                                                     \
58      fvMatrix<Type>& fvm = tfvm.ref();                                      \
59                                                                             \
```

At line 45 the code creates a reference to the mesh object. The term $\gamma|\mathbf{S}_f|$ is computed in line 47-50. The uncorrected matrix discretization is performed at line 52-57, and a reference to the created matrix object is returned in line 58.

The explicit non orthogonal correction for scalar diffusion is implemented inside an if statement starting at line 60 in gaussLaplacianSchemes.C as seen by

```
                              ─── gaussLaplacianSchemes.C ───
60      if (this->tsnGradScheme_().corrected())                                \
61      {                                                                      \
```

The code now enters another if statement given by

```
                              ─── gaussLaplacianSchemes.C ───
62          if (mesh.fluxRequired(vf.name()))                                  \
63          {                                                                  \
```

mesh.fluxRequired(vf.name()) is related to the dictionary fluxRequired in the case file system/fvSchemes. An example of this is found in the tutorial standingWave which is found at $FOAM_TUTORIALS/multiphase/interIsoFoam/standingWave. In system/fvSchemes, we find

```
                              ─── system/fvSchemes ───
50  fluxRequired
51  {
52      default         no;
53      p_rgh;
54      pcorr;
55      alpha.water;
56  }
```

It is seen that fluxRequired is turned off as default, however it is turned on for p_rgh, pcorr and alpha.water. p_rgh could for example correspond to vf in the code section line 62 just above from gaussLaplacianSchemes.C.

When fluxRequired is turned on, the code executes

```
                              ─── gaussLaplacianSchemes.C ───
64              fvm.faceFluxCorrectionPtr() = new                              \
65              GeometricField<Type, fvsPatchField, surfaceMesh>               \
66              (                                                              \
```

```
67              gammaMagSf*this->tsnGradScheme_().correction(vf)          \
68          );                                                            \
69                                                                        \
70          fvm.source() -=                                               \
71              mesh.V()*                                                 \
72              fvc::div                                                  \
73              (                                                         \
74                  *fvm.faceFluxCorrectionPtr()                          \
75              )().primitiveField();                                     \
76      }                                                                 \
```

At `line 64-68` the non orthogonal face flux correction is assigned to the face flux field for non orthogonal correction, which is a private data member declared at

─────── fvMatrix.H ───────
```
143         //- Face flux field for non-orthogonal correction
144         mutable GeometricField<Type, fvsPatchField, surfaceMesh>
145             *faceFluxCorrectionPtr_;
```

Access to this private member data is provided by

─────── fvMatrix.H ───────
```
338             //- Return pointer to face-flux non-orthogonal correction field
339             surfaceTypeFieldPtr& faceFluxCorrectionPtr()
340             {
341                 return faceFluxCorrectionPtr_;
342             }
```

The face flux correction assigned to `faceFluxCorrectionPtr_` is
`gammaMagSf*this->tsnGradScheme_().correction(vf)`
and this corresponds to the non orthogonal term inside the summation over faces from Equation (2.19) which reads

$$\underbrace{\gamma_f|\mathbf{S}_f|}_{\texttt{(gammaMagSf)}} \quad \underbrace{\mathbf{k} \bullet (\nabla\phi)_f}_{\texttt{(this->tsnGradScheme\_().correction(vf))}} \tag{2.25}$$

This is the non orthogonal correction to the face flux, thereby the name: faceFluxCorrectionPtr. The face flux correction pointer `faceFluxCorrectionPtr_` is used when we are calling the `flux()` method in the `fvMatrix` class. The `flux()` method returns the flux calculated from the matrix coefficients, and therefore we have to add the explicit contribution to get the correct flux, when we want to account for mesh non orthogonality. The `flux()` method is implemented in `fvMatrix.C` at `Line 923-1009`, and `faceFluxCorrectionPtr_` is used at `Line 1005`, where it is added to the flux computed from the matrix coefficients.

At `line 70-75` in `gaussLaplacianSchemes.C` the non orthogonal correction is subtracted from the matrix source vector, because it is moved from left to right side of the equal sign in the system equation $\mathbf{Ax} = \mathbf{b}$. Now we need the cell centred non orthogonal correction according to Equation (2.19), where the non orthogonal term is

$$\sum_{f=1}^{n_f} \gamma_f|\mathbf{S}_f|\mathbf{k} \bullet (\nabla\phi)_f \tag{2.26}$$

The explicit divergence `fvc::div` at line 72 `gaussLaplacianSchemes.C` calls the surface integration function `fvc::surfaceIntegrate`, and in this function the sum is divided by the cell volume. That is why the cell volume is multiplied with the internal field returned from the divergence operation at line 70-75 in `gaussLaplacianSchemes.C`.

When `fluxRequired` is turned off, the code executes

```
──────────────────────────── gaussLaplacianSchemes.C ────────────────────────────
77          else                                                        \
78          {                                                           \
79              fvm.source() -=                                         \
80                  mesh.V()*                                           \
81                  fvc::div                                            \
82                  (                                                   \
83                      gammaMagSf*this->tsnGradScheme_().correction(vf) \
84                  )().primitiveField();                               \
85          }                                                           \
86      }                                                               \
```

In this case the code just adds the non orthogonal source term to the matrix source vector, as just described for the previous section of code.

Non orthogonal and skew meshes

In Figure 2.1 case c) the mesh is non orthogonal and the vector from P to N (denoted **d**) does not intersect the face centre. In this case we will make an error when estimating the gradient at the cell face, because we are interpolating to the intersection point between the face and **d** instead of interpolating to the face centre which has been assumed when applying the mid point rule for the face integration.

Skewness correction is mentioned at page 254 [3] in relation to the description of the diffusion term. The skewness correction of the gradient is discussed at page 275-280 [3] and a presentation of how to implement a gradient computed using Gauss' theorem and interpolation that account for mesh skewness is presented at page 297 [3]. The face gradient in the non orthogonal term from the laplacian discretization can be corrected for mesh skewness using the following settings for a dependent variable ϕ

```
gradSchemes
{
    default         none;
    grad(phi)       Gauss skewCorrected linear;
}
```

The source files of this scheme is found in the folder

```
$FOAM_SRC/finiteVolume/interpolation/surfaceInterpolation/schemes/skewCorrected
```

and the description from `skewCorrected.H` reads

```
──────────────────────────── skewCorrected.H ────────────────────────────
32  Description
33      Skewness-corrected interpolation scheme that applies an explicit
34      correction to given scheme.
```

Explicit evaluation with scalar diffusion

The definition of the specialisation of the explicit evaluation function for scalar diffusion `fvcLaplacian` is given by

```
──────────────────────────── gaussLaplacianSchemes.C ────────────────────────────
92  template<>                                                                      \
93  Foam::tmp<Foam::GeometricField<Foam::Type, Foam::fvPatchField, Foam::volMesh>>  \
94  Foam::fv::gaussLaplacianScheme<Foam::Type, Foam::scalar>::fvcLaplacian          \
95  (                                                                               \
96      const GeometricField<scalar, fvsPatchField, surfaceMesh>& gamma,            \
97      const GeometricField<Type, fvPatchField, volMesh>& vf                       \
```

```
98    )                                                                     \
99    {                                                                     \
100       const fvMesh& mesh = this->mesh();                                \
101                                                                         \
102       tmp<GeometricField<Type, fvPatchField, volMesh>> tLaplacian       \
103       (                                                                 \
104           fvc::div(gamma*this->tsnGradScheme_().snGrad(vf)*mesh.magSf())\
105       );                                                                \
106                                                                         \
107       tLaplacian.ref().rename                                           \
108       (                                                                 \
109           "laplacian(" + gamma.name() + ',' + vf.name() + ')'           \
110       );                                                                \
111                                                                         \
112       return tLaplacian;                                                \
113   }
```

At line 102-105 the code evaluates the Laplacian term explicitly according to

$$\sum_{f=1}^{n_f} \gamma_f |\mathbf{S}_f| \mathbf{n}_f \bullet (\nabla\phi)_f \tag{2.27}$$

where $\texttt{fvc::div} = \sum_{f=1}^{n_f}$, $\texttt{gamma} = \gamma_f$, $\texttt{mesh.magSf()} = |\mathbf{S}_f|$ and
$\texttt{this->tsnGradScheme\_().snGrad(vf)} = \mathbf{n}_f \bullet (\nabla\phi)_f$.

2.2.2 Gauss discretization of Laplacian with diffusion tensor coefficient

The laplacian operator can handle scalar diffusion, as presented in previous section, however it can also handle directional diffusion represented by a tensor. This is for example needed in continuum mechanics for anisotropic materials or in fluid mechanics for fluid shear stresses.

As before Gauss's theorem is used to convert the volume integral into a surface integral, which gives

$$\int_{V_P} \nabla \bullet (\boldsymbol{\gamma}\nabla\phi) \ \mathrm{d}V = \int_{S_P} \mathrm{d}S \bullet (\boldsymbol{\gamma}\nabla\phi) \tag{2.28}$$

and as before we calculate the surface integral as a summation over the cell faces and use the midpoint rule to calculate the integral over each face, which gives

$$\int_{S_P} \mathrm{d}S \bullet (\boldsymbol{\gamma}\nabla\phi) = \sum_{f=1}^{n_f} \mathbf{S}_f \bullet (\boldsymbol{\gamma}\nabla\phi)_f \tag{2.29}$$

The diffusion coefficient $\boldsymbol{\gamma}_f$ is a second rank tensor

$$\boldsymbol{\gamma}_f = \begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} \\ \gamma_{21} & \gamma_{22} & \gamma_{23} \\ \gamma_{31} & \gamma_{32} & \gamma_{33} \end{bmatrix} \tag{2.30}$$

which consists of two bases as illustrated in Figure 2.3.
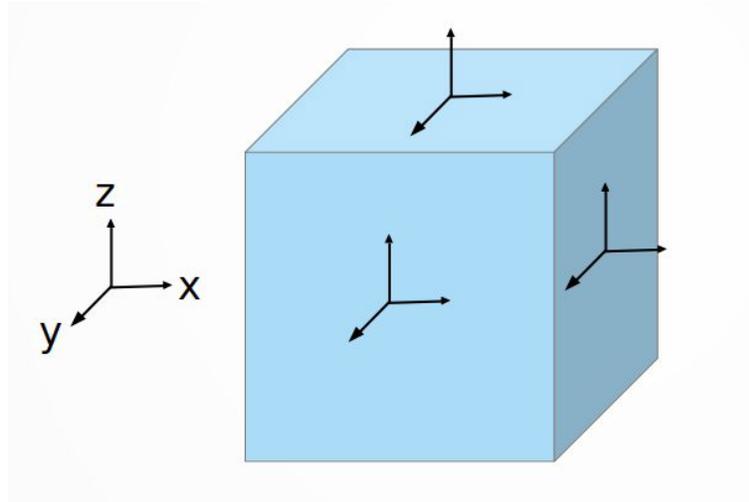
Figure 2.3: Illustration of tensor bases.

The cube with three surfaces that are orthogonal to each other represent the first basis. The second basis is the three directions in space which are orthogonal to each other. The two bases are combined and we thereby end up with 9 components in the tensor, as illustrated by the 9 arrows on the cube.

The implicit discretization of the laplacian operator with a diffusion tensor is performed in the function `fvmLaplacian`, which is one of the pure virtual functions in the base class `laplacianScheme`. The declaration of `fvmLaplacian` is given by

```
                         ────────────── gaussLaplacianScheme.H ──────────────
130          tmp<fvMatrix<Type>> fvmLaplacian
131          (
132              const GeometricField<GType, fvsPatchField, surfaceMesh>&,
133              const GeometricField<Type, fvPatchField, volMesh>&
134          );
```

where we see that it takes a surface field at line 132, that corresponds to γ_f in the theory. The second input at line 133 is a volume field, which corresponds to the dependent variable ϕ. The function returns a finite volume matrix, which represents a Laplacian term defined in a solver for example. The implementation of the function is given by

```
                         ────────────── gaussLaplacianScheme.C ──────────────
157   template<class Type, class GType>
158   tmp<fvMatrix<Type>>
159   gaussLaplacianScheme<Type, GType>::fvmLaplacian
160   (
161       const GeometricField<GType, fvsPatchField, surfaceMesh>& gamma,
162       const GeometricField<Type, fvPatchField, volMesh>& vf
163   )
164   {
165       const fvMesh& mesh = this->mesh();
166
167       const surfaceVectorField Sn(mesh.Sf()/mesh.magSf());
168
169       const surfaceVectorField SfGamma(mesh.Sf() & gamma);
170       const GeometricField<scalar, fvsPatchField, surfaceMesh> SfGammaSn
171       (
172           SfGamma & Sn
173       );
```

```cpp
174      const surfaceVectorField SfGammaCorr(SfGamma - SfGammaSn*Sn);

175

176      tmp<fvMatrix<Type>> tfvm = fvmLaplacianUncorrected

177      (

178          SfGammaSn,

179          this->tsnGradScheme_().deltaCoeffs(vf),

180          vf

181      );

182      fvMatrix<Type>& fvm = tfvm.ref();

183

184      tmp<GeometricField<Type, fvsPatchField, surfaceMesh>> tfaceFluxCorrection

185          = gammaSnGradCorr(SfGammaCorr, vf);

186

187      if (this->tsnGradScheme_().corrected())

188      {

189          tfaceFluxCorrection.ref() +=

190              SfGammaSn*this->tsnGradScheme_().correction(vf);

191      }

192

193      fvm.source() -= mesh.V()*fvc::div(tfaceFluxCorrection())().primitiveField();

194

195      if (mesh.fluxRequired(vf.name()))

196      {

197          fvm.faceFluxCorrectionPtr() = tfaceFluxCorrection.ptr();

198      }

199

200      return tfvm;

201 }
```

I will now explain the theory, and relate it to the above code. Equation (2.29) can be rewritten to

$$\sum_{f=1}^{n_f} \mathbf{S}_f \bullet (\boldsymbol{\gamma}\nabla\phi)_f = \sum_{f=1}^{n_f} (\mathbf{S}_f \bullet \boldsymbol{\gamma}_f) \bullet (\nabla\phi)_f \tag{2.31}$$

after some matrix algebra which I have written out to enhance the understanding.

$$\mathbf{S}_f \bullet (\boldsymbol{\gamma}\nabla\phi)_f = (\mathbf{S}_f)^{\mathrm{T}}(\boldsymbol{\gamma}\nabla\phi)_f$$

$$\mathbf{S}_f \bullet (\boldsymbol{\gamma}\nabla\phi)_f = \begin{bmatrix} S_{f,1} & S_{f,2} & S_{f,3} \end{bmatrix} \begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} \\ \gamma_{21} & \gamma_{22} & \gamma_{23} \\ \gamma_{31} & \gamma_{32} & \gamma_{33} \end{bmatrix} \begin{bmatrix} \left(\frac{\partial\phi}{\partial x}\right)_f \\ \left(\frac{\partial\phi}{\partial y}\right)_f \\ \left(\frac{\partial\phi}{\partial z}\right)_f \end{bmatrix}$$

$$\mathbf{S}_f \bullet (\boldsymbol{\gamma}\nabla\phi)_f = \begin{bmatrix} S_{f,1} & S_{f,2} & S_{f,3} \end{bmatrix} \begin{bmatrix} \left(\frac{\partial\phi}{\partial x}\right)_f \gamma_{11} + \left(\frac{\partial\phi}{\partial y}\right)_f \gamma_{21} + \left(\frac{\partial\phi}{\partial z}\right)_f \gamma_{31} \\ \left(\frac{\partial\phi}{\partial x}\right)_f \gamma_{12} + \left(\frac{\partial\phi}{\partial y}\right)_f \gamma_{22} + \left(\frac{\partial\phi}{\partial z}\right)_f \gamma_{32} \\ \left(\frac{\partial\phi}{\partial x}\right)_f \gamma_{13} + \left(\frac{\partial\phi}{\partial y}\right)_f \gamma_{23} + \left(\frac{\partial\phi}{\partial z}\right)_f \gamma_{33} \end{bmatrix}$$

$$\mathbf{S}_f \bullet (\boldsymbol{\gamma}\nabla\phi)_f = S_{f,1}\left(\left(\frac{\partial\phi}{\partial x}\right)_f \gamma_{11} + \left(\frac{\partial\phi}{\partial y}\right)_f \gamma_{21} + \left(\frac{\partial\phi}{\partial z}\right)_f \gamma_{31}\right)$$

$$+ S_{f,2}\left(\left(\frac{\partial\phi}{\partial x}\right)_f \gamma_{12} + \left(\frac{\partial\phi}{\partial y}\right)_f \gamma_{22} + \left(\frac{\partial\phi}{\partial z}\right)_f \gamma_{32}\right)$$

$$+ S_{f,3}\left(\left(\frac{\partial\phi}{\partial x}\right)_f \gamma_{13} + \left(\frac{\partial\phi}{\partial y}\right)_f \gamma_{23} + \left(\frac{\partial\phi}{\partial z}\right)_f \gamma_{33}\right)$$

$$\mathbf{S}_f \bullet (\boldsymbol{\gamma}\nabla\phi)_f = \begin{bmatrix} S_{f,1}\gamma_{11} + S_{f,2}\gamma_{21} + S_{f,3}\gamma_{31} \\ S_{f,1}\gamma_{12} + S_{f,2}\gamma_{22} + S_{f,3}\gamma_{32} \\ S_{f,1}\gamma_{13} + S_{f,2}\gamma_{23} + S_{f,3}\gamma_{33} \end{bmatrix} \begin{bmatrix} \left(\frac{\partial\phi}{\partial x}\right)_f \\ \left(\frac{\partial\phi}{\partial y}\right)_f \\ \left(\frac{\partial\phi}{\partial z}\right)_f \end{bmatrix}$$

$$\mathbf{S}_f \bullet (\boldsymbol{\gamma}\nabla\phi)_f = \left(\begin{bmatrix} S_{f,1} & S_{f,2} & S_{f,3} \end{bmatrix} \begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} \\ \gamma_{21} & \gamma_{22} & \gamma_{23} \\ \gamma_{31} & \gamma_{32} & \gamma_{33} \end{bmatrix}\right) \begin{bmatrix} \left(\frac{\partial\phi}{\partial x}\right)_f \\ \left(\frac{\partial\phi}{\partial y}\right)_f \\ \left(\frac{\partial\phi}{\partial z}\right)_f \end{bmatrix}$$

$$\mathbf{S}_f \bullet (\boldsymbol{\gamma}\nabla\phi)_f = ((\mathbf{S}_f)^{\mathrm{T}}\boldsymbol{\gamma}_f)(\nabla\phi)_f$$

$$\mathbf{S}_f \bullet (\boldsymbol{\gamma}\nabla\phi)_f = (\mathbf{S}_f \bullet \boldsymbol{\gamma}_f) \bullet (\nabla\phi)_f$$

The term $\mathbf{S}_f \bullet \boldsymbol{\gamma}_f$ represent the inner product of a vector $\mathbf{S}_f$ and a tensor $\boldsymbol{\gamma}_f$ which is defined in OpenFOAM as

$$\texttt{SfGamma} = \mathbf{S}_f \bullet \boldsymbol{\gamma}_f = \begin{bmatrix} S_{f,1} & S_{f,2} & S_{f,3} \end{bmatrix} \begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} \\ \gamma_{21} & \gamma_{22} & \gamma_{23} \\ \gamma_{31} & \gamma_{32} & \gamma_{33} \end{bmatrix} = \begin{bmatrix} S_{f,1}\gamma_{11} + S_{f,2}\gamma_{21} + S_{f,3}\gamma_{31} \\ S_{f,1}\gamma_{12} + S_{f,2}\gamma_{22} + S_{f,3}\gamma_{32} \\ S_{f,1}\gamma_{13} + S_{f,2}\gamma_{23} + S_{f,3}\gamma_{33} \end{bmatrix} \quad (2.32)$$

The equivalent line in the code is

```
━━━━━━━━━ gaussLaplacianScheme.C ━━━━━━━━━
169      const surfaceVectorField SfGamma(mesh.Sf() & gamma);
```

Figure 2.4 gives a visual representation of `SfGamma`, along with the projected vectors `SfGammaSn*Sn` and `SfGammaCorr`. The calculation of the projected vectors is explained after the figure.
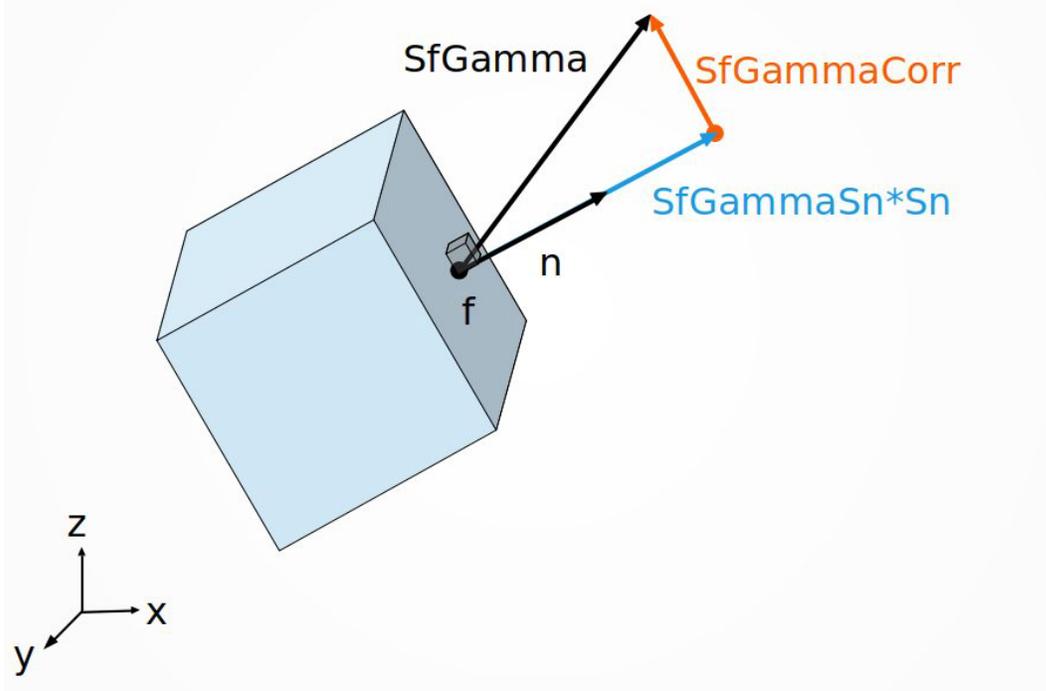
Figure 2.4: Illustration of mathematical operations performed when the diffusion coefficient is a tensor $\boldsymbol{\gamma}_f$. The operations are performed in the function `fvmLaplacian` in `gaussLaplacianScheme.C, Line: 167 – 174`.

`SfGamma` represent the diffusion vector through the cell face (f) defined by the surface area vector $\mathbf{S}_f$. Note here that `SfGamma` does not have to be orthogonal to the cell face. The face summation in Equation (2.31) is separated in the face orthogonal diffusion $(\mathbf{S}_f \bullet \boldsymbol{\gamma}_f)^\perp$ and the face parallel diffusion $(\mathbf{S}_f \bullet \boldsymbol{\gamma}_f)^\parallel$, because we cannot express the face parallel diffusion from the value of ϕ in the straddling cell centres owner and neighbour. The separation gives

$$\sum_{f=1}^{n_f}(\mathbf{S}_f \bullet \boldsymbol{\gamma}_f) \bullet (\nabla\phi)_f = \sum_{f=1}^{n_f}(\mathbf{S}_f \bullet \boldsymbol{\gamma}_f)^\perp \bullet (\nabla\phi)_f + (\mathbf{S}_f \bullet \boldsymbol{\gamma}_f)^\parallel \bullet (\nabla\phi)_f \qquad (2.33)$$

where $(\mathbf{S}_f \bullet \boldsymbol{\gamma}_f)^\perp = $ `SfGammaSn*Sn` and $(\mathbf{S}_f \bullet \boldsymbol{\gamma}_f)^\parallel = $ `SfGammaCorr`. The face orthogonal diffusion $(\mathbf{S}_f \bullet \boldsymbol{\gamma}_f)^\perp \bullet (\nabla\phi)_f$ can be corrected for mesh non orthogonality, using the same procedure as performed for the scalar diffusion case, where $\mathbf{n}_f \bullet (\nabla\phi)_f = |\boldsymbol{\Delta}|\frac{\phi_N-\phi_P}{|\mathbf{d}|} + \mathbf{k} \bullet (\nabla\phi)_f$. The non orthogonal mesh correction gives

$$
\begin{aligned}
(\mathbf{S}_f \bullet \boldsymbol{\gamma}_f)^\perp \bullet (\nabla\phi)_f &= |(\mathbf{S}_f \bullet \boldsymbol{\gamma}_f)^\perp| \ \mathbf{n}_f \bullet (\nabla\phi)_f \\
&= |(\mathbf{S}_f \bullet \boldsymbol{\gamma}_f)^\perp|\boldsymbol{\Delta} \bullet (\nabla\phi)_f + |(\mathbf{S}_f \bullet \boldsymbol{\gamma}_f)^\perp|\mathbf{k} \bullet (\nabla\phi)_f \\
&= |(\mathbf{S}_f \bullet \boldsymbol{\gamma}_f)^\perp||\boldsymbol{\Delta}|\mathbf{n}_f \bullet (\nabla\phi)_f + |(\mathbf{S}_f \bullet \boldsymbol{\gamma}_f)^\perp|\mathbf{k} \bullet (\nabla\phi)_f \\
&= |(\mathbf{S}_f \bullet \boldsymbol{\gamma}_f)^\perp||\boldsymbol{\Delta}|\frac{\phi_N-\phi_P}{|\mathbf{d}|} + |(\mathbf{S}_f \bullet \boldsymbol{\gamma}_f)^\perp|\mathbf{k} \bullet (\nabla\phi)_f \\
&= |(\mathbf{S}_f \bullet \boldsymbol{\gamma}_f)^{\perp,\boldsymbol{\Delta}}|\frac{\phi_N-\phi_P}{|\mathbf{d}|} + (\mathbf{S}_f \bullet \boldsymbol{\gamma}_f)^{\perp,\mathbf{k}} \bullet (\nabla\phi)_f \qquad (2.34)
\end{aligned}
$$

and the full summation now reads

$$\sum_{f=1}^{n_f}(\mathbf{S}_f \bullet \boldsymbol{\gamma}_f) \bullet (\nabla\phi)_f = \sum_{f=1}^{n_f}|(\mathbf{S}_f \bullet \boldsymbol{\gamma}_f)^{\perp,\boldsymbol{\Delta}}|\frac{\phi_N-\phi_P}{|\mathbf{d}|} + (\mathbf{S}_f \bullet \boldsymbol{\gamma}_f)^{\perp,\mathbf{k}} \bullet (\nabla\phi)_f + (\mathbf{S}_f \bullet \boldsymbol{\gamma}_f)^\parallel \bullet (\nabla\phi)_f$$

$$(2.35)$$

where $(\mathbf{S}_f \bullet \boldsymbol{\gamma}_f)^{\perp} = |(\mathbf{S}_f \bullet \boldsymbol{\gamma}_f)^{\perp}| \, \mathbf{n}_f$, and $|(\mathbf{S}_f \bullet \boldsymbol{\gamma}_f)^{\perp}|$ is `SfGammaSn`. `SfGammaSn` is the magnitude of the projection of `SfGamma` on to the surface normal vector $\mathbf{n}_f$. The calculation is given by

$$
\begin{aligned}
\texttt{SfGammaSn} = (\mathbf{S}_f \bullet \boldsymbol{\gamma}_f) \bullet \mathbf{n}_f &= \begin{bmatrix} S_{f,1}\gamma_{11} + S_{f,2}\gamma_{21} + S_{f,3}\gamma_{31} \\ S_{f,1}\gamma_{12} + S_{f,2}\gamma_{22} + S_{f,3}\gamma_{32} \\ S_{f,1}\gamma_{13} + S_{f,2}\gamma_{23} + S_{f,3}\gamma_{33} \end{bmatrix} \begin{bmatrix} n_{f,1} \\ n_{f,2} \\ n_{f,3} \end{bmatrix} \\
&= (S_{f,1}\gamma_{11} + S_{f,2}\gamma_{21} + S_{f,3}\gamma_{31})n_{f,1} \\
&+ (S_{f,1}\gamma_{12} + S_{f,2}\gamma_{22} + S_{f,3}\gamma_{32})n_{f,2} \\
&+ (S_{f,1}\gamma_{13} + S_{f,2}\gamma_{23} + S_{f,3}\gamma_{33})n_{f,3}
\end{aligned} \tag{2.36}
$$

The normal vector $\mathbf{n}_f$ is defined as `Sn` in line 167, which reads

```
————————————————————— gaussLaplacianScheme.C —————————————————————
167        const surfaceVectorField Sn(mesh.Sf()/mesh.magSf());
```

and `SfGammaSn` is calculated by the code

```
————————————————————— gaussLaplacianScheme.C —————————————————————
170        const GeometricField<scalar, fvsPatchField, surfaceMesh> SfGammaSn
171        (
172            SfGamma & Sn
173        );
```

To get the projected magnitude `SfGammaSn` as a vector `SfGammaSn*Sn` we need to scale it with the surface normal vector $\mathbf{n}_f$, as given by

$$
\texttt{SfGammaSn*Sn} = ((\mathbf{S}_f \bullet \boldsymbol{\gamma}_f) \bullet \mathbf{n}_f) \cdot \mathbf{n}_f =
$$

$$
\begin{bmatrix} S_{f,1}\gamma_{11} + S_{f,2}\gamma_{21} + S_{f,3}\gamma_{31} \\ S_{f,1}\gamma_{12} + S_{f,2}\gamma_{22} + S_{f,3}\gamma_{32} \\ S_{f,1}\gamma_{13} + S_{f,2}\gamma_{23} + S_{f,3}\gamma_{33} \end{bmatrix} \cdot \begin{bmatrix} n_{f,1} \\ n_{f,2} \\ n_{f,3} \end{bmatrix} =
$$

$$
\begin{bmatrix} \left((S_{f,1}\gamma_{11} + S_{f,2}\gamma_{21} + S_{f,3}\gamma_{31})n_{f,1} + (S_{f,1}\gamma_{12} + S_{f,2}\gamma_{22} + S_{f,3}\gamma_{32})n_{f,2} + (S_{f,1}\gamma_{13} + S_{f,2}\gamma_{23} + S_{f,3}\gamma_{33})n_{f,3}\right) n_{f,1} \\ \left((S_{f,1}\gamma_{11} + S_{f,2}\gamma_{21} + S_{f,3}\gamma_{31})n_{f,1} + (S_{f,1}\gamma_{12} + S_{f,2}\gamma_{22} + S_{f,3}\gamma_{32})n_{f,2} + (S_{f,1}\gamma_{13} + S_{f,2}\gamma_{23} + S_{f,3}\gamma_{33})n_{f,3}\right) n_{f,2} \\ \left((S_{f,1}\gamma_{11} + S_{f,2}\gamma_{21} + S_{f,3}\gamma_{31})n_{f,1} + (S_{f,1}\gamma_{12} + S_{f,2}\gamma_{22} + S_{f,3}\gamma_{32})n_{f,2} + (S_{f,1}\gamma_{13} + S_{f,2}\gamma_{23} + S_{f,3}\gamma_{33})n_{f,3}\right) n_{f,3} \end{bmatrix} \tag{2.37}
$$

Figure 2.4 shows the geometrical representation of `SfGammaSn*Sn`, which represents the diffusion in the direction orthogonal to the cell face. The cross diffusion parallel to the cell face $(\mathbf{S}_f \bullet \boldsymbol{\gamma}_f)^{\|}$ has the name `SfGammaCorr` in the code, and it is calculated as

$$
\texttt{SfGammaCorr} = \mathbf{S}_f \bullet \boldsymbol{\gamma}_f - ((\mathbf{S}_f \bullet \boldsymbol{\gamma}_f) \bullet \mathbf{n}_f) \cdot \mathbf{n}_f = \tag{2.38}
$$

$$
\texttt{SfGamma - SfGammaSn*Sn} =
$$

$$
\begin{bmatrix} S_{f,1}\gamma_{11} + S_{f,2}\gamma_{21} + S_{f,3}\gamma_{31} \\ S_{f,1}\gamma_{12} + S_{f,2}\gamma_{22} + S_{f,3}\gamma_{32} \\ S_{f,1}\gamma_{13} + S_{f,2}\gamma_{23} + S_{f,3}\gamma_{33} \end{bmatrix} - \tag{2.39}
$$

$$
\begin{bmatrix} \left((S_{f,1}\gamma_{11} + S_{f,2}\gamma_{21} + S_{f,3}\gamma_{31})n_{f,1} + (S_{f,1}\gamma_{12} + S_{f,2}\gamma_{22} + S_{f,3}\gamma_{32})n_{f,2} + (S_{f,1}\gamma_{13} + S_{f,2}\gamma_{23} + S_{f,3}\gamma_{33})n_{f,3}\right) n_{f,1} \\ \left((S_{f,1}\gamma_{11} + S_{f,2}\gamma_{21} + S_{f,3}\gamma_{31})n_{f,1} + (S_{f,1}\gamma_{12} + S_{f,2}\gamma_{22} + S_{f,3}\gamma_{32})n_{f,2} + (S_{f,1}\gamma_{13} + S_{f,2}\gamma_{23} + S_{f,3}\gamma_{33})n_{f,3}\right) n_{f,2} \\ \left((S_{f,1}\gamma_{11} + S_{f,2}\gamma_{21} + S_{f,3}\gamma_{31})n_{f,1} + (S_{f,1}\gamma_{12} + S_{f,2}\gamma_{22} + S_{f,3}\gamma_{32})n_{f,2} + (S_{f,1}\gamma_{13} + S_{f,2}\gamma_{23} + S_{f,3}\gamma_{33})n_{f,3}\right) n_{f,3} \end{bmatrix} \tag{2.40}
$$

`SfGammaCorr` is illustrated in Figure 2.4, which shows that the vector will point in some direction in a plane parallel to the cell face. The calculation of `SfGammaCorr` involves `SfGammaSn*Sn` and it is performed by the code

```
————————————————————— gaussLaplacianScheme.C —————————————————————
174        const surfaceVectorField SfGammaCorr(SfGamma - SfGammaSn*Sn);
```

The diffusion through each face has now been separated in an orthogonal diffusion magnitude `SfGammaSn` and cross diffusion vector parallel to the face `SfGammaCorr`. Furthermore the orthogonal diffusion has been separated in a orthogonal and a non orthogonal contribution with respect to the vector **d** from owner to neighbour cell centre.

Face orthogonal diffusion

The first term in Equation (2.35), $|(\mathbf{S}_f \bullet \boldsymbol{\gamma}_f)^{\perp,\Delta}| \frac{\phi_N - \phi_P}{|\mathbf{d}|} f$, is treated implicitly and it is handled by the same function as for scalar diffusion. The function is called at line 176 where the code reads.

```
                          gaussLaplacianScheme.C
176      tmp<fvMatrix<Type>> tfvm = fvmLaplacianUncorrected
177      (
178          SfGammaSn,
179          this->tsnGradScheme_().deltaCoeffs(vf),
180          vf
181      );
182      fvMatrix<Type>& fvm = tfvm.ref();
```

The code accounts for the orthogonal vector $\boldsymbol{\Delta}$ in the same way as presented for scalar diffusion, via the surface normal gradient scheme in the input `this->tsnGradScheme_().deltaCoeffs(vf)`. This input is linked to whatever type of surface normal gradient which we choose to use in `system/fvSchemes`.

Face parallel diffusion

The third term in Equation (2.35), $(\mathbf{S}_f \bullet \boldsymbol{\gamma}_f)^{\|} \bullet (\nabla \phi)_f$ can only be handled explicitly and it is handled by the code

```
                          gaussLaplacianScheme.C
184      tmp<GeometricField<Type, fvsPatchField, surfaceMesh>> tfaceFluxCorrection
185          = gammaSnGradCorr(SfGammaCorr, vf);
186
187      if (this->tsnGradScheme_().corrected())
188      {
189          tfaceFluxCorrection.ref() +=
190              SfGammaSn*this->tsnGradScheme_().correction(vf);
191      }
192
193      fvm.source() -= mesh.V()*fvc::div(tfaceFluxCorrection())().primitiveField();
194
195      if (mesh.fluxRequired(vf.name()))
196      {
197          fvm.faceFluxCorrectionPtr() = tfaceFluxCorrection.ptr();
198      }
```

The function `gammaSnGradCorr` is called in line 184-185. This function is defined by

```
                          gaussLaplacianScheme.C
95   template<class Type, class GType>
96   tmp<GeometricField<Type, fvsPatchField, surfaceMesh>>
97   gaussLaplacianScheme<Type, GType>::gammaSnGradCorr
98   (
99       const surfaceVectorField& SfGammaCorr,
100      const GeometricField<Type, fvPatchField, volMesh>& vf
101  )
102  {
103      const fvMesh& mesh = this->mesh();
104
```

```
105    tmp<GeometricField<Type, fvsPatchField, surfaceMesh>> tgammaSnGradCorr
106    (
107        new GeometricField<Type, fvsPatchField, surfaceMesh>
108        (
109            IOobject
110            (
111                "gammaSnGradCorr("+vf.name()+')',
112                vf.instance(),
113                mesh,
114                IOobject::NO_READ,
115                IOobject::NO_WRITE
116            ),
117            mesh,
118            SfGammaCorr.dimensions()
119           *vf.dimensions()*mesh.deltaCoeffs().dimensions()
120        )
121    );
122    tgammaSnGradCorr.ref().oriented() = SfGammaCorr.oriented();
123
124    for (direction cmpt = 0; cmpt < pTraits<Type>::nComponents; cmpt++)
125    {
126        tgammaSnGradCorr.ref().replace
127        (
128            cmpt,
129            fvc::dotInterpolate(SfGammaCorr, fvc::grad(vf.component(cmpt)))
130        );
131    }
132
133    return tgammaSnGradCorr;
134 }
```

The function creates a new surface field in line 105-121 as an instance of the dependent variable
vf. In line 122 the settings in oriented() is transferred. Then the function loops through each
component of the dependent variable vf and computes the parallel diffusion correction face flux
$(\mathbf{S}_f \bullet \boldsymbol{\gamma}_f)^{\|} \bullet \nabla(\phi)_f$ with the code

```
fvc::dotInterpolate(SfGammaCorr, fvc::grad(vf.component(cmpt)))
```

Non orthogonal correction of orthogonal diffusion

The second term in Equation (2.35), $(\mathbf{S}_f \bullet \boldsymbol{\gamma}_f)^{\perp,\mathbf{k}} \bullet (\nabla\phi)_f$, accounts for mesh non orthogonality and
SfGammaSn is corrected according to the chosen surface normal gradient scheme. The implementation
is given by the lines

```
─────────────── gaussLaplacianScheme.C ───────────────
187    if (this->tsnGradScheme_().corrected())
188    {
189        tfaceFluxCorrection.ref() +=
190            SfGammaSn*this->tsnGradScheme_().correction(vf);
191    }
```

The contributions from diffusion parallel to the face and possibly mesh non orthogonality correction
is added to the matrix source vector by the code

```
─────────────── gaussLaplacianScheme.C ───────────────
193    fvm.source() -= mesh.V()*fvc::div(tfaceFluxCorrection())().primitiveField();
```

As in the case with scalar diffusion, we need to account for the explicit fluxes in the `fvMatrix` method `flux()`. So the explicit flux corrections is assigned to the face flux pointer by the code

─────────────────────────── gaussLaplacianScheme.C ───────────────────────────
```
195     if (mesh.fluxRequired(vf.name()))
196     {
197         fvm.faceFluxCorrectionPtr() = tfaceFluxCorrection.ptr();
198     }
```

Explicit evaluation without diffusion parameter

This function must be defined according to the base class, and it evaluates the Laplacian operator explicitly without any diffusion coefficient, hence it corresponds to evaluating

$$\nabla \bullet (\nabla \phi) \tag{2.41}$$

The function declaration is given by

─────────────────────────── gaussLaplacianScheme.H ───────────────────────────
```
125         tmp<GeometricField<Type, fvPatchField, volMesh>> fvcLaplacian
126         (
127             const GeometricField<Type, fvPatchField, volMesh>&
128         );
```

where it is seen that the function only takes one input which is a volume field ϕ. The function returns a geometric field of same type as the input field. The function definition is given by

─────────────────────────── gaussLaplacianScheme.C ───────────────────────────
```
137 template<class Type, class GType>
138 tmp<GeometricField<Type, fvPatchField, volMesh>>
139 gaussLaplacianScheme<Type, GType>::fvcLaplacian
140 (
141     const GeometricField<Type, fvPatchField, volMesh>& vf
142 )
143 {
144     const fvMesh& mesh = this->mesh();
145
146     tmp<GeometricField<Type, fvPatchField, volMesh>> tLaplacian
147     (
148         fvc::div(this->tsnGradScheme_().snGrad(vf)*mesh.magSf())
149     );
150
151     tLaplacian.ref().rename("laplacian(" + vf.name() + ')');
152
153     return tLaplacian;
154 }
```

The code tells us that output is calculated as the divergence of the surface normal gradient of ϕ multiplied with the magnitude of the surface area vector, i.e. the face area.

Explicit evaluation with diffusion tensor

The function must be defined according to the base class, and it evaluates the Laplacian operator explicitly including a diffusion coefficient, hence it corresponds to evaluating

$$\nabla \bullet (\gamma \nabla \phi) \tag{2.42}$$

The function is declared by

```
                          gaussLaplacianScheme.H
136          tmp<GeometricField<Type, fvPatchField, volMesh>> fvcLaplacian
137          (
138              const GeometricField<GType, fvsPatchField, surfaceMesh>&,
139              const GeometricField<Type, fvPatchField, volMesh>&
140          );
141     };
```

and the function definition is given by

```
                          gaussLaplacianScheme.C
204     template<class Type, class GType>
205     tmp<GeometricField<Type, fvPatchField, volMesh>>
206     gaussLaplacianScheme<Type, GType>::fvcLaplacian
207     (
208         const GeometricField<GType, fvsPatchField, surfaceMesh>& gamma,
209         const GeometricField<Type, fvPatchField, volMesh>& vf
210     )
211     {
212         const fvMesh& mesh = this->mesh();
213
214         const surfaceVectorField Sn(mesh.Sf()/mesh.magSf());
215         const surfaceVectorField SfGamma(mesh.Sf() & gamma);
216         const GeometricField<scalar, fvsPatchField, surfaceMesh> SfGammaSn
217         (
218             SfGamma & Sn
219         );
220         const surfaceVectorField SfGammaCorr(SfGamma - SfGammaSn*Sn);
221
222         tmp<GeometricField<Type, fvPatchField, volMesh>> tLaplacian
223         (
224             fvc::div
225             (
226                 SfGammaSn*this->tsnGradScheme_().snGrad(vf)
227               + gammaSnGradCorr(SfGammaCorr, vf)
228             )
229         );
230
231         tLaplacian.ref().rename
232         (
233             "laplacian(" + gamma.name() + ',' + vf.name() + ')'
234         );
235
236         return tLaplacian;
237     }
```

The code performs exactly the same operations as in `fvmLaplacian`, hence they are not repeated
again. The only difference is that the terms is evaluated explicitly.

2.3   Implementation

This chapter describes aspects in the source code of the Gauss discretization scheme for the laplacian
operator, its base class and how the call gets from an application to the discretization scheme. The
scheme is found in the class `gaussLaplacianScheme`, which is located at

```
$FOAM_SRC/finiteVolume/finiteVolume/laplacianSchemes/gaussLaplacianScheme
```

There are three files in the folder, which are

```
gaussLaplacianScheme.H
gaussLaplacianScheme.C
gaussLaplacianSchemes.C
```

`gaussLaplacianScheme.H` is the class declaration, `gaussLaplacianScheme.C` is the class definition and `gaussLaplacianSchemes.C` is a class specialisation, that makes a special definition only used when the diffusion parameter is scalar.

The `gaussLaplacianScheme` class is a templated class with two template parameters `Type` and `GType`, and it is derived from a templated abstract base class `laplacianScheme` with the same template parameters. Macros are used to emulate partial specialisation of the laplacian functions for scalar diffusion, where `GType = scalar`. We will go into some code details, when we go through the source code in the coming sections, however the link between code and theory is found in the theoretical section.

2.3.1 From solver call to source code

Before going into the details with the source code, I would like to explain how the laplacian operator is called from an application and what happens inside the code when we run a simulation. The laplacian operator can be called in the construction of an `fvMatrix<Type>` object, where the available matrix types for the template parameter `Type` are

```
scalar
vector
sphericalTensor
symTensor
tensor
```

For example we could express the equation

$$\nabla \bullet (\gamma \ \nabla \phi) = 0 \tag{2.43}$$

where γ (`gamma`) is a scalar diffusivity and ϕ (`phi`) is a scalar field, as

```
fvMatrix<scalar> phiEqn
(
    fvm::laplacian(gamma, phi)
);
```

When the right hand side is not specified, OpenFOAM automatically knows that it is zero.

The call `fvm::laplacian(gamma, phi)` will look for a function with matching input and output types in the namespace definition `fvmLaplacian`, which declares and defines the function `laplacian` in namespace `fvm`. The source code of this class is found at

```
$FOAM_SRC/finiteVolume/finiteVolume/fvm
```

The function `laplacian` in `fvmLaplacian` is overloaded, and the namespace definition defines 16 different declarations. I will not go through the specific path through all these functions as it will vary depending on the specific call, but the `laplacian` functions call each other, and at some point on the way, we will end up in the function

```
template<class Type, class GType>
tmp<fvMatrix<Type>>
laplacian
(
```

```
    const GeometricField<GType, fvPatchField, volMesh>& gamma,
    const GeometricField<Type, fvPatchField, volMesh>& vf,
    const word& name
)
{
    return fv::laplacianScheme<Type, GType>::New
    (
        vf.mesh(),
        vf.mesh().laplacianScheme(name)
    ).ref().fvmLaplacian(gamma, vf);
}
```

if the diffusivity γ is a volume field, i.e. the values are placed in the cell-centres. The function returns the output from the function `laplacianScheme<Type, GType>::fvmLaplacian` in the class `laplacianScheme`. The call to `laplacianScheme<Type, GType>::fvmLaplacian` is performed by first calling the selector function `fv::laplacianScheme<Type, GType>::New`, which returns an output of type `tmp<laplacianScheme<Type, GType>>`. This output calls the function `ref()` from the class `tmp`. The output of `ref()` is a reference to the new object of type `laplacianScheme<Type, GType>`. From this new object we call the function `laplacianScheme<Type, GType>::fvmLaplacian` with input `gamma` and `vf`.

The function `laplacianScheme<Type, GType>::fvmLaplacian` returns

`fvmLaplacian(tinterpGammaScheme_().interpolate(gamma)(), vf)`

which is a call to the function `fvmLaplacian` in the class, that is associated with the type name, which has been specified in the `system/fvSchemes` under `laplacianSchemes`, since this input is being passed to the `New` function in `laplacianScheme<Type, GType>`. If we write the keyword `Gauss` in `laplacianSchemes`, then

`fvmLaplacian(tinterpGammaScheme_().interpolate(gamma)(), vf)`

is a call to the function `gaussLaplacianScheme<Type, GType>::fvmLaplacian` in the class `gaussLaplacianScheme`.

2.3.2 Description of laplacianScheme class

In the previous section we found that an object of the class `laplacianScheme` was created from our call to the function `laplacian` in the namespace `fvm` (declared in `fvmLaplacian`). `laplacianScheme` is an abstract base class, and the source code is found in the folder

`$FOAM_SRC/finiteVolume/finiteVolume/laplacianSchemes/laplacianScheme`

The purpose of this class is to create a common interface for all laplacian discretization schemes. It is thus the intention that all laplacian discretization schemes should inherit from this abstract base class, which will provide some basic functionality and enforce a certain structure, so that we can always call the laplacian operator in the same way no matter what scheme we choose. Now I will go through the declaration and the source, since we need to know about the underlying abstract base class, before we can start to modify the Gauss scheme for the laplacian operator (`gaussLaplacianScheme`).

First part of declaration file

We will start in the class declaration `laplacianScheme.H`. The class is protected against multiple includes via

───── laplacianScheme.H ─────
```
40   #ifndef laplacianScheme_H
41   #define laplacianScheme_H
```

that first examine if `laplacianScheme_H` is defined in line 40, and if not, it is defined in line 41. If `laplacianScheme_H` is defined, the code will jump to the end of the file at line 264.

───── laplacianScheme.H ─────
```
264  #endif
```

If `laplacianScheme_H` is not defined, the declaration will be included. First the needed class declarations are specified in line 43-49.

───── laplacianScheme.H ─────
```
43   #include "tmp.H"
44   #include "volFieldsFwd.H"
45   #include "surfaceFieldsFwd.H"
46   #include "linear.H"
47   #include "correctedSnGrad.H"
48   #include "typeInfo.H"
49   #include "runTimeSelectionTables.H"
```

The next section of code is

───── laplacianScheme.H ─────
```
53   namespace Foam
54   {
55
56   template<class Type>
57   class fvMatrix;
58
59   class fvMesh;
60
61   // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
62
63   namespace fv
64   {
```

The code enters namespace `Foam` and makes two forward declarations of the classes `template<class Type> class fvMatrix` and `class fvMesh`. Now the class `laplacianScheme` will know about these two classes, even though they may not have been declared yet. They should just be declared at some point. At line 63 the code enters the namespace `fv`. This means, that if we want to call the class without being inside a namespace we need to access it by `Foam::fv::laplacianScheme`.

The class declaration of `laplacianScheme` is found between the code

```
                             ─── laplacianScheme.H ───
70  template<class Type, class GType>
71  class laplacianScheme
72  :
73      public refCount
74  {
```

and the closing bracket

```
                             ─── laplacianScheme.H ───
206 };
```

`template<class Type, class GType>` defines that the class is a templated class with two template parameters, that is named `Type` and `GType` in this case. `Type` specifies the matrix type in the templated matrix class `fvMatrix<Type>`. If `Type=scalar`, we have an `fvMatrix<scalar>`, which is better known by the typedef `fvScalarMatrix`. `GType` is related to the diffusion parameter γ, hence for a scalar diffusion parameter `GType=scalar`, we would have an object of the class `GeometricField<scalar, fvPatchField, volMesh>`. This more commonly known by the typedef name `volScalarField`.

`class laplacianScheme` defines the name of the class, and the colon at line 72 should be read as "is derived from". Hence the class is derived from `refCount` with maximum visibility defined by the keyword `public`.

Protected member data

The class has three protected data members as seen from

```
                             ─── laplacianScheme.H ───
76  protected:
77
78      // Protected data
79
80          const fvMesh& mesh_;
81          tmp<surfaceInterpolationScheme<GType>> tinterpGammaScheme_;
82          tmp<snGradScheme<Type>> tsnGradScheme_;
```

Line 80 declares a constant reference to the mesh. Line 81 declares a temporary surface interpolation scheme object, which is related to the interpolation of γ from cell-centres to face-centres. Line 82 declares a temporary surface normal gradient scheme, which is related to the non orthogonal correction of the surface normal gradient in the laplacian operator. The protected data members are initialised in the initialiser list of the constructors.

Runtime type information

In the next section of code we now enter the public part of the class, and the first public declaration is a pure virtual function named `type` with no inputs and it returns a constant word reference, as seen by

```
                             ─── laplacianScheme.H ───
96  public:
97
```

```
98      //- Runtime type information
99      virtual const word& type() const = 0;
```

The declarations tells, that all classes derived from this class must define this function. It is not easy to see how this function is redefined in the derived classes, however the redefinition is performed at line 82 in gaussLaplacianScheme.H, where the code reads

```
————————————————————————— gaussLaplacianScheme.H —————————————————————————
81      //- Runtime type information
82      TypeName("Gauss");
```

TypeName is a macro that needs a string input, which is "Gauss". In short the macro declares a function type_() that returns the word "Gauss". The function type is redefined to a virtual function that returns a static const ::Foam::word typeName. I will get back to this in more detail in Section 2.4.4.

Declare run-time constructor selection tables

The code below declares the run-time constructor selection tables

```
————————————————————————————— laplacianScheme.H —————————————————————————————
102      // Declare run-time constructor selection tables
103
104          declareRunTimeSelectionTable
105          (
106              tmp,
107              laplacianScheme,
108              Istream,
109              (const fvMesh& mesh, Istream& schemeData),
110              (mesh, schemeData)
111          );
```

The code creates a run-time selection table for the laplacianScheme base class, which holds constructor pointers on the table. The definition is given by

```
————————————————————————————— runTimeSelectionTables.H —————————————————————————————
47   //- Declare a run-time selection
48   #define declareRunTimeSelectionTable(autoPtr,baseType,argNames,argList,parList)\
49                                                                                  \
50       /* Construct from argList function pointer type */                         \
51       typedef autoPtr<baseType> (*argNames##ConstructorPtr)argList;              \
52                                                                                  \
53       /* Construct from argList function table type */                           \
54       typedef HashTable<argNames##ConstructorPtr, word, string::hash>            \
55           argNames##ConstructorTable;                                            \
56                                                                                  \
57       /* Construct from argList function pointer table pointer */                \
58       static argNames##ConstructorTable* argNames##ConstructorTablePtr_;         \
59                                                                                  \
60       /* Table constructor called from the table add function */                 \
61       static void construct##argNames##ConstructorTables();                      \
62                                                                                  \
63       /* Table destructor called from the table add function destructor */       \
64       static void destroy##argNames##ConstructorTables();                        \
65                                                                                  \
66       /* Class to add constructor from argList to table */                       \
67       template<class baseType##Type>                                             \
68       class add##argNames##ConstructorToTable                                    \
```

35

```
69      {                                                                    \
70      public:                                                              \
71                                                                           \
72          static autoPtr<baseType> New argList                            \
73          {                                                                \
74              return autoPtr<baseType>(new baseType##Type parList);        \
75          }                                                                \
76                                                                           \
77          add##argNames##ConstructorToTable                               \
78          (                                                                \
79              const word& lookup = baseType##Type::typeName               \
80          )                                                                \
81          {                                                                \
82              construct##argNames##ConstructorTables();                   \
83              if (!argNames##ConstructorTablePtr_->insert(lookup, New))    \
84              {                                                            \
85                  std::cerr<< "Duplicate entry " << lookup                \
86                      << " in runtime selection table " << #baseType      \
87                      << std::endl;                                        \
88                  error::safePrintStack(std::cerr);                       \
89              }                                                            \
90          }                                                                \
91                                                                           \
92          ~add##argNames##ConstructorToTable()                            \
93          {                                                                \
94              destroy##argNames##ConstructorTables();                     \
95          }                                                                \
96      };                                                                   \
97                                                                           \
98      /* Class to add constructor from argList to table */                \
99      /* Remove only the entry (not the table) upon destruction */         \
100     template<class baseType##Type>                                      \
101     class addRemovable##argNames##ConstructorToTable                    \
102     {                                                                    \
103         /* retain lookup name for later removal */                      \
104         const word& lookup_;                                            \
105                                                                          \
106     public:                                                              \
107                                                                          \
108         static autoPtr<baseType> New argList                            \
109         {                                                                \
110             return autoPtr<baseType>(new baseType##Type parList);       \
111         }                                                                \
112                                                                          \
113         addRemovable##argNames##ConstructorToTable                      \
114         (                                                                \
115             const word& lookup = baseType##Type::typeName               \
116         )                                                                \
117         :                                                                \
118             lookup_(lookup)                                             \
119         {                                                                \
120             construct##argNames##ConstructorTables();                   \
121             argNames##ConstructorTablePtr_->set(lookup, New);           \
122         }                                                                \
```

```
123                                                                          \
124            ~addRemovable##argNames##ConstructorToTable()                 \
125            {                                                             \
126                if (argNames##ConstructorTablePtr_)                       \
127                {                                                         \
128                    argNames##ConstructorTablePtr_->erase(lookup_);       \
129                }                                                         \
130            }                                                             \
131        };
```

Later when we have declared the derived classes in the run-time selection table, we look in this table every time we implement a term in an equation, because the selector function `New` looks in this run-time selection table for a scheme that matches the given input by the user. If the scheme is not defined, the code will return a list defined by the run-time selection table with the possible options. Note the places with `baseType##Type::typeName`, as this will be defined later, at the construction of the actual scheme.

Constructors

The constructors are declared and defined by the code shown below:

laplacianScheme.H

```
114        // Constructors
115
116            //- Construct from mesh
117            laplacianScheme(const fvMesh& mesh)
118            :
119                mesh_(mesh),
120                tinterpGammaScheme_(new linear<GType>(mesh)),
121                tsnGradScheme_(new correctedSnGrad<Type>(mesh))
122            {}
123
124            //- Construct from mesh and Istream
125            laplacianScheme(const fvMesh& mesh, Istream& is)
126            :
127                mesh_(mesh),
128                tinterpGammaScheme_(nullptr),
129                tsnGradScheme_(nullptr)
130            {
131                tinterpGammaScheme_ = tmp<surfaceInterpolationScheme<GType>>
132                (
133                    surfaceInterpolationScheme<GType>::New(mesh, is)
134                );
135
136                tsnGradScheme_ = tmp<snGradScheme<Type>>
137                (
138                    snGradScheme<Type>::New(mesh, is)
139                );
140            }
141
142            //- Construct from mesh, interpolation and snGradScheme schemes
143            laplacianScheme
144            (
145                const fvMesh& mesh,
146                const tmp<surfaceInterpolationScheme<GType>>& igs,
147                const tmp<snGradScheme<Type>>& sngs
```

```
148              )
149              :
150                  mesh_(mesh),
151                  tinterpGammaScheme_(igs),
152                  tsnGradScheme_(sngs)
153              {}
```

There are three different ways to construct a new object. The difference between the constructors is how the member data `tinterpGammaScheme_` and `tsnGradScheme_` are initialised.

Selectors

A selector function is used in the run-time selection mechanism to construct schemes defined by the user. If the scheme is a valid option according to the run-time selection table, an instance of this scheme is created. If the scheme is invalid the program will abort and throw an error message to the user. The code to declare the selector function `New` is given by

```
                         ─── laplacianScheme.H ───
156      // Selectors
157
158          //- Return a pointer to a new laplacianScheme created on freestore
159          static tmp<laplacianScheme<Type, GType>> New
160          (
161              const fvMesh& mesh,
162              Istream& schemeData
163          );
```

From a C++ perspective this is just another member function. However it is seen that it is static, which means that the function can be called by the class, and then it will apply to all objects of the class.

The definition is found in `laplacianScheme.C` and is given by

```
                         ─── laplacianScheme.C ───
45   template<class Type, class GType>
46   tmp<laplacianScheme<Type, GType>> laplacianScheme<Type, GType>::New
47   (
48       const fvMesh& mesh,
49       Istream& schemeData
50   )
51   {
52       if (fv::debug)
53       {
54           InfoInFunction << "Constructing laplacianScheme<Type, GType>" << endl;
55       }
56
57       if (schemeData.eof())
58       {
59           FatalIOErrorInFunction(schemeData)
60               << "Laplacian scheme not specified" << endl << endl
61               << "Valid laplacian schemes are :" << endl
62               << IstreamConstructorTablePtr_->sortedToc()
63               << exit(FatalIOError);
64       }
65
66       const word schemeName(schemeData);
```

```
67
68        auto cstrIter = IstreamConstructorTablePtr_->cfind(schemeName);
69
70        if (!cstrIter.found())
71        {
72            FatalIOErrorInFunction(schemeData)
73                << "Unknown laplacian scheme "
74                << schemeName << nl << nl
75                << "Valid laplacian schemes are :" << endl
76                << IstreamConstructorTablePtr_->sortedToc()
77                << exit(FatalIOError);
78        }
79
80        return cstrIter()(mesh, schemeData);
81    }
```

The first if statement returns a debug massage if it is turned on. The second if statement is activated if we have not specified the entry for a term used in the application. The code will tell us which scheme is missing and give us our options. In line 66 the scheme name is extracted, and the code looks for the scheme name in the run-time selection table in line 68. The third if statement forces the code execution to stop and returns an error message with a sorted list of the available options found in `IstreamConstructorTablePtr_`, if the user specified scheme is not available. The pointer `IstreamConstructorTablePtr_` is linked to the run-time selection table, which contains information about available schemes. If the scheme name is found, the selector function will return a temporary object of class `tmp` with the run-time selected laplacian scheme.

Member functions

The first member function is an access function, see below:

laplacianScheme.H
```
172            //- Return mesh reference
173            const fvMesh& mesh() const
174            {
175                return mesh_;
176            }
```

It returns a constant reference to the protected data member `mesh_`.
The class then declares a pure virtual and a virtual function `fvmLaplacian` that returns a matrix from the discretization, see below:

laplacianScheme.H
```
178            virtual tmp<fvMatrix<Type>> fvmLaplacian
179            (
180                const GeometricField<GType, fvsPatchField, surfaceMesh>&,
181                const GeometricField<Type, fvPatchField, volMesh>&
182            ) = 0;
183
184            virtual tmp<fvMatrix<Type>> fvmLaplacian
185            (
186                const GeometricField<GType, fvPatchField, volMesh>&,
187                const GeometricField<Type, fvPatchField, volMesh>&
188            );
```

The virtual function is defined in `laplacianScheme.C` as seen from

```
                                laplacianScheme.C
86  template<class Type, class GType>
87  tmp<fvMatrix<Type>>
88  laplacianScheme<Type, GType>::fvmLaplacian
89  (
90      const GeometricField<GType, fvPatchField, volMesh>& gamma,
91      const GeometricField<Type, fvPatchField, volMesh>& vf
92  )
93  {
94      return fvmLaplacian(tinterpGammaScheme_().interpolate(gamma)(), vf);
95  }
```

The virtual function receives a volumetric diffusion field `gamma`, which is interpolated to the cell faces using the user specified interpolation scheme for `gamma`. The interpolated `gamma` field is used in the call to the pure virtual `fvmLaplacian` function that is returned from the virtual function `fvmLaplacian`. Hence it is just a matter of reformatting the input.

The function `fvcLaplacian` is used to evaluate the laplacian term explicitly, and therefore it returns a geometric field. The declaration is seen below:

```
                                laplacianScheme.H
190         virtual tmp<GeometricField<Type, fvPatchField, volMesh>> fvcLaplacian
191         (
192             const GeometricField<Type, fvPatchField, volMesh>&
193         ) = 0;
194
195         virtual tmp<GeometricField<Type, fvPatchField, volMesh>> fvcLaplacian
196         (
197             const GeometricField<GType, fvsPatchField, surfaceMesh>&,
198             const GeometricField<Type, fvPatchField, volMesh>&
199         ) = 0;
200
201         virtual tmp<GeometricField<Type, fvPatchField, volMesh>> fvcLaplacian
202         (
203             const GeometricField<GType, fvPatchField, volMesh>&,
204             const GeometricField<Type, fvPatchField, volMesh>&
205         );
```

The declaration forces the derived classes to redefine the two pure virtual functions, where there is an option to evaluate a laplacian term explicitly with or without the diffusion coefficient `gamma`. The virtual function `fvcLaplacian` serves the same purpose as the virtual function `fvmLaplacian`. The definition of `fvcLaplacian` is found in `laplacianScheme.C`, see below:

```
                                laplacianScheme.C
98   template<class Type, class GType>
99   tmp<GeometricField<Type, fvPatchField, volMesh>>
100  laplacianScheme<Type, GType>::fvcLaplacian
101  (
102      const GeometricField<GType, fvPatchField, volMesh>& gamma,
103      const GeometricField<Type, fvPatchField, volMesh>& vf
104  )
105  {
106      return fvcLaplacian(tinterpGammaScheme_().interpolate(gamma)(), vf);
107  }
```

The cell-centred `gamma` field is interpolated to the cell faces, and a call to the second `fvcLaplacian` pure virtual function is returned with the interpolated `gamma` field.

Add the patch constructor functions to the hash tables

After the `laplacianScheme` class declaration, we find two macro definitions, seen below:

```
                                      ── laplacianScheme.H ──
221 #define makeFvLaplacianTypeScheme(SS, GType, Type)                          \
222     typedef Foam::fv::SS<Foam::Type, Foam::GType> SS##Type##GType;          \
223     defineNamedTemplateTypeNameAndDebug(SS##Type##GType, 0);                \
224                                                                             \
225     namespace Foam                                                          \
226     {                                                                       \
227         namespace fv                                                        \
228         {                                                                   \
229             typedef SS<Type, GType> SS##Type##GType;                        \
230                                                                             \
231             laplacianScheme<Type, GType>::                                  \
232                 addIstreamConstructorToTable<SS<Type, GType>>               \
233                 add##SS##Type##GType##IstreamConstructorToTable_;           \
234         }                                                                   \
235     }
236
237
238 #define makeFvLaplacianScheme(SS)                                           \
239                                                                             \
240 makeFvLaplacianTypeScheme(SS, scalar, scalar)                              \
241 makeFvLaplacianTypeScheme(SS, symmTensor, scalar)                          \
242 makeFvLaplacianTypeScheme(SS, tensor, scalar)                             \
243 makeFvLaplacianTypeScheme(SS, scalar, vector)                             \
244 makeFvLaplacianTypeScheme(SS, symmTensor, vector)                         \
245 makeFvLaplacianTypeScheme(SS, tensor, vector)                             \
246 makeFvLaplacianTypeScheme(SS, scalar, sphericalTensor)                    \
247 makeFvLaplacianTypeScheme(SS, symmTensor, sphericalTensor)                \
248 makeFvLaplacianTypeScheme(SS, tensor, sphericalTensor)                    \
249 makeFvLaplacianTypeScheme(SS, scalar, symmTensor)                         \
250 makeFvLaplacianTypeScheme(SS, symmTensor, symmTensor)                     \
251 makeFvLaplacianTypeScheme(SS, tensor, symmTensor)                         \
252 makeFvLaplacianTypeScheme(SS, scalar, tensor)                             \
253 makeFvLaplacianTypeScheme(SS, symmTensor, tensor)                         \
254 makeFvLaplacianTypeScheme(SS, tensor, tensor)
```

The first macro definition `makeFvLaplacianTypeScheme(SS, GType, Type)` adds a single patch constructor function to a hash table. Again this is related to the run-time selection mechanism. In the beginning of `laplacianScheme.H`, the tables where declared, and now the code defines macros that will be used by the derived classes to add the derived class scheme options to the run-time selection table. The macro `makeFvLaplacianScheme(SS)` calls `makeFvLaplacianTypeScheme` for all the possible combinations of the template parameters `Type` and `GType`. The field `gamma` can be a `scalar`, `symmTensor` or `tensor`. The created `fvMatrix` can be a matrix of `scalar`, `vector`, `sperical Tensor`, `symmTensor` or `tensor`. The input `SS` is a scheme string with the name of the class derived from `laplacianScheme`.

End of laplacianScheme.H

At the end of the declaration we find the code:

```
                          ──────── laplacianScheme.H ────────
258   #ifdef NoRepository
259       #include "laplacianScheme.C"
260   #endif
```

For templated classes, the source code is included at the end of the declaration, if the repository is not already known, because a templated class is not compiled. To compile a templated class, we need to supply valid template parameters.

2.3.3 Description of gaussLaplacianScheme

In the previous section, we learned about the abstract base class `laplacianScheme`, which enforces a certain code structure to supply some functionality to all the classes derived from it. We will now take a look at the derived class `gaussLaplacianScheme`, which is the only existing class derived from `laplacianScheme` in the official release of OpenFOAM-1906, i.e. Gauss discretization is the only option for the laplacian operator. The section will only cover code, that is not related to theory.

The first section of code in `gaussLaplacianScheme.H` is

```
                          ──────── gaussLaplacianScheme.H ────────
40    #ifndef gaussLaplacianScheme_H
41    #define gaussLaplacianScheme_H
42
43    #include "laplacianScheme.H"
44
45    // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
46
47    namespace Foam
48    {
49
50    // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
51
52    namespace fv
53    {
54
55    /*---------------------------------------------------------------------------*\
56                        Class gaussLaplacianScheme Declaration
57    \*---------------------------------------------------------------------------*/
58
59    template<class Type, class GType>
60    class gaussLaplacianScheme
61    :
62        public fv::laplacianScheme<Type, GType>
63    {
```

The class is guarded against multiple inclusions in the code. The class needs to know about the `laplacianScheme` class declaration. The code enters first the namespace `Foam` and next namespace `fv`. The class declaration starts at line 59 with the template specification, the class name at line 60 and finally it is specified that the class is derived publicly from `laplacianScheme`. The class declaration is then performed after the open bracket at line 63.

Remove defaults

When a class is derived from a base class it inherits a lot of functionality from the base classes (The base class can also inherit from other classes). In this case there is a default copy and assignment constructor, which is deleted as seen below:

```
                                    gaussLaplacianScheme.H
72          //- No copy construct
73          gaussLaplacianScheme(const gaussLaplacianScheme&) = delete;
74
75          //- No copy assignment
76          void operator=(const gaussLaplacianScheme&) = delete;
```

Runtime type name

The next piece of code defines the keyword for our scheme, which we need to specify in our case directory file system/fvSchemes. The code is:

```
                                    gaussLaplacianScheme.H
81       //- Runtime type information
82       TypeName("Gauss");
```

There is a more detailed discussion of the macro TypeName in Section 2.4.4.

Constructors

The declaration and definition of the class constructors is seen below:

```
                                    gaussLaplacianScheme.H
87          //- Construct null
88          gaussLaplacianScheme(const fvMesh& mesh)
89          :
90              laplacianScheme<Type, GType>(mesh)
91          {}
92
93          //- Construct from Istream
94          gaussLaplacianScheme(const fvMesh& mesh, Istream& is)
95          :
96              laplacianScheme<Type, GType>(mesh, is)
97          {}
98
99          //- Construct from mesh, interpolation and snGradScheme schemes
100         gaussLaplacianScheme
101         (
102             const fvMesh& mesh,
103             const tmp<surfaceInterpolationScheme<GType>>& igs,
104             const tmp<snGradScheme<Type>>& sngs
105         )
106         :
107             laplacianScheme<Type, GType>(mesh, igs, sngs)
108         {}
```

There are the same three constructors, as for the base class, and it is seen that constructor calls the base class constructor. No additional input is initialised in the initialiser list. If we want some private member data in our class, we can initialise them after the initialising call to the base class constructor.

Emulate partial-specialisation

After the class declaration in gaussLaplacianScheme.H, we find the macro definition shown below:

```
                                    gaussLaplacianScheme.H
147  #define defineFvmLaplacianScalarGamma(Type)                              \
148                                                                           \
```

```
149   template<>                                                            \
150   tmp<fvMatrix<Type>> gaussLaplacianScheme<Type, scalar>::fvmLaplacian   \
151   (                                                                     \
152       const GeometricField<scalar, fvsPatchField, surfaceMesh>&,        \
153       const GeometricField<Type, fvPatchField, volMesh>&                \
154   );                                                                    \
155                                                                         \
156   template<>                                                            \
157   tmp<GeometricField<Type, fvPatchField, volMesh>>                      \
158   gaussLaplacianScheme<Type, scalar>::fvcLaplacian                      \
159   (                                                                     \
160       const GeometricField<scalar, fvsPatchField, surfaceMesh>&,        \
161       const GeometricField<Type, fvPatchField, volMesh>&                \
162   );
```

The macro function declares the functions `fvmLaplacian` and `fvcLaplacian` for a scalar diffusion coefficient γ, as function of the template parameter `Type`. After the macro has been defined, it is called, see below.

gaussLaplacianScheme.H
```
165   defineFvmLaplacianScalarGamma(scalar);
166   defineFvmLaplacianScalarGamma(vector);
167   defineFvmLaplacianScalarGamma(sphericalTensor);
168   defineFvmLaplacianScalarGamma(symmTensor);
169   defineFvmLaplacianScalarGamma(tensor);
```

Now the code declares five different versions of `fvmLaplacian` and `fvcLaplacian` according to the five different finite volume matrix template options. This is called emulation of a partial specialisation. The definition of these functions are performed in the file `gaussLaplacianSchemes.C`, where it will become clear, why this specialisation is made.

End of gaussLaplacianScheme.H

The header file ends with the code:

gaussLaplacianScheme.H
```
172   // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
173
174   } // End namespace fv
175
176   // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
177
178   } // End namespace Foam
179
180   // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
181
182   #ifdef NoRepository
183       #include "gaussLaplacianScheme.C"
184   #endif
185
186   // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
187
188   #endif
```

The namespaces are closed, and the source file is included, if it is needed. Finally the include guarding if statement is closed.

gaussLaplacianSchemes.C

This file is compiled by the finiteVolume library in OpenFOAM. The file adds the derived class schemes to the run-time selection table, and defines the implementation of the partial specialisations for the cases where γ is of type `scalar`. The file starts by including the necessary declarations.

```
                    ─── gaussLaplacianSchemes.C ───
28   #include "gaussLaplacianScheme.H"
29   #include "fvMesh.H"
```

Then a call is made to the macro `makeFvLaplacianScheme` with `SS = gaussLaplacianScheme`, as given by

```
                    ─── gaussLaplacianSchemes.C ───
33   makeFvLaplacianScheme(gaussLaplacianScheme)
```

`makeFvLaplacianScheme` is defined in base class header file `laplacianScheme.H`, and it adds all the possible combinations of the template parameters to the run-time selection tables.

The next piece of code is

```
                    ─── gaussLaplacianSchemes.C ───
35   #define declareFvmLaplacianScalarGamma(Type)                                    \
36                                                                                   \
```

The macro `declareFvmLaplacianScalarGamma(Type)` is defined and it contains the implementation for the template specialisations of `fvmLaplacian` and `fvcLaplacian` when the diffusion coefficient is a surface scalar field. After the macro definition, it is used to define the `fvmLaplacian` and `fvcLaplacian` function for each possible finite volume matrix template parameter `Type`, as seen below:

```
                    ─── gaussLaplacianSchemes.C ───
116  declareFvmLaplacianScalarGamma(scalar);
117  declareFvmLaplacianScalarGamma(vector);
118  declareFvmLaplacianScalarGamma(sphericalTensor);
119  declareFvmLaplacianScalarGamma(symmTensor);
120  declareFvmLaplacianScalarGamma(tensor);
```

2.4 Description of matrix assembly in OpenFOAM

The matrix discretization in OpenFOAM is based on ldu-adressing. A description of the matrix assembly is found in the book by Moukalled et al. [3, p. 191-202] and in the slideshow presentation by Dr. Thorsten Grahs [4].

A matrix made with the `fvMatrix` class in OpenFOAM is based on the class `lduMatrix`. The class is based on the concept that a matrix can be separated into the lower triangle with coefficients below the diagonal (l = lower), the coefficients in the diagonal (d = diagonal) and the upper triangle with coefficients above the diagonal (u = upper). Equation (2.44) shows the ldu matrix separation with color coding.

$$\mathbf{A} = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & \cdots & a_{0,j} \\ a_{1,0} & a_{1,1} & a_{1,2} & \cdots & a_{1,j} \\ a_{2,0} & a_{2,1} & a_{2,2} & \cdots & a_{2,j} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{i,0} & a_{i,1} & a_{i,2} & \cdots & a_{i,j} \end{bmatrix} \tag{2.44}$$

The blue coefficients are the lower matrix off diagonal elements, the red coefficients are the upper matrix off diagonal elements and the black coefficients are the matrix diagonal. The matrix is treated as a sparse matrix, where only the non-zero components are saved in three vectors **l**, **d** and **u** representing the lower off diagonal, the diagonal and the upper off diagonal. Lets consider the example in Figure 2.5, which shows a 2D domain with 9 cells and the numbering of the internal faces and boundary faces. I have left out the faces on the front and the back, they would be in patch 5 and 6.
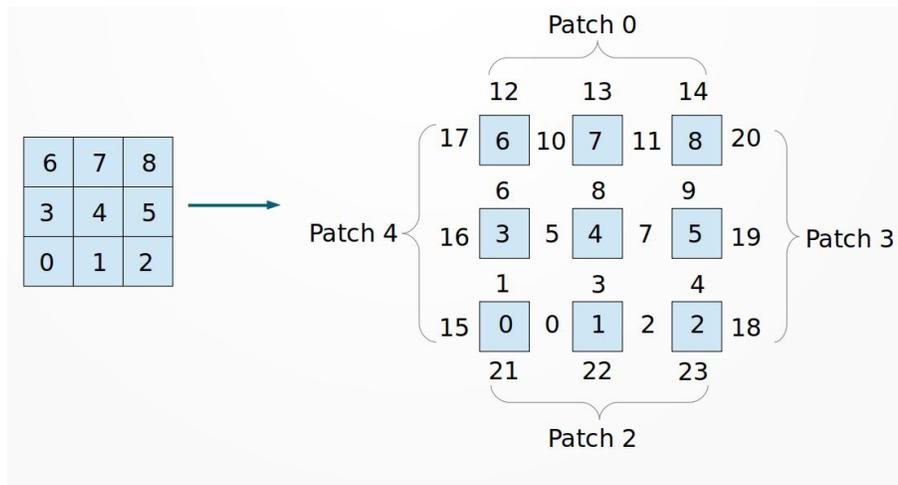

Figure 2.5: Example domain with 9 cells, 11 internal faces and 30 boundary faces. The top, bottom, left and right side of the domain is 4 patches, and each patch has 3 boundary faces. The remaining 18 boundary faces, not included in the illustration, are found on the back side and front side of the cells.

First I will discuss how the internal faces are handled, and then the boundary faces.

2.4.1 Internal cell faces

OpenFOAM numbers the internal faces according to ascending cell number. Face number 0 is between cell 0 and 1, and the lowest cell number owns face 0, hence cell 0 is the owner of face 0 and

cell 1 is the neighbour of face 0. The system matrix for this mesh is presented in Equation (2.45).

$$\mathbf{A} = \begin{bmatrix} a_{0,0} & a_{0,1} & 0 & a_{0,3} & 0 & 0 & 0 & 0 & 0 \\ a_{1,0} & a_{1,1} & a_{1,2} & 0 & a_{1,4} & 0 & 0 & 0 & 0 \\ 0 & a_{2,1} & a_{2,2} & 0 & 0 & a_{2,5} & 0 & 0 & 0 \\ a_{3,0} & 0 & 0 & a_{3,3} & a_{3,4} & 0 & a_{3,6} & 0 & 0 \\ 0 & a_{4,1} & 0 & a_{4,3} & a_{4,4} & a_{4,5} & 0 & a_{4,7} & 0 \\ 0 & 0 & a_{5,2} & 0 & a_{5,4} & a_{5,5} & 0 & 0 & a_{5,8} \\ 0 & 0 & 0 & a_{6,3} & 0 & 0 & a_{6,6} & a_{6,7} & 0 \\ 0 & 0 & 0 & 0 & a_{7,4} & 0 & a_{7,6} & a_{7,7} & a_{7,8} \\ 0 & 0 & 0 & 0 & 0 & a_{8,5} & 0 & a_{8,7} & a_{8,8} \end{bmatrix} \tag{2.45}$$

Each row in the matrix corresponds to a cell in the mesh. So for instance in cell 3 there are 3 internal faces. $a_{3,0}$ comes from face 1 looking from cell 3 to 0, $a_{3,3}$ comes from cell 3, $a_{3,4}$ comes from face 5 looking from cell 3 to 4 and $a_{3,6}$ comes from face 6 looking from cell 3 to 6.

The non-zero coefficients are saved in three data members of the lduMatrix class, which are `lower()`, `diag()` and `upper()`. For this example matrix the coefficients would be arranged as

$$\texttt{lower()} = (a_{1,0}\ a_{3,0}\ a_{2,1}\ a_{4,1}\ a_{5,2}\ a_{4,3}\ a_{6,3}\ a_{5,4}\ a_{7,4}\ a_{8,5}\ a_{7,6}\ a_{8,7}) \tag{2.46}$$

$$\texttt{diag()} = (a_{0,0}\ a_{1,1}\ a_{2,2}\ a_{3,3}\ a_{4,4}\ a_{5,5}\ a_{6,6}\ a_{7,7}\ a_{8,8}) \tag{2.47}$$

$$\texttt{upper()} = (a_{0,1}\ a_{0,3}\ a_{1,2}\ a_{1,4}\ a_{2,5}\ a_{3,4}\ a_{3,6}\ a_{4,5}\ a_{4,7}\ a_{5,8}\ a_{6,7}\ a_{7,8}) \tag{2.48}$$

The off-diagonal coefficients are sorted with respect to ascending face index 0 .. 11. The diagonal coefficients are sorted with respect to the cell index 0 .. 8.

The sub-indices for the coefficients a are the location in the matrix (row,column). They are saved in two vectors `lowerAddr()` and `upperAddr()`. `lowerAddr()` keeps the sub-indices that describe the matrix row number (first sub-index) for all elements from the lower triangle `lower()`. `upperAddr()` keeps the indices that describe the matrix row number (first sub-index) for all elements from the upper triangle `upper()`.

The matrix position of the elements from the lower triangle of the matrix, a, is described by

$$a_{\texttt{lowerAddr()},\texttt{upperAddr()}} \tag{2.49}$$

Likewise the position of the elements from the upper triangle of the matrix, a, is described by

$$a_{\texttt{upperAddr()},\texttt{lowerAddr()}} \tag{2.50}$$

So the matrix is forced to have a symmetric structure, but it does not have to be symmetric in terms of the coefficient values.

The last addressing vector is `ownerStartAddr()`, which keeps the element position at which a new row starts in `upper()` and at which a new column starts in `lower()`. So for the example the addressing vectors are

$$\texttt{upperAddr()} = (0,\ 0,\ 1,\ 1,\ 2,\ 3,\ 3,\ 4,\ 4,\ 5,\ 6,\ 7)$$

$$\texttt{lowerAddr()} = (1,\ 3,\ 2,\ 4,\ 5,\ 4,\ 6,\ 5,\ 7,\ 8,\ 7,\ 8)$$

$$\texttt{ownerStartAddr()} = (0,\ 2,\ 4,\ 5,\ 7,\ 9,\ 10,\ 11)$$

So `upperAddr()` is the first index of the elements in `upper()` and the second index in `lower()`. `lowerAddr()` is the first index of the elements in `lower()` and the second index in `upper()`.
It important to understand that `upper()` contains the coefficients that belongs to the owner cell of

each face. Likewise `lower()` contains the coefficients that belongs to the neighbour cell of each face. Furthermore we can see that an off diagonal coefficient is non-zero when a face is shared between an owner and neighbour cell corresponding to the position in the matrix.

For the addressing vectors it is important to realise that `upperAddr()` contains the owner cell index for each face index sorted in ascending order. Likewise `lowerAddr()` contains the neighbour cell index for each face index sorted in ascending order.

The discretization of a term in each cell will give a contribution to the diagonal, but we may also have contributions to the matrix source vector. These contributions are added to `source()`, which has an entry for each cell.

2.4.2 Boundary cell faces

The boundary cell faces are treated differently from the internal faces. The coefficients from the boundary face that goes into the system matrix is saved in `internalCoeffs()[patchI][patchFaceI]`, and the contribution source vector is saved in `boundaryCoeffs()[patchI][patchFaceI]`. `patchI` refers to the patch indices and a patch is a group of faces that represent a part of the domain boundary for instance the bottom of the domain. The patches are defined during the meshing process. `patchFaceI` is the index of each face in a patch, so for instance Patch 2 contains the face indices `faceI = (21 22 23)` and the corresponding patch face index is then `patchFaceI = (0 1 2)`. This is important to remember!

A boundary face has no neighbour cell, only a owner cell. This means that the boundary faces in each cell only contributes to the discretization of the governing equation in the cell to which the boundary face belongs, i.e. the diagonal matrix coefficients. The exception is coupled boundaries, which do have a cell on both sides of the boundary.

2.4.3 Matrix assembly of laplacian operator on orthogonal mesh

The discretization in OpenFOAM is based on the faces instead of the cells, so for each internal face we have to specify the contribution associated with the owner cell and the neighbour cell. For each cell we have to specify the contribution to the diagonal and the matrix source vector. For the boundary faces we have to assign the contribution of the matrix diagonal and the matrix source vector. When we have done this for all faces, the matrix is assembled.

Let us take a look at how the matrix is assembled, when the laplacian operator is discretized with the standard Gauss scheme in OpenFOAM. The laplacian operates on a scalar field ϕ which is associated with scalar diffusion field γ. The mesh is assumed to be orthogonal, as illustrated in Figure 2.1 case a), where vector from owner cell centre to neighbour cell centre passes through the shared face centre. The discretization valid for orthogonal meshes when looking from owner cell (P) towards neighbour cell (N) is given by

$$\int_{V_P} \nabla \bullet (\gamma \nabla \phi) \; \mathrm{d}V = \sum_{f=1}^{n_f} \gamma_f |\mathbf{S}_f| \frac{\phi_N - \phi_P}{|\mathbf{d}|}$$

$$= \sum_{f=1}^{n_f} \gamma_f |\mathbf{S}_f| \frac{1}{|\mathbf{d}|} (\phi_N - \phi_P)$$

$$= \sum_{f=1}^{n_f} \underbrace{\gamma_f |\mathbf{S}_f| \frac{1}{|\mathbf{d}|}}_{a_{P,N}} \phi_N \underbrace{-\gamma_f |\mathbf{S}_f| \frac{1}{|\mathbf{d}|}}_{a_{P,P}} \phi_P \qquad (2.51)$$

The discretization looking from neighbour towards owner cell is given by

$$
\int_{V_N} \nabla \bullet (\gamma \nabla \phi) \, \mathrm{d}V = \sum_{f=1}^{n_f} \gamma_f(|\mathbf{S}_f|) \frac{\phi_P - \phi_N}{|\mathbf{d}|}
$$
$$
= \sum_{f=1}^{n_f} \gamma_f(-|\mathbf{S}_f|) \frac{1}{|\mathbf{d}|} (\phi_N - \phi_P)
$$
$$
= \sum_{f=1}^{n_f} \underbrace{-\gamma_f|\mathbf{S}_f| \frac{1}{|\mathbf{d}|}}_{a_{N,N}} \phi_N + \underbrace{\gamma_f|\mathbf{S}_f| \frac{1}{|\mathbf{d}|}}_{a_{N,P}} \phi_P \qquad (2.52)
$$

Note how the sign of the surface area vector magnitude is changed in the second line of Equation (2.51). This treatment of the surface area vector is implicitly accounted for in the OpenFOAM summations. The sub-indices of the matrix coefficients a gives the matrix row as the first index and matrix column as the second index. Sub-index N is the neighbour cell index and P is the owner cell index. It is important to note that the off-diagonal coefficients $a_{P,N}$ and $a_{N,P}$, arising from the discretization at each face, are equal to each other. Combined with the fact that boundary conditions only contribute to the matrix diagonal, it is now known that the **resulting matrix** from the discretization of the entire domain **is symmetric**.

The code which implements the presented discretization is now described in relation to theory. First the code constructs a new matrix object in the lines

```
gaussLaplacianScheme.C
55    tmp<fvMatrix<Type>> tfvm
56    (
57        new fvMatrix<Type>
58        (
59            vf,
60            deltaCoeffs.dimensions()*gammaMagSf.dimensions()*vf.dimensions()
61        )
62    );
63    fvMatrix<Type>& fvm = tfvm.ref();
```

where a new matrix object is created at some location in memory, which is managed by the tmp class object `tfvm`. A reference to the matrix object is saved in the variable `fvm`. The link between the theoretical expression in Equation (2.51) and variable names in the code is presented in Table 2.1.

Table 2.1: Link between theoretical terms and variable names in OpenFOAM.

Code variable	Theoretical term
`vf`	ϕ
`deltaCoeffs`	$\frac{|\mathbf{n}_f|}{|\mathbf{d}|} = \frac{1}{|\mathbf{d}|}$
`gammaMagSf`	$\gamma_f|\mathbf{S}_f|$

The matrix assembly in the original `gaussLaplacianScheme` class is performed by the lines

```
gaussLaplacianScheme.C
65    fvm.upper() = deltaCoeffs.primitiveField()*gammaMagSf.primitiveField();
66    fvm.negSumDiag();
```

where `fvm.upper()` contains the off diagonal coefficients $a_{P,N}$ related with the owner cell (P) of each internal face (f). `fvm.negSumDiag()` is a function that computes the diagonal coefficients

based on the off-diagonal coefficients. The contribution to the diagonal coefficient in each cell from the internal faces can be computed by

$$a_C = -\sum_{f=1}^{n_f} a_{C,N} \tag{2.53}$$

where the summation run through the internal faces of a cell.

This fact should be clear from Equation (2.51) and (2.52), where the off-diagonal term is identical to the diagonal term except for change in sign. **It is noted that `fvm.lower()` is not assigned, and this tells the code that the matrix is symmetric. It is important to avoid accessing the field `fvm.lower()`, because this will turn on a switch which makes the code think that the matrix is asymmetric.**

The current implementation does not facilitate a direct link between theory and implementation.

Internal faces

All the pieces of code presented in this section is located in the same file. The matrix assembly with contribution from internal faces can be programmed starting with a loop given by

```
for(label facei=0; facei<fvm.lduAddr().upperAddr().size(); facei++)
{
```

which runs through all internal faces of the mesh from 0 to the number of internal faces. In the next section of code I define the cell index of the owner index of `facei` using the matrix ldu adressing vector `upperAddr()`, and the neighbour cell index to `facei` using ldu adressing vector `lowerAddr()`.

```
// Cell indices P (owner) and N (neighbour)
label owner = fvm.lduAddr().upperAddr()[facei];      // P index
label neighbour = fvm.lduAddr().lowerAddr()[facei];  // N index
```

Now we can start assigning the coefficients presented in Equation (2.51) and (2.52). The next section of code shows the assignment of $a_{P,P}$ from Equation (2.51)

```
// Assign contributions to the diagonal matrix coefficient:
    //- Looking from P -> N (a_(P,P))
    fvm.diag()[owner] -= deltaCoeffs.primitiveField()[facei]
                      * gammaMagSf.primitiveField()[facei];
```

The coefficient is subtracted from the element in the diagonal vector `diag()` at element with index `owner`. The next code listing shows similarly how $a_{N,N}$ is subtracted

```
    //- Looking from N -> P (a_(N,N))
    fvm.diag()[neighbour] -= deltaCoeffs.primitiveField()[facei]
                          * gammaMagSf.primitiveField()[facei];
```

from element with index `neighbour` in the diagonal vector `diag()`. The next code listing shows how the off diagonal coefficient $a_{P,N}$ is added to the matrix.

```
// Assign contributions to off diagonal matrix coefficient:
        //- Looking from P -> N (a_(P,N))
        //  Assigning matrix coefficient in upper triangle.
        fvm.upper()[facei] += deltaCoeffs.primitiveField()[facei]
                          * gammaMagSf.primitiveField()[facei];
```

The off diagonal coefficient $a_{P,N}$ is associated with cell P and it is therefore added to element `facei` in vector `upper()`. The next listing of code is only given to show how the lower matrix triangle would be assembled, if the matrix had been asymmetric.

```
/* ============================================================================= *\
|    NOTE: THIS PART SHOULD ONLY BE ACTIVE IF YOU HAVE AN ASYMMETRIC MATRIX!   |
|                //- Looking from N -> P (a_(N,P))                             |
|                //  Assigning matrix coefficient in lower triangle.           |
|                fvm.lower()[facei] += deltaCoeffs.primitiveField()[facei]     |
|                                    * gammaMagSf.primitiveField()[facei];     |
\* ============================================================================= */
```

The off diagonal coefficient $a_{N,P}$ is associated with cell N and it is therefore added to element `facei` in vector `lower()`. Finally the loop over internal faces is closed as given by

```
    }
```

Boundary faces

The matrix contribution from the boundary faces is handled separately. The first listing of code initiates a loop over the patches of the field `vf` which corresponded to ϕ in the theory.

```
                              gaussLaplacianScheme.C
68      forAll(vf.boundaryField(), patchi)
69      {
```

The loop is specified using the `forAll` macro. `boundaryField()` contains a `fvPatchField<Type>` field for each patch. Each patch is linked to a boundary condition specification in the files of the standard case directory folder `0/`. The loop runs over the patch index `patchi`. The next section of code defines constant references to the boundary fields of `patchi` from the fields `vf`, `gammaMagSf` and `deltaCoeffs`.

```
                              gaussLaplacianScheme.C
70          const fvPatchField<Type>& pvf = vf.boundaryField()[patchi];
71          const fvsPatchScalarField& pGamma = gammaMagSf.boundaryField()[patchi];
72          const fvsPatchScalarField& pDeltaCoeffs =
73              deltaCoeffs.boundaryField()[patchi];
```

The reference `pvf` refers to the boundary field of `vf` for `patchi`. `pGamma` refers to the values of `gammaMagSf` at the boundary face centres in `patchi`. `pDeltaCoeffs` refers to the values of `deltaCoeffs` at the boundary face centres in `patchi`.

Figure 2.6 shows how the boundary condition specification can separated in two cases
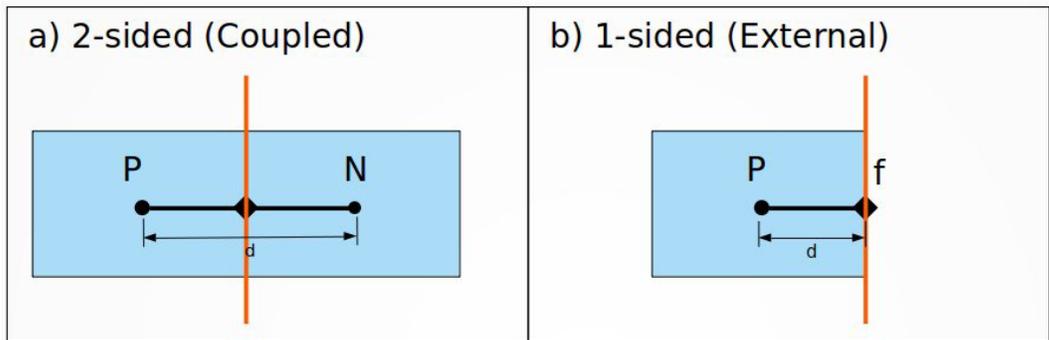
Figure 2.6: Illustration of a coupled and external boundary condition.

which are a) a 2-sided boundary face (coupled) b) a 1-sided boundary face (External). An example of a coupled boundary condition is the cyclic boundary condition, which can be used to create a cyclic domain where the outlet of the domain is linked to the inlet of the domain. The 1-sided external boundary condition is the traditional domain boundary condition, where we do not have a cell on the other side. Therefore we have a Dirichlet, Neumann or Robin condition to determine the value of the dependent field at the boundary face. The next listing shows how the 2-sided coupled boundary condition is implemented as

```
                         _____ gaussLaplacianScheme.C _____
75            if (pvf.coupled())
76            {
77                fvm.internalCoeffs()[patchi] =
78                    pGamma*pvf.gradientInternalCoeffs(pDeltaCoeffs);
79                fvm.boundaryCoeffs()[patchi] =
80                    -pGamma*pvf.gradientBoundaryCoeffs(pDeltaCoeffs);
81            }
```

and the one-sided boundary condition is implemented by

```
                         _____ gaussLaplacianScheme.C _____
82            else
83            {
84                fvm.internalCoeffs()[patchi] = pGamma*pvf.gradientInternalCoeffs();
85                fvm.boundaryCoeffs()[patchi] = -pGamma*pvf.gradientBoundaryCoeffs();
86            }
87        }
```

For a fixed value boundary condition, we need to know the value at the boundary face centre point and this value is denoted ϕ_b. The discretization for a cell with one of more boundary faces can then be written as

$$
\int_{V_P} \nabla \bullet (\gamma \nabla \phi) \ \mathrm{d}V = \sum_{f=1}^{n_f} \gamma_f |\mathbf{S}_f| \mathbf{n}_f \bullet (\nabla \phi)_f
$$

$$
= \underbrace{\sum_{f,i=1}^{n_{f,i}} \gamma_f |\mathbf{S}_f| \frac{\phi_N - \phi_P}{|\mathbf{d}|}}_{Internal\ faces} + \underbrace{\sum_{f,b=1}^{n_{f,b}} \gamma_f |\mathbf{S}_f| \frac{\phi_b - \phi_P}{|\mathbf{d_n}|}}_{Boundary\ faces} \tag{2.54}
$$

where $\mathbf{d_n}$ is the projection of $\mathbf{d}$ on the direction of the surface normal vector $\mathbf{n}_f$. The implicit contributions related to ϕ_P and ϕ_N is added to the system matrix as previously described and the explicit contribution related to ϕ_b is added to the source vector. For a single boundary face we have

$$
\gamma_f |\mathbf{S}_f| \frac{\phi_b - \phi_P}{|\mathbf{d_n}|} = \underbrace{\gamma_f |\mathbf{S}_f|}_{\text{pGamma}} \underbrace{\frac{1}{|\mathbf{d_n}|}}_{\text{pvf.gradientBoundaryCoeffs()}} \phi_b + \underbrace{\gamma_f |\mathbf{S}_f|}_{\text{pGamma}} \underbrace{-\frac{1}{|\mathbf{d_n}|}}_{\text{pvf.gradientInternalCoeffs()}} \phi_P \tag{2.55}
$$

The negative sign in

```
fvm.boundaryCoeffs()[patchi] = -pGamma*pvf.gradientBoundaryCoeffs();
```

is needed because the term is a source term and therefore moved to the right-hand side of the equal sign in matrix system $\mathbf{Ax} = \mathbf{b}$.

The fixed value boundary condition is called `fixedValue` in OpenFOAM, and we can identify the coefficients returned by `gradientBoundaryCoeffs()` and `gradientInternalCoeffs()` in the file `fixedValueFvPatchField.C` which reads

```
                        ── fixedValueFvPatchField.C ──
139  template<class Type>
140  Foam::tmp<Foam::Field<Type>>
141  Foam::fixedValueFvPatchField<Type>::gradientInternalCoeffs() const
142  {
143      return -pTraits<Type>::one*this->patch().deltaCoeffs();
144  }
145
146
147  template<class Type>
148  Foam::tmp<Foam::Field<Type>>
149  Foam::fixedValueFvPatchField<Type>::gradientBoundaryCoeffs() const
150  {
151      return this->patch().deltaCoeffs()*(*this);
152  }
```

From the code it is seen that `gradientInternalCoeffs()` returns

`-pTraits<Type>::one*this->patch().deltaCoeffs();`

which is exactly equal to

$$\underbrace{-1}_{\texttt{-pTraits<Type>::one}} \quad \underbrace{\frac{1}{|\mathbf{d_n}|}}_{\texttt{this->patch().deltaCoeffs()}}$$

`gradientBoundaryCoeffs()` returns

`this->patch().deltaCoeffs()*(*this);`

which corresponds to

$$\underbrace{\frac{1}{|\mathbf{d_n}|}}_{\texttt{this->patch().deltaCoeffs()}} \quad \underbrace{\phi_b}_{\texttt{(*this)}}$$

2.4.4 Find implementation of Gauss Laplacian scheme options

When we specify the keyword `Gauss` for a laplacianScheme in `fvSchemes`, it works, because the keyword is defined in the declaration for the Gauss Laplacian scheme. The code is seen below:

```
                        ── gaussLaplacianScheme.H ──
81       //- Runtime type information
82       TypeName("Gauss");
```

`TypeName` is a macro that requires the keyword as a string. The definition of the macro is found in `typeInfo.H`, see below:

```
                        ── typeInfo.H ──
71  //- Declare a ClassName() with extra virtual type info
72  #define TypeName(TypeNameString)                                      \
73      ClassName(TypeNameString);                                        \
74      virtual const word& type() const { return typeName; }
```

At line 74 it is seen that the function `type()` is redefined, as required in the abstract base class `laplacianScheme`, see the code below:

```
                         ─── laplacianScheme.H ───
96  public:
97
98      //- Runtime type information
99      virtual const word& type() const = 0;
```

The redefined `type()` function returns the variable `typeName`, which is declared by a macro call in the macro `className`, seen below:

```
                         ─── className.H ───
65  //- Add typeName information from argument \a TypeNameString to a class.
66  //  Also declares debug information.
67  #define ClassName(TypeNameString)                                          \
68      ClassNameNoDebug(TypeNameString);                                      \
69      static int debug
```

The macro `ClassNameNoDebug` declares `typeName` and the macro definition is shown below:

```
                         ─── className.H ───
39  //- Add typeName information from argument \a TypeNameString to a class.
40  //  Without debug information
41  #define ClassNameNoDebug(TypeNameString)                                   \
42      static const char* typeName_() { return TypeNameString; }              \
43      static const ::Foam::word typeName
```

To summarize the function `type()` has been changed from a pure virtual function to a virtual function that returns `typeName` which is declared as a `static const ::Foam::word`.

The function `typeName_()` is called, when the code calls the macro `makeFvLaplacianScheme` in `gaussLaplacianSchemes.C`

```
                         ─── gaussLaplacianSchemes.C ───
33  makeFvLaplacianScheme(gaussLaplacianScheme)
```

The definition of the macro `makeFvLaplacianScheme` is

```
                         ─── laplacianScheme.H ───
238  #define makeFvLaplacianScheme(SS)                                         \
239                                                                            \
240  makeFvLaplacianTypeScheme(SS, scalar, scalar)                            \
241  makeFvLaplacianTypeScheme(SS, symmTensor, scalar)                        \
242  makeFvLaplacianTypeScheme(SS, tensor, scalar)                            \
243  makeFvLaplacianTypeScheme(SS, scalar, vector)                            \
244  makeFvLaplacianTypeScheme(SS, symmTensor, vector)                        \
245  makeFvLaplacianTypeScheme(SS, tensor, vector)                            \
246  makeFvLaplacianTypeScheme(SS, scalar, sphericalTensor)                   \
247  makeFvLaplacianTypeScheme(SS, symmTensor, sphericalTensor)               \
248  makeFvLaplacianTypeScheme(SS, tensor, sphericalTensor)                   \
249  makeFvLaplacianTypeScheme(SS, scalar, symmTensor)                        \
250  makeFvLaplacianTypeScheme(SS, symmTensor, symmTensor)                    \
251  makeFvLaplacianTypeScheme(SS, tensor, symmTensor)                        \
252  makeFvLaplacianTypeScheme(SS, scalar, tensor)                            \
253  makeFvLaplacianTypeScheme(SS, symmTensor, tensor)                        \
254  makeFvLaplacianTypeScheme(SS, tensor, tensor)
```

`makeFvLaplacianTypeScheme` is another which is called several times and the definition is

```
                         ─── laplacianScheme.H ───
210  #define makeFvLaplacianTypeScheme(SS, GType, Type)                        \
211      typedef Foam::fv::SS<Foam::Type, Foam::GType> SS##Type##GType;        \
```

```
212        defineNamedTemplateTypeNameAndDebug(SS##Type##GType, 0);                    \
213                                                                                     \
214        namespace Foam                                                               \
215        {                                                                            \
216            namespace fv                                                             \
217            {                                                                        \
218                typedef SS<Type, GType> SS##Type##GType;                             \
219                                                                                     \
220                laplacianScheme<Type, GType>::                                       \
221                    addIstreamConstructorToTable<SS<Type, GType>>                    \
222                    add##SS##Type##GType##IstreamConstructorToTable_;                \
223            }                                                                        \
224        }
```

At line 211 the code creates the typedef `SS##Type##GType` for the class defined based on the macro input parameters. `SS##Type##GType` is passed as input to the macro `defineNamedTemplateTypeNameAndDebug` which is defined as

```
─────────────────────── className.H ───────────────────────
136  #define defineNamedTemplateTypeNameAndDebug(Type, DebugSwitch)                     \
137      defineNamedTemplateTypeName(Type);                                             \
138      defineNamedTemplateDebugSwitch(Type, DebugSwitch)
```

So now `Type = SS##Type##GType`, i.e. our class typedef. `Type` is passed into the macro `defineNamedTemplateTypeName`, which is defined as

```
─────────────────────── className.H ───────────────────────
113  #define defineNamedTemplateTypeName(Type)                                          \
114      defineTemplateTypeNameWithName(Type, Type::typeName_())
```

And we have finally reached the point, where the code calls the function `typeName_()` from the input class `Type`, which we remember corresponds to the class given by the typedef `SS##Type##GType` defined as

```
typedef Foam::fv::SS<Foam::Type, Foam::GType> SS##Type##GType;
```

`Type::typeName_()` will return the string which was passed into the macro `TypeName` at the beginning of this section.

```
─────────────────────── gaussLaplacianScheme.H ───────────────────────
82       TypeName("Gauss");
```

The macro `defineTemplateTypeNameWithName` is defined as

```
─────────────────────── className.H ───────────────────────
100  #define defineTemplateTypeNameWithName(Type, Name)                                 \
101      template<>                                                                     \
102      defineTypeNameWithName(Type, Name)
```

where the macro `defineTypeNameWithName` is defined as

```
─────────────────────── className.H ───────────────────────
92   #define defineTypeNameWithName(Type, Name)                                         \
93       const ::Foam::word Type::typeName(Name)
```

This is a call to the copy constructor in the class `word`, where the declared variable `typeName` from the laplacianScheme base class is set to the given string `"Gauss"`, so that the function `type()` returns this string. The link is `baseType##Type::typeName`, which I emphasized when the macro `declareRunTimeSelectionTable` was presented under Section 2.3.2 describing the `laplacianScheme` base class.

Now I will use an example to explain the order of the keywords that must be defined when specifying a laplacian scheme in `system/fvSchemes`. The example is

`laplacian(gamma,phi) Gauss linear corrected;`

The first keyword `Gauss` is used when the code calls the function

`fv::laplacianScheme<Type, GType>::New( ... )`

with some input in the function `laplacian` defined in `fvmLaplacian.C`. The selector function `New` in the class `laplacianScheme` will look for a laplacian scheme to construct, which is associated with the given keyword `Gauss`. In this case it would be the scheme implemented in the class `gaussLaplacianScheme`.

The next two keywords `linear` and `corrected` are needed during the construction of the laplacian scheme `gaussLaplacianScheme`, which is derived from the base class `laplacianScheme`. Therefore the base class `laplacianScheme` constructor is called when the `gaussLaplacianScheme` constructor is called. The `laplacianScheme` constructor must define two protected data members

```
tmp<surfaceInterpolationScheme<GType>> tinterpGammaScheme_;
tmp<snGradScheme<Type>> tsnGradScheme_;
```

where `tinterpGammaScheme_` is the cell centre to face centre interpolation scheme used for the diffusion coefficient `gamma`, and `tsnGradScheme_` is the surface normal gradient scheme that determines how we treat the non-orthogonal correction. The `laplacianScheme` constructor which needs the keywords is defined as

```
──────────────────────── laplacianScheme.H ────────────────────────
124          //- Construct from mesh and Istream
125          laplacianScheme(const fvMesh& mesh, Istream& is)
126          :
127              mesh_(mesh),
128              tinterpGammaScheme_(nullptr),
129              tsnGradScheme_(nullptr)
130          {
131              tinterpGammaScheme_ = tmp<surfaceInterpolationScheme<GType>>
132              (
133                  surfaceInterpolationScheme<GType>::New(mesh, is)
134              );
135
136              tsnGradScheme_ = tmp<snGradScheme<Type>>
137              (
138                  snGradScheme<Type>::New(mesh, is)
139              );
140          }
```

The keyword `linear` is used in the selector function

`surfaceInterpolationScheme<GType>::New(mesh, is)`

which looks for a surface interpolation scheme associated with the keyword `linear`. If the keyword is not found the execution is terminated and the user receives an error message, that includes a list of the available options.

The keyword `corrected` is used in the selector function

`snGradScheme<Type>::New(mesh, is)`

which looks for a surface normal gradient scheme associated with the keyword `corrected`. If the keyword is not found the execution is terminated and the user receives an error message, that includes a list of the available options.

2.5   Create your own copy

The class `gaussLaplacianScheme` is part of the shared object library file `libfiniteVolume.so`, which solvers can link to dynamically. This essentially means that we can modify the library and the changed functionalities will be included in all applications, that link to the library.

It is good practice to separate the original OpenFOAM code from your own modified code, which should be placed in the directory `$WM_PROJECT_USER_DIR/src`. Therefore we first need to create our own finiteVolume library, which we will call `libmyFiniteVolume`. In this library we will place our own copy of `gaussLaplacianScheme`, which we will name `myGaussLaplacianScheme`.

2.5.1   Getting started

First ensure that you have sourced your installation of OpenFOAM. Then execute commands below, which will create a copy of the folder `gaussLaplacianScheme` in your user directory with the same structure as in the original source code.

```
src
mkdir $WM_PROJECT_USER_DIR/src
cp -r --parents finiteVolume/finiteVolume/laplacianSchemes/gaussLaplacianScheme \
$WM_PROJECT_USER_DIR/src
```

2.5.2   Rename files and folders

When we copy existing source code, we should always rename to avoid ambiguity, that will lead to errors and may require reinstallation of OpenFOAM.

```
cd $WM_PROJECT_USER_DIR/src/finiteVolume/finiteVolume/laplacianSchemes
mv gaussLaplacianScheme myGaussLaplacianScheme
cd myGaussLaplacianScheme
mv gaussLaplacianScheme.C myGaussLaplacianScheme.C
mv gaussLaplacianScheme.H myGaussLaplacianScheme.H
mv gaussLaplacianSchemes.C myGaussLaplacianSchemes.C
```

2.5.3   Change class name in source files

```
sed -i s/gaussLaplacianScheme/myGaussLaplacianScheme/g \
myGaussLaplacianScheme.H
sed -i s/gaussLaplacianScheme/myGaussLaplacianScheme/g \
myGaussLaplacianScheme.C
sed -i s/gaussLaplacianScheme/myGaussLaplacianScheme/g \
myGaussLaplacianSchemes.C
```

2.5.4   Change type name

Open `myGaussLaplacianScheme.H` and modify line 82, which currently reads

```
    TypeName("Gauss");
```

to the new type name which I have chosen should be `myGauss`

```
    TypeName("myGauss");
```

It can also be performed with a sed command as follows

```
sed -i s/'TypeName("Gauss");'/'TypeName("myGauss");'/g myGaussLaplacianScheme.H
```

It is good practice to test a sed command by removing `-i`, then you can see the modifications in the terminal without modifying the file.

2.5.5 Create Make folder

Our new library needs a `Make` folder with two files, that are read by the compiler.

Now create a `Make` folder:

```
mkdir $WM_PROJECT_USER_DIR/src/finiteVolume/Make
```

2.5.6 Create Make/files

The first file is `files`. The source files that should be compiled in the library are listed first, and then at the last line the location and name of the compiled shared object library file is defined.

Create `files`:

```
vi $WM_PROJECT_USER_DIR/src/finiteVolume/Make/files
```

The content of `files` should be:

```
laplacianSchemes = finiteVolume/laplacianSchemes
$(laplacianSchemes)/myGaussLaplacianScheme/myGaussLaplacianSchemes.C

LIB = $(FOAM_USER_LIBBIN)/libmyFiniteVolume
```

Save and close `files` by typing command: `SHIFT+zz`.

2.5.7 Create Make/options

The second file is `options`. Here we give the path to the lnInclude folder of the needed libraries, and we link to the shared object library files in the second section. Note here that the OpenFOAM library headers are included by default.

Create `options`:

```
vi $WM_PROJECT_USER_DIR/src/finiteVolume/Make/options
```

The content of `options` should be:

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

LIB_LIBS = \
    -lOpenFOAM \
    -lfileFormats \
    -lsurfMesh \
    -lmeshTools \
    -lfiniteVolume
```

Save and close `options` by typing command: `SHIFT+zz`.

2.5.8 Compile myFiniteVolume library

Compile the library by executing:

```
wmake $WM_PROJECT_USER_DIR/src/finiteVolume
```

Before you modify the code further, you should test the scheme by applying it in an existing tutorial.

```
cp -r $FOAM_TUTORIALS/incompressible/icoFoam/cavity/cavity $FOAM_RUN/myGaussCavity
```

Open scheme settings in editor, for example with vi

```
vi $FOAM_RUN/myGaussCavity/system/fvSchemes
```

and modify the subdictionary `laplacianSchemes` from using `Gauss` defined as

```
laplacianSchemes
{
    default         Gauss linear orthogonal;
}
```

to use the new scheme `myGauss` defined as

```
laplacianSchemes
{
    default         myGauss linear orthogonal;
}
```

We need to tell the case about our shared object library `libmyFiniteVolume.so`, which is located in the folder `$WM_PROJECT_USER_DIR/platforms`. This is done at the end of `controlDict`. To include the library append

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

to the end of `controlDict`.
The commands to build the mesh and run the case are

```
blockMesh -case $FOAM_RUN/myGaussCavity >& $FOAM_RUN/myGaussCavity/log.blockMesh
icoFoam -case $FOAM_RUN/myGaussCavity >& $FOAM_RUN/myGaussCavity/log.icoFoam
```

Now open the log file and verify that it worked.

Chapter 3

GFM-Gauss scheme

3.1 How to use it

The new scheme is specified in case file `system/fvSchemes` in the subdictionary `laplacianSchemes`. The keyword for the new scheme is `GFMGauss`, so the specification for the laplacian scheme `laplacian(gamma,phi)` would be

`laplacian(gamma,phi)  GFMGauss linear corrected;`

where the diffusion coefficient is interpolated linearly (`linear`) and we include an explicit non-orthogonal correction (`corrected`).

The scheme should only be used for the dynamic pressure in the solvers `interIsoFoam` and `interFoam`. The scheme requires a step interpolated density field `rho` and a void fraction field `alpha1`. Furthermore the scheme needs a new class called `interfaceJump`. For further details read the implementation Section 3.3.

3.2 Theory

The theory presented in this section is an elaboration of the theory presented by Vukcevic et al. [1].

3.2.1 Governing equations

Before we get to the GFM-Gauss discretization, I will present the theoretical background.

Continuity equation

The continuity equation for an incompressible fluid is given by

$$\nabla \bullet (\mathbf{U}) = 0 \tag{3.1}$$

where ρ is the fluid density and $\mathbf{U}$ is the velocity field. Here we have divided the original continuity equation with the density, such that it is eliminated from the equation.

Momentum equations

The momentum equation for a Newtonian incompressible fluid subjected to gravity is given by

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \bullet (\mathbf{U}\mathbf{U}) - \nabla \bullet (\nu \nabla \mathbf{U}) = -\frac{1}{\rho}\nabla p_{total} + \mathbf{g} \tag{3.2}$$

where t is time, ν is the kinematic viscosity of the fluid, $\mathbf{g}$ is the gravitational acceleration vector and p_{total} is the total pressure. The right hand side of the momentum equation is rewritten to

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \bullet (\mathbf{UU}) - \nabla \bullet (\nu \nabla \mathbf{U}) = -\frac{1}{\rho} \nabla p_{total} + \nabla (\mathbf{g} \bullet \mathbf{x}) \Leftrightarrow \tag{3.3}$$

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \bullet (\mathbf{UU}) - \nabla \bullet (\nu \nabla \mathbf{U}) = -\frac{1}{\rho} \nabla \left(p_{total} + \rho \mathbf{g} \bullet \mathbf{x} \right) \Leftrightarrow \tag{3.4}$$

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \bullet (\mathbf{UU}) - \nabla \bullet (\nu \nabla \mathbf{U}) = -\frac{1}{\rho} \nabla \left(p \right) \tag{3.5}$$

$\mathbf{x}$ is the location in space and the dynamic pressure p has been introduced. The dynamic pressure is the residual of the total pressure minus the hydrostatic component given by

$$p = p_{total} - \rho \mathbf{g} \bullet \mathbf{x} \tag{3.6}$$

Pressure equation

For incompressible flows, we loose the connection between the momentum equation and the continuity equation. Therefore we need to create an additional equation that contain both velocity and pressure like the three momentum equations. The fourth equation is created by combining the momentum equation and the continuity equation, which leads to the pressure equation. The pressure equation is a Poisson equation.

First we need to discretize part of the momentum equation. The Finite Volume method is constructed by integrating in space and time over a control volume. When the pressure gradient is left non-discretized in the momentum equation, we get

$$\int_t^{t+\Delta t} \left[\int_{V_p} \frac{\partial \mathbf{U}}{\partial t} \, \mathrm{d}V + \int_{V_P} \nabla \bullet (\mathbf{UU}) \, \mathrm{d}V - \int_{V_P} \nabla \bullet (\nu \nabla \mathbf{U}) \, \mathrm{d}V \right] \mathrm{d}t = -\frac{1}{\rho} \nabla p \tag{3.7}$$

It is assumed that the control volume does not change in time, and we use the midpoint rule to approximate the volume integral.

$$\int_{V_P} \frac{\partial \mathbf{U}}{\partial t} \, \mathrm{d}V = \left(\frac{\partial \mathbf{U}}{\partial t} \right)_P V_P \tag{3.8}$$

The time derivative is calculated according to a chosen scheme, for example Euler or Crank-Nicholson.

The convective term is discretized using Gauss theorem to convert the volume integral into a surface integral. The surface integral is approximated as a summation over the control volume faces. The non-linearity is usually resolved with Picard iteration, where one of the velocity fields is treated implicitly and the other explicitly from previous iteration result. To distinguish between the two velocity fields, a new scalar face flux variable F_f is introduced as $F_f = \mathbf{S}_f \bullet \mathbf{U}_f$. The discretization is given by

$$\int_{V_p} \nabla \bullet (\mathbf{UU}) \, \mathrm{d}V = \int_{S_p} \mathrm{d}S \bullet (\mathbf{UU}) \tag{3.9}$$

$$= \sum_{f=1}^{n_f} \mathbf{S}_f \bullet (\mathbf{UU})_f \tag{3.10}$$

$$= \sum_{f=1}^{n_f} F_f \mathbf{U}_f \tag{3.11}$$

where the velocity at the control volume faces is given by

$$\mathbf{U}_f = f_x \mathbf{U}_P + (1 - f_x) \mathbf{U}_N \tag{3.12}$$

The diffusive term is also discretized using Gauss theorem, which gives

$$\int_{V_P} \nabla \bullet (\nu \nabla \mathbf{U}) \, dV = \int_{S_P} dS \bullet (\nu \nabla \mathbf{U}) \tag{3.13}$$

$$= \sum_{f=1}^{n_f} \mathbf{S}_f \bullet (\nu \nabla \mathbf{U})_f \tag{3.14}$$

$$= \sum_{f=1}^{n_f} \nu_f \mathbf{S}_f \bullet (\nabla \mathbf{U})_f \tag{3.15}$$

The discretization of the term $\mathbf{S}_f \bullet (\nabla \mathbf{U})_f$ is further separated in a orthogonal part, which is treated implicitly, and a non-orthogonal part, which is treated explicitly. The separation is given by

$$\mathbf{S}_f \bullet (\nabla \mathbf{U})_f = |\mathbf{S}_f| \frac{\mathbf{U}_N - \mathbf{U}_P}{|\mathbf{d}|} \tag{3.16}$$

$$= \underbrace{|\mathbf{\Delta}_f| \frac{\mathbf{U}_N - \mathbf{U}_P}{|\mathbf{d}|}}_{Orthogonal} + \underbrace{\mathbf{k} \bullet (\nabla \mathbf{U})_f}_{Non\ orthogonal} \tag{3.17}$$

If we do not treat the non orthogonal term explicitly, the system matrix will loose diagonally dominance, which is an important property to preserve in the equation system. The explicit evaluation of the gradient term $(\nabla \mathbf{U})_f$ is given by a linear interpolation of the gradient

$$(\nabla \mathbf{U})_f = f_x (\nabla \mathbf{U})_P + (1 - f_x)(\nabla \mathbf{U})_N \tag{3.18}$$

where $(\nabla \mathbf{U})_P$ and $(\nabla \mathbf{U})_N$ is given by

$$(\nabla \mathbf{U})_P = \frac{1}{V_P} \sum_{f=1}^{n_f} \mathbf{S}_f \mathbf{U}_f \tag{3.19}$$

$$(\nabla \mathbf{U})_N = \frac{1}{V_N} \sum_{f=1}^{n_f} \mathbf{S}_f \mathbf{U}_f \tag{3.20}$$

To summarize the discretized momentum equation is now given by

$$\int_t^{t+\Delta t} \left[\left(\frac{\partial \mathbf{U}}{\partial t} \right)_P V_P + \sum_{f=1}^{n_f} F_f \mathbf{U}_f - \sum_{f=1}^{n_f} \nu_f \left(\underbrace{|\mathbf{\Delta}_f| \frac{\mathbf{U}_N - \mathbf{U}_P}{|\mathbf{d}|}}_{Orthogonal} + \underbrace{\mathbf{k} \bullet (\nabla \mathbf{U})_f}_{Non-orthogonal} \right) \right] dt = -\frac{1}{\rho} \nabla p \tag{3.21}$$

where the dynamic pressure gradient is left undiscretized. Now it is possible to collect terms that relates to the unknown $\mathbf{U}_P$ and $\mathbf{U}_N$. Terms that are evaluated explicitly are collected in a single source term, $\mathbf{S}_P$. We can therefore introduce a compact notation for the final dicretisation given by

$$a_P \mathbf{U}_p + \sum_N a_N \mathbf{U}_N - \mathbf{S}_P = -\frac{1}{\rho} \nabla p \tag{3.22}$$

where we assume that a temporal discretization has been applied to replace the time integrals on the left hand side.

The continuity equation is discretized using Gauss theorem to convert volume integrals to surface integrals. The surface integral is approximated by a summation of the control volume faces,

where the midpoint rule is used to approximate the integral over each face. This gives

$$\int_{V_P} \nabla \bullet (\mathbf{U}) \, \mathrm{d}V = \int_{S_P} \mathrm{d}S \bullet (\mathbf{U}) \tag{3.23}$$

$$= \sum_{f=1}^{n_f} \mathbf{S}_f \bullet \mathbf{U}_f \tag{3.24}$$

The discretized continuity equation is

$$\sum_{f=1}^{n_f} \mathbf{S}_f \bullet \mathbf{U}_f = 0 \tag{3.25}$$

Now we apply Rhie-Chow Momentum Interpolation Method to achieve an expression for the face velocity. The first step is to isolate $\mathbf{U}_p$, which gives

$$\mathbf{U}_p = \frac{-\frac{1}{\rho}\nabla p - \sum_N a_N \mathbf{U}_N + \mathbf{S}_P}{a_P} \tag{3.26}$$

We now rewrite the equation to

$$\mathbf{U}_p = \frac{-\sum_N a_N \mathbf{U}_N + \mathbf{S}_P}{a_P} - \frac{1}{a_P}\frac{1}{\rho}\nabla p \tag{3.27}$$

and introduce the operator $\mathcal{H}(\mathbf{U})$ which is defined as

$$\mathcal{H}(\mathbf{U}) = -\sum_N a_N \mathbf{U}_N + \mathbf{S}_P \tag{3.28}$$

The velocity is now given by

$$\mathbf{U}_p = \frac{\mathcal{H}(\mathbf{U})}{a_P} - \frac{1}{a_P}\frac{1}{\rho}\nabla p \tag{3.29}$$

The Rhie-Chow interpolation is applied and gives

$$\mathbf{U}_f = \left(\frac{\mathcal{H}(\mathbf{U})}{a_P}\right)_f - \left(\frac{1}{a_P}\right)_f \left(\frac{1}{\rho}\nabla p\right)_f \tag{3.30}$$

The final step is to insert the expression for the face velocity into the discretized continuity equation, which gives

$$\sum_{f=1}^{n_f} \mathbf{S}_f \bullet \left(\frac{1}{a_P}\right)_f \left(\frac{1}{\rho}\nabla p\right)_f = \sum_{f=1}^{n_f} \mathbf{S}_f \bullet \left(\frac{\mathcal{H}(\mathbf{U})}{a_P}\right)_f \tag{3.31}$$

The right hand side is a source term, which is the flux of the estimated velocity field from the momentum equation with the effect of the pressure gradient term. The left hand side is pressure laplacian term, which is discretized with the modified Gauss discretization scheme.

3.2.2   Jump conditions at the surface

At the free surface between air and water the density changes from the density of water to the density of air. This jump in density at the surface is denoted square brackets, and the definition is

$$[\rho] = \rho^- - \rho^+ \tag{3.32}$$

where ρ^- is the lighter fluid, and ρ^+ is the heavier fluid. The kinematic boundary condition at the free surface is

$$[\mathbf{u}] = \mathbf{u}^- - \mathbf{u}^+ = \mathbf{0} \tag{3.33}$$

where superscript $^-$ means very close to the surface on the light phase side, and $^+$ means very close to the surface on the heavy phase side. Hence, it is assumed that there is no jump in the velocity field. The tangential stress balance is simplified to

$$[\nabla_n \mathbf{u}_t] = \mathbf{0} \tag{3.34}$$

This physically means that the water and air have the same velocity at the interface, instead of a stress balance. This form is obtained by neglecting surface divergence of surface tension and surface gradient of the normal velocity component. The dynamic boundary condition for the dynamic pressure is

$$[p] = -[\rho]\mathbf{g} \bullet \mathbf{x} = \mathcal{H} \tag{3.35}$$

where the dynamic pressure p is defined as

$$p = p_{tot} - \rho \mathbf{g} \bullet \mathbf{x} \tag{3.36}$$

The discretized jump condition is

$$[p] = -[\rho]\mathbf{g} \bullet \mathbf{x} \Leftrightarrow$$
$$p^- - p^+ = -(\rho^- - \rho^+)\mathbf{g} \bullet \mathbf{x} \Leftrightarrow$$
$$p^- - p^+ = (\rho^+ - \rho^-)\mathbf{g} \bullet \mathbf{x} \Leftrightarrow \tag{3.37}$$

The right hand side is simplified by introducing $\mathcal{H}$, which is defined as

$$\mathcal{H} = (\rho^+ - \rho^-)\mathbf{g} \bullet \mathbf{x_\Gamma} \tag{3.38}$$

The dynamic boundary condition is obtained by neglecting surface tension effects and using the above pressure decomposition. From inspection of the incompressible Navier-Stokes equation when the simplified tangential stress balance is assumed an additional dynamic boundary condition can be defined as

$$\left[\frac{\nabla p}{\rho}\right] = \frac{(\nabla p)^-}{\rho^-} - \frac{(\nabla p)^+}{\rho^+} = \mathbf{0} \tag{3.39}$$

This condition is used to derive the extrapolated pressure values, which are the ghost values in this method.

Matrix symmetry conservation after jump discretization

According to the additional dynamic boundary condition the product of ρ and ∇p is continuous even though they individually are discontinuous at the interface. This fact is important, since this ensures that the discretized laplacian matrix including the discretized jump conserves the existing matrix symmetry in the laplacian operator.

3.2.3 Additional definitions

The position of the surface is determined from the normalised distance factor called λ, which is defined as

$$\lambda = \frac{\alpha_P - 0.5}{\alpha_P - \alpha_N} \tag{3.40}$$

The position of the surface is found from

$$\mathbf{x_\Gamma} = \mathbf{x_P} + \lambda \mathbf{d_f} \tag{3.41}$$

where $\mathbf{x_\Gamma}$ is the position of the free surface, $\mathbf{x_P}$ is the position of the owner cell centre and $\mathbf{d_f}$ is the vector from owner cell centre to neighbour cell centre. The inverse density is defined as

$$\beta^+ = \frac{1}{\rho^+} \tag{3.42}$$

and

$$\beta^- = \frac{1}{\rho^-} \tag{3.43}$$

3.2.4   Derivation of extrapolated pressure values

In this section I have written out the derivation of the extrapolated ghost values used in this method. There are 4 cases to consider

1. Owner cell is wet, and we are discretizing the equation in the owner cell, so we are looking through the cell face from the owner cell towards the neighbour cell.

2. Owner cell is wet, and we are discretizing the equation in the neighbour cell, so we are looking through cell face from the neighbour cell towards the owner cell.

3. Owner cell is dry, and we are discretizing the equation in the owner cell, so we are looking through the cell face from the owner cell towards the neighbour cell.

4. Owner cell is dry, and we are discretizing the equation in the neighbour cell, so we are looking through cell face from the neighbour cell towards the owner cell.

Figure 3.1 illustrates the interpolation in case 1 and 2.
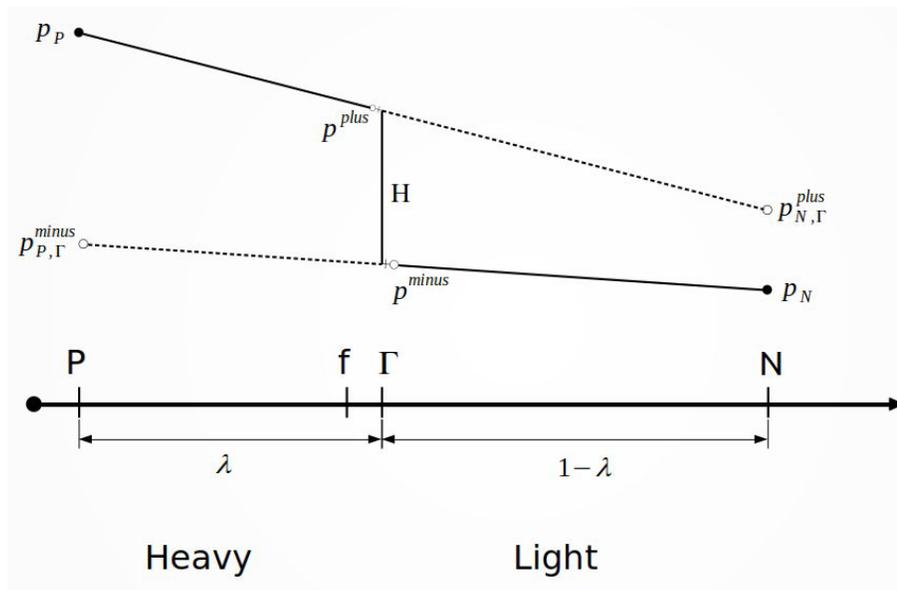

Figure 3.1: One-sided interpolation for a wet owner cell.

Figure 3.2 illustrates the interpolation in case 3 and 4.
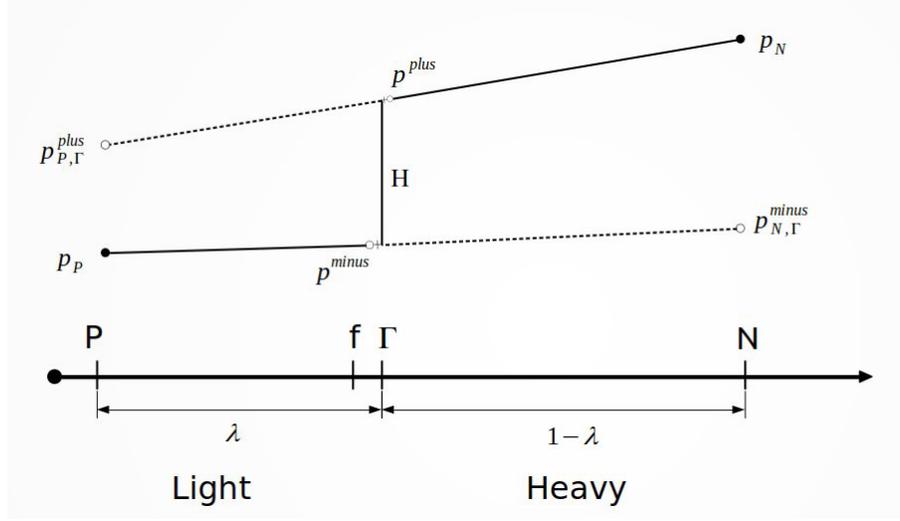
Figure 3.2: One-sided interpolation for a dry owner cell.

In the next four sections, that correspond to the 4 cases, the extrapolated pressure values are expressed in terms of the jump $\mathcal{H}$, and the cell centre values p_P and p_N. This is achieved by using the presented jump conditions at the free surface.

Wet owner, owner towards neighbour

The additional dynamic jump condition Equation (3.39) is discretized

$$\beta^- \frac{p_N - p^-}{1 - \lambda} - \beta^+ \frac{p^+ - p_P}{\lambda} = 0 \tag{3.44}$$

The dynamic jump condition Equation (3.35) is given by

$$p^- - p^+ = \mathcal{H} \tag{3.45}$$

which yields an expression for p^- given by

$$p^- = p^+ + \mathcal{H} \tag{3.46}$$

Now insert Equation (3.46) in Equation (3.44) and isolate p^+

$$\beta^- \frac{p_N - (p^+ + \mathcal{H})}{1 - \lambda} - \beta^+ \frac{p^+ - p_P}{\lambda} = 0 \Leftrightarrow$$

$$\frac{\beta^-}{1 - \lambda}(p_N - p^+ - \mathcal{H}) - \frac{\beta^+}{\lambda}(p^+ - p_P) = 0 \Leftrightarrow$$

$$\frac{\beta^- \lambda}{(1 - \lambda)\lambda}(p_N - p^+ - \mathcal{H}) - \frac{\beta^+ (1 - \lambda)}{(1 - \lambda)\lambda}(p^+ - p_P) = 0 \Leftrightarrow$$

$$\beta^- \lambda (p_N - p^+ - \mathcal{H}) - \beta^+ (1 - \lambda)(p^+ - p_P) = 0 \Leftrightarrow$$

$$p^+ (\beta^- \lambda + \beta^+ (1 - \lambda)) = \beta^- \lambda p_N - \beta^- \lambda \mathcal{H} + \beta^+ (1 - \lambda) p_P \Leftrightarrow$$

$$p^+ = \frac{\beta^- \lambda}{\beta^- \lambda + \beta^+ (1 - \lambda)} p_N - \frac{\beta^- \lambda}{\beta^- \lambda + \beta^+ (1 - \lambda)} \mathcal{H} + \frac{\beta^+ (1 - \lambda)}{\beta^- \lambda + \beta^+ (1 - \lambda)} p_P \tag{3.47}$$

The expression is simplified by introducing

$$\overline{\beta_w} = \beta^- \lambda + \beta^+ (1 - \lambda) \tag{3.48}$$

which yields

$$p^+ = \frac{\beta^-\lambda}{\overline{\beta_w}}p_N - \frac{\beta^-\lambda}{\overline{\beta_w}}\mathcal{H} + \frac{\beta^+(1-\lambda)}{\overline{\beta_w}}p_P \tag{3.49}$$

The value p_P and p^+ is then used to define the gradient in a linear extrapolation from our current position in the owner cell, and the expression for the value in the neighbouring cell centre becomes

$$
\begin{aligned}
p_{N,\Gamma}^+ &= p_P + \frac{p^+ - p_P}{\lambda}\\
&= p_P + \frac{\frac{\beta^-\lambda}{\overline{\beta_w}}p_N - \frac{\beta^-\lambda}{\overline{\beta_w}}\mathcal{H} + \frac{\beta^+(1-\lambda)}{\overline{\beta_w}}p_P}{\lambda} - \frac{p_P}{\lambda}\\
&= p_P + \frac{\beta^-}{\overline{\beta_w}}p_N - \frac{\beta^-}{\overline{\beta_w}}\mathcal{H} + \frac{\beta^+(1-\lambda)}{\overline{\beta_w}\lambda}p_P - \frac{p_P}{\lambda}
\end{aligned}
\tag{3.50}
$$

Equation (3.48) is rewritten to

$$\overline{\beta_w} = \beta^-\lambda + \beta^+(1-\lambda)$$

$$1 = \frac{\beta^-\lambda}{\overline{\beta_w}} + \frac{\beta^+(1-\lambda)}{\overline{\beta_w}}$$

$$1 - \frac{\beta^-\lambda}{\overline{\beta_w}} = \frac{\beta^+(1-\lambda)}{\overline{\beta_w}}$$

and this is used to change the fourth term in Equation (3.50), which yields the final expression of the extrapolation formula

$$p_{N,\Gamma}^+ = p_P + \frac{\beta^-}{\overline{\beta_w}}p_N - \frac{\beta^-}{\overline{\beta_w}}\mathcal{H} + \left(1 - \frac{\beta^-\lambda}{\overline{\beta_w}}\right)\frac{1}{\lambda}p_P - \frac{p_P}{\lambda}$$

$$p_{N,\Gamma}^+ = \frac{\beta^-}{\overline{\beta_w}}p_N - \frac{\beta^-}{\overline{\beta_w}}\mathcal{H} + \left(1 + \frac{1}{\lambda} - \frac{\beta^-}{\overline{\beta_w}} - \frac{1}{\lambda}\right)p_P \tag{3.51}$$

$$p_{N,\Gamma}^+ = \frac{\beta^-}{\overline{\beta_w}}p_N - \frac{\beta^-}{\overline{\beta_w}}\mathcal{H} + \left(1 - \frac{\beta^-}{\overline{\beta_w}}\right)p_P \tag{3.52}$$

Wet owner, neighbour towards owner

Equation (3.45) is used to get an expression for p^+ given by

$$p^+ = p^- - \mathcal{H} \tag{3.53}$$

Now insert Equation (3.53) in Equation (3.44) and isolate p^-

$$\beta^- \frac{p_N - p^-}{1-\lambda} - \beta^+ \frac{(p^- - \mathcal{H}) - p_P}{\lambda} = 0 \Leftrightarrow$$

$$\frac{\beta^-}{1-\lambda}(p_N - p^-) - \frac{\beta^+}{\lambda}(p^- - \mathcal{H} - p_P) = 0 \Leftrightarrow$$

$$\frac{\beta^-\lambda}{(1-\lambda)\lambda}(p_N - p^-) - \frac{\beta^+(1-\lambda)}{(1-\lambda)\lambda}(p^- - \mathcal{H} - p_P) = 0 \Leftrightarrow$$

$$\beta^-\lambda(p_N - p^-) - \beta^+(1-\lambda)(p^- - \mathcal{H} - p_P) = 0 \Leftrightarrow$$

$$p^-(\beta^-\lambda + \beta^+(1-\lambda)) = \beta^-\lambda p_N + \beta^+(1-\lambda)\mathcal{H} + \beta^+(1-\lambda)p_P \Leftrightarrow$$

$$p^- = \frac{\beta^-\lambda}{\beta^-\lambda + \beta^+(1-\lambda)}p_N + \frac{\beta^+(1-\lambda)}{\beta^-\lambda + \beta^+(1-\lambda)}\mathcal{H} + \frac{\beta^+(1-\lambda)}{\beta^-\lambda + \beta^+(1-\lambda)}p_P \tag{3.54}$$

Definition from Equation (3.48) is used again to simplify the expression to

$$p^- = \frac{\beta^-\lambda}{\overline{\beta_w}}p_N + \frac{\beta^+(1-\lambda)}{\overline{\beta_w}}\mathcal{H} + \frac{\beta^+(1-\lambda)}{\overline{\beta_w}}p_P \tag{3.55}$$

The value p_N and p^- is then used to define the gradient in a linear extrapolation from our current position in the neighbour cell, and the expression for the value in the owner cell centre becomes

$$
\begin{aligned}
p^-_{P,\Gamma} &= p_N + \frac{p^- - p_N}{1 - \lambda} \\
&= p_N + \frac{\frac{\beta^-\lambda}{\overline{\beta_w}}p_N + \frac{\beta^+(1-\lambda)}{\overline{\beta_w}}\mathcal{H} + \frac{\beta^+(1-\lambda)}{\overline{\beta_w}}p_P}{1 - \lambda} - \frac{p_N}{1 - \lambda} \\
&= p_N + \frac{\beta^-\lambda}{\overline{\beta_w}(1-\lambda)}p_N + \frac{\beta^+}{\overline{\beta_w}}\mathcal{H} + \frac{\beta^+}{\overline{\beta_w}}p_P - \frac{p_N}{1 - \lambda} \\
&= p_N\left(1 + \frac{\beta^-\lambda}{\overline{\beta_w}(1-\lambda)} - \frac{1}{1-\lambda}\right) + \frac{\beta^+}{\overline{\beta_w}}\mathcal{H} + \frac{\beta^+}{\overline{\beta_w}}p_P \\
&= p_N\left(1 + \frac{\beta^-\lambda}{\overline{\beta_w}}\frac{1}{(1-\lambda)} - \frac{1}{1-\lambda}\right) + \frac{\beta^+}{\overline{\beta_w}}\mathcal{H} + \frac{\beta^+}{\overline{\beta_w}}p_P
\end{aligned}
\tag{3.56}
$$

Equation (3.48) is rewritten to

$$
\begin{aligned}
\overline{\beta_w} &= \beta^-\lambda + \beta^+(1-\lambda) \\
1 &= \frac{\beta^-\lambda}{\overline{\beta_w}} + \frac{\beta^+(1-\lambda)}{\overline{\beta_w}} \\
\frac{\beta^-\lambda}{\overline{\beta_w}} &= 1 - \frac{\beta^+(1-\lambda)}{\overline{\beta_w}}
\end{aligned}
$$

and this is used to change the fourth term in Equation (3.56), which yields the final expression of the extrapolation formula

$$
\begin{aligned}
p^-_{P,\Gamma} &= p_N\left(1 + \left(1 - \frac{\beta^+(1-\lambda)}{\overline{\beta_w}}\right)\frac{1}{(1-\lambda)} - \frac{1}{1-\lambda}\right) + \frac{\beta^+}{\overline{\beta_w}}\mathcal{H} + \frac{\beta^+}{\overline{\beta_w}}p_P \\
&= p_N\left(1 + \frac{1}{(1-\lambda)} - \frac{\beta^+}{\overline{\beta_w}} - \frac{1}{1-\lambda}\right) + \frac{\beta^+}{\overline{\beta_w}}\mathcal{H} + \frac{\beta^+}{\overline{\beta_w}}p_P \\
&= p_N\left(1 - \frac{\beta^+}{\overline{\beta_w}}\right) + \frac{\beta^+}{\overline{\beta_w}}\mathcal{H} + \frac{\beta^+}{\overline{\beta_w}}p_P
\end{aligned}
\tag{3.57}
$$

Dry owner, owner towards neighbour

The additional dynamic jump condition (3.39) is discretized

$$
\beta^+\frac{p_N - p^+}{1 - \lambda} - \beta^-\frac{p^- - p_P}{\lambda} = 0
\tag{3.58}
$$

Now insert Equation (3.53) in Equation (3.58) and isolate p^+

$$
\begin{aligned}
\beta^+\frac{p_N - (p^- - \mathcal{H})}{1 - \lambda} - \beta^-\frac{p^- - p_P}{\lambda} &= 0 \Leftrightarrow \\
\frac{\beta^+}{1 - \lambda}(p_N - p^- + \mathcal{H}) - \frac{\beta^-}{\lambda}(p^- - p_P) &= 0 \Leftrightarrow \\
\frac{\beta^+\lambda}{(1-\lambda)\lambda}(p_N - p^- + \mathcal{H}) - \frac{\beta^-(1-\lambda)}{(1-\lambda)\lambda}(p^- - p_P) &= 0 \Leftrightarrow \\
\beta^+\lambda(p_N - p^- + \mathcal{H}) - \beta^-(1-\lambda)(p^- - p_P) &= 0 \Leftrightarrow \\
p^-(\beta^+\lambda + \beta^-(1-\lambda)) &= \beta^+\lambda p_N + \beta^+\lambda\mathcal{H} + \beta^-(1-\lambda)p_P \Leftrightarrow \\
p^+ = \frac{\beta^+\lambda}{\beta^+\lambda + \beta^-(1-\lambda)}p_N + \frac{\beta^+\lambda}{\beta^+\lambda + \beta^-(1-\lambda)}\mathcal{H} &+ \frac{\beta^-(1-\lambda)}{\beta^+\lambda + \beta^-(1-\lambda)}p_P
\end{aligned}
\tag{3.59}
$$

The expression is simplified by introducing

$$\overline{\beta_d} = \beta^+ \lambda + \beta^- (1 - \lambda) \tag{3.60}$$

which yields

$$p^+ = \frac{\beta^+ \lambda}{\overline{\beta_d}} p_N + \frac{\beta^+ \lambda}{\overline{\beta_d}} \mathcal{H} + \frac{\beta^- (1 - \lambda)}{\overline{\beta_d}} p_P \tag{3.61}$$

The value p_P and p^- is then used to define the gradient in a linear extrapolation from our current position in the owner cell, and the expression for the value in the neighbouring cell centre becomes

$$
\begin{aligned}
p_{N,\Gamma}^- &= p_P + \frac{p^+ - p_P}{\lambda} \\
&= p_P + \frac{\frac{\beta^+ \lambda}{\overline{\beta_d}} p_N + \frac{\beta^+ \lambda}{\overline{\beta_d}} \mathcal{H} + \frac{\beta^- (1-\lambda)}{\overline{\beta_d}} p_P}{\lambda} - \frac{p_P}{\lambda} \\
&= p_P + \frac{\beta^+}{\overline{\beta_d}} p_N + \frac{\beta^+}{\overline{\beta_d}} \mathcal{H} + \frac{\beta^- (1-\lambda)}{\overline{\beta_d}} \frac{1}{\lambda} p_P - \frac{p_P}{\lambda}
\end{aligned}
\tag{3.62}
$$

Equation (3.60) is rewritten to

$$
\begin{aligned}
\overline{\beta_d} &= \beta^+ \lambda + \beta^- (1 - \lambda) \\
1 &= \frac{\beta^+ \lambda}{\overline{\beta_d}} + \frac{\beta^- (1 - \lambda)}{\overline{\beta_d}} \\
\frac{\beta^- (1 - \lambda)}{\overline{\beta_d}} &= 1 - \frac{\beta^+ \lambda}{\overline{\beta_d}}
\end{aligned}
$$

and this is used to change the fourth term in Equation (3.62), which yields the final expression of the extrapolation formula

$$
p_{N,\Gamma}^- = p_P + \frac{\beta^+}{\overline{\beta_d}} p_N + \frac{\beta^+}{\overline{\beta_d}} \mathcal{H} + \left(1 - \frac{\beta^+ \lambda}{\overline{\beta_d}} \right) \frac{1}{\lambda} p_P - \frac{p_P}{\lambda}
$$

$$
p_{N,\Gamma}^- = \frac{\beta^+}{\overline{\beta_d}} p_N + \frac{\beta^+}{\overline{\beta_d}} \mathcal{H} + \left(1 + \frac{1}{\lambda} - \frac{\beta^+}{\overline{\beta_d}} - \frac{1}{\lambda} \right) p_P \tag{3.63}
$$

$$
p_{N,\Gamma}^- = \frac{\beta^+}{\overline{\beta_d}} p_N + \frac{\beta^+}{\overline{\beta_d}} \mathcal{H} + \left(1 - \frac{\beta^+}{\overline{\beta_d}} \right) p_P \tag{3.64}
$$

Dry owner, neighbour towards owner

Insert Equation (3.46) in Equation (3.58) and isolate p^+

$$
\beta^+ \frac{p_N - p^+}{1 - \lambda} - \beta^- \frac{(p^+ + \mathcal{H}) - p_P}{\lambda} = 0 \Leftrightarrow
$$

$$
\frac{\beta^+}{1 - \lambda} (p_N - p^+) - \frac{\beta^-}{\lambda} (p^+ + \mathcal{H} - p_P) = 0 \Leftrightarrow
$$

$$
\frac{\beta^+ \lambda}{(1 - \lambda) \lambda} (p_N - p^+) - \frac{\beta^- (1 - \lambda)}{(1 - \lambda) \lambda} (p^+ + \mathcal{H} - p_P) = 0 \Leftrightarrow
$$

$$
\beta^+ \lambda (p_N - p^+) - \beta^- (1 - \lambda)(p^+ + \mathcal{H} - p_P) = 0 \Leftrightarrow
$$

$$
p^+ (\beta^+ \lambda + \beta^- (1 - \lambda)) = \beta^+ \lambda p_N - \beta^- (1 - \lambda) \mathcal{H} + \beta^- (1 - \lambda) p_P \Leftrightarrow
$$

$$
p^+ = \frac{\beta^+ \lambda}{\beta^+ \lambda + \beta^- (1 - \lambda)} p_N - \frac{\beta^- (1 - \lambda)}{\beta^+ \lambda + \beta^- (1 - \lambda)} \mathcal{H} + \frac{\beta^- (1 - \lambda)}{\beta^+ \lambda + \beta^- (1 - \lambda)} p_P \tag{3.65}
$$

Definition from Equation (3.60) is used again to simplify the expression to

$$
p^+ = \frac{\beta^+ \lambda}{\overline{\beta_d}} p_N - \frac{\beta^- (1 - \lambda)}{\overline{\beta_d}} \mathcal{H} + \frac{\beta^- (1 - \lambda)}{\overline{\beta_d}} p_P \tag{3.66}
$$

The value p_N and p^+ is then used to define the gradient in a linear extrapolation from our current position in the neighbour cell, and the expression for the value in the owner cell centre becomes

$$
\begin{aligned}
p_{P,\Gamma}^+ &= p_N + \frac{p^+ - p_N}{1 - \lambda} \\
&= p_N + \frac{\frac{\beta^+ \lambda}{\overline{\beta_d}} p_N - \frac{\beta^- (1-\lambda)}{\overline{\beta_d}} \mathcal{H} + \frac{\beta^- (1-\lambda)}{\overline{\beta_d}} p_P}{1 - \lambda} - \frac{p_N}{1 - \lambda} \\
&= p_N + \frac{\beta^+ \lambda}{\overline{\beta_d}(1 - \lambda)} p_N - \frac{\beta^-}{\overline{\beta_d}} \mathcal{H} + \frac{\beta^-}{\overline{\beta_d}} p_P - \frac{p_N}{1 - \lambda} \\
&= p_N \left(1 + \frac{\beta^+ \lambda}{\overline{\beta_d}(1 - \lambda)} - \frac{1}{1 - \lambda} \right) - \frac{\beta^-}{\overline{\beta_d}} \mathcal{H} + \frac{\beta^-}{\overline{\beta_d}} p_P \\
&= p_N \left(1 + \frac{\beta^+ \lambda}{\overline{\beta_d}} \frac{1}{(1 - \lambda)} - \frac{1}{1 - \lambda} \right) - \frac{\beta^-}{\overline{\beta_d}} \mathcal{H} + \frac{\beta^-}{\overline{\beta_d}} p_P
\end{aligned}
\tag{3.67}
$$

Equation (3.60) is rewritten to

$$
\begin{aligned}
\overline{\beta_d} &= \beta^+ \lambda + \beta^- (1 - \lambda) \\
1 &= \frac{\beta^+ \lambda}{\overline{\beta_d}} + \frac{\beta^- (1 - \lambda)}{\overline{\beta_d}} \\
\frac{\beta^+ \lambda}{\overline{\beta_d}} &= 1 - \frac{\beta^- (1 - \lambda)}{\overline{\beta_d}}
\end{aligned}
$$

and this is used to change the fourth term in Equation (3.67), which yields the final expression of the extrapolation formula

$$
\begin{aligned}
p_{P,\Gamma}^+ &= p_N \left(1 + \left(1 - \frac{\beta^- (1 - \lambda)}{\overline{\beta_d}} \right) \frac{1}{(1 - \lambda)} - \frac{1}{1 - \lambda} \right) - \frac{\beta^-}{\overline{\beta_d}} \mathcal{H} + \frac{\beta^-}{\overline{\beta_d}} p_P \\
&= p_N \left(1 + \frac{1}{(1 - \lambda)} - \frac{\beta^-}{\overline{\beta_d}} - \frac{1}{1 - \lambda} \right) - \frac{\beta^-}{\overline{\beta_d}} \mathcal{H} + \frac{\beta^-}{\overline{\beta_d}} p_P \\
&= p_N \left(1 - \frac{\beta^-}{\overline{\beta_d}} \right) - \frac{\beta^-}{\overline{\beta_d}} \mathcal{H} + \frac{\beta^-}{\overline{\beta_d}} p_P
\end{aligned}
\tag{3.68}
$$

3.2.5   discretization of laplacian looking from owner to neighbour

Owner is wet

Extrapolation to get value at N seen from perspective of P, where P is a wet cell:

$$
p_{N,\Gamma}^{P=wet(+)} = \frac{\beta^-}{\overline{\beta_w}} p_N + \left(1 - \frac{\beta^-}{\overline{\beta_w}} \right) p_P - \frac{\beta^-}{\overline{\beta_w}} \mathcal{H}
\tag{3.69}
$$

The pressure laplacian discretization:

$$
\sum_f \left(\frac{1}{a_P} \right)_f (\beta_{f\Gamma}) |\mathbf{s}_f| \frac{(p_N - p_P)_\Gamma}{|\mathbf{d}_f|} = \sum_{f\cancel{V}} \left(\frac{1}{a_P} \right)_f (\beta)_f |\mathbf{s}_f| \frac{p_N - p_P}{|\mathbf{d}_f|}
$$

$$
+ \sum_{f\Gamma} \left(\frac{1}{a_P} \right)_f (\beta)_{f\Gamma} |\mathbf{s}_f| \frac{p_{N,\Gamma}^{P=wet(+)} - p_P}{|\mathbf{d}_f|}
\tag{3.70}
$$

where $(\beta)_{f\Gamma} = \beta^+$, because P is wet. $(\beta)_f$ is used for faces shares either 2 wet or 2 dry cells, hence if there are 2 wet cells use β^+, and if there are 2 dry cells use β^-.

$$\sum_{f\Gamma} \left(\frac{1}{a_P}\right)_f (\beta)_{f\Gamma} |\mathbf{s}_f| \frac{p_{N,\Gamma}^{P=wet(+)} - p_P}{|\mathbf{d}_f|} \tag{3.71}$$

$$= \sum_{f\Gamma} \left(\frac{1}{a_P}\right)_f \beta^+ |\mathbf{s}_f| \frac{\left[\frac{\beta^-}{\overline{\beta_w}}p_N + \left(1 - \frac{\beta^-}{\overline{\beta_w}}\right)p_P - \frac{\beta^-}{\overline{\beta_w}}\mathcal{H}\right] - p_P}{|\mathbf{d}_f|} \tag{3.72}$$

$$= \sum_{f\Gamma} \left(\frac{1}{a_P}\right)_f \beta^+ \frac{|\mathbf{s}_f|}{|\mathbf{d}_f|} \left[\frac{\beta^-}{\overline{\beta_w}}p_N + \left(1 - \frac{\beta^-}{\overline{\beta_w}}\right)p_P - \frac{\beta^-}{\overline{\beta_w}}\mathcal{H} - p_P\right] \tag{3.73}$$

$$= \sum_{f\Gamma} \left(\frac{1}{a_P}\right)_f \beta^+ \frac{|\mathbf{s}_f|}{|\mathbf{d}_f|} \left[\frac{\beta^-}{\overline{\beta_w}}p_N + p_P - \frac{\beta^-}{\overline{\beta_w}}p_P - \frac{\beta^-}{\overline{\beta_w}}\mathcal{H} - p_P\right] \tag{3.74}$$

$$= \sum_{f\Gamma} \left(\frac{1}{a_P}\right)_f \beta^+ \frac{|\mathbf{s}_f|}{|\mathbf{d}_f|} \left[\frac{\beta^-}{\overline{\beta_w}}p_N - \frac{\beta^-}{\overline{\beta_w}}p_P - \frac{\beta^-}{\overline{\beta_w}}\mathcal{H}\right] \tag{3.75}$$

$$= \underbrace{\sum_{f\Gamma} \left(\frac{1}{a_P}\right)_f \frac{|\mathbf{s}_f|}{|\mathbf{d}_f|} \frac{\beta^+\beta^-}{\overline{\beta_w}}}_{a_{PN}} p_N - \underbrace{\left(\frac{1}{a_P}\right)_f \frac{|\mathbf{s}_f|}{|\mathbf{d}_f|} \frac{\beta^+\beta^-}{\overline{\beta_w}}}_{d_P} p_P - \underbrace{\left(\frac{1}{a_P}\right)_f \frac{|\mathbf{s}_f|}{|\mathbf{d}_f|} \frac{\beta^+\beta^-}{\overline{\beta_w}}}_{S_P} \mathcal{H} \tag{3.76}$$

Owner is dry

Extrapolation to get value at N seen from perspective of P, where P is a dry cell:

$$p_{N,\Gamma}^{P=dry(-)} = \frac{\beta^+}{\overline{\beta_d}}p_N + \left(1 - \frac{\beta^+}{\overline{\beta_d}}\right)p_P + \frac{\beta^+}{\overline{\beta_d}}\mathcal{H} \tag{3.77}$$

The pressure laplacian discretization:

$$\sum_{f} \left(\frac{1}{a_P}\right)_f (\beta_{f\Gamma})|\mathbf{s}_f| \frac{(p_N - p_P)_\Gamma}{|\mathbf{d}_f|} = \sum_{f\mathcal{V}} \left(\frac{1}{a_P}\right)_f (\beta)_f |\mathbf{s}_f| \frac{p_N - p_P}{|\mathbf{d}_f|}$$
$$+ \sum_{f\Gamma} \left(\frac{1}{a_P}\right)_f (\beta)_{f\Gamma} |\mathbf{s}_f| \frac{p_{N,\Gamma}^{P=dry(-)} - p_P}{|\mathbf{d}_f|} \tag{3.78}$$

where $(\beta)_{f\Gamma} = \beta^-$, because P is dry. $(\beta)_f$ is used for faces shares either 2 wet or 2 dry cells, hence if there are 2 wet cells use β^+, and if there are 2 dry cells use β^-.

$$\sum_{f\Gamma} \left(\frac{1}{a_P}\right)_f (\beta)_{f\Gamma} |\mathbf{s}_f| \frac{p_{N,\Gamma}^{P=dry(-)} - p_P}{|\mathbf{d}_f|} \tag{3.79}$$

$$= \sum_{f\Gamma} \left(\frac{1}{a_P}\right)_f (\beta)_{f\Gamma} |\mathbf{s}_f| \frac{\left[\frac{\beta^+}{\overline{\beta_d}}p_N + \left(1 - \frac{\beta^+}{\overline{\beta_d}}\right)p_P + \frac{\beta^+}{\overline{\beta_d}}\mathcal{H}\right] - p_P}{|\mathbf{d}_f|} \tag{3.80}$$

$$= \sum_{f\Gamma} \left(\frac{1}{a_P}\right)_f \beta^- \frac{|\mathbf{s}_f|}{|\mathbf{d}_f|} \left[\frac{\beta^+}{\overline{\beta_d}}p_N + \left(1 - \frac{\beta^+}{\overline{\beta_d}}\right)p_P + \frac{\beta^+}{\overline{\beta_d}}\mathcal{H} - p_P\right] \tag{3.81}$$

$$= \sum_{f\Gamma} \left(\frac{1}{a_P}\right)_f \frac{|\mathbf{s}_f|}{|\mathbf{d}_f|} \left[\frac{\beta^-\beta^+}{\overline{\beta_d}}p_N - \frac{\beta^-\beta^+}{\overline{\beta_d}}p_P + \frac{\beta^-\beta^+}{\overline{\beta_d}}\mathcal{H}\right] \tag{3.82}$$

$$= \underbrace{\sum_{f\Gamma} \left(\frac{1}{a_P}\right)_f \frac{|\mathbf{s}_f|}{|\mathbf{d}_f|} \frac{\beta^-\beta^+}{\overline{\beta_d}}}_{a_{PN}} p_N - \underbrace{\left(\frac{1}{a_P}\right)_f \frac{|\mathbf{s}_f|}{|\mathbf{d}_f|} \frac{\beta^-\beta^+}{\overline{\beta_d}}}_{d_P} p_P + \underbrace{\left(\frac{1}{a_P}\right)_f \frac{|\mathbf{s}_f|}{|\mathbf{d}_f|} \frac{\beta^-\beta^+}{\overline{\beta_d}}}_{S_P} \mathcal{H} \tag{3.83}$$

3.2.6 discretization of laplacian looking from neighbour to owner

Owner is wet

Extrapolation to get value at P seen from perspective of N, where P is a wet cell:

$$p_{P,\Gamma}^{P=wet(-)} = \frac{\beta^+}{\overline{\beta_w}}p_P + \left(1 - \frac{\beta^+}{\overline{\beta_w}}\right)p_N + \frac{\beta^+}{\overline{\beta_w}}\mathcal{H} \tag{3.84}$$

The pressure laplacian discretization:

$$\sum_f \left(\frac{1}{a_P}\right)_f (\beta_{f\Gamma})|\mathbf{s}_f|\frac{(p_P - p_N)_\Gamma}{|\mathbf{d}_f|} = \sum_{f\slashed{\Gamma}} \left(\frac{1}{a_P}\right)_f (\beta)_f|\mathbf{s}_f|\frac{p_P - p_N}{|\mathbf{d}_f|}$$

$$+ \sum_{f\Gamma} \left(\frac{1}{a_P}\right)_f (\beta)_{f\Gamma}|\mathbf{s}_f|\frac{p_{P,\Gamma}^{P=wet(-)} - p_N}{|\mathbf{d}_f|} \tag{3.85}$$

where $(\beta)_{f\Gamma} = \beta^-$, because N is dry. $(\beta)_f$ is used for faces shares either 2 wet or 2 dry cells, hence if there are 2 wet cells use β^+, and if there are 2 dry cells use β^-.

$$\sum_{f\Gamma} \left(\frac{1}{a_P}\right)_f (\beta)_{f\Gamma}|\mathbf{s}_f|\frac{p_{P,\Gamma}^{P=wet(-)} - p_N}{|\mathbf{d}_f|} \tag{3.86}$$

$$= \sum_{f\Gamma} \left(\frac{1}{a_P}\right)_f \beta^-|\mathbf{s}_f|\frac{\left[\frac{\beta^+}{\overline{\beta_w}}p_P + \left(1 - \frac{\beta^+}{\overline{\beta_w}}\right)p_N + \frac{\beta^+}{\overline{\beta_w}}\mathcal{H}\right] - p_N}{|\mathbf{d}_f|} \tag{3.87}$$

$$= \sum_{f\Gamma} \left(\frac{1}{a_P}\right)_f \frac{|\mathbf{s}_f|}{|\mathbf{d}_f|}\left[\frac{\beta^-\beta^+}{\overline{\beta_w}}p_P + \left(\beta^- - \frac{\beta^-\beta^+}{\overline{\beta_w}}\right)p_N + \frac{\beta^-\beta^+}{\overline{\beta_w}}\mathcal{H} - \beta^-p_N\right] \tag{3.88}$$

$$= \sum_{f\Gamma} \left(\frac{1}{a_P}\right)_f \frac{|\mathbf{s}_f|}{|\mathbf{d}_f|}\left[\frac{\beta^-\beta^+}{\overline{\beta_w}}p_P + \beta^-p_N - \frac{\beta^-\beta^+}{\overline{\beta_w}}p_N - \frac{\beta^-\beta^+}{\overline{\beta_w}}\mathcal{H} - \beta^-p_N\right] \tag{3.89}$$

$$= \sum_{f\Gamma} \underbrace{-\left(\frac{1}{a_P}\right)_f \frac{|\mathbf{s}_f|}{|\mathbf{d}_f|}\frac{\beta^-\beta^+}{\overline{\beta_w}}p_N}_{d_N} + \underbrace{\left(\frac{1}{a_P}\right)_f \frac{|\mathbf{s}_f|}{|\mathbf{d}_f|}\frac{\beta^-\beta^+}{\overline{\beta_w}}p_P}_{a_{NP}} + \underbrace{\left(\frac{1}{a_P}\right)_f \frac{|\mathbf{s}_f|}{|\mathbf{d}_f|}\frac{\beta^-\beta^+}{\overline{\beta_w}}\mathcal{H}}_{S_N} \tag{3.90}$$

Owner is dry

Extrapolation to get value at P seen from perspective of N, where P is a dry cell:

$$p_{P,\Gamma}^{P=dry(+)} = \frac{\beta^-}{\overline{\beta_d}}p_P + \left(1 - \frac{\beta^-}{\overline{\beta_d}}\right)p_N - \frac{\beta^-}{\overline{\beta_d}}\mathcal{H} \tag{3.91}$$

The pressure laplacian discretization:

$$\sum_f \left(\frac{1}{a_P}\right)_f (\beta_{f\Gamma})|\mathbf{s}_f|\frac{(p_P - p_N)_\Gamma}{|\mathbf{d}_f|} = \sum_{f\slashed{\Gamma}} \left(\frac{1}{a_P}\right)_f (\beta)_f|\mathbf{s}_f|\frac{p_P - p_N}{|\mathbf{d}_f|}$$

$$+ \sum_{f\Gamma} \left(\frac{1}{a_P}\right)_f (\beta)_{f\Gamma}|\mathbf{s}_f|\frac{p_{P,\Gamma}^{P=dry(+)} - p_N}{|\mathbf{d}_f|} \tag{3.92}$$

where $(\beta)_{f\Gamma} = \beta^+$, because N is wet. $(\beta)_f$ is used for faces shares either 2 wet or 2 dry cells, hence if there are 2 wet cells use β^+, and if there are 2 dry cells use β^-.

$$\sum_{f\Gamma} \left(\frac{1}{a_P}\right)_f (\beta)_{f\Gamma} |\mathbf{s}_f| \frac{p_{P,\Gamma}^{P=dry(+)} - p_N}{|\mathbf{d}_f|} \tag{3.93}$$

$$= \sum_{f\Gamma} \left(\frac{1}{a_P}\right)_f \beta^+ |\mathbf{s}_f| \frac{\left[\frac{\beta^-}{\overline{\beta_d}} p_P + \left(1 - \frac{\beta^-}{\overline{\beta_d}}\right) p_N - \frac{\beta^-}{\overline{\beta_d}} \mathcal{H}\right] - p_N}{|\mathbf{d}_f|} \tag{3.94}$$

$$= \sum_{f\Gamma} \left(\frac{1}{a_P}\right)_f \frac{|\mathbf{s}_f|}{|\mathbf{d}_f|} \left[\frac{\beta^+\beta^-}{\overline{\beta_d}} p_P + \left(\beta^+ - \frac{\beta^+\beta^-}{\overline{\beta_d}}\right) p_N - \frac{\beta^+\beta^-}{\overline{\beta_d}} \mathcal{H} - \beta^+ p_N\right] \tag{3.95}$$

$$= \sum_{f\Gamma} \left(\frac{1}{a_P}\right)_f \frac{|\mathbf{s}_f|}{|\mathbf{d}_f|} \left[\frac{\beta^+\beta^-}{\overline{\beta_d}} p_P + \beta^+ p_N - \frac{\beta^+\beta^-}{\overline{\beta_d}} p_N - \frac{\beta^+\beta^-}{\overline{\beta_d}} \mathcal{H} - \beta^+ p_N\right] \tag{3.96}$$

$$= \sum_{f\Gamma} \left(\frac{1}{a_P}\right)_f \frac{|\mathbf{s}_f|}{|\mathbf{d}_f|} \left[\frac{\beta^+\beta^-}{\overline{\beta_d}} p_P - \frac{\beta^+\beta^-}{\overline{\beta_d}} p_N - \frac{\beta^+\beta^-}{\overline{\beta_d}} \mathcal{H}\right] \tag{3.97}$$

$$= \sum_{f\Gamma} \underbrace{-\left(\frac{1}{a_P}\right)_f \frac{|\mathbf{s}_f|}{|\mathbf{d}_f|} \frac{\beta^+\beta^-}{\overline{\beta_d}}}_{d_N} p_N + \underbrace{\left(\frac{1}{a_P}\right)_f \frac{|\mathbf{s}_f|}{|\mathbf{d}_f|} \frac{\beta^+\beta^-}{\overline{\beta_d}}}_{a_{NP}} p_P - \underbrace{\left(\frac{1}{a_P}\right)_f \frac{|\mathbf{s}_f|}{|\mathbf{d}_f|} \frac{\beta^+\beta^-}{\overline{\beta_d}} \mathcal{H}}_{S_N} \tag{3.98}$$

3.2.7 Summarized matrix coefficient contributions

The matrix coefficient contributions from the derivation in the previous section are collected in Table 3.1, where I have also specified the matrix coefficients for non-surface cells.

Table 3.1: Pressure laplacian matrix coefficient contributions

	Looking from P to N			
	P wet	P dry	P & N wet	P & N dry
$d_P\ (a_{PP})$	$-\left(\frac{1}{a_P}\right)_f \frac{\lvert\mathbf{s}_f\rvert}{\lvert\mathbf{d}_f\rvert}\frac{\beta^+\beta^-}{\overline{\beta_w}}$	$-\left(\frac{1}{a_P}\right)_f \frac{\lvert\mathbf{s}_f\rvert}{\lvert\mathbf{d}_f\rvert}\frac{\beta^-\beta^+}{\overline{\beta_d}}$	$-\left(\frac{1}{a_P}\right)_f \beta^+\frac{\lvert\mathbf{s}_f\rvert}{\lvert\mathbf{d}_f\rvert}$	$-\left(\frac{1}{a_P}\right)_f \beta^-\frac{\lvert\mathbf{s}_f\rvert}{\lvert\mathbf{d}_f\rvert}$
a_{PN}	$\left(\frac{1}{a_P}\right)_f \frac{\lvert\mathbf{s}_f\rvert}{\lvert\mathbf{d}_f\rvert}\frac{\beta^+\beta^-}{\overline{\beta_w}}$	$\left(\frac{1}{a_P}\right)_f \frac{\lvert\mathbf{s}_f\rvert}{\lvert\mathbf{d}_f\rvert}\frac{\beta^-\beta^+}{\overline{\beta_d}}$	$\left(\frac{1}{a_P}\right)_f \beta^+\frac{\lvert\mathbf{s}_f\rvert}{\lvert\mathbf{d}_f\rvert}$	$\left(\frac{1}{a_P}\right)_f \beta^-\frac{\lvert\mathbf{s}_f\rvert}{\lvert\mathbf{d}_f\rvert}$
S_P	$\left(\frac{1}{a_P}\right)_f \frac{\lvert\mathbf{s}_f\rvert}{\lvert\mathbf{d}_f\rvert}\frac{\beta^+\beta^-}{\overline{\beta_w}}\mathcal{H}$	$-\left(\frac{1}{a_P}\right)_f \frac{\lvert\mathbf{s}_f\rvert}{\lvert\mathbf{d}_f\rvert}\frac{\beta^-\beta^+}{\overline{\beta_d}}\mathcal{H}$	-	-
$S_{nonOrth}$	$\mathbf{k}\bullet\left(\frac{1}{a_P}\right)_f(\beta^+\nabla p)_f^0$	$\mathbf{k}\bullet\left(\frac{1}{a_P}\right)_f(\beta^-\nabla p)_f^0$	$\mathbf{k}\bullet\left(\frac{1}{a_P}\right)_f(\beta^+\nabla p)_f^0$	$\mathbf{k}\bullet\left(\frac{1}{a_P}\right)_f(\beta^-\nabla p)_f^0$
	Looking from N to P			
	P wet	P dry	P & N wet	P & N dry
$d_N\ (a_{NN})$	$-\left(\frac{1}{a_P}\right)_f \frac{\lvert\mathbf{s}_f\rvert}{\lvert\mathbf{d}_f\rvert}\frac{\beta^-\beta^+}{\overline{\beta_w}}$	$-\left(\frac{1}{a_P}\right)_f \frac{\lvert\mathbf{s}_f\rvert}{\lvert\mathbf{d}_f\rvert}\frac{\beta^+\beta^-}{\overline{\beta_d}}$	$-\left(\frac{1}{a_P}\right)_f \beta^+\frac{\lvert\mathbf{s}_f\rvert}{\lvert\mathbf{d}_f\rvert}$	$-\left(\frac{1}{a_P}\right)_f \beta^-\frac{\lvert\mathbf{s}_f\rvert}{\lvert\mathbf{d}_f\rvert}$
a_{NP}	$\left(\frac{1}{a_P}\right)_f \frac{\lvert\mathbf{s}_f\rvert}{\lvert\mathbf{d}_f\rvert}\frac{\beta^-\beta^+}{\overline{\beta_w}}$	$\left(\frac{1}{a_P}\right)_f \frac{\lvert\mathbf{s}_f\rvert}{\lvert\mathbf{d}_f\rvert}\frac{\beta^+\beta^-}{\overline{\beta_d}}$	$\left(\frac{1}{a_P}\right)_f \beta^+\frac{\lvert\mathbf{s}_f\rvert}{\lvert\mathbf{d}_f\rvert}$	$\left(\frac{1}{a_P}\right)_f \beta^-\frac{\lvert\mathbf{s}_f\rvert}{\lvert\mathbf{d}_f\rvert}$
S_N	$-\left(\frac{1}{a_P}\right)_f \frac{\lvert\mathbf{s}_f\rvert}{\lvert\mathbf{d}_f\rvert}\frac{\beta^-\beta^+}{\overline{\beta_w}}\mathcal{H}$	$\left(\frac{1}{a_P}\right)_f \frac{\lvert\mathbf{s}_f\rvert}{\lvert\mathbf{d}_f\rvert}\frac{\beta^+\beta^-}{\overline{\beta_d}}\mathcal{H}$	-	-
$S_{nonOrth}$	$\mathbf{k}\bullet\left(\frac{1}{a_P}\right)_f(\beta^+\nabla p)_f^{i-1}$	$\mathbf{k}\bullet\left(\frac{1}{a_P}\right)_f(\beta^-\nabla p)_f^{i-1}$	$\mathbf{k}\bullet\left(\frac{1}{a_P}\right)_f(\beta^+\nabla p)_f^{i-1}$	$\mathbf{k}\bullet\left(\frac{1}{a_P}\right)_f(\beta^-\nabla p)_f^{i-1}$
	Definitions			
	$\overline{\beta_d}=\lambda\beta^+ +(1-\lambda)\beta^-$		$\overline{\beta_w}=\lambda\beta^- +(1-\lambda)\beta^+$	
	$\beta^+=\frac{1}{\rho^+}$		$\beta^-=\frac{1}{\rho^-}$	
	$\lambda=\frac{\alpha_P-0.5}{\alpha_P-\alpha_N}$		$\mathbf{x_\Gamma}=\mathbf{x_P}+\lambda\mathbf{d_f}$	
	$\mathcal{H}=(\rho^- -\rho^+)\mathbf{g}\bullet\mathbf{x_\Gamma}$		$f_x=\vec{fN}/\vec{PN}$	
	$\underbrace{\left(\frac{1}{a_P}\right)_f\lvert\mathbf{s}_f\rvert}_{\text{gammaMagSf}}\ \underbrace{\frac{1}{\lvert\mathbf{d}_f\rvert}}_{\text{deltaCoeffs}}\ \underbrace{\frac{\beta^+\beta^-}{\overline{\beta_w}}}_{\text{surfaceCoeffs}}$		$(\beta\nabla p)_f^{i-1}=f_x(\beta\nabla p)_P^{i-1}+(1-f_x)(\beta\nabla p)_N^{i-1}$	

3.3 Implementation

This section presents how the theory described in Section 3.2 has been implemented by modifying the class `gaussLaplacianScheme` in two steps.

1. First step is to modify the class, such that it only works for a single combination of template parameters, which is `Type = scalar` and `GType = scalar`. This is implemented in `myScalarGaussLaplacianScheme`.

2. The second step is to implement the new theory valid for scalar diffusion and scalar dependent variable. This is implemented in `GFMGaussLaplacianScheme`.

The starting point is our own copy of `gaussLaplacianScheme` that we created in Section 2.5 under the new name `myGaussLaplacianScheme`.

3.3.1 Remove template functionality

The first aspect to consider is how to remove the template functionality of the class, since the presented theory is intended for the dynamic pressure field `p_rgh`, which is a cell-centred scalar field (In OpenFOAM syntax: `volScalarField`). The first template parameter `Type` is substituted with `scalar` and the second template parameter `GType` is substituted with `scalar`. The specialisation of the member function `fvmLaplacian` and `fvcLaplacian` is moved from `discontinuousGaussLaplacianSchemes.C` to `discontinuousGaussLaplacianScheme.C`, because the new class will only contain this specialisation with the template parameter `Type = scalar`.

First we need to create a new copy of the `myGaussLaplacianScheme` called `myScalarGaussLaplacianScheme`. This performed by the following terminal commands:

```
cp -r $WM_PROJECT_USER_DIR/src/finiteVolume/finiteVolume/laplacianSchemes/myGaussLaplacianScheme \
$WM_PROJECT_USER_DIR/src/finiteVolume/finiteVolume/laplacianSchemes/myScalarGaussLaplacianScheme
cd $WM_PROJECT_USER_DIR/src/finiteVolume/finiteVolume/laplacianSchemes/myScalarGaussLaplacianScheme
mv myGaussLaplacianScheme.C myScalarGaussLaplacianScheme.C
mv myGaussLaplacianScheme.H myScalarGaussLaplacianScheme.H
mv myGaussLaplacianSchemes.C myScalarGaussLaplacianSchemes.C
sed -i s/myGaussLaplacianScheme/myScalarGaussLaplacianScheme/g myScalarGaussLaplacianScheme.H
sed -i s/myGaussLaplacianScheme/myScalarGaussLaplacianScheme/g myScalarGaussLaplacianScheme.C
sed -i s/myGaussLaplacianScheme/myScalarGaussLaplacianScheme/g myScalarGaussLaplacianSchemes.C
sed -i s/'TypeName("myGauss");'/'TypeName("myScalarGauss");'/g myScalarGaussLaplacianScheme.H
```

We are now ready to modify the files. The modification steps are

1. Remove the declaration and definition of the function `gammaSnGradCorr`.

   ```
   sed -i -e '66,71d' myScalarGaussLaplacianScheme.H
   sed -i -e '91,132d' myScalarGaussLaplacianScheme.C
   ```

2. Remove `template<class Type, class GType>` in source and header.

   ```
   sed -i '/template<class Type, class GType>/d' myScalarGaussLaplacianScheme.H
   sed -i '/template<class Type, class GType>/d' myScalarGaussLaplacianScheme.C
   ```

3. Swap `myScalarGaussLaplacianScheme<Type, GType>` with `myScalarGaussLaplacianScheme` in source.

   ```
   sed -i s/'myScalarGaussLaplacianScheme<Type, GType>'/'myScalarGaussLaplacianScheme'/g \
   myScalarGaussLaplacianScheme.C
   ```

4. Remove inclusion of `myScalarGaussLaplacianScheme.C` in header file.

   ```
   sed -i -e '175,180d' myScalarGaussLaplacianScheme.H
   ```

5. Remove macro `defineFvmLaplacianScalarGamma` and the calls to this macro.

   ```
   sed -i -e '137,164d' myScalarGaussLaplacianScheme.H
   ```

6. Copy implementation of

   ```
   myScalarGaussLaplacianScheme::fvcLaplacian
   (
       const GeometricField<scalar, fvsPatchField, surfaceMesh>& gamma,
       const GeometricField<scalar, fvPatchField, volMesh>& vf
   )
   ```

   and

   ```
   myScalarGaussLaplacianScheme::fvmLaplacian
   (
       const GeometricField<scalar, fvsPatchField, surfaceMesh>& gamma,
       const GeometricField<scalar, fvPatchField, volMesh>& vf
   )
   ```

   from `myScalarGaussLaplacianSchemes.C` to `myScalarGaussLaplacianScheme.C` and remove line shifts \.

7. Remove `myScalarGaussLaplacianSchemes.C`

8. Replace template parameter `Type` and `GType` with `scalar` in header file.

   ```
   sed -i s/'<Type, GType>'/'<scalar, scalar>'/g myScalarGaussLaplacianScheme.H
   sed -i s/'<GType>'/'<scalar>'/g myScalarGaussLaplacianScheme.H
   sed -i s/'<Type>'/'<scalar>'/g myScalarGaussLaplacianScheme.H
   ```

9. Replace template parameter `Type` and `GType` with `scalar` in source file.

   ```
   sed -i s/'<GType>'/'<scalar>'/g myScalarGaussLaplacianScheme.C
   sed -i s/'<Type>'/'<scalar>'/g myScalarGaussLaplacianScheme.C
   ```

10. Add `makeMyGaussLaplacianScheme(SS)`

    ```
    // Add the patch constructor functions to the hash tables
    #define makeMyGaussLaplacianScheme(SS)\
        typedef Foam::scalar Type;\
        typedef Foam::scalar GType;\
        typedef Foam::fv::SS SS##Type##GType;\
        defineTypeName(SS##Type##GType);\
        namespace Foam\
        {\
            namespace fv\
            {\
                typedef SS SS##Type##GType;\
    \
                laplacianScheme<Type, GType>::\
                    addIstreamConstructorToTable<SS>\
    ```

```
                add##SS##Type##GType##IstreamConstructorToTable_;\
        }\
    }
```

```
// Define symbol lookup:
//  SS: Name of current class
makeMyGaussLaplacianScheme(myScalarGaussLaplacianScheme)
```

after `#include "fvMatrices.H"` and before `namespace Foam` in myScalarGaussLaplacianScheme.C.

11. Open `Make/files`

    ```
    vi $WM_PROJECT_USER_DIR/src/finiteVolume/Make/files
    ```

    and modify the content to

    ```
    laplacianSchemes = finiteVolume/laplacianSchemes
    $(laplacianSchemes)/myGaussLaplacianScheme/myGaussLaplacianSchemes.C
    $(laplacianSchemes)/myScalarGaussLaplacianScheme/myScalarGaussLaplacianScheme.C
    LIB = $(FOAM_USER_LIBBIN)/libmyFiniteVolume
    ```

12. Compile:

    ```
    wmake $WM_PROJECT_USER_DIR/src/finiteVolume
    ```

    The library is located at:

    ```
    $FOAM_USER_LIBBIN/libmyFiniteVolume.so
    ```

    Test implementation:

    ```
    cp -r $FOAM_TUTORIALS/incompressible/icoFoam/cavity/cavity $FOAM_RUN/myScalarGaussCavity
    sed -i '/Gauss linear orthogonal;/c \
    default none; \
    laplacian(nu,U) Gauss linear orthogonal; \
    laplacian((1|A(U)),p) myScalarGauss linear orthogonal;' \
    $FOAM_RUN/myScalarGaussCavity/system/fvSchemes
    sed -i '$a libs ("libmyFiniteVolume.so");' $FOAM_RUN/myScalarGaussCavity/system/controlDict
    blockMesh -case $FOAM_RUN/myScalarGaussCavity
    icoFoam -case $FOAM_RUN/myScalarGaussCavity
    ```

We have now a working copy of the laplacian scheme that only works for a scalar dependent variable and a scalar diffusion parameter. Now we can proceed with the implementation of the theory.

3.3.2 Implementation of GFM-method

Until now we have just modified the code and retained a part of the existing functionalities. It is time to implement the GFM-method in the laplacian operator.

New class: interfaceJump

I needed to implement several new functions to implement the method. The new member functions are implemented in a separate class, that is constructed at the beginning of the function `fvmLaplacianUncorrected`. I have chosen to implement the functions in a separate class to make it possible to use the same functions in other classes, without having multiple implementations to

debug.

The source files for the class is `interfaceJump.H` and `interfaceJump.C`. Please open the source files to follow the explanation.

Equation (3.40) is implemented in the private member function `interfaceJump::calcLambda()`, and it uses another private member function `interfaceJump::identifySurfaceFaces()` to identify the interface faces, which needs to be treated with the GFM one-sided interpolation method. `interfaceJump::calcLambda()` is called in the class constructor, when a new object is created.

The class provides a public function `interfaceJump::H(label facei)`, which evaluates Equation (3.35) for a single face with index `facei`. $\mathcal{H}$ is calculated at the position of the interface defined by Equation (3.41).

$\overline{\beta_w}$ defined in Equation (3.48) is calculated by `interfaceJump::dimBetaBarWet(label facei)` which returns the value with dimensions or `interfaceJump::betaBarWet(label facei)` which just returns a scalar. The input is an index of an interface face `facei`, and the function then reads the calculated value of λ for `facei`.

$\overline{\beta_d}$ defined in Equation (3.60) is calculated by `interfaceJump::dimBetaBarDry(label facei)` which returns the value with dimensions or `interfaceJump::betaBarDry(label facei)` which just returns a scalar.

The class constructor needs a reference to the dependent variable `vf` and the void fraction field for the heavy phase`alpha1` which corresponds to α_1 in the theory. From these two input the needed variables are initialised in the initializer list.

The class provides a series of access function, which give the user access to read the private data members. I have commented the code in the class, so that it should be self explaining together with this text.

Place the class folder `interfaceJump` in the folder

`$WM_PROJECT_USER_DIR/src/finiteVolume/finiteVolume`

The compilation of this class into the library is presented in the next section.

Create new copy myScalarGaussLaplacianScheme class

The starting point is the class `myScalarGaussLaplacianScheme`, which was created in Section 3.3.1. We are now at the final step, where the theory needs to be implemented. First we create new copy of `myScalarGaussLaplacianScheme` called `GFMGaussLaplacianScheme` with terminal commands:

```
cp -r $WM_PROJECT_USER_DIR/src/finiteVolume/finiteVolume/laplacianSchemes\
/myScalarGaussLaplacianScheme $WM_PROJECT_USER_DIR/src/finiteVolume/finiteVolume\
/laplacianSchemes/GFMGaussLaplacianScheme
cd $WM_PROJECT_USER_DIR/src/finiteVolume/finiteVolume/laplacianSchemes/GFMGaussLaplacianScheme
mv myScalarGaussLaplacianScheme.C GFMGaussLaplacianScheme.C
mv myScalarGaussLaplacianScheme.H GFMGaussLaplacianScheme.H
sed -i s/myScalarGaussLaplacianScheme/GFMGaussLaplacianScheme/g GFMGaussLaplacianScheme.H
sed -i s/myScalarGaussLaplacianScheme/GFMGaussLaplacianScheme/g GFMGaussLaplacianScheme.C
sed -i s/'TypeName("myScalarGauss");'/'TypeName("GFMGauss");'/g GFMGaussLaplacianScheme.H
sed -i '/myScalarGaussLaplacianScheme.C/a \
$(laplacianSchemes)/GFMGaussLaplacianScheme/GFMGaussLaplacianScheme.C\
interfaceJump = finiteVolume/interfaceJump\
$(interfaceJump)/interfaceJump.C' $WM_PROJECT_USER_DIR/src/finiteVolume/Make/files
```

```
wmake $WM_PROJECT_USER_DIR/src/finiteVolume
```

Before we proceed we test that new copy `GFMGaussLaplacianScheme` works:

```
cp -r $FOAM_TUTORIALS/incompressible/icoFoam/cavity/cavity $FOAM_RUN/GFMGaussCavity
sed -i '/Gauss linear orthogonal;/c \
default none; \
laplacian(nu,U) Gauss linear orthogonal; \
laplacian((1|A(U)),p) GFMGauss linear orthogonal;' \
$FOAM_RUN/GFMGaussCavity/system/fvSchemes
sed -i '$a libs ("libmyFiniteVolume.so");' $FOAM_RUN/GFMGaussCavity/system/controlDict
blockMesh -case $FOAM_RUN/GFMGaussCavity
icoFoam -case $FOAM_RUN/GFMGaussCavity
```

Modifications in GFMGaussLaplacianScheme class

The header `myGaussLaplacianScheme.H` is modified as follows. After

```
#include "laplacianScheme.H"
```

insert

```
#include "interfaceJump.H"
```

to include the new class with functions needed by GFM.
After

```
class GFMGaussLaplacianScheme
:
    public fv::laplacianScheme<scalar, scalar>
{
```

insert

```
    // Private Member Data

        //- VOF field reference
        const volScalarField& alpha1_;

        //- Density field reference
        const volScalarField& rho_;
```

to declare `alpha1_` needed to construct instance of `interfaceJump` class. `rho_` is needed in this class.
Replace

```
        //- Construct null
        GFMGaussLaplacianScheme(const fvMesh& mesh)
        :
            laplacianScheme<scalar, scalar>(mesh)
        {}

        //- Construct from Istream
        GFMGaussLaplacianScheme(const fvMesh& mesh, Istream& is)
        :
```

```
        laplacianScheme<scalar, scalar>(mesh, is)
    {}


    //- Construct from mesh, interpolation and snGradScheme schemes
    GFMGaussLaplacianScheme
    (
        const fvMesh& mesh,
        const tmp<surfaceInterpolationScheme<scalar>>& igs,
        const tmp<snGradScheme<scalar>>& sngs
    )
    :
        laplacianScheme<scalar, scalar>(mesh, igs, sngs)
    {}
```

with

```
    //- Construct null
    GFMGaussLaplacianScheme(const fvMesh& mesh)
    :
        laplacianScheme<scalar, scalar>(mesh),
        alpha1_(mesh.lookupObject<volScalarField>("alpha.water")),
        rho_(mesh.lookupObject<volScalarField>("rho"))
    {}


    //- Construct from Istream
    GFMGaussLaplacianScheme(const fvMesh& mesh, Istream& is)
    :
        laplacianScheme<scalar, scalar>(mesh, is),
        alpha1_(mesh.lookupObject<volScalarField>("alpha.water")),
        rho_(mesh.lookupObject<volScalarField>("rho"))
    {}


    //- Construct from mesh, interpolation and snGradScheme schemes
    GFMGaussLaplacianScheme
    (
        const fvMesh& mesh,
        const tmp<surfaceInterpolationScheme<scalar>>& igs,
        const tmp<snGradScheme<scalar>>& sngs
    )
    :
        laplacianScheme<scalar, scalar>(mesh, igs, sngs),
        alpha1_(mesh.lookupObject<volScalarField>("alpha.water")),
        rho_(mesh.lookupObject<volScalarField>("rho"))
    {}
```

to initialise the new private data members `alpha1_` and `rho_`.
Add void fraction field and density as inputs to `fvmLaplacianUncorrected` by replacing

```
    static tmp<fvMatrix<scalar>> fvmLaplacianUncorrected
    (
        const surfaceScalarField& gammaMagSf,
        const surfaceScalarField& deltaCoeffs,
        const GeometricField<scalar, fvPatchField, volMesh>&
    );
```

with

```
        static tmp<fvMatrix<scalar>> fvmLaplacianUncorrected
        (
            const surfaceScalarField& gammaMagSf,
            const surfaceScalarField& deltaCoeffs,
            const GeometricField<scalar, fvPatchField, volMesh>&,
            const volScalarField& alpha1,
            const volScalarField& rho
        );
```

We cannot use private data members in a static function, so we need pass them to the function as references.

The source `GFMGaussLaplacianScheme.C` is modified as follows.
The function definition of `fvmLaplacianUncorrected` is modified according to the declaration, which means that we need to replace

```
GFMGaussLaplacianScheme::fvmLaplacianUncorrected
(
    const surfaceScalarField& gammaMagSf,
    const surfaceScalarField& deltaCoeffs,
    const GeometricField<scalar, fvPatchField, volMesh>& vf
)
```

with

```
GFMGaussLaplacianScheme::fvmLaplacianUncorrected
(
    const surfaceScalarField& gammaMagSf,
    const surfaceScalarField& deltaCoeffs,
    const GeometricField<scalar, fvPatchField, volMesh>& vf,
    const volScalarField& alpha1,
    const volScalarField& rho
)
```

Before

```
    tmp<fvMatrix<scalar>> tfvm
```

we add

```
    interfaceJump intface(vf,alpha1);
    const boolList& sFaces = intface.sFaces();
```

to create an instance `intface` of class `interfaceJump`. A boolean list with length equal to number of faces is returned from `interfaceJump`. For each face index `sFaces` will say if the face is a surface face (true) or a regular face (false).
The unit of the matrix is corrected by replacing

```
deltaCoeffs.dimensions()*gammaMagSf.dimensions()*vf.dimensions()
```

with

```
deltaCoeffs.dimensions()*gammaMagSf.dimensions()*vf.dimensions()/dimDensity
```

because the density is now part of the pressure laplacian operator according to the theory presented.
Replace old matrix assembly

```
    fvm.upper() = deltaCoeffs.primitiveField()*gammaMagSf.primitiveField();
    fvm.negSumDiag();
```

with a new matrix assembly according to Section 2.4

```
    //- Generate upper diagonal matrix coefficients
    for(label facei=0; facei<fvm.lduAddr().upperAddr().size(); facei++)
    {
        label owner = fvm.lduAddr().upperAddr()[facei];
        label neighbour = fvm.lduAddr().lowerAddr()[facei];
        if (sFaces[facei])
        {
            // This is a surface cell using GFM interpolation.
            if (alpha1[owner] > 0.5)
            {
                // P is wet:
                // ---------
                // - Looking from P -> N
                fvm.diag()[owner] -= deltaCoeffs.primitiveField()[facei]
                                    * gammaMagSf.primitiveField()[facei]
                                    * intface.betaPlus().value()
                                    * intface.betaMinus().value()
                                    / intface.betaBarWet(facei);

                fvm.upper()[facei] += deltaCoeffs.primitiveField()[facei]
                                     * gammaMagSf.primitiveField()[facei]
                                     * intface.betaPlus().value()
                                     * intface.betaMinus().value()
                                     / intface.betaBarWet(facei);

                fvm.source()[owner] += deltaCoeffs.primitiveField()[facei]
                                      * gammaMagSf.primitiveField()[facei]
                                      * intface.betaPlus().value()
                                      * intface.betaMinus().value()
                                      / intface.betaBarWet(facei)
                                      * intface.H(facei).value();

                // - Looking from N -> P
                fvm.diag()[neighbour] -= deltaCoeffs.primitiveField()[facei]
                                        * gammaMagSf.primitiveField()[facei]
                                        * intface.betaPlus().value()
                                        * intface.betaMinus().value()
                                        / intface.betaBarWet(facei);

                fvm.source()[neighbour] -= deltaCoeffs.primitiveField()[facei]
                                          * gammaMagSf.primitiveField()[facei]
                                          * intface.betaPlus().value()
                                          * intface.betaMinus().value()
```

```
                                    / intface.betaBarWet(facei)
                                    * intface.H(facei).value();


        }
        else
        {
            // P is dry
            // ---------
            // - Looking from P -> N
            fvm.diag()[owner] -= deltaCoeffs.primitiveField()[facei]
                                * gammaMagSf.primitiveField()[facei]
                                * intface.betaPlus().value()
                                * intface.betaMinus().value()
                                / intface.betaBarDry(facei);

            fvm.upper()[facei] += deltaCoeffs.primitiveField()[facei]
                                * gammaMagSf.primitiveField()[facei]
                                * intface.betaPlus().value()
                                * intface.betaMinus().value()
                                / intface.betaBarDry(facei);

            fvm.source()[owner] -= deltaCoeffs.primitiveField()[facei]
                                 * gammaMagSf.primitiveField()[facei]
                                 * intface.betaPlus().value()
                                 * intface.betaMinus().value()
                                 / intface.betaBarDry(facei)
                                 * intface.H(facei).value();

            // - Looking from N -> P
            fvm.diag()[neighbour] -= deltaCoeffs.primitiveField()[facei]
                                   * gammaMagSf.primitiveField()[facei]
                                   * intface.betaPlus().value()
                                   * intface.betaMinus().value()
                                   / intface.betaBarDry(facei);

            fvm.source()[neighbour] += deltaCoeffs.primitiveField()[facei]
                                     * gammaMagSf.primitiveField()[facei]
                                     * intface.betaPlus().value()
                                     * intface.betaMinus().value()
                                     / intface.betaBarDry(facei)
                                     * intface.H(facei).value();
        }
    }
    else
    {
        // This is not a surface cell.
        // Both cells around the face are either wet or dry
        if (alpha1[owner] > 0.5)
        {
            // P & N are both wet
            // Assign contributions to the diagonal matrix coefficient:
            // - Looking from P -> N
            fvm.diag()[owner] -= deltaCoeffs.primitiveField()[facei]
                                * gammaMagSf.primitiveField()[facei]
```

```
                                     * intface.betaPlus().value();

            // - Looking from N -> P
            fvm.diag()[neighbour] -= deltaCoeffs.primitiveField()[facei]
                                  * gammaMagSf.primitiveField()[facei]
                                  * intface.betaPlus().value();

            // Assign the matrix coefficient in the upper triangle:
            fvm.upper()[facei] += deltaCoeffs.primitiveField()[facei]
                               * gammaMagSf.primitiveField()[facei]
                               * intface.betaPlus().value();
        }
        else
        {
            // P & N are both dry
            // Assign contributions to the diagonal matrix coefficient:
            // - Looking from P -> N
            fvm.diag()[owner] -= deltaCoeffs.primitiveField()[facei]
                              * gammaMagSf.primitiveField()[facei]
                              * intface.betaMinus().value();

            // - Looking from N -> P
            fvm.diag()[neighbour] -= deltaCoeffs.primitiveField()[facei]
                                  * gammaMagSf.primitiveField()[facei]
                                  * intface.betaMinus().value();

            // Assign the matrix coefficient in the upper triangle:
            fvm.upper()[facei] += deltaCoeffs.primitiveField()[facei]
                               * gammaMagSf.primitiveField()[facei]
                               * intface.betaMinus().value();

        }
    }
}
```

The loop runs through all internal faces in the mesh. For each face the owner and neighbour cell is assigned.

The first if statement `if (sFaces[facei])` checks if the current face is a surface face or a regular face. If the condition is true it is a surface face and we need to perform the special one-sided GFM interpolation. However before we do this we have to check if the owner cell is wet or dry, and this is the purpose of the condition `if (alpha1[owner] > 0.5)`. If the owner cell is wet ($\alpha > 0.5$) then we will use column 1 of Table 3.1 first looking from P to N, and then looking from N to P as seen from the comments in the code. When we look from N to P it is important to note that I do not assign any contribution to `fvm.lower()`, because I want to tell OpenFOAM that this matrix is symmetric.

If the face is a regular face we will enter the `else` statement of the first `if` condition, and once again we need to find out if both cell P and N are wet $\alpha > 0.5$ or otherwise dry. For a wet case the heavy inverse density `betaPlus()` is used and for a dry case the light inverse density `betaMinus()` is used. Otherwise it is just the standard discretization procedure for the regular face.

The next step is to modify the implementation of boundary conditions. After

```
        const fvsPatchScalarField& pDeltaCoeffs =
            deltaCoeffs.boundaryField()[patchi];
```

insert

```
        const fvPatchScalarField& pRho = rho.boundaryField()[patchi];
```

which is the boundary field of the density for the current patch index. We need to multiply with
the inverse density, which just dividing the existing expressions with the density. Therefore replace

```
        fvm.internalCoeffs()[patchi] =
            pGamma*pvf.gradientInternalCoeffs(pDeltaCoeffs);
        fvm.boundaryCoeffs()[patchi] =
            -pGamma*pvf.gradientBoundaryCoeffs(pDeltaCoeffs);
```

with

```
        fvm.internalCoeffs()[patchi] =
            pGamma/pRho*pvf.gradientInternalCoeffs(pDeltaCoeffs);
        fvm.boundaryCoeffs()[patchi] =
            -pGamma/pRho*pvf.gradientBoundaryCoeffs(pDeltaCoeffs);
```

and replace

```
        fvm.internalCoeffs()[patchi] = pGamma*pvf.gradientInternalCoeffs();
        fvm.boundaryCoeffs()[patchi] = -pGamma*pvf.gradientBoundaryCoeffs();
```

with

```
fvm.internalCoeffs()[patchi] = pGamma/pRho*pvf.gradientInternalCoeffs();
fvm.boundaryCoeffs()[patchi] = -pGamma/pRho*pvf.gradientBoundaryCoeffs();
```

Now we are done with `fvmLaplacianUncorrected`. Please note that I have not ensured that density
field is step wise interpolated in the boundary conditions. When using this scheme in a solver it is
important to step interpolate the density so that it only can take the value of the heavy phase or
the light phase.

The function `fvcLaplacian(vf)` is modified by replacing

```
    tmp<GeometricField<scalar, fvPatchField, volMesh>> tLaplacian
    (
        fvc::div(this->tsnGradScheme_().snGrad(vf)*mesh.magSf())
    );
```

with

```
    const volScalarField betavf(vf/rho_);

    tmp<GeometricField<scalar, fvPatchField, volMesh>> tLaplacian
    (
        fvc::div(this->tsnGradScheme_().snGrad(betavf)*mesh.magSf())
    );
```

to account for the introduced density field.
The function `fvmLaplacian` is modified by replacing

```
    tmp<fvMatrix<scalar>> tfvm = fvmLaplacianUncorrected
    (
        gammaMagSf,
        this->tsnGradScheme_().deltaCoeffs(vf),
        vf
    );
```

with

```
    tmp<fvMatrix<scalar>> tfvm = fvmLaplacianUncorrected
    (
        gammaMagSf,
        this->tsnGradScheme_().deltaCoeffs(vf),
        vf,
        alpha1_,
        rho_
    );
```

to pass in the new inputs to `fvmLaplacianUncorrected`. Furthermore add

```
        const volScalarField betavf(vf/rho_);
```

after

```
    if (this->tsnGradScheme_().corrected())
    {
```

and replace

```
gammaMagSf*this->tsnGradScheme_().correction(vf)
```

with

```
gammaMagSf*this->tsnGradScheme_().correction(betavf)
```

at two locations which are

```
            fvm.faceFluxCorrectionPtr() = new
            GeometricField<scalar, fvsPatchField, surfaceMesh>
            (
                gammaMagSf*this->tsnGradScheme_().correction(vf)
            );
```

and

```
            fvm.source() -=
                mesh.V()*
                fvc::div
                (
                    gammaMagSf*this->tsnGradScheme_().correction(vf)
                )().primitiveField();
```

In the function `fvcLaplacian(gamma,vf)` replace

```
    tmp<GeometricField<scalar, fvPatchField, volMesh>> tLaplacian
    (
        fvc::div(gamma*this->tsnGradScheme_().snGrad(vf)*mesh.magSf())
    );
```

with

```
    const volScalarField betavf(vf/rho_);

    tmp<GeometricField<scalar, fvPatchField, volMesh>> tLaplacian
    (
        fvc::div(gamma*this->tsnGradScheme_().snGrad(betavf)*mesh.magSf())
    );
```

to account for the density field.

Now we are ready to compile by `wmake $WM_PROJECT_USER_DIR/src/finiteVolume` and test the code, which is described in the next chapter.

Chapter 4

Inclined square test

Section 4.1 presents all the settings which the two test cases have in common. Section presents how the two cases differ from each other.

4.1   Common case description

The newly implemented scheme is tested using a very simple test case with a square domain with side lengths equal to 1. The square is inclined 26.6 °. A water surface is defined at a height of 0.7 m on the z-axis and the surface is parallel to the xy-plane. The void fraction field in Figure 4.1 is specified with the pre-processing utility `setAlphaField`.
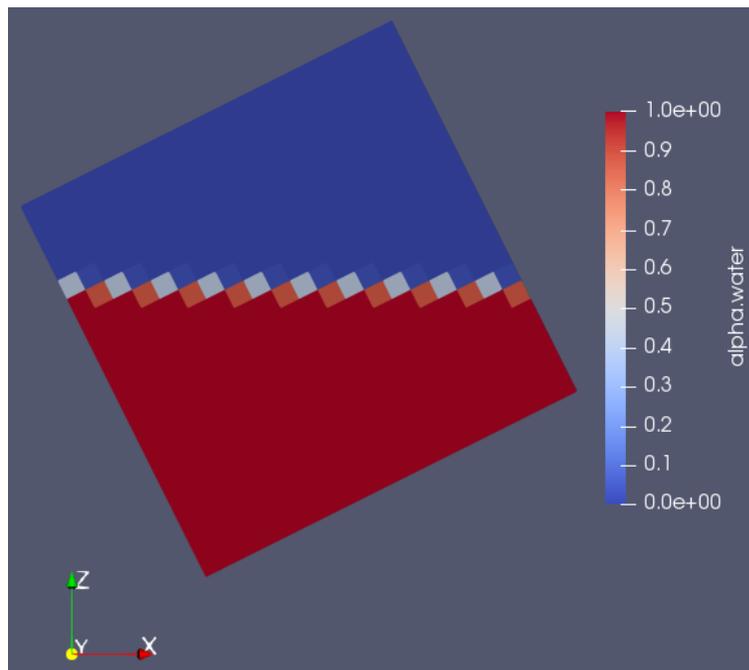
Figure 4.1: Void fraction distribution for the heavy phase.

I have used the case `standingWave` found at

`$FOAM_TUTORIALS/multiphase/interIsoFoam/standingWave`

as my starting point, and modified the case according to the following description.

4.1.1 Boundary conditions

There are three fields, `p_rgh`, `alpha.water` and `U`. The case is defined as a 2D case, so the front and back patches are defined as `empty` for all three fields. For `p_rgh` and `alpha.water` the domain boundary condition is set to a zero normal gradient known as `zeroGradient`. The mathematical expression for this Neumann condition is for a generic parameter ϕ given by

$$\frac{\partial \phi}{\partial \mathbf{n}} = 0 \tag{4.1}$$

where $\mathbf{n}$ is the normal vector going out the domain.

The velocity field `U` is initialised to zero, and the boundary condition is a dirichlet condition, where the value is fixed to zero.

4.1.2 Constant

The diffusion coefficient for the modified scheme has been added

```
gamma           1;
```

to the end of `constant/transportDict`.

4.1.3 System

The dictionaries `refineMeshDict1`, `refineMeshDict2`, `topoSetDict1`, `topoSetDict2` and `decomposeParDict` are removed.

Settings in setAlphaFieldDict

The original settings given by

```
field       alpha.water;
type        sin;
direction   (1 0 0);
up          (0 0 1);
origin      (-0.5 0 0.5);
period      2;
amplitude   0.05;
```

is substituted with new settings given by

```
field       alpha.water;
type        plane;
origin      (0 0 0.7);
direction   (0 0 1);
```

that describes a plane water surface parallel to the xy-plane elevated 0.7 m in the z-direction.

Settings in blockMeshDict

The mesh size changed from 50 x 50 to 20 x 20.

4.2 Case specific settings

4.2.1 interIsoFoam

The dynamic pressure distribution using the standard Gauss discretization is computed with the existing solver `interIsoFoam`.

Settings in controlDict

We only need to compute a single time step. The dictionary is modified to reflect the settings given by

```
application     interIsoFoam;
endTime         0.001;
adjustTimeStep  no;
```

Settings in fvSchemes

The original Gauss discretization scheme is defined as

```
default  Gauss linear corrected;
```

4.2.2 laplacianVOFFoam

A new solver is needed, since it would require more modifications than only the laplacian operator to implement the GFM method in `interIsoFoam`. However I need many of the two-phase functionalities from `interIsoFoam` solver to make a valid solver for a still water surface.
The dynamic pressure distribution using the modified Gauss discretization with the Ghost Fluid Method is computed with a new solver `laplacianVOFFoam`, which is created based on `createFields.H` from `interIsoFoam` and the solver `laplacianFoam`. The solver is only valid for a still water surface.

Solver definition

The solver definition is given by

```
                            ──── laplacianVOFFoam.C ────
57  #include "fvCFD.H"
58  #include "immiscibleIncompressibleTwoPhaseMixture.H"
59
60  // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
61
62  int main(int argc, char *argv[])
63  {
64      argList::addNote
65      (
66          "Pressure equation solver for dynamic pressure in two non-moving phases."
67      );
68
69      #include "setRootCase.H"
70      #include "createTime.H"
71      #include "createMesh.H"
72      #include "createFields.H"
73
74      // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
75
76      Info<< "\nCalculating dynamic pressure distribution\n" << endl;
77
78      fvScalarMatrix pEqn
79      (
80          fvm::laplacian(gamma, p_rgh)
81      );
82      pEqn.setReference(pRefCell, pRefValue);
83      solve(pEqn);
84
```

```
 85        p == p_rgh + rho*gh;
 86
 87        p += dimensionedScalar
 88            (
 89            "p",
 90            p.dimensions(),
 91            pRefValue - getRefCellValue(p, pRefCell)
 92        );
 93        p_rgh = p - rho*gh;
 94
 95        runTime++;
 96        runTime.write();
 97        runTime.printExecutionTime(Info);
 98
 99        Info<< "End\n" << endl;
100
101        return 0;
102    }
```

The diffusion coefficient `gamma` is `rAUf` in the interIsoFoam solver. This term comes from the momentum equation, but this is not implemented in this simple solver, because The `createFields.H` file is given by

─────────────────────── createFields.H ───────────────────────
```
 1   Info<< "Reading field p_rgh\n" << endl;
 2   volScalarField p_rgh
 3   (
 4       IOobject
 5       (
 6           "p_rgh",
 7           runTime.timeName(),
 8           mesh,
 9           IOobject::MUST_READ,
10           IOobject::AUTO_WRITE
11       ),
12       mesh
13   );
14
15   Info<< "Reading field U\n" << endl;
16   volVectorField U
17   (
18       IOobject
19       (
20           "U",
21           runTime.timeName(),
22           mesh,
23           IOobject::MUST_READ,
24           IOobject::AUTO_WRITE
25       ),
26       mesh
27   );
28
29   #include "createPhi.H"
30
31   Info<< "Reading transportProperties\n" << endl;
```

```
32   IOdictionary transportProperties
33   (
34       IOobject
35       (
36           "transportProperties",
37           runTime.constant(),
38           mesh,
39           IOobject::MUST_READ,
40           IOobject::NO_WRITE
41       )
42   );
43
44   immiscibleIncompressibleTwoPhaseMixture mixture(U, phi);
45
46   volScalarField& alpha1(mixture.alpha1());
47   volScalarField& alpha2(mixture.alpha2());
48
49   const dimensionedScalar& rho1 = mixture.rho1();
50   const dimensionedScalar& rho2 = mixture.rho2();
51
52   // Need to store rho
53   volScalarField rho
54   (
55       IOobject
56       (
57           "rho",
58           runTime.timeName(),
59           mesh,
60           IOobject::READ_IF_PRESENT
61       ),
62       alpha1*rho1 + alpha2*rho2
63   );
64   // Step interpolate density field:
65       //- Loop over all cells
66       forAll(rho.internalField(),celli)
67       {
68           // Wet cell
69           if (alpha1[celli] > 0.5)
70           {
71               rho[celli] = rho1.value();
72           }
73           // Dry cell
74           else
75           {
76               rho[celli] = rho2.value();
77           }
78       }
79
80       // Loop over boundary field
81       forAll(mesh.boundary(), patchi)
82       {
83           scalarField& bScalarField = rho.boundaryFieldRef(false)[patchi];
84
85           forAll(mesh.boundary()[patchi],facei)
```

```
 86              {
 87                  // Boundary cellID
 88                  const label& bCell = mesh.boundaryMesh()[patchi].faceCells()[facei];
 89                  // Wet cell
 90                  if (alpha1[bCell] > 0.5)
 91                  {
 92                      bScalarField[facei] = rho1.value();
 93                  }
 94                  // Dry cell
 95                  else
 96                  {
 97                      bScalarField[facei] = rho2.value();
 98                  }
 99              }
100          }
101
102   // Store for old times.
103   rho.oldTime();
104
105   #include "readGravitationalAcceleration.H"
106   #include "readhRef.H"
107   #include "gh.H"
108
109   volScalarField p
110   (
111       IOobject
112       (
113           "p",
114           runTime.timeName(),
115           mesh,
116           IOobject::NO_READ,
117           IOobject::AUTO_WRITE
118       ),
119       p_rgh + rho*gh
120   );
121
122   label pRefCell = 0;
123   scalar pRefValue = 0.0;
124
125   p += dimensionedScalar
126   (
127       "p",
128       p.dimensions(),
129       pRefValue - getRefCellValue(p, pRefCell)
130   );
131   p_rgh = p - rho*gh;
132
133   Info<< "Reading pressure equation diffusivity gamma\n" << endl;
134
135   dimensionedScalar gamma("gamma", dimTime, transportProperties);
```

I have created a setup that is very similar to that found for interIsoFoam, because the discretization scheme presented here is intented for incompressible multiphase flows.

Settings in controlDict

There is no time dependency the equation solved, so we can just specify a single time step. The dictionary from the `standingWave` case is modified to reflect the settings given by

```
application    laplacianVOFFoam;
endTime        0.001;
adjustTimeStep  no;
```

Settings in fvSchemes

The modified Ghost Fluid Method Gauss discretization scheme is defined by

```
default   GFMGauss linear corrected;
```

4.3   Run the cases

The case executed with the `Allrun` script, and cleaned with the `Allclean` script.

4.3.1   Allclean

The `Allclean` script contains

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


cleanCase0
```

The script uses the build in clean function to clean the case.

4.3.2   Allrun

The `Allrun` script contains

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


restore0Dir

runApplication blockMesh
runApplication transformPoints -rotate "((1 0 0) (1 0 0.5))"

runApplication setAlphaField

runApplication $(getApplication)

touch output.foam
```

The script restores the `0/` directory from `0.orig/`, constructs the mesh with blockMesh, rotates the geometry with the angle between the two specified vectors, sets the void fraction field based on `setAlphaFieldDict` and finally the solver specified in `controlDict` is executed. At the end an empty (.foam) file is created for post-processing in Paraview.

4.4 Post-processing

Figure 4.2 shows a visual comparison of the dynamic pressure distribution calculated with the two different schemes.
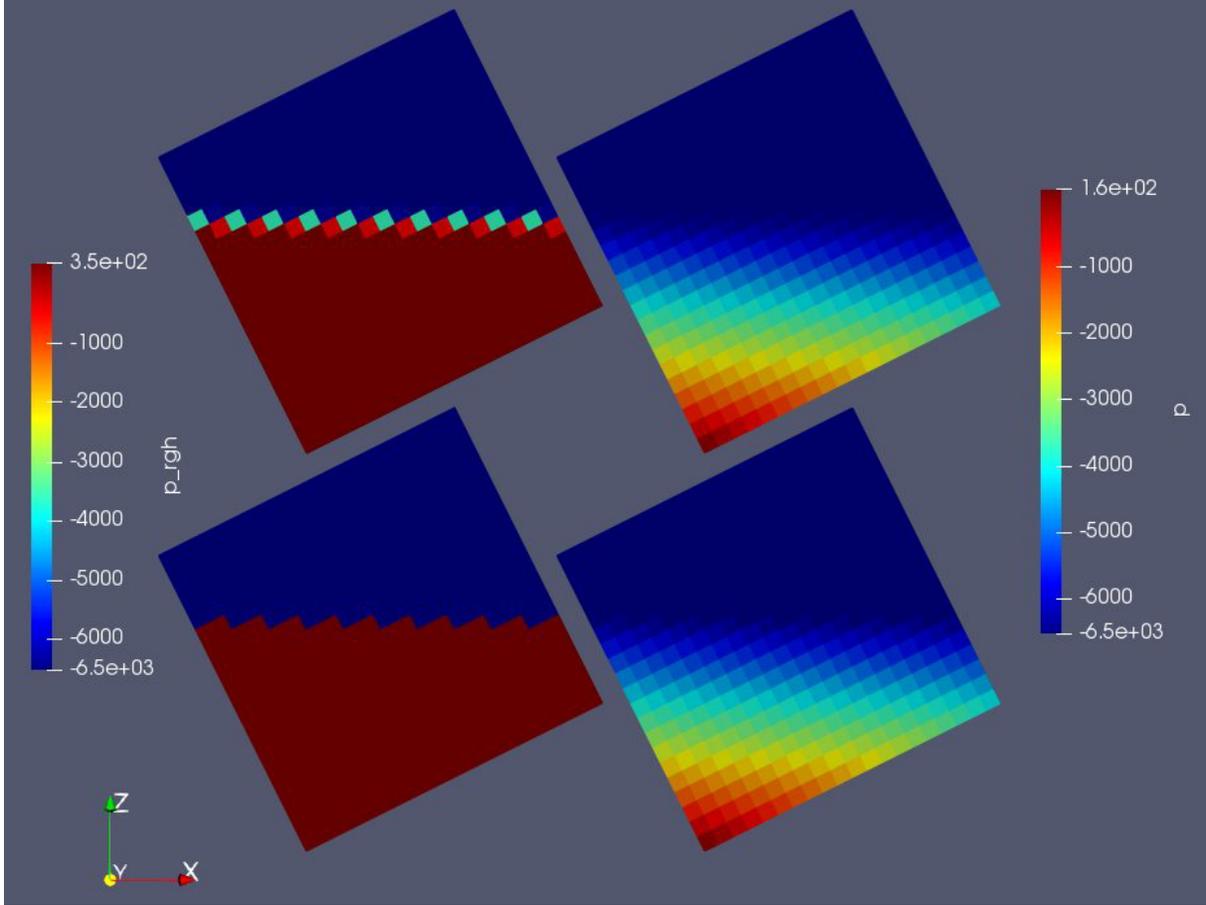
Figure 4.2: Left: Dynamic pressure distribution and Right: Total pressure distribution of same case with two different laplacian discretization schemes. Gauss discretization with the Ghost Fluid Method was used for the top row, and standard Gauss discretization was used for the bottom row. The fvSchemes settings were, Top: `GFMGauss linear corrected`, Bottom: `Gauss linear corrected`.

The range of the dynamic pressure distribution is the same, however it seen that the Gauss discretization with the Ghost Fluid Method retains the jump at the interface between air and water. The jump at the surface is verified by evaluating the jump condition

$$p^- - p^+ = -(\rho^- - \rho^+)\mathbf{g} \bullet \mathbf{x} = -(1000-1)\ \frac{kg}{m^3} \cdot 9.81\ \frac{m}{s^2} \cdot 0.7\ m = -6860.13\frac{kg}{m\ s^2}$$

This matches well with scale on the figure, which gives

$$p^- - p^+ = -6500 - 350 = -6850$$

Study questions

1. How do you specify a standard Gauss Laplacian discretization in `system/fvSchemes` with linear interpolation of diffusion and non orthogonal correction defined by the class `corrected`?

2. How do you account for skewness in the explicit gradient evaluation with settings in `system/fvSchemes`?

3. Why do we need to separate the case with a diffusion tensor from the case with a diffusion scalar?

4. What is the key difference between normal Gauss discretization and Gauss discretization with GFM?

5. What is the purpose of the abstract base class `laplacianScheme`?

6. What macro is used to define the class type name used in `system/fvSchemes`?

7. What do you loop over when you assemble the system matrix?

8. What is the role of `gaussLaplacianSchemes.C`?

9. How do you step interpolate the density field in an application?

10. For which solvers can the GFM method be used for?

Bibliography

[1] V. Vukčević, H. Jasak, and I. Gatin, "Implementation of the Ghost Fluid Method for free surface flows in polyhedral Finite Volume framework," *Computers and Fluids*, vol. 153, pp. 1–19, 2017.

[2] H. Jasak, *Error Analysis and Estimation for the Finite Volume Method with Applications to Fluid Flows*. PhD thesis, 1996.

[3] F. Moukalled, L. Mangani, and M. Darwish, *The Finite Volume Method in Computational Fluid Dynamics - An Advanced Introduction with OpenFOAM and Matlab*. 2016.

[4] T. Grahs, "https://www.foamacademy.com/wp-content/uploads/2018/03/OF2018matrix.pdf," 2018.