A Combined Droplet Evaporation/Break-up Model for the Atomization of Sprays

Adam O'Brien

Student Number: 1000195022

Course: Multi-phase Flows
Date: December 16, 2014

Contents

1 Objective

The objective of this report is to present a code, written in the popular programming language Python, which is used to model the evaporation and break-up of a continuous stream of liquid droplets within a cross-flow. This simplified approach is used to emulate the process of a spray undergoing atomization.

In section 2, the basic advection scheme for a droplet in cross-flow is presented. In section 3, the semi-empirical evaporation model, assuming constant droplet temperature, will be presented. In section 4, the Taylor Analogy Break-up (TAB) model will be presented. The TAB model simplifies the dynamics of an oscillating droplet by modelling it as a spring-mass-damper harmonic oscillator. In section 5, select results will be shown. Finally, in section **??**, some final remarks will be made.

2 Droplet Advection

Droplet advection is accomplished numerically, using a second-order predictor-corrector time integration scheme. The predictor-corrector scheme has the form

$$f(t, y) = \frac{dy}{dt} \tag{1}$$

$$\tilde{y}^{n+1} = y^n + \Delta t f(t^n, y^n) \tag{2}$$

where $\tilde{y}^{n+1}$ is the predictor step, and Δt is the time step. This step is identical to Euler time integration. The corrector step is given by

$$y^{n+1} = y^n + \frac{1}{2}\Delta t(f(t^n, y^n) + f(t^{n+1}, \tilde{y}^{n+1})) \tag{3}$$

In order to evaluate $f(t, y)$, the equation for drag on a sphere, given by

$$\vec{F}_{drag} = \frac{1}{2}\rho_g |\vec{v}_{rel}|^2 c_d A_{ref} \cdot \vec{n}_{v_{rel}} \tag{4}$$

is used, where ρ_g is the air around the droplet based on T_{ref} (introduced later), $\vec{v}_{rel}$ is the relative velocity between the droplet and the freestream, c_d is the drag coefficient which is empirically correlated, A_{ref} is the the sphere reference area, and $\vec{n}_{v_{rel}}$ is a unit vector in the direction of $\vec{v}_{rel}$. The drag

coefficient correlation is not given here for the sake of conciseness, but many correlations are readily available in the literature.

3 Evaporation Model

The evaporation model assumes that the droplet can be modelled from the well known "D^2-law", which has the form

$$D^2 = D_0^2 - \lambda t \tag{5}$$

where D_0 is the original droplet diameter, and λ is the evaporation constant under quiescient conditions. For a single droplet, the rate of mass loss due to evaporation can be expressed as

$$\frac{\delta m}{\delta t} = \frac{\pi}{4}\rho_l \lambda D \tag{6}$$

where ρ_l is the liquid density. Equation (6) can be modified to account for the effects of forced convection, or cross-flow, by using the following modification:

$$\frac{\delta m'}{\delta t} = 2\pi D \left(\frac{k_g}{c_g}\right) \ln(1 + B_M) \left[1 + 0.3 Re_d^{0.5} Pr_g^{0.33}\right] \tag{7}$$

where k_g, c_g, Re_d and Pr_g are, respectively, the thermal conductivity, the specific heat, the Reynolds number and the Prandtl number relative to the cross-flow surrounding the droplet. The mass transfer number, B_M, is computed as

$$B_M = \frac{Y_{l,s}}{1 - Y_{l,s}} \tag{8}$$

where $Y_{l,s}$ is the liquid mass fraction at the droplet surface, and must be obtained through the use of the Clausius-Clapeyron relationship. The Clausius-Clapeyron relationship is given by

$$Y_{l,s} = \frac{1}{1 + \frac{PM_g}{(P_{l,s}-1)M_d}} \tag{9}$$

where P is the pressure surrounding the droplet, M_g is the molecular weight of the air, $P_{l,s}$ is the vapour pressure of water and M_d is the molecular weight of water. The vapour pressue of water can be found from any

number of empirical correlations, with the temperature (T_{ref}) assumed to be an average between the ambient air temperature and the temperature of the droplet. Here, the following average is used:

$$T_{ref} = T_{droplet} + \frac{T_{ambient} - T_{droplet}}{3} \tag{10}$$

With T_{ref}, the vapour pressure can be correlated. The correlation is not given here for conciseness, but they are readily available within the experimental literature.

Once the surface vapour mass fraction ($Y_{l,s}$) is known, equation (8) is used to obtain B_M, and the modified droplet evaporation constant, λ', is computed by

$$\lambda' = \frac{8k_g \ln(1 + B_M)}{c_g \rho_l} \left[1 + 0.3 Re_d^{0.5} Pr_g^{0.33}\right] \tag{11}$$

where the "prime" superscript denotes a quantity which has been modified to account for advection. Equation 5 can then be used to update the droplet size at the current time step.

4 Taylor Analogy Break-up (TAB) Model

In addition to evaporation, the spray atomization model also includes a break-up model. The Taylor Analogy Break-up (TAB) model assumes the droplet can be represented by a simple spring-mass-dampter system, where the effects of surface tension are represented by the spring, the droplet mass distribution by the mass, and the effects of viscous dissipation by the damper. The model is shown graphically in figure 4.

The equation for a spring-mass-damper system is given as follows:

$$\frac{d^2y}{dt^2} + 2\zeta\omega_0 \frac{dy}{dt} + \omega_0^2 y = F(t) \tag{12}$$

where y is the displacement, ζ is the damping coefficient, ω_0 is the undamped natural frequency, and $F(t)$ is a time-dependant forcing term.

For a simple droplet, the equation of motion was determined to be

$$\frac{d^2y}{dt^2} + \frac{4C_d\mu}{\rho_l d^2}\frac{dy}{dt} + \frac{8C_k\sigma}{\rho_l d^3} y = \frac{4C_f \rho_g |\vec{v}_{rel}|^2}{C_b \rho_l d^2} \tag{13}$$

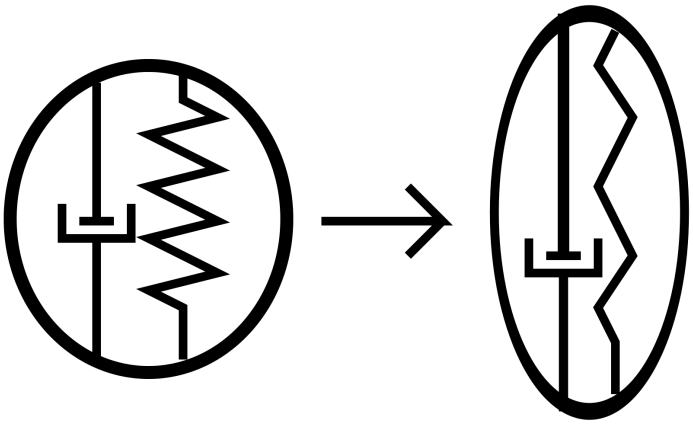

Figure 1: Droplet deformation modelled by the Taylor Analogy Break-up (TAB) model.

where μ, d, and σ are the droplet viscosity, diameter and surface tension coefficient respectively and C_d, C_k, C_f and C_b are empirically determined constants with the following values:

$$C_d = 10 \tag{14a}$$

$$C_k = 8 \tag{14b}$$

$$C_f = 2/3 \text{ (gradual loading)} \tag{14c}$$

$$C_f = 1/3 \text{ (sudden loading)} \tag{14d}$$

$$C_b = 1/2 \tag{14e}$$

The value of y is non-dimensional, and takes the form

$$y = \frac{4\Delta d_{min}}{d_0} \tag{15}$$

where d_{min} describes the deviation between the minor drop diameter and that of its original diameter, d_0.

The solution to equation 13 has the form:

$$y(t) = We_c + e^{-t/\tau_D}\left\{(y_0 + We_c)\cos(\omega t) + \frac{1}{\omega}\left(\frac{dy_0}{dt} + \frac{y_0 - We_c}{\tau_D}\right)\sin(\omega t)\right\} \tag{16}$$

where

$$\omega = \left\{ \frac{8C_k\sigma}{\rho_l d^3} - \frac{1}{\tau_D^2} \right\}^{\frac{1}{2}} \tag{17}$$

$$\tau_D = \frac{\rho_l d^2}{2C_d\mu} \tag{18}$$

and the critical Weber number, We_c, is

$$We_c = We\left(\frac{C_f}{C_k C_b}\right) \tag{19}$$

When considering a single discrete time step, and assuming that the droplet initially is not deformed nor deforming ($y_0 = \frac{dy_0}{dt} = 0$), equation (16) and its derivative take the forms

$$y^{n+1} = We_c + e^{-\Delta t/\tau_D}\left\{(y^n + We_c)\cos(\omega\Delta t) + \frac{1}{\omega}\left(\left(\frac{dy}{dt}\right)^n + \frac{y^n - We_c}{\tau_D}\right)\sin(\omega\Delta t)\right\} \tag{20}$$

$$\left(\frac{dy}{dt}\right)^{n+1} = \frac{We_c - y^{n+1}}{\tau_D} + \omega e^{-\Delta t/\tau_D}\left\{\frac{1}{\omega}\left(\left(\frac{dy}{dt}\right)^n + \frac{y^n - We_c}{\tau_D}\right)\cos(\omega\Delta t) - (y^n - We_c)\sin(\omega\Delta t)\right\} \tag{21}$$

From equations (20) and (21), the distortion of each droplet may be tracked. Breakup is said to occur when $y(t) \geq 1$.

After a break-up occurs, the size of the child droplets must be determined. This is done by equating the energy of the parent droplet to the combined energy of the child droplets. The energy of the parent droplet is

$$E_{parent} = 4\pi r^2\sigma + K\frac{\pi}{5}\rho_l r^5\left[\left(\frac{dy}{dt}\right)^2 + \omega^2 y^2\right] \tag{22}$$

where K is the ratio of total energy in distortion and oscillation, and is set to

$$K = \frac{10}{3} \tag{23}$$

The energy of the child droplets can then be shown to be

$$E_{child} = 4\pi r^2 \sigma \frac{r}{r_{32}} + \frac{\pi}{6}\rho_l r^5 \left(\frac{dy}{dt}\right)^2 \tag{24}$$

where r_{32} is the Sauter mean radius of the droplet distribution. Setting $y = 1$, subbing in equation (17), and setting equations (22) and (24) equal to each other, one obtains the equation for the Sauter mean radius as

$$r_{32} = \frac{r}{1 + \frac{8Ky^2}{20} + \frac{\rho_l r^3 (dy/dt)^2}{\sigma}\left(\frac{6K-5}{120}\right)} \tag{25}$$

The droplets can then be assumed to be normally distributed about r_{32}, with the number of droplets determined through mass conservation.

The velocity of the child droplets must also be computed. A normal component of velocity is added onto the droplet, which has the form

$$|\vec{v}_{normal}| = C_v C_b r \left(\frac{dy}{dt}\right) \tag{26}$$

where C_v is a constant on the order of 1.

5 Results

The following section contains results obtained from coding the evaporation and TAB droplet advection code. The code itself as been included in the appendices.

A boTAB Code - Main Module

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

"""
boTAB
=====
This solver uses the popular TAB model to simulate the
    atomization of droplets
Author: Adam O'Brien
"""

from input import *
from math import exp, cos, sin, sqrt
from fluid import *
from evaporation import *
from TAB import *
from output import *
import copy as cp

def main():

    print ""
    print "boTAB |"
    print "-------"
    print "        Compute the break-up of a drop in a
        uniform cross-flow", "\n"

    # Open up a configuration file

    userInput = readInputFile()

    freestream = Freestream()
    initialDroplet = Droplet()
    dropletInlet = DropletInlet()

```

```
34      # Set object parameters from the input file
35
36      setObjectParametersFromInput(userInput, freestream,
            initialDroplet, dropletInlet)
37
38      # Set-up the simulation parameters in accordance
            with the input
39
40      maxTime = userInput["maxTime"]
41      nTimeSteps = userInput["nTimeSteps"]
42
43      # Initialize a droplet list, with one copy of the
            initial droplet
44
45      droplets = [cp.deepcopy(initialDroplet)]
46
47      # Initialize misc parameters
48
49      dt = maxTime/nTimeSteps
50      t = [0.]
51      nChildDroplets = 0
52
53      # Begin the simulation
54
55      print "\nBeginning time-stepping..."
56
57      ##############################################
58      #                                            #
59      #            Main Iteration Loop             #
60      #                                            #
61      ##############################################
62
63      for stepNo in range(1, nTimeSteps + 1):
64
65          for droplet in droplets:
66
67              droplet.advectPredictorCorrector(freestream
                    , dt)
```

```
68
69          evaporate(freestream, droplets, dt)
70          nChildDroplets += breakupTab(freestream,
                droplets, dt)
71
72          dropletInlet.addDrops(initialDroplet, droplets,
                dt)
73          t.append(t[-1] + dt)
74
75          if stepNo%(nTimeSteps/20) == 0:
76
77              completionPercentage = float(stepNo)/float(
                    nTimeSteps)*100.
78
79              print "
                    ---------------------------------------------------------------------
                    "
80              print "Time-stepping completion      : %s%%"
                    %(completionPercentage)
81              print "Number of droplets in domain  :", len
                    (droplets)
82              print "Simulation time elapsed       : %s
                    seconds"%(t[-1])
83              print "Simulation time remaining     : %s
                    seconds"%(maxTime - t[-1])
84              print "Number of child drops         :",
                    nChildDroplets
85
86      print "\nTime-stepping complete. Finalizing output
            ..."
87
88      plotDroplets(droplets)
89
90  # Execute the main function
91
92  if __name__ == "__main__":
93      main()
```

B boTAB Code - Fluid Module

```
1
2 #/usr/bin/env python2
3 # -*- coding: utf-8 -*-
4
5 """
6 boTAB
7 =====
8 This module contains classes and function for the
      modelling of droplets in
9 freestream flows
10 Author: Adam O'Brien
11 """
12
13 from math import sqrt, pi, exp
14 import copy as cp
15 import random
16
17 # Vector class for position/velocity
18
19 class Vector(object):
20
21     def __init__(self, x, y):
22         self.x = x
23         self.y = y
24
25     def __repr__(self):
26
27         return "%s, %s"%(self.x, self.y)
28
29     def __add__(self, other):
30
31         return Vector(self.x + other.x, self.y + other.
              y)
32
33     def __sub__(self, other):
```

```
34
35          return Vector(self.x - other.x, self.y - other.
                y)
36
37      def __mul__(self, other):
38
39          return Vector(self.x*other, self.y*other)
40
41      def mag(self):
42          return sqrt(self.x**2 + self.y**2)
43
44      def normalVector(self):
45          return Vector(self.y, -self.x)
46
47      def scale(self, other):
48          return Vector(self.x*other, self.y*other)
49
50      def rVector(self, other):
51
52          return other - self
53
54      def unitVector(self):
55
56          return self.scale(1./self.mag())
57
58  def dot(u, v):
59
60      return u.x*v.x + u.y*v.y
61
62  # Freestream class for representing the freestream flow
63
64  class Freestream(object):
65
66      def __init__(self, velocity = Vector(40., 0.),
            gravity = Vector(0., 0.)):
67
68          # constructed properties
69
```

```
70          self.velocity = velocity
71          self.gravity = gravity
72
73          # default air proplerties (can be changed)
74
75          self.mu = 18.27e-6
76          self.Pambient = 101325.
77          self.Tambient = 800.
78          self.Cp = 1.0005
79          self.k = 0.0257
80          self.M = 28.97
81          self.Pr = self.Cp*self.mu/self.k
82
83          # Computed properties
84
85          self.rho = self.Pambient/(286.9*self.Tambient)
86
87 # Droplet class for representing droplets
88
89 class Droplet(object):
90
91      def __init__(self, radius = 5e-4, position = Vector
            (0., 0.), velocity = Vector(0., 10.)):
92
93          # constructed properties
94
95          self.radius = radius
96          self.position = position
97          self.velocity = velocity
98
99          # default water properties (can be changed)
100
101         self.rho = 998.
102         self.mu = 8.94e-4
103         self.sigma = 0.07262
104         self.Tboil = 373.
105         self.Tcrit = 647.096
106         self.T = 400.
```

```
107             self.L = 2257.
108             self.Cp = 4.183
109             self.k = 0.58
110             self.M = 18.01528
111 
112             # TAB properties
113 
114             self.y = 0.
115             self.dydt = 0.
116 
117         def __repr__(self):
118 
119             return "Radius: %s\nPosition: %s\nVelocity: %s"
                    %(str(self.radius), \
120             self.position, self.velocity)
121 
122         def diameter(self):
123 
124             return 2.*self.radius
125 
126         def volume(self):
127 
128             return (4./3.)*pi*self.radius**3
129 
130         def area(self):
131 
132             return pi*self.radius**2
133 
134         def mass(self):
135 
136             return self.volume()*self.rho
137 
138         def Pvap(self, freestream):
139 
140             Tref = (2./3.)*self.T + (1./3.)*freestream.
                    Tambient
141 
142             a1 = -7.85951783
```

```
143             a2 = 1.84408259
144             a3 = -11.7866497
145             a4 = 22.6807411
146             a5 = -15.9618719
147             a6 = 1.80122502
148 
149             tau = 1. - Tref/self.Tcrit
150 
151             pOverPc = exp((a1*tau + a2*tau**1.5 + a3*tau**3
                    + a4*tau**3.5 + a5*tau**4 + a6*tau**7.5)*
                    self.Tcrit/self.T)
152 
153             return pOverPc*22064.
154 
155         def We(self, freestream):
156 
157             vRel = self.velocity.rVector(freestream.
                    velocity)
158 
159             return freestream.rho*dot(vRel, vRel)*self.
                    radius/self.sigma
160 
161         def Re(self, freestream):
162 
163             return freestream.rho*(freestream.velocity -
                    self.velocity).mag()*self.diameter()/
                    freestream.mu
164 
165         def dragCoefficient(self, freestream):
166 
167             # This drag coefficient for a sphere is based
                    on the correlation of
168             # F.A. Morrison in "An Introduction to Fluid
                    Mechanics"
169 
170             Re = self.Re(freestream)
171 
```

```
172             return 24./Re + 2.6*(Re/5.)/(1. + (Re/5.)
                    **1.52) + 0.411*(Re/263000.)**-7.94/(1. + (
                    Re/263000.)**-8.) + (Re**0.8/461000.)
173
174         def dragForce(self, freestream):
175
176             Cd = self.dragCoefficient(freestream)
177
178             vRel = freestream.velocity - self.velocity
179
180             return vRel.unitVector()*0.5*freestream.rho*dot
                    (vRel, vRel)*Cd*self.area()
181
182         def acceleration(self, freestream):
183
184             return self.dragForce(freestream).scale(1./self
                    .mass()) + freestream.gravity
185
186         def advectEuler(self, freestream, dt):
187
188             a = self.acceleration(freestream)
189
190             self.position += self.velocity*dt + a*0.5*dt**2
191             self.velocity += a*dt
192
193         def advectPredictorCorrector(self, freestream, dt):
194
195             originalPosition = self.position
196
197             a = self.acceleration(freestream)
198             f1 = self.velocity + a*0.5*dt
199
200             self.position += f1*dt
201
202             a = self.acceleration(freestream)
203             f2 = self.velocity + a*0.5*dt
204
```

```
205             self.position = originalPosition + (f1 + f2)
                    *0.5*dt
206             self.velocity += a*dt
207
208         def printAll(self):
209
210             print "Radius:", self.radius
211             print "Rho:", self.rho
212             print "mu:", self.mu
213             print "sigma:", self.sigma
214             print "Boiling Temp:", self.boilingTemp
215             print "Latent Heat:", self.latentHeat
216             print "Specific Heat:", self.specificHeat
217             print "k:", self.k
218             print "Position:", self.position
219             print "Velocity:", self.velocity
220
221     class DropletInlet(object):
222
223         def __init__(self, newDropletFrequency = 1000,
                inletWidth = 0.005, velocityDeviation = 0.):
224
225             self.newDropletFrequency = newDropletFrequency
226             self.inletWidth = inletWidth
227             self.velocityDeviation = velocityDeviation
228             self.timeSinceLastDroplet = 0.
229             self.newDropletPeriod = 1./self.
                    newDropletFrequency
230             self.dropsAdded = 0
231
232         def addDrops(self, initialDroplet, droplets, dt):
233
234             self.timeSinceLastDroplet += dt
235
236             nDropsToAdd = int(self.timeSinceLastDroplet/
                    self.newDropletPeriod)
237             self.timeSinceLastDroplet -= float(nDropsToAdd)
                    *self.newDropletPeriod
```

```
238
239             if nDropsToAdd > 0:
240                 self.dropsAdded += nDropsToAdd
241
242             for i in range(0, nDropsToAdd):
243
244                 droplets.append(cp.deepcopy(initialDroplet)
                        )
245                 droplets[-1].position.x += random.uniform
                        (-0.5*self.inletWidth, 0.5*self.
                        inletWidth)
246                 droplets[-1].velocity.x += random.
                        normalvariate(0., self.velocityDeviation
                        .x)
247                 droplets[-1].velocity.y += random.
                        normalvariate(0., self.velocityDeviation
                        .y)
```

C boTAB Code - Input Module

```
1
2 # -*- coding: utf-8 -*-
3
4 """
5 boTAB
6 =====
7 This module is for simple file input and configuration
8 Author: Adam O'Brien
9 """
10
11 from fluid import Vector
12
13 def str2num(string):
14
15     if string.partition("/")[1] == "/":
16
17         string = string.partition("/")
18
19         return float(string[0])/float(string[2])
20
21     elif string.partition(",")[1] == ",":
22
23         string = string.partition(",")
24
25         return Vector(float(string[0]), float(string
               [2]))
26
27     else:
28
29         try:
30
31             return int(string)
32
33         except ValueError:
34
```

```
35              return float(string)
36
37
38  def process(inputDict, line):
39
40      line = line.replace("\n", "")
41      line = line.replace(" ", "")
42      line = line.partition("#")
43      line = line[0]
44
45      if line == "":
46          return
47
48      line = line.partition("=")
49
50      if not line[1] == "=":
51          print "Warnining, potentially bad input:", line
52
53      inputDict[line[0]] = str2num(line[2])
54
55      return
56
57  def readInputFile(filename = "config.in"):
58
59      userInput = {}
60
61      print "Reading from input file \"config.in\"..."
62
63      with open(filename) as inFile:
64
65          for line in inFile:
66
67              process(userInput, line)
68
69      print "The following input parameters have been
            loaded from \"{}\"".format(filename), "\n"
70
71      for parameter in userInput:
```

```
72
73              print parameter, "=", userInput[parameter]
74
75      print "Finished reading from input file \"config.in\"."
76
77      return userInput
78
79  def setObjectParametersFromInput(userInput, freestream, droplet, inlet):
80
81      # Freestrean properties
82
83      freestream.rho = userInput["freestreamRho"]
84      freestream.mu = userInput["freestreamMu"]
85      freestream.Tambient = userInput["freestreamTambient"]
86      freestream.Cp = userInput["freestreamCp"]
87      freestream.K = userInput["freestreamK"]
88      freestream.velocity = userInput["freestreamVelocity"]
89      freestream.gravity = userInput["freestreamGravity"]
90      freestream.M = userInput["freestreamM"]
91
92      # Initial droplet properties
93
94      droplet.radius = userInput["dropletRadius"]
95      droplet.rho = userInput["dropletRho"]
96      droplet.mu = userInput["dropletMu"]
97      droplet.sigma = userInput["dropletSigma"]
98      droplet.Tboil = userInput["dropletTboil"]
99      droplet.L = userInput["dropletL"]
100     droplet.Cp = userInput["dropletCp"]
101     droplet.K = userInput["dropletK"]
102     droplet.position = userInput["dropletPosition"]
103     droplet.velocity = userInput["dropletVelocity"]
104     droplet.M = userInput["dropletM"]
105
```

```
106     # Droplet inlet properties
107
108     inlet.newDropletFrequency = userInput["
            inletDropletCreationFrequency"]
109     inlet.inletWidth = userInput["inletWidth"]
110     inlet.velocityDeviation = userInput["
            inletVelocityDeviation"]
```

D boTAB Code - Output Module

```
1
2 # -*- coding: utf-8 -*-
3
4 """
5 boTAB
6 =====
7 This module is for various plotting outputs
8 Author: Adam O'Brien
9 """
10
11 import matplotlib.pyplot as plt
12 import matplotlib.animation as animation
13
14 def plotDroplets(droplets):
15
16     xcoords = [droplet.position.x for droplet in
        droplets]
17     ycoords = [droplet.position.y for droplet in
        droplets]
18     radii = [50000.*droplet.radius for droplet in
        droplets]
19
20     plt.axis('equal')
21     plt.grid(True)
22     plt.xlabel('x (m)', fontsize=16)
23     plt.ylabel('y (m)', fontsize=16)
24
25     plt.scatter(xcoords, ycoords, s=radii, alpha=0.5)
26
27     plt.show()
```

E boTAB Code - Evaporation Module

```
1
2 # -*- coding: utf-8 -*-
3
4 """
5 boTAB
6 =====
7 This module contains classes and function for the
      modelling of droplet
8 evaporation in freestream flows
9 Author: Adam O'Brien
10 """
11
12 from math import log, sqrt, fabs, pi
13
14 def clausiusClapeyron(freestream, droplet):
15
16     # This function determines the water vapour mass
          fraction at the surface
17     # of a droplet
18
19     return 1./(1. + freestream.Pambient*freestream.M/((
          droplet.Pvap(freestream) - 1.)*droplet.M))
20
21 def evaporate(freestream, droplets, dt):
22
23     for i in range(0, len(droplets)):
24
25         Yls = clausiusClapeyron(freestream, droplets[i
              ])
26
27         BM = Yls/(1. - Yls)
28
29         Re = droplets[i].Re(freestream)
30         Pr = freestream.Pr
31
```

```
32          gamma = 8.*freestream.k*log(1. + BM)/(
                freestream.Cp*droplets[i].rho)*(1. + 0.3*Re
                **0.5*Pr**(1./3.))
33
34          D2 = droplets[i].diameter()**2 - gamma*dt
35
36          if D2 > 0.:
37
38              droplets[i].radius = 0.5*D2**0.5
39
40          else:
41
42              droplets[i].radius = 0.
43
44      # Discard any droplets that have a radius less than
            the tolerance, ie they
45      # are completely evaporated
46
47      droplets[:] = [droplet for droplet in droplets if
          not droplet.radius <= 1e-10]
```

F boTAB Code - TAB Module

```
1
2 # -*- coding: utf-8 -*-
3
4 """
5 boTAB
6 =====
7 This module contains the functions necessary for TAB
      breakup
8 Author: Adam O'Brien
9 """
10
11 from math import exp, sin, cos, fabs
12 import numpy as np
13 import copy as cp
14 from fluid import *
15
16 # Model constants
17
18 Cb = 0.5
19 Ck = 8.
20 Cd = 5.
21 Cf = 1./3.
22 Cd = 5.
23 K = 10./3.
24 Cv = 1.
25
26 def getSMR(droplet):
27
28     # This function computes the Sauter Mean Radius (
          SMR) of the child droplets after a break-up
29
30     rOverRmean = 1. + (K/5.)*Ck*Cb**2 + droplet.rho*
          droplet.radius**3/droplet.sigma*Cb**2*droplet.
          dydt**2*(6.*K - 5.)/30.
31
```

```
32      return droplet.radius/rOverRmean
33
34  def breakupTab(freestream, droplets, dt):
35
36      newDroplets = []
37
38      for droplet in droplets:
39
40          Wec = droplet.We(freestream)*Cf/(Ck*Cb)
41          td = droplet.rho*droplet.diameter()**2/(2.*Cd*
                droplet.mu)
42          omega = (8.*Ck*droplet.sigma/(droplet.rho*
                droplet.diameter()**3) - (1./td**2))**0.5
43
44          yn = droplet.y
45
46          droplet.y = Wec + exp(-dt/td)* \
47                      ((yn - Wec)*cos(omega*dt) + 1./
                           omega*(droplet.dydt + (yn - Wec)
                           /td)*sin(omega*dt))
48
49          droplet.dydt = (Wec - droplet.y)/td + \
50                         omega*exp(-dt/td)* \
51                         (1./omega*(droplet.dydt + (yn -
                              Wec)/td)*cos(omega*dt) - (yn
                              - Wec)*sin(omega*dt))
52
53          if droplet.y >= 1.:
54
55              rSmr = getSMR(droplet)
56
57              # Begin sampling droplets
58
59              volOfNewDrops = 0.
60
61              while True:
62
```

```
63                  newRadius = np.random.normal(rSmr, 0.2*
                        rSmr)
64
65                  if newRadius < 0.:
66
67                      continue
68
69                  randomNo = np.random.uniform(-1, 1)
70
71                  newVelocity = droplet.velocity +
                        droplet.velocity.normalVector().
                        unitVector()*Cb*droplet.radius*
                        droplet.dydt*(randomNo/fabs(randomNo
                        ))
72
73                  newDroplet = Droplet(newRadius, cp.copy
                        (droplet.position), cp.copy(
                        newVelocity))
74
75                  newDroplets.append(newDroplet)
76
77                  volOfNewDrops += newDroplets[-1].volume
                        ()
78
79                  if volOfNewDrops >= droplet.volume():
80
81                      break
82
83      # Remove the old parent droplets
84
85      droplets[:] = [droplet for droplet in droplets if
            droplet.y < 1.]
86
87      # Add the newly created child droplets
88
89      for droplet in newDroplets:
90
91          droplets.append(cp.deepcopy(droplet))
```

```
92
93      # Return the number of droplets created
94
95      return len(newDroplets)
```