A Quantitative Analysis Aerodynamic Forces on Various Cambered Airfoils

Joel Howard

Introduction

One of the main factors that affects the performance of an aircraft is the lift-to-drag (L/D) ratio, which determines the fuel economy, glide ratio, and climb rate, among other performance metrics. Due to the importance of this ratio, much effort has been placed into optimizing the shape of a wing (airfoil) to maximize the L/D ratio. In this paper, we perform wind-tunnel simulations using various airfoil designs to determine the effects of thickness, camber (curvature), and angle of attack on the lift-to-drag ratio achieved. To do so, we use a basic implementation of the Smooth Particle Hydrodynamics (SPH) mesh-free formulation of compressible fluid dynamics, along with a simulated test chamber in which the particles can interact both with the walls and with the object.

SPH Theory

SPH is a Langrangian code that follows individual particles (as opposed to maintaining a grid), using a smoothing function to simulate the effects of pressure and density gradients. In this implementation, the density of each particle is given by a weighted average of the locations of nearby particles. This continuous approximation is given by

$$\rho_j = \sum_i m_i W_{ji}$$

where i sums over all particles, the mass is normalized to 1 for all particles, and the smoothing kernel W_{ji} (as a function of the distance between two particles) is given by

$$W(r,h) = \frac{1}{\pi h^3} \begin{cases} 1 - \frac{3}{2}x^2 + \frac{3}{4}x^3 & 0 \leq x \leq 1 \\ \frac{1}{4}(2-x)^3 & 1 \leq x \leq 2 \\ 0 & x \geq 2 \end{cases}$$

Notation: 2h is the maximum distance over which a particle exerts an influence, and $x = r/h$.

The equations of motion of the system are given by the conservation of momentum equation

$$\frac{dv_j}{dt} = -\sum_i m_i \left(\frac{P_j}{\rho_j^2} + \frac{P_i}{\rho_i^2}\right) \nabla W_{ji}$$

along with the equation of state, relating pressure (P) to density (rho):

$$P = \kappa(s)\rho^\gamma$$

For simplicity, this simulation only considers isentropic flow, allowing $\kappa(s)$ to be constant (normalized to 1). Furthermore, considering the air particles as a monatomic ideal gas, the adiabatic index is set to 5/3 (Cossins).

Cambered Airfoils

The shape of an airfoil determines how well it generates both lift and drag, with optimal design maximizing lift and minimizing drag. The simplest airfoils developed by the National Advisory Committee for Aeronautics (NACA), called the '4-digit' series, use four parameters to determine the shape of a cambered airfoil: chord length, maximum camber, location of maximum camber, and maximum thickness. The chord length (c) is the length of the straight line running between the front and rear of the airfoil, who's angle above the horizontal defines the angle of attack. This is not to be confused with the camber line, which is moves between the same two points, but which is equidistant from the upper and lower surfaces. The maximum camber (m) determines the degree of asymmetry between the upper and lower surface of the airfoil, with a camber of 0 defining a symmetrical airfoil (Marzocca). The location of the maximum camber (p), as well as the maximum thickness (t), are both given as a percentage of the chord length. Using these four parameters, the equation of a NACA 4-digit airfoil is given by:

$$x_U = x - y_t sin(\theta),\ y_U = y_C + y_t cos(\theta)$$
$$x_L = x + y_t sin(\theta),\ y_L = y_C - y_t cos(\theta)$$
$$y_C = \begin{cases} m\frac{x}{p^2}(2p - \frac{x}{c}) & 0 \leq x \leq pc \\ m\frac{c-x}{(1-p)^2}(1 + \frac{x}{c} - 2p) & pc \leq x \leq c \end{cases}$$
$$y_T = 5tc[.2969\sqrt{\frac{x}{c}} - .1260\frac{x}{c} - .3516(\frac{x}{c})^2 + .2843(\frac{x}{c})^3 - .1015(\frac{x}{c})^4]$$

Notation: x refers to the distance along the chord line, x_U, y_U give the coordinates of the upper surface, and x_L, y_L give the coordinates of the lower surface.

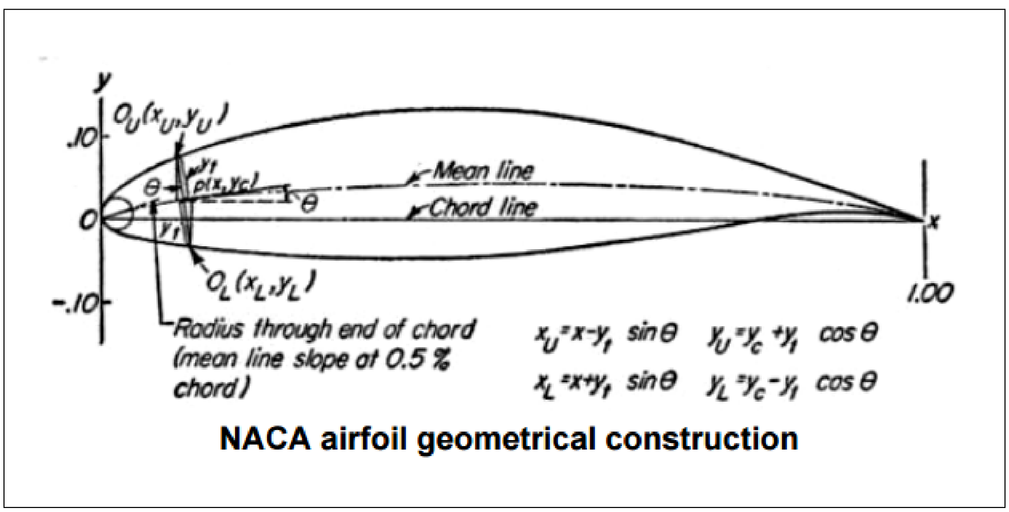

Figure 1: NACA 4-Digit Airfoil

The first digit of an airfoil's name is denotes m, the second digit p, and the final two digits are reserved for t. For example, an airfoil with a 1% camber located 30% of the way along the chord, and a maximum thickness of 12% would be the NACA 1312. The airfoils chosen were the NACA 0009, which is symmetric and thin, the NACA 6409, which has a 6% camber, and the NACA 6424, which is thick. Each of these were tested at angle of attacks of 0, 15, and 30 degrees above the horizontal. The first two differ only in camber amount, and the second two differ only in thickness, so this suite of three airfoil designs will allow us to determine the effect of both camber and thickness on an airfoil's aerodynamic properties.

NACA airfoils used:

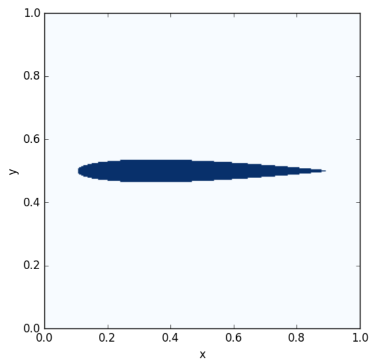

Figure 2: NACA 0009

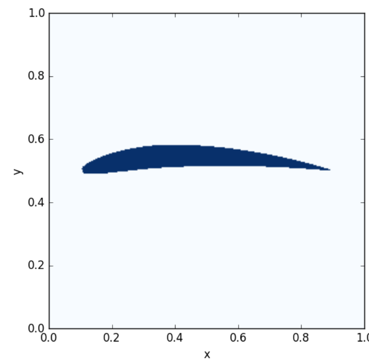

Figure 3: NACA 6409

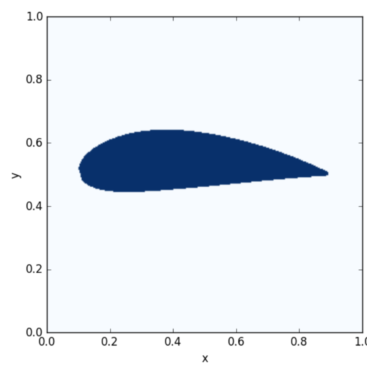

Figure 4: NACA 6424

Implementation Details

The basic experiment, using a Python implementation of the SPH code, involves placing an arbitrary airfoil in a wind tunnel and measuring the resulting aggregate lift and drag forces. This is accomplished by randomly generating a distribution of particles that interact with each other, the airfoil, and the top, left, and bottom walls. When a particle hits the right wall, it disappears and is regenerated on the left wall with a positive x-velocity and random y-velocity. The randomness of placement and velocity simulates directional, yet still stochastic, motion.

Lift and drag forces are calculated by summing the momentum transfer of a particle each time it collides with the airfoil, and dividing by the total number of time steps. As only the ratio of lift/drag force is of interest, no attention was paid to making the actual masses or velocities of the particles realistic.

The evolution of the system involves first determining the acceleration on each particle due to each other particle according to the equations of motion, then updating the positions of each particle, and finally determining the new density of each particle, again requiring an iteration over all other particles. This particular implementation, then, goes as $O(n^2)$, though the author will note that use of more advanced data structures (such as AVL trees or linked lists) could drop the complexity to $O(nLog(n))$, as particles only need to check other particles within the maximum interaction range of $2h$.

For these simulations, each particle was given a smoothing length of h=.1, thus interacting with a rough maximum of 10% of the particles in the simulation at any given time. Furthermore, due to computational limitations, 400 particles were used, with 1000 time steps. To justify these choices, we found that the lift to drag ratio for 400 particles asymptotes to $\pm 5\%$ of a value within about 400 time steps, shown in Fig. 5. Thus, to accommodate variation due to the randomized particle generation all simulations were performed three times, the average of which for each airfoil configuration is reported in Fig. 6.

The walls of the simulation, as well as the airfoil itself, are implemented as a boundary of connected lines, each of which responds to an incident particle like a flat surface. Every time the particle updates its velocity, it checks to see if it falls within the "realm" of each flat surface, defined by the minimum and maximum x and y values of the object (essentially a rectangle). Only when within the realm of the object does the particle iterate over all points of the object, determining which line segment it will bounce off of. Thus, the computational cost of including a single object goes roughly as $n*m*a$, where n is the number of particles, m is the number of boundary points, and a is the fractional area that the object's "realm" encompasses.

A more detailed walk-through of this Python SPH implementation is given in the appendix.

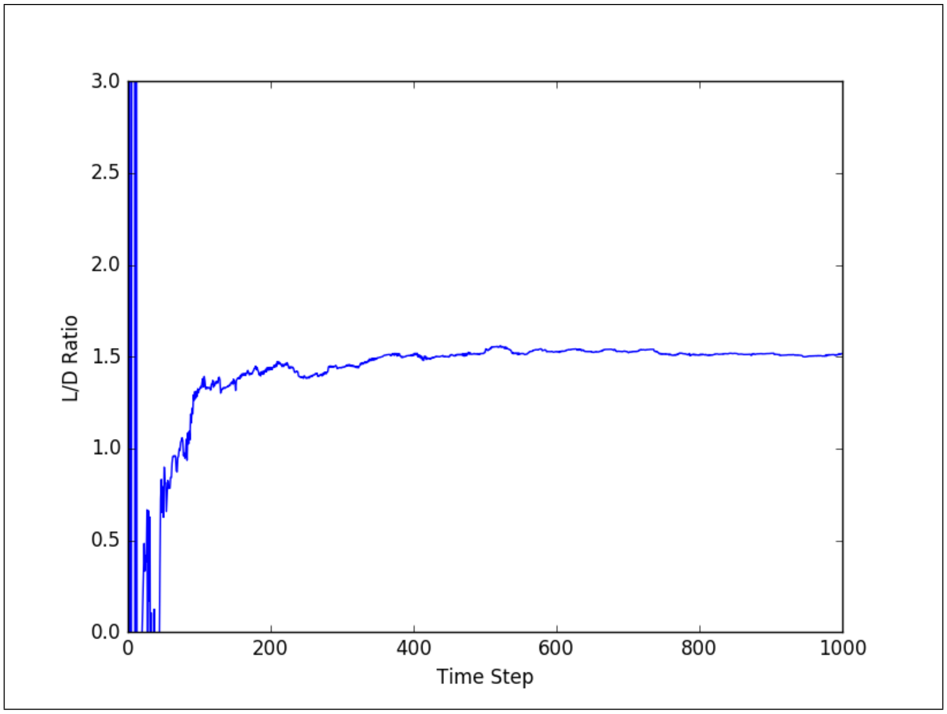

Figure 5: L/D Ratio vs. Time step. It asymptotes within around 400 time steps.

Results

As shown in Fig. 6, all airfoils attained a maximum L/D ratio at a 15-degree angle of attack, with the symmetric airfoil, 0009, performing the best. As expected, a zero degree angle of attack produced a nominal L/D ratio, as to first order, due to the lack of vortices, these airfoils approximate the effects of a flat plane responding to incident particles.

The camber comparison, between 0009 and 6409, reveals that, to within error, changing the camber from 0 to 6% had little effect.

Varying the thickness (6409 and 6424), however, dramatically reduced the L/D ratio, due most likely to the increased cross-section of the airfoil tip (which produces most of the drag).

Name	Lift/Drag	m	p	t	Angle
"0009"	0.1266666667	0	0	0.09	0
"0009"	2.25	0	0	0.09	15
"0009"	1.44	0	0	0.09	30
"6409"	-0.2933333333	6	4	0.09	0
"6409"	2.2	6	4	0.09	15
"6409"	1.49	6	4	0.09	30
"6424"	-0.19	6	4	0.24	0
"6424"	1.29	6	4	0.24	15
"6424"	1.103333333	6	4	0.24	30

Figure 6: Results. Each line is the average of three trials.

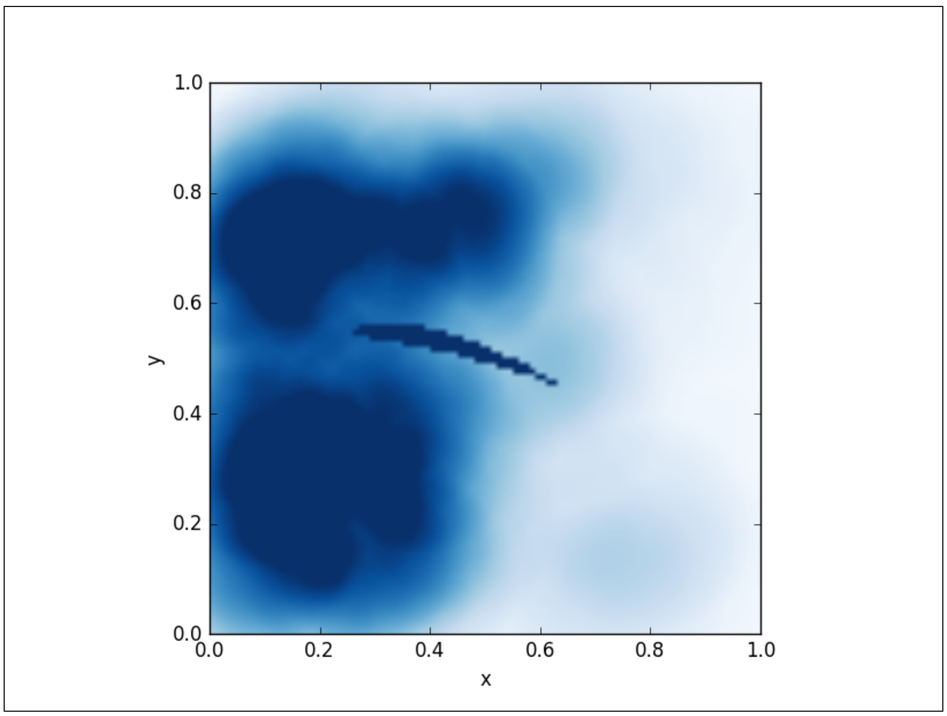

Figure 7: One frame of an example simulation output.

Pitfalls

The greatest sources of threats to validity, in no particular order, for these results are as follows:

- Boundary Force: As implemented, the particles act as continuous density distributions when interacting with each other, but as billiard balls when interacting with boundaries. In high pressure areas, this still generates the intended effect of higher collisions with a nearby boundary, but is far less rigorous than the model proposed by Monaghan and Kos (Monaghan), in which the force exerted on a boundary is proportional instead to the density distribution of inbound particles, and requires more careful particle spacing on boundaries.

- Floor/Ceiling Effects: The floor and ceiling were implemented in order to keep particles in the chamber long enough for them to bounce off of each other in a random fashion. Because these airfoils do not exhibit rotational symmetry, however, parity between the dynamics in the upper and lower half of the simulation diminishes as the angle of attack increases. This effect should go to zero as the relative width of the airfoil to the width of the chamber decreases, but doing so requires a comparatively larger chamber with far more particles.

- Viscosity: This simulation does not include fluid viscosity, which is important in generating circulation patters around the airfoil that lead to vortices. These vortices, due to conservation of angular momentum, cause the airfoil to experience slightly higher airspeeds over the upper surface, which is critical to the Bernouilli formulation of how lift works (Nakamura).

- Number of Particles: It very well may be that these simulations may not have nearly enough particles or time steps to distill out all of the complex effects that tie in to generating lift. To handle significantly more of either, parallelization of the code would be required, a skill yet to be developed by the author.

- Boundary Penetration: Particles interact across the boundary of the object due to its width being small compared to the size of h. This means the object, while effective at deflecting particles, doesn't accurately separate fluids in different regions from each other. Unfortunately, the smoothing length can only be so small before this just becomes a billiard-ball simulation, and the airfoil can only be so thick relative to the size of the domain before the floor/ceiling effects become too prominent.

Works Cited

1. Cossins, Peter J. "Smoothed Particle Hydrodynamics." The Gravitational Instability and Its Role in the Evolution of Protostellar and Protoplanetary Discs. U of Leicester. Print.

2. Marzocca, Pier. "The NACA Airfoil Series." Clarkson University. Web. <http://people.clarkson.edu/~pmarzocc/AE429/The NACA airfoil series.pdf>.

3. Nakamura, Mealani. "How an Airfoil Works." Reports on How Things Work. Massachusetts Institute of Technology, 1999. Web. <http://web.mit.edu/2.972/www/reports/airfoil/airfoil.html>.

4. Monaghan, J. J., and A. Kos. "Solitary Waves on a Cretan Beach." J. Waterway, Port, Coastal, Ocean Eng. Journal of Waterway, Port, Coastal, and Ocean Engineering 125.3 (1999): 145-55. Web.

Works Referenced

1. M. Muller, D. Charypar, and M. Gross . Particle-based fluid simulation for interactive applications, in Proceedings of Eurographics/SIGGRAPH Symposium on Computer Animation.

2. Singh, Anand Pratap. Smoothed Particle Dynamics. Tech. Mumbai: Department of Aerospace Engineering, 2010. Print.

3. Denker, John S. See How It Flies: Perceptions, Procedures & Principles of Flight. McGraw, 1995. Print.

A Code Used

```
from pylab import *
from matplotlib.widgets import Slider, Button, RadioButtons
import matplotlib.pyplot as plt
import numpy as np
from sympy import *
import matplotlib.animation as animation
import time
import matplotlib.path as mplPath

#Variables
resolution=100#The pixel resolution of the animation.
mass=.1#An arbitrary mass assigned to each particle
wallbuffer=.02;#The buffer around the walls and the object such that particles don't "hop" over them.
timeSteps=1000;
numParticles=400;
dT=.001#The dt time step.
h = .1#The smoothing length used.
animationTimeInterval=10#Determines the speed of the animation
startingVelocity=8#The starting velocity of all particles.
kickVelocity=8#The x- and y- velocities given to the particles when teleported to the left wall.
xMax=1.0;#Normalized boundaries.
yMax=1.0;#...
gamma=5.0/3#Monatomic gas adiabatic index.
numAttributes=6;#Setting up the multi-dimensional particle matrix.
xPosition=0;#...
yPosition=1;#...
xVelocity=2;#...
yVelocity=3;#...
density=4;#...
pressure=5;#...
pM=np.zeros((numParticles,numAttributes,timeSteps))#pM is the "particle Matrix", it stores all information about each particle.
numBoundaryPoints=200#Number of points defining the airfoil boundary, creating a closed contour connected by straight lines.
oM=np.zeros((numBoundaryPoints,3))#Matrix storing x,y coordinates of the airfoil boundary
liftDragRatio=np.zeros(timeSteps)#A matrix storing the lift/drag ratio as a function of time.
#Airfoil Parameters
c=.4#Chord length
m=.06#Maximum camber
p=.4#Location of maximum camber
t=.24#Maximum thickness
rotationAngle=30#Angle of attack
xStart=.25#x,y coordinates of the leading edge of the airfoil
yStart=.5#...
objectMomentum=np.array([0.0,0.0])#Stores the cumulative momentum imparted to the airfoil.
#Calculate the kernel using Sympy, then lambdify it to make the calculations faster. Used in 'kernel' and 'gradkernel' functions
x1 = Symbol('x1')
y1 = Symbol('y1')
x2 = Symbol('x2')
y2 = Symbol('y2')
r=((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1))**.5
kernelClose=(1/(h**3*math.pi))*(1-1.5*(r/h)**2+0.75*(r/h)**3)
```

```python
kernelMed=(1/(h**3*math.pi))*(0.25*(2-r/h)**3)
gradKernelClosex2=kernelClose.diff(x2)
gradKernelClosey2=kernelClose.diff(y2)
gradKernelMedx2=kernelMed.diff(x2)
gradKernelMedy2=kernelMed.diff(y2)
kernelClose=lambdify((x1,y1,x2,y2),kernelClose,"numpy")
kernelMed=lambdify((x1,y1,x2,y2),kernelMed,"numpy")
gradKernelClosex2=lambdify((x1,y1,x2,y2),gradKernelClosex2,"numpy")
gradKernelClosey2=lambdify((x1,y1,x2,y2),gradKernelClosey2,"numpy")
gradKernelMedx2=lambdify((x1,y1,x2,y2),gradKernelMedx2,"numpy")
gradKernelMedy2=lambdify((x1,y1,x2,y2),gradKernelMedy2,"numpy")
#Functions

def generateAirfoil(c,m,p,t,numBoundaryPoints,oM,xStart,yStart,rotationAngleDegrees):
  for i in range(0,int(numBoundaryPoints/2)):#Iterate over half of the specified number of boundary points.
    x=(i/(numBoundaryPoints/2))*.99*c+.005*c#Making sure the first and last points aren't directly on top of each other.
    yt=5*t*c*(.2969*(x/c)**.5-.1260*x/c-.3516*(x/c)**2+.2843*(x/c)**3-.1015*(x/c)**4)#Run the airfoil calculation
    if (x<=p*c):#...
      yc=(m*x/p**2)*(2*p-x/c)#...
      dyc=2*m/p**2*(p-x/c)#...
    else:#...
      yc=m*(c-x)/(1-p)**2*(1+x/c-2*p)#...
      dyc=2*m/(1-p)**2*(p-x/c)#...
    theta=np.arctan(dyc)#...
    xUpper=x-yt*math.sin(theta)+xStart#...
    yUpper=yc+yt*math.cos(theta)+yStart#...
    xLower=x+yt*math.sin(theta)+xStart#...
    yLower=yc-yt*math.cos(theta)+yStart#...
    oM[i,0]=xUpper#Save the values on opposite sides of the object matrix, as the object matrix needs the points to be clockwise consecutive.
    oM[i,1]=yUpper#...
    oM[numBoundaryPoints-1-i,0]=xLower#...
    oM[numBoundaryPoints-1-i,1]=yLower#...
#Rotate by angle of attack
  centerPoint=np.array([xStart+.5*c,yStart])
  rotationAngle=-1*rotationAngleDegrees*2*math.pi/360.0 #-1 due to rotating clockwise
  rotationMatrix=np.array([[np.cos(rotationAngle),-1*np.sin(rotationAngle)],[np.sin(rotationAngle),np.cos(rotationAngle)]])#Rotate clockwise
  for i in range(0,numBoundaryPoints):#Just geometry to get the correct rotation.
    relativePoint=[oM[i,0],oM[i,1]]-centerPoint
    newRelativePoint=np.dot(rotationMatrix,relativePoint)
    newPoint=newRelativePoint+centerPoint
    oM[i,0]=newPoint[0]
    oM[i,1]=newPoint[1]

def currentDensity(j,t):#Function to update the density of a certain particle j at time t.
  rho=0
  for i in range(0,numParticles):#Iterate over all particles
    x1=pM[i,xPosition,t]
    y1=pM[i,yPosition,t]
    x2=pM[j,xPosition,t]
    y2=pM[j,yPosition,t]
    radiusCalcSquared=(x2-x1)*(x2-x1)+(y2-y1)*(y2-y1)
```

```
    if (radiusCalcSquared<(2*h)*(2*h)):#If within the kernel range...
      rho+=kernel(pM[i,xPosition,t],pM[i,yPosition,t],pM[j,xPosition,t],pM[j,yPosition,t],radiusCalcSquared)#Perform the sum of the kernels.
  return mass*rho;
def kernel(x1Input,y1Input,x2Input,y2Input,radiusSquared):#Function that calculates the piecewise kernel function.
  radius=math.sqrt(radiusSquared)#Square roots take so long to compute! This one line is killing my code :(
  if(radius>=0 and radius<h):
    return kernelClose(x1Input,y1Input,x2Input,y2Input)
  elif(radius>=h and radius<2*h):
    return kernelMed(x1Input,y1Input,x2Input,y2Input)
  else:
    return 0
def gradkernel(x1Input,y1Input,x2Input,y2Input,radiusSquared):#Function that calculates the gradient of the piecewise kernel function.
  radius=math.sqrt(radiusSquared)
  accel=np.array([0.0,0.0])
  if(radius>=0 and radius<h):
    accel[0]=gradKernelClosex2(x1Input,y1Input,x2Input,y2Input)
    accel[1]=gradKernelClosey2(x1Input,y1Input,x2Input,y2Input)
    return accel
  elif(radius>=h and radius<2*h):
    accel[0]=gradKernelMedx2(x1Input,y1Input,x2Input,y2Input)
    accel[1]=gradKernelMedy2(x1Input,y1Input,x2Input,y2Input)
    return accel
  else:
    return 0
def newVelocity(j,t,accel):#Function that updates the velocity of particle j at time t given the calculated net acceleration on the particle.
  if(abs(pM[j,xPosition,t-1])<wallbuffer):#If close to the left wall, bounce off.
    return np.array([abs(pM[j,xVelocity,t-1]),pM[j,yVelocity,t-1]])
  elif(abs(xMax-pM[j,xPosition,t-1])<wallbuffer):#If close to the right wall, teleport to the left wall. Wind tunnel!
    pM[j,xPosition,t-1]=0
    pM[j,yPosition,t-1]=rand()
    return np.array([kickVelocity,(2*rand()-1)*kickVelocity])#random y=velocity. Particles leave in 45 degree cone.
  elif(abs(pM[j,yPosition,t-1])<wallbuffer):#If close to the bottom, bounce off.
    return np.array([pM[j,xVelocity,t-1],abs(pM[j,yVelocity,t-1])])
  elif(abs(yMax-pM[j,yPosition,t-1])<wallbuffer):#If close to the top, bounce off.
    return np.array([pM[j,xVelocity,t-1],-1*abs(pM[j,yVelocity,t-1])])
  #Otherwise, if it enters the "possibly hitting the object" zone...
  elif(pM[j,xPosition,t-1]>=xMinObject-wallbuffer and pM[j,xPosition,t-1]<=xMaxObject+wallbuffer and pM[j,yPosition,t-1]>=yMinObject-wallbuffer and pM[j,yPosition,t-1]<=yMaxObject+wallbuffer):#If close to the object...
    x0=pM[j,xPosition,t-1]
    y0=pM[j,yPosition,t-1]
    for p in range(0,numBoundaryPoints):#Figure out which two adjacent points on the boundary define a line that it'll bounce off of. Geometry...
      currentPoint=p#...
      nextPoint=(p+1)%numBoundaryPoints#Boundary points define a closed loop, so modulus connects first and last point.
      x1=oM[currentPoint,0]#Current and next point coordinates
      y1=oM[currentPoint,1]
      x2=oM[nextPoint,0]
      y2=oM[nextPoint,1]
```

```
            distanceBetweenPoints=oM[currentPoint,2]
            distanceToLine=abs((y2-y1)*x0-(x2-x1)*y0+x2*y1-y2*x1)/distanceBetweenPoints
            distanceToLineSquared=distanceToLine*distanceToLine
            distanceBetweenPointsSquared=distanceBetweenPoints*distanceBetweenPoints
            distanceX0P1Squared=(x1-x0)*(x1-x0)+(y1-y0)*(y1-y0)
            distanceX0P2Squared=(x2-x0)*(x2-x0)+(y2-y0)*(y2-y0)
            #Determine if the incident particle falls in the rectangle defined by the line between the two boundary points, and the wallbuffer values extending away from that line in both directions. If so, the particle bounces off that wall.
            if (distanceToLine<=wallbuffer and distanceX0P1Squared-distanceToLineSquared<=distanceBetweenPointsSquared and distanceX0P2Squared-distanceToLineSquared<=distanceBetweenPointsSquared):#If bouncing off p,p+1...
                vX0=pM[j,xVelocity,t-1]
                vY0=pM[j,yVelocity,t-1]
                theta=np.arctan(vY0/vX0)
                alpha=np.arctan((y2-y1)/(x2-x1))
                rotationAngle=2*(alpha-theta)
                rotationMatrix=np.array([[np.cos(rotationAngle),-1*np.sin(rotationAngle)],[np.sin(rotationAngle),np.cos(rotationAngle)]])
                newVelocity=np.dot(rotationMatrix,[vX0,vY0])#Reflect the object via angle in=angle out.
                newObjectMomentum=objectMomentum-mass*(newVelocity-[vX0,vY0])#Calculate momentum transferred to the object
                objectMomentum[0]=newObjectMomentum[0]
                objectMomentum[1]=newObjectMomentum[1]
                return newVelocity
        return np.array([pM[j,xVelocity,t-1]+accel[0]*dT,pM[j,yVelocity,t-1]+accel[1]*dT])#Otherwise, return v+at
    else:
        return np.array([pM[j,xVelocity,t-1]+accel[0]*dT,pM[j,yVelocity,t-1]+accel[1]*dT])#Otherwise, return v+at




#Create the object
generateAirfoil(c,m,p,t,numBoundaryPoints,oM,xStart,yStart,rotationAngle)
xMinObject=np.min(oM[:,0])#Create the "might be hitting the airfoil" zone simply based on the max and min x,y values. A rectangle.
xMaxObject=np.max(oM[:,0])
yMinObject=np.min(oM[:,1])
yMaxObject=np.max(oM[:,1])
for p in range(0,numBoundaryPoints):#Save the distances between adjacent points, so we only need to calculate the darn square roots once!
    currentPoint=p
    nextPoint=(p+1)%numBoundaryPoints
    x1=oM[currentPoint,0]
    y1=oM[currentPoint,1]
    x2=oM[nextPoint,0]
    y2=oM[nextPoint,1]
    oM[currentPoint,2]=math.sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1))
#Initialize particle matrix
for i in range(0,numParticles):
    x=rand()
    y=rand()
```

```
    while(x>=xMinObject and x<=xMaxObject and y>=yMinObject and y<=yMaxObject):
      #Generate random positions OUTSIDE the bounds of the object
      x=rand()
      y=rand()
    pM[i,xPosition,0]=x
    pM[i,yPosition,0]=y
    pM[i,xVelocity,0]=startingVelocity*(2*rand()-1)#x,y velocities start out totally randomized, but with magnitude startingVelocity.
    pM[i,yVelocity,0]=startingVelocity*(2*rand()-1)
for i in range(0,numParticles):#Now that we have the position values, compute the densities
    pM[i,density,0]=currentDensity(i,0)
    pM[i,pressure,0]=pM[i,density,0]**gamma
#Fill the particle matrix via SPH.
for t in range(1,timeSteps):#For each timestep...
    if(objectMomentum[0]!=0):#So we avoid divide-by-zero errors...
      liftDragRatio[t]=objectMomentum[1]/objectMomentum[0]#Store the L/D Ratio
    #Determine the new positions and velocities
    for j in range(0,numParticles):#For each particle...
      accel=np.array([0.0,0.0],dtype=float)
      for i in range(0,numParticles):#Get new position and velocity by iterating over all other particles...
        x1=pM[i,xPosition,t-1]
        y1=pM[i,yPosition,t-1]
        x2=pM[j,xPosition,t-1]
        y2=pM[j,yPosition,t-1]
        radiusCalcSquared=(x2-x1)*(x2-x1)+(y2-y1)*(y2-y1)
        if (i!=j and radiusCalcSquared<(2*h)*(2*h)):#If within the kernel range, and not itself...
          #Add to the net acceleration
          accel+=-((pM[j,density,t-1])**(gamma-2)+(pM[i,density,t-1])**(gamma-2))*gradkernel(pM[i,xPosition,t-1],pM[i,yPosition,t-1],pM[j,xPosition,t-1],pM[j,yPosition,t-1],radiusCalcSquared)#Assemble the acceleration
      newV=newVelocity(j,t,accel*mass)#Determine the new velocity from the old velocity and old acceleration.
      pM[j,xVelocity,t]=newV[0]
      pM[j,yVelocity,t]=newV[1]
      pM[j,xPosition,t]=pM[j,xPosition,t-1]+pM[j,xVelocity,t]*dT#Determine the new position from the old position and new velocity
      pM[j,yPosition,t]=pM[j,yPosition,t-1]+pM[j,yVelocity,t]*dT
    #Finally, update the densities and pressures based on the new positions.
    for j in range(0,numParticles):
      pM[j,density,t]=currentDensity(j,t)
      pM[j,pressure,t]=pM[j,density,t]**gamma#Equation of state
#Make the visual
maxRadiusRes=int(2*h*resolution)
maxRadiusResSquared=maxRadiusRes*maxRadiusRes
oneOverMaxRadiusResSquared=1.0/maxRadiusResSquared
intensityBlock=np.zeros((resolution,resolution,timeSteps))
ims = []
for t in range(0,timeSteps):#For each time ste...
    Intensity=np.zeros((resolution,resolution))
    for j in range(0,numParticles):#For each particle, add values to the "intensity" array such that the density appears, through the color gradient, to drop off quadratically.
      resolutionDownIndex=int(     (1-pM[j,yPosition,t])*(resolution-1)          )
      resolutionAcrossIndex=int(   (pM[j,xPosition,t])*(resolution-1)          )
      rho=pM[j,density,t]
      if(resolutionAcrossIndex-maxRadiusRes<0):#The rest of this is just pixel dancing.
```

```python
      minXRange=0;
    else:
      minXRange=resolutionAcrossIndex-maxRadiusRes
    if(resolutionAcrossIndex+maxRadiusRes>resolution-1):
      maxXRange=resolution
    else:
      maxXRange=resolutionAcrossIndex+maxRadiusRes
    if(resolutionDownIndex-maxRadiusRes<0):
      minYRange=0;
    else:
      minYRange=resolutionDownIndex-maxRadiusRes
    if(resolutionDownIndex+maxRadiusRes>resolution-1):
      maxYRange=resolution
    else:
      maxYRange=resolutionDownIndex+maxRadiusRes
    for x in range(minXRange,maxXRange):
      for y in range(minYRange,maxYRange):
        x1=x
        y1=y
        x2=resolutionAcrossIndex
        y2=resolutionDownIndex
        radiusResSquared=(x2-x1)*(x2-x1)+(y2-y1)*(y2-y1)
        if(radiusResSquared<=maxRadiusResSquared):
          Intensity[y,x]+=rho*(1-radiusResSquared*oneOverMaxRadiusResSquared)#Visually drops off quadratically. Avoided a costly square root!
  intensityBlock[:,:,t]=Intensity
maxRho=np.max(intensityBlock)
avgRho=np.mean(intensityBlock[:,:,0])

bbPath = mplPath.Path(oM[:,:2])#Define a closed path by the contour of the airfoil.
objectArray=np.zeros((0,2))
for xAcross in range(0,resolution):#Iterate over each pixel, determining if it falls within the contour of the object.
  for yDown in range(0,resolution):
    if(bbPath.contains_point((xAcross/resolution,1-yDown/resolution))):
      objectArray=np.append(objectArray,[[yDown,xAcross]],axis=0)
for t in range(0,timeSteps):
  for i in range(0,objectArray.shape[0]):
    intensityBlock[objectArray[i,0],objectArray[i,1],t]=maxRho#If so, make the value at that point the maximum density value, so it shows up dark blue.
#Draw the animation
fig=plt.figure()
ims=[[plt.imshow(intensityBlock[:,:,t], cmap='Blues', vmin=0,vmax=2*avgRho, animated=True, extent=[0,1,0,1])] for t in range(0,timeSteps)]
ani = animation.ArtistAnimation(fig, ims, interval=animationTimeInterval, blit=True, repeat_delay=1000)

Writer=animation.writers['ffmpeg']#Save the animation
writer=Writer(fps=60,metadata=dict(artist='Me'),bitrate=10000)
ani.save('Test.mp4',writer=writer)

plt.show(block=False)
```

v11.py