

Giskard's CFD Learning Tricks

CFD and Scientific Computing

2015-05-17 • OPENFOAM

OpenFOAM 中的 div 与 snGrad 操作符

OpenFOAM 的方便之处之一是利用 C++ 的类模板和函数重载等技术定义了很多各种离散操作符，如 `div`, `laplacian`, `grad` 等等。利用这些操作符，很容易就能对偏微分方程进行离散，并构建起线性方程组。但是，这些操作符真正执行的运算，却需要结合有限体积方法的本质来理解一番才能真正掌握。下面尝试着对 OpenFOAM 中的 `div` 和 `snGrad` 操作符进行一点解读。

1. div 操作符的本质

`div` 操作符表面看，是计算散度的，实际上，在 OpenFAOM 中，`div` 操作符的作用是**加和**，比如说 $\nabla \cdot (UU)$ ，在 OpenFOAM 中表示为 `fvm::div(phi,U)`，这段代码真正执行的是 $\sum_f U_f \phi_f$ 运算，即将每个网格包含的面上的流率与速度乘积，然后再加起来。再比如，`twoPhaseEulerFoam` 的 `UaEqn` 方程有一项是 `fvm::sp(fvc::div(phia),Ua)`，其对应的公式是 $U_a(\nabla \cdot U_a)$ 。为什么是这样呢？以下试图对背后的原理做一点解释：

单相流的动量守恒方程的微分形式如下：

$$\frac{\partial U}{\partial t} + \nabla \cdot (UU) + \nabla \cdot dev(-\nu_{eff}(\nabla U + (\nabla U)^T)) = -\nabla p + Q$$

为了使用有限体积方法，需要将动量方程写成积分形式，即

$$\int_V \left[\frac{\partial U}{\partial t} + \nabla \cdot (UU) + \nabla \cdot dev(-\nu_{eff}(\nabla U + (\nabla U)^T)) \right] dV = \int_V [-\nabla p + Q] dV$$

这里只分析 $\int_V \nabla \cdot (UU) dV$ 这一项，利用高斯定理，可以将这一个个体积分，转化成对包围该体积微元的表面的面积分： $\oint_{\partial V} (UU) \cdot dS$ ，其中 dS 是面积微元矢量。又因为实际操作中，一个体积微元总是由有限的几个面组成的，所以

$$\oint_{\partial V} (UU) \cdot dS = \sum_f \int_{S_f} (UU) \cdot dS_f = \sum_f (UU)_f \cdot S_f$$

, 其中

$$(UU)_f = \frac{1}{mag(S_f)} \int_{S_f} (UU) \cdot dS_f$$

。

这里做一个近似:

$$(UU)_f \approx (U_f U_f)$$

于是得到

$$\sum_f (UU)_f \cdot S_f \approx \sum_f (U_f U_f) \cdot S_f$$

运用张量计算规则 $[\mathbf{uv} \cdot \mathbf{w}] = \mathbf{u}(\mathbf{v} \cdot \mathbf{w})$, 得到

$$\sum_f (U_f U_f) \cdot S_f = \sum_f U_f (U_f \cdot S_f) = \sum_f U_f \phi_f$$

综上, OpenFOAM中的代码 `fvm::div(UU)` 对应的公式其实是 $\int_V \nabla \cdot (UU) dV$, 而根据推导, 在有限体积方法中 $\int_V \nabla \cdot (UU) dV = \sum_f (U_f \phi_f)$, 所以, `fvm::div(UU)` 实质上进行的运算是, **把网格当作体积微元, 将网格中心的速度 U 插值到包围该网格的所有面上的面心上得到 U_f , 并计算每个面上的速度通量 ϕ_f , 然后返回每一个面上的速度与通量乘积的加和。** U_f 的计算需要用到本网格与邻近网格的速度值, 离散格式的作用就体现在如果利用本网格与邻近网格的速度得到面上的速度。

再来看上面提到的另一项 $U_a(\nabla \cdot U_a)$, 根据上面类似的推导

$$\int_V [U_a(\nabla \cdot U_a)] dV = U_a \int_V (\nabla \cdot U_a) dV$$

注意, 这里之所以能这样变换, 是因为在一个体积微元 dV 内, U_a 是常数。根据高斯定理

$$\int_V (\nabla \cdot U_a) dV = \oint_{\partial V} U_a \cdot dS = \sum_f (U_a)_f \cdot S_f = \sum_f (\phi_a)_f$$

于是得到

$$\int_V [U_a(\nabla \cdot U_a)] dV = U_a \sum_f (\phi_a)_f$$

这一项在OpenFOAM中的表达是 `fvm::sp(fvc::div(phia),Ua)`, 含义就很明显了,这一项相当于是一个系数与需要求解的量 U_a 的乘积, 所以被当作隐式的源项来处理。注意我先前以为把 $(\nabla \cdot U_a)$ 当作显式处理是人为简化的结果, 其实不然, 这是自然而然的结果。而在这里也可以看出, $\sum_f (\phi_a)_f$ 对应的代码是 `fvc::div(phia)`, 也印证了上面的观点, 即 `div` 操作符**本质上是在作加和运算**。

2. grad 与 snGrad

在 twoPhaseEulerFoam 中, $\frac{\nabla\alpha}{\alpha}$ 在两个不同的地方用了两种不同的表示。

$$\nabla \cdot \left\{ [\nu_{eff,a} \frac{\nabla\alpha}{\alpha}] [U_a] \right\}$$

对应的代码是: `fvm::div(phiRa,Ua)` , 其中

```
1 phiRa=-fvc::interpolate(nuEffa)*mesh.magSf()*fvc::snGrad(alpha)
2      /fvc::interpolate(alpha + scalar(0.001));
```

而另一项

$$[\frac{\nabla\alpha}{\alpha}] \cdot Rca$$

对应的代码是 `fvc::grad(alpha)/fvc::average(alpha + scalar(0.001)) & Rca`

下面是我对这个的理解:

对于

$$\nabla \cdot \left\{ [\nu_{eff,a} \frac{\nabla\alpha}{\alpha}] [U_a] \right\}$$

其处理方法跟 $\nabla \cdot (U_a U_a)$ 是一样的, 因为 $\frac{\nabla\alpha}{\alpha}$ 也是一个矢量。phiRa 相当于是 $[\nu_{eff,a} \frac{\nabla\alpha}{\alpha}]$ 这个矢量的界面通量, 类比于 phia。phia 的定义是 `linearInterpolate(Ua) & mesh.Sf()`, 而 phiRa 理论上应该也可以定义成类似于 `linearInterpolate(gradalpha) & mesh.Sf()`, 前提是要先定义一个

`volVectorField gradalpha=-nuEffa*fvc::grad(alpha)/alpha`。但是考虑到界面通量本质上是先将一个 `volVectorField` 插值到面上, 并乘以面积矢量, 对于 $\frac{\nabla\alpha}{\alpha}$, 完全可以直接求出每个面上的 $\frac{\nabla\alpha}{\alpha}$, 然后乘以面积矢量, 而不需要先建立体中心的 $\frac{\nabla\alpha}{\alpha}$ 再插值到面上。snGrad 就是这样一个用来求面上的梯度量的函数, 它求解面上的梯度时, 采用如下公式:

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

即用相邻网格的值减去面所属网格的值, 再除以两个网格中心矢量的模。注意这个处理对于正交网格是精确的, 对于非正交网格, 只是一个近似处理。而且要注意, 这样求得的面上的梯度值已经是一个标量了。再回到

phiRa 的定义, 既然 `fvc::snGrad(alpha)` 已经是标量了, 那只要再乘以面积矢量的模 `mesh.magSf()`, 便是界面上的通量了。`fvc::interpolate(nuEffa)` 和 `fvc::interpolate(alpha + scalar(0.001))` 则分别是将 `volField` 插值到面。

再来看

$$[\frac{\nabla\alpha}{\alpha}] \cdot Rca$$

这一项是被当作显式的源项来处理，所以，我们需要得到的是一个volVceterField。所以这里将 $\nabla\alpha$ 处理成volVectorField，再与作为volTensorField的Rca进行点乘，得到的就是volVceterField。因此，这里的 $\frac{\nabla\alpha}{\alpha}$ 对应的代码是fvc::grad(alpha)/fvc::average(alpha + scalar(0.001))。注意这里的fvc::average()函数的定义如下：

```
1  43 template<class Type>
2  44 tmp<GeometricField<Type, fvPatchField, volMesh> >
3  45 average
4  46 (
5  47     const GeometricField<Type, fvsPatchField, surfaceMesh>& ssf
6  48 )
7  49 {
8  50     const fvMesh& mesh = ssf.mesh();
9  51
10 52     tmp<GeometricField<Type, fvPatchField, volMesh> > taverage
11 53     (
12 54         new GeometricField<Type, fvPatchField, volMesh>
13 55         (
14 56             IOobject
15 57             (
16 58                 "average("+ssf.name()+')',
17 59                 ssf.instance(),
18 60                 mesh,
19 61                 IOobject::NO_READ,
20 62                 IOobject::NO_WRITE
21 63             ),
22 64             mesh,
23 65             ssf.dimensions()
24 66         )
25 67     );
26 68
27 69     GeometricField<Type, fvPatchField, volMesh>& av = taverage();
28 70
29 71     av.internalField() =
30 72     (
31 73         surfaceSum(mesh.magSf()*ssf)/surfaceSum(mesh.magSf())
32 74     )().internalField();
33 75
34 76     typename GeometricField<Type, fvPatchField, volMesh>::
35 77     GeometricBoundaryField& bav = av.boundaryField();
36 78
37 79     forAll(bav, patchi)
```

```

38  80      {
39  81          bav[patchi] = ssf.boundaryField()[patchi];
40  82      }
41  83
42  84      av.correctBoundaryConditions();
43  85
44  86      return taverage;
45  87  }
46  88
47  89
48  90  template<class Type>
49  91  tmp<GeometricField<Type, fvPatchField, volMesh> >
50  92  average
51  93  (
52  94      const tmp<GeometricField<Type, fvsPatchField, surfaceMesh> >& tssf
53  95  )
54  96  {
55  97      tmp<GeometricField<Type, fvPatchField, volMesh> > taverage
56  98      (
57  99          fvc::average(tssf())
58 100      );
59 101      tssf.clear();
60 102      return taverage;
61 103  }
62 104
63 105
64 106  template<class Type>
65 107  tmp<GeometricField<Type, fvPatchField, volMesh> >
66 108  average
67 109  (
68 110      const GeometricField<Type, fvPatchField, volMesh>& vtf
69 111  )
70 112  {
71 113      return fvc::average(linearInterpolate(vtf));
72 114  }
73 115
74 116
75 117  template<class Type>
76 118  tmp<GeometricField<Type, fvPatchField, volMesh> >
77 119  average
78 120  (
79 121      const tmp<GeometricField<Type, fvPatchField, volMesh> >& tvtf
80 122  )
81 123  {

```

```
82 124     tmp<GeometricField<Type, fvPatchField, volMesh> > taverage
83 125     (
84 126         fvc::average(tvtf())
85 127     );
86 128     tvtf.clear();
87 129     return taverage;
88 130 }
```

可以看出，`fvc::average()` 函数的返回类型是 `tmp<GeometricField<Type, fvPatchField, volMesh> >`，对应 `alpha`，返回类型就是 `tmp<GeometricField<scalar, fvPatchField, volMesh> >`，即 `tmp<volScalarField>`。 `average` 函数的核心定义见代码71-74行，可见 `average` 采用的是面积加权平均，即

$$\bar{\phi} = \frac{\sum_f \phi_f * |S_f|}{\sum_f |S_f|}$$

。如果给 `average` 的参数类型是 `surfaceField`，那么直接计算平均值后返回 `volField`，如果给的参数的 `volField`，那就先将 `volField` 插值到面，计算平均值后再返回 `volField`，见代码106-114行。

参考资料

1. <http://www.openfoam.org/docs/cpp/>
2. OpenFOAM: A little User-Manual, Gerhard Holzinger, <https://github.com/ParticulateFlow/OSCCAR-doc>

#Code Explained #OpenFOAM

↪ Share

NEWER

twoPhaseEulerFoam 全解读之一

OLDER

利用functionObjects对指定区域内进行后处理

CATEGORIES

C++ (2)

DEM (1)

OpenFOAM (44)