

# Simulating Two-Phase Incompressible Flow using Level Set Approach

MAE510 - Computational Moving Interfaces - Course Project

Instructor - Prof. David Salac

University at Buffalo

Pranav Ladkat (5009-4671)

## 1 Abstract

A level set approach for computing solutions to incompressible two-phase flow is presented. The interface between two fluids is described as zero level set of a smooth function. The incompressible flow field is solved using second order Projection Method by Kim and Moin[1]. A Continuum Surface Force method[2] is used for modeling Surface Tension. The case of low density fluid bubble rising in denser fluid is considered.

**Keywords:** Level Sets, Two-Phase Incompressible Flow, Projection Method.

## 2 Introduction

## 3 Description of Algorithm

### 3.1 Governing Equations

In the present study, we consider the motion of low density fluid rising in denser fluid. As the velocity field developed is much less than the velocity of sound, the incompressible form of Navier-Stokes equation is modeled with the incompressibility constraint. The equations of motion are:

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla) \mathbf{u} = \frac{1}{\rho} (-\nabla p + \nabla \cdot (2\mu \mathcal{D}) - \sigma k \delta(d) \mathbf{n}) + \mathcal{F} \quad (1)$$

$$\nabla \cdot \mathbf{u} = 0 \quad (2)$$

where  $\mathbf{u} = (u, v)$  is the fluid velocity,  $\rho$  is the density,  $\mu$  is viscosity,  $\mathcal{D} = \frac{1}{2}(\nabla \mathbf{u} + \nabla \mathbf{u}^T)$  is the rate of deformation tensor and  $\mathcal{F}$  is the body force due to gravity. The surface tension term is considered to be a force concentrated on the interface. The derivation of this form can be found in [2]. Surface tension is denoted by  $\sigma$ , curvature of the front by  $k$ , Dirac delta function by  $\delta$ , and the normal to interface by  $\mathbf{n}$ . Boundary of the domain is assumed as solid wall and the no slip boundary condition is applied.

The non-dimensional form of (1) is

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla) \mathbf{u} + \mathbf{g}_u + \frac{1}{\rho} \left( -\nabla p + \frac{1}{Re} \nabla \cdot (2\mu \mathcal{D}) - \frac{1}{B} k \delta(d) \mathbf{n} \right) \quad (3)$$

The key parameters are  $\rho_b/\rho_c$  and  $\mu_b/\mu_c$ , which are dimensionless density and viscosity inside the bubble respectively.  $\rho_b$  and  $\mu_b$  represents density and viscosity of fluid inside bubble respectively, and  $\rho_c$  and  $\mu_c$  denotes the same for outer continuum fluid; The Reynolds Number is  $Re = (2R)^{3/2} \sqrt{g \rho_c / \mu_c}$ ; and the Bond Number,  $B = 4\rho_c g R^2 / \sigma$ . Where characteristic length  $L = R$  and characteristic velocity  $U = \sqrt{gR}$  is used.  $R$  denotes the initial radius of bubble. The dimensionless density and viscosity outside the bubble are equal to 1.  $\mathbf{g}_u$  denotes the unit gravitational force.

### 3.2 Level Set Description

In the present study, the interface between gas and liquid is tracked with the level set function  $\phi$  and the interface  $\Gamma$ , is the zero level set of  $\phi$ :

$$\Gamma = \{\mathbf{x} | \phi(\mathbf{x}, t) = 0\}$$

The level set function  $\phi$  is positive in the liquid and negative in gas. Hence we have

$$\phi(x, t) = \begin{cases} > 0 & \text{if } x \in \text{the liquid} \\ = 0 & \text{if } x \in \Gamma \\ < 0 & \text{if } x \in \text{the gas} \end{cases} \quad (4)$$

The following equation will move the zero level set of  $\phi$  exactly as the actual bubble interface moves.

$$\frac{\partial \phi}{\partial t} + \mathbf{u} \cdot \nabla \phi = 0 \quad (5)$$

Since  $\phi$  is a smooth function, unlike  $\rho$  and  $\mu$ , the above equation can be easily solved numerically than

solving for  $\rho$  and  $\mu$  separately which are discontinuous at the interface. Hence,  $\rho$  and  $\mu$  in terms of  $\phi$  can be written as

$$\rho(\phi) = H(\phi) + \frac{\rho_b}{\rho_c}(1 - H(\phi)) \quad (6)$$

$$\mu(\phi) = H(\phi) + \frac{\mu_b}{\mu_c}(1 - H(\phi)) \quad (7)$$

Where,  $H(\phi)$  is a smoothed Heaviside function to avoid numerical instabilities. The smoothed Heaviside function is defined as

$$H(\phi) = \begin{cases} 0 & \text{if } \phi < -\epsilon \\ \frac{1}{2} \left[ 1 + \frac{\phi}{\epsilon} + \frac{1}{\pi} \sin\left(\frac{\pi\phi}{\epsilon}\right) \right] & \text{if } |\phi| \leq \epsilon \\ 1 & \text{if } \phi > \epsilon \end{cases} \quad (8)$$

The parameter  $\epsilon$  can be used to define the interface thickness and its value is set to  $\alpha\Delta x$ , where  $\alpha = \frac{3}{2}$  is used in the presents study which ensures that  $H(\phi)$  varies smoothly over three grid spacings.

The surface tension force represented by  $\frac{1}{B}k\delta(d)\mathbf{n}$  in (3), can be cast in the level set formulation as pointed out in [2]

$$\frac{1}{B}k\delta(d)\mathbf{n} = \frac{1}{B}k(\phi)\delta(\phi)\nabla\phi \quad (9)$$

where the curvature is

$$k(\phi) = \nabla \cdot \left( \frac{\nabla\phi}{|\nabla\phi|} \right) = \frac{\phi_{xx}\phi_y^2 + \phi_{yy}\phi_x^2 - 2\phi_{xy}\phi_x\phi_y}{(\phi_x^2 + \phi_y^2)^{3/2}} \quad (10)$$

If  $\phi$  is maintained as signed distance function, as is done here, then we may numerically approximate  $\delta(\phi)$  by a mollified delta function as

$$\delta(\phi) = \begin{cases} \frac{1}{2\epsilon} \left[ 1 + \cos\left(\frac{\pi\phi}{\epsilon}\right) \right] & \text{if } |\phi| < \epsilon, \\ 0 & \text{otherwise.} \end{cases} \quad (11)$$

where  $\epsilon$  is same as that used in Heaviside function (8).

### 3.3 Reinitialization

While equation (5) will move the level set  $\phi = 0$  at the correct velocity,  $\phi$  will not longer be a distance function and  $\phi$  can become irregular after some period of time. Hence frequent reinitialization is necessary to keep  $\phi$  a distance function. In order to do so, we must be able to solve the following problem/; given a level set function  $\phi(x, t)$ , reinitialize it so that it is a distance function for  $|\phi| < \epsilon$  without changing its zero level set. This is achieved if we solve

$$\frac{\partial\phi}{\partial\tau} = \text{sign}(\phi_0) \left( 1 - \sqrt{\phi_x^2 + \phi_y^2} \right) \quad (12)$$

$$\phi(x, 0) = \phi_0(x) \quad (13)$$

Where,  $\tau$  is pseudo time. For numerical purposes it is useful to smooth the sign function. Here, smoothed sign function used is

$$\text{sign}(\phi) = \frac{\phi}{\sqrt{\phi^2 + |\nabla\phi|^2 h^2}} \quad (14)$$

The equation (12) is a hyperbolic equation with the characteristic velocities pointing outwards from the interface in the direction of the normal. This means that  $\phi$  will be reinitialized to  $|\nabla\phi| = 1$  near the interface first. Since we need  $\phi$  to be distance function near interface, it is only necessary to solve equation (12) for few iterations.

### 3.4 Projection Method

Projection Method is a way of solving Incompressible Navier-Stokes equation which has been a most preferred choice for transient flow simulations. In its most basic form, the projection method requires the solution of advection-diffusion equations, which are then projected onto the space of divergence-free vector fields. The projection exploits the orthogonality of the pressure gradient with divergence-free velocity fields that is manifest in the Hodge decomposition. In the present study, pressure-free projection method by Kim and Moin [1] is used. In projection methods, the momentum equation is split in to two equations. The first equation predicts an intermediate velocity which is in general not divergence free, and in second step the incompressibility condition is enforced by solving an elliptic equation.

For the diffusion-convection step we solve for the intermediate velocity using

$$\frac{\partial\mathbf{u}}{\partial t} = -[(\mathbf{u} \cdot \nabla) \mathbf{u}]^{n+1/2} + \frac{1}{\rho} \left( \frac{1}{2} \mathcal{L}^n - \mathcal{T}^n \right) + \mathbf{g}_u \quad (15)$$

Where,  $\partial\mathbf{u}/\partial t$  represents temporal discretization as discussed in section (4.1),  $\mathcal{L}$  represents second order finite difference approximation to  $(1/Re)\nabla \cdot (2\mu(\phi)\mathcal{D})$ ,  $\mathcal{T}$  is a finite difference approximation to  $(1/B)k(\phi)\delta(\phi)\nabla\phi$  and  $\mathbf{g}_u$  represents unit gravitational force in downward direction. Solution of the equation (15) will result in the intermediate velocity field  $\mathbf{u}^*$ .

The incompressibility condition is enforced by solving the following elliptic equation

$$\frac{\nabla \cdot \mathbf{u}^*}{\Delta t} = \nabla \cdot \left( \frac{1}{\rho} (\nabla Q) \right) \quad (16)$$

where  $Q$  is the scalar field which behaves similar to the pressure. After computing  $Q$ , the velocities can

be updated to time level  $n + 1$  using

$$\mathbf{n}^{n+1} = \mathbf{u}^* - \Delta t \left( \frac{1}{\rho} \nabla Q \right) \quad (17)$$

which will be divergence-free.

### 3.5 Algorithm Summary

The algorithm is summarized in the following steps:  
*Step 1.* Initialize  $\phi(x, t)$  to be signed normal distance to the front.

*Step 2.* Define variables  $\rho(\phi)$  and  $\mu(\phi)$  using equations (6) and (7), respectively.

*Step 3.* Solve for intermediate velocity field by equation (15), projection step by equation (16) and then update velocity to time level  $n + 1$  using equation (17).

*Step 4.* Advect the level set function using equation (5)

*Step 5.* Reinitialize level set function by solving equation (12) to steady state.

*Step 6.* We have now advanced one time step, repeat *Step 2* to *Step 5* till solution reaches final time.

## 4 Discretization

A collocated grid is used where velocity, pressure and level set functions are described at the grid points. For the uniform grid spacing, we have  $\Delta x = \Delta y = h$ . With  $h$  as mesh size, we define

$$\begin{aligned} \mathbf{x}_{i,j} &= (i * h, j * h) \\ \mathbf{u}_{i,j} &= \mathbf{u}(\mathbf{x}_{i,j}) \\ \phi_{i,j} &= \phi(\mathbf{x}_{i,j}) \\ i &= 0 \dots M, j = 0 \dots N \end{aligned}$$

The domain used in the present simulations is a square box with  $Mh = Nh = 7R$ . The arrangement of the discrete variables  $\mathbf{u}$ ,  $p$  and  $\phi$  on the grid is shown in figure (1). All variables are located at cell centroid.

### 4.1 Temporal Discretization

1. Adam Bashforth Time stepping for intermediate velocity  $\mathbf{U}^*$ :

The time derivative in equation (15) are approximated using second order accurate Adams-Bashforth method to compute intermediate velocity

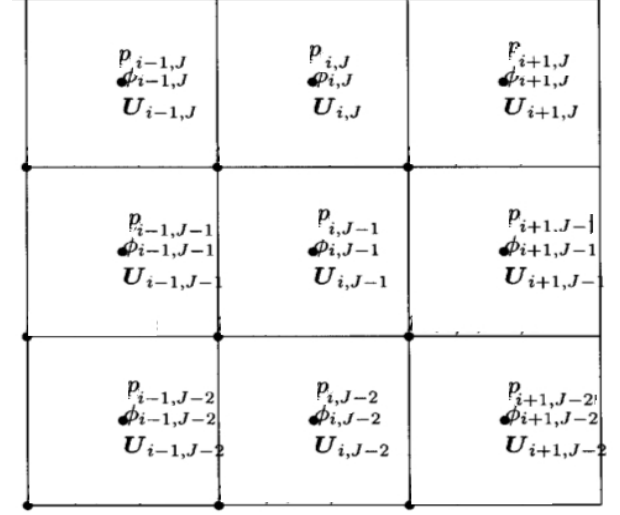


Figure 1: Collocated grid arrangement where discrete variables  $\mathbf{u}$ ,  $p$  and  $\phi$  are located

field. Hence the equation (15) is written as

$$\begin{aligned} \mathbf{u}^* &= \mathbf{u}^n \\ &+ \frac{\Delta t^n}{2} \left( \frac{\Delta t^n + 2\Delta t^{n-1}}{\Delta t^{n-1}} RHS^n - \frac{\Delta t^n}{\Delta t^{n-1}} RHS^{n-1} \right) \end{aligned} \quad (18)$$

where  $\Delta t^n$  and  $\Delta t^{n-1}$  denotes the time step size at time level  $n$  and  $n - 1$  respectively.  $RHS^n$  is  $[-[(\mathbf{u} \cdot \nabla) \mathbf{u}]^{n+1/2} + (\frac{1}{2} \mathcal{L}^n - \mathcal{T}^n) / \rho + \mathbf{g}_u]^n$  and  $RHS^{n-1}$  is defined similarly.

2. Level Set update for  $\phi^{n+1}$ :

For updating the level set function to a new time step, an explicit Euler time stepping is used for approximating time derivative in equation (5), which can then be written as

$$\frac{\phi^{n+1} - \phi}{\Delta t} + \mathbf{u} \cdot \nabla(\phi) = 0 \quad (19)$$

### 4.2 Spatial Discretization

This section describes all the spatial discretization used for the governing equations. As noted earlier, all discretization uses collocated grid.

#### 4.2.1 Convective Term

The discretization of the convective term in this algorithm is very similar to the discretization used by [1]. It is a second order Adam-Bashforth discretization in

conjunction with centered difference approximation.

$$[(\mathbf{u} \cdot \nabla) \mathbf{u}]^{n+1/2} = \frac{\Delta t^n}{2} \left( \frac{\Delta t^n + 2\Delta t^{n-1}}{\Delta t^{n-1}} [(\mathbf{u} \cdot \nabla) \mathbf{u}]^n - \frac{\Delta t^n}{\Delta t^{n-1}} [(\mathbf{u} \cdot \nabla) \mathbf{u}]^{n-1} \right) \quad (20)$$

where,

$$[(\mathbf{u} \cdot \nabla) \mathbf{u}] = \begin{cases} u_{i,j} \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} + v_{i,j} \frac{u_{i,j+1} - u_{i,j-1}}{2\Delta y} \\ u_{i,j} \frac{v_{i+1,j} - v_{i-1,j}}{2\Delta x} + v_{i,j} \frac{v_{i,j+1} - v_{i,j-1}}{2\Delta y} \end{cases} \quad (21)$$

#### 4.2.2 Viscous Term

The viscous terms are discretized in straight forward central difference scheme. The viscous term  $\nabla \cdot [\mu(\nabla \mathbf{u} + \nabla \mathbf{u}^T)]$  is written in  $x$  and  $y$  component as:

$$\text{x-direction : } \frac{\partial}{\partial x} \left( 2\mu \frac{\partial u}{\partial x} \right) + \frac{\partial}{\partial y} \left[ \mu \left( \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \right) \right], \quad (22)$$

$$\text{y-direction : } \frac{\partial}{\partial x} \left[ \mu \left( \frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} \right) \right] + \frac{\partial}{\partial y} \left( 2\mu \frac{\partial v}{\partial y} \right). \quad (23)$$

And the derivatives are discretized as:

$$\frac{\partial}{\partial x} \left( \mu \frac{\partial u}{\partial x} \right) = \frac{\mu_{i+1/2,j} \left( \frac{\partial u}{\partial x} \right)_{i+1/2,j} - \mu_{i-1/2,j} \left( \frac{\partial u}{\partial x} \right)_{i-1/2,j}}{\Delta x} \quad (24)$$

Similar formulation for  $\frac{\partial}{\partial y} \left( \mu \frac{\partial u}{\partial y} \right)$  and other terms is used.

$$\begin{aligned} \frac{\partial}{\partial x} \left( \mu \frac{\partial u}{\partial y} \right) = & \mu_{i+1/2,j} \frac{u_{i+1,j+1} - u_{i+1,j-1} + u_{i,j+1} - u_{i,j-1}}{4\Delta x^2} \\ & - \mu_{i-1/2,j} \frac{u_{i,j+1} - u_{i,j-1} + u_{i-1,j+1} - u_{i-1,j-1}}{4\Delta x^2} \end{aligned} \quad (25)$$

Similar formulation is used for  $\frac{\partial}{\partial y} \left( \mu \frac{\partial u}{\partial x} \right)$  and other similar cross derivative terms.

Where,

$$\mu_{i+1/2,j} = \frac{\mu_{i+1,j} + \mu_{i,j}}{2} \quad (26)$$

$$\left( \frac{\partial u}{\partial x} \right)_{i+1/2,j} = \frac{u_{i+1,j} - u_{i-1,j}}{\Delta x} \quad (27)$$

#### 4.2.3 Curvature Terms

The surface tension equation used is (9), where  $k(\phi)$  and  $\delta(\phi)$  is given by equation (10) and (11), respectively. The curvature term is discretized making use of equation (24) and (25). The value for curvature  $k(\phi)$  is restricted between  $\pm \frac{1}{\Delta x}$ .

#### 4.2.4 Discretization of projection step

The projection step is given by equation (16). The correct discretization of the projection step operators is very essential on collocated grid to avoid possible checkerboard effect.

1. Discretization of  $\nabla \cdot \mathbf{u}^*$  :

$$\nabla \cdot \mathbf{u}^* = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}$$

where,

$$\frac{\partial u}{\partial x} = \begin{cases} \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} & \text{when not at boundary} \\ \frac{-3u_{i,j} + 4u_{i+1,j} - u_{i+2,j}}{2\Delta x} & \text{at left boundary} \\ \frac{3u_{i,j} - 4u_{i-1,j} + u_{i-2,j}}{2\Delta x} & \text{at right boundary} \end{cases} \quad (28)$$

Similar formulation is used for  $\frac{\partial v}{\partial y}$ .

2. Discretization of  $\nabla \cdot \left( \frac{1}{\rho} (\nabla Q) \right)$  :

$$\nabla \cdot \left( \frac{1}{\rho} (\nabla Q) \right) = \frac{\partial}{\partial x} \left( \frac{1}{\rho} (\nabla Q) \right) + \frac{\partial}{\partial y} \left( \frac{1}{\rho} (\nabla Q) \right) \quad (29)$$

$$\begin{aligned} \frac{\partial}{\partial x} \left( \frac{1}{\rho} (\nabla Q) \right) = & \frac{1}{\rho_{i+1/2,j}} \left( \frac{Q_{i+1,j} - Q_{i,j}}{\Delta x} \right) \\ & - \frac{1}{\rho_{i-1/2,j}} \left( \frac{Q_{i,j} - Q_{i-1,j}}{\Delta x} \right) \end{aligned} \quad (30)$$

Similar discretization is used for  $\frac{\partial}{\partial y} \left( \frac{1}{\rho} (\nabla Q) \right)$ .

Also,

$$\frac{1}{\rho_{i+1/2,j}} = \frac{2}{\rho_{i+1,j} + \rho_{i,j}} \quad (31)$$

#### 4.2.5 Reinitialization Equation

The reinitialization equation is given by (12) and due to its hyperbolic nature, it needs to be discretized using conservative schemes. Here first order Godunov scheme is used, also known as automatic upwind scheme. The discrete form of the equation (12) can be written as

$$\phi^{n+1} = \phi^n - \Delta t * \hat{H}(\phi_{i,j}) \quad (32)$$

where  $\hat{H}(\phi_x, \phi_y)$  denotes discretized form of  $\text{sign}(\phi_0) \left(1 - \sqrt{\phi_x^2 + \phi_y^2}\right)$ .

We define,

$$\begin{aligned} a &= D_x^- \phi_{i,j} = (\phi_{i,j} - \phi_{i-1,j})/\Delta x \\ b &= D_x^+ \phi_{i,j} = (\phi_{i+1,j} - \phi_{i,j})/\Delta x \\ c &= D_y^- \phi_{i,j} = (\phi_{i,j} - \phi_{i,j-1})/\Delta y \\ d &= D_y^+ \phi_{i,j} = (\phi_{i,j+1} - \phi_{i,j})/\Delta y \\ S &= \text{sign}(\phi_0) \end{aligned} \quad (33)$$

and

$$\begin{aligned} \hat{H}(\phi_{i,j}) &= S^+ (\sqrt{\max((a^+)^2, (b^-)^2) + \max((c^+)^2, (d^-)^2)} - 1) \\ &+ S^- (\sqrt{\max((a^-)^2, (b^+)^2) + \max((c^-)^2, (d^+)^2)} - 1) \end{aligned} \quad (34)$$

where the notation,

$$\begin{aligned} F^+ &= \max(F, 0) \\ F^- &= \min(F, 0) \end{aligned} \quad (35)$$

A Sub-cell fix by Russo and Smereka [4] is used near the interface to achieve 2nd order accuracy near interface. We update  $\phi$  near the interface using

$$\phi^{n+1} = \phi^n - \frac{\Delta T}{\Delta x} (S|\phi_{i,j}^0| - D_{i,j}) \quad (36)$$

where,

$$D_{i,j} = \frac{2\Delta x \phi_{i,j}^0}{\sqrt{(\phi_{i+1,j}^0 - \phi_{i-1,j}^0)^2 + (\phi_{i,j+1}^0 - \phi_{i,j-1}^0)^2}} \quad (37)$$

And this subcell fix is used when any of the following condition is satisfied.

$$\begin{aligned} &(\phi_{i+1,j}^0 * \phi_{i,j}^0 < 0), \text{ or} \\ &(\phi_{i-1,j}^0 * \phi_{i,j}^0 < 0), \text{ or} \\ &(\phi_{i,j+1}^0 * \phi_{i,j}^0 < 0), \text{ or} \\ &(\phi_{i,j-1}^0 * \phi_{i,j}^0 < 0) \end{aligned} \quad (38)$$

#### 4.2.6 Level Set Equation

The level set equation is given by (5). A simple explicit euler scheme is used for approximating time derivative and can be written in discretized form as

$$\phi_{i,j}^{n+1} = \phi_{i,j}^n - \Delta t (u\phi_x + v\phi_y)_{i,j} \quad (39)$$

A simple first order upwind scheme is used for spatial discretization,

$$(\phi_x)_{i,j} = \begin{cases} (\phi_{i,j} - \phi_{i-1,j})/\Delta x, & \text{when } u_{i,j} > 0 \\ (\phi_{i+1,j} - \phi_{i,j})/\Delta x, & \text{when } u_{i,j} < 0 \\ 0, & \text{otherwise} \end{cases} \quad (40)$$

Similar discretization is used for  $\phi_y$ .

### 5 Computation of $\Delta t$

An adaptive time step is used in the current work. The time step is computed at each time steps which obeys CFD condition due to convective terms, restriction due to source terms (gravity and surface tension) and due to viscous terms. It is computed using following criteria:

$$\begin{aligned} \Delta t_s &= \sqrt{\frac{B(\rho_b + \rho_c)}{8\pi}} \Delta x^{3/2} \\ \Delta t_v &= \min \left( \frac{3}{14} (\rho Re \Delta x^2 / \mu) \right) \\ \Delta t_c &= \min \left( \frac{\Delta x}{|\mathbf{u}|} \right) \\ \Delta t_F &= \sqrt{\frac{2\Delta x}{\mathcal{F}^n}} \text{ where, } \mathcal{F}^n = \left| \frac{\mathcal{L}^n}{\rho} - \frac{\mathcal{T}^n}{\rho} + \mathbf{g}_u \right| \end{aligned} \quad (41)$$

$$\Delta t = 0.5 \min(\Delta t_s, \Delta t_v, \Delta t_c, \Delta t_F)$$

### 6 Volume Correction

When Level set Method is used in multiphase flow problems, there is a significant volume loss (or mass loss) observed. The attempt to overcome this drawback is studied by various researchers and many different methods has been proposed. This error usually comes from the low order accurate discretization of reinitialization equation and can be reduced significantly by using higher order schemes such as 3<sup>rd</sup>/5<sup>th</sup> order accurate ENO or WENO. For the present work, a simple volume correction is added to the  $\phi$  after the interface is advanced in time by

$$\phi^{n+1} = \phi^{n+1} + \Delta \phi \quad (42)$$

where  $\Delta\phi$  is a volume correction term given by,

$$\Delta\phi = \frac{V^0 - V^n}{L^n} \quad (43)$$

where,  $V^0$  is the volume of the initial bubble at time  $t = 0$  and  $V^n$  is the volume of the bubble at time  $t = n$ . And  $L^n$  denotes the length of the interface at time  $t = n$ . The volume and the length of the bubble can be computed by,

$$V = \int_{\Omega} (1 - H(\phi)) d\Omega = \sum_{i,j} (1 - H(\phi_{i,j})) \Delta x \Delta y \quad (44)$$

$$L = \int_{\Omega} \delta(\phi) d\Omega = \sum_{i,j} (\delta(\phi_{i,j})) \Delta x \Delta y \quad (45)$$

$H$  and  $\delta$  are smoothed Heaviside and delta functions and are given by equations (8) and (11) respectively.

## 7 Analysis of results

The above code is tested with the classical problem of multiphase flow which simulates a flow of lower density bubble rising in denser fluid. The key parameters are density ratio ( $\rho_c/\rho_b$ ), viscosity ratio ( $\mu_c/\mu_b$ ), Bond number  $B = 4\rho_c g R^2/\sigma$ , and Reynolds number  $Re = (2R)^{3/2} \sqrt{g\rho_c/\mu_c}$ . The radius of the bubble,  $R$  is taken as 1 unit, computational domain is  $7R \times 7R$ . The grid of  $80 \times 80$  nodes is used for the results shown in subsequent sections. No slip boundary condition is applied to all 4 boundaries of the domain hence the flow is generated solely due to density difference. The result is shown for two different cases.

### 7.1 Grid Independent study

The grid independent study is performed to obtain a grid independent solution which does not change much when the grid is refined further. The figures below shows the solution of interface and flow field at time  $t = 3$  sec for  $Re = 100$ ,  $B = 200$ , density and viscosity ratio of 40. The figure (2) shows the interface at 3 sec with the velocity field for the different grids. It can be seen that the solution obtained from  $80 \times 80$  grid matched closely with  $100 \times 100$  grid.

### 7.2 Case 1: Flow for $Re = 5$ , $Bo = 1$

This case simulates flow with very low Reynolds number ( $Re = 5$ ) and low bond number ( $B = 1$ ) i.e. high surface tension. The density and viscosity ratio of 40 is used and the grid of  $80 \times 80$  is used. It can be seen that at low Reynolds number, the bubble does not

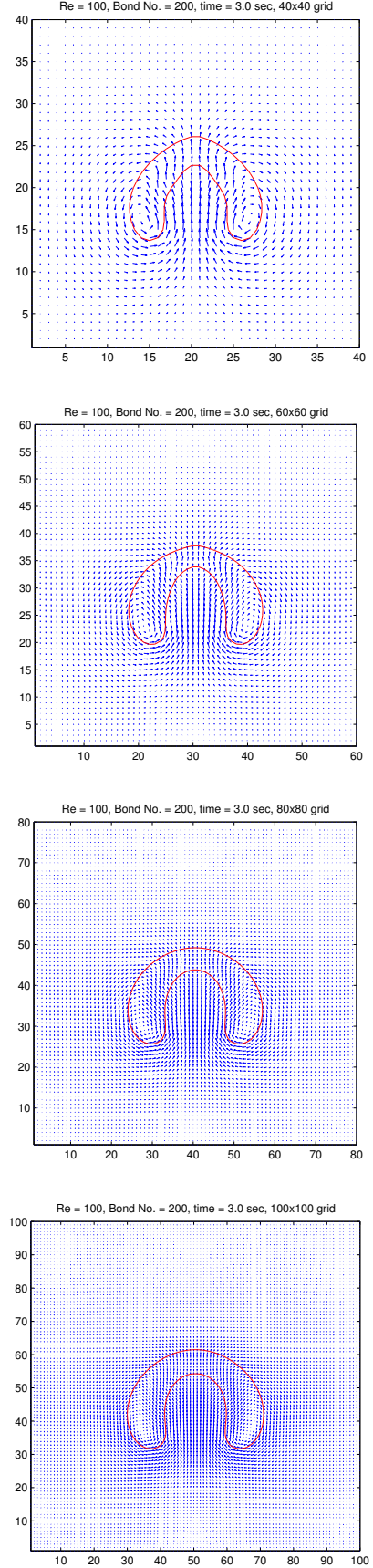


Figure 2: Grid dependence test for grid of  $40 \times 40$ ,  $60 \times 60$ ,  $80 \times 80$ ,  $100 \times 100$

deform much and obtains a steady state ellipsoidal shape and continues to rise up. The solution at time 1, 2, 3 and 4.2 seconds is shown in figure (3).

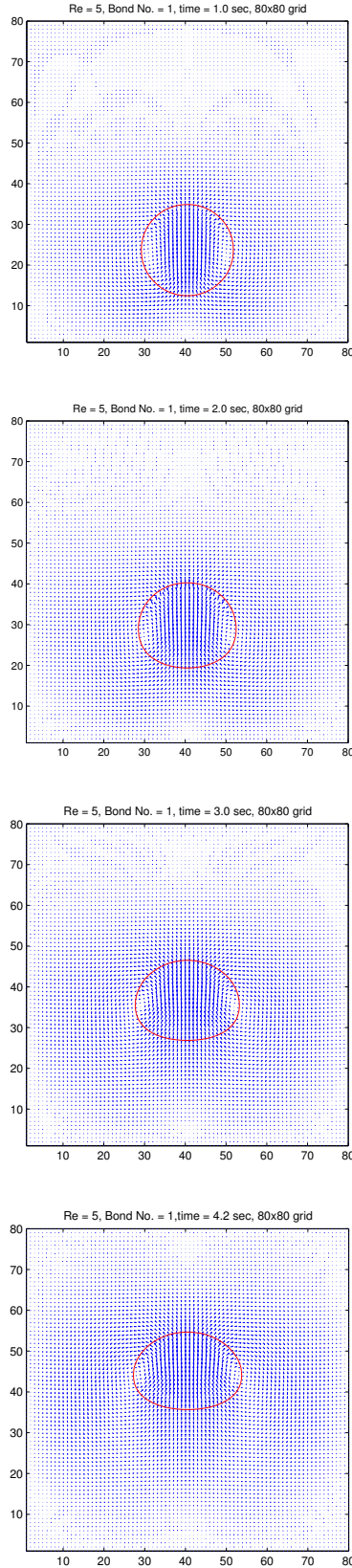


Figure 3: Solution of flow for Reynolds number 5 and Bond Number 1 at different times

### 7.3 Case 2: Flow for $Re = 100$ , $Bo = 200$

The flow with Reynolds number 100 and Bond number 200 i.e. low surface tension is simulated with density and viscosity ratio of 40. The grid of  $80 \times 80$  is used. The behavior of the bubble is very different than that of  $Re = 1$  and  $Bo = 5$  case. The bubbles deforms significantly due to very low surface tension and relatively higher Reynolds number.

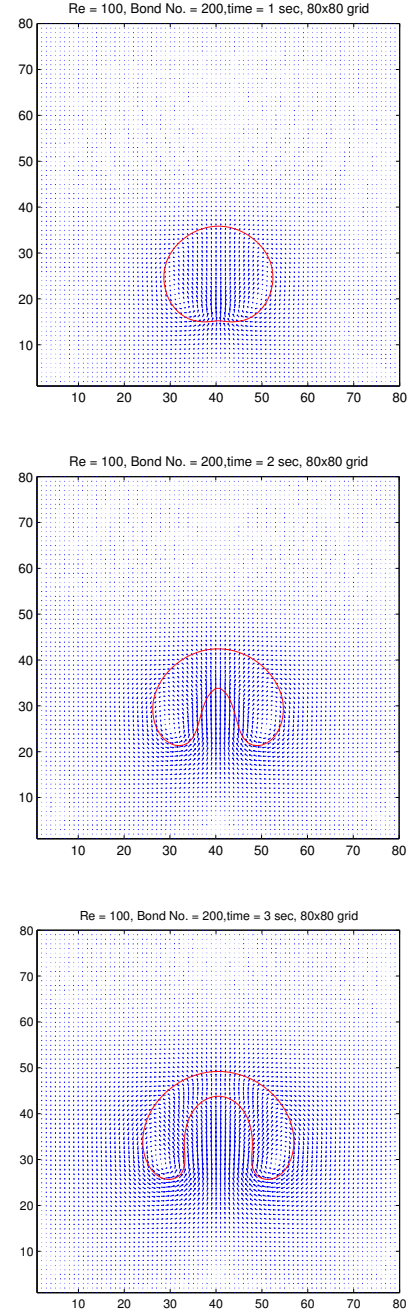


Figure 4: Solution of flow for Reynolds number 100 and Bond Number 200 at different times

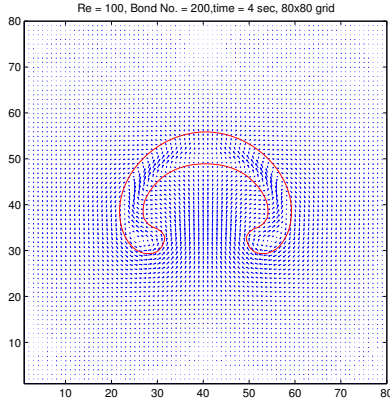


Figure 5: Solution of flow for Reynolds number 100 and Bond Number 200 at different times

#### 7.4 Effect of Volume Correction

The volume correction employed in this code is mentioned in detailed in section (6). The effect of volume correction is shown in the figure (6) where the volume of bubble at each iterations is plotted for the simulation of case 2 with correction term and without. The volume loss can be clearly seen when no volume correction is considered. For the case where volume correction is considered, the volume is very close to the initial volume but is oscillating slightly.

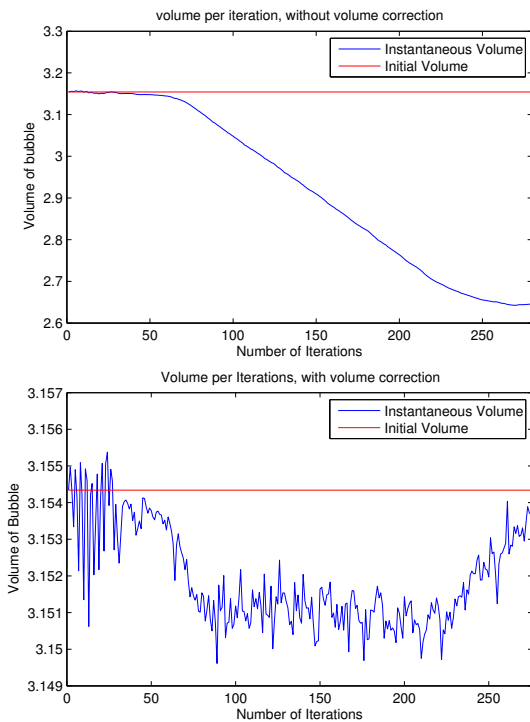


Figure 6: instantaneous Volume at each iteration, with and without volume correction

## 8 Conclusion

A two-phase incompressible Navier-Stokes flow solver with Level Set method to track the interface is developed and tested with the classical problem on rising bubble. The resulting flow field by solving Navier-Stokes equation is second order accurate in space and time however the discretization of level set is first order accurate. Moreover the effects of surface tension are modeled by using Continuum Surface Force method which is first order near interface. Due to low order discretization used in level set, there is significant volume loss which is corrected by a volume correction method mentioned in section 6. The projection step is solved using GMRES solver built in PETSc library. The density and viscosity ratio used in the simulation are moderate and could get solution for density ratio higher than 100. I think with the use of proper pre-conditioner this limit can be extended to allow higher density ratios.

## References

- [1] J. Kim and P. Moin, *Application of a Fractional-Step Method to Incompressible Navier-Stokes equations*, J. Computational Physics, **59**, 308-323 (1985)
- [2] J. U. Brackbill, *A Continuum Method for Modeling Surface Tension*, J. Computational Physics, **100**, 335-354 (1992)
- [3] David L. Brown, Ricardo Cortez and Michael L. Minion, *Accurate Projection Methods for the Incompressible NavierStokes Equations*, J. Computational Physics, **168**, 464-499 (2001)
- [4] G. Russo and P. Smereka, *A Remark on Computing Distance Functions*, J. Computational Physics, **163**, 51-67 (2000)



# Appendix: C Program

```
1 static char help[] = "Solves 2D multiphase flow problem using level set formulation.\n\n";
2
3 #include <petscdm.h>
4 #include <petscdmda.h>
5 #include <petscksp.h>
6 #include <sys/types.h>
7 #include <sys/stat.h>
8 #include <engine.h> //matlab engine
9
10
11 typedef struct{
12     Vec U, V, P, rho, mu;
13 }Solution;
14
15 typedef struct{
16
17     Solution soln;
18     Solution solstar;
19     Solution solnpl;
20     Solution solnml;
21     Vec RHSu, RHSv;
22
23     PetscReal h, dt, dt_old, Re, Bond, time;
24     PetscReal smallval;
25
26     Vec RHS;
27     Mat Jac;
28     PetscInt counter, M, N;
29
30     Vec phi_0, phi, phi_new, H;
31     Vec normalX, normalY;
32     PetscReal Volume0, VolumeNew, Length, dphi;
33
34     double rho_ratio, mu_ratio;
35     double gravity, sigma, R;
36
37 }UserContext;
38
39
40
41 /* User defined functions*/
42 extern PetscErrorCode CreateStructures(DM, UserContext *);
43 extern PetscErrorCode DestroyStructures(UserContext *);
44 extern PetscErrorCode PredictVelocity(DM, UserContext *);
45 extern PetscErrorCode WriteVec(Vec, char const *);
46 extern PetscErrorCode ComputeExactSol(DM, UserContext *);
47 extern PetscErrorCode ComputeRHS(DM, UserContext *);
48 extern PetscErrorCode ComputeMatrix(DM, UserContext *);
49 extern PetscErrorCode WriteMat(Mat, char const *);
50 extern PetscErrorCode ProjectionStep(KSP, UserContext *);
51 extern PetscErrorCode UpdateVelocity(DM, UserContext *);
52 extern PetscErrorCode DefineLevelSet(DM, UserContext *);
53 extern PetscErrorCode Reinitialize(DM, UserContext *);
54 extern PetscReal sign(PetscReal);
55 extern PetscErrorCode DefineVariables(DM, UserContext *);
56 extern PetscErrorCode AdvectInterface(DM, UserContext *);
57 extern PetscErrorCode WriteOutput(UserContext *);
58 extern PetscErrorCode ComputeTimeStep(UserContext *);
59 extern PetscErrorCode ConserveMass(DM, UserContext *);
60 extern PetscErrorCode Screenshot(UserContext *, Engine *);
61
62
63
64 #undef __FUNCT__
65 #define __FUNCT__ "main"
66 int main(int argc, char **args){
67
68     KSP ksp;
69     DM da;
70     UserContext user;
71     PetscErrorCode ierr;
72
73     /* start matlab engine */
74     //Engine *ep = engOpen(NULL);
75
76     /* Define parameters */
77     user.R = 1.0; // Radius of bubble
78     user.M = 80; user.N = user.M; // grid size
79     user.h = 7*user.R/(user.M-1);
80     user.rho_ratio = 1.0/40.0; // viscosity ratio
81     user.mu_ratio = 1.0/40.0; // density ratio
82     user.Re = 100.0; // reynolds number
83     user.Bond = 200.0; // bond number
84     user.smallval = user.h*user.h;
85     int itmax = 2000; // max iterations
86     user.counter = 0;
87     user.time = 0.0;
88
89     ierr = PetscInitialize(&argc,&args,(char*)0,help);CHKERRQ(ierr);
90     ierr = DMDACreate2d(PETSC_COMM_WORLD,DM_BOUNDARY_NONE,DM_BOUNDARY_NONE, DMDA_STENCIL_BOX, user.M,user.M,
91         PETSC_DECIDE,PETSC_DECIDE,1,1,NULL,NULL,&da); CHKERRQ(ierr);
92
93     ierr = KSPCreate(PETSC_COMM_WORLD,&ksp); CHKERRQ(ierr);
94     ierr = CreateStructures(da,&user); CHKERRQ(ierr);
95
96     PetscPrintf(PETSC_COMM_WORLD,"Reynolds no.= %g Bond no.= %g\n",
97         user.Re, user.Bond);
98
99     ierr = DefineLevelSet(da,&user); CHKERRQ(ierr);
```

```

100 ierr = Reinitialize(da,&user); CHKERRQ(ierr);
101
102 for(user.counter = 0; user.counter < itmax; user.counter++){
103
104     PetscPrintf(PETSC_COMM_WORLD, "\n\nIteration: %d\n", user.counter+1);
105
106     ierr = ComputeTimeStep(&user); CHKERRQ(ierr);
107     ierr = AdvectInterface(da,&user); CHKERRQ(ierr);
108     ierr = Reinitialize(da,&user); CHKERRQ(ierr);
109     ierr = DefineVariables(da,&user); CHKERRQ(ierr);
110     ierr = ConserveMass(da,&user); CHKERRQ(ierr);
111     ierr = PredictVelocity(da,&user); CHKERRQ(ierr);
112     ierr = ComputeRHS(da,&user); CHKERRQ(ierr);
113     ierr = ComputeMatrix(da,&user); CHKERRQ(ierr);
114     ierr = Projection_Step(ksp,&user); CHKERRQ(ierr);
115     ierr = UpdateVelocity(da,&user); CHKERRQ(ierr);
116     //ierr = Screenshot(&user,ep); CHKERRQ(ierr);
117
118     if(user.time >= 4.0){
119         break;
120     }
121 }
122
123 ierr = WriteOutput(&user); CHKERRQ(ierr);
124
125 ierr = DestroyStructures(&user); CHKERRQ(ierr);
126 ierr = KSPDestroy(&ksp); CHKERRQ(ierr);
127 //ierr = DMDestroy(&da); CHKERRQ(ierr);
128
129 ierr = PetscFinalize();
130
131 return(0);
132 }
133
134
135
136
137
138 /* Create vectors */
139 #undef __FUNCT__
140 #define __FUNCT__ "CreateStructures"
141 PetscErrorCode CreateStructures(DM da, UserContext *user){
142
143     PetscErrorCode ierr;
144
145     ierr = DMCreateGlobalVector(da,&user->soln.U); CHKERRQ(ierr);
146     ierr = DMCreateGlobalVector(da,&user->soln.V); CHKERRQ(ierr);
147     ierr = DMCreateGlobalVector(da,&user->soln.P); CHKERRQ(ierr);
148     ierr = DMCreateGlobalVector(da,&user->soln.rho); CHKERRQ(ierr);
149     ierr = DMCreateGlobalVector(da,&user->soln.mu); CHKERRQ(ierr);
150     ierr = DMCreateGlobalVector(da,&user->solstar.U); CHKERRQ(ierr);
151     ierr = DMCreateGlobalVector(da,&user->solstar.V); CHKERRQ(ierr);
152     ierr = DMCreateGlobalVector(da,&user->solstar.P); CHKERRQ(ierr);
153     ierr = DMCreateGlobalVector(da,&user->solnpl.U); CHKERRQ(ierr);
154     ierr = DMCreateGlobalVector(da,&user->solnpl.V); CHKERRQ(ierr);
155     ierr = DMCreateGlobalVector(da,&user->solnpl.P); CHKERRQ(ierr);
156     ierr = DMCreateGlobalVector(da,&user->solnpl.rho); CHKERRQ(ierr);
157     ierr = DMCreateGlobalVector(da,&user->solnpl.mu); CHKERRQ(ierr);
158     ierr = DMCreateGlobalVector(da,&user->RHS); CHKERRQ(ierr);
159     ierr = DMCreateGlobalVector(da,&user->solnm1.U); CHKERRQ(ierr);
160     ierr = DMCreateGlobalVector(da,&user->solnm1.V); CHKERRQ(ierr);
161     ierr = DMCreateGlobalVector(da,&user->phi); CHKERRQ(ierr);
162     ierr = DMCreateGlobalVector(da,&user->phi_0); CHKERRQ(ierr);
163     ierr = DMCreateGlobalVector(da,&user->phi_new); CHKERRQ(ierr);
164     ierr = DMCreateGlobalVector(da,&user->H); CHKERRQ(ierr);
165     ierr = DMCreateGlobalVector(da,&user->normalX); CHKERRQ(ierr);
166     ierr = DMCreateGlobalVector(da,&user->normalY); CHKERRQ(ierr);
167     ierr = DMCreateGlobalVector(da,&user->RHSu); CHKERRQ(ierr);
168     ierr = DMCreateGlobalVector(da,&user->RHSv); CHKERRQ(ierr);
169
170     ierr = DMCreateMatrix(da,&user->Jac); CHKERRQ(ierr);
171     return(0);
172 }
173
174
175
176 /* Destroy vectors */
177 #undef __FUNCT__
178 #define __FUNCT__ "DestroyStructures"
179 PetscErrorCode DestroyStructures(UserContext *user){
180
181     PetscErrorCode ierr;
182
183     ierr = VecDestroy(&user->soln.U); CHKERRQ(ierr);
184     ierr = VecDestroy(&user->soln.V); CHKERRQ(ierr);
185     ierr = VecDestroy(&user->soln.P); CHKERRQ(ierr);
186     ierr = VecDestroy(&user->soln.rho); CHKERRQ(ierr);
187     ierr = VecDestroy(&user->soln.mu); CHKERRQ(ierr);
188     ierr = VecDestroy(&user->solstar.U); CHKERRQ(ierr);
189     ierr = VecDestroy(&user->solstar.V); CHKERRQ(ierr);
190     ierr = VecDestroy(&user->solstar.P); CHKERRQ(ierr);
191     ierr = VecDestroy(&user->solnpl.U); CHKERRQ(ierr);
192     ierr = VecDestroy(&user->solnpl.V); CHKERRQ(ierr);
193     ierr = VecDestroy(&user->solnpl.P); CHKERRQ(ierr);
194     ierr = VecDestroy(&user->solnpl.rho); CHKERRQ(ierr);
195     ierr = VecDestroy(&user->solnpl.mu); CHKERRQ(ierr);
196     ierr = VecDestroy(&user->RHS); CHKERRQ(ierr);
197     ierr = VecDestroy(&user->solnm1.U); CHKERRQ(ierr);
198     ierr = VecDestroy(&user->solnm1.V); CHKERRQ(ierr);
199     ierr = VecDestroy(&user->phi); CHKERRQ(ierr);
200     ierr = VecDestroy(&user->phi_0); CHKERRQ(ierr);
201     ierr = VecDestroy(&user->phi_new); CHKERRQ(ierr);
202     ierr = VecDestroy(&user->H); CHKERRQ(ierr);

```

```

203 ierr = VecDestroy(&user->normalX); CHKERRQ(ierr);
204 ierr = VecDestroy(&user->normalY); CHKERRQ(ierr);
205 ierr = VecDestroy(&user->RHSu); CHKERRQ(ierr);
206 ierr = VecDestroy(&user->RHSv); CHKERRQ(ierr);
207
208 ierr = MatDestroy(&user->Jac); CHKERRQ(ierr);
209 return(0);
210 }
211
212
213
214
215 #undef __FUNCT__
216 #define __FUNCT__ "WriteVec"
217 PetscErrorCode WriteVec(Vec vec, char const *name){
218     PetscViewer viewer;
219     PetscErrorCode ierr;
220
221     /* Create "Output" directory */
222     struct stat st = {0};
223     if(stat("Output",&st) == -1)
224         mkdir("Output",0777);
225
226     char filename[64] = "Output/Vec_"; char pfix[12] = ".m";
227     strcat(filename,name); strcat(filename,pfix);
228     ierr = PetscObjectSetName((PetscObject)vec,name); CHKERRQ(ierr);
229     ierr = PetscViewerASCIIOpen(PETSC_COMM_WORLD, filename, &viewer); CHKERRQ(ierr);
230     ierr = PetscViewerSetFormat(viewer, PETSC_VIEWER_ASCII_MATLAB); CHKERRQ(ierr);
231     ierr = VecView(vec,viewer); CHKERRQ(ierr);
232     ierr = PetscViewerDestroy(&viewer); CHKERRQ(ierr);
233     PetscFunctionReturn(0);
234 }
235
236
237
238
239 #undef __FUNCT__
240 #define __FUNCT__ "PredictVelocity"
241 extern PetscErrorCode PredictVelocity(DM da, UserContext *user){
242
243     PetscErrorCode ierr;
244     PetscReal *_Un, *_Vn, *_Pn, *_Ustr, *_Vstr;
245     PetscReal *_Unml, *_Vnml, *_Pstr, *_mu, *_rho;
246     PetscReal *_N1, *_N2, *_phi, *_H, *_RHSu, *_RHSv;
247     PetscReal h,dt, dt_old;
248     PetscInt i,j,xs,ys,xm,ym;
249     PetscReal ux, uy, vx, vy;
250     PetscReal convect, viscous, curvature, delta, surftension;
251     double pi = 3.1415926535, eps = 1.5*user->h;
252     PetscReal GV1,GV2;
253     PetscReal phi_x, phi_y, phi_xx, phi_yy, phi_xy;
254     PetscReal mu1, mu2,mu3,mu4;
255
256     h = user->h; dt = user->dt; dt_old = user->dt_old;
257
258     ierr = DMDAVecGetArray(da,user->soln.U,&_Un); CHKERRQ(ierr);
259     ierr = DMDAVecGetArray(da,user->soln.V,&_Vn); CHKERRQ(ierr);
260     ierr = DMDAVecGetArray(da,user->soln.P,&_Pn); CHKERRQ(ierr);
261     ierr = DMDAVecGetArray(da,user->solstar.U,&_Ustr); CHKERRQ(ierr);
262     ierr = DMDAVecGetArray(da,user->solstar.V,&_Vstr); CHKERRQ(ierr);
263     ierr = DMDAVecGetArray(da,user->solnml.U,&_Unml); CHKERRQ(ierr);
264     ierr = DMDAVecGetArray(da,user->solnml.V,&_Vnml); CHKERRQ(ierr);
265     ierr = DMDAVecGetArray(da,user->solstar.P,&_Pstr); CHKERRQ(ierr);
266     ierr = DMDAVecGetArray(da,user->soln.mu,&_mu); CHKERRQ(ierr);
267     ierr = DMDAVecGetArray(da,user->soln.rho,&_rho); CHKERRQ(ierr);
268     ierr = DMDAVecGetArray(da,user->normalX,&_N1); CHKERRQ(ierr);
269     ierr = DMDAVecGetArray(da,user->normalY,&_N2); CHKERRQ(ierr);
270     ierr = DMDAVecGetArray(da,user->phi,&_phi); CHKERRQ(ierr);
271     ierr = DMDAVecGetArray(da,user->H,&_H); CHKERRQ(ierr);
272     ierr = DMDAVecGetArray(da,user->RHSu,&_RHSu); CHKERRQ(ierr);
273     ierr = DMDAVecGetArray(da,user->RHSv,&_RHSv); CHKERRQ(ierr);
274
275     ierr = DMDAGetCorners(da,&xs,&ys,NULL,&xm,&ym,NULL); CHKERRQ(ierr);
276
277     for (j=ys; j<ys+ym; j++) {
278         for (i=xs; i<xs+xm; i++){
279             if(i > 0 && i < user->M-1 && j > 0 && j < user->M-1){
280
281                 // u direction
282
283                 /* convective term */
284                 if(user->counter == 0){
285                     /* central difference */
286                     ux = 0.5*(-_Un[j][i+1] - _Un[j][i-1])/h;
287                     uy = 0.5*(-_Un[j+1][i] - _Un[j-1][i])/h;
288                     convect = _Un[j][i]*ux + _Vn[j][i]*uy;
289                 }else if(user->counter > 0){
290                     /* Adam Bashford */
291
292                     convect = 0.5*((dt + 2*dt_old)/dt_old)*(_Un[j][i]*(0.5*(-_Un[j][i+1]-_Un[j][i-1])/h)
293                         + _Vn[j][i]*(0.5*(-_Un[j+1][i] - _Un[j-1][i])/h))
294                         -(dt/dt_old)*(_Unml[j][i]*(0.5*(-_Unml[j][i+1] - _Unml[j][i-1])/h) + _Vnml[j][i]
295                         *(0.5*(-_Unml[j+1][i] - _Unml[j-1][i])/h));
296                 }
297                 mu1 = 0.5*(-_mu[j][i+1] + _mu[j][i]);
298                 mu2 = 0.5*(-_mu[j][i-1] + _mu[j][i]);
299                 mu3 = 0.5*(-_mu[j+1][i] + _mu[j][i]);
300                 mu4 = 0.5*(-_mu[j-1][i] + _mu[j][i]);
301
302                 viscous = (2*mu1*(-_Un[j][i+1] - _Un[j][i])/h) - 2*mu2*(-_Un[j][i] - _Un[j][i-1])/h)/h
303                     + (mu3*(-_Un[j+1][i] - _Un[j][i])/h) - mu4*(-_Un[j][i] - _Un[j-1][i])/h)/h
304                     + (mu3*(-_Vn[j+1][i+1] - _Vn[j+1][i-1] + _Vn[j][i+1] - _Vn[j][i-1])/(4*h))
305                     - mu4*(-_Vn[j][i+1] - _Vn[j][i-1] + _Vn[j-1][i+1] - _Vn[j-1][i-1])/(4*h))/h;

```

```

306 viscous = viscous/(user->Re*_rho[j][i]);
307
308
309 /* surface tension term */
310 phi_x = (-phi[j][i+1] - phi[j][i-1])/(2*h);
311 phi_y = (-phi[j+1][i] - phi[j-1][i])/(2*h);
312 phi_xx = (-phi[j][i+1] + phi[j][i-1] - 2*phi[j][i])/(h*h);
313 phi_yy = (-phi[j+1][i] + phi[j-1][i] - 2*phi[j][i])/(h*h);
314 phi_xy = (-phi[j+1][i+1] - phi[j-1][i+1] - phi[j+1][i-1] + phi[j-1][i-1])/(4*h*h);
315
316 GV1 = (-phi[j][i+1] - phi[j][i-1])/(2*h);
317 GV2 = (-phi[j+1][i] - phi[j-1][i])/(2*h);
318
319 curvature = (phi_xx*(phi_y*phi_y) + phi_yy*(phi_x*phi_x) - 2*phi_xy*phi_x*phi_y)
320 / pow((phi_x*phi_x + phi_y*phi_y + user->smallval),1.5);
321
322 // limit curvature
323 if (curvature < -1.0/h){
324     curvature = -1.0/h;
325 }else if (curvature > 1.0/h){
326     curvature = 1.0/h;
327 }
328
329 // delta function
330 if (fabs(phi[j][i]) <= 1.5*h){
331     delta = 0.5*(1 + cos(pi*phi[j][i]/eps))/eps;
332 }else if (fabs(phi[j][i]) > 1.5*h){
333     delta = 0.0;
334 }
335
336 surftension = curvature*delta*GV1/(user->Bond*_rho[j][i]);
337
338
339 /* compute U star */
340
341 if (user->counter == 0){
342     /* Backward Euler */
343     _Ustr[j][i] = _Un[j][i] + dt*(-convect + viscous - surftension);
344     _RHSu[j][i] = -convect + viscous - surftension;
345 }
346 else if (user->counter > 0){
347     /* Adam Bashford */
348     _Ustr[j][i] = _Un[j][i] + dt*(0.5*(((dt + 2*dt_old)/dt_old)*(-convect + viscous - surftension)
349     - (dt/dt_old)*_RHSu[j][i]));
350     _RHSu[j][i] = -convect + viscous - surftension;
351 }
352
353 // v direction
354
355 /* convective term */
356 if (user->counter == 0){
357     // central difference
358     vx = 0.5*(-Vn[j][i+1] - Vn[j][i-1])/h;
359     vy = 0.5*(-Vn[j+1][i] - Vn[j-1][i])/h;
360     convect = _Un[j][i]*vx + _Vn[j][i]*vy;
361 }
362 else if (user->counter > 0){
363     /* Adam Bashford */
364     convect = 0.5*(((dt + 2*dt_old)/dt_old)*(_Un[j][i]*(0.5*(-Vn[j][i+1] - Vn[j][i-1])/h)
365     + _Vn[j][i]*(0.5*(-Vn[j+1][i] - Vn[j-1][i])/h))
366     - (dt/dt_old)*(_Unml[j][i]*(0.5*(-Vnml[j][i+1] - Vnml[j][i-1])/h)
367     + _Vnml[j][i]*(0.5*(-Vnml[j+1][i] - Vnml[j-1][i])/h)));
368 }
369
370 viscous = (mul*((-Vn[j][i+1] - Vn[j][i])/h) - mu2*((-Vn[j][i] - Vn[j][i-1])/h))/h
371 + (mul*((-Un[j+1][i+1] - Un[j-1][i+1] + Un[j+1][i] - Un[j-1][i])/(4*h))
372 - mu2*((-Un[j+1][i] - Un[j-1][i] + Un[j+1][i-1] - Un[j-1][i-1])/(4*h)))/h
373 + (2*mu3*((-Vn[j+1][i] - Vn[j][i])/h) - 2*mu4*((-Vn[j][i] - Vn[j-1][i])/h))/h;
374
375
376 viscous = viscous/(user->Re*_rho[j][i]);
377
378 /* surface tension term */
379 surftension = curvature*delta*GV2/(user->Bond*_rho[j][i]);
380
381 /* compute V star */
382 if (user->counter == 0){
383     _Vstr[j][i] = -Vn[j][i] + dt*(-convect + viscous - surftension -1);
384     _RHSv[j][i] = -convect + viscous - surftension -1;
385 }
386 else if (user->counter > 0){
387     /* Adam Bashford */
388     _Vstr[j][i] = -Vn[j][i] + dt*(0.5*(((dt + 2*dt_old)/dt_old)*(-convect + viscous - surftension -1)
389     - (dt/dt_old)*_RHSv[j][i]));
390     _RHSv[j][i] = -convect + viscous - surftension -1;
391 }
392
393 // boundary condition
394 }else if (i == 0 || i == user->M-1 || j == 0 || j == user->M-1){
395
396     /* 4 corners */
397     if (i == 0 && j == 0){
398         _Ustr[j][i] = _Un[j][i];
399         _Vstr[j][i] = -Vn[j][i];
400     }else if (i == 0 && j == user->M-1){
401         _Ustr[j][i] = _Un[j][i];
402         _Vstr[j][i] = -Vn[j][i];
403     }else if (i == user->M-1 && j == 0){
404         _Ustr[j][i] = _Un[j][i];
405         _Vstr[j][i] = -Vn[j][i];
406     }else if (i == user->M-1 && j == user->M-1){
407         _Ustr[j][i] = _Un[j][i];
408         _Vstr[j][i] = -Vn[j][i];

```

```

409     }
410
411     /* normal B.C. */
412     if (i == 0 || i == user->M-1){
413         _Ustr[j][i] = _Un[j][i];
414     }
415
416     if (j == 0 || j == user->M-1){
417         _Vstr[j][i] = _Vn[j][i];
418     }
419
420     /* tangent b.c */
421     if (j == 0 || j == user->M-1){
422         if (i != 0 && i != user->M-1){
423             _Ustr[j][i] = _Un[j][i] + dt*( _Pstr[j][i+1] - _Pstr[j][i-1])/(2*h);
424         }
425     }
426
427     if (i == 0 && i == user->M-1){
428         if (j != 0 && j != user->M-1){
429             _Vstr[j][i] = _Vn[j][i] + dt*( _Pstr[j+1][i] - _Pstr[j-1][i])/(2*h);
430         }
431     }
432 }
433 }
434 }
435 }
436 }
437
438 ierr = DMDAVecRestoreArray(da, user->soln.U, &_Un); CHKERRQ(ierr);
439 ierr = DMDAVecRestoreArray(da, user->soln.V, &_Vn); CHKERRQ(ierr);
440 ierr = DMDAVecRestoreArray(da, user->soln.P, &_Pn); CHKERRQ(ierr);
441 ierr = DMDAVecRestoreArray(da, user->solstar.U, &_Ustr); CHKERRQ(ierr);
442 ierr = DMDAVecRestoreArray(da, user->solstar.V, &_Vstr); CHKERRQ(ierr);
443 ierr = DMDAVecRestoreArray(da, user->solnml.U, &_Unml); CHKERRQ(ierr);
444 ierr = DMDAVecRestoreArray(da, user->solnml.V, &_Vnml); CHKERRQ(ierr);
445 ierr = DMDAVecRestoreArray(da, user->solstar.P, &_Pstr); CHKERRQ(ierr);
446 ierr = DMDAVecRestoreArray(da, user->soln.mu, &_mu); CHKERRQ(ierr);
447 ierr = DMDAVecRestoreArray(da, user->soln.rho, &_rho); CHKERRQ(ierr);
448 ierr = DMDAVecRestoreArray(da, user->normalX, &_N1); CHKERRQ(ierr);
449 ierr = DMDAVecRestoreArray(da, user->normalY, &_N2); CHKERRQ(ierr);
450 ierr = DMDAVecRestoreArray(da, user->phi, &_phi); CHKERRQ(ierr);
451 ierr = DMDAVecRestoreArray(da, user->H, &_H); CHKERRQ(ierr);
452 ierr = DMDAVecRestoreArray(da, user->RHSu, &_RHSu); CHKERRQ(ierr);
453 ierr = DMDAVecRestoreArray(da, user->RHSv, &_RHSv); CHKERRQ(ierr);
454
455 return(0);
456 }
457
458
459
460
461 #undef __FUNCT__
462 #define __FUNCT__ "ComputeRHS"
463 PetscErrorCode ComputeRHS(DM da, UserContext *user){
464
465     PetscErrorCode ierr;
466     PetscReal **_Ustr, **_Vstr, **_b;
467     PetscInt i, j, xs, ys, xm, ym;
468     PetscReal dt = user->dt, h = user->h, ux, vy;
469
470     ierr = DMDAVecGetArray(da, user->solstar.U, &_Ustr); CHKERRQ(ierr);
471     ierr = DMDAVecGetArray(da, user->solstar.V, &_Vstr); CHKERRQ(ierr);
472     ierr = DMDAVecGetArray(da, user->RHS, &_b); CHKERRQ(ierr);
473     ierr = DMDAGetCorners(da, &xs, &ys, NULL, &xm, &ym, NULL); CHKERRQ(ierr);
474
475     for (i = xs; i < xs+xm; i++){
476         for (j = ys; j < ys+ym; j++){
477
478             if (i == 0){
479                 ux = 0.5*(-3*_Ustr[j][i] + 4*_Ustr[j][i+1] - _Ustr[j][i+2])/h;
480             } else if (i == user->M-1){
481                 ux = 0.5*(3*_Ustr[j][i] - 4*_Ustr[j][i-1] + _Ustr[j][i-2])/h;
482             } else if (i > 0 && i < user->M-1){
483                 ux = (0.5*(-_Ustr[j][i+1] - _Ustr[j][i-1]))/h;
484                 //ux = (0.5*(-_Ustr[j][i] + _Ustr[j][i+1]) - 0.5*(-_Ustr[j][i] + _Ustr[j][i-1]))/h;
485             }
486
487             if (j == 0){
488                 vy = 0.5*(-3*_Vstr[j][i] + 4*_Vstr[j+1][i] - _Vstr[j+2][i])/h;
489             } else if (j == user->M-1){
490                 vy = 0.5*(3*_Vstr[j][i] - 4*_Vstr[j-1][i] + _Vstr[j-2][i])/h;
491             } else if (j > 0 && j < user->M-1){
492                 vy = (0.5*(-_Vstr[j+1][i] - _Vstr[j-1][i]))/h;
493                 //vy = (0.5*(-_Ustr[j][i] + _Ustr[j+1][i]) - 0.5*(-_Ustr[j][i] + _Ustr[j-1][i]))/h;
494             }
495
496             _b[j][i] = (ux + vy)/dt;
497         }
498     }
499 }
500
501 ierr = VecAssemblyBegin(user->RHS); CHKERRQ(ierr);
502 ierr = VecAssemblyEnd(user->RHS); CHKERRQ(ierr);
503
504 ierr = DMDAVecRestoreArray(da, user->solstar.U, &_Ustr); CHKERRQ(ierr);
505 ierr = DMDAVecRestoreArray(da, user->solstar.V, &_Vstr); CHKERRQ(ierr);
506 ierr = DMDAVecRestoreArray(da, user->RHS, &_b); CHKERRQ(ierr);
507
508 MatNullSpace nullspace;
509 ierr = MatNullSpaceCreate(PETSC.COMM_WORLD, PETSC.TRUE, 0, 0, &nullspace); CHKERRQ(ierr);
510 ierr = MatNullSpaceRemove(nullspace, user->RHS); CHKERRQ(ierr);
511

```

```

512 ierr = MatNullSpaceDestroy(&nullspace); CHKERRQ(ierr);
513
514
515 return(0);
516 }
517
518
519
520 #undef __FUNCT__
521 #define __FUNCT__ "ComputeMatrix"
522 PetscErrorCode ComputeMatrix(DM da, UserContext *user){
523
524   PetscErrorCode ierr;
525   PetscInt i,j,xs,ys,xm,ym;
526   PetscReal v[5];
527   MatStencil row, col[5];
528   PetscReal h = user->h, h2 = h*h;
529   PetscReal a1, a2, a3, a4, a5;
530   PetscReal **rho;
531
532   ierr = DMDAVecGetArray(da, user->soln.rho, &rho);
533   ierr = DMDAGetCorners(da, &xs, &ys, NULL, &xm, &ym, NULL); CHKERRQ(ierr);
534
535   if(user->counter > 0){
536     ierr = MatZeroEntries(user->Jac);
537   }
538
539
540   for (j=ys; j<ys+ym; j++) {
541     for (i=xs; i<xs+xm; i++) {
542
543       row.i = i; row.j = j;
544
545       /* compute coefficients */
546       if(i == 0){
547         a2 = 0;
548         a4 = (2.0/(_rho[j][i] + _rho[j][i+1]))/h2;
549       } else if(i == user->M-1){
550         a2 = (2.0/(_rho[j][i] + _rho[j][i-1]))/h2;
551         a4 = 0;
552       } else if(i != 0 && i != user->M-1){
553         a2 = (2.0/(_rho[j][i] + _rho[j][i-1]))/h2;
554         a4 = (2.0/(_rho[j][i] + _rho[j][i+1]))/h2;
555       }
556
557       if(j == 0){
558         a1 = 0;
559         a5 = (2.0/(_rho[j][i] + _rho[j+1][i]))/h2;
560       } else if(j == user->N-1){
561         a1 = (2.0/(_rho[j][i] + _rho[j-1][i]))/h2;
562         a5 = 0;
563       } else if(j != 0 && j != user->N-1){
564         a1 = (2.0/(_rho[j][i] + _rho[j-1][i]))/h2;
565         a5 = (2.0/(_rho[j][i] + _rho[j+1][i]))/h2;
566       }
567
568       a3 = -(a1 + a2 + a4 + a5);
569
570       /* assemble matrix */
571
572       if(i == 0 && j == 0){
573         v[0] = a3;          col[0].i = i;   col[0].j = j;
574         v[1] = a4;          col[1].i = i+1; col[1].j = j;
575         v[2] = a5;          col[2].i = i;   col[2].j = j+1;
576         ierr = MatSetValuesStencil(user->Jac, 1, &row, 3, col, v, INSERT_VALUES); CHKERRQ(ierr);
577       }
578       else if(i == 0 && j != 0 && j != user->N-1){
579         v[0] = a1;          col[0].i = i;   col[0].j = j-1;
580         v[1] = a3;          col[1].i = i;   col[1].j = j;
581         v[2] = a4;          col[2].i = i+1; col[2].j = j;
582         v[3] = a5;          col[3].i = i;   col[3].j = j+1;
583         ierr = MatSetValuesStencil(user->Jac, 1, &row, 4, col, v, INSERT_VALUES); CHKERRQ(ierr);
584       }
585       else if(i == 0 && j == user->N-1){
586         v[0] = a1;          col[0].i = i;   col[0].j = j-1;
587         v[1] = a3;          col[1].i = i;   col[1].j = j;
588         v[2] = a4;          col[2].i = i+1; col[2].j = j;
589         ierr = MatSetValuesStencil(user->Jac, 1, &row, 3, col, v, INSERT_VALUES); CHKERRQ(ierr);
590       }
591       else if(i != 0 && i != user->M-1 && j == 0){
592         v[0] = a2;          col[0].i = i-1; col[0].j = j;
593         v[1] = a3;          col[1].i = i;   col[1].j = j;
594         v[2] = a4;          col[2].i = i+1; col[2].j = j;
595         v[3] = a5;          col[3].i = i;   col[3].j = j+1;
596         ierr = MatSetValuesStencil(user->Jac, 1, &row, 4, col, v, INSERT_VALUES); CHKERRQ(ierr);
597       }
598       else if(i == user->M-1 && j == 0){
599         v[0] = a2;          col[0].i = i-1; col[0].j = j;
600         v[1] = a3;          col[1].i = i;   col[1].j = j;
601         v[2] = a5;          col[2].i = i;   col[2].j = j+1;
602         ierr = MatSetValuesStencil(user->Jac, 1, &row, 3, col, v, INSERT_VALUES); CHKERRQ(ierr);
603       }
604       else if(i == user->M-1 && j != 0 && j != user->N-1){
605         v[0] = a1;          col[0].i = i;   col[0].j = j-1;
606         v[1] = a2;          col[1].i = i-1; col[1].j = j;
607         v[2] = a3;          col[2].i = i;   col[2].j = j;
608         v[3] = a5;          col[3].i = i;   col[3].j = j+1;
609         ierr = MatSetValuesStencil(user->Jac, 1, &row, 4, col, v, INSERT_VALUES); CHKERRQ(ierr);
610       }
611       else if(i == user->M-1 && j == user->N-1){
612         v[0] = a1;          col[0].i = i;   col[0].j = j-1;
613         v[1] = a2;          col[1].i = i-1; col[1].j = j;
614         v[2] = a3;          col[2].i = i;   col[2].j = j;

```

```

615         ierr = MatSetValuesStencil(user->Jac,1,&row,3,col,v,INSERT_VALUES); CHKERRQ(ierr);
616     }
617     else if(i != 0 && i != user->M-1 && j == user->N-1){
618         v[0] = a1;          col[0].i = i;   col[0].j = j-1;
619         v[1] = a2;          col[1].i = i-1; col[1].j = j;
620         v[2] = a3;          col[2].i = i;   col[2].j = j;
621         v[3] = a4;          col[3].i = i+1; col[3].j = j;
622         ierr = MatSetValuesStencil(user->Jac,1,&row,4,col,v,INSERT_VALUES); CHKERRQ(ierr);
623     }
624     else if(i > 0 && i < user->M-1 && j > 0 && j < user->N-1){
625         v[0] = a1;          col[0].i = i;   col[0].j = j-1;
626         v[1] = a2;          col[1].i = i-1; col[1].j = j;
627         v[2] = a3;          col[2].i = i;   col[2].j = j;
628         v[3] = a4;          col[3].i = i+1; col[3].j = j;
629         v[4] = a5;          col[4].i = i;   col[4].j = j+1;
630         ierr = MatSetValuesStencil(user->Jac,1,&row,5,col,v,INSERT_VALUES); CHKERRQ(ierr);
631     }
632 }
633 }
634 }
635
636 ierr = MatAssemblyBegin(user->Jac,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
637 ierr = MatAssemblyEnd(user->Jac,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
638
639 ierr = MatSetOption(user->Jac,MAT_NEW_NONZERO_LOCATIONS,PETSC_TRUE); CHKERRQ(ierr);
640
641 ierr = DMDAVecRestoreArray(da,user->soln.rho,&.rho); CHKERRQ(ierr);
642
643 MatNullSpace nullspace;
644 ierr = MatNullSpaceCreate(PETSC_COMM_WORLD,PETSC_TRUE,0,0,&nullspace); CHKERRQ(ierr);
645 ierr = MatSetNullSpace(user->Jac,nullspace); CHKERRQ(ierr);
646 ierr = MatNullSpaceDestroy(&nullspace); CHKERRQ(ierr);
647
648 return(0);
649 }
650
651
652
653
654 #undef __FUNCT__
655 #define __FUNCT__ "WriteMat"
656 PetscErrorCode WriteMat(Mat mat,char const *name){
657
658     PetscViewer viewer;
659     PetscErrorCode ierr;
660
661     /* Create "Output" directory */
662     struct stat st = {0};
663     if(stat("Output",&st) == -1)
664         mkdir("Output",0777);
665
666     char filename[64] = "Output/Mat_"; char pfix[12] = ".m";
667     strcat(filename,name); strcat(filename,pfix);
668     ierr = PetscObjectSetName((PetscObject)mat,name); CHKERRQ(ierr);
669     ierr = PetscViewerASCIIOpen(PETSC_COMM_WORLD,filename,&viewer); CHKERRQ(ierr);
670     ierr = PetscViewerSetFormat(viewer,PETSC_VIEWER_ASCII_MATLAB); CHKERRQ(ierr);
671     ierr = MatView(mat,viewer); CHKERRQ(ierr);
672     ierr = PetscViewerDestroy(&viewer); CHKERRQ(ierr);
673     PetscFunctionReturn(0);
674 }
675
676
677
678
679
680 #undef __FUNCT__
681 #define __FUNCT__ "Projection_Step"
682 PetscErrorCode Projection_Step(KSP ksp, UserContext *user){
683
684     PetscErrorCode ierr;
685     MatNullSpace nullsp;
686     PetscInt itn;
687
688     ierr = KSPSetOperators(ksp,user->Jac,user->Jac); CHKERRQ(ierr);
689
690     if (user->counter > 0){
691         ierr = KSPSetInitialGuessNonzero(ksp,PETSC_TRUE);
692     }
693
694
695     ierr = MatNullSpaceCreate(PETSC_COMM_WORLD,PETSC_TRUE,0,NULL,&nullsp); CHKERRQ(ierr);
696     ierr = KSPSetNullSpace(ksp,nullsp); CHKERRQ(ierr);
697     ierr = KSPSetTolerances(ksp,1.e-9,PETSC_DEFAULT,PETSC_DEFAULT,PETSC_DEFAULT); CHKERRQ(ierr);
698
699     ierr = KSPSetFromOptions(ksp); CHKERRQ(ierr);
700     ierr = KSPSetUp(ksp); CHKERRQ(ierr);
701     ierr = KSPSolve(ksp,user->RHS,user->solstar.P); CHKERRQ(ierr);
702     ierr = MatNullSpaceDestroy(&nullsp); CHKERRQ(ierr);
703     ierr = KSPGetIterationNumber(ksp,&itn); CHKERRQ(ierr);
704     ierr = PetscPrintf(PETSC_COMM_WORLD,"KSP: %d\n",itn); CHKERRQ(ierr);
705
706     ierr = MatNullSpaceCreate(PETSC_COMM_WORLD,PETSC_TRUE,0,0,&nullsp); CHKERRQ(ierr);
707     ierr = MatNullSpaceRemove(nullsp,user->solstar.P); CHKERRQ(ierr);
708     ierr = MatNullSpaceDestroy(&nullsp); CHKERRQ(ierr);
709
710     return(0);
711 }
712 }
713
714
715
716
717 #undef __FUNCT__

```



```

718 #define __FUNCT__ "UpdateVelocity"
719 extern PetscErrorCode UpdateVelocity(DM da, UserContext *user){
720
721     PetscErrorCode ierr;
722     PetscReal **_Unp1, **_Vnp1, **_Ustr, **_Vstr, **_Pnp1, **_Pstr, **_Pn, **_Un, **_Vn, **_rho;
723     PetscInt i,j,xs,ys,xm,ym;
724     PetscReal dt = user->dt, h = user->h;
725
726     ierr = DMDAVecGetArray(da,user->solnp1.U,&_Unp1); CHKERRQ(ierr);
727     ierr = DMDAVecGetArray(da,user->solnp1.V,&_Vnp1); CHKERRQ(ierr);
728     ierr = DMDAVecGetArray(da,user->solnp1.P,&_Pnp1); CHKERRQ(ierr);
729     ierr = DMDAVecGetArray(da,user->solstar.U,&_Ustr); CHKERRQ(ierr);
730     ierr = DMDAVecGetArray(da,user->solstar.V,&_Vstr); CHKERRQ(ierr);
731     ierr = DMDAVecGetArray(da,user->solstar.P,&_Pstr); CHKERRQ(ierr);
732     ierr = DMDAVecGetArray(da,user->soln.P,&_Pn); CHKERRQ(ierr);
733     ierr = DMDAVecGetArray(da,user->soln.U,&_Un); CHKERRQ(ierr);
734     ierr = DMDAVecGetArray(da,user->soln.V,&_Vn); CHKERRQ(ierr);
735     ierr = DMDAVecGetArray(da,user->soln.rho,&_rho); CHKERRQ(ierr);
736
737     ierr = DMDAGetCorners(da,&xs,&ys,NULL,&xm,&ym,NULL); CHKERRQ(ierr);
738
739     for (j=ys; j<ys+ym; j++) {
740         for (i=xs; i<xs+xm; i++) {
741             if (i > 0 && i < user->M-1 && j > 0 && j < user->M-1){
742                 _Unp1[j][i] = _Ustr[j][i] - dt*(1.0/_rho[j][i])*((_Pstr[j][i+1] - _Pstr[j][i-1])/(2*h));
743                 _Vnp1[j][i] = _Vstr[j][i] - dt*(1.0/_rho[j][i])*((_Pstr[j+1][i] - _Pstr[j-1][i])/(2*h));
744                 _Pnp1[j][i] = _Pstr[j][i];
745             }
746             else if (i==0 || j==0 || i==user->M-1 || j==user->M-1){
747                 _Unp1[j][i] = _Un[j][i];
748                 _Vnp1[j][i] = _Vn[j][i];
749                 _Pnp1[j][i] = _Pstr[j][i];
750             }
751         }
752     }
753
754     ierr = DMDAVecRestoreArray(da,user->solnp1.U,&_Unp1); CHKERRQ(ierr);
755     ierr = DMDAVecRestoreArray(da,user->solnp1.V,&_Vnp1); CHKERRQ(ierr);
756     ierr = DMDAVecRestoreArray(da,user->solnp1.P,&_Pnp1); CHKERRQ(ierr);
757     ierr = DMDAVecRestoreArray(da,user->solstar.U,&_Ustr); CHKERRQ(ierr);
758     ierr = DMDAVecRestoreArray(da,user->solstar.V,&_Vstr); CHKERRQ(ierr);
759     ierr = DMDAVecRestoreArray(da,user->solstar.P,&_Pstr); CHKERRQ(ierr);
760     ierr = DMDAVecRestoreArray(da,user->soln.P,&_Pn); CHKERRQ(ierr);
761     ierr = DMDAVecRestoreArray(da,user->soln.U,&_Un); CHKERRQ(ierr);
762     ierr = DMDAVecRestoreArray(da,user->soln.V,&_Vn); CHKERRQ(ierr);
763     ierr = DMDAVecRestoreArray(da,user->soln.rho,&_rho); CHKERRQ(ierr);
764
765     ierr = VecCopy(user->soln.U,user->solnm1.U); CHKERRQ(ierr);
766     ierr = VecCopy(user->soln.V,user->solnm1.V); CHKERRQ(ierr);
767
768     ierr = VecCopy(user->solnp1.U,user->soln.U); CHKERRQ(ierr);
769     ierr = VecCopy(user->solnp1.V,user->soln.V); CHKERRQ(ierr);
770     ierr = VecCopy(user->solnp1.P,user->soln.P); CHKERRQ(ierr);
771
772     return(0);
773 }
774
775 #undef __FUNCT__
776 #define __FUNCT__ "DefineLevelSet"
777 PetscErrorCode DefineLevelSet(DM da, UserContext *user){
778
779     PetscErrorCode ierr;
780     PetscInt i,j,xs,ys,xm,ym;
781     DM cda;
782     Vec xy;
783     DMDACoor2d **_xy;
784     PetscReal **_phi, R = user->R;
785     PetscReal xmin, xmax, ymin, ymax, xbubble, ybubble;
786
787     xmin = 0.0; xmax = 7*R; ymin = 0.0; ymax = 7*R;
788     xbubble = xmin + fabs(xmax-xmin)/2; ybubble = ymin + fabs(xmax-xmin)/4;
789
790     ierr = DMDASetUniformCoordinates(da,xmin,xmax,ymin,ymax,0.0,1.0); CHKERRQ(ierr);
791
792     /* Get coordinates */
793     ierr = DMGetCoordinateDM(da,&cda); CHKERRQ(ierr);
794     ierr = DMGetCoordinatesLocal(da,&xy); CHKERRQ(ierr);
795     ierr = DMDAVecGetArray(cda,xy,&_xy); CHKERRQ(ierr);
796
797     /* Get Vec arrays */
798     ierr = DMDAVecGetArray(da,user->phi.0,&_phi); CHKERRQ(ierr);
799
800     ierr = DMDAGetCorners(da,&xs,&ys,NULL,&xm,&ym,NULL); CHKERRQ(ierr);
801
802     /* Compute exact solution */
803     for (i = xs; i < xs+xm; i++){
804         for (j = ys; j < ys+ym; j++){
805             _phi[j][i] = pow((-_xy[j][i].x - xbubble),2) + pow((-_xy[j][i].y - ybubble),2) - R*R;
806         }
807     }
808
809     /* Restore Vec arrays */
810     ierr = DMDAVecRestoreArray(da,user->phi.0,&_phi); CHKERRQ(ierr);

```



```

821 |
822 |     ierr = VecDestroy(&xy); CHKERRQ(ierr);
823 |     ierr = DMDestroy(&cda); CHKERRQ(ierr);
824 |
825 |     return(0);
826 | }
827 |
828 |
829 |
830 |
831 |
832 |
833 | #undef __FUNCT__
834 | #define __FUNCT__ "Reinitialize"
835 | PetscErrorCode Reinitialize(DM da, UserContext *user){
836 |
837 |     PetscErrorCode ierr;
838 |     PetscReal **_phi_0, **_phi, **_phi_new, **_Un, **_Vn;
839 |     PetscInt i,j,xs,ys,xm,ym;
840 |     PetscReal s, a, b, c, d, a_plus, a_minus, b_plus, b_minus, c_plus,
841 |         c_minus, d_plus, d_minus, s_plus, s_minus, D, phi_x, phi_y;
842 |
843 |     PetscReal phix, phiy, eps = user->h*user->h;
844 |
845 |     PetscReal h = user->h, norm;
846 |
847 |     int t, itmax = 5*user->M;           // max iterations;
848 |     PetscReal dt = 0.5*h;              // time step size(dTau)
849 |
850 |     if(user->counter == 0){
851 |         itmax = 5*user->M;              // max iterations
852 |     } else if(user->counter > 0){
853 |         itmax = 50;                    // max iterations
854 |     }
855 |
856 |
857 |     ierr = VecCopy(user->phi_0, user->phi); CHKERRQ(ierr);
858 |
859 |
860 |     ierr = DMDAVecGetArray(da, user->phi_0, &_phi_0); CHKERRQ(ierr);
861 |     ierr = DMDAVecGetArray(da, user->phi, &_phi); CHKERRQ(ierr);
862 |     ierr = DMDAVecGetArray(da, user->phi_new, &_phi_new); CHKERRQ(ierr);
863 |     ierr = DMDAVecGetArray(da, user->soln.U, &_Un); CHKERRQ(ierr);
864 |     ierr = DMDAVecGetArray(da, user->soln.V, &_Vn); CHKERRQ(ierr);
865 |
866 |     ierr = DM DAGetCorners(da, &xs, &ys, NULL, &xm, &ym, NULL); CHKERRQ(ierr);
867 |
868 |     for(t = 0; t < itmax; t++){
869 |
870 |         for (j=ys; j<ys+ym; j++) {
871 |             for (i=xs; i<xs+xm; i++) {
872 |
873 |                 /* Compute a, b, c, d */
874 |                 if (i == 0){ // B.C.on a using Ghost Node
875 |                     a = (-phi[j][i + 1] - _phi[j][i]) / h;
876 |                 } else{
877 |                     a = (-phi[j][i] - _phi[j][i - 1]) / h;
878 |                 }
879 |
880 |                 if (i == user->M - 1){ // B.C.on b using Ghost Node
881 |                     b = (-phi[j][i] - _phi[j][i - 1]) / h;
882 |                 } else{
883 |                     b = (-phi[j][i + 1] - _phi[j][i]) / h;
884 |                 }
885 |
886 |                 if (j == 0){ //B.C.on c using Ghost Node
887 |                     c = (-phi[j + 1][i] - _phi[j][i]) / h;
888 |                 } else{
889 |                     c = (-phi[j][i] - _phi[j - 1][i]) / h;
890 |                 }
891 |                 if (j == user->N - 1){ //B.C.on d using Ghost Node
892 |                     d = (-phi[j][i] - _phi[j - 1][i]) / h;
893 |                 } else{
894 |                     d = (-phi[j + 1][i] - _phi[j][i]) / h;
895 |                 }
896 |
897 |                 // Compute phi_x
898 |                 if (i == 0){
899 |                     phi_x = (-phi[j][i + 1] - _phi[j][i]) / h;
900 |                 }
901 |                 else if (i == user->N - 1){
902 |                     phi_x = (-phi[j][i] - _phi[j][i - 1]) / h;
903 |                 }
904 |                 else{ phi_x = (-phi[j][i + 1] - _phi[j][i - 1]) / (2 * h); }
905 |
906 |                 // Compute phi_y
907 |                 if (j == 0){
908 |                     phi_y = (-phi[j + 1][i] - _phi[j][i]) / h;
909 |                 }
910 |                 else if (j == user->N - 1){
911 |                     phi_y = (-phi[j][i] - _phi[j - 1][i]) / h;
912 |                 }
913 |                 else{ phi_y = (-phi[j + 1][i] - _phi[j - 1][i]) / (2 * h); }
914 |
915 |                 /* mollified sign function */
916 |                 s = -phi[j][i] / (sqrt(pow(_phi[j][i], 2) + (pow(phi_x, 2) + pow(phi_y, 2))*pow(h, 2)));
917 |
918 |                 /* Compute positive and negative contribution */
919 |                 a_plus = fmax(a, 0.0);
920 |                 a_minus = fmin(a, 0.0);
921 |                 b_plus = fmax(b, 0.0);
922 |                 b_minus = fmin(b, 0.0);
923 |                 c_plus = fmax(c, 0.0);

```

```

924         c_minus = fmin(c, 0.0);
925         d_plus = fmax(d, 0.0);
926         d_minus = fmin(d, 0.0);
927         s_plus = fmax(s, 0.0);
928         s_minus = fmin(s, 0.0);
929
930
931         int flag = 0;
932         /* if interface exists -- Use Subcell Fix by Russo and Smereka */
933         if (i > 0 && i < user->M-1 && j > 0 && j < user->N-1){
934             if (_phi[j][i]*_phi[j][i+1] < 0 || _phi[j][i]*_phi[j][i-1] < 0
935                 || _phi[j][i]*_phi[j+1][i] < 0 || _phi[j][i]*_phi[j-1][i] < 0){
936
937                 phix = ((_phi_0[j][i] + _phi_0[j][i+1])/2 - (_phi_0[j][i] + _phi_0[j][i-1])/2)/h;
938                 phiy = ((_phi_0[j][i] + _phi_0[j+1][i])/2 - (_phi_0[j][i] + _phi_0[j-1][i])/2)/h;
939
940                 D = _phi_0[j][i] / sqrt(pow(phix,2) + pow(phiy,2) + eps);
941                 // sub-cell fix
942                 _phi_new[j][i] = _phi[j][i] - (dt / h)*(sign(_phi_0[j][i])*fabs(_phi[j][i]) - D);
943                 flag = 1;
944             }
945         }
946     }
947
948     if(flag == 0){
949         _phi_new[j][i] = _phi[j][i] - dt*(s_plus*(pow((fmax(pow(a_plus, 2), pow(b_minus, 2))),
950             + (fmax(pow(c_plus, 2), pow(d_minus, 2))), 0.5) - 1)
951             + s_minus*(pow((fmax(pow(a_minus, 2), pow(b_plus, 2))),
952             + (fmax(pow(c_minus, 2), pow(d_plus, 2))), 0.5) - 1));
953     }
954 }
955
956 }
957
958 ierr = VecXPY(user->phi, -1.0, user->phi_new); CHKERRQ(ierr);
959 ierr = VecNorm(user->phi, NORM_2, &norm);
960 if(norm <= 0.000001){
961     ierr = VecCopy(user->phi_new, user->phi); CHKERRQ(ierr);
962     break;
963 }
964 ierr = VecCopy(user->phi_new, user->phi); CHKERRQ(ierr);
965
966 }
967
968 PetscPrintf(PETSC_COMM_WORLD, "SDF: %d and norm = %e\n", t, norm);
969
970
971 ierr = DMDataVecRestoreArray(da, user->phi_0, &_phi_0); CHKERRQ(ierr);
972 ierr = DMDataVecRestoreArray(da, user->phi, &_phi); CHKERRQ(ierr);
973 ierr = DMDataVecRestoreArray(da, user->phi_new, &_phi_new); CHKERRQ(ierr);
974 ierr = DMDataVecRestoreArray(da, user->soln.U, &_Un); CHKERRQ(ierr);
975 ierr = DMDataVecRestoreArray(da, user->soln.V, &_Vn); CHKERRQ(ierr);
976
977 return(0);
978 }
979
980
981
982
983
984 #undef __FUNCT__
985 #define __FUNCT__ "sign"
986 PetscReal sign(PetscReal sgn){
987     if (sgn < 0){ return (-1.0); }
988     else if (sgn > 0){ return (1.0); }
989     else return (0.0);
990 }
991
992
993
994
995 #undef __FUNCT__
996 #define __FUNCT__ "DefineVariables"
997 PetscErrorCode DefineVariables(DM da, UserContext *user){
998
999     PetscErrorCode ierr;
1000     PetscInt i, j, xs, ys, xm, ym;
1001     PetscReal **_H, **_phi, **_Mu, **_rho;
1002     PetscReal eps = 1.5*user->h;
1003     double pi = 3.1415926535, rho_ratio = user->rho_ratio, mu_ratio = user->mu_ratio;
1004
1005     ierr = DMDataVecGetArray(da, user->soln.mu, &_Mu); CHKERRQ(ierr);
1006     ierr = DMDataVecGetArray(da, user->soln.rho, &_rho); CHKERRQ(ierr);
1007     ierr = DMDataVecGetArray(da, user->phi, &_phi); CHKERRQ(ierr);
1008     ierr = DMDataVecGetArray(da, user->H, &_H); CHKERRQ(ierr);
1009
1010     ierr = DMDataGetCorners(da, &xs, &ys, NULL, &xm, &ym, NULL); CHKERRQ(ierr);
1011
1012     for (j=ys; j<ys+ym; j++){
1013         for (i=xs; i<xs+xm; i++){
1014
1015             /* compute heaviside */
1016             if (_phi[j][i] < -eps)
1017                 _H[j][i] = 0;
1018             else if (fabs(_phi[j][i]) <= eps)
1019                 _H[j][i] = 0.5*(1.0 + _phi[j][i]/eps + sin(pi*_phi[j][i]/eps)/pi);
1020             else if (_phi[j][i] > eps)
1021                 _H[j][i] = 1;
1022
1023             /* Define Viscosity */
1024             _Mu[j][i] = _H[j][i] + (mu_ratio)*(1.0 - _H[j][i]);
1025
1026             /* Define Density */

```

```

1027         _rho[j][i] = _H[j][i] + (rho_ratio)*(1.0 - _H[j][i]);
1028     }
1029 }
1030
1031 ierr = DMDataVecRestoreArray(da, user->soln.mu, &_mu); CHKERRQ(ierr);
1032 ierr = DMDataVecRestoreArray(da, user->soln.rho, &_rho); CHKERRQ(ierr);
1033 ierr = DMDataVecRestoreArray(da, user->phi, &_phi); CHKERRQ(ierr);
1034 ierr = DMDataVecRestoreArray(da, user->H, &_H); CHKERRQ(ierr);
1035
1036 return(0);
1037 }
1038
1039 #undef __FUNCT__
1040 #define __FUNCT__ "AdvectionInterface"
1041 PetscErrorCode AdvectionInterface(DM da, UserContext *user){
1042
1043     PetscErrorCode ierr;
1044     PetscReal **_phi_new, **_phi, **_U, **_V;
1045     PetscReal phix, phiy;
1046     PetscReal h = user->h, dt = user->dt;
1047     PetscInt i, j, xs, ys, xm, ym;
1048
1049     ierr = DMDataVecGetArray(da, user->phi_new, &_phi_new); CHKERRQ(ierr);
1050     ierr = DMDataVecGetArray(da, user->phi, &_phi); CHKERRQ(ierr);
1051     ierr = DMDataVecGetArray(da, user->soln.U, &_U); CHKERRQ(ierr);
1052     ierr = DMDataVecGetArray(da, user->soln.V, &_V); CHKERRQ(ierr);
1053
1054     ierr = DMDataGetCorners(da, &xs, &ys, NULL, &xm, &ym, NULL); CHKERRQ(ierr);
1055
1056     for (j=ys; j<ys+ym; j++){
1057         for (i=xs; i<xs+xm; i++){
1058             if (i == 0){
1059                 phix = (-_phi[j][i+1] - _phi[j][i]) / h;
1060             } else if (i == user->M-1){
1061                 phix = (-_phi[j][i] - _phi[j][i-1]) / h;
1062             } else if (i != 0 && i != user->M-1){
1063                 if (_U[j][i] > 0){
1064                     phix = (-_phi[j][i] - _phi[j][i-1])/h;
1065                 } else if (_U[j][i] < 0){
1066                     phix = (-_phi[j][i+1] - _phi[j][i])/h;
1067                 } else if (_U[j][i] == 0){
1068                     phix = 0.0;
1069                 }
1070             }
1071
1072             if (j == 0){
1073                 phiy = (-_phi[j+1][i] - _phi[j][i]) / h;
1074             } else if (j == user->N-1){
1075                 phiy = (-_phi[j][i] - _phi[j-1][i]) / h;
1076             } else if (j != 0 && j != user->N-1){
1077                 if (_V[j][i] > 0){
1078                     phiy = (-_phi[j][i] - _phi[j-1][i])/h;
1079                 } else if (_V[j][i] < 0){
1080                     phiy = (-_phi[j+1][i] - _phi[j][i])/h;
1081                 } else if (_V[j][i] == 0){
1082                     phiy = 0.0;
1083                 }
1084             }
1085
1086             _phi_new[j][i] = _phi[j][i] - dt*(_U[j][i]*phix + _V[j][i]*phiy);
1087         }
1088     }
1089
1090     ierr = DMDataVecRestoreArray(da, user->phi_new, &_phi_new); CHKERRQ(ierr);
1091     ierr = DMDataVecRestoreArray(da, user->phi, &_phi); CHKERRQ(ierr);
1092     ierr = DMDataVecRestoreArray(da, user->soln.U, &_U); CHKERRQ(ierr);
1093     ierr = DMDataVecRestoreArray(da, user->soln.V, &_V); CHKERRQ(ierr);
1094
1095     ierr = VecCopy(user->phi_new, user->phi_0); CHKERRQ(ierr);
1096
1097     return(0);
1098 }
1099
1100 #undef __FUNCT__
1101 #define __FUNCT__ "ComputeTimeStep"
1102 extern PetscErrorCode ComputeTimeStep(UserContext *user){
1103
1104     PetscErrorCode ierr;
1105     PetscReal dts, dtv, dtv2, dtc, dtc1, dtc2, dtf, dtf1, dtf2;
1106     PetscReal rhoWater = 1.0, rhoAir = 0.001226;
1107     PetscReal muWater = 0.01137, muAir = 0.000178;
1108     double pi = 3.1415926535;
1109     PetscReal umax, umin, vmax, vmin, u, v, RHSumax, RHSumin, RHSvmax, RHSvmin, f1, f2;
1110
1111     dts = sqrt(((rhoWater + rhoAir)*user->Bond)/(8.0*pi))*pow(user->h, 1.5);
1112
1113     dtv1 = (3.0/14.0*user->Re)*rhoWater*(user->h*user->h)/muWater;
1114
1115     dtv2 = (3.0/14.0*user->Re)*rhoAir*(user->h*user->h)/muAir;
1116
1117     ierr = VecMax(user->soln.U, NULL, &umax); CHKERRQ(ierr);
1118     ierr = VecMin(user->soln.U, NULL, &umin); CHKERRQ(ierr);
1119     ierr = VecMax(user->soln.V, NULL, &vmax); CHKERRQ(ierr);
1120     ierr = VecMin(user->soln.V, NULL, &vmin); CHKERRQ(ierr);
1121     ierr = VecMax(user->RHSu, NULL, &RHSumax); CHKERRQ(ierr);
1122     ierr = VecMin(user->RHSu, NULL, &RHSumin); CHKERRQ(ierr);
1123     ierr = VecMax(user->RHSv, NULL, &RHSvmax); CHKERRQ(ierr);
1124 }

```

```

1130     ierr = VecMin(user->RHSv,NULL,&RHSvmin); CHKERRQ(ierr);
1131
1132     u = fmax(fabs(umax),fabs(umin));
1133     v = fmax(fabs(vmax),fabs(vmin));
1134
1135     dtc1 = user->h/u;
1136     dtc2 = user->h/v;
1137
1138     dtv = fmin(dtv1,dtv2);
1139     dtc = fmin(dtc1,dtc2);
1140
1141     f1 = fmax(fabs(RHSumax),fabs(RHSumin));
1142     f2 = fmax(fabs(RHSvmax),fabs(RHSvmin));
1143
1144     dtf1 = 2*user->h/f1;
1145     dtf2 = 2*user->h/f2;
1146
1147     dtf = fmin(dtf1,dtf2);
1148
1149     user->dt_old = user->dt;
1150
1151     user->dt = 0.5*fmin(fmin(fmin(dts,dtv),dtc),dtf);
1152
1153     user->time = user->time + user->dt;
1154
1155     PetscPrintf(PETSC_COMM_WORLD,"Physical Time: %g\n",user->time);
1156     PetscPrintf(PETSC_COMM_WORLD,"dt: %g\n",user->dt);
1157     PetscPrintf(PETSC_COMM_WORLD,"umax: %g, vmax: %g\n",u,v);
1158
1159     return(0);
1160 }
1161
1162
1163
1164
1165 #undef __FUNCT__
1166 #define __FUNCT__ "WriteOutput"
1167 extern PetscErrorCode WriteOutput(UserContext *user){
1168
1169     PetscErrorCode ierr;
1170
1171     /* normal to interface */
1172     //WriteVec(user->normalX,"normalX"); CHKERRQ(ierr);
1173     //WriteVec(user->normalY,"normalY"); CHKERRQ(ierr);
1174
1175     /* write intermediate velocity field */
1176     //ierr = WriteVec(user->solstar.U,"Ustar"); CHKERRQ(ierr);
1177     //ierr = WriteVec(user->solstar.V,"Vstar"); CHKERRQ(ierr);
1178
1179     /* RHS of projection step */
1180     //WriteVec(user->RHS,"b"); CHKERRQ(ierr);
1181
1182     /* write jacobian matrix */
1183     //ierr = WriteMat(user->Jac,"Jac"); CHKERRQ(ierr);
1184
1185     /* write the KSP solution */
1186     //ierr = WriteVec(user->solstar.P,"KSP"); CHKERRQ(ierr);
1187
1188     /* velocity at n+1 */
1189     ierr = WriteVec(user->solnpl.U,"Unpl"); CHKERRQ(ierr);
1190     ierr = WriteVec(user->solnpl.V,"Vnpl"); CHKERRQ(ierr);
1191     //ierr = WriteVec(user->solnpl.P,"Pnpl"); CHKERRQ(ierr);
1192
1193     /* define level set */
1194     //ierr = WriteVec(user->phi_0,"levelset"); CHKERRQ(ierr);
1195
1196     /* reinitialize */
1197     //ierr = WriteVec(user->phi,"phi"); CHKERRQ(ierr);
1198
1199     /* define variables */
1200     //WriteVec(user->H,"Heaviside");
1201     //WriteVec(user->soln.mu,"Mu");
1202     //WriteVec(user->soln.rho,"Rho");
1203
1204     /* advect level set */
1205     ierr = WriteVec(user->phi_new,"phinpl"); CHKERRQ(ierr);
1206
1207     return(0);
1208 }
1209
1210
1211
1212
1213
1214 #undef __FUNCT__
1215 #define __FUNCT__ "ConserveMass"
1216 PetscErrorCode ConserveMass(DM da, UserContext *user){
1217
1218     PetscErrorCode ierr;
1219     PetscReal h = user->h;
1220     PetscInt i,j,xs,ys,xm,ym;
1221     PetscReal **H, **_phi, delta;
1222     double pi = 3.1415926535, eps = 2*user->h;
1223
1224     ierr = DMDAVecGetArray(da,user->H,&_H);
1225     ierr = DMDAVecGetArray(da,user->phi,&_phi);
1226
1227     ierr = DMDAGetCorners(da,&xs,&ys,NULL,&xm,&ym,NULL); CHKERRQ(ierr);
1228
1229     user->VolumeNew = 0.0; user->Length = 0.0;
1230
1231     for (j=ys; j<ys+ym; j++){
1232         for (i=xs; i<xs+xm; i++){

```

```

1233
1234     /* compute volume */
1235     if(user->counter == 0){
1236         user->Volume0 += (1.0 - .H[j][i])*h*h;
1237     }
1238
1239     else if(user->counter > 0){
1240         user->VolumeNew += (1.0 - .H[j][i])*h*h;
1241
1242         /* compute length */
1243         if( fabs(_phi[j][i]) <= 1.5*h){
1244             delta = 0.5*(1 + cos(pi*_phi[j][i]/eps))/eps;
1245         }else if(fabs(_phi[j][i]) > 1.5*h){
1246             delta = 0.0;
1247         }
1248         user->Length += delta*h*h;
1249     }
1250 }
1251 }
1252 }
1253
1254 if(user->counter > 0){
1255     /* compute dphi */
1256     user->dphi = (user->VolumeNew - user->Volume0) / user->Length;
1257
1258     for (j=ys; j<ys+ym; j++) {
1259         for (i=xs; i<xs+xm; i++) {
1260             /* update phi */
1261             _phi[j][i] += user->dphi;
1262         }
1263     }
1264 }
1265
1266
1267 if(user->counter == 0){
1268     PetscPrintf(PETSC_COMM_WORLD,"Initial volume: %g\n",user->Volume0);
1269 }else if(user->counter > 0){
1270     printf("New volume: %g and Length: %g\n",user->VolumeNew, user->Length);
1271     printf("volume change: %g and dphi: %g\n",user->VolumeNew - user->Volume0,user->dphi);
1272 }
1273
1274
1275 ierr = DMDAVecRestoreArray(da,user->H,&.H);
1276 ierr = DMDAVecRestoreArray(da,user->phi,&_phi);
1277
1278
1279 if(user->counter > 0){
1280     FILE *volume, *length;
1281     volume = fopen("volume.dat","a+");
1282     length = fopen("length.dat","a+");
1283     fprintf(volume,"%g\t",user->VolumeNew);
1284     fprintf(length,"%g\t",user->Length);
1285     fclose(volume);
1286     fclose(length);
1287 }
1288
1289 return(0);
1290 }
1291
1292
1293
1294
1295 // can be used only if matlab is configured
1296 #undef __FUNCT__
1297 #define __FUNCT__ "Screenshot"
1298 PetscErrorCode Screenshot(UserContext *user,Engine *ep){
1299
1300     PetscErrorCode ierr;
1301
1302     /* velocity at n+1 */
1303     ierr = WriteVec(user->solnpl.U,"Unp1"); CHKERRQ(ierr);
1304     ierr = WriteVec(user->solnpl.V,"Vnp1"); CHKERRQ(ierr);
1305     ierr = WriteVec(user->solnpl.P,"Pnp1"); CHKERRQ(ierr);
1306     /* level set */
1307     ierr = WriteVec(user->phi_new,"phinp1"); CHKERRQ(ierr);
1308
1309     char counter[16]; sprintf(counter,"counter = %4d",user->counter);
1310     char time[16]; sprintf(time,"time = %.3f",user->time);
1311
1312     engEvalString(ep,counter);
1313     engEvalString(ep,time);
1314
1315     if(user->counter == 0){
1316         engEvalString(ep,"cd Output");
1317     }
1318     engEvalString(ep,"screenshot");
1319
1320
1321     return(0);
1322 }

```