Dissertation Notes

Erol Bicer

2022-12-18

Contents

Preface

My landing page.

IATE

Literature Review

Deneme

OpenFOAM

Source Term Treatment

The partial differential equations are converted into sets of algebraic equations through equation discretization that is usually in the form of equation (1) where $[A]$ is the sparse coefficient matrix, $[x]$ is the unknown column vector, and $[b]$ is the source vector. The coefficients of the algebraic equations are stored in the $[A]$ matrix through the `fvMatrix` template class after the discretization.

$$[A][x] = [b] \tag{1}$$

Source terms can be specified as Explicit, Implicit, and Implicit/Explicit (Hybrid hereinafter) through `Su()`, `Sp()`, and `SuSp()` functions, respectively, in OpenFOAM. These functions handle the incorporation of the source terms into the coefficient matrices and source term vectors. The Explicit treatment through `Su()` alters the coefficients in the source vector (i.e. $[b]$) while the Implicit treatment through `Sp()` modifies only the diagonal coefficients in the coefficient matrix (i.e. $[A]$). The Hybrid treatment automatically selects between the Implicit and Explicit treatment depending on the sign of the coefficients. The Implicit treatment is selected if the coefficients are greater than zero or the Explicit treatment is selected if the coefficients are less than zero. Though this sounds arbitrary, there is a good reason behind it which perhaps can best be explained with an example.

Let's look at the steady one-dimensional heat conduction example given in Patankar (2018) as given in equation (2) where k is the thermal conductivity, T is the temperature, and S is the rate of heat generation per unit volume.

$$\frac{d}{dx}\left(k\frac{dT}{dx}\right) + S = 0 \tag{2}$$

The discretization of equation (2) can be done after utilizing the widely used grid-point notation where P is the central node, and E and W are the neighbors. More on the notation can be found at Patankar (2018).

$$\begin{aligned} \left(k\frac{dT}{dx}\right)_e - \left(k\frac{dT}{dx}\right)_w + \int_w^e S dx = 0 \\ \frac{k_e\left(T_E - T_P\right)}{(\delta x)_e} - \frac{k_w\left(T_P - T_W\right)}{(\delta x)_w} + \bar{S}\Delta x = 0 \end{aligned} \tag{3}$$

where;

$$\bar{S} = S_C + S_P T_P \tag{4}$$

$\bar{S}$ is the average value of source therm over the control volume which is consisted of a constant S_C and a coefficient S_P of the dependent variable T_P. It can be seen that the $\bar{S}$ is constructed as a linear source which will be discussed more in detail later. The discretized equation (3) can now be written in a more general format as shown in equation (5).

$$a_P T_P = a_E T_E + a_W T_W + b \tag{5}$$

where;

$$\begin{aligned} a_E &= \frac{k_e}{(\delta x)_e} \\ a_W &= \frac{k_w}{(\delta x)_w} \\ a_P &= a_E + a_W - S_P\Delta x \\ b &= S_C\Delta x \end{aligned} \tag{6}$$

It is known that in practical applications, the source could be dependent or independent of the dependent variable (i.e. T) in the algebraic equation. This is the reason why the source is defined as it is in equation (4). Because the discretized equations are linear, the source term can also only have a linear dependence. However, there could be sources that are nonlinear in certain applications. Therefore, a linearization must be done to be able to incorporate the source terms into the discretized equations. This is done by iterative procedures by splitting the S_C and S_P parts in an intuitive way. However, the details will not be discussed here. What we are trying to understand is how this source term is incorporated into the diagonal coefficient matrix and source vector. The reasoning lay in equation (6).

In Patankar (2018), four basic rules were laid out to best represent the physics by the proposed procedure. Rule number 2 requires all positive coefficients meaning that the neighbors and the center-point coefficients must have the same sign. Rule number 3 is about the source term linearization. According to the rule, only a negative slope linearization is accepted. And the acceptance

criterion is that the S_P in equation (4) must be **always negative or equal to zero** when the source term is linearized. The reasoning can be seen in equation (6) that $-S_P\Delta x$ is added to the a_P which constructs the diagonal terms of the matrix equation. Considering the diagonal dominance, which basically demands a positive (rule number 2) and large a_P, it is possible to get negative values if S_P is positive. Therefore, in order to ensure all positive (and large) diagonal coefficients, S_P must be negative or equal to zero. As for the constant part (S_C) of the source, it can be seen in b in equation (6) that it is added to the source directly as it is.

Considering the above explanation, now the source term treatment in OpenFOAM can be explained further. Due to the matrix operations in OpenFOAM, equation (1) is in fact constructed as shown in equation (7) by sending the source term to the left-hand side (LHS). This will be useful later once the individual source treatment functions are introduced.

$$[A][x] + [-b] = 0 \tag{7}$$

It is now possible to investigate the coded source term treatment functions; `Su()`, `Sp()`, and `SuSp()` more in detail. The source codes and the header files of the functions can be found in the `~\src\finiteVolume\finiteVolume\fvm` directory with the names `fvmSup.C` and `fvmSup.H`. It should be noted that these functions can both be utilized for `finiteVolumeMethod` (fvm::) and `finiteVolumeCalculus` (fvc::) static functions however the explanation here is based on the `fvm::` one. These functions use two arguments, for instance, Su(Φ,x). The first argument Φ is the field variable of the source term and x is the dependent variable of the solved equation.

- **Explicit Source Tem Treatment, `Su()`**

The first one to explore is the Explicit treatment through the `Su()` function in OpenFOAM. Snippet #1 shows how the `Su()` is implemented. Looking at line 21 of Snippet #1, one can see that `su.field()` (the first argument in the function, i.e. Φ) is subtracted from the `fvm.source()` matrix which is the $[-b]$ in equation (7). This confirms that `Su()` is implemented in agreement with the theory.

```
1  template<class Type>
2  Foam::tmp<Foam::fvMatrix<Type>>
3  Foam::fvm::Su
4  (
5      const DimensionedField<Type, volMesh>& su,
6      const GeometricField<Type, fvPatchField, volMesh>& vf
7  )
8  {
```

```
9       const fvMesh& mesh = vf.mesh();
10
11      tmp<fvMatrix<Type>> tfvm
12      (
13          new fvMatrix<Type>
14          (
15              vf,
16              dimVol*su.dimensions()
17          )
18      );
19      fvMatrix<Type>& fvm = tfvm.ref();
20
21      fvm.source() -= mesh.V()*su.field();
22
23      return tfvm;
24     }
```

Snippet #1: The explicit source term function, `Su()`, in OpenFOAM

- **Implicit Source Tem Treatment, `Sp()`**

The next one to explore is the Implicit treatment through the `Sp()` function in OpenFOAM. Snippet #2 shows how the `Sp()` is implemented. Looking at line 21 of Snippet #2, one can see that `sp.field()` (the first argument in the function, i.e. Φ) is added to the `fvm.diag()` matrix which is the diagonal part of $[A]$ in equation (7). This confirms that `Sp()` is implemented in agreement with the theory.

Another important difference between `Su()` and `Sp()`is the returned unit which the difference can be seen in line 16 of both Snippet #1 and #2. The reasoning can be seen in (4) and (7) where Sc is constant and Sp is multiplied by the dependent variable. So, this is reflected by multiplying the unit of the solved variable (the second argument in the functions, i.e. x) through `vf.dimensions()` in line 16 of the Snippets.

```
1   template<class Type>
2   Foam::tmp<Foam::fvMatrix<Type>>
3   Foam::fvm::Sp
4   (
5       const volScalarField::Internal& sp,
6       const GeometricField<Type, fvPatchField, volMesh>& vf
7   )
8   {
```

```
9       const fvMesh& mesh = vf.mesh();
10
11      tmp<fvMatrix<Type>> tfvm
12      (
13          new fvMatrix<Type>
14          (
15              vf,
16              dimVol*sp.dimensions()*vf.dimensions()
17          )
18      );
19      fvMatrix<Type>& fvm = tfvm.ref();
20
21      fvm.diag() += mesh.V()*sp.field();
22
23      return tfvm;
24  }
```

Snippet #2: The explicit source term function, `Sp()`, in OpenFOAM

- **Hybrid Source Tem Treatment, `SuSp()`**

The last one to explore is the Hybrid treatment through the `SuSp()` function in OpenFOAM. Snippet #3 shows how the `SuSp()` is implemented. Looking at lines 21 through 24 of Snippet #3, one can see that `susp.field()` (the first argument in the function, i.e. Φ) is either subtracted from `fvm.source()` matrix which is the $[-b]$ or added to the `fvm.diag()` matrix which is the diagonal part of $[A]$ in equation (7). This confirms that `SuSp()` is implemented in agreement with the theory.

It would be useful to go through the selective logic in the code to understand how the code decides whether it is implicit or explicit. Let's think of a positive `susp.field()` (the first argument in the function, i.e. Φ), first in line 21, the code compares it with scalar zero and selects the maximum. Since it's positive, the `susp.field()` is selected and multiplied by the volume and added to the diagonal coefficient matrix. It continues to line 23, and compares a positive `susp.field()` with a scalar zero to select the minimum. Of course, zero is selected and then multiplied by volume. Therefore no changes are done to the source matrix. The reverse can also be thought for a negative `susp.field()`. This time zero is selected for the diagonal matrix hence no changes are made to it however a negative `susp.field()` is selected for the source matrix, multiplied with the dependent variable and subtracted from itself. So, in short, it can be said that if the first argument of the `SuSp()` function is positive the function directly changes the diagonals of the coefficient matrix, and if it is negative

then it is multiplied with the independent variable, and changes the source coefficients.

```
template<class Type>
Foam::tmp<Foam::fvMatrix<Type>>
Foam::fvm::SuSp
(
    const volScalarField::Internal& susp,
    const GeometricField<Type, fvPatchField, volMesh>& vf
)
{
    const fvMesh& mesh = vf.mesh();

    tmp<fvMatrix<Type>> tfvm
    (
        new fvMatrix<Type>
        (
            vf,
            dimVol*susp.dimensions()*vf.dimensions()
        )
    );
    fvMatrix<Type>& fvm = tfvm.ref();

    fvm.diag() += mesh.V()*max(susp.field(), scalar(0));

    fvm.source() -= mesh.V()*min(susp.field(), scalar(0))
        *vf.primitiveField();

    return tfvm;
}
```

Snippet #3: The hybrid source term function, `SuSp()`, in OpenFOAM

Bibliography

Patankar, S. V. (2018). *Numerical Heat Transfer and Fluid Flow*. CRC Press.