Details of the divergence term fvm::div(phi, T)

Details of the divergence term fvm::div(phi, T)

We will go through how `div(phi, T)` works using Doxygen.

This is an example how you can understand the code in OpenFOAM which might be useful if you want to investigate a piece of OpenFOAM code.

Details of the divergence term fvm::div(phi, T)

- In this section, we will go through the implementation of `div(phi, T)` function. Since depending on our selection during runtime in fvSchemes file, different implementations are used, we assume that we have selected the following in the fvSchemes file.

```
div(phi,T)      Gauss upwind;
```

- This selection means that for discretization of a term like $\nabla \cdot (\mathbf{U}T)$, we first transform the volume integral to a surface integral using Gauss's theorem and then we use upwind interpolation to obtain the interpolated values to the faces, T_f.

$$\underbrace{\int_V \nabla \cdot (\mathbf{U}T)\, dV = \oint_S dS \cdot \mathbf{U}T}_{\text{Gauss's theorem}} = \sum_f \mathbf{S}_f \cdot \mathbf{U}_f T_f = \sum_f \mathbf{F} T_f$$

- Note that in the above equation $\mathbf{F}$ which is the flux of $\mathbf{U}$ at the faces, is equivalent to `phi` in OpenFOAM notation.

Details of the divergence term fvm::div(phi, T)

- To start with, we go to the start page of the Doxygen document of OpenFOAM `https://www.openfoam.com/documentation/guides/latest/doc/`
- `fvm::div(phi, T)` means that the `div(phi, T)` is implemented in the namespace `fvm`. To find the implementation of `div(phi, T)`, we can search for the namespace `fvm` in search bar on the top right and select the first option.
- As we can see in the description, this namespace includes functions used for implicit discretization.
- We find four `div` functions in this namespace. The one which its parameters match with the parameters in `div(phi,T)` is,

```
div (const surfaceScalarField &flux, const GeometricField< Type, fvPatchField, volMesh > &vf)
```

- If we check the implementation of this function, we can see that this function calls another `div` function in its return line.

```
return fvm::div(flux, vf, "div("+flux.name()+','+vf.name()+')');
```

- Compared to the previous `div` function, the function has additional argument which is a string based on the name of the flux and the field. For `div(phi, T)`, this string would be `"div(phi, T)"`.

Details of the divergence term fvm::div(phi, T)

- If we check this `div` function, we can see that this function first calls a function called `New` which belongs to the class in the `convectionScheme` class and then applies the operator `()` and lastly calls `fvmDiv` function which belongs to the object returned by the new function followed by the operator `()`.

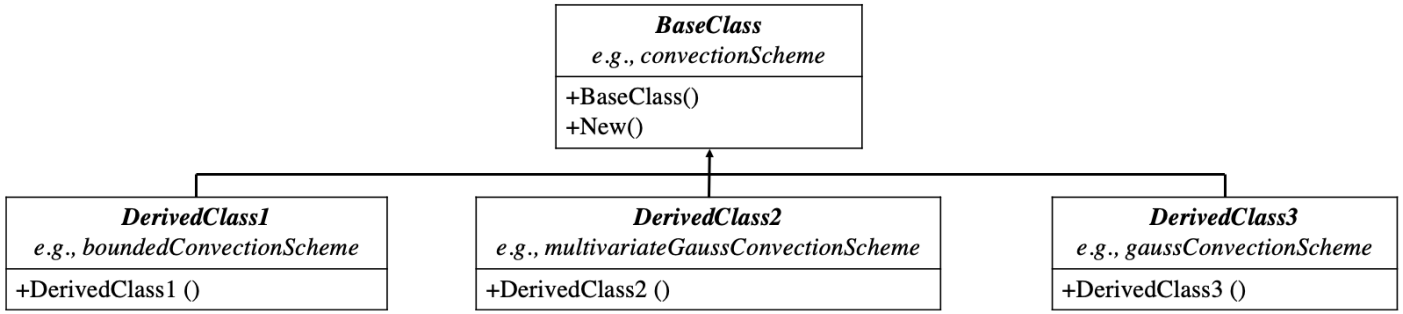
```
return fv::convectionScheme<Type>::New
    (
        vf.mesh(),
        flux,
        vf.mesh().divScheme(name)
    )().fvmDiv(flux, vf);
```

- If we check the implementation of this class, we can see that the function is defined as static which means that we can call it without creating an object of `convectionScheme` class. We can also see that the function returns an object of the class `tmp` with the template parameter set to `convectionScheme` class.

```
static tmp<convectionScheme<Type>> New
(   const fvMesh & mesh,
    const surfaceScalarField & faceFlux,
    Istream & schemeData
)
```

Details of the divergence term fvm::div(phi, T)

- The `New` function is a selector function in the runtime selection mechanism in OpenFOAM. It is implemented in the base class and the purpose is to create an object of derived classes based on the input we provide during the runtime.



- `tmp` is a wrapper class for managing temporary objects. It allows the function to return an object without the need to copy the object and deleting the original one. This would lead to less peak memory usage when a large object is returned by a function.
- The `tmp` class is extensively used in OpenFOAM especially when an object is returned by a function. The operator `()` and the function `ref()` in this class are also used extensively.
- The operator `()` returns a constant reference to the temporary object in the `tmp` object.
- The `ref()` returns a reference to the temporary object in the `tmp` object. Using this function we can change the temporary object itself.

Details of the divergence term fvm::div(phi, T)

- To sum up, the `fv::convectionScheme<Type>::New` followed by () operator returns an object of one of `convectionScheme`'s sub-classes based on input at the runtime.
- Now let's check the implementation of `fvmDiv(flux, vf)` function. If we search for this function in Doxygen, we can find four implementations. One is the implementation in `convectionScheme` class which is the base class and the others are the implementations in the sub-classes of `convectionScheme` class.
- As shown before, depending on the output of new function, `fvmDiv(flux, vf)` implemented in one of the derived classes of `convectionScheme` class is called. Here, we assume that the new function returns an object of `gaussConvectionScheme` class, so we only check the implementation of `fvmDiv(flux, vf)` in this class.
- The `fvmDiv` function first creates a constant reference to the output of the `weights(vf)` function which belongs to the object `tinterpScheme_()`.

```
tmp<surfaceScalarField> tweights = tinterpScheme_().weights(vf);
const surfaceScalarField& weights = tweights();
```

- Note that `tinterpScheme_` is an object of the `tmp` class with the template variable set to `surfaceInterpolationScheme` class. As mentioned before the operator `()` return a constant reference to the object which `tmp` object is wrapped around. Therefore, `tinterpScheme_()` returns a constant reference to an object of `surfaceInterpolationScheme` class.

Details of the divergence term fvm::div(phi, T)

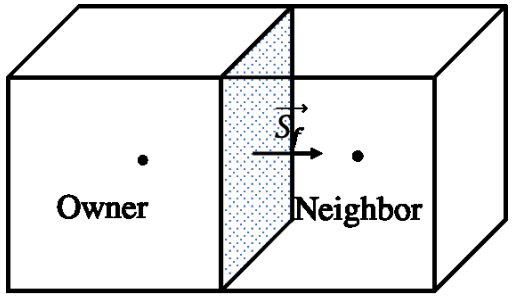
- If we check the constructor of this class, we see that `tinterpScheme_` is initialized by the new function in `surfaceInterpolationScheme` class. This new function returns the a temporary object of the derived classes of the `surfaceInterpolationScheme` depending on the input in the divScheme dictionary.
- To see what `weights()` function returns, we should check the implementation of this function in `surfaceInterpolationScheme` and its derived classes.
- In `surfaceInterpolationScheme` class, this function is a pure virtual function meaning that its implementation is in the derived classes.
- All of the interpolation schemes which are derived from `surfaceInterpolationScheme` class have their own implementation of `weights()` function. Here, we only check the implementation for the upwind scheme.
- We can check the inheritance diagram of `surfaceInterpolationScheme` class to find `upwind` class. We should also note that that `upwind` class in not a direct derived class of the `surfaceInterpolationScheme` class but it is a derived class of `limitedSurfaceInterpolation` class which is a derived class of `surfaceInterpolationScheme` class.

Details of the divergence term fvm::div(phi, T)

- The `weights()` function in `upwind` class returns one if the `this->faceFlux_` is zero or positive, otherwise.

```
//- Return the interpolation weighting factors
        tmp<surfaceScalarField> weights() const
        {
            return pos0(this->faceFlux_);
        }
```

- This is the weighting factor of the owner cells in the interpolated value on the faces. To clearly show this, consider the upwind interpolation on the highlighted face. If we assume that the flux is positive (from owner to neighbor in OpenFOAM notation), the `weights()` function return one which is the weighting factor of the owner cell. The weighting factor of the neighbor cell is simply `1-weights()`.



Details of the divergence term fvm::div(phi, T)

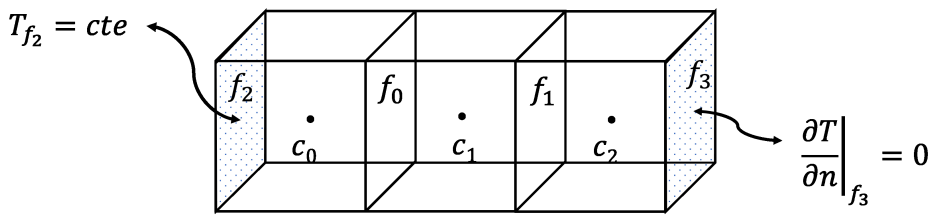
- Now let's go back to the implementation of `fvmDiv(flux, vf)` function in `gaussConvectionScheme` class. To sum up what we understood so far, the `weights` stores the weighing factor of owner cells in the interpolated value on the faces.
- The function then creates a temporary object of the `fvMatrix` class which is constructed using the new keyword. Then the `ref()` function in `tmp` class is called and its return is stored in the object `fvm`. Note the `ref()` function in `tmp` class returns a reference to the empty object of `fvMatrix` which `tfvm` is wrapped around.

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

- In the rest of the code, the function stores the information in **Ax=b** in the fvMatrix, `fvm`. Before going into the detail of the code, we will have a look at how a linear equation system, **Ax=b** is created when div(phi, T) is discretized on a very simple domain. Then we will connect the coefficients in this linear equation system with the coefficients which are stored in the fvMatrix, `fvm`.

Details of the divergence term fvm::div(phi, T)

- The domain is 1D in OpenFOAM and has 3 cells with the labels {0,1,2}. There are two internal faces with the labels {0,1} and two non-empty boundary faces with the label {2,3}. For the boundary face 2, we assume that the boundary condition is `fixedValue` while for the boundary face 3, we assume that the boundary condition is `zeroGradient`.



- For each cell of the domain, we apply the finite volume discretization of div(phi, T) as follows,

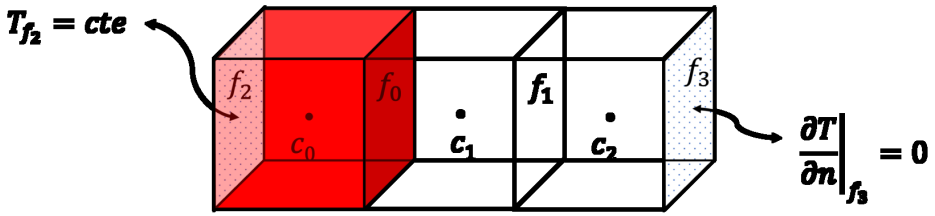
$$\underbrace{\int_V \nabla \cdot (\mathbf{U}T)\, dV = \oint_S dS \cdot \mathbf{U}T}_{\text{Gauss's theorem}} = \sum_f \mathbf{S}_f \cdot \mathbf{U}_f T_f = \sum_f \mathbf{F} T_f$$

Details of the divergence term fvm::div(phi, T)

- For the cell 0, one face is a boundary face with fixed value and the other one is the internal face 0. The discretization would be

$$\sum_f \mathbf{F} T_f = \underbrace{F_0(w_0 T_0 + (1-w_0)T_1)}_{f_0} + \underbrace{F_2 T_{f_2}}_{f_2} = (F_0 w_0)T_0 + (F_0(1-w_0))T_1 + F_2 T_{f2}$$

where F_2 is the flux at the boundary face 2, T_{f2} is a given value at the boundary face 2, F_0 is the flux at the face 0 and w_0 is the weighting factor of the owner cell in the interpolation on the face 0. Note here that for the face 0, the owner cell is cell 0. In OpenFOAM, for each face, the cell with smaller label is the owner and the cell with larger label is the neighbor.

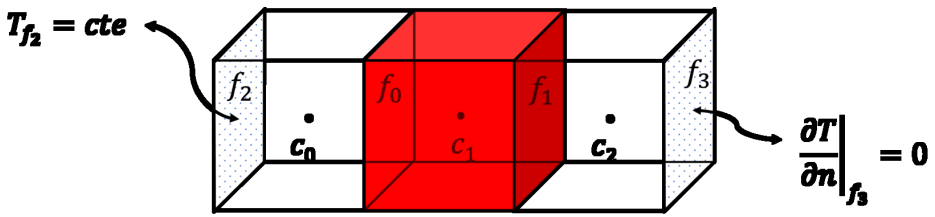


Details of the divergence term fvm::div(phi, T)

- For the cell 1, we have two internal faces. The discretization would be,

$$\sum_f \mathbf{F} T_f = \underbrace{-F_0(w_0 T_0 + (1-w_0)T_1)}_{f_0} + \underbrace{F_1(w_1 T_1 + (1-w_1)T_2)}_{f_1}$$
$$= (-F_0 w_0)T_0 + (-F_0(1-w_0) + F_1 w_1)T_1 + (F_1(1-w_1))T_2$$

where F_1 is the flux at the face 1 and w_1 is the weighting factor of the owner cell in the interpolation on the face 1.

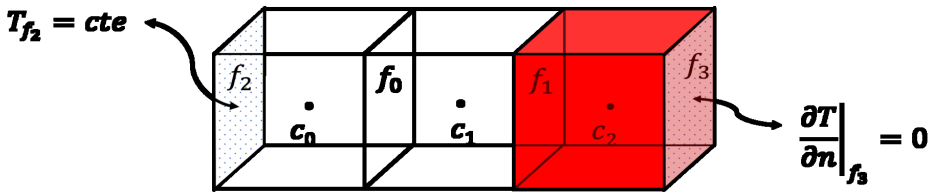


Details of the divergence term fvm::div(phi, T)

- For the cell 2, we have one internal face and one boundary face. The discretization would be,

$$\sum_f \mathbf{F} T_f = \underbrace{-F_1(w_1 T_1 + (1-w_1)T_2)}_{f_1} + \underbrace{F_3 T_2}_{f_3} = (-F_1 w_1)T_1 + (-F_1(1-w_1) + F_3)T_2$$

where F_1 is the flux at the face 1 and w_1 is the weighting factor of the owner cell in the interpolation on the face 1.



Details of the divergence term fvm::div(phi, T)

- Using the discretized equations in the previous slides, we can create a linear equation system, **Ax=b**, as below.

$$\begin{bmatrix} F_0 w_0 & F_0(1-w_0) & 0 \\ -F_0 w_0 & -F_0(1-w_0)+F_1 w_1 & F_1(1-w_1) \\ 0 & -F_1 w_1 & -F_1(1-w_1)+F_3 \end{bmatrix} \begin{bmatrix} T_0 \\ T_1 \\ T_2 \end{bmatrix} = \begin{bmatrix} -F_2 T_{f2} \\ 0 \\ 0 \end{bmatrix}$$

- Now, we can have a look at the code and make a connection between the coefficients the above system and the information stored in the fvMatrix, `fvm`.
- After creating an empty matrix, the code assigns values to two functions in `fvm` fvMatrix, `lower()` and `upper()`.

```
fvm.lower() = -weights.primitiveField()*faceFlux.primitiveField();
fvm.upper() = fvm.lower() + faceFlux.primitiveField();
```

- The implementation of these functions are not in the `fvMatrix` class but in its base class, `lduMatrix`.

Details of the divergence term fvm::div(phi, T)

- Here, the `lower()` and `upper()` functions returns a reference to zero `scalarFields` which their pointers are stored in `fvm` object. By assigning values to these functions, these zero `scalarFields` get values.
- On the other side of these assignments, the `primitiveField()` function belonging to `weights` and `faceFlux` are called. Note that both `weights` and `faceFlux` are a `surfaceScalarField` which is a `typedef` of `GeometricField` class. If we check the implementation of this class, we can see that `primitiveField()` returns a constant reference to the values for internal faces.
- We know from previous slides that the `weights` returns the weighting factor of the owner cells. Therefore, for our simple domain,
  `fvm.lower()` = $\{-w_0F_0, -w_1F_1\}$: Non-zero coefficients in the lower triangle of **A**
  `fvm.upper()` = $\{-w_0F_0 + F_0, -w_1F_1 + F_1\}$: Non-zero coefficients in the upper triangle of **A**

$$\begin{bmatrix} F_0w_0 & F_0(1-w_0) & 0 \\ -F_0w_0 & -F_0(1-w_0)+F_1w_1 & F_1(1-w_1) \\ 0 & -F_1w_1 & -F_1(1-w_1)+F_3 \end{bmatrix} \begin{bmatrix} T_0 \\ T_1 \\ T_2 \end{bmatrix} = \begin{bmatrix} -F_2T_{f2} \\ 0 \\ 0 \end{bmatrix}$$

Details of the divergence term fvm::div(phi, T)

- In the next line of the code, the function `fvm.negSumDiag()` is called. Similar to the `lower()` and `upper()` function, the implementation of this function is in the `lduMatrix` class. We can check the implementation of this function in this class.
- The `negSumDiag()` first stores the constant reference to return of the functions `lower()` and `upper()`.

```
const scalarField& Lower = const_cast<const lduMatrix&>(*this).lower();
const scalarField& Upper = const_cast<const lduMatrix&>(*this).upper();
```

- The function then calls the function `diag()` and store the reference to its return in `Diag`.

```
scalarField& Diag = diag();
```

- `diag()` function is implemented in `lduMatrix` class. It returns a reference to zero `scalarField` in this case.
- The function then calls `lowerAddr()` and `upperAddr()` functions on the return of `lduAddr()` functions. The returns of these functions are stored in the constant reference variables to `u` and `l`.

```
const labelUList& l = lduAddr().lowerAddr();
const labelUList& u = lduAddr().upperAddr();
```

Details of the divergence term fvm::div(phi, T)

- `lowerAddr()` and `upperAddr()` returns addressing for the coefficients stored in `lower()` and `upper()`. The reason for the need to have this addressing is that OpenFOAM only stores the non-zero values of the off-diagonal coefficients. However,in order to construct the **Ax=B** which is needed for solving the equation, the exact position of the coefficients are needed. `lowerAddr()` and `upperAddr()` have information about these positions.
- `lowerAddr()` returns the label list of the owner cells while `upperAddr()` returns the label list of the neighbor cells. In our simple domain, the `lowerAddr()` and the `upperAddr()` return,

```
l: lowerAddr(): {0,1}
u : upperAddr(): {1,2}
```

- As mentioned before, the `lower()` and the `upper()` contain the non-zero off-diagonal coefficients. You can use the `lowerAddr()` and `upperAddr()` to get the position of these coefficients.

`Lower` = $\{-F_0 w_0, -F_1 w_1\} : \{a_{1,0}, a_{2,1}\} : \{a_{u,l}\}$
`Upper` = $\{F_0(1-w_0), F_1(1-w_1)\} : \{a_{0,1}, a_{1,2}\} : \{a_{l,u}\}$

Details of the divergence term fvm::div(phi, T)

- The last part of the code, loop over the faces and assigns values to `Diag`. To understand what is exactly assigned to `Diag`, we simply perform the loop for faces in our simple domain.

$$\texttt{face = 0} \rightarrow \begin{cases} \texttt{l[0] = 0} \\ \texttt{u[0] = 1} \end{cases} \rightarrow \begin{cases} \texttt{Diag[0] = Diag[0]- Lower[0]} = 0 + F_0 w_0 \\ \texttt{Diag[1] = Diag[1] - Upper[0]} = 0 - F_0(1 - w_0) \end{cases}$$

$$\texttt{face = 1} \rightarrow \begin{cases} \texttt{l[1] = 1} \\ \texttt{u[1] = 2} \end{cases} \rightarrow \begin{cases} \texttt{Diag[1] = Diag[1]- Lower[1]} = -F_0(1 - w_0) + F_1 w_1 \\ \texttt{Diag[2] = Diag[2] - Upper[1]} = 0 - F_1(1 - w_1) \end{cases}$$

- If we compare `Diag` with the system of linear equation, what is stored in `Diag` is the diagonal coefficients without the contribution of the boundary condition.

$$\begin{bmatrix} F_0 w_0 & F_0(1 - w_0) & 0 \\ -F_0 w_0 & -F_0(1 - w_0) + F_1 w_1 & F_1(1 - w_1) \\ 0 & -F_1 w_1 & -F_1(1 - w_1) + F_3 \end{bmatrix} \begin{bmatrix} T_0 \\ T_1 \\ T_2 \end{bmatrix} = \begin{bmatrix} -F_2 T_{f2} \\ 0 \\ 0 \end{bmatrix}$$

Details of the divergence term fvm::div(phi, T)

- We saw that the boundary faces can also contribute to the diagonal coefficients and source terms depending on what boundary condition is used. In the following line, the diagonal and the source term contribution of the boundary faces are added to `fvm`.

```
forAll(vf.boundaryField(), patchi)
{
    const fvPatchField<Type>& psf = vf.boundaryField()[patchi];
    const fvsPatchScalarField& patchFlux = faceFlux.boundaryField()[patchi];
    const fvsPatchScalarField& pw = weights.boundaryField()[patchi];

    fvm.internalCoeffs()[patchi] = patchFlux*psf.valueInternalCoeffs(pw);
    fvm.boundaryCoeffs()[patchi] = -patchFlux*psf.valueBoundaryCoeffs(pw);
}
```

- The code first gets access to the values, (`psf`), fluxes (`patchFlux`), weighting factors of the boundary faces, (`pw`).
- It assigns values to the return of `internalCoeffs()` and `boundaryCoeffs()` using the flux values and return of the functions `valueInternalCoeffs(pw)` and `valueBoundaryCoeffs(pw)`.
- The function `internalCoeffs()` returns a reference to a private member data `internalCoeffs_` which is of type `FieldField`, i.e. a Field of Fields. The size of the outer Field is the number of the patches and each inner Fields are initialized as a zero Field with the size of the faces in each patch.

Details of the divergence term fvm::div(phi, T)

- The function `boundaryCoeffs()` returns a reference to a private member data `boundaryCoeffs_`. `boundaryCoeffs_` is of type `FieldField`, i.e. a Field of Fields and is set zero in the constructor.
- To understand what is stored in the return of the functions, `boundaryCoeffs()` and `internalCoeffs()` , we need to know what is returned by the functions `valueInternalCoeffs()` and `valueBoundaryCoeffs()`.
- If we search for these functions in Doxygen, we can see that these functions are implemented in the classes related to boundary conditions. Let's check the implementation these functions for the `fixedValue` boundary condition.
- The class for the `fixedValue` boundary condition is `fixedValueFvsPatchField`. The `valueInternalCoeffs()` function in this class returns a zero volField and the `valueBoundaryCoeffs()` returns `*this` which is the `vf.boundaryField()[patchi]`, i.e the values at the boundary faces.
- For our simple domain, the boundary condition for face 2 is set `fixedValue`. For this face,

  `fvm.internalCoeffs()[patchi]` : 0
  `fvm.boundaryCoeffs()[patchi]` : $-F_2 T_{f_2}$

Details of the divergence term fvm::div(phi, T)

- We can also look for the implementation of these functions in the `zeroGradientFvPatchField` which is the class for the zero gradient boundary condition. The `valueInternalCoeffs()` function in this class returns a one volField and the `valueBoundaryCoeffs()` returns zero volField.
- For our simple domain and the boundary condition for face 2 where the boundary condition is set to `fixedValue`. For this face,

  `fvm.internalCoeffs()[patchi]` : F_3
  `fvm.boundaryCoeffs()[patchi]` : 0

- `fvm.internalCoeffs()` stores the diagonal terms coming from the boundary faces a while `fvm.boundaryCoeffs()` stores the source terms coming from the boundary faces.

$$\begin{bmatrix} F_0 w_0 & F_0(1-w_0) & 0 \\ -F_0 w_0 & -F_0(1-w_0)+F_1 w_1 & F_1(1-w_1) \\ 0 & -F_1 w_1 & -F_1(1-w_1)+F_3 \end{bmatrix} \begin{bmatrix} T_0 \\ T_1 \\ T_2 \end{bmatrix} = \begin{bmatrix} -F_2 T_{f2} \\ 0 \\ 0 \end{bmatrix}$$

Details of the divergence term fvm::div(phi, T)

- The last part of the function checks whether the return of `corrected()` function in `tinterpScheme_` is true or not. If it is true, the function the adds the contribution of the correction in the interpolation scheme to `fvm`.

```
if (tinterpScheme_().corrected())
{
    fvm += fvc::surfaceIntegrate(faceFlux*tinterpScheme_().correction(vf));
}
```

- To find the implementation of the `corrected()` and `correction(vf)` functions, we should look into the `surfaceInterpolationScheme` class as `tinterpScheme_` is an object of this class.
- Based on the description of the `corrected()` function, it returns true if the interpolation scheme has an explicit correction. In `surfaceInterpolationScheme`, this function returns False. Note, however, that the function is defined as virtual which means that the function can be defined in derived classes, i.e, interpolation schemes. There are some derived classes where this function returns True.

Details of the divergence term fvm::div(phi, T)

- For the `correction(vf)` function, the description says that it returns the explicit correction of the face interpolation. We can find the implementation of this function in `surfaceInterpolationScheme`. However, this function is redefined in all of the derived classes where `corrected()` returns True. So, the implementation of `correction(vf)` in `surfaceInterpolationScheme` class is never called.
- The last line of the code finally returns the `fvMatrix`, `fvm`.