viscosity.h Documentation

Haochen(Langford) Huang

October 24, 2023

1 Introduction and Background

viscosity.h is constructed exclusively for NS solver 'centered.h' (see corresponding doc for more information.), whose purpose is to solve viscos equation implicitly using poisson eqaution solver built in 'poisson.h'. The governing equation is

$$\rho_{n+\frac{1}{2}}[\frac{\mathbf{u}^* - \mathbf{u}'}{\Delta t}] = \nabla \cdot [2\mu_{n+\frac{1}{2}}\mathbf{D}^*] \tag{1}$$

where $\mathbf{D}^* = [\nabla \mathbf{u}^* + (\nabla \mathbf{u}^*)^T]/2$, $\mathbf{u}'$ is known variable and $\mathbf{u}^*$ is the desired output.

Consider intergral form of Eq.1

$$\int_\Omega \rho_{n+\frac{1}{2}}[\frac{\mathbf{u}^* - \mathbf{u}'}{\Delta t}]dV = \int_\Omega \nabla \cdot [2\mu_{n+\frac{1}{2}}\mathbf{D}^*]dV = \int_{\partial\Omega}[2\mu_{n+\frac{1}{2}}\mathbf{D}^*] \cdot \mathbf{n}dS \tag{2}$$

The discrete form of this equation reads (component form is presented instead of tensor for sake of convenience).

$$\Delta\rho_{n+\frac{1}{2}}[\frac{u_i^* - u_i'}{\Delta t}] = \sum_{f=1}^{6}(n_f\mu_{n+\frac{1}{2}}(\frac{\partial u_j^*}{\partial x_i} + \frac{\partial u_i^*}{\partial x_j}))_f \quad i = x, y, z \quad j = normal(f) \tag{3}$$

where f in summation and subscript represents surfaces of single cell, j represents coordinate component of face normal disregarding negative or positive direction, n_f indicates whether the face normal is consistent with positive direction of coordinate, taking 2D cell as an example

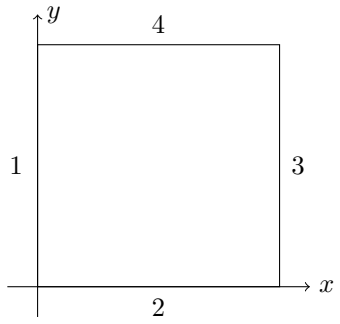

Figure 1: 2D cell example.

Unfold *R.H.S.* of Eq.3 based on 2D cell depicted by Fig.1 yields

$$R.H.S = [\mu_3(\frac{\partial u_x^*}{\partial x_i} + \frac{\partial u_i^*}{\partial x})_3 - \mu_1(\frac{\partial u_x^*}{\partial x_i} + \frac{\partial u_i^*}{\partial x})_1 + \mu_4(\frac{\partial u_y^*}{\partial x_i} + \frac{\partial u_i^*}{\partial y})_4 - \mu_2(\frac{\partial u_y^*}{\partial x_i} + \frac{\partial u_i^*}{\partial y})_2] \quad i = x, y \tag{4}$$

Constructing derivative is simple for aligned direction but cumbersome for spliting direction.

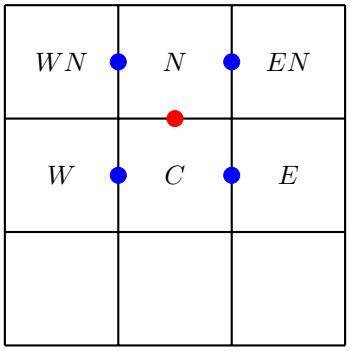

Figure 2: Sketch for derivative caculating.

Still take $2D$ cell as an example, as shown in Fig.2, aligned direction derivative $e.g.(\frac{\partial u_y^*}{\partial y})_n$ (lower case indicates corresponding cell face) can be obtained by simply cnetral difference scheme

$$(\frac{\partial u_y^*}{\partial x_y})_n = \frac{u_N^* - u_C^*}{\Delta} \tag{5}$$

However, for spliting direction derivative $e.g.(\frac{\partial u_y^*}{\partial x})_n$ highlighted by red point above, there is no direct method to caculate the desired but average the ambient derivatives (highlighted by blue point), which are obtained following the direct method.

$$(\frac{\partial u_y^*}{\partial x})_n = \frac{(\frac{\partial u_y^*}{\partial x})_{wn} + (\frac{\partial u_y^*}{\partial x})_{en} + (\frac{\partial u_y^*}{\partial x})_w + (\frac{\partial u_y^*}{\partial x})_e}{4} \tag{6}$$

Now consider constructing x component of Eq.3 for cell locates $(0,0)$. We herein denote x, y components of $\mathbf{u}$ as u, v but not u_x, u_y for sake of clearity. The $L.H.S$ can be directly expressed as

$$L.H.S = \Delta\rho_{n+\frac{1}{2}}[\frac{u_{0,0}^* - u_{0,0}'}{\Delta t}] \tag{7}$$

Then the first two terms of Eq.4 which can be obtained directly are

$$R.H.SF2 = 2\mu_{(\frac{1}{2},0)}(\frac{u_{1,0}^* - u_{0,0}^*}{\Delta}) - 2\mu_{(-\frac{1}{2},0)}(\frac{u_{0,0}^* - u_{-1,0}^*}{\Delta}) \tag{8}$$

Finally the last two terms that is caculated by averaging yeilds

$$\begin{aligned} R.H.SL2 = \mu_{(0,\frac{1}{2})}[&\frac{u_{(0,1)}^* - u_{(0,0)}^*}{\Delta} + \frac{v_{(1,1)}^* - v_{(-1,1)}^* + v_{(1,0)}^* - v_{(-1,0)}^*}{4\Delta}] \\ -\mu_{(0,-\frac{1}{2})}[&\frac{u_{(0,0)}^* - u_{(0,-1)}^*}{\Delta} + \frac{v_{(1,0)}^* - v_{(-1,0)}^* + v_{(1,-1)}^* - v_{(-1,-1)}^*}{4\Delta}] \end{aligned} \tag{9}$$

Rearrenge the equations we have

$$u_{(0,0)}^* = \frac{\frac{\Delta t}{\rho}(2\mu_{(\frac{1}{2},0)}u_{(1,0)}^* + 2\mu_{(-\frac{1}{2},0)}u_{(-1,0)}^* + \mathscr{A} - \mathscr{B}) + \Delta^2 u_{(0,0)}'}{\Delta^2 + \frac{\Delta t}{\rho}(2\mu_{(\frac{1}{2},0)} + 2\mu_{(-\frac{1}{2},0)} + \mu_{(0,\frac{1}{2})} + \mu_{(0,-\frac{1}{2})})} \tag{10}$$

where

$$\mathscr{A} = \mu_{(0,\frac{1}{2})}(u_{(0,1)}^* + \frac{v_{(1,1)}^* + v_{(1,0)}^* - v_{(-1,1)}^* - v_{(-1,0)}^*}{4}) \tag{11}$$

$$\mathscr{B} = \mu_{(0,-\frac{1}{2})}(-u_{(0,-1)}^* + \frac{v_{(1,-1)}^* + v_{(1,0)}^* - v_{(-1,-1)}^* - v_{(1,0)}^*}{4}) \tag{12}$$

Now, we obtain the implicit discrete linear expressions for desired valuable $\mathbf{u}^*$. Moreover, if we modify the governing equation into a more general form

$$\mathbf{u}' = \mathbf{u}^* - \frac{\Delta t\nabla\cdot[2\mu_{n+\frac{1}{2}}\mathbf{D}^*]}{\rho_{n+\frac{1}{2}}} = \mathscr{D}(\mathbf{u}^*) \tag{13}$$

here $\mathscr{D}$ is a linear operator that satisfies

$$RES^k = \mathbf{u}' - \mathscr{D}(\mathbf{u}^{*,k}) \tag{14}$$

$$\mathscr{D}(\delta\mathbf{u}^{*,k}) = RES^k \tag{15}$$

$$\mathbf{u}^{*,k+1} = \mathbf{u}^{*,k} + \delta\mathbf{u}^{*,k} \tag{16}$$

We denote by k the kth iterate approximating result of corresponding value. Then the multicycle sovler constructed in 'poisson.h' can be applied to solve viscosity equation with specific relaxation and residual fundtions, which are built in this headfile.

2 Functions

2.1 Predefine

2.1.1 Parameters

Name	Data type	Status	Option	Representation (before/after)
mu	face vector	unchange	complusory	$\mu^{n+\frac{1}{2}}$
rho	scalar	unchange	complusory	$\rho^{n+\frac{1}{2}}$
dt	double	unchange	complusory	Δt

2.1.2 Program Workflow

Pre-Define
Define data structure Viscosity called in the total sovler
mu=$\mu^{n+\frac{1}{2}}$ *rho*=$\rho^{n+\frac{1}{2}}$
dt=Δt
macro
macro defined here to set factor *lambda* especially for 'axi.h'. Otherwise, for cases in Cartesian mesh, the factor is set to be unity.

```
1   #include "poisson.h"
2
3   struct Viscosity {
4     face vector mu;
5     scalar rho;
6     double dt;
7   };
8
9   #if AXI
10  # define lambda ((coord){1., 1. +
    ↪  dt/rho[]*(mu.x[] + mu.x[1] + \
11                 mu.y[] +
↪   mu.y[0,1])/2./sq(y)})
12  #else // not AXI
13  # if dimension == 1
14  #   define lambda ((coord){1.})
15  # elif dimension == 2
16  #   define lambda ((coord){1.,1.})
17  # elif dimension == 3
18  #   define lambda
    ↪  ((coord){1.,1.,1.})
19  #endif
20  #endif
```

2.2 relax_viscosity

2.2.1 Parameters

Name	Data type	Status	Option	Representation (before/after)
a	scalar*	update	complusory	$\delta\mathbf{u}^{*,k}/\delta\mathbf{u}^{*,k+1}$
b	scalar*	unchange	complusory	RES
dt	double	unchange	complusory	Δt
l	int	unchange	complusory	mesh level
$data$	struct Vsicosity	unchange	complusory	$\mu^{n+\frac{1}{2}}, \rho^{n+\frac{1}{2}}, \Delta t$

2.2.2 Worth Mentioning Details

The purpose of this function is to implement Eq.10 and push forward iteration of G-S (or Jacobi) method for a single round on level l mesh. Note the object processed is $\delta\mathbf{u}^*$ defined previously instead of $\mathbf{u}^*$. Therefore, the equation it solves is Eq.15 with given residual RES computed in another function.

The Basilisk self-defined macro `foreach_level_or_leaf(l)` takes l as its parameter and traverses each cell on level l or leaf cell (the cell that cannot be divided) whose level is less than l. Interested readers are referred to 'poisson.h Documentation' or Van Hooft *et.al.*[1] for more information about mesh heiarchy in Basilisk.

2.2.3 Program Workflow

Starting Point
input:
$a = \delta\mathbf{u}^*$ $b = RES$
l=*meshlevel*
$data = \mu^{n+\frac{1}{2}}, \rho^{n+\frac{1}{2}}, \Delta t$
data assignment
$\mu, \rho, \Delta t$ is extracted from $data$
u and r serve as pointer and point to $\delta\mathbf{u}^*, RES$ respectively
iteration type
The algorithm of iteration is selected by macro, if JACOBI is false, then a pointer w is created and points to u (a). Then all modification hereinafter to w is direct exert on u.

```
1  static void relax_viscosity (scalar *
   ↪  a, scalar * b, int l, void *
   ↪  data)
2  {
3    struct Viscosity * p = (struct
     ↪  Viscosity *) data;
4    (const) face vector mu = p->mu;
5    (const) scalar rho = p->rho;
6    double dt = p->dt;
7    vector u = vector(a[0]), r =
     ↪  vector(b[0]);
8
9  #if JACOBI
10   vector w[];
11 #else
12   vector w = u;
13 #endif
```

Relaxation Compute
implement of Eq.10 for different dimension.
$w = \delta \mathbf{u}_{TBD}^{*,k+1}$

```
  foreach_level_or_leaf (l) {
    foreach_dimension()
     w.x[] = (dt/rho[]*(2.*mu.x[1]*u.x[1]
     ↪  + 2.*mu.x[]*u.x[-1]
#if dimension > 1
     + mu.y[0,1]*(u.x[0,1] +
      (u.y[1,0] + u.y[1,1])/4. -
      (u.y[-1,0] + u.y[-1,1])/4.)
     - mu.y[]*(- u.x[0,-1] +
         (u.y[1,-1] + u.y[1,0])/4. -
         (u.y[-1,-1] + u.y[-1,0])/4.)
#endif
#if dimension > 2
     + mu.z[0,0,1]*(u.x[0,0,1] +
        (u.z[1,0,0] + u.z[1,0,1])/4. -
        (u.z[-1,0,0] + u.z[-1,0,1])/4.)
     - mu.z[]*(- u.x[0,0,-1] +
         (u.z[1,0,-1] + u.z[1,0,0])/4. -
         (u.z[-1,0,-1] + u.z[-1,0,0])/4.)
#endif
     ) + r.x[]*sq(Delta))/
(sq(Delta)*lambda.x +
↪  dt/rho[]*(2.*mu.x[1] + 2.*mu.x[]
#if dimension > 1
     + mu.y[0,1] + mu.y[]
#endif
#if dimension > 2
     + mu.z[0,0,1] + mu.z[]
#endif
                                ));
  }
```

Update
If $JACOBI == TRUE$
u=$\delta \mathbf{u}^{*,k+1} = \frac{1}{3}\delta \mathbf{u}^{*,k} + \frac{2}{3}\delta \mathbf{u}_{TBD}^{*,k+1}$
else ($G - S$ iteration)
u=$\delta \mathbf{u}^{*,k+1} = \delta \mathbf{u}_{TBD}^{*,k+1}$
a is modified along with u

```
  #if JACOBI
    foreach_level_or_leaf (l)
      foreach_dimension()
        u.x[] = (u.x[] + 2.*w.x[])/3.;
  #endif

  #if TRASH
    vector u1[];
    foreach_level_or_leaf (l)
      foreach_dimension()
        u1.x[] = u.x[];
    trash ({u});
    foreach_level_or_leaf (l)
      foreach_dimension()
        u.x[] = u1.x[];
  #endif
  }
```

2.3 residual_viscosity

2.3.1 Parameters

Name	Data type	Status	Option	Representation (before/after)
a	scalar*	unchange	complusory	$\mathbf{u}^{*,k}$
b	scalar*	unchange	complusory	$\mathbf{u}'$
resl	scalar*	**output**	complusory	$RES^k = \mathbf{u}' - \mathscr{D}(\mathbf{u}^{*,k})$
data	struct Vsicosity	unchange	complusory	$\mu^{n+\frac{1}{2}}, \rho^{n+\frac{1}{2}}, \Delta t$

2.3.2 Worth Mentioning Details

This function updates residual by solving Eq.13 and returns the maximum component. Two computational methods are provided for tree and non-tree grid. The first method for tree grid costs more computational resource compared with the second one due to extra traversal or cell face but preserves 2nd order on tree grid. While the second method is of 2nd order with caresian but 1st with tree grid.

Consider quadtree for 2D cases shown in Fig.3 where coarser grid is of level l and finer grid is of level $l+1$. ■ represents example targert cell whose residual will be compute to display the difference between two methods. ● represents active point located at cell center where stores value for each cell. ● and ● are ghost cells which do not exist initially. ● is caculated by bilinear interpolation and is called as adjacent cell for finer cell while ● is obtained by averaging and is called from coarser cell. Moreover, cell faces highlighted by letters is another issue that woth mentioning. As can be seen from figure, target cell has five marked faces instead of four that face A is divided into subfaces A_1 and A_2 due to tree grid. However when called face value from centered cell perspective *i.e.* invoke value on face A from target cell, the value returned is the average of A_1 and A_2.

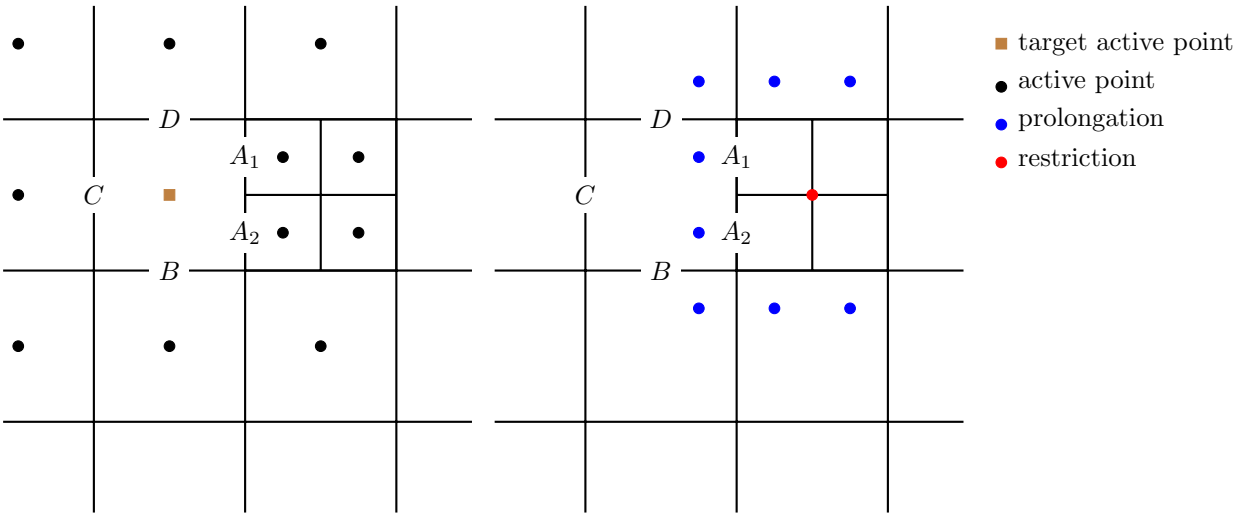

Figure 3: Sketch for 2D quadtree sample, active point and those gost cells are depicted separately for clearity.

The first method which conserves 2nd order on tree grid caculates and stores value at each face. Therefore, ● as well as ● at level $l+1$ is used to compute desired value at face A_1 and A_2. Then value on target cell is obtained by linear combination of its face value. In conclusion, all marked points (● is used when compute face value on B, D) in Fig.3 participate in determining target value.

However, second method employs direct way to caculate target value from cell centered stand. Therefore, target value is only decided by ● and ● on level l. Thus, applying second method to tree grid will lead to accuracy decrease. Otherwise, both methods act the same way on cartesian and the second method runs even faster.

2.3.3 Program Workflow

Starting Point
input:
$a = \mathbf{u}^{*,k}$ $b = \mathbf{u}'$
$resl$=RES^k
$data = \mu^{n+\frac{1}{2}}, \rho^{n+\frac{1}{2}}, \Delta t$
data assignment
$\mu, \rho, \Delta t$ is extracted from $data$
u, r and res serve as pointer and point to $\delta\mathbf{u}^{*,k}, \mathbf{u}', RES^k$ respectively.

```
static double residual_viscosity (scalar * a, scalar * b, scalar * resl,
          void * data)
{
  struct Viscosity * p = (struct Viscosity *) data;
  (const) face vector mu = p->mu;
  (const) scalar rho = p->rho;
  double dt = p->dt;
  vector u = vector(a[0]), r = vector(b[0]), res = vector(resl[0]);
  double maxres = 0.;
```


Tree Grid Update
ghost cell:
Set prolongation ghost cells for all components manually.
tree-flux compute
Compute flux tau on each face for three dimension based on Eq.4.
tree-flux assmeble
Traverse all cell and assmeble $\mathbf{D}^*$ as d from cell centered perspective.
residual compute
Then the residual is updated by Eq.13 and stored in res. The maximum of all components is returned through double data $maxres$.

```
#if TREE
  boundary ({u});

  foreach_dimension() {
    face vector taux[];
    foreach_face(x)
      taux.x[] = 2.*mu.x[]*(u.x[] - u.x[-1])/Delta;
    #if dimension > 1
      foreach_face(y)
        taux.y[] = mu.y[]*(u.x[] - u.x[0,-1] +
        (u.y[1,-1] + u.y[1,0])/4. -
        (u.y[-1,-1] + u.y[-1,0])/4.)/Delta;
    #endif
    #if dimension > 2
      foreach_face(z)
        taux.z[] = mu.z[]*(u.x[] - u.x[0,0,-1] +
        (u.z[1,0,-1] + u.z[1,0,0])/4. -
        (u.z[-1,0,-1] + u.z[-1,0,0])/4.)/Delta;
    #endif
    foreach (reduction(max:maxres)) {
      double d = 0.;
      foreach_dimension()
        d += taux.x[1] - taux.x[];
      res.x[] = r.x[] - lambda.x*u.x[] + dt/rho[]*d/Delta;
      if (fabs (res.x[]) > maxres)
        maxres = fabs (res.x[]);
    }
  }
```

NonTree Grid Update residual compute

Then the residual is updated by Eq.13 and stored in *res*. The maximum of all components is returned through double data *maxres*.

```
#else
  foreach (reduction(max:maxres))
    foreach_dimension() {
      res.x[] = r.x[] - lambda.x*u.x[] +
        dt/rho[]*(2.*mu.x[1,0]*(u.x[1] -
        ↪ u.x[])
        - 2.*mu.x[]*(u.x[] - u.x[-1])
  #if dimension > 1
        + mu.y[0,1]*(u.x[0,1] - u.x[] +
             (u.y[1,0] + u.y[1,1])/4. -
             (u.y[-1,0] + u.y[-1,1])/4.)
        - mu.y[]*(u.x[] - u.x[0,-1] +
            (u.y[1,-1] + u.y[1,0])/4. -
            (u.y[-1,-1] + u.y[-1,0])/4.)
        #endif
  #if dimension > 2
        + mu.z[0,0,1]*(u.x[0,0,1] - u.x[]
        ↪ +
           (u.z[1,0,0] + u.z[1,0,1])/4. -
           (u.z[-1,0,0] +
           ↪ u.z[-1,0,1])/4.)
        - mu.z[]*(u.x[] - u.x[0,0,-1] +
            (u.z[1,0,-1] + u.z[1,0,0])/4.
            ↪ -
            (u.z[-1,0,-1] +
            ↪ u.z[-1,0,0])/4.)
        #endif
                    )/sq(Delta);
      if (fabs (res.x[]) > maxres)
        maxres = fabs (res.x[]);
    }
#endif
  return maxres;
}

#undef lambda
```

2.4 viscosity

2.4.1 Parameters

Name	Data type	Status	Option/Default	Representation (before/after)
u	vector	update	complusory	$\mathbf{u}'/\mathbf{u}^*$
mu	face vector	unchange	complusory	$\mu^{n+\frac{1}{2}}$
rho	scalar	unchange	complusory	$\rho^{n+\frac{1}{2}}$
dt	double	unchange	complusory	Δt
nrelax	int	unchange	optional/4	$max\,of\,iteration$
res	scalar*	**output**	optional/NULL	RES

2.4.2 Worth Mentioning Details

The function to assemble all the tools built in this headfile or in 'poisson.h' and solve the governing equation Eq.1. Details about implicit iteration solver construction used here are explored in 'poisson.h Documentation' which shall not be repeated again.

2.4.3 Program Workflow

Solver Construction
input:
$u = \mathbf{u}'$ $mu = \mu^{n+\frac{1}{2}}$ $rho = \rho^{n+\frac{1}{2}}$
$dt = \Delta t$ $res = empty$
$nrelax = iteration\,number$
data assignment
r is created to store $\mathbf{u}'$ and u now serves as $\mathbf{u}^*$ which will be updated. Then mu and rho is restricted to all level and p is defined as self made struct and to store $\mu, \rho, \Delta t$.
iteration solver
Solver is built using **mg_solve** from 'poisson.h'

```
trace
mgstats viscosity (vector u, face vector mu, scalar rho, double dt, int nrelax = 4, scalar * res = NULL)
{
  vector r[];
  foreach()
    foreach_dimension()
      r.x[] = u.x[];

  restriction ({mu,rho});
  struct Viscosity p = { mu, rho, dt };
  return mg_solve ((scalar *){u}, (scalar *){r}, residual_viscosity,
  relax_viscosity, &p, nrelax, res);
}
```

2.5 viscosity_explicit

2.5.1 Parameters

Name	Data type	Status	Option/Default	Representation (before/after)
u	vector	update	complusory	$\mathbf{u}'/\mathbf{u}^*$
mu	face vector	unchange	complusory	$\mu^{n+\frac{1}{2}}$
rho	scalar	unchange	complusory	$\rho^{n+\frac{1}{2}}$
dt	double	unchange	complusory	Δt

2.5.2 Worth Mentioning Details

Different from implicit form shown in Eq.1, the governing equation of this function is explicit

$$\rho_{n+\frac{1}{2}}[\frac{\mathbf{u}^* - \mathbf{u}'}{\Delta t}] = \nabla \cdot [2\mu_{n+\frac{1}{2}}\mathbf{D}'] \tag{17}$$

Then

$$\mathbf{u}^* = \mathbf{u}' + \frac{\Delta t \nabla \cdot [2\mu_{n+\frac{1}{2}}\mathbf{D}']}{\rho_{n+\frac{1}{2}}} \tag{18}$$

According to Sec.2.3, if a, b of **viscosity_residual** are set to be the same and equal to $\mathbf{u}'$ then the output yields

$$RES = \frac{\Delta t \nabla \cdot [2\mu_{n+\frac{1}{2}}\mathbf{D}']}{\rho_{n+\frac{1}{2}}} \tag{19}$$

Hence

$$\mathbf{u}^* = \mathbf{u}' + \frac{\Delta t \nabla \cdot [2\mu_{n+\frac{1}{2}}\mathbf{D}']}{\rho_{n+\frac{1}{2}}} = \mathbf{u}' + RES \tag{20}$$

2.5.3 Program Workflow

Solver Construction

input:

$u = \mathbf{u}'$ $mu = \mu^{n+\frac{1}{2}}$ $rho = \rho^{n+\frac{1}{2}}$

$dt = \Delta t$

explicit solver

$$r = \frac{\Delta t \nabla \cdot [2\mu_{n+\frac{1}{2}} \mathbf{D}']}{\rho_{n+\frac{1}{2}}}$$

u then updates to $\mathbf{u}^*$

```
trace
mgstats viscosity_explicit (vector u,
↪  face vector mu, scalar rho, double
↪  dt)
{
  vector r[];
  mgstats mg = {0};
  struct Viscosity p = { mu, rho, dt };
  mg.resb = residual_viscosity ((scalar
  ↪  *){u}, (scalar *){u}, (scalar
  ↪  *){r}, &p);
  foreach()
    foreach_dimension()
      u.x[] += r.x[];
  return mg;
}
```

References

[1] J Antoon Van Hooft et al. “Towards adaptive grids for atmospheric boundary-layer simulations”. In: *Boundary-layer meteorology* 167 (2018), pp. 421–443.