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 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 Code1 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 nth 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

Listing 1: Python Code 1A

```
#=================================================================
# CodeC1-advChi.py
# A very simple Navier-Stokes solver for a drop falling in a
# rectangular box, using a conservative form of the equations.
# A first-order explicit projection method and centered in space
# discretizationa are used. The marker function is advected by
# an advection diffusion equation.
# Original Matlab code by Gretar Tryggvason
# Python code converted by Tingyi Lu on 7/25/2018
#=================================================================

import numpy as np
import matplotlib.pyplot as plt
import time as time0

Lx=1.0;Ly=1.0;gx=0.0;gy=-100.0  # Domain size and
rho1=1.0; rho2=2.0; m0=0.01      # physical variables
unorth=0;usouth=0;veast=0;vwest=0;time=0.0
rad=0.15;xc=0.5;yc=0.7 # Initial drop size and location

#-------------------- Numerical variables ----------------------
nx=32;ny=32;dt=0.00125;nstep=400; maxit=200; maxError=0.001; beta=1.5

#-------------------- Zero various arrys -----------------------
u=np.zeros((nx+2,ny+3));  v=np.zeros((nx+3,ny+2));  p=np.zeros((nx+3,ny+3))
ut=np.zeros((nx+2,ny+3)); vt=np.zeros((nx+3,ny+2)); tmp1=np.zeros((nx+3,ny+3))
uu=np.zeros((nx+2,ny+2)); vv=np.zeros((nx+2,ny+2)); tmp2=np.zeros((nx+3,ny+3))
r=np.zeros((nx+3,ny+3)); chi=np.zeros((nx+3,ny+3))


dx=Lx/nx;dy=Ly/ny                         # Set the grid
x=np.linspace(-1.5*dx, (nx+0.5)*dx, nx+3)
y=np.linspace(-1.5*dy, (ny+0.5)*dy, ny+3)
xh=np.linspace(-dx, nx*dx, nx+2)
yh=np.linspace(-dy, ny*dy, ny+2)

X, Y = np.meshgrid(x,y)

#-------------------- Initial Conditions -----------------------
r = np.zeros((nx+3,nx+3))+rho1 # Set density

for i in range(nx+3):

    for j in range(ny+3):                      # for the domain and the drop
        if((x[i]-xc)**2+(y[j]-yc)**2 <rad**2):
            r[i,j] = rho2
            chi[i,j] = 1.0

plt.contour(X,Y,chi.T)
plt.axis('scaled')
plt.axis([0, Lx, 0, Ly], aspect=1)

time_start = time0.time()

#--------------------- START TIME LOOP ------------------------
for istep in range(nstep):
    print(istep)

    #--- ADVECT marker using centered difference plus diffusion ----
    chio=chi.copy()

    for i in range(2,nx+2):
        for j in range(2,ny+2):
            chi[i,j]= (chio[i,j]-(0.5*dt/dx)*(u[i,j]*(chio[i+1,j]
               +chio[i,j])-u[i-1,j]*(chio[i-1,j]+chio[i,j]))
               -(0.5* dt/dy)*(v[i,j]*(chio[i,j+1]
               +chio[i,j])-v[i,j-1]*(chio[i,j-1]+chio[i,j])  )
```

```
                +(m0*dt/dx/dx)*(chio[i+1,j]-2.0*chio[i,j]+chio[i-1,j])
                +(m0*dt/dy/dy)*(chio[i,j+1]-2.0*chio[i,j]+chio[i,j-1]) )

    #--------------------- Update the density -------------------------
    ro=r.copy()
    r = rho1 + (rho2-rho1)*chi

    #-------------- Set tangential velocity at boundaries ------------
    u[:,1]=2*usouth-u[:,2];u[:,ny+2]=2*unorth-u[:,ny+1]
    v[1,:]=2*vwest-v[2,:];v[nx+2,:]=2*veast-v[nx+1,:]

    #--------------- Find the predicted velocities -------------------
    # Temporary u-velocity

    for i in range(2,nx+1):
        for j in range(2,ny+2):       # Temporary u-velocity
            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* (
                      -(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)
                      -(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])
                              -(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]))
                       +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)
                            + 0.5*(ro[i+1,j]+ro[i,j])*gx ) ) )

     #Temporary v-velocity
    for i in range(2,nx+2):
        for j in range(2,ny+1):        # Temporary v-velocity
            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* (
               -(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])
                       - (ro[i,j]+ro[i,j+1]+ro[i-1,j+1]+ro[i-1,j])*
                            (u[i-1,j+1]+u[i-1,j])*(v[i,j]+v[i-1,j]) )
                 -(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 )
                 +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)
                            + 0.5*(ro[i,j+1]+ro[i,j])*gy  ) ) )

    #------------------ Solve the Pressure Equation -----------------
    rt=r.copy(); lrg=1e20   # Compute source term and the coefficient for p[i,j]
    rt[:,1]=lrg;rt[:,ny+2]=lrg
    rt[1,:]=lrg;rt[nx+2,:]=lrg

    for i in range(2,nx+2):
        for j in range(2,ny+2):
            tmp1[i,j]= (0.5/dt)*( (ut[i,j]-ut[i-1,j])/dx+(vt[i,j]-vt[i,j-1])/dy )
            tmp2[i,j]= (1.0/( (1./dx)*(1./(dx*(rt[i+1,j]+rt[i,j]))+
                            1./(dx*(rt[i-1,j]+rt[i,j])) )+
                      (1./dy)*(1./(dy*(rt[i,j+1]+rt[i,j]))+
                            1./(dy*(rt[i,j-1]+rt[i,j])) )   ))

    for it in range(maxit):                # Solve for pressure by SOR
        oldp=p.copy()

        #Red & Black SOR
        for ipass in range(2):
            rb = ipass
            for j in range(2,ny+2):
                for i in range(2+rb, nx+2, 2):
                    p[i,j] = ( (1.0-beta)*p[i,j] + beta*tmp2[i,j]*(
                             (1.0/dx/dx)* ( p[i+1,j]/(rt[i+1,j]+rt[i,j])
                                          +p[i-1,j]/(rt[i-1,j]+rt[i,j]))
                            +(1.0/dy/dy)* ( p[i,j+1]/(rt[i,j+1]+rt[i,j])
                                          +p[i,j-1]/(rt[i,j-1]+rt[i,j]))
                            - tmp1[i,j] ) )
                rb=1-rb


        if (np.abs(oldp-p)).max() < maxError :
            break

    # Correct the u-velocity
```

```
    for i in range(2,nx+1):
        for j in range(2,ny+2):
            u[i,j]=ut[i,j]-dt*(2.0/dx)*(p[i+1,j]-p[i,j])/(r[i+1,j]+r[i,j])

    # Correct the v-velocity
    for i in range(2,nx+2):
        for j in range(2,ny+1):
            v[i,j]=vt[i,j]-dt*(2.0/dy)*(p[i,j+1]-p[i,j])/(r[i,j+1]+r[i,j])

    #------------------ Plot the results ----------------------------
    time=time+dt                    # plot the results
    uu[1:nx+2,1:ny+2]=(0.5*(u[1:nx+2,2:ny+3]+u[1:nx+2,1:ny+2]))
    vv[1:nx+2,1:ny+2]=(0.5*(v[2:nx+3,1:ny+2]+v[1:nx+2,1:ny+2]))


    plt.cla()
    plt.contour(x[2:nx+2],y[2:ny+2],chi.T[2:nx+2,2:ny+2])
    plt.quiver(xh[1:],yh[1:],uu.T[1:,1:],vv.T[1:,1:])

    plt.pause(0.01)


print('time_elapsed= %s' % (time0.time() - time_start) )
plt.close()


                  # End of time step
```

Listing 2: Python Code 1B

```
#===========================================================================
# CodeC1-advChi.py
# A very simple Navier-Stokes solver for a drop falling in a
# rectangular box, using a conservative form of the equations.
# A first-order explicit projection method and centered in space
# discretizationa are used. The marker function is advected by
# an advection diffusion equation.
# Original Matlab code by Gretar Tryggvason
# Python code converted by Tingyi Lu on 7/25/2018
#===========================================================================

import numpy as np
import matplotlib.pyplot as plt
import time as time0

Lx=1.0;Ly=1.0;gx=0.0;gy=-100.0  # Domain size and
rho1=1.0; rho2=2.0; m0=0.01     # physical variables
unorth=0;usouth=0;veast=0;vwest=0;time=0.0
rad=0.15;xc=0.5;yc=0.7 # Initial drop size and location

#--------------------- Numerical variables -----------------------
nx=32;ny=32;dt=0.00125;nstep=400; maxit=200; maxError=0.001; beta=1.5

#--------------------- Zero various arrys -----------------------
u=np.zeros((nx+2,ny+3));  v=np.zeros((nx+3,ny+2));  p=np.zeros((nx+3,ny+3))
ut=np.zeros((nx+2,ny+3)); vt=np.zeros((nx+3,ny+2)); tmp1=np.zeros((nx+3,ny+3))
uu=np.zeros((nx+2,ny+2)); vv=np.zeros((nx+2,ny+2)); tmp2=np.zeros((nx+3,ny+3))
r=np.zeros((nx+3,ny+3)); chi=np.zeros((nx+3,ny+3))

dx=Lx/nx;dy=Ly/ny                          # Set the grid
x=np.linspace(-1.5*dx, (nx+0.5)*dx, nx+3)
y=np.linspace(-1.5*dy, (ny+0.5)*dy, ny+3)
xh=np.linspace(-dx, nx*dx, nx+2)
yh=np.linspace(-dy, ny*dy, ny+2)

X, Y = np.meshgrid(x,y)

#--------------------- Initial Conditions -----------------------
r = np.zeros((nx+3,nx+3))+rho1 # Set density

for i in range(nx+3):
    for j in range(ny+3):                    # for the domain and the drop
        if((x[i]-xc)**2+(y[j]-yc)**2 <rad**2):
            r[i,j] = rho2
            chi[i,j] = 1.0

plt.contour(X,Y,chi.T)
plt.axis('scaled')
plt.axis([0, Lx, 0, Ly], aspect=1)

time_start = time0.time()

#---------------------- START TIME LOOP ------------------------
for istep in range(nstep):
    print(istep)

    #--- ADVECT marker using centered difference plus diffusion ----
    chio=chi.copy()

    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]
                +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]))
                -(0.5* dt/dy)*(v[2:nx+2,2:ny+2]*(chio[2:nx+2,3:ny+3]
                +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])  )
                +(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])
                +(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]))

    #-------------------- Update the density -----------------------
```

```
    ro=r.copy()
    r = rho1 + (rho2-rho1)*chi

    #-------------- Set tangential velocity at boundaries -----------
    u[:,1]=2*usouth-u[:,2];u[:,ny+2]=2*unorth-u[:,ny+1]
    v[1,:]=2*vwest-v[2,:];v[nx+2,:]=2*veast-v[nx+1,:]

    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*
                       (ro[3:nx+2,2:ny+2]+ro[2:nx+1,2:ny+2])*u[2:nx+1,2:ny+2]+ dt* (
            -(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-
                ro[2:nx+1,2:ny+2]*(u[2:nx+1,2:ny+2]+u[1:nx,2:ny+2])**2)
        -(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])*
                        (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])
                      -(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]
                        +u[2:nx+1,1:ny+1])*(v[3:nx+2,1:ny+1]+v[2:nx+1,1:ny+1]))
                   +m0*((u[3:nx+2,2:ny+2]-2*u[2:nx+1,2:ny+2]+u[1:nx,2:ny+2])/dx**2+
                        (u[2:nx+1,3:ny+3]-2*u[2:nx+1,2:ny+2]+u[2:nx+1,1:ny+1])/dy**2)
                 + 0.5*(ro[3:nx+2,2:ny+2]+ro[2:nx+1,2:ny+2])*gx ) ))

    vt[2:nx+2,2:ny+1]=((2.0/(r[2:nx+2,3:ny+2]+r[2:nx+2,2:ny+1]))*(0.5*
                       (ro[2:nx+2,3:ny+2]+ro[2:nx+2,2:ny+1])*v[2:nx+2,2:ny+1]+ dt* (
        -(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])*
                        (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])
                     - (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])*
                        (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]) )
            -(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-
                        ro[2:nx+2,2:ny+1]*(v[2:nx+2,2:ny+1]+v[2:nx+2,1:ny])**2 )
                   +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+
                        (v[2:nx+2,3:ny+2]-2*v[2:nx+2,2:ny+1]+v[2:nx+2,1:ny])/dy**2)
                 + 0.5*(ro[2:nx+2,3:ny+2]+ro[2:nx+2,2:ny+1])*gy  ) ))

    #------------------ Solve the Pressure Equation ----------------
    rt=r.copy(); lrg=1e20   # Compute source term and the coefficient for p[i,j]
    rt[:,1]=lrg;rt[:,ny+2]=lrg
    rt[1,:]=lrg;rt[nx+2,:]=lrg


    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+
                                     (vt[2:nx+2,2:ny+2]-vt[2:nx+2,1:ny+1])/dy ))
    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]))+
                                        1./(dx*(rt[1:nx+1,2:ny+2]+rt[2:nx+2,2:ny+2])) )+
                               (1./dy)*(1./(dy*(rt[2:nx+2,3:ny+3]+rt[2:nx+2,2:ny+2]))+
                                        1./(dy*(rt[2:nx+2,1:ny+1]+rt[2:nx+2,2:ny+2])) )   ))

    for it in range(maxit):                 # Solve for pressure by SOR
        oldp=p.copy()

        #Red & Black SOR
        for ipass in range(2):
            rb = ipass
            p[2+rb:nx+2:2,2:ny+2:2] = ( (1.0-beta)*p[2+rb:nx+2:2,2:ny+2:2] + beta*tmp2[2+rb:nx+2:2,2:ny+2:2]*(
                         (1.0/dx/dx)* ( p[3+rb:nx+3:2,2:ny+2:2]/(rt[3+rb:nx+3:2,2:ny+2:2]+rt[2+rb:nx+2:2,2:ny+2:2])
                                       +p[1+rb:nx+1:2,2:ny+2:2]/(rt[1+rb:nx+1:2,2:ny+2:2]+rt[2+rb:nx+2:2,2:ny+2:2]))
                        +(1.0/dy/dy)* ( p[2+rb:nx+2:2,3:ny+3:2]/(rt[2+rb:nx+2:2,3:ny+3:2]+rt[2+rb:nx+2:2,2:ny+2:2])
                                       +p[2+rb:nx+2:2,1:ny+1:2]/(rt[2+rb:nx+2:2,1:ny+1:2]+rt[2+rb:nx+2:2,2:ny+2:2]))
                        - tmp1[2+rb:nx+2:2,2:ny+2:2] ) )

            rb=1-ipass
            p[2+rb:nx+2:2,3:ny+2:2] = ( (1.0-beta)*p[2+rb:nx+2:2,3:ny+2:2] + beta*tmp2[2+rb:nx+2:2,3:ny+2:2]*(
                         (1.0/dx/dx)* ( p[3+rb:nx+3:2,3:ny+2:2]/(rt[3+rb:nx+3:2,3:ny+2:2]+rt[2+rb:nx+2:2,3:ny+2:2])
                                       +p[1+rb:nx+1:2,3:ny+2:2]/(rt[1+rb:nx+1:2,3:ny+2:2]+rt[2+rb:nx+2:2,3:ny+2:2]))
                        +(1.0/dy/dy)* ( p[2+rb:nx+2:2,4:ny+3:2]/(rt[2+rb:nx+2:2,4:ny+3:2]+rt[2+rb:nx+2:2,3:ny+2:2])
                                       +p[2+rb:nx+2:2,2:ny+1:2]/(rt[2+rb:nx+2:2,2:ny+1:2]+rt[2+rb:nx+2:2,3:ny+2:2]))
                        - tmp1[2+rb:nx+2:2,3:ny+2:2] ) )

        p[1,:] = p[2,:]; p[nx+2,:] = p[nx+1,:]
        p[:,1] = p[:,2]; p[:,ny+2] = p[:,ny+1]

```

```python
        if (np.abs(oldp-p)).max() < maxError :
            break

    # Correct the u-velocity
    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+1,2:ny+2]))

    # Correct the v-velocity
    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+2,2:ny+1]))

    #------------------ Plot the results ----------------------------
    time=time+dt                   # plot the results
    uu[1:nx+2,1:ny+2]=(0.5*(u[1:nx+2,2:ny+3]+u[1:nx+2,1:ny+2]))
    vv[1:nx+2,1:ny+2]=(0.5*(v[2:nx+3,1:ny+2]+v[1:nx+2,1:ny+2]))

    plt.cla()
    plt.contour(x[2:nx+2],y[2:ny+2],chi.T[2:nx+2,2:ny+2])
    plt.quiver(xh[1:],yh[1:],uu.T[1:,1:],vv.T[1:,1:])

    plt.pause(0.01)

print('time_elapsed= %s' % (time0.time() - time_start) )

plt.close()

                  # End of time step
```

Listing 3: Python Code 2A

```python
#==========================================================================
# CodeC2-frt.m
# A very simple Navier-Stokes solver for a drop falling in a
# rectangular box, using a conservative form of the equations.
# A first-order explicit projection method and centered in space
# discretizationa are used. The marker function is advected by
# Tront Tracking.
# Original Matlab code by Gretar Tryggvason
# Python code converted by Tingyi Lu on 7/25/2018
#==========================================================================

import numpy as np
import matplotlib.pyplot as plt
import time as time0

Lx=1.0;Ly=1.0;gx=0.0;gy=-100.0  # Domain size and
rho1=1.0; rho2=2.0; m0=0.01      # physical variables
unorth=0;usouth=0;veast=0;vwest=0;time=0.0
rad=0.15;xc=0.5;yc=0.7 # Initial drop size and location

#-------------------- Numerical variables -----------------------
nx=32;ny=32;dt=0.00125;nstep=400; maxit=200;maxError=0.001;beta=1.5;
Nf=100; Maxf=2000

#-------------------- Zero various arrys ------------------------
u=np.zeros((nx+1,ny+2));  v=np.zeros((nx+2,ny+1));  p=np.zeros((nx+2,ny+2))
ut=np.zeros((nx+1,ny+2)); vt=np.zeros((nx+2,ny+1)); tmp1=np.zeros((nx+2,ny+1))
uu=np.zeros((nx+1,ny+1)); vv=np.zeros((nx+1,ny+1)); tmp2=np.zeros((nx+2,ny+2))
r=np.zeros((nx+2,ny+2)); chi=np.zeros((nx+2,ny+2))
d=np.zeros((nx+1,nx+1))
xf=np.zeros(Maxf); yf=np.zeros(Maxf)
uf=np.zeros(Maxf); vf=np.zeros(Maxf)

dx=Lx/nx;dy=Ly/ny                          # Set the grid
x=np.linspace(-.5*dx, (nx+0.5)*dx, nx+2)
y=np.linspace(-.5*dx, (ny+0.5)*dy, ny+2)
xh=np.linspace(0, Lx, nx+1)
yh=np.linspace(0, Ly, ny+1)


#-------------------- Initial Conditions ------------------------
r[:,:]=rho1                        # Set density

for i in range(1,nx+1):                       # for the domain and the drop
    for j in range(1,ny+1):
        if((x[i]-xc)**2+(y[j]-yc)**2 < rad**2):
            r[i,j]=rho2
            chi[i,j]=1.0


for l in range(Nf+2):
    xf[l]=xc-rad*np.sin(2.0*np.pi*l/Nf)       # Initialize
    yf[l]=yc+rad*np.cos(2.0*np.pi*l/Nf)    # the Front


plt.plot(xf[0:Nf],yf[0:Nf],'k',linewidth=3)
plt.axis('scaled')
plt.axis([0,Lx,0,Ly], aspect=1)
plt.pause(0.0001)

time_start = time0.time()

#---------------------- START TIME LOOP -------------------------
for istep in range(nstep):
    print(istep)


```

```python
#----------------------- Advect the Front -----------------------

for l in range(1,Nf+1):                     # Interpolate the Front Velocities
    ip=np.int(xf[l]/dx); jp=np.int((yf[l]+0.5*dy)/dy)
    ax=xf[l]/dx-ip; ay=(yf[l]+0.5*dy)/dy-jp
    uf[l]=((1.0-ax)*(1.0-ay)*u[ip,jp]+ax*(1.0-ay)*u[ip+1,jp]+
             (1.0-ax)*ay*u[ip,jp+1]+ax*ay*u[ip+1,jp+1])

    ip=np.int((xf[l]+0.5*dx)/dx); jp=np.int(yf[l]/dy)
    ax=(xf[l]+0.5*dx)/dx-ip; ay=yf[l]/dy-jp
    vf[l]=((1.0-ax)*(1.0-ay)*v[ip,jp]+ax*(1.0-ay)*v[ip+1,jp]+
             (1.0-ax)*ay*v[ip,jp+1]+ax*ay*v[ip+1,jp+1])

for i in range(1,Nf+1):
    xf[i]= xf[i]+dt*uf[i]; yf[i]=yf[i]+dt*vf[i]          #Move the
xf[0]=xf[Nf];yf[0]=yf[Nf];xf[Nf+1]=xf[1];yf[Nf+1]=yf[1] # Front


#--------------- Update the marker function ----------------------
d[:,:]=2

for l in range(1,Nf+1):
    nfx=-(yf[l+1]-yf[l])/dx
    nfy=(xf[l+1]-xf[l])/dy  # Normal vector
    ds=np.sqrt(nfx*nfx+nfy*nfy); nfx=nfx/ds; nfy=nfy/ds
    xfront=0.5*(xf[l]+xf[l+1]); yfront=0.5*(yf[l]+yf[l+1])
    ip=np.int((xfront+0.5*dx)/dx); jp=np.int((yfront+0.5*dy)/dy)

    d1=np.sqrt(((xfront-  x[ip])/dx)**2+((yfront-  y[jp])/dy)**2)
    d2=np.sqrt(((xfront-x[ip+1])/dx)**2+((yfront-  y[jp])/dy)**2)
    d3=np.sqrt(((xfront-x[ip+1])/dx)**2+((yfront-y[jp+1])/dy)**2)
    d4=np.sqrt(((xfront-  x[ip])/dx)**2+((yfront-y[jp+1])/dy)**2)

    if d1<d[ip,jp]:
        d[ip,jp]=d1.copy()
        dn1=(x[ip]-xfront)*nfx/dx+(y[jp]-yfront)*nfy/dy
        chi[ip,jp]=0.5*(1.0+np.sign(dn1))
        if abs(dn1)<0.5:
            chi[ip,jp]=0.5+dn1

    if d2<d[ip+1,jp]:
        d[ip+1,jp]=d2.copy()
        dn2=(x[ip+1]-xfront)*nfx/dx+(y[jp]-yfront)*nfy/dy
        chi[ip+1,jp]=0.5*(1.0+np.sign(dn2))
        if abs(dn2)<0.5:
            chi[ip+1,jp]=0.5+dn2

    if d3<d[ip+1,jp+1]:
        d[ip+1,jp+1]=d3.copy()
        dn3=(x[ip+1]-xfront)*nfx/dx+(y[jp+1]-yfront)*nfy/dy
        chi[ip+1,jp+1]=0.5*(1.0+np.sign(dn3))
        if abs(dn3)<0.5:
            chi[ip+1,jp+1]=0.5+dn3

    if d4<d[ip,jp+1]:
        d[ip,jp+1]=d4.copy()
        dn4=(x[ip]-  xfront)*nfx/dx+(y[jp+1]-yfront)*nfy/dy
        chi[ip,jp+1]=  0.5*(1.0+np.sign(dn4))
        if abs(dn4)<0.5:
            chi[ip,jp+1]=  0.5+dn4


#-------------------- Update the density ------------------------

ro=r.copy()
r = rho1+(rho2-rho1)*chi

#------------- Set tangential velocity at boundaries -----------
```

```python
    u[:,0]=2*usouth-u[:,1];u[:,ny+1]=2*unorth-u[:,ny]
    v[0,:]=2*vwest-v[1,:];v[nx+1,:]=2*veast-v[nx,:]

    #-------------- Find the predicted velocities ------------------
    for i in range(1,nx):
        for j in range(1,ny+1):      # Temporary u-velocity
            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* (
               -(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)
               -(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])
                                        -(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]))
                  +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)
                            + 0.5*(ro[i+1,j]+ro[i,j])*gx ) ))



    for i in range(1,nx+1):
        for j in range(1,ny):       # Temporary v-velocity
            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* (
                    -(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])
                  - (ro[i,j]+ro[i,j+1]+ro[i-1,j+1]+ro[i-1,j])*
                                     (u[i-1,j+1]+u[i-1,j])*(v[i,j]+v[i-1,j]) )
                    -(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 )
                    +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)
                            + 0.5*(ro[i,j+1]+ro[i,j])*gy   )))


    #------------------ Solve the Pressure Equation ----------------
    rt=r.copy(); lrg=1000   # Compute source term and the coefficient for p(i,j)
    rt[:,0]=lrg;rt[:,ny+1]=lrg
    rt[0,:]=lrg;rt[nx+1,:]=lrg

    for i in range(1,nx+1):
        for j in range(1,ny+1):
            tmp1[i,j]= (0.5/dt)*( (ut[i,j]-ut[i-1,j])/dx+(vt[i,j]-vt[i,j-1])/dy )
            tmp2[i,j]=(1.0/( (1./dx)*(1./(dx*(rt[i+1,j]+rt[i,j]))+
                              1./(dx*(rt[i-1,j]+rt[i,j])) )+
                     (1./dy)*(1./(dy*(rt[i,j+1]+rt[i,j]))+
                              1./(dy*(rt[i,j-1]+rt[i,j])) )   ))


    for it in range(maxit):                # Solve for pressure by SOR
        oldArray=p.copy()
        #Red & Black SOR
        for ipass in range(2):
            rb = ipass
            for j in range(1,ny+1):
                for i in range(1+rb, nx+1, 2):
                    p[i,j] = ( (1.0-beta)*p[i,j] + beta*tmp2[i,j]*(
                             (1.0/dx/dx)*( p[i+1,j]/(rt[i+1,j]+rt[i,j])
                                          +p[i-1,j]/(rt[i-1,j]+rt[i,j]))
                            +(1.0/dy/dy)*( p[i,j+1]/(rt[i,j+1]+rt[i,j])
                                          +p[i,j-1]/(rt[i,j-1]+rt[i,j]))
                            - tmp1[i,j] ) )
                rb=1-rb


        if (np.abs(oldArray-p)).max() <maxError:
            break

    for i in range(1,nx):
        for j in range(1,ny+1):   # Correct the u-velocity
            u[i,j]=ut[i,j]-dt*(2.0/dx)*(p[i+1,j]-p[i,j])/(r[i+1,j]+r[i,j])

    for i in range(1,nx+1):
        for j in range(1,ny):   # Correct the v-velocity
```

```python
            v[i,j]=vt[i,j]-dt*(2.0/dy)*(p[i,j+1]-p[i,j])/(r[i,j+1]+r[i,j])

    #--------------- Add and deleate points in the Front -----------
    xfold=xf.copy();yfold=yf.copy(); j=0
    for l in range(1,Nf+1):
        ds=np.sqrt( ((xfold[l]-xf[j])/dx)**2 + ((yfold[l]-yf[j])/dy)**2)
        if ds > 0.5:
            j=j+1;xf[j]=0.5*(xfold[l]+xf[j-1]);yf[j]=0.5*(yfold[l]+yf[j-1])
            j=j+1;xf[j]=xfold[l];yf[j]=yfold[l]
        elif 0.25<=ds<=0.5:
            j=j+1;xf[j]=xfold[l];yf[j]=yfold[l]

    Nf=j-1
    xf[0]=xf[Nf];yf[0]=yf[Nf];xf[Nf+1]=xf[1];yf[Nf+1]=yf[1]

#------------------ Plot the results ----------------------------

    time+=dt                    # plot the results
    uu[0:nx+1,0:ny+1]=0.5*(u[0:nx+1,1:ny+2]+u[0:nx+1,0:ny+1])
    vv[0:nx+1,0:ny+1]=0.5*(v[1:nx+2,0:ny+1]+v[0:nx+1,0:ny+1])

    plt.cla()
    plt.contour(x[1:nx+1],y[1:ny+1],chi.T[1:nx+1,1:ny+1])
    plt.quiver(xh[:],yh[:],uu.T[:,:],vv.T[:,:])

    plt.plot(xf[0:Nf],yf[0:Nf],'k',linewidth=3)
    plt.pause(0.0001)

print('time_elapsed= %s' % (time0.time() - time_start) )

                  # End of time step
```

Listing 4: Python Code 2B

```python
#======================================================================
# CodeC2-frt.m
# A very simple Navier-Stokes solver for a drop falling in a
# rectangular box, using a conservative form of the equations.
# A first-order explicit projection method and centered in space
# discretizationa are used. The marker function is advected by
# Tront Tracking.
# Original Matlab code by Gretar Tryggvason
# Python code converted by Tingyi Lu on 7/25/2018
#======================================================================

import numpy as np
import matplotlib.pyplot as plt
import time as time0

Lx=1.0;Ly=1.0;gx=0.0;gy=-100.0  # Domain size and
rho1=1.0; rho2=2.0; m0=0.01      # physical variables
unorth=0;usouth=0;veast=0;vwest=0;time=0.0
rad=0.15;xc=0.5;yc=0.7 # Initial drop size and location

#--------------------- Numerical variables -----------------------
nx=32;ny=32;dt=0.00125;nstep=400; maxit=200;maxError=0.001;beta=1.5;
Nf=100; Maxf=2000

#--------------------- Zero various arrys ------------------------
u=np.zeros((nx+1,ny+2));  v=np.zeros((nx+2,ny+1));  p=np.zeros((nx+2,ny+2))
ut=np.zeros((nx+1,ny+2)); vt=np.zeros((nx+2,ny+1)); tmp1=np.zeros((nx+2,ny+1))
uu=np.zeros((nx+1,ny+1)); vv=np.zeros((nx+1,ny+1)); tmp2=np.zeros((nx+2,ny+2))
r=np.zeros((nx+2,ny+2)); chi=np.zeros((nx+2,ny+2))
d=np.zeros((nx+1,nx+1))
xf=np.zeros(Maxf); yf=np.zeros(Maxf)
uf=np.zeros(Maxf); vf=np.zeros(Maxf)

dx=Lx/nx;dy=Ly/ny                          # Set the grid
x=np.linspace(-.5*dx, (nx+0.5)*dx, nx+2)
y=np.linspace(-.5*dx, (ny+0.5)*dy, ny+2)
xh=np.linspace(0, Lx, nx+1)
yh=np.linspace(0, Ly, ny+1)


#--------------------- Initial Conditions ------------------------
r[:,:]=rho1                         # Set density

for i in range(1,nx+1):                        # for the domain and the drop
    for j in range(1,ny+1):
        if((x[i]-xc)**2+(y[j]-yc)**2 < rad**2):
            r[i,j]=rho2
            chi[i,j]=1.0


for l in range(Nf+2):
    xf[l]=xc-rad*np.sin(2.0*np.pi*l/Nf)       # Initialize
    yf[l]=yc+rad*np.cos(2.0*np.pi*l/Nf)    # the Front


plt.plot(xf[0:Nf],yf[0:Nf],'k',linewidth=3)
plt.axis('scaled')
plt.axis([0,Lx,0,Ly], aspect=1)
plt.pause(0.0001)

time_start = time0.time()

#---------------------- START TIME LOOP -------------------------
for istep in range(nstep):
    print(istep)


```

```python
#------------------------ Advect the Front -------------------------

for l in range(1,Nf+1):                          # Interpolate the Front Velocities
    ip=np.int(xf[l]/dx); jp=np.int((yf[l]+0.5*dy)/dy)
    ax=xf[l]/dx-ip; ay=(yf[l]+0.5*dy)/dy-jp
    uf[l]=((1.0-ax)*(1.0-ay)*u[ip,jp]+ax*(1.0-ay)*u[ip+1,jp]+
             (1.0-ax)*ay*u[ip,jp+1]+ax*ay*u[ip+1,jp+1])

    ip=np.int((xf[l]+0.5*dx)/dx); jp=np.int(yf[l]/dy)
    ax=(xf[l]+0.5*dx)/dx-ip; ay=yf[l]/dy-jp
    vf[l]=((1.0-ax)*(1.0-ay)*v[ip,jp]+ax*(1.0-ay)*v[ip+1,jp]+
             (1.0-ax)*ay*v[ip,jp+1]+ax*ay*v[ip+1,jp+1])

xf[1:Nf+1]+=dt*uf[1:Nf+1]
yf[1:Nf+1]+=dt*vf[1:Nf+1] # Move the
xf[0]=xf[Nf];yf[0]=yf[Nf];xf[Nf+1]=xf[1];yf[Nf+1]=yf[1] # Front


#--------------- Update the marker function ----------------------
d[:,:]=2

for l in range(1,Nf+1):
    nfx=-(yf[l+1]-yf[l])/dx
    nfy=(xf[l+1]-xf[l])/dy  # Normal vector
    ds=np.sqrt(nfx*nfx+nfy*nfy); nfx=nfx/ds; nfy=nfy/ds
    xfront=0.5*(xf[l]+xf[l+1]); yfront=0.5*(yf[l]+yf[l+1])
    ip=np.int((xfront+0.5*dx)/dx); jp=np.int((yfront+0.5*dy)/dy)

    d1=np.sqrt(((xfront-  x[ip])/dx)**2+((yfront-  y[jp])/dy)**2)
    d2=np.sqrt(((xfront-x[ip+1])/dx)**2+((yfront-  y[jp])/dy)**2)
    d3=np.sqrt(((xfront-x[ip+1])/dx)**2+((yfront-y[jp+1])/dy)**2)
    d4=np.sqrt(((xfront-  x[ip])/dx)**2+((yfront-y[jp+1])/dy)**2)

    if d1<d[ip,jp]:
        d[ip,jp]=d1.copy()
        dn1=(x[ip]-xfront)*nfx/dx+(y[jp]-yfront)*nfy/dy
        chi[ip,jp]=0.5*(1.0+np.sign(dn1))
        if abs(dn1)<0.5:
            chi[ip,jp]=0.5+dn1

    if d2<d[ip+1,jp]:
        d[ip+1,jp]=d2.copy()
        dn2=(x[ip+1]-xfront)*nfx/dx+(y[jp]-yfront)*nfy/dy
        chi[ip+1,jp]=0.5*(1.0+np.sign(dn2))
        if abs(dn2)<0.5:
            chi[ip+1,jp]=0.5+dn2

    if d3<d[ip+1,jp+1]:
        d[ip+1,jp+1]=d3.copy()
        dn3=(x[ip+1]-xfront)*nfx/dx+(y[jp+1]-yfront)*nfy/dy
        chi[ip+1,jp+1]=0.5*(1.0+np.sign(dn3))
        if abs(dn3)<0.5:
            chi[ip+1,jp+1]=0.5+dn3

    if d4<d[ip,jp+1]:
        d[ip,jp+1]=d4.copy()
        dn4=(x[ip]-  xfront)*nfx/dx+(y[jp+1]-yfront)*nfy/dy
        chi[ip,jp+1]=  0.5*(1.0+np.sign(dn4))
        if abs(dn4)<0.5:
            chi[ip,jp+1]=  0.5+dn4


#--------------------- Update the density -------------------------

ro=r.copy()
r = rho1+(rho2-rho1)*chi

#-------------- Set tangential velocity at boundaries -----------
```

```
u[:,0]=2*usouth-u[:,1];u[:,ny+1]=2*unorth-u[:,ny]
v[0,:]=2*vwest-v[1,:];v[nx+1,:]=2*veast-v[nx,:]

#-------------- Find the predicted velocities ------------------
                        # Temporary u-velocity
ut[1:nx,1:ny+1]=((2.0/(r[2:nx+1,1:ny+1]+r[1:nx,1:ny+1]))*(0.5*
                   (ro[2:nx+1,1:ny+1]+ro[1:nx,1:ny+1])*u[1:nx,1:ny+1]+ dt* (
    -(0.25/dx)*(ro[2:nx+1,1:ny+1]*(u[2:nx+1,1:ny+1]+u[1:nx,1:ny+1])**2-
                ro[1:nx,1:ny+1]*(u[1:nx,1:ny+1]+u[:nx-1,1:ny+1])**2)
      -(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])*
                (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])
                    -(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]
                      +u[1:nx,:ny])*(v[2:nx+1,:ny]+v[1:nx,:ny]))
                 +m0*((u[2:nx+1,1:ny+1]-2*u[1:nx,1:ny+1]+u[:nx-1,1:ny+1])/dx**2+
                      (u[1:nx,2:ny+2]-2*u[1:nx,1:ny+1]+u[1:nx,:ny])/dy**2)
               + 0.5*(ro[2:nx+1,1:ny+1]+ro[1:nx,1:ny+1])*gx ) ))


             # Temporary v-velocity
vt[1:nx+1,1:ny]=((2.0/(r[1:nx+1,2:ny+1]+r[1:nx+1,1:ny]))*(0.5*
                     (ro[1:nx+1,2:ny+1]+ro[1:nx+1,1:ny])*v[1:nx+1,1:ny]+ dt* (
      -(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])*
                      (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])
                   - (ro[1:nx+1,1:ny]+ro[1:nx+1,2:ny+1]+ro[:nx,2:ny+1]+ro[:nx,1:ny])*
                      (u[:nx,2:ny+1]+u[:nx,1:ny])*(v[1:nx+1,1:ny]+v[:nx,1:ny]) )
          -(0.25/dy)*(ro[1:nx+1,2:ny+1]*(v[1:nx+1,2:ny+1]+v[1:nx+1,1:ny])**2-
                ro[1:nx+1,1:ny]*(v[1:nx+1,1:ny]+v[1:nx+1,:ny-1])**2 )
                 +m0*((v[2:nx+2,1:ny]-2*v[1:nx+1,1:ny]+v[:nx,1:ny])/dx**2+
            (v[1:nx+1,2:ny+1]-2*v[1:nx+1,1:ny]+v[1:nx+1,:ny-1])/dy**2)
               + 0.5*(ro[1:nx+1,2:ny+1]+ro[1:nx+1,1:ny])*gy  ) ))



#------------------ Solve the Pressure Equation ----------------
rt=r.copy(); lrg=1000   # Compute source term and the coefficient for p(i,j)
rt[:,0]=lrg;rt[:,ny+1]=lrg
rt[0,:]=lrg;rt[nx+1,:]=lrg

tmp1[1:nx+1,1:ny+1]=((0.5/dt)*( (ut[1:nx+1,1:ny+1]-ut[0:nx,1:ny+1])/dx+
                                (vt[1:nx+1,1:ny+1]-vt[1:nx+1,0:ny])/dy ))
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]))+
                             1./(dx*(rt[0:nx,1:ny+1]+rt[1:nx+1,1:ny+1])) )+
                  (1./dy)*(1./(dy*(rt[1:nx+1,2:ny+2]+rt[1:nx+1,1:ny+1]))+
                             1./(dy*(rt[1:nx+1,0:ny]+rt[1:nx+1,1:ny+1])) )   ))

for it in range(maxit):                 # Solve for pressure by SOR
    oldArray=p.copy()
        #Red & Black SOR
    for ipass in range(2):
        rb = ipass
        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]*(
                     (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])
                                  +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]))
                    +(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])
                                  +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]))
                    - tmp1[1+rb:nx+1:2,1:ny+1:2] ) )

        rb=1-ipass
        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]*(
                     (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])
                                  +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]))
                    +(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])
                                  +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]))
                    - tmp1[1+rb:nx+1:2,2:ny+1:2] ) )

    p[0,:] = p[1,:]; p[nx+1,:] = p[nx,:]
    p[:,0] = p[:,1]; p[:,ny+1] = p[:,ny]

```

```
        if (np.abs(oldArray-p)).max() <maxError:
            break

            # Correct the u-velocity
    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])
           # Correct the v-velocity
    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])


    #---------------- Add and deleate points in the Front ------------
    xfold=xf.copy();yfold=yf.copy(); j=0
    for l in range(1,Nf+1):
        ds=np.sqrt( ((xfold[l]-xf[j])/dx)**2 + ((yfold[l]-yf[j])/dy)**2)
        if ds > 0.5:
            j=j+1;xf[j]=0.5*(xfold[l]+xf[j-1]);yf[j]=0.5*(yfold[l]+yf[j-1])
            j=j+1;xf[j]=xfold[l];yf[j]=yfold[l]
        elif 0.25<=ds<=0.5:
            j=j+1;xf[j]=xfold[l];yf[j]=yfold[l]

    Nf=j-1
    xf[0]=xf[Nf];yf[0]=yf[Nf];xf[Nf+1]=xf[1];yf[Nf+1]=yf[1]

#------------------- Plot the results -----------------------------

    time+=dt                    # plot the results
    uu[0:nx+1,0:ny+1]=0.5*(u[0:nx+1,1:ny+2]+u[0:nx+1,0:ny+1])
    vv[0:nx+1,0:ny+1]=0.5*(v[1:nx+2,0:ny+1]+v[0:nx+1,0:ny+1])

    plt.cla()
    plt.contour(x[1:nx+1],y[1:ny+1],chi.T[1:nx+1,1:ny+1])
    plt.quiver(xh[:],yh[:],uu.T[:,:],vv.T[:,:])

    plt.plot(xf[0:Nf],yf[0:Nf],'k',linewidth=3)
    plt.pause(0.0001)

print('time_elapsed= %s' % (time0.time() - time_start) )

                  # End of time step
```

Listing 5: Python Code 3A

```python
#===================================================================
# CodeC3-frt-st-RK3.m
# A very simple Navier-Stokes solver for a drop falling in a
# rectangular box, using a conservative form of the equations.
# A 3-order explicit projection method and centered in space
# discretizationa are used. The density is advected by a front
# tracking scheme and surface tension and variable viscosity is
# included. This version uses a simple method to create the
# marker function.
# Original Matlab code by Gretar Tryggvason
# Python code converted by Tingyi Lu on 7/25/2018
#===================================================================

import numpy as np
import matplotlib.pyplot as plt
import time as time0

Lx=1.0;Ly=1.0;gx=0.0;gy=-100.0; sigma=10 # Domain size and
rho1=1.0; rho2=2.0; m1=0.01; m2=0.02      # physical variables
unorth=0; usouth=0; veast=0; vwest=0; time=0.0
rad=0.15; xc=0.5; yc=0.7           # Initial drop size and location

#--------------------- Numerical variables -----------------------
nx=32;ny=32;dt=0.001;nstep=400; maxit=200;maxError=0.001;beta=1.5; Nf=100; Maxf=2000

#--------------------- Zero various arrys ------------------------
u=np.zeros((nx+1,ny+2));  v=np.zeros((nx+2,ny+1));  p=np.zeros((nx+2,ny+2))
ut=np.zeros((nx+1,ny+2)); vt=np.zeros((nx+2,ny+1)); tmp1=np.zeros((nx+2,ny+2))
uu=np.zeros((nx+1,ny+1)); vv=np.zeros((nx+1,ny+1)); tmp2=np.zeros((nx+2,ny+2))
fx=np.zeros((nx+2,ny+2)); fy=np.zeros((nx+2,ny+2)); r=np.zeros((nx+2,ny+2))
r=np.zeros((nx+2,ny+2));  chi=np.zeros((nx+2,ny+2))
m=np.zeros((nx+2,ny+2));  d=np.zeros((nx+2,ny+2))
xf=np.zeros(Maxf); yf=np.zeros(Maxf)
uf=np.zeros(Maxf); vf=np.zeros(Maxf)
tx=np.zeros(Maxf); ty=np.zeros(Maxf)
un=np.zeros((nx+1,ny+2)); vn=np.zeros((nx+2,ny+1))   # Used for
rn=np.zeros((nx+2,ny+2)); mn=np.zeros((nx+2,ny+2))   # higher order
xfn=np.zeros(Maxf); yfn=np.zeros(Maxf)                # in time
Area=np.zeros(nstep);CentroidX=np.zeros(nstep);CentroidY=np.zeros(nstep)
Time1=np.zeros(nstep)


dx=Lx/nx;dy=Ly/ny                                     # Set the grid
x=np.linspace(-.5*dx, (nx+0.5)*dx, nx+2)
y=np.linspace(-.5*dx, (ny+0.5)*dy, ny+2)
xh=np.linspace(0, Lx, nx+1)
yh=np.linspace(0, Ly, ny+1)

#--------------------- Initial Conditions ------------------------
r[:,:]=rho1;m[:,:]=m1 # Set density and viscosity

for i in range(1,nx+1):                       # for the domain and the drop
    for j in range(1,ny+1):
        if((x[i]-xc)**2+(y[j]-yc)**2 < rad**2):
            r[i,j]=rho2
            m[i,j]=m2
            chi[i,j]=1.0

for l in range(Nf+2):
    xf[l]=xc-rad*np.sin(2.0*np.pi*l/Nf)       # Initialize
    yf[l]=yc+rad*np.cos(2.0*np.pi*l/Nf)       # the Front

plt.plot(xf[0:Nf],yf[0:Nf],'k',linewidth=3)
plt.axis('scaled')
plt.axis([0,Lx,0,Ly], aspect=1)
plt.pause(0.0001)

```

```python
time_start = time0.time()

#---------------------- START TIME LOOP ------------------------
for istep in range(nstep):
    print(istep)
    un=u.copy(); vn=v.copy(); rn=r.copy(); mn=m.copy(); xfn=xf.copy(); yfn=yf.copy()  # Higher order
    for substep in range(3):                         # in time

#---------------------- Advect the Front -----------------------
        for l in range(1,Nf+1):                      # Interpolate the Front Velocities
            ip=np.int(xf[l]/dx); jp=np.int((yf[l]+0.5*dy)/dy)
            ax=xf[l]/dx-ip; ay=(yf[l]+0.5*dy)/dy-jp
            uf[l]=((1.0-ax)*(1.0-ay)*u[ip,jp]+ax*(1.0-ay)*u[ip+1,jp]+
                 (1.0-ax)*ay*u[ip,jp+1]+ax*ay*u[ip+1,jp+1])

            ip=np.int((xf[l]+0.5*dx)/dx); jp=np.int(yf[l]/dy)
            ax=(xf[l]+0.5*dx)/dx-ip; ay=yf[l]/dy-jp
            vf[l]=((1.0-ax)*(1.0-ay)*v[ip,jp]+ax*(1.0-ay)*v[ip+1,jp]+
                 (1.0-ax)*ay*v[ip,jp+1]+ax*ay*v[ip+1,jp+1])

        for l in range(1,Nf+1):
            xf[l]=xf[l]+dt*uf[l]; yf[l]=yf[l]+dt*vf[l]
        xf[0]=xf[Nf];yf[0]=yf[Nf];xf[Nf+1]=xf[1];yf[Nf+1]=yf[1]

#-------------- Update the marker function ---------------------
        d[:,:]=2

        for l in range(1,Nf+1):
            nfx=-(yf[l+1]-yf[l])/dx
            nfy=(xf[l+1]-xf[l])/dy  # Normal vector
            ds=np.sqrt(nfx*nfx+nfy*nfy); nfx=nfx/ds; nfy=nfy/ds
            xfront=0.5*(xf[l]+xf[l+1]); yfront=0.5*(yf[l]+yf[l+1])
            ip=np.int((xfront+0.5*dx)/dx); jp=np.int((yfront+0.5*dy)/dy)

            d1=np.sqrt(((xfront-  x[ip])/dx)**2+((yfront-  y[jp])/dy)**2)
            d2=np.sqrt(((xfront-x[ip+1])/dx)**2+((yfront-  y[jp])/dy)**2)
            d3=np.sqrt(((xfront-x[ip+1])/dx)**2+((yfront-y[jp+1])/dy)**2)
            d4=np.sqrt(((xfront-  x[ip])/dx)**2+((yfront-y[jp+1])/dy)**2)

            if d1<d[ip,jp]:
                d[ip,jp]=d1.copy()
                dn1=(x[ip]-xfront)*nfx/dx+(y[jp]-yfront)*nfy/dy
                chi[ip,jp]=0.5*(1.0+np.sign(dn1))
                if abs(dn1)<0.5:
                    chi[ip,jp]=0.5+dn1

            if d2<d[ip+1,jp]:
                d[ip+1,jp]=d2.copy()
                dn2=(x[ip+1]-xfront)*nfx/dx+(y[jp]-yfront)*nfy/dy
                chi[ip+1,jp]=0.5*(1.0+np.sign(dn2))
                if abs(dn2)<0.5:
                    chi[ip+1,jp]=0.5+dn2

            if d3<d[ip+1,jp+1]:
                d[ip+1,jp+1]=d3.copy()
                dn3=(x[ip+1]-xfront)*nfx/dx+(y[jp+1]-yfront)*nfy/dy
                chi[ip+1,jp+1]=0.5*(1.0+np.sign(dn3))
                if abs(dn3)<0.5:
                    chi[ip+1,jp+1]=0.5+dn3

            if d4<d[ip,jp+1]:
                d[ip,jp+1]=d4.copy()
                dn4=(x[ip]-  xfront)*nfx/dx+(y[jp+1]-yfront)*nfy/dy
                chi[ip,jp+1]=  0.5*(1.0+np.sign(dn4))
                if abs(dn4)<0.5:
                    chi[ip,jp+1]=  0.5+dn4

#-------------------- Update the density -----------------------
```

```
        ro=r.copy()
        r = rho1+(rho2-rho1)*chi

#------------------- Find surface tension -------------------------
        fx=np.zeros((nx+2,ny+2));fy=np.zeros((nx+2,ny+2))  # Set fx & fy to zero

        for l in range(Nf+1):
            ds=np.sqrt((xf[l+1]-xf[l])**2+(yf[l+1]-yf[l])**2)
            tx[l]=(xf[l+1]-xf[l])/ds
            ty[l]=(yf[l+1]-yf[l])/ds # Tangent vectors

        tx[Nf+1]=tx[1];ty[Nf+1]=ty[1]

        for l in range(1,Nf+1):            # Distribute to the fixed grid
            nfx=sigma*(tx[l]-tx[l-1]);nfy=sigma*(ty[l]-ty[l-1])

            ip=np.int(xf[l]/dx); jp=np.int((yf[l]+0.5*dy)/dy)
            ax=xf[l]/dx-ip; ay=(yf[l]+0.5*dy)/dy-jp
            fx[ip,jp]     +=(1.0-ax)*(1.0-ay)*nfx/dx/dy
            fx[ip+1,jp]   +=ax*(1.0-ay)*nfx/dx/dy
            fx[ip,jp+1]   +=(1.0-ax)*ay*nfx/dx/dy
            fx[ip+1,jp+1]+=ax*ay*nfx/dx/dy

            ip=np.int((xf[l]+0.5*dx)/dx); jp=np.int(yf[l]/dy)
            ax=(xf[l]+0.5*dx)/dx-ip; ay=yf[l]/dy-jp
            fy[ip,jp]     +=(1.0-ax)*(1.0-ay)*nfy/dx/dy
            fy[ip+1,jp]   +=ax*(1.0-ay)*nfy/dx/dy
            fy[ip,jp+1]   +=(1.0-ax)*ay*nfy/dx/dy
            fy[ip+1,jp+1]+=ax*ay*nfy/dx/dy

        fx[:,1]=fx[:,1]+fx[:,0]            # Correct boundary
        fx[:,ny]=fx[:,ny]+fx[:,ny+1]   # values for the
        fy[1,:]=fy[1,:]+fy[0,:]            # surface force
        fy[nx,:]=fy[nx,:]+fy[nx+1,:]   # on the grid

#-------------- Set tangential velocity at boundaries ------------
        u[:,0]=2*usouth-u[:,1];u[:,ny+1]=2*unorth-u[:,ny]
        v[0,:]=2*vwest-v[1,:];v[nx+1,:]=2*veast-v[nx,:]


#--------------- Find the predicted velocities -------------------
        for i in range(1,nx):
            for j in range(1,ny+1):       # Temporary u-velocity-advection
                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* (
                    -(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)
                    -(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])
                    -(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]))
                                    + 0.5*(ro[i+1,j]+ro[i,j])*gx + fx[i,j]) ))

        for i in range(1,nx+1):
            for j in range(1,ny):        # Temporary v-velocity-advection
                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* (
                         -(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])
                       - (ro[i,j]+ro[i,j+1]+ro[i-1,j+1]+ro[i-1,j])*
                                          (u[i-1,j+1]+u[i-1,j])*(v[i,j]+v[i-1,j]) )
                         -(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 )
                                    + 0.5*(ro[i,j+1]+ro[i,j])*gy + fy[i,j] )))

        for i in range(1,nx):
            for j in range(1,ny+1):       # Temporary u-velocity-viscosity
                ut[i,j]=(ut[i,j]+(2.0/(r[i+1,j]+r[i,j]))*dt*(
                        +(1./dx)*2.*(m[i+1,j]*(1./dx)*(u[i+1,j]-u[i,j]) -
                           m[i,j]  *(1./dx)*(u[i,j]-u[i-1,j]) )
                 +(1./dy)*( 0.25*(m[i,j]+m[i+1,j]+m[i+1,j+1]+m[i,j+1])*
                    ((1./dy)*(u[i,j+1]-u[i,j]) + (1./dx)*(v[i+1,j]-v[i,j]) ) -
```

```
                        0.25*(m[i,j]+m[i+1,j]+m[i+1,j-1]+m[i,j-1])*
                  ((1./dy)*(u[i,j]-u[i,j-1])+ (1./dx)*(v[i+1,j-1]- v[i,j-1])) ) ))


        for i in range(1,nx+1):
            for j in range(1,ny):        # Temporary v-velocity-viscosity
                vt[i,j]=(vt[i,j]+(2.0/(r[i,j+1]+r[i,j]))*dt*(
                  +(1./dx)*( 0.25*(m[i,j]+m[i+1,j]+m[i+1,j+1]+m[i,j+1])*
                 ((1./dy)*(u[i,j+1]-u[i,j]) + (1./dx)*(v[i+1,j]-v[i,j]) ) -
                       0.25*(m[i,j]+m[i,j+1]+m[i-1,j+1]+m[i-1,j])*
                ((1./dy)*(u[i-1,j+1]-u[i-1,j])+ (1./dx)*(v[i,j]- v[i-1,j])) )
                 +(1./dy)*2.*(m[i,j+1]*(1./dy)*(v[i,j+1]-v[i,j]) -
                        m[i,j] *(1./dy)*(v[i,j]-v[i,j-1]) ) ))


#------------------ Solve the Pressure Equation ----------------
        rt=r.copy(); lrg=1000    # Compute source term and the coefficient for p(i,j)
        rt[:,0]=lrg;rt[:,ny+1]=lrg
        rt[0,:]=lrg;rt[nx+1,:]=lrg

        for i in range(1,nx+1):
            for j in range(1,ny+1):
                tmp1[i,j]= (0.5/dt)*( (ut[i,j]-ut[i-1,j])/dx+(vt[i,j]-vt[i,j-1])/dy )
                tmp2[i,j]=(1.0/( (1./dx)*(1./(dx*(rt[i+1,j]+rt[i,j]))+
                                   1./(dx*(rt[i-1,j]+rt[i,j])) )+
                          (1./dy)*(1./(dy*(rt[i,j+1]+rt[i,j]))+
                                   1./(dy*(rt[i,j-1]+rt[i,j])) )   ))

        for it in range(maxit):                  # Solve for pressure by Red-Black SOR
            oldArray=p.copy()
            for ipass in range(2):
                 rb = ipass
                 for j in range(1,ny+1):
                     for i in range(1+rb, nx+1, 2):
                         p[i,j] = ( (1.0-beta)*p[i,j] + beta*tmp2[i,j]*(
                                  (1.0/dx/dx)* ( p[i+1,j]/(rt[i+1,j]+rt[i,j])
                                                +p[i-1,j]/(rt[i-1,j]+rt[i,j]))
                                 +(1.0/dy/dy)* ( p[i,j+1]/(rt[i,j+1]+rt[i,j])
                                                +p[i,j-1]/(rt[i,j-1]+rt[i,j]))
                                 - tmp1[i,j] ) )
                     rb=1-rb

            if (np.abs(oldArray-p)).max() <maxError:
                break

        for i in range(1,nx):
            for j in range(1,ny+1):    # Correct the u-velocity
                u[i,j]=ut[i,j]-dt*(2.0/dx)*(p[i+1,j]-p[i,j])/(r[i+1,j]+r[i,j])

        for i in range(1,nx+1):
            for j in range(1,ny):    # Correct the v-velocity
                v[i,j]=vt[i,j]-dt*(2.0/dy)*(p[i,j+1]-p[i,j])/(r[i,j+1]+r[i,j])

        for i in range(0,nx+2):
            for j in range(0,ny+2): # Update the viscosity
                m[i,j]=m1+(m2-m1)*chi[i,j]

        if substep==1: # Higher order (RK-3) in time
            u=0.75*un+0.25*u; v=0.75*vn+0.25*v; r=0.75*rn+0.25*r
            m=0.75*mn+0.25*m; xf=0.75*xfn+0.25*xf; yf=0.75*yfn+0.25*yf
        elif substep==2:
            u=(1/3)*un+(2/3)*u; v=(1/3)*vn+(2/3)*v; r=(1/3)*rn+(2/3)*r;
            m=(1/3)*mn+(2/3)*m; xf=(1/3)*xfn+(2/3)*xf; yf=(1/3)*yfn+(2/3)*yf;

#-------------- Add and deleate points in the Front -----------
    xfold=xf.copy();yfold=yf.copy(); j=0
    for l in range(1,Nf+1):
        ds=np.sqrt( ((xfold[l]-xf[j])/dx)**2 + ((yfold[l]-yf[j])/dy)**2)
```

```
        if ds > 0.5:
            j=j+1;xf[j]=0.5*(xfold[l]+xf[j-1]);yf[j]=0.5*(yfold[l]+yf[j-1])
            j=j+1;xf[j]=xfold[l];yf[j]=yfold[l]
        elif 0.25<=ds<=0.5:
            j=j+1;xf[j]=xfold[l];yf[j]=yfold[l]

    Nf=j-1
    xf[0]=xf[Nf];yf[0]=yf[Nf];xf[Nf+1]=xf[1];yf[Nf+1]=yf[1]

#------------------ Compute Diagnostic quantitites ---------------
    Area[istep]=0; CentroidX[istep]=0; CentroidY[istep]=0; Time1[istep]=time

    for l in range(1,Nf+1):
        Area[istep]+=0.25*((xf[l+1]+xf[l])*(yf[l+1]-yf[l])-(yf[l+1]+yf[l])*(xf[l+1]-xf[l]))
        CentroidX[istep]+=0.125*((xf[l+1]+xf[l])**2+(yf[l+1]+yf[l])**2)*(yf[l+1]-yf[l])
        CentroidY[istep]-=0.125*((xf[l+1]+xf[l])**2+(yf[l+1]+yf[l])**2)*(xf[l+1]-xf[l])


    CentroidX[istep]=CentroidX[istep]/Area[istep];CentroidY[istep]=CentroidY[istep]/Area[istep]

#------------------- Plot the results -----------------------------
    time+=dt                    # plot the results
    uu[0:nx+1,0:ny+1]=0.5*(u[0:nx+1,1:ny+2]+u[0:nx+1,0:ny+1])
    vv[0:nx+1,0:ny+1]=0.5*(v[1:nx+2,0:ny+1]+v[0:nx+1,0:ny+1])

    plt.cla()
    plt.contour(x[1:nx+1],y[1:ny+1],chi.T[1:nx+1,1:ny+1])
    plt.quiver(xh[:],yh[:],uu.T[:,:],vv.T[:,:])

    plt.plot(xf[0:Nf],yf[0:Nf],'k',linewidth=3)
    plt.axis([0,Lx,0,Ly], aspect=1)
    plt.pause(0.0001)

print('time_elapsed= %s' % (time0.time() - time_start) )


#------ Extra commands for interactive processing ---------------
#plt.figure(2)
#plt.plot(Time1,Area,'r',linewidth=2)
#plt.axis([0,dt*nstep,0,0.1])
#plt.show()
```

Listing 6: Python Code 3B

```
#==============================================================
# CodeC3-frt-st-RK3.m
# A very simple Navier-Stokes solver for a drop falling in a
# rectangular box, using a conservative form of the equations.
# A 3-order explicit projection method and centered in space
# discretizationa are used. The density is advected by a front
# tracking scheme and surface tension and variable viscosity is
# included. This version uses a simple method to create the
# marker function.
# Original Matlab code by Gretar Tryggvason
# Python code converted by Tingyi Lu on 7/25/2018
#==============================================================

import numpy as np
import matplotlib.pyplot as plt
import time as time0

Lx=1.0;Ly=1.0;gx=0.0;gy=-100.0; sigma=10 # Domain size and
rho1=1.0; rho2=2.0; m1=0.01; m2=0.02     # physical variables
unorth=0; usouth=0; veast=0; vwest=0; time=0.0
rad=0.15; xc=0.5; yc=0.7          # Initial drop size and location

#-------------------- Numerical variables ----------------------
nx=32;ny=32;dt=0.001;nstep=400; maxit=200;maxError=0.001;beta=1.5; Nf=100; Maxf=2000

#-------------------- Zero various arrys -----------------------
u=np.zeros((nx+1,ny+2));  v=np.zeros((nx+2,ny+1));  p=np.zeros((nx+2,ny+2))
ut=np.zeros((nx+1,ny+2)); vt=np.zeros((nx+2,ny+1)); tmp1=np.zeros((nx+2,ny+2))
uu=np.zeros((nx+1,ny+1)); vv=np.zeros((nx+1,ny+1)); tmp2=np.zeros((nx+2,ny+2))
fx=np.zeros((nx+2,ny+2)); fy=np.zeros((nx+2,ny+2)); r=np.zeros((nx+2,ny+2))
r=np.zeros((nx+2,ny+2));  chi=np.zeros((nx+2,ny+2))
m=np.zeros((nx+2,ny+2));  d=np.zeros((nx+2,ny+2))
xf=np.zeros(Maxf); yf=np.zeros(Maxf)
uf=np.zeros(Maxf); vf=np.zeros(Maxf)
tx=np.zeros(Maxf); ty=np.zeros(Maxf)
un=np.zeros((nx+1,ny+2)); vn=np.zeros((nx+2,ny+1))   # Used for
rn=np.zeros((nx+2,ny+2)); mn=np.zeros((nx+2,ny+2))   # higher order
xfn=np.zeros(Maxf); yfn=np.zeros(Maxf)               # in time
Area=np.zeros(nstep);CentroidX=np.zeros(nstep);CentroidY=np.zeros(nstep)
Time1=np.zeros(nstep)


dx=Lx/nx;dy=Ly/ny                              # Set the grid
x=np.linspace(-.5*dx, (nx+0.5)*dx, nx+2)
y=np.linspace(-.5*dx, (ny+0.5)*dy, ny+2)
xh=np.linspace(0, Lx, nx+1)
yh=np.linspace(0, Ly, ny+1)

#-------------------- Initial Conditions -----------------------
r[:,:]=rho1;m[:,:]=m1 # Set density and viscosity

for i in range(1,nx+1):                # for the domain and the drop
    for j in range(1,ny+1):
        if((x[i]-xc)**2+(y[j]-yc)**2 < rad**2):
            r[i,j]=rho2
            m[i,j]=m2
            chi[i,j]=1.0

for l in range(Nf+2):
    xf[l]=xc-rad*np.sin(2.0*np.pi*l/Nf)      # Initialize
    yf[l]=yc+rad*np.cos(2.0*np.pi*l/Nf)      # the Front

plt.plot(xf[0:Nf],yf[0:Nf],'k',linewidth=3)
plt.axis('scaled')
plt.axis([0,Lx,0,Ly], aspect=1)
plt.pause(0.0001)

```

```
time_start = time0.time()

#----------------------- START TIME LOOP --------------------------
for istep in range(nstep):
    print(istep)
    un=u.copy(); vn=v.copy(); rn=r.copy(); mn=m.copy(); xfn=xf.copy(); yfn=yf.copy()  # Higher order
    for substep in range(3):                            # in time

#------------------------ Advect the Front ------------------------
        for l in range(1,Nf+1):                        # Interpolate the Front Velocities
            ip=np.int(xf[l]/dx); jp=np.int((yf[l]+0.5*dy)/dy)
            ax=xf[l]/dx-ip; ay=(yf[l]+0.5*dy)/dy-jp
            uf[l]=((1.0-ax)*(1.0-ay)*u[ip,jp]+ax*(1.0-ay)*u[ip+1,jp]+
                 (1.0-ax)*ay*u[ip,jp+1]+ax*ay*u[ip+1,jp+1])

            ip=np.int((xf[l]+0.5*dx)/dx); jp=np.int(yf[l]/dy)
            ax=(xf[l]+0.5*dx)/dx-ip; ay=yf[l]/dy-jp
            vf[l]=((1.0-ax)*(1.0-ay)*v[ip,jp]+ax*(1.0-ay)*v[ip+1,jp]+
                 (1.0-ax)*ay*v[ip,jp+1]+ax*ay*v[ip+1,jp+1])

        xf[1:Nf+1]+=dt*uf[1:Nf+1]
        yf[1:Nf+1]+=dt*vf[1:Nf+1] # Move the
        xf[0]=xf[Nf];yf[0]=yf[Nf];xf[Nf+1]=xf[1];yf[Nf+1]=yf[1] # Front


#--------------- Update the marker function ----------------------
        d[:,:]=2

        for l in range(1,Nf+1):
            nfx=-(yf[l+1]-yf[l])/dx
            nfy=(xf[l+1]-xf[l])/dy  # Normal vector
            ds=np.sqrt(nfx*nfx+nfy*nfy); nfx=nfx/ds; nfy=nfy/ds
            xfront=0.5*(xf[l]+xf[l+1]); yfront=0.5*(yf[l]+yf[l+1])
            ip=np.int((xfront+0.5*dx)/dx); jp=np.int((yfront+0.5*dy)/dy)

            d1=np.sqrt(((xfront-  x[ip])/dx)**2+((yfront-  y[jp])/dy)**2)
            d2=np.sqrt(((xfront-x[ip+1])/dx)**2+((yfront-  y[jp])/dy)**2)
            d3=np.sqrt(((xfront-x[ip+1])/dx)**2+((yfront-y[jp+1])/dy)**2)
            d4=np.sqrt(((xfront-  x[ip])/dx)**2+((yfront-y[jp+1])/dy)**2)

            if d1<d[ip,jp]:
                d[ip,jp]=d1.copy()
                dn1=(x[ip]-xfront)*nfx/dx+(y[jp]-yfront)*nfy/dy
                chi[ip,jp]=0.5*(1.0+np.sign(dn1))
                if abs(dn1)<0.5:
                    chi[ip,jp]=0.5+dn1

            if d2<d[ip+1,jp]:
                d[ip+1,jp]=d2.copy()
                dn2=(x[ip+1]-xfront)*nfx/dx+(y[jp]-yfront)*nfy/dy
                chi[ip+1,jp]=0.5*(1.0+np.sign(dn2))
                if abs(dn2)<0.5:
                    chi[ip+1,jp]=0.5+dn2

            if d3<d[ip+1,jp+1]:
                d[ip+1,jp+1]=d3.copy()
                dn3=(x[ip+1]-xfront)*nfx/dx+(y[jp+1]-yfront)*nfy/dy
                chi[ip+1,jp+1]=0.5*(1.0+np.sign(dn3))
                if abs(dn3)<0.5:
                    chi[ip+1,jp+1]=0.5+dn3

            if d4<d[ip,jp+1]:
                d[ip,jp+1]=d4.copy()
                dn4=(x[ip]-  xfront)*nfx/dx+(y[jp+1]-yfront)*nfy/dy
                chi[ip,jp+1]=  0.5*(1.0+np.sign(dn4))
                if abs(dn4)<0.5:
                    chi[ip,jp+1]=  0.5+dn4
```

```
#--------------------- Update the density ------------------------
    ro=r.copy()
    r = rho1+(rho2-rho1)*chi

#------------------- Find surface tension ------------------------
    fx=np.zeros((nx+2,ny+2));fy=np.zeros((nx+2,ny+2))  # Set fx & fy to zero

    for l in range(Nf+1):
        ds=np.sqrt((xf[l+1]-xf[l])**2+(yf[l+1]-yf[l])**2)
        tx[l]=(xf[l+1]-xf[l])/ds
        ty[l]=(yf[l+1]-yf[l])/ds # Tangent vectors

    tx[Nf+1]=tx[1];ty[Nf+1]=ty[1]

    for l in range(1,Nf+1):           # Distribute to the fixed grid
        nfx=sigma*(tx[l]-tx[l-1]);nfy=sigma*(ty[l]-ty[l-1])

        ip=np.int(xf[l]/dx); jp=np.int((yf[l]+0.5*dy)/dy)
        ax=xf[l]/dx-ip; ay=(yf[l]+0.5*dy)/dy-jp
        fx[ip,jp]    +=(1.0-ax)*(1.0-ay)*nfx/dx/dy
        fx[ip+1,jp]  +=ax*(1.0-ay)*nfx/dx/dy
        fx[ip,jp+1]  +=(1.0-ax)*ay*nfx/dx/dy
        fx[ip+1,jp+1]+=ax*ay*nfx/dx/dy

        ip=np.int((xf[l]+0.5*dx)/dx); jp=np.int(yf[l]/dy)
        ax=(xf[l]+0.5*dx)/dx-ip; ay=yf[l]/dy-jp
        fy[ip,jp]    +=(1.0-ax)*(1.0-ay)*nfy/dx/dy
        fy[ip+1,jp]  +=ax*(1.0-ay)*nfy/dx/dy
        fy[ip,jp+1]  +=(1.0-ax)*ay*nfy/dx/dy
        fy[ip+1,jp+1]+=ax*ay*nfy/dx/dy

    fx[:,1]=fx[:,1]+fx[:,0]           # Correct boundary
    fx[:,ny]=fx[:,ny]+fx[:,ny+1]  # values for the
    fy[1,:]=fy[1,:]+fy[0,:]           # surface force
    fy[nx,:]=fy[nx,:]+fy[nx+1,:]  # on the grid
#------------- Set tangential velocity at boundaries -----------
    u[:,0]=2*usouth-u[:,1];u[:,ny+1]=2*unorth-u[:,ny]
    v[0,:]=2*vwest-v[1,:];v[nx+1,:]=2*veast-v[nx,:]


#-------------- Find the predicted velocities ------------------

                              # Temporary u-velocity-advection
    ut[1:nx,1:ny+1]=( (2.0/(r[2:nx+1,1:ny+1]+r[1:nx,1:ny+1]))
                *(0.5*(ro[2:nx+1,1:ny+1]+ro[1:nx,1:ny+1])*u[1:nx,1:ny+1]+ dt* (
        -(0.25/dx)*(  ro[2:nx+1,1:ny+1]*(u[2:nx+1,1:ny+1]+u[1:nx,1:ny+1])**2
                  -ro[1:nx,1:ny+1]*(u[1:nx,1:ny+1]+u[:nx-1,1:ny+1])**2)
      -(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])*
                    (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])
                   -(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]+u[1:nx,:ny])*(v[2:nx+1,:ny]+v[1:nx,:ny]))
              + 0.5*(ro[2:nx+1,1:ny+1]+ro[1:nx,1:ny+1])*gx + fx[1:nx,1:ny+1]) ))

               # Temporary v-velocity-advection
    vt[1:nx+1,1:ny]=((2.0/(r[1:nx+1,2:ny+1]+r[1:nx+1,1:ny]))*
                   (0.5*(ro[1:nx+1,2:ny+1]+ro[1:nx+1,1:ny])*v[1:nx+1,1:ny]+ dt* (
         -(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])*
                         (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])
                      - (ro[1:nx+1,1:ny]+ro[1:nx+1,2:ny+1]+ro[:nx,2:ny+1]+ro[:nx,1:ny])*
                         (u[:nx,2:ny+1]+u[:nx,1:ny])*(v[1:nx+1,1:ny]+v[:nx,1:ny]) )
             -(0.25/dy)*(ro[1:nx+1,2:ny+1]*(v[1:nx+1,2:ny+1]+v[1:nx+1,1:ny])**2-
                         ro[1:nx+1,1:ny]*(v[1:nx+1,1:ny]+v[1:nx+1,:ny-1])**2 )
                + 0.5*(ro[1:nx+1,2:ny+1]+ro[1:nx+1,1:ny])*gy + fy[1:nx+1,1:ny] ) ))



      #Temporary u-velocity-viscosity
    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*(
        +(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]) -
```

```
                      m[1:nx,1:ny+1]  *(1./dx)*(u[1:nx,1:ny+1]-u[0:nx-1,1:ny+1]) )
        +(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])*
             ((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]) ) -
                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])*
             ((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])) ) ))

        #Temporary v-velocity-viscosity
      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*(
           +(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])*
           ((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]) ) -
                  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])*
           ((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])) )
           +(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]) -
                   m[1:nx+1,1:ny] *(1./dy)*(v[1:nx+1,1:ny]-v[1:nx+1,0:ny-1]) ) ))

#------------------ Solve the Pressure Equation ----------------
      rt=r.copy(); lrg=1000   # Compute source term and the coefficient for p(i,j)
      rt[:,0]=lrg;rt[:,ny+1]=lrg
      rt[0,:]=lrg;rt[nx+1,:]=lrg

      tmp1[1:nx+1,1:ny+1]=((0.5/dt)*( (ut[1:nx+1,1:ny+1]-ut[0:nx,1:ny+1])/dx+
                                    (vt[1:nx+1,1:ny+1]-vt[1:nx+1,0:ny])/dy))
      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]))+
                                 1./(dx*(rt[0:nx,1:ny+1]+rt[1:nx+1,1:ny+1])) )+
                           (1./dy)*(1./(dy*(rt[1:nx+1,2:ny+2]+rt[1:nx+1,1:ny+1]))+
                                 1./(dy*(rt[1:nx+1,0:ny]+rt[1:nx+1,1:ny+1])) )   ))


      for it in range(maxit):                # Solve for pressure by SOR
          oldArray=p.copy()

          #Red & Black SOR
          for ipass in range(2):
              rb = ipass
              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]*(
                        (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])
                                  +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]))
                       +(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])
                                  +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]))
                       - tmp1[1+rb:nx+1:2,1:ny+1:2] ) )

              rb=1-ipass
              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]*(
                        (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])
                                  +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]))
                       +(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])
                                  +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]))
                       - tmp1[1+rb:nx+1:2,2:ny+1:2] ) )

          p[0,:] = p[1,:]; p[nx+1,:] = p[nx,:]
          p[:,0] = p[:,1]; p[:,ny+1] = p[:,ny]


          if (np.abs(oldArray-p)).max() <maxError:
              break

              # Correct the u-velocity
      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])
             # Correct the v-velocity
      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])
          #Update the viscosity
      m[0:nx+1,0:ny+2]=m1+(m2-m1)*chi[0:nx+1,0:ny+2]


      if substep==1: # Higher order (RK-3) in time
          u=0.75*un+0.25*u; v=0.75*vn+0.25*v; r=0.75*rn+0.25*r
          m=0.75*mn+0.25*m; xf=0.75*xfn+0.25*xf; yf=0.75*yfn+0.25*yf
      elif substep==2:
```

```
            u=(1/3)*un+(2/3)*u; v=(1/3)*vn+(2/3)*v; r=(1/3)*rn+(2/3)*r;
            m=(1/3)*mn+(2/3)*m; xf=(1/3)*xfn+(2/3)*xf; yf=(1/3)*yfn+(2/3)*yf;

#---------------- Add and deleate points in the Front ------------
    xfold=xf.copy();yfold=yf.copy(); j=0
    for l in range(1,Nf+1):
        ds=np.sqrt( ((xfold[l]-xf[j])/dx)**2 + ((yfold[l]-yf[j])/dy)**2)
        if ds > 0.5:
            j=j+1;xf[j]=0.5*(xfold[l]+xf[j-1]);yf[j]=0.5*(yfold[l]+yf[j-1])
            j=j+1;xf[j]=xfold[l];yf[j]=yfold[l]
        elif 0.25<=ds<=0.5:
            j=j+1;xf[j]=xfold[l];yf[j]=yfold[l]

    Nf=j-1
    xf[0]=xf[Nf];yf[0]=yf[Nf];xf[Nf+1]=xf[1];yf[Nf+1]=yf[1]

#------------------ Compute Diagnostic quantitites ---------------
    Area[istep]=0; CentroidX[istep]=0; CentroidY[istep]=0; Time1[istep]=time

    for l in range(1,Nf+1):
        Area[istep]+=0.25*((xf[l+1]+xf[l])*(yf[l+1]-yf[l])-(yf[l+1]+yf[l])*(xf[l+1]-xf[l]))
        CentroidX[istep]+=0.125*((xf[l+1]+xf[l])**2+(yf[l+1]+yf[l])**2)*(yf[l+1]-yf[l])
        CentroidY[istep]-=0.125*((xf[l+1]+xf[l])**2+(yf[l+1]+yf[l])**2)*(xf[l+1]-xf[l])

    CentroidX[istep]=CentroidX[istep]/Area[istep];CentroidY[istep]=CentroidY[istep]/Area[istep]

#------------------- Plot the results -----------------------------
    time+=dt                  # plot the results
    uu[0:nx+1,0:ny+1]=0.5*(u[0:nx+1,1:ny+2]+u[0:nx+1,0:ny+1])
    vv[0:nx+1,0:ny+1]=0.5*(v[1:nx+2,0:ny+1]+v[0:nx+1,0:ny+1])

    plt.cla()
    plt.contour(x[1:nx+1],y[1:ny+1],chi.T[1:nx+1,1:ny+1])

    plt.quiver(xh,yh,uu.T,vv.T)

    plt.plot(xf[0:Nf],yf[0:Nf],'k',linewidth=3)

    plt.axis([0,Lx,0,Ly], aspect=1)
    plt.pause(0.0001)

print('time_elapsed= %s' % (time0.time() - time_start) )

#------ Extra commands for interactive processing ---------------
#plt.figure(2)
#plt.plot(Time1,Area,'r',linewidth=2)
#plt.axis([0,dt*nstep,0,0.1])
#plt.show()
```