

# 2D Front Tracking Method in Python

Tingyi Lu

July 27, 2018

## 1 Introduction

Two immiscible fluids, which are separated by a sharp interface, generally exist in nature and many industrial processes. Front-tracking method uses an explicit projection finite-volume method to solve the flow in a fixed grid, and massless marker points to represent and track the interface. The method was originally developed by Unverdi and Tryggvason in 1992.

Currently, Python is becoming the world's most popular coding language. As Python is open source and free, anyone can easily gain access to it. It is a powerful language and well-known, making it a good choice to write the front-tracking code.

Professor Tryggvason has written the 2D front-tracking code in Matlab, and we will convert that code to Python. In this code, we used Python version 3.6 along with NumPy package for scientific computing and Matplotlib library for visualization. We have converted all 3 Matlab codes to Python ones. Each one has two versions: version A has regular for loops, while version B has been vectorized for the most part.

All Python codes are listed within this report and computation times are all measured with my laptop. I list all major modifications for each code in the following sections.

## 2 Code 1

Matlab and NumPy Python have a lot in common, but there are a few key differences. The main difference is Matlab uses one-based indexing for arrays while Python uses zero-based indexing. There are two ways to work around this difference. Both versions A and B of Code 1 demonstrate one method in which index 0 is ignored and all arrays start with index 1. The endpoint for each variable increases by one in order to compensate for the fact that Python does not include the endpoint in the range. The size of each array will be larger by 1 on each dimension because of the 0 that is not counted. For each variable, the start point stays the same. However, the end point will increase by 1 because the endpoint in Matlab is included in the range while the endpoint in Python is not included.

The format of “for loops” between Python and Matlab is also different. In Python, the format is as follows: “for [variable] in range( ):”. Unlike Matlab, where an “end” is used to signify the end of the loop, Python uses indentation within a loop. Likewise, “if loops” and “else if loops” are formatted in a similar manner.

There are some simple stylistic differences that need to be changed. For example, to create an array of zeros, Python uses `numpy.zeros((Nx,Ny))` with Nx and Ny being the size of the 1st and 2nd dimensions. Additionally, in Matlab, parenthesis surround the start point and end point of a variable, but brackets are used in Python. Also, in Python, two variables cannot be directly set equal to each other so `.copy()` must be added to the variable on the right of the equal sign. The semicolons or ellipses at the very end of each line in Matlab are not necessary in Python. However, keeping them will not affect the code. For long statements that extend over multiple lines, the statement can be wrapped by enclosing it within parenthesis. For the print function, the printed variables need to be between the parenthesis in a statement like `print( )`.

The Code 1 in two versions are listed in Listing 1 and 2, separately. In version A, no vectorization is used. But in version B, vectorization replaces all nested “for loops”. “For loops” can be vectorized by replacing all instances of a variable with its specified indexing range. For example, the corresponding code to Code 1A's

```

for i in range(2,nx+2):
for j in range(2,ny+2):
    chi[i, j] = ...

```

is

```

chi[2:nx+2,2:ny+2]

```

in Code 1B. As seen, the range of  $i$  and  $j$  are replaced with their respective indexing range. If version A has  $[i+1,j]$ , then in version B both the start point and end point increase by 1 for  $i$  only, giving  $[3:nx+3,2:ny+2]$ . Similarly, if version A has  $[i,j-1]$ , then version B has  $[2:nx+2,1:ny+1]$ . Other variations of this have a similar approach.

The benefits of using vectorization is that is much faster than for loops. In the case of Code 1, the non-vectorized version A takes 90.97 seconds to run completely for 400 steps without visualization, while the vectorized B version only takes 3.68 second. Version B is nearly 25 times faster than version A, showing how much more effective vectorization is.

However, the Pressure Solver SOR(Successive OverRelaxation) used in Matlab can't be easily vectorized, because  $p[i,j]$  is showed in both sides of the equation. Instead, Red-Black SOR is used in both versions 1A and 1B, and vectorized in 1B.

### 3 Code 2

The python code for code 2 is listed in Listing 3 and 4.

Code 2A and 2B are approached in nearly the same way as Code 1A and 1B. Please refer to the previous section for the methodology. One difference is we used an index beginning with zero. In this method, Python's numbering system is fully utilized and starts on 0. Although the index system does not have a direct correspondence, with Matlab's  $n$ th term being indexed at  $n-1$  in Python, the size of the arrays are the same in both. Because of this, the value of the start point must be 1 lower while the end index number stays the same.

The other difference occurs in that Python can't use a dynamic matrix size as in Matlab. Instead, we introduced a variable  $Maxf$ , with a value of 2000, to give the maximum size for all front arrays, and their actual size will still be  $Nf$ .

Again, just like in Code 1, version B is much faster than version A. Since Code 2 is a longer code, both the A version and B version take longer. However, version B is approximately 11.5 times as fast, taking only 16.25 seconds compared to the 187 seconds version A took to run the entire code. This lower speed ratio compared to Code 1 is caused by the fact that the front parts' for loops were not vectorized.

### 4 Code 3

The python code for code 3 is listed in Listing 5 and 6.

Code 3 is very similar to Code 2. They are mostly identical, with Code 3 having several additional sections. However, those new sections can be approached with the same methods as described before. This code also utilizes zero-based indexing, like in Code 2.

Code 3B takes only 30.12 seconds compared to version 3A taking 520 seconds, making it more than 17 times as fast.

## Listings

1	Python Code 1A . . . . .	4
2	Python Code 1B . . . . .	7
3	Python Code 2A . . . . .	10
4	Python Code 2B . . . . .	14
5	Python Code 3A . . . . .	18
6	Python Code 3B . . . . .	23

Listing 1: Python Code 1A

```

1  #=====
2  # CodeC1-advChi.py
3  # A very simple Navier-Stokes solver for a drop falling in a
4  # rectangular box, using a conservative form of the equations.
5  # A first-order explicit projection method and centered in space
6  # discretization are used. The marker function is advected by
7  # an advection diffusion equation.
8  # Original Matlab code by Gretar Tryggvason
9  # Python code converted by Tingyi Lu on 7/25/2018
10 #=====
11
12 import numpy as np
13 import matplotlib.pyplot as plt
14 import time as time0
15
16 Lx=1.0;Ly=1.0;gx=0.0;gy=-100.0 # Domain size and
17 rho1=1.0; rho2=2.0; m0=0.01 # physical variables
18 unorth=0;usouth=0;veast=0;vwest=0;time=0.0
19 rad=0.15;xc=0.5;yc=0.7 # Initial drop size and location
20
21 #----- Numerical variables -----
22 nx=32;ny=32;dt=0.00125;nstep=400; maxit=200; maxError=0.001; beta=1.5
23
24 #----- Zero various arrays -----
25 u=np.zeros((nx+2,ny+3)); v=np.zeros((nx+3,ny+2)); p=np.zeros((nx+3,ny+3))
26 ut=np.zeros((nx+2,ny+3)); vt=np.zeros((nx+3,ny+2)); tmp1=np.zeros((nx+3,ny+3))
27 uu=np.zeros((nx+2,ny+2)); vv=np.zeros((nx+2,ny+2)); tmp2=np.zeros((nx+3,ny+3))
28 r=np.zeros((nx+3,ny+3)); chi=np.zeros((nx+3,ny+3))
29
30
31 dx=Lx/nx;dy=Ly/ny # Set the grid
32 x=np.linspace(-1.5*dx, (nx+0.5)*dx, nx+3)
33 y=np.linspace(-1.5*dy, (ny+0.5)*dy, ny+3)
34 xh=np.linspace(-dx, nx*dx, nx+2)
35 yh=np.linspace(-dy, ny*dy, ny+2)
36
37 X, Y = np.meshgrid(x,y)
38
39 #----- Initial Conditions -----
40 r = np.zeros((nx+3,nx+3))+rho1 # Set density
41
42 for i in range(nx+3):
43     for j in range(ny+3): # for the domain and the drop
44         if((x[i]-xc)**2+(y[j]-yc)**2 < rad**2):
45             r[i,j] = rho2
46             chi[i,j] = 1.0
47
48
49 plt.contour(X,Y,chi,T)
50 plt.axis('scaled')
51 plt.axis([0, Lx, 0, Ly], aspect=1)
52
53 time_start = time0.time()
54
55 #----- START TIME LOOP -----
56 for istep in range(nstep):
57     print(istep)
58
59     #--- ADVECT marker using centered difference plus diffusion ---
60     chio=chi.copy()
61
62     for i in range(2,nx+2):
63         for j in range(2,ny+2):
64             chi[i,j]= (chio[i,j]-(0.5*dt/dx)*(u[i,j]*(chio[i+1,j]
65                 +chio[i,j])-u[i-1,j]*(chio[i-1,j]+chio[i,j]))
66                 -(0.5* dt/dy)*(v[i,j]*(chio[i,j+1]
67                 +chio[i,j])-v[i,j-1]*(chio[i,j-1]+chio[i,j]) )

```

```

68         +(m0*dt/dx/dx)*(chio[i+1,j]-2.0*chio[i,j]+chio[i-1,j])
69         +(m0*dt/dy/dy)*(chio[i,j+1]-2.0*chio[i,j]+chio[i,j-1]) )
70
71     #----- Update the density -----
72     ro=r.copy()
73     r = rho1 + (rho2-rho1)*chi
74
75     #----- Set tangential velocity at boundaries -----
76     u[:,1]=2*usouth-u[:,2];u[:,ny+2]=2*unorth-u[:,ny+1]
77     v[1,:]=2*vwest-v[2,:];v[nx+2,:]=2*veast-v[nx+1,:]
78
79     #----- Find the predicted velocities -----
80     # Temporary u-velocity
81
82     for i in range(2,nx+1):
83         for j in range(2,ny+2): # Temporary u-velocity
84             ut[i,j]= ( (2.0/(r[i+1,j]+r[i,j]))*( 0.5*(ro[i+1,j]+ro[i,j])*u[i,j]+ dt* (
85                 -(0.25/dx)*(ro[i+1,j]*(u[i+1,j]+u[i,j])**2-ro[i,j]*(u[i,j]+u[i-1,j])**2)
86                 -(0.0625/dy)*( (ro[i,j]+ro[i+1,j]+ro[i,j+1]+ro[i+1,j+1])*(u[i,j+1]+u[i,j])*(v[i+1,j]+v[i,j])
87                     -(ro[i,j]+ro[i+1,j]+ro[i+1,j-1]+ro[i,j-1])*(u[i,j]+u[i,j-1])*(v[i+1,j-1]+v[i,j-1]))
88                 +m0*((u[i+1,j]-2*u[i,j]+u[i-1,j])/dx**2+ (u[i,j+1]-2*u[i,j]+u[i,j-1])/dy**2)
89                 + 0.5*(ro[i+1,j]+ro[i,j])*gx ) ) )
90
91     #Temporary v-velocity
92     for i in range(2,nx+2):
93         for j in range(2,ny+1): # Temporary v-velocity
94             vt[i,j]= ( (2.0/(r[i,j+1]+r[i,j]))*(0.5*(ro[i,j+1]+ro[i,j])*v[i,j]+ dt* (
95                 -(0.0625/dx)*( (ro[i,j]+ro[i+1,j]+ro[i+1,j+1]+ro[i,j+1])*(u[i,j]+u[i,j+1])*(v[i,j]+v[i+1,j])
96                     -(ro[i,j]+ro[i,j+1]+ro[i-1,j+1]+ro[i-1,j])*(
97                         (u[i-1,j+1]+u[i-1,j])*(v[i,j]+v[i-1,j]) )
98                 -(0.25/dy)*ro[i,j+1]*(v[i,j+1]+v[i,j])**2-ro[i,j]*(v[i,j]+v[i,j-1])**2 )
99                 +m0*((v[i+1,j]-2*v[i,j]+v[i-1,j])/dx**2+(v[i,j+1]-2*v[i,j]+v[i,j-1])/dy**2)
100                 + 0.5*(ro[i,j+1]+ro[i,j])*gy ) ) )
101
102     #----- Solve the Pressure Equation -----
103     rt=r.copy(); lrg=1e20 # Compute source term and the coefficient for p[i,j]
104     rt[:,1]=lrg;rt[:,ny+2]=lrg
105     rt[1,:]=lrg;rt[nx+2,:]=lrg
106
107     for i in range(2,nx+2):
108         for j in range(2,ny+2):
109             tmp1[i,j]= (0.5/dt)*((ut[i,j]-ut[i-1,j])/dx+(vt[i,j]-vt[i,j-1])/dy )
110             tmp2[i,j]= (1.0/( (1./dx)*(1./(dx*(rt[i+1,j]+rt[i,j]))+
111                 1./(dx*(rt[i-1,j]+rt[i,j])) )+
112                 (1./dy)*(1./(dy*(rt[i,j+1]+rt[i,j]))+
113                 1./(dy*(rt[i,j-1]+rt[i,j])) ) ) )
114
115     for it in range(maxit): # Solve for pressure by SOR
116         oldp=p.copy()
117
118         #Red & Black SOR
119         for ipass in range(2):
120             rb = ipass
121             for j in range(2,ny+2):
122                 for i in range(2+rb, nx+2, 2):
123                     p[i,j]= ( (1.0-beta)*p[i,j] + beta*tmp2[i,j]*(
124                         (1.0/dx/dx)*( p[i+1,j]/(rt[i+1,j]+rt[i,j])
125                             +p[i-1,j]/(rt[i-1,j]+rt[i,j]))
126                         +(1.0/dy/dy)*( p[i,j+1]/(rt[i,j+1]+rt[i,j])
127                             +p[i,j-1]/(rt[i,j-1]+rt[i,j]))
128                         - tmp1[i,j] ) )
129                 rb=1-rb
130
131             if (np.abs(oldp-p)).max() < maxError :
132                 break
133
134     # Correct the u-velocity

```

```

136     for i in range(2,nx+1):
137         for j in range(2,ny+2):
138             u[i,j]=ut[i,j]-dt*(2.0/dx)*(p[i+1,j]-p[i,j])/(r[i+1,j]+r[i,j])
139
140     # Correct the v-velocity
141     for i in range(2,nx+2):
142         for j in range(2,ny+1):
143             v[i,j]=vt[i,j]-dt*(2.0/dy)*(p[i,j+1]-p[i,j])/(r[i,j+1]+r[i,j])
144
145     #----- Plot the results -----
146     time=time+dt                # plot the results
147     uu[1:nx+2,1:ny+2]=(0.5*(u[1:nx+2,2:ny+3]+u[1:nx+2,1:ny+2]))
148     vv[1:nx+2,1:ny+2]=(0.5*(v[2:nx+3,1:ny+2]+v[1:nx+2,1:ny+2]))
149
150
151     plt.cla()
152     plt.contour(x[2:nx+2],y[2:ny+2],chi.T[2:nx+2,2:ny+2])
153     plt.quiver(xh[1:],yh[1:],uu.T[1:,1:],vv.T[1:,1:])
154
155     plt.pause(0.01)
156
157
158     print('time.elapsed= %s' % (time0.time() - time_start) )
159     plt.close()
160
161
162     # End of time step

```

Listing 2: Python Code 1B

```

1  #=====
2  # CodeC1-advChi.py
3  # A very simple Navier-Stokes solver for a drop falling in a
4  # rectangular box, using a conservative form of the equations.
5  # A first-order explicit projection method and centered in space
6  # discretization are used. The marker function is advected by
7  # an advection diffusion equation.
8  # Original Matlab code by Greta Tryggvason
9  # Python code converted by Tingyi Lu on 7/25/2018
10 #=====
11
12 import numpy as np
13 import matplotlib.pyplot as plt
14 import time as time0
15
16 Lx=1.0;Ly=1.0;gx=0.0;gy=-100.0 # Domain size and
17 rho1=1.0; rho2=2.0; m0=0.01 # physical variables
18 unorth=0;usouth=0;veast=0;vwest=0;time=0.0
19 rad=0.15;xc=0.5;yc=0.7 # Initial drop size and location
20
21 #----- Numerical variables -----
22 nx=32;ny=32;dt=0.00125;nstep=400; maxit=200; maxError=0.001; beta=1.5
23
24 #----- Zero various arrays -----
25 u=np.zeros((nx+2,ny+3)); v=np.zeros((nx+3,ny+2)); p=np.zeros((nx+3,ny+3))
26 ut=np.zeros((nx+2,ny+3)); vt=np.zeros((nx+3,ny+2)); tmp1=np.zeros((nx+3,ny+3))
27 uu=np.zeros((nx+2,ny+2)); vv=np.zeros((nx+2,ny+2)); tmp2=np.zeros((nx+3,ny+3))
28 r=np.zeros((nx+3,ny+3)); chi=np.zeros((nx+3,ny+3))
29
30 dx=Lx/nx;dy=Ly/ny # Set the grid
31 x=np.linspace(-1.5*dx, (nx+0.5)*dx, nx+3)
32 y=np.linspace(-1.5*dy, (ny+0.5)*dy, ny+3)
33 xh=np.linspace(-dx, nx*dx, nx+2)
34 yh=np.linspace(-dy, ny*dy, ny+2)
35
36 X, Y = np.meshgrid(x,y)
37
38 #----- Initial Conditions -----
39 r = np.zeros((nx+3,nx+3))+rho1 # Set density
40
41 for i in range(nx+3):
42     for j in range(ny+3): # for the domain and the drop
43         if((x[i]-xc)**2+(y[j]-yc)**2 < rad**2):
44             r[i,j] = rho2
45             chi[i,j] = 1.0
46
47 plt.contour(X,Y,chi.T)
48 plt.axis('scaled')
49 plt.axis([0, Lx, 0, Ly], aspect=1)
50
51 time_start = time0.time()
52
53 #----- START TIME LOOP -----
54 for istep in range(nstep):
55     print(istep)
56
57     #--- ADVECT marker using centered difference plus diffusion ---
58     chio=chi.copy()
59
60     chi[2:nx+2,2:ny+2]=(chio[2:nx+2,2:ny+2]-(0.5*dt/dx)*(u[2:nx+2,2:ny+2]*(chio[3:nx+3,2:ny+2]
61         +chio[2:nx+2,2:ny+2]) - u[1:nx+1,2:ny+2]*(chio[1:nx+1,2:ny+2]+chio[2:nx+2,2:ny+2]))
62         -(0.5* dt/dy)*(v[2:nx+2,2:ny+2]*(chio[2:nx+2,3:ny+3]
63         +chio[2:nx+2,2:ny+2]) - v[2:nx+2,1:ny+1]*(chio[2:nx+2,1:ny+1]+chio[2:nx+2,2:ny+2]))
64         +(m0*dt/dx/dx)*(chio[3:nx+3,2:ny+2]-2.0*chio[2:nx+2,2:ny+2]+chio[1:nx+1,2:ny+2]))
65         +(m0*dt/dy/dy)*(chio[2:nx+2,3:ny+3]-2.0*chio[2:nx+2,2:ny+2]+chio[2:nx+2,1:ny+1]))
66
67     #----- Update the density -----

```

```

68 ro=r.copy()
69 r = rho1 + (rho2-rho1)*chi
70
71 #----- Set tangential velocity at boundaries -----
72 u[:,1]=2*usouth-u[:,2];u[:,ny+2]=2*unorth-u[:,ny+1]
73 v[1,:]=2*vwest-v[2,:];v[nx+2,:]=2*veast-v[nx+1,:]
74
75 ut[2:nx+1,2:ny+2]=((2.0/(r[3:nx+2,2:ny+2]+r[2:nx+1,2:ny+2]))*(0.5*
76     (ro[3:nx+2,2:ny+2]+ro[2:nx+1,2:ny+2])*u[2:nx+1,2:ny+2]+ dt* (
77     -(0.25/dx)*(ro[3:nx+2,2:ny+2]*(u[3:nx+2,2:ny+2]+u[2:nx+1,2:ny+2])**2-
78     ro[2:nx+1,2:ny+2]*(u[2:nx+1,2:ny+2]+u[1:nx,2:ny+2])**2)
79     -(0.0625/dy)*( ro[2:nx+1,2:ny+2]+ro[3:nx+2,2:ny+2]+ro[2:nx+1,3:ny+3]+ro[3:nx+2,3:ny+3])*
80     (u[2:nx+1,3:ny+3]+u[2:nx+1,2:ny+2])*(v[3:nx+2,2:ny+2]+v[2:nx+1,2:ny+2])
81     -(ro[2:nx+1,2:ny+2]+ro[3:nx+2,2:ny+2]+ro[3:nx+2,1:ny+1]+ro[2:nx+1,1:ny+1])*(u[2:nx+1,2:ny+2]
82     +u[2:nx+1,1:ny+1])*(v[3:nx+2,1:ny+1]+v[2:nx+1,1:ny+1]))
83     +m0*((u[3:nx+2,2:ny+2]-2*u[2:nx+1,2:ny+2]+u[1:nx,2:ny+2])/dx**2+
84     (u[2:nx+1,3:ny+3]-2*u[2:nx+1,2:ny+2]+u[2:nx+1,1:ny+1])/dy**2)
85     + 0.5*(ro[3:nx+2,2:ny+2]+ro[2:nx+1,2:ny+2])*gx ) )
86
87 vt[2:nx+2,2:ny+1]=((2.0/(r[3:nx+2,3:ny+2]+r[2:nx+2,2:ny+1]))*(0.5*
88     (ro[2:nx+2,3:ny+2]+ro[2:nx+2,2:ny+1])*v[2:nx+2,2:ny+1]+ dt* (
89     -(0.0625/dx)*( ro[2:nx+2,2:ny+1]+ro[3:nx+3,2:ny+1]+ro[3:nx+3,3:ny+2]+ro[2:nx+2,3:ny+2])*
90     (u[2:nx+2,2:ny+1]+u[2:nx+2,3:ny+2])*(v[2:nx+2,2:ny+1]+v[3:nx+3,2:ny+1])
91     -( ro[2:nx+2,2:ny+1]+ro[2:nx+2,3:ny+2]+ro[1:nx+1,3:ny+2]+ro[1:nx+1,2:ny+1])*
92     (u[1:nx+1,3:ny+2]+u[1:nx+1,2:ny+1])*(v[2:nx+2,2:ny+1]+v[1:nx+1,2:ny+1]) )
93     -(0.25/dy)*(ro[2:nx+2,3:ny+2]*(v[2:nx+2,3:ny+2]+v[2:nx+2,2:ny+1])**2-
94     ro[2:nx+2,2:ny+1]*(v[2:nx+2,2:ny+1]+v[2:nx+2,1:ny])**2 )
95     +m0*((v[3:nx+3,2:ny+1]-2*v[2:nx+2,2:ny+1]+v[1:nx+1,2:ny+1])/dx**2+
96     (v[2:nx+2,3:ny+2]-2*v[2:nx+2,2:ny+1]+v[2:nx+2,1:ny])/dy**2)
97     + 0.5*(ro[2:nx+2,3:ny+2]+ro[2:nx+2,2:ny+1])*gy ) )
98
99 #----- Solve the Pressure Equation -----
100 rt=r.copy(); lrg=1e20 # Compute source term and the coefficient for p[i,j]
101 rt[:,1]=lrg;rt[:,ny+2]=lrg
102 rt[1,:]=lrg;rt[nx+2,:]=lrg
103
104
105 tmp1[2:nx+2,2:ny+2]= ((0.5/dt)*( (ut[2:nx+2,2:ny+2]-ut[1:nx+1,2:ny+2])/dx+
106     (vt[2:nx+2,2:ny+2]-vt[2:nx+2,1:ny+1])/dy ))
107 tmp2[2:nx+2,2:ny+2]=(1.0/( (1./dx)*(1./(dx*(rt[3:nx+3,2:ny+2]+rt[2:nx+2,2:ny+2]))+
108     1./(dx*(rt[1:nx+1,2:ny+2]+rt[2:nx+2,2:ny+2])) ) +
109     (1./dy)*(1./(dy*(rt[2:nx+2,3:ny+3]+rt[2:nx+2,2:ny+2]))+
110     1./(dy*(rt[2:nx+2,1:ny+1]+rt[2:nx+2,2:ny+2])) ) ) )
111
112 for it in range(maxit): # Solve for pressure by SOR
113     oldp=p.copy()
114
115     #Red & Black SOR
116     for ipass in range(2):
117         rb = ipass
118         p[2+rb:nx+2,2:ny+2:2] = ( (1.0-beta)*p[2+rb:nx+2,2:ny+2:2] + beta*tmp2[2+rb:nx+2,2:ny+2:2]*(
119             (1.0/dx/dx)*( p[3+rb:nx+3,2:ny+2:2]/(rt[3+rb:nx+3,2:ny+2:2]+rt[2+rb:nx+2,2:ny+2:2])
120             +p[1+rb:nx+1,2:ny+2:2]/(rt[1+rb:nx+1,2:ny+2:2]+rt[2+rb:nx+2,2:ny+2:2]))
121             +(1.0/dy/dy)*( p[2+rb:nx+2,3:ny+3:2]/(rt[2+rb:nx+2,3:ny+3:2]+rt[2+rb:nx+2,2:ny+2:2])
122             +p[2+rb:nx+2,2,1:ny+1:2]/(rt[2+rb:nx+2,2,1:ny+1:2]+rt[2+rb:nx+2,2:ny+2:2]))
123             - tmp1[2+rb:nx+2,2:ny+2:2] ) )
124
125         rb=1-ipass
126         p[2+rb:nx+2,3:ny+2:2] = ( (1.0-beta)*p[2+rb:nx+2,3:ny+2:2] + beta*tmp2[2+rb:nx+2,3:ny+2:2]*(
127             (1.0/dx/dx)*( p[3+rb:nx+3,3:ny+2:2]/(rt[3+rb:nx+3,3:ny+2:2]+rt[2+rb:nx+2,3:ny+2:2])
128             +p[1+rb:nx+1,3:ny+2:2]/(rt[1+rb:nx+1,3:ny+2:2]+rt[2+rb:nx+2,3:ny+2:2]))
129             +(1.0/dy/dy)*( p[2+rb:nx+2,4:ny+3:2]/(rt[2+rb:nx+2,4:ny+3:2]+rt[2+rb:nx+2,3:ny+2:2])
130             +p[2+rb:nx+2,2:ny+1:2]/(rt[2+rb:nx+2,2:ny+1:2]+rt[2+rb:nx+2,3:ny+2:2]))
131             - tmp1[2+rb:nx+2,3:ny+2:2] ) )
132
133     p[1,:]= p[2,:]; p[nx+2,:]= p[nx+1,:]
134     p[:,1]= p[:,2]; p[:,ny+2]= p[:,ny+1]
135

```



```

136     if (np.abs(oldp-p)).max() < maxError :
137         break
138
139     # Correct the u-velocity
140     u[2:nx+1,2:ny+2]=(ut[2:nx+1,2:ny+2]-dt*(2.0/dx)*(p[3:nx+2,2:ny+2]-p[2:nx+1,2:ny+2])/(r[3:nx+2,2:ny+2]+r[2:nx
141         +1,2:ny+2]))
142
143     # Correct the v-velocity
144     v[2:nx+2,2:ny+1]=(vt[2:nx+2,2:ny+1]-dt*(2.0/dy)*(p[2:nx+2,3:ny+2]-p[2:nx+2,2:ny+1])/(r[2:nx+2,3:ny+2]+r[2:nx
145         +2,2:ny+1]))
146
147     #----- Plot the results -----
148     time=time+dt           # plot the results
149     uu[1:nx+2,1:ny+2]=(0.5*(u[1:nx+2,2:ny+3]+u[1:nx+2,1:ny+2]))
150     vv[1:nx+2,1:ny+2]=(0.5*(v[2:nx+3,1:ny+2]+v[1:nx+2,1:ny+2]))
151
152     plt.cla()
153     plt.contour(x[2:nx+2],y[2:ny+2],chi.T[2:nx+2,2:ny+2])
154     plt.quiver(xh[1:],yh[1:],uu.T[1:,1:],vv.T[1:,1:])
155
156     plt.pause(0.01)
157
158     print('time.elapsed= %s' % (time0.time() - time_start) )
159
160     plt.close()
161
162     # End of time step

```

Listing 3: Python Code 2A

```

1  #=====
2  # CodeC2-frt.m
3  # A very simple Navier-Stokes solver for a drop falling in a
4  # rectangular box, using a conservative form of the equations.
5  # A first-order explicit projection method and centered in space
6  # discretization are used. The marker function is advected by
7  # Tront Tracking.
8  # Original Matlab code by Gretar Tryggvason
9  # Python code converted by Tingyi Lu on 7/25/2018
10 #=====
11
12 import numpy as np
13 import matplotlib.pyplot as plt
14 import time as time0
15
16 Lx=1.0;Ly=1.0;gx=0.0;gy=-100.0 # Domain size and
17 rho1=1.0; rho2=2.0; m0=0.01 # physical variables
18 unorth=0;usouth=0;veast=0;vwest=0;time=0.0
19 rad=0.15;xc=0.5;yc=0.7 # Initial drop size and location
20
21 #----- Numerical variables -----
22 nx=32;ny=32;dt=0.00125;nstep=400; maxit=200;maxError=0.001;beta=1.5;
23 Nf=100; Maxf=2000
24
25 #----- Zero various arrays -----
26 u=np.zeros((nx+1,ny+2)); v=np.zeros((nx+2,ny+1)); p=np.zeros((nx+2,ny+2))
27 ut=np.zeros((nx+1,ny+2)); vt=np.zeros((nx+2,ny+1)); tmp1=np.zeros((nx+2,ny+1))
28 uu=np.zeros((nx+1,ny+1)); vv=np.zeros((nx+1,ny+1)); tmp2=np.zeros((nx+2,ny+2))
29 r=np.zeros((nx+2,ny+2)); chi=np.zeros((nx+2,ny+2))
30 d=np.zeros((nx+1,nx+1))
31 xf=np.zeros(Maxf); yf=np.zeros(Maxf)
32 uf=np.zeros(Maxf); vf=np.zeros(Maxf)
33
34 dx=Lx/nx;dy=Ly/ny # Set the grid
35 x=np.linspace(-.5*dx, (nx+0.5)*dx, nx+2)
36 y=np.linspace(-.5*dx, (ny+0.5)*dy, ny+2)
37 xh=np.linspace(0, Lx, nx+1)
38 yh=np.linspace(0, Ly, ny+1)
39
40
41 #----- Initial Conditions -----
42 r[:,]=rho1 # Set density
43
44 for i in range(1,nx+1): # for the domain and the drop
45     for j in range(1,ny+1):
46         if((x[i]-xc)**2+(y[j]-yc)**2 < rad**2):
47             r[i,j]=rho2
48             chi[i,j]=1.0
49
50
51 for l in range(Nf+2):
52     xf[l]=xc-rad*np.sin(2.0*np.pi*l/Nf) # Initialize
53     yf[l]=yc+rad*np.cos(2.0*np.pi*l/Nf) # the Front
54
55
56 plt.plot(xf[0:Nf],yf[0:Nf],'k',linewidth=3)
57 plt.axis('scaled')
58 plt.axis([0,Lx,0,Ly], aspect=1)
59 plt.pause(0.0001)
60
61 time.start = time0.time()
62
63 #----- START TIME LOOP -----
64 for istep in range(nstep):
65     print(istep)
66
67

```

```

68 #----- Advect the Front -----
69
70 for l in range(1,Nf+1): # Interpolate the Front Velocities
71     ip=np.int(xf[l]/dx); jp=np.int((yf[l]+0.5*dy)/dy)
72     ax=xf[l]/dx-ip; ay=(yf[l]+0.5*dy)/dy-jp
73     uf[l]=((1.0-ax)*(1.0-ay)*u[ip,jp]+ax*(1.0-ay)*u[ip+1,jp]+
74             (1.0-ax)*ay*u[ip,jp+1]+ax*ay*u[ip+1,jp+1])
75
76     ip=np.int((xf[l]+0.5*dx)/dx); jp=np.int(yf[l]/dy)
77     ax=(xf[l]+0.5*dx)/dx-ip; ay=yf[l]/dy-jp
78     vf[l]=((1.0-ax)*(1.0-ay)*v[ip,jp]+ax*(1.0-ay)*v[ip+1,jp]+
79             (1.0-ax)*ay*v[ip,jp+1]+ax*ay*v[ip+1,jp+1])
80
81 for i in range(1,Nf+1):
82     xf[i]= xf[i]+dt*uf[i]; yf[i]=yf[i]+dt*vf[i] #Move the
83     xf[0]=xf[Nf];yf[0]=yf[Nf];xf[Nf+1]=xf[1];yf[Nf+1]=yf[1] # Front
84
85
86 #----- Update the marker function -----
87 d[:,:]=2
88
89 for l in range(1,Nf+1):
90     nfx=-(yf[l+1]-yf[l])/dx
91     nfy=(xf[l+1]-xf[l])/dy # Normal vector
92     ds=np.sqrt(nfx*nfx+nfy*nfy); nfx=nfx/ds; nfy=nfy/ds
93     xfront=0.5*(xf[l]+xf[l+1]); yfront=0.5*(yf[l]+yf[l+1])
94     ip=np.int((xfront+0.5*dx)/dx); jp=np.int((yfront+0.5*dy)/dy)
95
96     d1=np.sqrt(((xfront-x[ip])/dx)**2+((yfront-y[jp])/dy)**2)
97     d2=np.sqrt(((xfront-x[ip+1])/dx)**2+((yfront-y[jp])/dy)**2)
98     d3=np.sqrt(((xfront-x[ip+1])/dx)**2+((yfront-y[jp+1])/dy)**2)
99     d4=np.sqrt(((xfront-x[ip])/dx)**2+((yfront-y[jp+1])/dy)**2)
100
101     if d1<d[ip,jp]:
102         d[ip,jp]=d1.copy()
103         dn1=(x[ip]-xfront)*nfx/dx+(y[jp]-yfront)*nfy/dy
104         chi[ip,jp]=0.5*(1.0+np.sign(dn1))
105         if abs(dn1)<0.5:
106             chi[ip,jp]=0.5+dn1
107
108     if d2<d[ip+1,jp]:
109         d[ip+1,jp]=d2.copy()
110         dn2=(x[ip+1]-xfront)*nfx/dx+(y[jp]-yfront)*nfy/dy
111         chi[ip+1,jp]=0.5*(1.0+np.sign(dn2))
112         if abs(dn2)<0.5:
113             chi[ip+1,jp]=0.5+dn2
114
115     if d3<d[ip+1,jp+1]:
116         d[ip+1,jp+1]=d3.copy()
117         dn3=(x[ip+1]-xfront)*nfx/dx+(y[jp+1]-yfront)*nfy/dy
118         chi[ip+1,jp+1]=0.5*(1.0+np.sign(dn3))
119         if abs(dn3)<0.5:
120             chi[ip+1,jp+1]=0.5+dn3
121
122     if d4<d[ip,jp+1]:
123         d[ip,jp+1]=d4.copy()
124         dn4=(x[ip]-xfront)*nfx/dx+(y[jp+1]-yfront)*nfy/dy
125         chi[ip,jp+1]= 0.5*(1.0+np.sign(dn4))
126         if abs(dn4)<0.5:
127             chi[ip,jp+1]= 0.5+dn4
128
129
130 #----- Update the density -----
131
132 ro=r.copy()
133 r = rho1+(rho2-rho1)*chi
134
135 #----- Set tangential velocity at boundaries -----

```

```

136 u[:,0]=2*usouth-u[:,1];u[:,ny+1]=2*unorth-u[:,ny]
137 v[0,:]=2*vwest-v[1,:];v[nx+1,:]=2*veast-v[nx,:]
138
139 #----- Find the predicted velocities -----
140 for i in range(1,nx):
141     for j in range(1,ny+1): # Temporary u-velocity
142         ut[i,j]=((2.0/(r[i+1,j]+r[i,j]))*(0.5*(ro[i+1,j]+ro[i,j])*u[i,j]+ dt* (
143             -(0.25/dx)*(ro[i+1,j]*(u[i+1,j]+u[i,j])**2-ro[i,j]*(u[i,j]+u[i-1,j])**2)
144             -(0.0625/dy)*( ro[i,j]+ro[i+1,j]+ro[i,j+1]+ro[i+1,j+1])*
145                 (u[i,j+1]+u[i,j])*(v[i+1,j]+v[i,j])
146                 -(ro[i,j]+ro[i+1,j]+ro[i+1,j-1]+ro[i,j-1])*(u[i,j]
147                 +u[i,j-1])*(v[i+1,j-1]+v[i,j-1]))
148             +m0*((u[i+1,j]-2*u[i,j]+u[i-1,j])/dx**2+ (u[i,j+1]-2*u[i,j]+u[i,j-1])/dy**2)
149             + 0.5*(ro[i+1,j]+ro[i,j])*gx ) ))
150
151
152
153 for i in range(1,nx+1):
154     for j in range(1,ny): # Temporary v-velocity
155         vt[i,j]=((2.0/(r[i,j+1]+r[i,j]))*(0.5*(ro[i,j+1]+ro[i,j])*v[i,j]+ dt* (
156             -(0.0625/dx)*( ro[i,j]+ro[i+1,j]+ro[i+1,j+1]+ro[i,j+1])*
157                 (u[i,j]+u[i,j+1])*(v[i,j]+v[i+1,j])
158                 -(ro[i,j]+ro[i,j+1]+ro[i-1,j+1]+ro[i-1,j])*
159                 (u[i-1,j+1]+u[i-1,j])*(v[i,j]+v[i-1,j]) )
160             -(0.25/dy)*(ro[i,j+1]*(v[i,j+1]+v[i,j])**2-ro[i,j]*(v[i,j]+v[i,j-1])**2 )
161             +m0*((v[i+1,j]-2*v[i,j]+v[i-1,j])/dx**2+(v[i,j+1]-2*v[i,j]+v[i,j-1])/dy**2)
162             + 0.5*(ro[i,j+1]+ro[i,j])*gy ) ))
163
164
165 #----- Solve the Pressure Equation -----
166 rt=r.copy(); lrg=1000 # Compute source term and the coefficient for p(i,j)
167 rt[:,0]=lrg;rt[:,ny+1]=lrg
168 rt[0,:]=lrg;rt[nx+1,:]=lrg
169
170 for i in range(1,nx+1):
171     for j in range(1,ny+1):
172         tmp1[i,j]= (0.5/dt)* ( (ut[i,j]-ut[i-1,j])/dx+(vt[i,j]-vt[i,j-1])/dy )
173         tmp2[i,j]=(1.0/( (1./dx)*(1./(dx*(rt[i+1,j]+rt[i,j])))+
174             1./(dx*(rt[i-1,j]+rt[i,j])) )+
175             (1./dy)*(1./(dy*(rt[i,j+1]+rt[i,j]))+
176             1./(dy*(rt[i,j-1]+rt[i,j])) ) ) )
177
178
179 for it in range(maxit): # Solve for pressure by SOR
180     oldArray=p.copy()
181     #Red & Black SOR
182     for ipass in range(2):
183         rb = ipass
184         for j in range(1,ny+1):
185             for i in range(1+rb, nx+1, 2):
186                 p[i,j]= ( (1.0-beta)*p[i,j]+ beta*tmp2[i,j]*(
187                     (1.0/dx/dx)* ( p[i+1,j]/(rt[i+1,j]+rt[i,j])
188                     +p[i-1,j]/(rt[i-1,j]+rt[i,j]))
189                     +(1.0/dy/dy)* ( p[i,j+1]/(rt[i,j+1]+rt[i,j])
190                     +p[i,j-1]/(rt[i,j-1]+rt[i,j]))
191                     - tmp1[i,j] ) )
192             rb=1-rb
193
194
195     if (np.abs(oldArray-p)).max() < maxError:
196         break
197
198 for i in range(1,nx):
199     for j in range(1,ny+1): # Correct the u-velocity
200         u[i,j]=ut[i,j]-dt*(2.0/dx)*(p[i+1,j]-p[i,j])/(r[i+1,j]+r[i,j])
201
202 for i in range(1,nx+1):
203     for j in range(1,ny): # Correct the v-velocity

```

```

204         v[i,j]=vt[i,j]-dt*(2.0/dy)*(p[i,j+1]-p[i,j])/(r[i,j+1]+r[i,j])
205
206     #----- Add and delete points in the Front -----
207     xfold=xf.copy();yfold=yf.copy(); j=0
208     for l in range(1,Nf+1):
209         ds=np.sqrt( ((xfold[l]-xf[j])/dx)**2 + ((yfold[l]-yf[j])/dy)**2)
210         if ds > 0.5:
211             j=j+1;xf[j]=0.5*(xfold[l]+xf[j-1]);yf[j]=0.5*(yfold[l]+yf[j-1])
212             j=j+1;xf[j]=xfold[l];yf[j]=yfold[l]
213         elif 0.25<=ds<=0.5:
214             j=j+1;xf[j]=xfold[l];yf[j]=yfold[l]
215
216     Nf=j-1
217     xf[0]=xf[Nf];yf[0]=yf[Nf];xf[Nf+1]=xf[1];yf[Nf+1]=yf[1]
218
219     #----- Plot the results -----
220
221     time+=dt # plot the results
222     uu[0:nx+1,0:ny+1]=0.5*(u[0:nx+1,1:ny+2]+u[0:nx+1,0:ny+1])
223     vv[0:nx+1,0:ny+1]=0.5*(v[1:nx+2,0:ny+1]+v[0:nx+1,0:ny+1])
224
225     plt.cla()
226     plt.contour(x[1:nx+1],y[1:ny+1],chi.T[1:nx+1,1:ny+1])
227     plt.quiver(xh[:,yh:],uu.T[:,:],vv.T[:,:])
228
229     plt.plot(xf[0:Nf],yf[0:Nf], 'k',linewidth=3)
230     plt.pause(0.0001)
231
232     print('time.elapsed= %s' % (time0.time() - time_start) )
233
234     # End of time step

```

Listing 4: Python Code 2B

```

1  #=====
2  # CodeC2-frt.m
3  # A very simple Navier–Stokes solver for a drop falling in a
4  # rectangular box, using a conservative form of the equations.
5  # A first–order explicit projection method and centered in space
6  # discretization are used. The marker function is advected by
7  # Tront Tracking.
8  # Original Matlab code by Gretar Tryggvason
9  # Python code converted by Tingyi Lu on 7/25/2018
10 #=====
11
12 import numpy as np
13 import matplotlib.pyplot as plt
14 import time as time0
15
16 Lx=1.0;Ly=1.0;gx=0.0;gy=-100.0 # Domain size and
17 rho1=1.0; rho2=2.0; m0=0.01 # physical variables
18 unorth=0;usouth=0;veast=0;vwest=0;time=0.0
19 rad=0.15;xc=0.5;yc=0.7 # Initial drop size and location
20
21 #----- Numerical variables -----
22 nx=32;ny=32;dt=0.00125;nstep=400; maxit=200;maxError=0.001;beta=1.5;
23 Nf=100; Maxf=2000
24
25 #----- Zero various arrays -----
26 u=np.zeros((nx+1,ny+2)); v=np.zeros((nx+2,ny+1)); p=np.zeros((nx+2,ny+2))
27 ut=np.zeros((nx+1,ny+2)); vt=np.zeros((nx+2,ny+1)); tmp1=np.zeros((nx+2,ny+1))
28 uu=np.zeros((nx+1,ny+1)); vv=np.zeros((nx+1,ny+1)); tmp2=np.zeros((nx+2,ny+2))
29 r=np.zeros((nx+2,ny+2)); chi=np.zeros((nx+2,ny+2))
30 d=np.zeros((nx+1,nx+1))
31 xf=np.zeros(Maxf); yf=np.zeros(Maxf)
32 uf=np.zeros(Maxf); vf=np.zeros(Maxf)
33
34 dx=Lx/nx;dy=Ly/ny # Set the grid
35 x=np.linspace(-.5*dx, (nx+0.5)*dx, nx+2)
36 y=np.linspace(-.5*dx, (ny+0.5)*dy, ny+2)
37 xh=np.linspace(0, Lx, nx+1)
38 yh=np.linspace(0, Ly, ny+1)
39
40
41 #----- Initial Conditions -----
42 r[:,]=rho1 # Set density
43
44 for i in range(1,nx+1): # for the domain and the drop
45     for j in range(1,ny+1):
46         if((x[i]-xc)**2+(y[j]-yc)**2 < rad**2):
47             r[i,j]=rho2
48             chi[i,j]=1.0
49
50
51 for l in range(Nf+2):
52     xf[l]=xc-rad*np.sin(2.0*np.pi*l/Nf) # Initialize
53     yf[l]=yc+rad*np.cos(2.0*np.pi*l/Nf) # the Front
54
55
56 plt.plot(xf[0:Nf],yf[0:Nf],'k',linewidth=3)
57 plt.axis('scaled')
58 plt.axis([0,Lx,0,Ly], aspect=1)
59 plt.pause(0.0001)
60
61 time.start = time0.time()
62
63 #----- START TIME LOOP -----
64 for istep in range(nstep):
65     print(istep)
66
67

```

```

68 #----- Advect the Front -----
69
70 for l in range(1,Nf+1):           # Interpolate the Front Velocities
71     ip=np.int(xf[l]/dx); jp=np.int((yf[l]+0.5*dy)/dy)
72     ax=xf[l]/dx-ip; ay=(yf[l]+0.5*dy)/dy-jp
73     uf[l]=((1.0-ax)*(1.0-ay)*u[ip,jp]+ax*(1.0-ay)*u[ip+1,jp]+
74             (1.0-ax)*ay*u[ip,jp+1]+ax*ay*u[ip+1,jp+1])
75
76     ip=np.int((xf[l]+0.5*dx)/dx); jp=np.int(yf[l]/dy)
77     ax=(xf[l]+0.5*dx)/dx-ip; ay=yf[l]/dy-jp
78     vf[l]=((1.0-ax)*(1.0-ay)*v[ip,jp]+ax*(1.0-ay)*v[ip+1,jp]+
79             (1.0-ax)*ay*v[ip,jp+1]+ax*ay*v[ip+1,jp+1])
80
81 xf[1:Nf+1]+=dt*uf[1:Nf+1]
82 yf[1:Nf+1]+=dt*vf[1:Nf+1] # Move the
83 xf[0]=xf[Nf];yf[0]=yf[Nf];xf[Nf+1]=xf[1];yf[Nf+1]=yf[1] # Front
84
85
86 #----- Update the marker function -----
87 d[:,:]=2
88
89 for l in range(1,Nf+1):
90     nfx=-(yf[l+1]-yf[l])/dx
91     nfy=(xf[l+1]-xf[l])/dy # Normal vector
92     ds=np.sqrt(nfx*nfx+nfy*nfy); nfx=nfx/ds; nfy=nfy/ds
93     xfront=0.5*(xf[l]+xf[l+1]); yfront=0.5*(yf[l]+yf[l+1])
94     ip=np.int((xfront+0.5*dx)/dx); jp=np.int((yfront+0.5*dy)/dy)
95
96     d1=np.sqrt(((xfront-x[ip])/dx)**2+((yfront-y[jp])/dy)**2)
97     d2=np.sqrt(((xfront-x[ip+1])/dx)**2+((yfront-y[jp])/dy)**2)
98     d3=np.sqrt(((xfront-x[ip+1])/dx)**2+((yfront-y[jp+1])/dy)**2)
99     d4=np.sqrt(((xfront-x[ip])/dx)**2+((yfront-y[jp+1])/dy)**2)
100
101     if d1<d[ip,jp]:
102         d[ip,jp]=d1.copy()
103         dn1=(x[ip]-xfront)*nfx/dx+(y[jp]-yfront)*nfy/dy
104         chi[ip,jp]=0.5*(1.0+np.sign(dn1))
105         if abs(dn1)<0.5:
106             chi[ip,jp]=0.5+dn1
107
108     if d2<d[ip+1,jp]:
109         d[ip+1,jp]=d2.copy()
110         dn2=(x[ip+1]-xfront)*nfx/dx+(y[jp]-yfront)*nfy/dy
111         chi[ip+1,jp]=0.5*(1.0+np.sign(dn2))
112         if abs(dn2)<0.5:
113             chi[ip+1,jp]=0.5+dn2
114
115     if d3<d[ip+1,jp+1]:
116         d[ip+1,jp+1]=d3.copy()
117         dn3=(x[ip+1]-xfront)*nfx/dx+(y[jp+1]-yfront)*nfy/dy
118         chi[ip+1,jp+1]=0.5*(1.0+np.sign(dn3))
119         if abs(dn3)<0.5:
120             chi[ip+1,jp+1]=0.5+dn3
121
122     if d4<d[ip,jp+1]:
123         d[ip,jp+1]=d4.copy()
124         dn4=(x[ip]-xfront)*nfx/dx+(y[jp+1]-yfront)*nfy/dy
125         chi[ip,jp+1]= 0.5*(1.0+np.sign(dn4))
126         if abs(dn4)<0.5:
127             chi[ip,jp+1]= 0.5+dn4
128
129
130 #----- Update the density -----
131
132 ro=r.copy()
133 r = rho1+(rho2-rho1)*chi
134
135 #----- Set tangential velocity at boundaries -----

```

```

136 u[:,0]=2*usouth-u[:,1];u[:,ny+1]=2*unorth-u[:,ny]
137 v[0,:]=2*vwest-v[1,:];v[nx+1,:]=2*veast-v[nx,:]
138
139 #----- Find the predicted velocities -----
140 # Temporary u-velocity
141 ut[1:nx,1:ny+1]=((2.0/(r[2:nx+1,1:ny+1]+r[1:nx,1:ny+1]))*(0.5*
142 (ro[2:nx+1,1:ny+1]+ro[1:nx,1:ny+1])*u[1:nx,1:ny+1]+ dt* (
143 -(0.25/dx)*(ro[2:nx+1,1:ny+1]*(u[2:nx+1,1:ny+1]+u[1:nx,1:ny+1])**2-
144 ro[1:nx,1:ny+1]*(u[1:nx,1:ny+1]+u[nx-1,1:ny+1])**2)
145 -(0.0625/dy)*( (ro[1:nx,1:ny+1]+ro[2:nx+1,1:ny+1]+ro[1:nx,2:ny+2]+ro[2:nx+1,2:ny+2])*
146 (u[1:nx,2:ny+2]+u[1:nx,1:ny+1])*(v[2:nx+1,1:ny+1]+v[1:nx,1:ny+1])
147 -(ro[1:nx,1:ny+1]+ro[2:nx+1,1:ny+1]+ro[2:nx+1,:ny]+ro[1:nx,:ny])*(u[1:nx,1:ny+1]
148 +u[1:nx,:ny])*(v[2:nx+1,:ny]+v[1:nx,:ny]))
149 +m0*((u[2:nx+1,1:ny+1]-2*u[1:nx,1:ny+1]+u[nx-1,1:ny+1])/dx**2+
150 (u[1:nx,2:ny+2]-2*u[1:nx,1:ny+1]+u[1:nx,:ny])/dy**2)
151 + 0.5*(ro[2:nx+1,1:ny+1]+ro[1:nx,1:ny+1])*gx ) )
152
153 # Temporary v-velocity
154 vt[1:nx+1,1:ny]=((2.0/(r[1:nx+1,2:ny+1]+r[1:nx+1,1:ny]))*(0.5*
155 (ro[1:nx+1,2:ny+1]+ro[1:nx+1,1:ny])*v[1:nx+1,1:ny]+ dt* (
156 -(0.0625/dx)*( (ro[1:nx+1,1:ny]+ro[2:nx+2,1:ny]+ro[2:nx+2,2:ny+1]+ro[1:nx+1,2:ny+1])*
157 (u[1:nx+1,1:ny]+u[1:nx+1,2:ny+1])*(v[1:nx+1,1:ny]+v[2:nx+2,1:ny])
158 -(ro[1:nx+1,1:ny]+ro[1:nx+1,2:ny+1]+ro[:nx,2:ny+1]+ro[:nx,1:ny])*
159 (u[:nx,2:ny+1]+u[:nx,1:ny])*(v[1:nx+1,1:ny]+v[:nx,1:ny]) )
160 -(0.25/dy)*(ro[1:nx+1,2:ny+1]*(v[1:nx+1,2:ny+1]+v[1:nx+1,1:ny])**2-
161 ro[1:nx+1,1:ny]*(v[1:nx+1,1:ny]+v[1:nx+1,:ny-1])**2 )
162 +m0*((v[2:nx+2,1:ny]-2*v[1:nx+1,1:ny]+v[:nx,1:ny])/dx**2+
163 (v[1:nx+1,2:ny+1]-2*v[1:nx+1,1:ny]+v[1:nx+1,:ny-1])/dy**2)
164 + 0.5*(ro[1:nx+1,2:ny+1]+ro[1:nx+1,1:ny])*gy ) )
165
166 #----- Solve the Pressure Equation -----
167
168
169 rt=r.copy(); lrg=1000 # Compute source term and the coefficient for p(i,j)
170 rt[:,0]=lrg;rt[:,ny+1]=lrg
171 rt[0,:]=lrg;rt[nx+1,:]=lrg
172
173
174 tmp1[1:nx+1,1:ny+1]=((0.5/dt)*( (ut[1:nx+1,1:ny+1]-ut[0:nx,1:ny+1])/dx+
175 (vt[1:nx+1,1:ny+1]-vt[1:nx+1,0:ny])/dy ) )
176 tmp2[1:nx+1,1:ny+1]=(1.0/( (1./dx)*(1./(dx*(rt[2:nx+2,1:ny+1]+rt[1:nx+1,1:ny+1]))+
177 1./(dx*(rt[0:nx,1:ny+1]+rt[1:nx+1,1:ny+1])) )+
178 (1./dy)*(1./(dy*(rt[1:nx+1,2:ny+2]+rt[1:nx+1,1:ny+1]))+
179 1./(dy*(rt[1:nx+1,0:ny]+rt[1:nx+1,1:ny+1])) ) ) )
180
181 for it in range(maxit): # Solve for pressure by SOR
182     oldArray=p.copy()
183     #Red & Black SOR
184     for ipass in range(2):
185         rb = ipass
186         p[1+rb:nx+1:2,1:ny+1:2] = ( (1.0-beta)*p[1+rb:nx+1:2,1:ny+1:2] + beta*tmp2[1+rb:nx+1:2,1:ny+1:2]*(
187             (1.0/dx/dx)*( p[2+rb:nx+2:2,1:ny+1:2]/(rt[2+rb:nx+2:2,1:ny+1:2]+rt[1+rb:nx+1:2,1:ny+1:2])
188             +p[rb:nx:2,1:ny+1:2]/(rt[rb:nx:2,1:ny+1:2]+rt[1+rb:nx+1:2,1:ny+1:2]))
189             +(1.0/dy/dy)*( p[1+rb:nx+1:2,2:ny+2:2]/(rt[1+rb:nx+1:2,2:ny+2:2]+rt[1+rb:nx+1:2,1:ny+1:2])
190             +p[1+rb:nx+1:2,0:ny:2]/(rt[1+rb:nx+1:2,0:ny:2]+rt[1+rb:nx+1:2,1:ny+1:2]))
191             - tmp1[1+rb:nx+1:2,1:ny+1:2] ) )
192
193         rb=1-ipass
194         p[1+rb:nx+1:2,2:ny+1:2] = ( (1.0-beta)*p[1+rb:nx+1:2,2:ny+1:2] + beta*tmp2[1+rb:nx+1:2,2:ny+1:2]*(
195             (1.0/dx/dx)*( p[2+rb:nx+2:2,2:ny+1:2]/(rt[2+rb:nx+2:2,2:ny+1:2]+rt[1+rb:nx+1:2,2:ny+1:2])
196             +p[rb:nx:2,2:ny+1:2]/(rt[rb:nx:2,2:ny+1:2]+rt[1+rb:nx+1:2,2:ny+1:2]))
197             +(1.0/dy/dy)*( p[1+rb:nx+1:2,3:ny+2:2]/(rt[1+rb:nx+1:2,3:ny+2:2]+rt[1+rb:nx+1:2,2:ny+1:2])
198             +p[1+rb:nx+1:2,1:ny:2]/(rt[1+rb:nx+1:2,1:ny:2]+rt[1+rb:nx+1:2,2:ny+1:2]))
199             - tmp1[1+rb:nx+1:2,2:ny+1:2] ) )
200
201 p[0,:]=p[1,:]; p[nx+1,:]=p[nx,:]
202 p[:,0]=p[:,1]; p[:,ny+1]=p[:,ny]
203

```



```

204     if (np.abs(oldArray-p)).max() < maxError:
205         break
206
207         # Correct the u-velocity
208     u[1:nx,1:ny+1]=ut[1:nx,1:ny+1]-dt*(2.0/dx)*(p[2:nx+1,1:ny+1]-p[1:nx,1:ny+1])/(r[2:nx+1,1:ny+1]+r[1:nx,1:ny+1])
209     # Correct the v-velocity
210     v[1:nx+1,1:ny]=vt[1:nx+1,1:ny]-dt*(2.0/dy)*(p[1:nx+1,2:ny+1]-p[1:nx+1,1:ny])/(r[1:nx+1,2:ny+1]+r[1:nx+1,1:ny])
211
212
213     #----- Add and delete points in the Front -----
214     xfold=xf.copy();yfold=yf.copy(); j=0
215     for l in range(1,Nf+1):
216         ds=np.sqrt( ((xfold[l]-xf[j])/dx)**2 + ((yfold[l]-yf[j])/dy)**2)
217         if ds > 0.5:
218             j=j+1;xf[j]=0.5*(xfold[l]+xf[j-1]);yf[j]=0.5*(yfold[l]+yf[j-1])
219             j=j+1;xf[j]=xfold[l];yf[j]=yfold[l]
220         elif 0.25<=ds<=0.5:
221             j=j+1;xf[j]=xfold[l];yf[j]=yfold[l]
222
223     Nf=j-1
224     xf[0]=xf[Nf];yf[0]=yf[Nf];xf[Nf+1]=xf[1];yf[Nf+1]=yf[1]
225
226     #----- Plot the results -----
227
228     time+=dt           # plot the results
229     uu[0:nx+1,0:ny+1]=0.5*(u[0:nx+1,1:ny+2]+u[0:nx+1,0:ny+1])
230     vv[0:nx+1,0:ny+1]=0.5*(v[1:nx+2,0:ny+1]+v[0:nx+1,0:ny+1])
231
232     plt.cla()
233     plt.contour(x[1:nx+1],y[1:ny+1],chi.T[1:nx+1,1:ny+1])
234     plt.quiver(xh[:,yh:],uu.T[:,:],vv.T[:,:])
235
236     plt.plot(xf[0:Nf],yf[0:Nf],'k',linewidth=3)
237     plt.pause(0.0001)
238
239     print('time.elapsed= %s' % (time0.time() - time_start) )
240
241     # End of time step

```

Listing 5: Python Code 3A

```

1  #=====
2  # CodeC3-frt-st-RK3.m
3  # A very simple Navier-Stokes solver for a drop falling in a
4  # rectangular box, using a conservative form of the equations.
5  # A 3-order explicit projection method and centered in space
6  # discretization are used. The density is advected by a front
7  # tracking scheme and surface tension and variable viscosity is
8  # included. This version uses a simple method to create the
9  # marker function.
10 # Original Matlab code by Gretar Tryggvason
11 # Python code converted by Tingyi Lu on 7/25/2018
12 #=====
13
14 import numpy as np
15 import matplotlib.pyplot as plt
16 import time as time0
17
18 Lx=1.0;Ly=1.0;gx=0.0;gy=-100.0; sigma=10 # Domain size and
19 rho1=1.0; rho2=2.0; m1=0.01; m2=0.02 # physical variables
20 unorth=0; usouth=0; veast=0; vwest=0; time=0.0
21 rad=0.15; xc=0.5; yc=0.7 # Initial drop size and location
22
23 #----- Numerical variables -----
24 nx=32;ny=32;dt=0.001;nstep=400; maxit=200;maxError=0.001;beta=1.5; Nf=100; Maxf=2000
25
26 #----- Zero various arrays -----
27 u=np.zeros((nx+1,ny+2)); v=np.zeros((nx+2,ny+1)); p=np.zeros((nx+2,ny+2))
28 ut=np.zeros((nx+1,ny+2)); vt=np.zeros((nx+2,ny+1)); tmp1=np.zeros((nx+2,ny+2))
29 uu=np.zeros((nx+1,ny+1)); vv=np.zeros((nx+1,ny+1)); tmp2=np.zeros((nx+2,ny+2))
30 fx=np.zeros((nx+2,ny+2)); fy=np.zeros((nx+2,ny+2)); r=np.zeros((nx+2,ny+2))
31 r=np.zeros((nx+2,ny+2)); chi=np.zeros((nx+2,ny+2))
32 m=np.zeros((nx+2,ny+2)); d=np.zeros((nx+2,ny+2))
33 xf=np.zeros(Maxf); yf=np.zeros(Maxf)
34 uf=np.zeros(Maxf); vf=np.zeros(Maxf)
35 tx=np.zeros(Maxf); ty=np.zeros(Maxf)
36 un=np.zeros((nx+1,ny+2)); vn=np.zeros((nx+2,ny+1)) # Used for
37 rn=np.zeros((nx+2,ny+2)); mn=np.zeros((nx+2,ny+2)) # higher order
38 xfn=np.zeros(Maxf); yfn=np.zeros(Maxf) # in time
39 Area=np.zeros(nstep);CentroidX=np.zeros(nstep);CentroidY=np.zeros(nstep)
40 Time1=np.zeros(nstep)
41
42
43 dx=Lx/nx;dy=Ly/ny # Set the grid
44 x=np.linspace(-.5*dx, (nx+0.5)*dx, nx+2)
45 y=np.linspace(-.5*dx, (ny+0.5)*dy, ny+2)
46 xh=np.linspace(0, Lx, nx+1)
47 yh=np.linspace(0, Ly, ny+1)
48
49 #----- Initial Conditions -----
50 r[:,:]=rho1;m[:,:]=m1 # Set density and viscosity
51
52 for i in range(1,nx+1): # for the domain and the drop
53     for j in range(1,ny+1):
54         if((x[i]-xc)**2+(y[j]-yc)**2 < rad**2):
55             r[i,j]=rho2
56             m[i,j]=m2
57             chi[i,j]=1.0
58
59 for l in range(Nf+2):
60     xf[l]=xc-rad*np.sin(2.0*np.pi*l/Nf) # Initialize
61     yf[l]=yc+rad*np.cos(2.0*np.pi*l/Nf) # the Front
62
63 plt.plot(xf[0:Nf],yf[0:Nf],'k',linewidth=3)
64 plt.axis('scaled')
65 plt.axis([0,Lx,0,Ly], aspect=1)
66 plt.pause(0.0001)
67

```

```

68 time_start = time0.time()
69
70 #----- START TIME LOOP -----
71 for istep in range(nstep):
72     print(istep)
73     un=u.copy(); vn=v.copy(); rn=r.copy(); mn=m.copy(); xfn=xf.copy(); yfn=yf.copy() # Higher order
74     for substep in range(3): # in time
75
76     #----- Advect the Front -----
77     for l in range(1,Nf+1): # Interpolate the Front Velocities
78         ip=np.int(xf[l]/dx); jp=np.int((yf[l]+0.5*dy)/dy)
79         ax=xf[l]/dx-ip; ay=(yf[l]+0.5*dy)/dy-jp
80         uf[l]=((1.0-ax)*(1.0-ay)*u[ip,jp]+ax*(1.0-ay)*u[ip+1,jp]+
81             (1.0-ax)*ay*u[ip,jp+1]+ax*ay*u[ip+1,jp+1])
82
83         ip=np.int((xf[l]+0.5*dx)/dx); jp=np.int(yf[l]/dy)
84         ax=(xf[l]+0.5*dx)/dx-ip; ay=yf[l]/dy-jp
85         vf[l]=((1.0-ax)*(1.0-ay)*v[ip,jp]+ax*(1.0-ay)*v[ip+1,jp]+
86             (1.0-ax)*ay*v[ip,jp+1]+ax*ay*v[ip+1,jp+1])
87
88     for l in range(1,Nf+1):
89         xf[l]=xf[l]+dt*uf[l]; yf[l]=yf[l]+dt*vf[l]
90     xf[0]=xf[Nf]; yf[0]=yf[Nf]; xf[Nf+1]=xf[1]; yf[Nf+1]=yf[1]
91
92     #----- Update the marker function -----
93     d[:,:] = 2
94
95     for l in range(1,Nf+1):
96         nfx=-(yf[l+1]-yf[l])/dx
97         nfy=(xf[l+1]-xf[l])/dy # Normal vector
98         ds=np.sqrt(nfx*nfx+nfy*nfy); nfx=nfx/ds; nfy=nfy/ds
99         xfront=0.5*(xf[l]+xf[l+1]); yfront=0.5*(yf[l]+yf[l+1])
100         ip=np.int((xfront+0.5*dx)/dx); jp=np.int((yfront+0.5*dy)/dy)
101
102         d1=np.sqrt(((xfront-x[ip])/dx)**2+((yfront-y[jp])/dy)**2)
103         d2=np.sqrt(((xfront-x[ip+1])/dx)**2+((yfront-y[jp])/dy)**2)
104         d3=np.sqrt(((xfront-x[ip+1])/dx)**2+((yfront-y[jp+1])/dy)**2)
105         d4=np.sqrt(((xfront-x[ip])/dx)**2+((yfront-y[jp+1])/dy)**2)
106
107         if d1<d[ip,jp]:
108             d[ip,jp]=d1.copy()
109             dn1=(x[ip]-xfront)*nfx/dx+(y[jp]-yfront)*nfy/dy
110             chi[ip,jp]=0.5*(1.0+np.sign(dn1))
111             if abs(dn1)<0.5:
112                 chi[ip,jp]=0.5+dn1
113
114         if d2<d[ip+1,jp]:
115             d[ip+1,jp]=d2.copy()
116             dn2=(x[ip+1]-xfront)*nfx/dx+(y[jp]-yfront)*nfy/dy
117             chi[ip+1,jp]=0.5*(1.0+np.sign(dn2))
118             if abs(dn2)<0.5:
119                 chi[ip+1,jp]=0.5+dn2
120
121         if d3<d[ip+1,jp+1]:
122             d[ip+1,jp+1]=d3.copy()
123             dn3=(x[ip+1]-xfront)*nfx/dx+(y[jp+1]-yfront)*nfy/dy
124             chi[ip+1,jp+1]=0.5*(1.0+np.sign(dn3))
125             if abs(dn3)<0.5:
126                 chi[ip+1,jp+1]=0.5+dn3
127
128         if d4<d[ip,jp+1]:
129             d[ip,jp+1]=d4.copy()
130             dn4=(x[ip]-xfront)*nfx/dx+(y[jp+1]-yfront)*nfy/dy
131             chi[ip,jp+1]=0.5*(1.0+np.sign(dn4))
132             if abs(dn4)<0.5:
133                 chi[ip,jp+1]=0.5+dn4
134
135     #----- Update the density -----

```

```

136     ro=r.copy()
137     r = rho1+(rho2-rho1)*chi
138
139 #----- Find surface tension -----
140     fx=np.zeros((nx+2,ny+2));fy=np.zeros((nx+2,ny+2)) # Set fx & fy to zero
141
142     for l in range(Nf+1):
143         ds=np.sqrt((xf[l+1]-xf[l])**2+(yf[l+1]-yf[l])**2)
144         tx[l]=(xf[l+1]-xf[l])/ds
145         ty[l]=(yf[l+1]-yf[l])/ds # Tangent vectors
146
147     tx[Nf+1]=tx[1];ty[Nf+1]=ty[1]
148
149     for l in range(1,Nf+1): # Distribute to the fixed grid
150         nfx=sigma*(tx[l]-tx[l-1]);nfy=sigma*(ty[l]-ty[l-1])
151
152         ip=np.int(xf[l]/dx); jp=np.int((yf[l]+0.5*dy)/dy)
153         ax=xf[l]/dx-ip; ay=(yf[l]+0.5*dy)/dy-jp
154         fx[ip,jp] +=(1.0-ax)*(1.0-ay)*nfx/dx/dy
155         fx[ip+1,jp] +=ax*(1.0-ay)*nfx/dx/dy
156         fx[ip,jp+1] +=(1.0-ax)*ay*nfx/dx/dy
157         fx[ip+1,jp+1] +=ax*ay*nfx/dx/dy
158
159         ip=np.int((xf[l]+0.5*dx)/dx); jp=np.int(yf[l]/dy)
160         ax=(xf[l]+0.5*dx)/dx-ip; ay=yf[l]/dy-jp
161         fy[ip,jp] +=(1.0-ax)*(1.0-ay)*nfy/dx/dy
162         fy[ip+1,jp] +=ax*(1.0-ay)*nfy/dx/dy
163         fy[ip,jp+1] +=(1.0-ax)*ay*nfy/dx/dy
164         fy[ip+1,jp+1] +=ax*ay*nfy/dx/dy
165
166         fx[:,1]=fx[:,1]+fx[:,0] # Correct boundary
167         fx[:,ny]=fx[:,ny]+fx[:,ny+1] # values for the
168         fy[1,:]=fy[1,:]+fy[0,:] # surface force
169         fy[nx,:]=fy[nx,:]+fy[nx+1,:] # on the grid
170
171 #----- Set tangential velocity at boundaries -----
172     u[:,0]=2*usouth-u[:,1];u[:,ny+1]=2*unorth-u[:,ny]
173     v[0,:]=2*vwest-v[1,:];v[nx+1,:]=2*veast-v[nx,:]
174
175
176 #----- Find the predicted velocities -----
177     for i in range(1,nx):
178         for j in range(1,ny+1): # Temporary u-velocity-advection
179             ut[i,j]=((2.0/(r[i+1,j]+r[i,j]))*(0.5*(ro[i+1,j]+ro[i,j])*u[i,j]+ dt* (
180                 -(0.25/dx)*(ro[i+1,j]*(u[i+1,j]+u[i,j]))**2-ro[i,j]*(u[i,j]+u[i-1,j]))**2)
181                 -(0.0625/dy)*( (ro[i,j]+ro[i+1,j]+ro[i,j+1]+ro[i+1,j+1])*
182                     (u[i,j+1]+u[i,j])*(v[i+1,j]+v[i,j])
183                     -(ro[i,j]+ro[i+1,j]+ro[i+1,j-1]+ro[i,j-1])*(u[i,j]
184                     +u[i,j-1])*(v[i+1,j-1]+v[i,j-1]))
185                     + 0.5*(ro[i+1,j]+ro[i,j])*gx + fx[i,j]) ))
186
187     for i in range(1,nx+1):
188         for j in range(1,ny): # Temporary v-velocity-advection
189             vt[i,j]=((2.0/(r[i,j+1]+r[i,j]))*(0.5*(ro[i,j+1]+ro[i,j])*v[i,j]+ dt* (
190                 -(0.0625/dx)*( (ro[i,j]+ro[i+1,j]+ro[i+1,j+1]+ro[i,j+1])*
191                     (u[i,j]+u[i,j+1])*(v[i,j]+v[i+1,j])
192                     -(ro[i,j]+ro[i,j+1]+ro[i-1,j+1]+ro[i-1,j])*(
193                     (u[i-1,j+1]+u[i-1,j])*(v[i,j]+v[i-1,j]) )
194                     -(0.25/dy)*(ro[i,j+1]*(v[i,j+1]+v[i,j]))**2-ro[i,j]*(v[i,j]+v[i,j-1]))**2 )
195                     + 0.5*(ro[i,j+1]+ro[i,j])*gy + fy[i,j]) ))
196
197     for i in range(1,nx):
198         for j in range(1,ny+1): # Temporary u-velocity-viscosity
199             ut[i,j]=(ut[i,j]+(2.0/(r[i+1,j]+r[i,j]))*dt*(
200                 +(1./dx)*2.*(m[i+1,j]*(1./dx)*(u[i+1,j]-u[i,j]) -
201                 m[i,j] *(1./dx)*(u[i,j]-u[i-1,j]) )
202                 +(1./dy)*( 0.25*(m[i,j]+m[i+1,j]+m[i+1,j+1]+m[i,j+1])*
203                 ((1./dy)*(u[i,j+1]-u[i,j]) + (1./dx)*(v[i+1,j]-v[i,j]) ) -

```

```

204         0.25*(m[i,j]+m[i+1,j]+m[i+1,j-1]+m[i,j-1])*
205         ((1./dy)*(u[i,j]-u[i,j-1])+(1./dx)*(v[i+1,j-1]-v[i,j-1]))))
206
207
208     for i in range(1,nx+1):
209         for j in range(1,ny): # Temporary v-velocity-viscosity
210             vt[i,j]=(vt[i,j]+(2.0/(r[i,j+1]+r[i,j]))*dt*(
211                 +(1./dx)*(0.25*(m[i,j]+m[i+1,j]+m[i+1,j+1]+m[i,j+1])*
212                 ((1./dy)*(u[i,j+1]-u[i,j])+(1./dx)*(v[i+1,j]-v[i,j]))-
213                 0.25*(m[i,j]+m[i,j+1]+m[i-1,j+1]+m[i-1,j])*
214                 ((1./dy)*(u[i-1,j+1]-u[i-1,j])+(1./dx)*(v[i,j]-v[i-1,j]))))
215                 +(1./dy)*2.*(m[i,j+1]*(1./dy)*(v[i,j+1]-v[i,j])-
216                 m[i,j]*(1./dy)*(v[i,j]-v[i,j-1]))))
217
218
219 #----- Solve the Pressure Equation -----
220     rt=r.copy(); lrg=1000 # Compute source term and the coefficient for p(i,j)
221     rt[:,0]=lrg;rt[:,ny+1]=lrg
222     rt[0,:]=lrg;rt[nx+1,:]=lrg
223
224     for i in range(1,nx+1):
225         for j in range(1,ny+1):
226             tmp1[i,j]= (0.5/dt)*((ut[i,j]-ut[i-1,j])/dx+(vt[i,j]-vt[i,j-1])/dy)
227             tmp2[i,j]=(1.0/((1./dx)*(1./(dx*(rt[i+1,j]+rt[i,j]))+
228                 1./(dx*(rt[i-1,j]+rt[i,j])))+(
229                 (1./dy)*(1./(dy*(rt[i,j+1]+rt[i,j]))+
230                 1./(dy*(rt[i,j-1]+rt[i,j])))))
231
232     for it in range(maxit): # Solve for pressure by Red-Black SOR
233         oldArray=p.copy()
234         for ipass in range(2):
235             rb = ipass
236             for j in range(1,ny+1):
237                 for i in range(1+rb, nx+1, 2):
238                     p[i,j] = ((1.0-beta)*p[i,j] + beta*tmp2[i,j]*
239                         (1.0/dx/dx)*((p[i+1,j]/(rt[i+1,j]+rt[i,j]))
240                         +p[i-1,j]/(rt[i-1,j]+rt[i,j]))
241                         +(1.0/dy/dy)*((p[i,j+1]/(rt[i,j+1]+rt[i,j]))
242                         +p[i,j-1]/(rt[i,j-1]+rt[i,j]))
243                         - tmp1[i,j]))
244             rb=1-rb
245
246         if (np.abs(oldArray-p)).max() < maxError:
247             break
248
249     for i in range(1,nx):
250         for j in range(1,ny+1): # Correct the u-velocity
251             u[i,j]=ut[i,j]-dt*(2.0/dx)*(p[i+1,j]-p[i,j])/(r[i+1,j]+r[i,j])
252
253     for i in range(1,nx+1):
254         for j in range(1,ny): # Correct the v-velocity
255             v[i,j]=vt[i,j]-dt*(2.0/dy)*(p[i,j+1]-p[i,j])/(r[i,j+1]+r[i,j])
256
257     for i in range(0,nx+2):
258         for j in range(0,ny+2): # Update the viscosity
259             m[i,j]=m1+(m2-m1)*chi[i,j]
260
261     if substep==1: # Higher order (RK-3) in time
262         u=0.75*un+0.25*u; v=0.75*vn+0.25*v; r=0.75*rn+0.25*r
263         m=0.75*mn+0.25*m; xf=0.75*xfn+0.25*xf; yf=0.75*yfn+0.25*yf
264     elif substep==2:
265         u=(1/3)*un+(2/3)*u; v=(1/3)*vn+(2/3)*v; r=(1/3)*rn+(2/3)*r;
266         m=(1/3)*mn+(2/3)*m; xf=(1/3)*xfn+(2/3)*xf; yf=(1/3)*yfn+(2/3)*yf;
267
268 #----- Add and delete points in the Front -----
269     xfold=xf.copy(); yfold=yf.copy(); j=0
270     for l in range(1,Nf+1):
271         ds=np.sqrt(((xfold[l]-xf[j])/dx)**2 + ((yfold[l]-yf[j])/dy)**2)

```

```

272     if ds > 0.5:
273         j=j+1;xf[j]=0.5*(xfold[l]+xf[j-1]);yf[j]=0.5*(yfold[l]+yf[j-1])
274         j=j+1;xf[j]=xfold[l];yf[j]=yfold[l]
275     elif 0.25<=ds<=0.5:
276         j=j+1;xf[j]=xfold[l];yf[j]=yfold[l]
277
278     Nf=j-1
279     xf[0]=xf[Nf];yf[0]=yf[Nf];xf[Nf+1]=xf[1];yf[Nf+1]=yf[1]
280
281     #----- Compute Diagnostic quantitties -----
282     Area[istep]=0; CentroidX[istep]=0; CentroidY[istep]=0; Time1[istep]=time
283
284     for l in range(1,Nf+1):
285         Area[istep]+=0.25*((xf[l+1]+xf[l])*(yf[l+1]-yf[l])-(yf[l+1]+yf[l])*(xf[l+1]-xf[l]))
286         CentroidX[istep]+=0.125*((xf[l+1]+xf[l])**2+(yf[l+1]+yf[l])**2)*(yf[l+1]-yf[l])
287         CentroidY[istep]-=0.125*((xf[l+1]+xf[l])**2+(yf[l+1]+yf[l])**2)*(xf[l+1]-xf[l])
288
289     CentroidX[istep]=CentroidX[istep]/Area[istep];CentroidY[istep]=CentroidY[istep]/Area[istep]
290
291     #----- Plot the results -----
292     time+=dt # plot the results
293     uu[0:nx+1,0:ny+1]=0.5*(u[0:nx+1,1:ny+2]+u[0:nx+1,0:ny+1])
294     vv[0:nx+1,0:ny+1]=0.5*(v[1:nx+2,0:ny+1]+v[0:nx+1,0:ny+1])
295
296
297     plt.cla()
298     plt.contour(x[1:nx+1],y[1:ny+1],chi.T[1:nx+1,1:ny+1])
299     plt.quiver(xh[:,yh:],uu.T[:,:],vv.T[:,:])
300
301     plt.plot(xf[0:Nf],yf[0:Nf], 'k', linewidth=3)
302     plt.axis([0,Lx,0,Ly], aspect=1)
303     plt.pause(0.0001)
304
305     print('time.elapsed= %s' % (time0.time() - time_start) )
306
307
308     #----- Extra commands for interactive processing -----
309     #plt.figure(2)
310     #plt.plot(Time1,Area,'r',linewidth=2)
311     #plt.axis([0,dt*nstep,0,0.1])
312     #plt.show()

```

Listing 6: Python Code 3B

```

1  #=====
2  # CodeC3-frt-st-RK3.m
3  # A very simple Navier-Stokes solver for a drop falling in a
4  # rectangular box, using a conservative form of the equations.
5  # A 3-order explicit projection method and centered in space
6  # discretization are used. The density is advected by a front
7  # tracking scheme and surface tension and variable viscosity is
8  # included. This version uses a simple method to create the
9  # marker function.
10 # Original Matlab code by Gretar Tryggvason
11 # Python code converted by Tingyi Lu on 7/25/2018
12 #=====
13
14 import numpy as np
15 import matplotlib.pyplot as plt
16 import time as time0
17
18 Lx=1.0;Ly=1.0;gx=0.0;gy=-100.0; sigma=10 # Domain size and
19 rho1=1.0; rho2=2.0; m1=0.01; m2=0.02 # physical variables
20 unorth=0; usouth=0; veast=0; vwest=0; time=0.0
21 rad=0.15; xc=0.5; yc=0.7 # Initial drop size and location
22
23 #----- Numerical variables -----
24 nx=32;ny=32;dt=0.001;nstep=400; maxit=200;maxError=0.001;beta=1.5; Nf=100; Maxf=2000
25
26 #----- Zero various arrays -----
27 u=np.zeros((nx+1,ny+2)); v=np.zeros((nx+2,ny+1)); p=np.zeros((nx+2,ny+2))
28 ut=np.zeros((nx+1,ny+2)); vt=np.zeros((nx+2,ny+1)); tmp1=np.zeros((nx+2,ny+2))
29 uu=np.zeros((nx+1,ny+1)); vv=np.zeros((nx+1,ny+1)); tmp2=np.zeros((nx+2,ny+2))
30 fx=np.zeros((nx+2,ny+2)); fy=np.zeros((nx+2,ny+2)); r=np.zeros((nx+2,ny+2))
31 r=np.zeros((nx+2,ny+2)); chi=np.zeros((nx+2,ny+2))
32 m=np.zeros((nx+2,ny+2)); d=np.zeros((nx+2,ny+2))
33 xf=np.zeros(Maxf); yf=np.zeros(Maxf)
34 uf=np.zeros(Maxf); vf=np.zeros(Maxf)
35 tx=np.zeros(Maxf); ty=np.zeros(Maxf)
36 un=np.zeros((nx+1,ny+2)); vn=np.zeros((nx+2,ny+1)) # Used for
37 rn=np.zeros((nx+2,ny+2)); mn=np.zeros((nx+2,ny+2)) # higher order
38 xfn=np.zeros(Maxf); yfn=np.zeros(Maxf) # in time
39 Area=np.zeros(nstep);CentroidX=np.zeros(nstep);CentroidY=np.zeros(nstep)
40 Time1=np.zeros(nstep)
41
42
43 dx=Lx/nx;dy=Ly/ny # Set the grid
44 x=np.linspace(-.5*dx, (nx+0.5)*dx, nx+2)
45 y=np.linspace(-.5*dx, (ny+0.5)*dy, ny+2)
46 xh=np.linspace(0, Lx, nx+1)
47 yh=np.linspace(0, Ly, ny+1)
48
49 #----- Initial Conditions -----
50 r[:,]=rho1;m[:,]=m1 # Set density and viscosity
51
52 for i in range(1,nx+1): # for the domain and the drop
53     for j in range(1,ny+1):
54         if((x[i]-xc)**2+(y[j]-yc)**2 < rad**2):
55             r[i,j]=rho2
56             m[i,j]=m2
57             chi[i,j]=1.0
58
59 for l in range(Nf+2):
60     xf[l]=xc-rad*np.sin(2.0*np.pi*l/Nf) # Initialize
61     yf[l]=yc+rad*np.cos(2.0*np.pi*l/Nf) # the Front
62
63 plt.plot(xf[0:Nf],yf[0:Nf],'k',linewidth=3)
64 plt.axis('scaled')
65 plt.axis([0,Lx,0,Ly], aspect=1)
66 plt.pause(0.0001)
67

```

```

68 time_start = time0.time()
69
70 #----- START TIME LOOP -----
71 for istep in range(nstep):
72     print(istep)
73     un=u.copy(); vn=v.copy(); rn=r.copy(); mn=m.copy(); xfn=xf.copy(); yfn=yf.copy() # Higher order
74     for substep in range(3): # in time
75
76     #----- Advect the Front -----
77     for l in range(1,Nf+1): # Interpolate the Front Velocities
78         ip=np.int(xf[l]/dx); jp=np.int((yf[l]+0.5*dy)/dy)
79         ax=xf[l]/dx-ip; ay=(yf[l]+0.5*dy)/dy-jp
80         uf[l]=((1.0-ax)*(1.0-ay)*u[ip,jp]+ax*(1.0-ay)*u[ip+1,jp]+
81             (1.0-ax)*ay*u[ip,jp+1]+ax*ay*u[ip+1,jp+1])
82
83         ip=np.int((xf[l]+0.5*dx)/dx); jp=np.int(yf[l]/dy)
84         ax=(xf[l]+0.5*dx)/dx-ip; ay=yf[l]/dy-jp
85         vf[l]=((1.0-ax)*(1.0-ay)*v[ip,jp]+ax*(1.0-ay)*v[ip+1,jp]+
86             (1.0-ax)*ay*v[ip,jp+1]+ax*ay*v[ip+1,jp+1])
87
88         xf[1:Nf+1]+=dt*uf[1:Nf+1]
89         yf[1:Nf+1]+=dt*vf[1:Nf+1] # Move the
90         xf[0]=xf[Nf];yf[0]=yf[Nf];xf[Nf+1]=xf[1];yf[Nf+1]=yf[1] # Front
91
92
93     #----- Update the marker function -----
94     d[:,:]=2
95
96     for l in range(1,Nf+1):
97         nfx=-(yf[l+1]-yf[l])/dx
98         nfy=(xf[l+1]-xf[l])/dy # Normal vector
99         ds=np.sqrt(nfx*nfx+nfy*nfy); nfx=nfx/ds; nfy=nfy/ds
100         xfront=0.5*(xf[l]+xf[l+1]); yfront=0.5*(yf[l]+yf[l+1])
101         ip=np.int((xfront+0.5*dx)/dx); jp=np.int((yfront+0.5*dy)/dy)
102
103         d1=np.sqrt(((xfront-x[ip])/dx)**2+((yfront-y[jp])/dy)**2)
104         d2=np.sqrt(((xfront-x[ip+1])/dx)**2+((yfront-y[jp])/dy)**2)
105         d3=np.sqrt(((xfront-x[ip+1])/dx)**2+((yfront-y[jp+1])/dy)**2)
106         d4=np.sqrt(((xfront-x[ip])/dx)**2+((yfront-y[jp+1])/dy)**2)
107
108         if d1<d[ip,jp]:
109             d[ip,jp]=d1.copy()
110             dn1=(x[ip]-xfront)*nfx/dx+(y[jp]-yfront)*nfy/dy
111             chi[ip,jp]=0.5*(1.0+np.sign(dn1))
112             if abs(dn1)<0.5:
113                 chi[ip,jp]=0.5+dn1
114
115         if d2<d[ip+1,jp]:
116             d[ip+1,jp]=d2.copy()
117             dn2=(x[ip+1]-xfront)*nfx/dx+(y[jp]-yfront)*nfy/dy
118             chi[ip+1,jp]=0.5*(1.0+np.sign(dn2))
119             if abs(dn2)<0.5:
120                 chi[ip+1,jp]=0.5+dn2
121
122         if d3<d[ip+1,jp+1]:
123             d[ip+1,jp+1]=d3.copy()
124             dn3=(x[ip+1]-xfront)*nfx/dx+(y[jp+1]-yfront)*nfy/dy
125             chi[ip+1,jp+1]=0.5*(1.0+np.sign(dn3))
126             if abs(dn3)<0.5:
127                 chi[ip+1,jp+1]=0.5+dn3
128
129         if d4<d[ip,jp+1]:
130             d[ip,jp+1]=d4.copy()
131             dn4=(x[ip]-xfront)*nfx/dx+(y[jp+1]-yfront)*nfy/dy
132             chi[ip,jp+1]=0.5*(1.0+np.sign(dn4))
133             if abs(dn4)<0.5:
134                 chi[ip,jp+1]=0.5+dn4
135

```



```

136 #----- Update the density -----
137 ro=r.copy()
138 r = rho1+(rho2-rho1)*chi
139
140 #----- Find surface tension -----
141 fx=np.zeros((nx+2,ny+2));fy=np.zeros((nx+2,ny+2)) # Set fx & fy to zero
142
143 for l in range(Nf+1):
144     ds=np.sqrt((xf[l+1]-xf[l])**2+(yf[l+1]-yf[l])**2)
145     tx[l]=(xf[l+1]-xf[l])/ds
146     ty[l]=(yf[l+1]-yf[l])/ds # Tangent vectors
147
148 tx[Nf+1]=tx[1];ty[Nf+1]=ty[1]
149
150 for l in range(1,Nf+1): # Distribute to the fixed grid
151     nfx=sigma*(tx[l]-tx[l-1]);nfy=sigma*(ty[l]-ty[l-1])
152
153     ip=np.int(xf[l]/dx); jp=np.int((yf[l]+0.5*dy)/dy)
154     ax=xf[l]/dx-ip; ay=(yf[l]+0.5*dy)/dy-jp
155     fx[ip,jp] +=(1.0-ax)*(1.0-ay)*nfx/dx/dy
156     fx[ip+1,jp] +=ax*(1.0-ay)*nfx/dx/dy
157     fy[ip,jp+1] +=(1.0-ax)*ay*nfy/dx/dy
158     fy[ip+1,jp+1] +=ax*ay*nfy/dx/dy
159
160     ip=np.int((xf[l]+0.5*dx)/dx); jp=np.int(yf[l]/dy)
161     ax=(xf[l]+0.5*dx)/dx-ip; ay=yf[l]/dy-jp
162     fy[ip,jp] +=(1.0-ax)*(1.0-ay)*nfy/dx/dy
163     fy[ip+1,jp] +=ax*(1.0-ay)*nfy/dx/dy
164     fy[ip,jp+1] +=(1.0-ax)*ay*nfy/dx/dy
165     fy[ip+1,jp+1] +=ax*ay*nfy/dx/dy
166
167 fx[:,1]=fx[:,1]+fx[:,0] # Correct boundary
168 fx[:,ny]=fx[:,ny]+fx[:,ny+1] # values for the
169 fy[1,:]=fy[1,:]+fy[0,:] # surface force
170 fy[nx,:]=fy[nx,:]+fy[nx+1,:] # on the grid
171 #----- Set tangential velocity at boundaries -----
172 u[:,0]=2*usouth-u[:,1];u[:,ny+1]=2*unorth-u[:,ny]
173 v[0,:]=2*vwest-v[1,:];v[nx+1,:]=2*veast-v[nx,:]
174
175
176 #----- Find the predicted velocities -----
177
178 # Temporary u-velocity-advection
179 ut[1:nx,1:ny+1]=( (2.0/(r[2:nx+1,1:ny+1]+r[1:nx,1:ny+1]))
180     *(0.5*(ro[2:nx+1,1:ny+1]+ro[1:nx,1:ny+1])*u[1:nx,1:ny+1] + dt*(
181     -(0.25/dx)*( ro[2:nx+1,1:ny+1]*(u[2:nx+1,1:ny+1]+u[1:nx,1:ny+1])**2
182     -ro[1:nx,1:ny+1]*(u[1:nx,1:ny+1]+u[nx-1,1:ny+1])**2)
183     -(0.0625/dy)*( ro[1:nx,1:ny+1]+ro[2:nx+1,1:ny+1]+ro[1:nx,2:ny+2]+ro[2:nx+1,2:ny+2])*
184     (u[1:nx,2:ny+2]+u[1:nx,1:ny+1])*(v[2:nx+1,1:ny+1]+v[1:nx,1:ny+1])
185     -(ro[1:nx,1:ny+1]+ro[2:nx+1,1:ny+1]+ro[2:nx+1,:ny]+ro[1:nx,:ny])*
186     (u[1:nx,1:ny+1]+u[1:nx,:ny])*(v[2:nx+1,:ny]+v[1:nx,:ny]))
187     + 0.5*(ro[2:nx+1,1:ny+1]+ro[1:nx,1:ny+1])*gx + fx[1:nx,1:ny+1]) )
188
189 # Temporary v-velocity-advection
190 vt[1:nx+1,1:ny]=((2.0/(r[1:nx+1,2:ny+1]+r[1:nx+1,1:ny]))*
191     (0.5*(ro[1:nx+1,2:ny+1]+ro[1:nx+1,1:ny])*v[1:nx+1,1:ny] + dt*(
192     -(0.0625/dx)*( ro[1:nx+1,1:ny]+ro[2:nx+2,1:ny]+ro[2:nx+2,2:ny+1]+ro[1:nx+1,2:ny+1])*
193     (u[1:nx+1,1:ny]+u[1:nx+1,2:ny+1])*(v[1:nx+1,1:ny]+v[2:nx+2,1:ny])
194     -(ro[1:nx+1,1:ny]+ro[1:nx+1,2:ny+1]+ro[nx,2:ny+1]+ro[nx,1:ny])*
195     (u[nx,2:ny+1]+u[nx,1:ny])*(v[1:nx+1,1:ny]+v[nx,1:ny]) )
196     -(0.25/dy)*(ro[1:nx+1,2:ny+1]*(v[1:nx+1,2:ny+1]+v[1:nx+1,1:ny])**2-
197     ro[1:nx+1,1:ny]*(v[1:nx+1,1:ny]+v[1:nx+1,:ny-1])**2 )
198     + 0.5*(ro[1:nx+1,2:ny+1]+ro[1:nx+1,1:ny])*gy + fy[1:nx+1,1:ny] ) )
199
200
201 # Temporary u-velocity-viscosity
202 ut[1:nx,1:ny+1]=(ut[1:nx,1:ny+1]+(2.0/(r[2:nx+1,1:ny+1]+r[1:nx,1:ny+1]))*dt*(
203     +(1./dx)*2*(m[2:nx+1,1:ny+1]*(1./dx)*(u[2:nx+1,1:ny+1]-u[1:nx,1:ny+1]) -

```

```

204         m[1:nx,1:ny+1] *(1./dx)*(u[1:nx,1:ny+1]-u[0:nx-1,1:ny+1]) )
205     +(1./dy)*( 0.25*(m[1:nx,1:ny+1]+m[2:nx+1,1:ny+1]+m[2:nx+1,2:ny+2]+m[1:nx,2:ny+2])*
206         ((1./dy)*(u[1:nx,2:ny+2]-u[1:nx,1:ny+1]) + (1./dx)*(v[2:nx+1,1:ny+1]-v[1:nx,1:ny+1]) ) -
207         0.25*(m[1:nx,1:ny+1]+m[2:nx+1,1:ny+1]+m[2:nx+1,0:ny]+m[1:nx,0:ny])*
208         ((1./dy)*(u[1:nx,1:ny+1]-u[1:nx,0:ny])+(1./dx)*(v[2:nx+1,0:ny]-v[1:nx,0:ny])) ) )
209
210     #Temporary v-velocity-viscosity
211     vt[1:nx+1,1:ny]=(vt[1:nx+1,1:ny]+(2.0/(r[1:nx+1,2:ny+1]+r[1:nx+1,1:ny]))*dt*(
212         +(1./dx)*( 0.25*(m[1:nx+1,1:ny]+m[2:nx+2,1:ny]+m[2:nx+2,2:ny+1]+m[1:nx+1,2:ny+1])*
213         ((1./dy)*(u[1:nx+1,2:ny+1]-u[1:nx+1,1:ny]) + (1./dx)*(v[2:nx+2,1:ny]-v[1:nx+1,1:ny]) ) -
214         0.25*(m[1:nx+1,1:ny]+m[1:nx+1,2:ny+1]+m[0:nx,2:ny+1]+m[0:nx,1:ny])*
215         ((1./dy)*(u[0:nx,2:ny+1]-u[0:nx,1:ny])+(1./dx)*(v[1:nx+1,1:ny]-v[0:nx,1:ny])) )
216         +(1./dy)*2.*(m[1:nx+1,2:ny+1]*(1./dy)*(v[1:nx+1,2:ny+1]-v[1:nx+1,1:ny]) -
217         m[1:nx+1,1:ny] *(1./dy)*(v[1:nx+1,1:ny]-v[1:nx+1,0:ny-1]) ) ) )
218
219     #----- Solve the Pressure Equation -----
220     rt=r.copy(); lrg=1000 # Compute source term and the coefficient for p(i,j)
221     rt[:,0]=lrg;rt[:,ny+1]=lrg
222     rt[0,:]=lrg;rt[nx+1,:]=lrg
223
224     tmp1[1:nx+1,1:ny+1]=((0.5/dt)*( (ut[1:nx+1,1:ny+1]-ut[0:nx,1:ny+1])/dx+
225         (vt[1:nx+1,1:ny+1]-vt[1:nx+1,0:ny])/dy))
226     tmp2[1:nx+1,1:ny+1]=(1.0/( (1./dx)*(1./dx*(rt[2:nx+2,1:ny+1]+rt[1:nx+1,1:ny+1]))+
227         1./dx*(rt[0:nx,1:ny+1]+rt[1:nx+1,1:ny+1])) )+
228         (1./dy)*(1./dy*(rt[1:nx+1,2:ny+2]+rt[1:nx+1,1:ny+1]))+
229         1./dy*(rt[1:nx+1,0:ny]+rt[1:nx+1,1:ny+1])) ) )
230
231
232     for it in range(maxit): # Solve for pressure by SOR
233         oldArray=p.copy()
234
235         #Red & Black SOR
236         for ipass in range(2):
237             rb = ipass
238             p[1+rb:nx+1:2,1:ny+1:2] = ( (1.0-beta)*p[1+rb:nx+1:2,1:ny+1:2] + beta*tmp2[1+rb:nx+1:2,1:ny+1:2]*(
239                 (1.0/dx/dx)*( p[2+rb:nx+2:2,1:ny+1:2]/(rt[2+rb:nx+2:2,1:ny+1:2]+rt[1+rb:nx+1:2,1:ny+1:2])
240                 +p[rb:nx:2,1:ny+1:2]/(rt[rb:nx:2,1:ny+1:2]+rt[1+rb:nx+1:2,1:ny+1:2]))
241                 +(1.0/dy/dy)*( p[1+rb:nx+1:2,2:ny+2:2]/(rt[1+rb:nx+1:2,2:ny+2:2]+rt[1+rb:nx+1:2,1:ny+1:2])
242                 +p[1+rb:nx+1:2,0:ny:2]/(rt[1+rb:nx+1:2,0:ny:2]+rt[1+rb:nx+1:2,1:ny+1:2]))
243                 - tmp1[1+rb:nx+1:2,1:ny+1:2] ) )
244
245             rb=1-ipass
246             p[1+rb:nx+1:2,2:ny+1:2] = ( (1.0-beta)*p[1+rb:nx+1:2,2:ny+1:2] + beta*tmp2[1+rb:nx+1:2,2:ny+1:2]*(
247                 (1.0/dx/dx)*( p[2+rb:nx+2:2,2:ny+1:2]/(rt[2+rb:nx+2:2,2:ny+1:2]+rt[1+rb:nx+1:2,2:ny+1:2])
248                 +p[rb:nx:2,2:ny+1:2]/(rt[rb:nx:2,2:ny+1:2]+rt[1+rb:nx+1:2,2:ny+1:2]))
249                 +(1.0/dy/dy)*( p[1+rb:nx+1:2,3:ny+2:2]/(rt[1+rb:nx+1:2,3:ny+2:2]+rt[1+rb:nx+1:2,2:ny+1:2])
250                 +p[1+rb:nx+1:2,1:ny:2]/(rt[1+rb:nx+1:2,1:ny:2]+rt[1+rb:nx+1:2,2:ny+1:2]))
251                 - tmp1[1+rb:nx+1:2,2:ny+1:2] ) )
252
253             p[0,:]=p[1,:]; p[nx+1,:]=p[nx,:]
254             p[:,0]=p[:,1]; p[:,ny+1]=p[:,ny]
255
256
257             if (np.abs(oldArray-p)).max() < maxError:
258                 break
259
260             # Correct the u-velocity
261             u[1:nx,1:ny+1]=ut[1:nx,1:ny+1]-dt*(2.0/dx)*(p[2:nx+1,1:ny+1]-p[1:nx,1:ny+1])/(r[2:nx+1,1:ny+1]+r[1:nx,1:ny+1])
262             # Correct the v-velocity
263             v[1:nx+1,1:ny]=vt[1:nx+1,1:ny]-dt*(2.0/dy)*(p[1:nx+1,2:ny+1]-p[1:nx+1,1:ny])/(r[1:nx+1,2:ny+1]+r[1:nx+1,1:ny])
264             #Update the viscosity
265             m[0:nx+1,0:ny+2]=m1+(m2-m1)*chi[0:nx+1,0:ny+2]
266
267
268         if substep==1: # Higher order (RK-3) in time
269             u=0.75*un+0.25*u; v=0.75*vn+0.25*v; r=0.75*rn+0.25*r
270             m=0.75*mn+0.25*m; xf=0.75*xfn+0.25*xf; yf=0.75*yfn+0.25*yf
271         elif substep==2:

```

```

272     u=(1/3)*un+(2/3)*u; v=(1/3)*vn+(2/3)*v; r=(1/3)*rn+(2/3)*r;
273     m=(1/3)*mn+(2/3)*m; xf=(1/3)*xn+(2/3)*xf; yf=(1/3)*yn+(2/3)*yf;
274
275     #----- Add and delete points in the Front -----
276     xfold=xf.copy();yfold=yf.copy(); j=0
277     for l in range(1,Nf+1):
278         ds=np.sqrt( ((xfold[l]-xf[j])/dx)**2 + ((yfold[l]-yf[j])/dy)**2)
279         if ds > 0.5:
280             j=j+1;xf[j]=0.5*(xfold[l]+xf[j-1]);yf[j]=0.5*(yfold[l]+yf[j-1])
281             j=j+1;xf[j]=xfold[l];yf[j]=yfold[l]
282         elif 0.25<=ds<=0.5:
283             j=j+1;xf[j]=xfold[l];yf[j]=yfold[l]
284
285     Nf=j-1
286     xf[0]=xf[Nf];yf[0]=yf[Nf];xf[Nf+1]=xf[1];yf[Nf+1]=yf[1]
287
288     #----- Compute Diagnostic quantities -----
289     Area[istep]=0; CentroidX[istep]=0; CentroidY[istep]=0; Time1[istep]=time
290
291     for l in range(1,Nf+1):
292         Area[istep]+=0.25*((xf[l+1]+xf[l])*(yf[l+1]-yf[l])-(yf[l+1]+yf[l])*(xf[l+1]-xf[l]))
293         CentroidX[istep]+=0.125*((xf[l+1]+xf[l])**2+(yf[l+1]+yf[l])**2)*(yf[l+1]-yf[l])
294         CentroidY[istep]-=0.125*((xf[l+1]+xf[l])**2+(yf[l+1]+yf[l])**2)*(xf[l+1]-xf[l])
295
296     CentroidX[istep]=CentroidX[istep]/Area[istep];CentroidY[istep]=CentroidY[istep]/Area[istep]
297
298     #----- Plot the results -----
299     time+=dt # plot the results
300     uu[0:nx+1,0:ny+1]=0.5*(u[0:nx+1,1:ny+2]+u[0:nx+1,0:ny+1])
301     vv[0:nx+1,0:ny+1]=0.5*(v[1:nx+2,0:ny+1]+v[0:nx+1,0:ny+1])
302
303     plt.cla()
304     plt.contour(x[1:nx+1],y[1:ny+1],chi.T[1:nx+1,1:ny+1])
305
306     plt.quiver(xh,yh,uu.T,vv.T)
307
308     plt.plot(xf[0:Nf],yf[0:Nf], 'k',linewidth=3)
309
310     plt.axis([0,Lx,0,Ly], aspect=1)
311     plt.pause(0.0001)
312
313     print('time.elapsed= %s' % (time0.time() - time_start) )
314
315     #----- Extra commands for interactive processing -----
316     #plt.figure(2)
317     #plt.plot(Time1,Area,'r',linewidth=2)
318     #plt.axis([0,dt*nstep,0,0.1])
319     #plt.show()

```