Ilia Kheirkhah

11/30/2022

Table of Contents

MEAM 5450 Project 1 - Lumped Vortex Panel Method

Cl vs Angle of Attack for a Flat Plate Airfoil

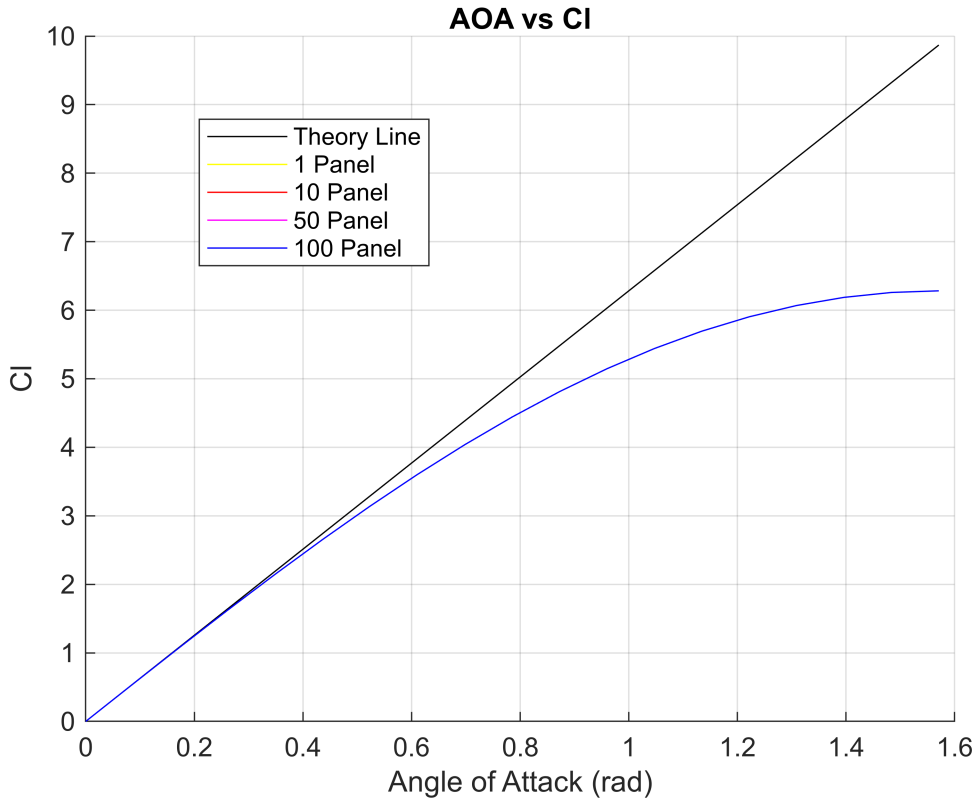
The first task we had was showing the relationship between c_l and α for a flat plate airfoil. I plan to vary the angle of attack in increments of 5 degrees up to 90 degrees. To find the lift coefficient, we split a flat plate into various Lumped Vortex Panels and found the net circulation. We then applied the Kutta-Joukowski theorem and found the lift coefficient from there. To test accuracy, we also varied the number of panels ranging from 1 to 100.

```
clear,clc;
Nrange = [1,10,50,100];
AOArange = 0:5:90;

clSet = zeros(length(Nrange),length(AOArange));

Vmag = 5; % m/s i think
x0 = 0; y0 = 0; % Leading Edge Location
c = 1; % Chord Length
m = 0; % Maximum Camber
p = .25; % Location of Maximum Camber, as fraction of chord length
res = 1e-6; % This is our convergence criteria. It just makes sure my code
% doesn't do a subset of the weird stuff it can do without me catching a
% subset of that subset.


for ml = 1:length(Nrange)
```

```
    N = Nrange(ml);% Number of Panels we want
    for n = 1:length(AOArange)
        AOA = AOArange(n); % Pick AOA from Range
        freestream = Uniform(Vmag,AOA); % Set freestream properties
        angle = 0; % We want airfoil to be flat and AOA to be controled by freesteam.
         % Generate Airfoil points using complex numbers
        airfoilSet = airfoil(x0,y0,angle,m,p,c,N+1);
        xset = real(airfoilSet); %
        yset = imag(airfoilSet);
        foil = VortexPanel.meshFoil(xset,yset);
        [~,cl] = VortexPanel.solve(foil,freestream.getVelocity(),Vmag,c,N,res);

        clSet(ml,n) = cl;
    end
end
```

Visualization

```
AOArange = deg2rad(AOArange);
clTheory = 2*pi*AOArange;
figure;
hold on;
plot(AOArange,clTheory,'k-');
plot(AOArange,clSet(1,:),'y-');
plot(AOArange,clSet(2,:),'r-');
plot(AOArange,clSet(3,:),'m-');
plot(AOArange,clSet(4,:),'b-');

legend(["Theory Line","1 Panel","10 Panel","50 Panel","100 Panel"])
legend("Position", [0.18568,0.62123,0.29592,0.23615])
title("AOA vs Cl")
xlabel("Angle of Attack (rad)")
ylabel("Cl")
grid on
```



Reflection

The above plot shows the relationsip between angle of attack and the coefficient of lift. We have also plotted a theory line with slope 2*pi to show the relationship between our calculated lift coefficient and the theory lift. There are two things to notice here. First, and most obviously, the calculated curve strays very far from the theory curve at larger angle of attacks. Second, the calculated curves are all identical regardless of the number of panels used.

Let us talk about the first issue. In our theory, we claimed that $c_l = 2\pi\alpha$ for a thin symmetric airfoil. However, now that we have run our calculations, we notice that they disagree with the theory. This error comes from the small angle approxmiation we made in deriving our theory. A more accurate form of our theory would be to say $c_l = 2\pi\sin(\alpha)$. This is where the curve we see in our calculations originates from.

The other problem is much more difficult to understand. In our theory, shown that our circulation distribution should vary along the length of our symmetric airfoil. At first, I believed that the graphs overlapping implied a problem in the code, specifically that the code resulted in a constant circulation distribution throughout. However, by printing the strength values calculated by the VortexPanel class, I could see this was not the case. I checked other sections of my code for bugs, but did not find any major errors that could result in this., but it should be noted that I am not sure what type of bug would result in this type of behavior.

Cl vs Angle of Attack for a Cambered Plate Airfoil

The second task we had was showing the relationship between c_l and α for a cambered plate airfoil. The same general approach as the previous section was applied, with the only change being how the geometry was

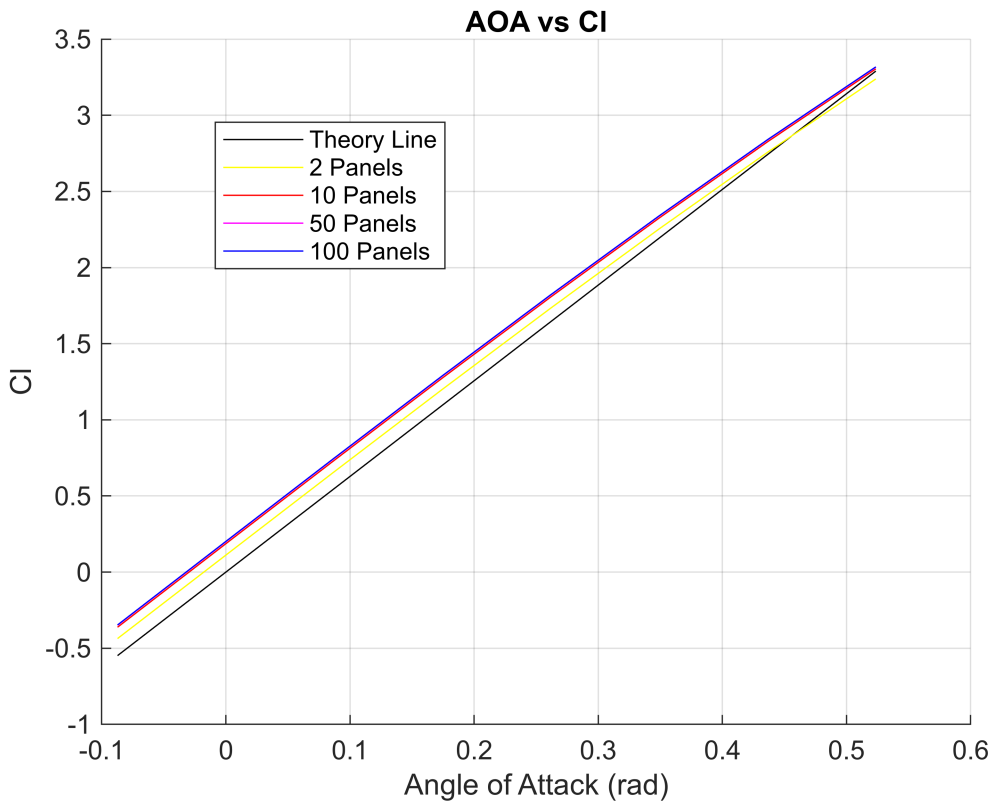
generated. Instead of a simple flat plate, we needed a good way to represent camber. In my implimentation, I chose to use the NACA 4 Series equations for the mean camber line. This allowed me the freedom to not only pick the extent of my camber but also the location. For this entire homework, I have placed the location of maximum camber at the quarter chord point.

The only major change done was reducing the maximum angle to only 15 degrees. In this section, we will want to worry less about the general behavior of the graph and pay more attention to the intercept of the graph. This is because one of the most important features of cambered airfoils is their ability to generate lift at zero angle of attack and sometimes even negative angle of attack. As such, we have also extended our angle of attack range slightly into the negative domain to be able to see this behavior. Together, this range will allow us to show the accuracy of the slope and have a good view of the intercepts.

```
clear,clc;
Nrange = [2,10,50,100];
AOArange = -5:5:30;

clSet = zeros(length(Nrange),length(AOArange));

Vmag = 5; % m/s i think
x0 = 0; y0 = 0; % Leading Edge Location
c = 1; % Chord Length
m = 0.02; % Maximum Camber
p = .25; % Location of Maximum Camber, as fraction of chord length
res = 1e-6; % This is our convergence criteria. It just makes sure my code
% doesn't do a subset of the weird stuff it can do without me catching a
% subset of that subset.

for ml = 1:length(Nrange)
    N = Nrange(ml);% Number of Panels we want
    for n = 1:length(AOArange)
        AOA = AOArange(n); % Pick AOA from Range
        freestream = Uniform(Vmag,AOA); % Set freestream properties

        % We want airfoil to be flat and AOA to be controled by freesteam.
        % This would control the airfoil angle in space
        angle = 0;
        % Generate Airfoil points using complex numbers
        airfoilSet = airfoil(x0,y0,angle,m,p,c,N+1);
        xset = real(airfoilSet); %
        yset = imag(airfoilSet);
        foil = VortexPanel.meshFoil(xset,yset);
        [~,cl] = VortexPanel.solve(foil,freestream.getVelocity(),Vmag,c,N,res);

        clSet(ml,n) = cl;
    end
end
```

Visualization

```
AOArange = deg2rad(AOArange);
clTheory = 2*pi*AOArange;
figure;
hold on;
plot(AOArange,clTheory,'k-');
plot(AOArange,clSet(1,:),'y-');
plot(AOArange,clSet(2,:),'r-');
plot(AOArange,clSet(3,:),'m-');
plot(AOArange,clSet(4,:),'b-');

legend(["Theory Line","2 Panels","10 Panels","50 Panels","100 Panels"])
legend("Position", [0.18568,0.62123,0.29592,0.23615])
title("AOA vs Cl")
xlabel("Angle of Attack (rad)")
ylabel("Cl")
grid on
```



Reflection

For a 2% cambered airfoil, very few panels are necessary to get a fairly accurate result. This can be seen in the close spacing of the lines and how quickly they converge onto on area. On the other hand, when I ran the simulation with very high camber (15%), the lines were much more spread apart and the actually accurate range did not start unti around 20 panels. This discretization error would likely be more pronounced with a full 2D airfoil as opposed to the line-foils we are using.

Calculated vs Theoretical Circulation

In this problem, we were tasked with comparing the analytical and calculated total circulation of a symmetric airfoil at a 5 degree angle of attack. We know from our theory that the expected total circulation is: $\Gamma = \pi \alpha c V_\infty$. It is important to remember that the angle of attack in this calculation needs to be in radians, not degrees. For this section, I chose to only run the simulation with 50 panels. In the first section we talked about the interesting behavior that the total circulation seems to be the same for any number of panels. To be safe, I wanted to go on the higher end of panels still, but went with only 50 to save some runtime as this program was starting to get bulky.

```
clear,clc;
N = 50;
AOA = 5;

Vmag = 5; % m/s i think
x0 = 0; y0 = 0; % Leading Edge Location
c = 1; % Chord Length
m = 0; % Maximum Camber
p = .25; % Location of Maximum Camber, as fraction of chord length
res = 1e-6; % This is our convergence criteria. It just makes sure my code
% doesn't do a subset of the weird stuff it can do without me catching a
% subset of that subset.

freestream = Uniform(Vmag,AOA); % Set freestream properties
angle = 0; % We want airfoil to be flat and AOA to
% be controled by freesteam.

% Generate Airfoil points using complex numbers
airfoilSet = airfoil(x0,y0,angle,m,p,c,N+1);
xset = real(airfoilSet); %
yset = imag(airfoilSet);
foil = VortexPanel.meshFoil(xset,yset);
[strengths,~] = VortexPanel.solve(foil,freestream.getVelocity(),Vmag,c,N,res);


totalCirc = sum(strengths);
expectedCirc = pi*deg2rad(AOA)*c*Vmag;

% Calculated Circulation
disp(totalCirc)
```

```
    1.3690
```

```
% Total Circulation
disp(expectedCirc)
```

```
    1.3708
```

Reflection

This is an interesting error. I varied the number of panels and always got the exact same total circulation even though the length of the strength vector was changing appropriately. This calculated circulation also varies by about 2e-3 from the expected circulation that the theory says we should find. I am not sure where this error is coming from, but I believe discovering the source of this error will lead me to the reason behind all the symmetric airfoil graphs being identical.

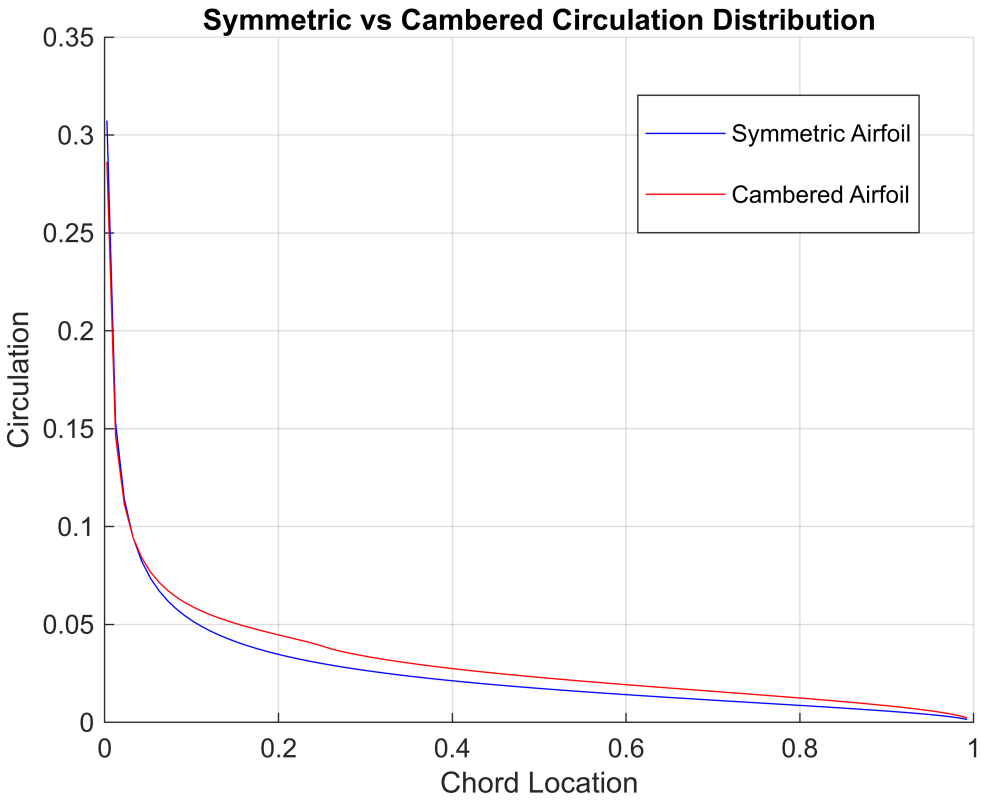
Circulation Distribution for Symmetric and Cambered Airfoil

Here we are tasked with comparing the circulation distribution between a symmetric and cambered airfoil using panel methods. This can be seen graphically very well. For this problem, I used the same angle of attack and number of panels for both the cambered and symmetric airfoils. I used a regular 2% cambered airfoil with the location of maximum camber at the quarter chord location.

```
clear,clc;
N = 100;
AOA = 10;

Vmag = 5; % m/s i think
x0 = 0; y0 = 0; % Leading Edge Location
c = 1; % Chord Length
m = [0,.02]; % Maximum Camber
p = .25; % Location of Maximum Camber, as fraction of chord length
res = 1e-6; % This is our convergence criteria. It just makes sure my code
% doesn't do a subset of the weird stuff it can do without me catching a
% subset of that subset.

freestream = Uniform(Vmag,AOA); % Set freestream properties
angle = 0; % We want airfoil to be flat and AOA to
% be controled by freesteam.

% Generate Airfoil points using complex numbers
% Symmetric Airfoil
airfoilSet = airfoil(x0,y0,angle,m(1),p,c,N+1);
xset = real(airfoilSet); %
yset = imag(airfoilSet);
foil = VortexPanel.meshFoil(xset,yset);
[symStrengths,~] = VortexPanel.solve(foil,freestream.getVelocity(),Vmag,c,N,res);

% Cambered Airfoil
airfoilSet = airfoil(x0,y0,angle,m(2),p,c,N+1);
xset = real(airfoilSet); %
yset = imag(airfoilSet);
foil = VortexPanel.meshFoil(xset,yset);
[camStrengths,~] = VortexPanel.solve(foil,freestream.getVelocity(),Vmag,c,N,res);

plotSet = xset(1:N)+.25*c/N;
```

Visualize

```
figure;
hold on;
plot(plotSet,symStrengths,'b-');
plot(plotSet,camStrengths,'r-');

legend(["Symmetric Airfoil","Cambered Airfoil"])
legend("Position", [0.57565,0.65484,0.31072,0.2389])
title("Symmetric vs Cambered Circulation Distribution")
xlabel("Chord Location")
ylabel("Circulation")
grid on
```



Reflection

The first thing we should note here is that the shape of the symmetric airfoil circulation distribution is correct. In the theory, we are told the expected equation for the circulation distribution is: $\gamma(\theta) = 2\alpha V_\infty \frac{1+\cos(\theta)}{\sin(\theta)}$. When translated into chord position instead of angle, this gives becomes a function roughly of the form $\frac{C}{\sqrt{x}}$ with some slight distinctions which is what we roughly see in the graph above. Although this is great, it is further complicating why our answer for problem 1 does not vary with the number of panels.

The cambered circulation distrbution is much more interesting. The irregular shape of the curve is interesting, but difficult to analyze. It is much more interesting to see that, at almost all locations, the cambered curve is above the symmetric curve. The exception to this is at the very start of the airfoil. Early in the airfoil, the

cambered version will have a lower circulation than the symmetric airfoil. However, since the cambered airfoil has significantly higher circulation for roughly 90% of the chord length, the lift of the cambered airfoil is higher. This is reasonable since we know both the symmetric and cambered airfoils will have the same slope, but the cambered airfoil will be shifted up same amount to allow for lift at zero angle of attack. This means that, at all points, we expect a higher lift from cambered airfoils than symmetric airfoils. The graph shown confirms this intuition.

Another important fact is that both graphs go to 0 as we approach the end of the chord at a chord location of 1. This means that the Kutta Condition is satisfied for our airfoils. As we talked about in class, each individual lumped vortex panel satisfies the kutta condition so we expect the complete airfoil to satisfy the Kutta Condition. This graph confirms this statement for both cambered and symmetric airfoils and further enforces the validity of my code.

High Lift Devices

Here, we are testing a two airfoil system using our vortex panel method. In the 3 scenarios, we vary the angle of attack of our airfoil system. We are told to position the airfoils at 5 degrees and 8 degrees AOA with respect to the freestream velocity. If we did this in all 3 situations we would expect the same results. Instead, I chose to hold the airfoils the same, 3 degrees off set from one another, and then vary the freestream as described. This would give us 3 different angles of attack.

In addition, we are told to vary the number of panels from 1 to N. I chose to make this number representative of the number of panels in each airfoil, not the total number of panels, which would now be 2N.

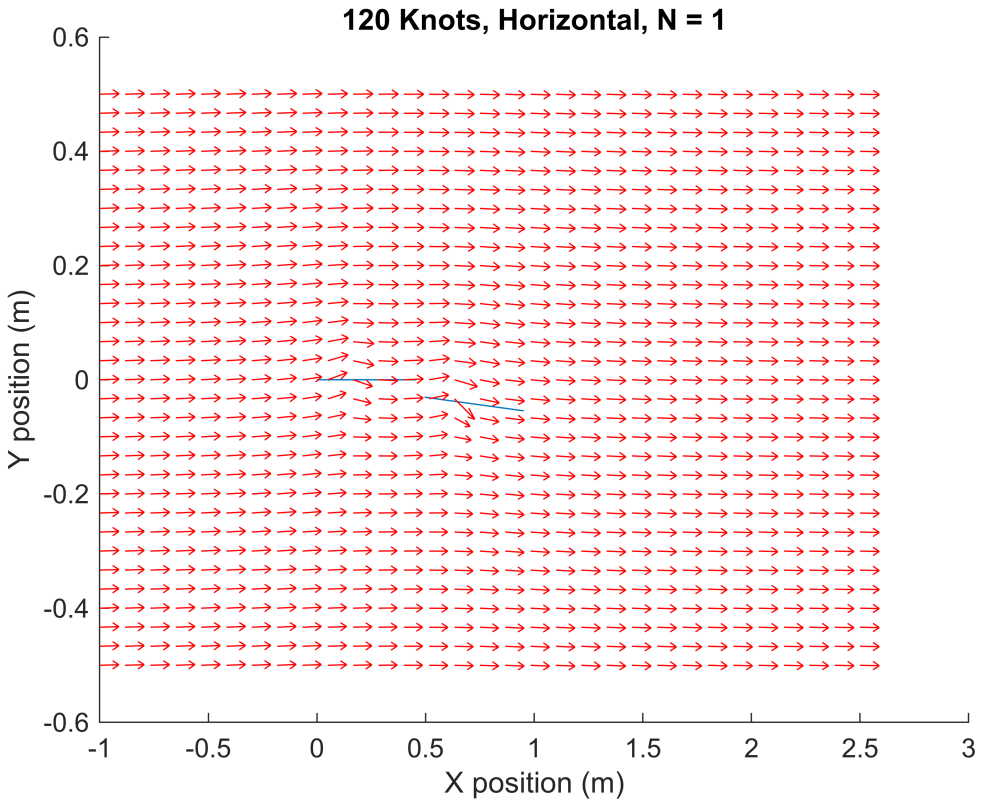
To calculate the total and individual lift, I applied the Kutta joukowski theorem. For the individual lift situations, I only summed half the strengths to get the appropriate lift.

```
clear; clc;

rho = 1.2; % Air density kg/m3
res = 1e-10; % Convergence Criteria for colocation points
% Upstream Airfoil Properties
x01 = 0; y01 = 0; % Leading Edge Location
c1 = 0.4572; % Chord Length, m
m1 = 0; % Maximum Camber
p1 = .25; % Location of Maximum Camber, as fraction of chord length
angle1 = 0;

% Downstream Airfoil Properties
x02 = x01 + .25*c1 + .381; y02 = y01-.0305; % Leading Edge Location
c2 = 0.4572; % Chord Length, m
m2 = 0; % Maximum Camber
p2 = .25; % Location of Maximum Camber, as fraction of chord length
angle2 = -3;

% Varying Parameters
hVel = 61.7333; %m/s
vVel = [0,-5.08,5.08]; %m/s
% One should notice that the vertical velocity is negative relative to what
```

```
% is expected for climb vs descent.

% These are the actual parameters our Uniform class take in
vMagRange = sqrt(hVel^2+vVel.^2); % Coupled to AOArange, not independent
AOArange = atan(vVel./hVel); % Coupled to vMagRange, not two independent variables
AOArange = rad2deg(AOArange);
Nrange = [1,10,50,100];

% Array to store data for post-processing
data = zeros(length(AOArange),length(Nrange),2);
% The first index will pick the external flow situation
% The second index will pick the panel resolution
% The third index holds both individual lift values. From this we can
% calculate the total lift. If the third index is 1, we are looking at the
% upstream circulation. If the index is 2, we are looking at the downstream
% circulation.

% These will be our parameters for the velocity field
xmin = -1;
ymin = -.5;
xmax = 2.5;
ymax = .5;
Ngrid = 31;
for i = 1:length(AOArange)
    % Take the current flow properties
    vMag = vMagRange(i);
    AOA = AOArange(i);

    freestream = Uniform(vMag,AOA);
    [gridMesh,vField] = freestream.velField(xmin,ymin,xmax,ymax,Ngrid,Ngrid);

    for j = 1:length(Nrange)
        % Make the airfoil geometry
        N = Nrange(j);
        air1 = airfoil(x01,y01,angle1,m1,p1,c1,N+1);
        xset1 = real(air1);
        yset1 = imag(air1);
        air2 = airfoil(x02,y02,angle2,m2,p2,c2,N+1);
        xset2 = real(air2);
        yset2 = imag(air2);

        % Mesh the airfoils and combine into 1 system
        foil1 = VortexPanel.meshFoil(xset1,yset1);
        foil2 = VortexPanel.meshFoil(xset2,yset2);

        system = [foil1,foil2];

        % Calculate strengths
        % Approach does not need to change since we have a simple set of
        % panels that influence each other fully.
```

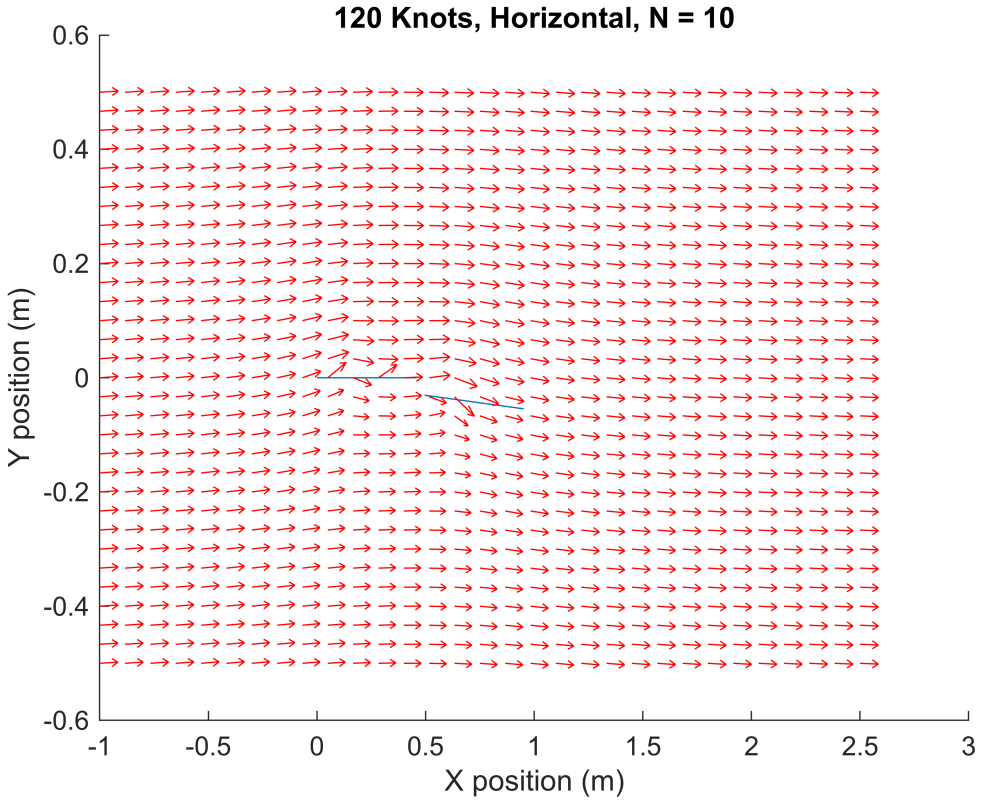
```
        [strengths,~] = VortexPanel.solve(system,freestream.getVelocity(),vMag,c1,N,res);

        % Go through and set each strength in system.
        for k = 1:length(system)
            system(k) = system(k).setStrength(strengths(k));
        end
        % Remember that this does not set the strengths in the foil
        % varaible, it just sets the strengths in the system array. To fix
        % this, we will go through and extract foil1 and foil2 from the
        % system array
        foil1 = system(1:N);
        foil2 = system(N+1:2*N);

        % Now, we want to save the individual total circulations in our
        % data array. This will then be used for post-processing the lift
        % data.
        data(i,j,1) = 0; % Upstream Circulation
        data(i,j,2) = 0; % Downstream Circulation

        for k = 1:length(foil1)
            data(i,j,1) = data(i,j,1) + foil1(k).getStrength();
            data(i,j,2) = data(i,j,2) + foil2(k).getStrength();

            % While we are looping through the inidivudal foils, we will
            % also just fill out the velocity field array

            [~,contr1] = foil1(k).velField(xmin,ymin,xmax,ymax,Ngrid,Ngrid);
            [~,contr2] = foil2(k).velField(xmin,ymin,xmax,ymax,Ngrid,Ngrid);

            vField = vField + contr1 + contr2;
        end

        % Now we create the flowfields for all scenarios.

        resolution = ['N = ', num2str(N)];
        switch i
            case 1
                climb = 'Horizontal';
            case 2
                climb = '1000 ft/min Climb';
            case 3
                climb = '1000 ft/min Descent';
        end

        tit = ['120 Knots, ',climb,', ',resolution];

        figure;
        hold on;
        title(tit);
        for k = 1:length(system)
```

```
            system(k).simpleDraw();
        end

        quiver(real(gridMesh),imag(gridMesh),real(vField),imag(vField),'color',[1,0,0]);

        xlabel("X position (m)");
        ylabel("Y position (m)");

        upstreamCirc = data(i,j,1);
        aftCirc = data(i,j,2);

        upstreamLift = rho*vMag*upstreamCirc;
        downstreamLift = rho*vMag*aftCirc;
        totalLift = upstreamLift+downstreamLift;

        text1 = ['Upstream Airfoil Lift: ',num2str(upstreamLift),'N'];
        text2 = ['Downstream Airfoil Lift: ',num2str(downstreamLift),'N'];
        text3 = ['Total System Lift: ',num2str(totalLift),'N'];

        disp(tit);
        disp(text1)
        disp(text2)
        disp(text3)
    end
end
```

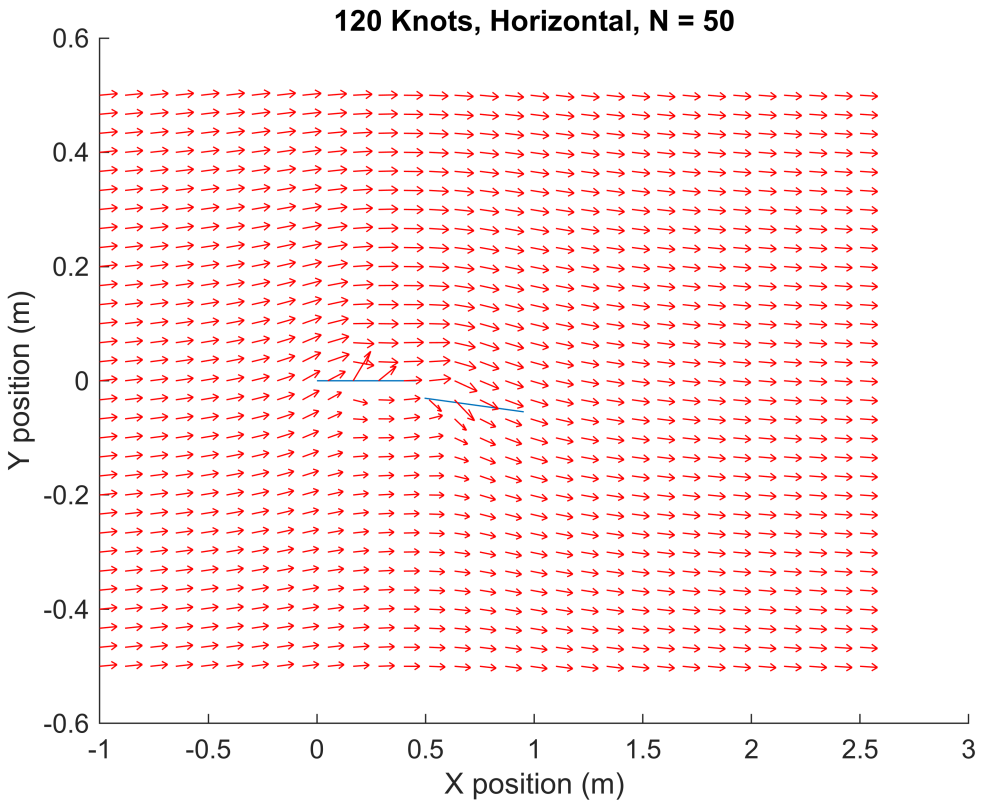



```
120 Knots, Horizontal, N = 1
```

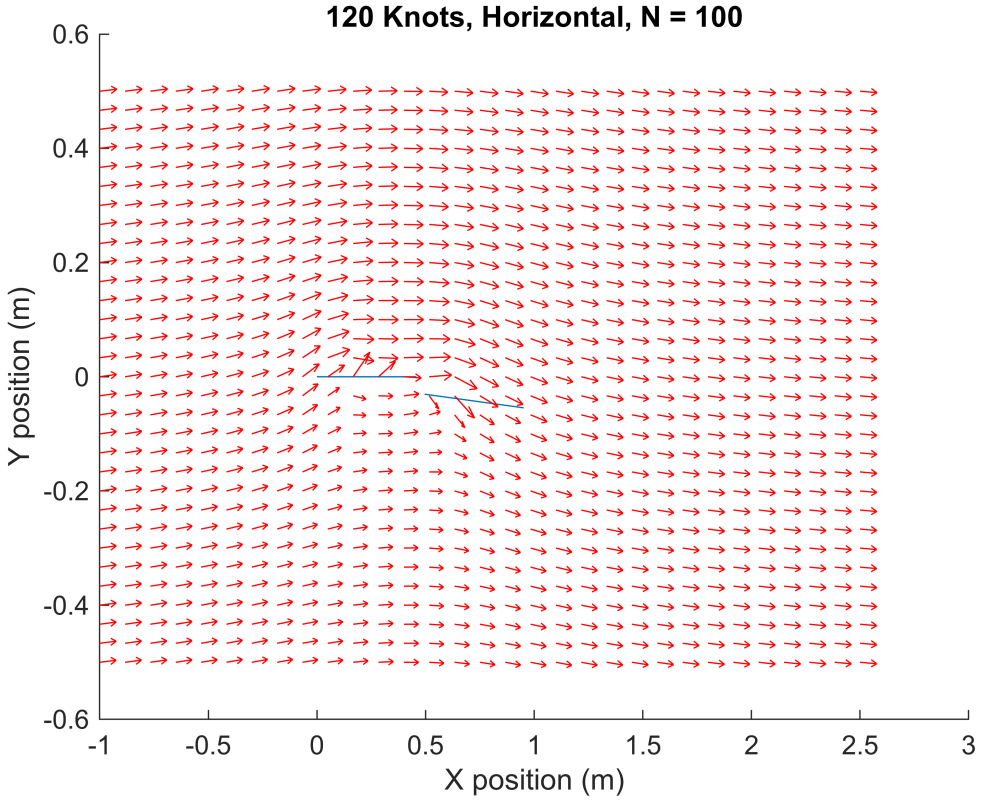
```
Upstream Airfoil Lift: 228.7149N
Downstream Airfoil Lift: 271.6732N
Total System Lift: 500.3881N
```




```
120 Knots, Horizontal, N = 10
Upstream Airfoil Lift: 280.3668N
Downstream Airfoil Lift: 240.3869N
Total System Lift: 520.7536N
```

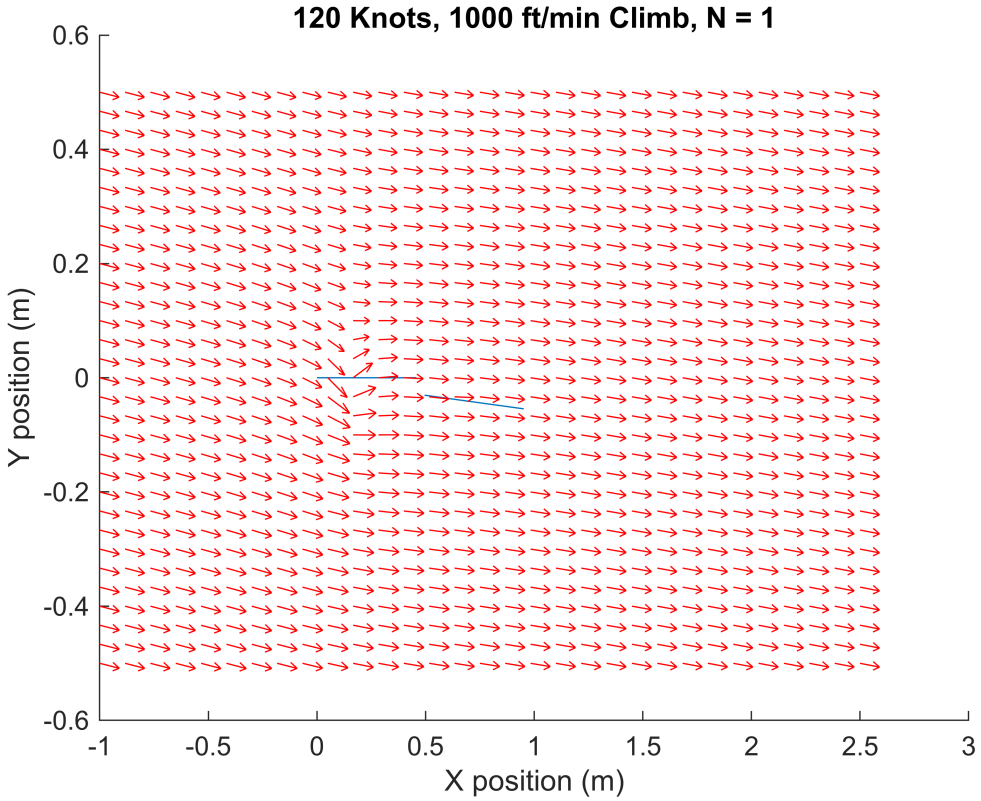



```
120 Knots, Horizontal, N = 50
Upstream Airfoil Lift: 280.843N
Downstream Airfoil Lift: 239.8538N
Total System Lift: 520.6968N
```

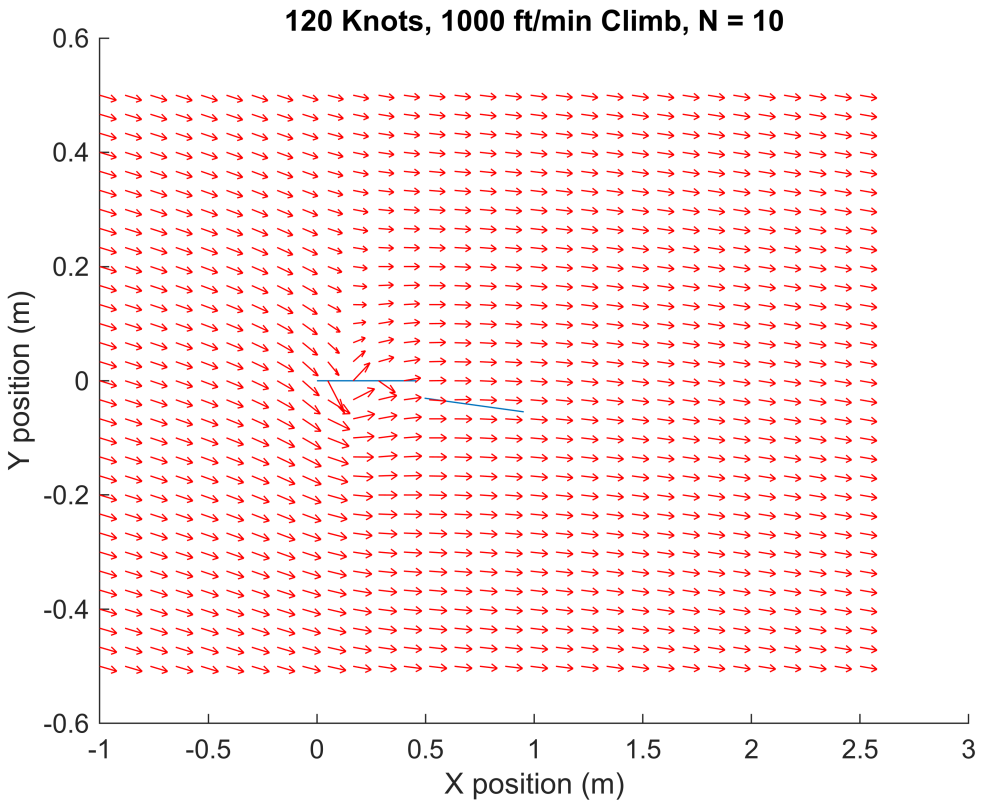



```
120 Knots, Horizontal, N = 100
```

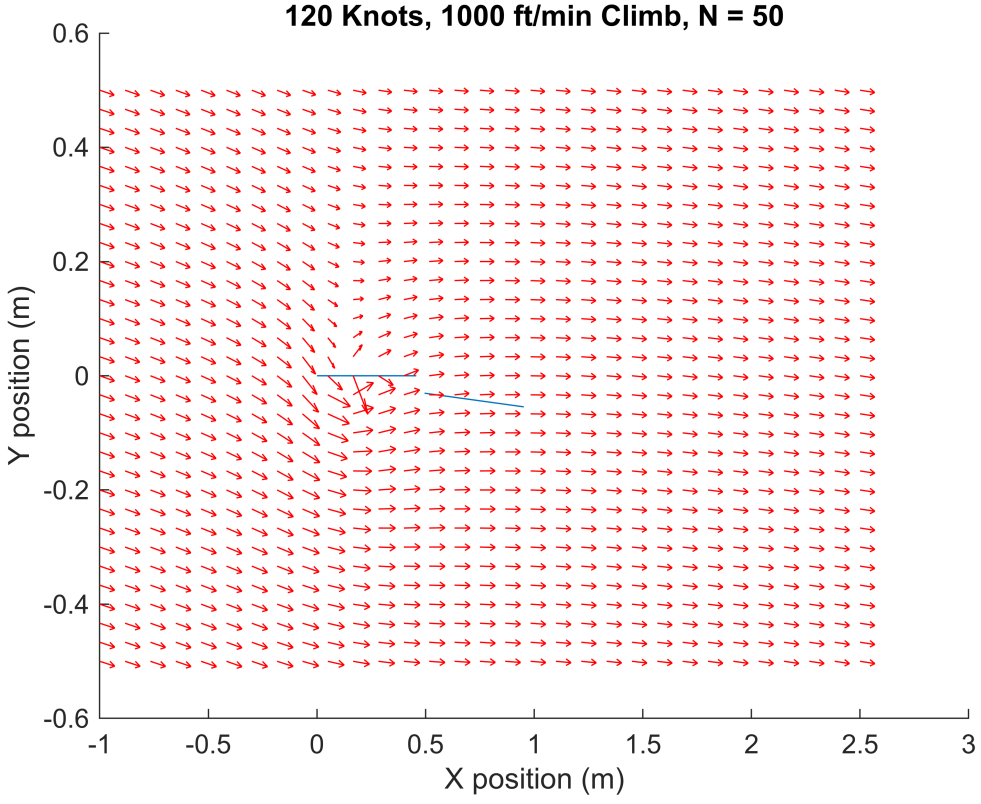
```
Upstream Airfoil Lift: 280.8502N
Downstream Airfoil Lift: 239.8407N
Total System Lift: 520.6909N
```




```
120 Knots, 1000 ft/min Climb, N = 1
Upstream Airfoil Lift: -559.4527N
Downstream Airfoil Lift: -20.3055N
Total System Lift: -579.7582N
```

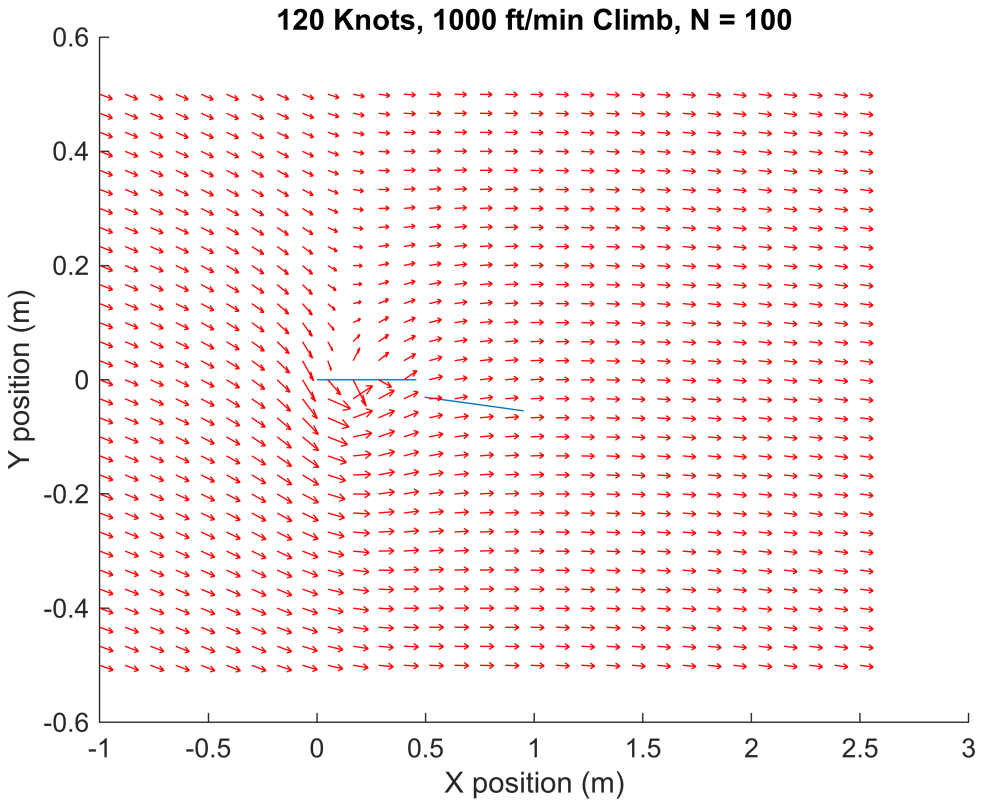



```
120 Knots, 1000 ft/min Climb, N = 10
Upstream Airfoil Lift: -539.4386N
Downstream Airfoil Lift: -14.6797N
Total System Lift: -554.1182N
```

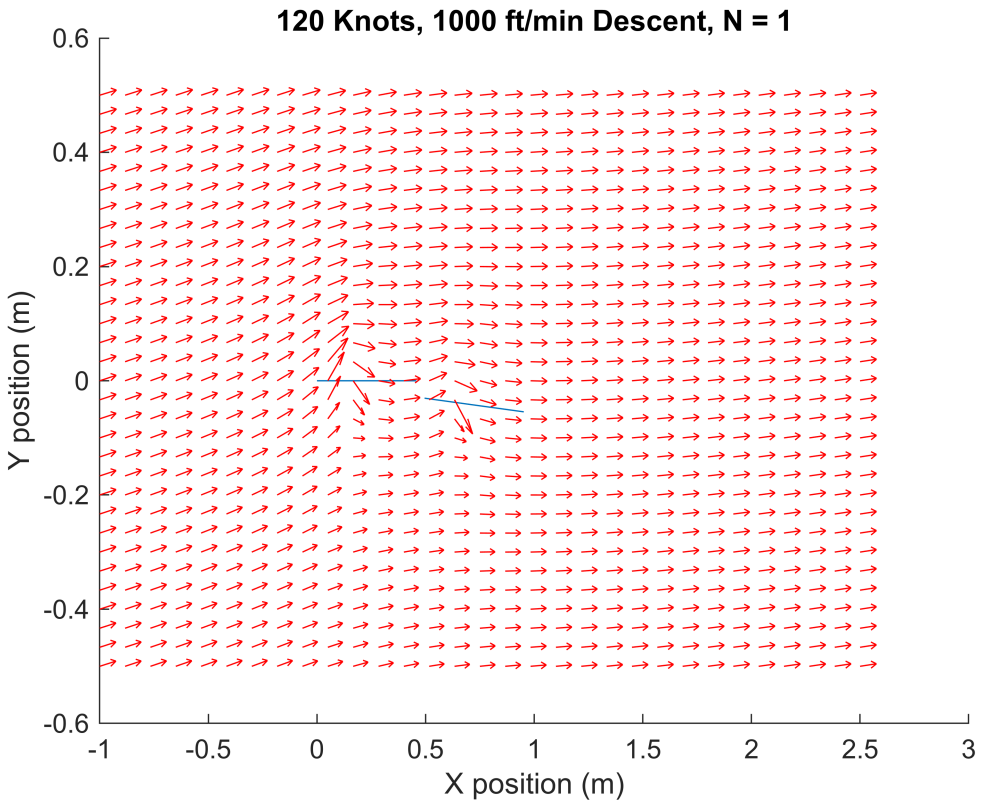



```
120 Knots, 1000 ft/min Climb, N = 50
```

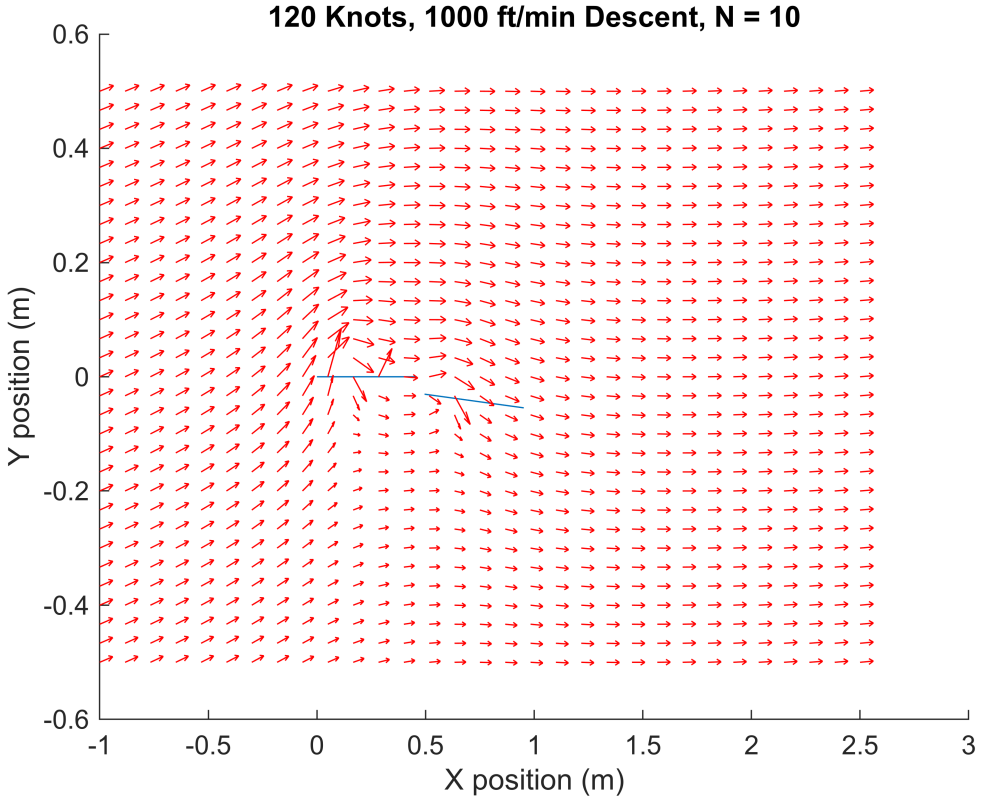
```
Upstream Airfoil Lift: -538.8448N
Downstream Airfoil Lift: -14.7282N
Total System Lift: -553.573N
```




```
120 Knots, 1000 ft/min Climb, N = 100
Upstream Airfoil Lift: -538.8273N
Downstream Airfoil Lift: -14.7291N
Total System Lift: -553.5564N
```

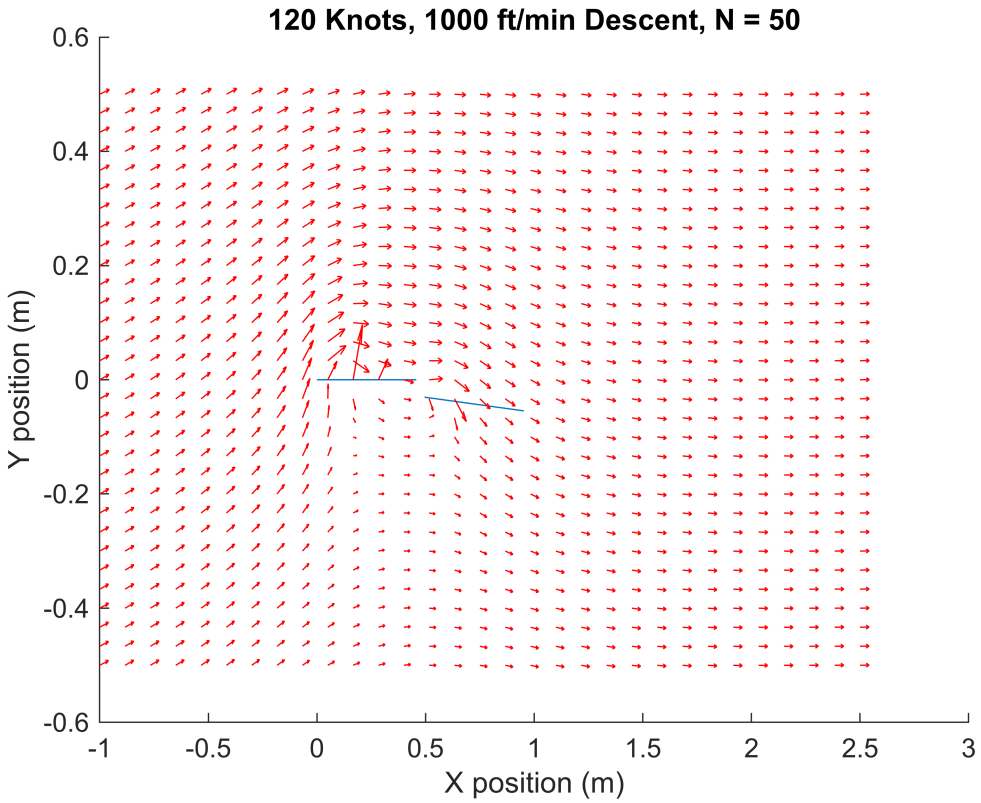



```
120 Knots, 1000 ft/min Descent, N = 1
Upstream Airfoil Lift: 1018.4286N
Downstream Airfoil Lift: 565.4884N
Total System Lift: 1583.9171N
```

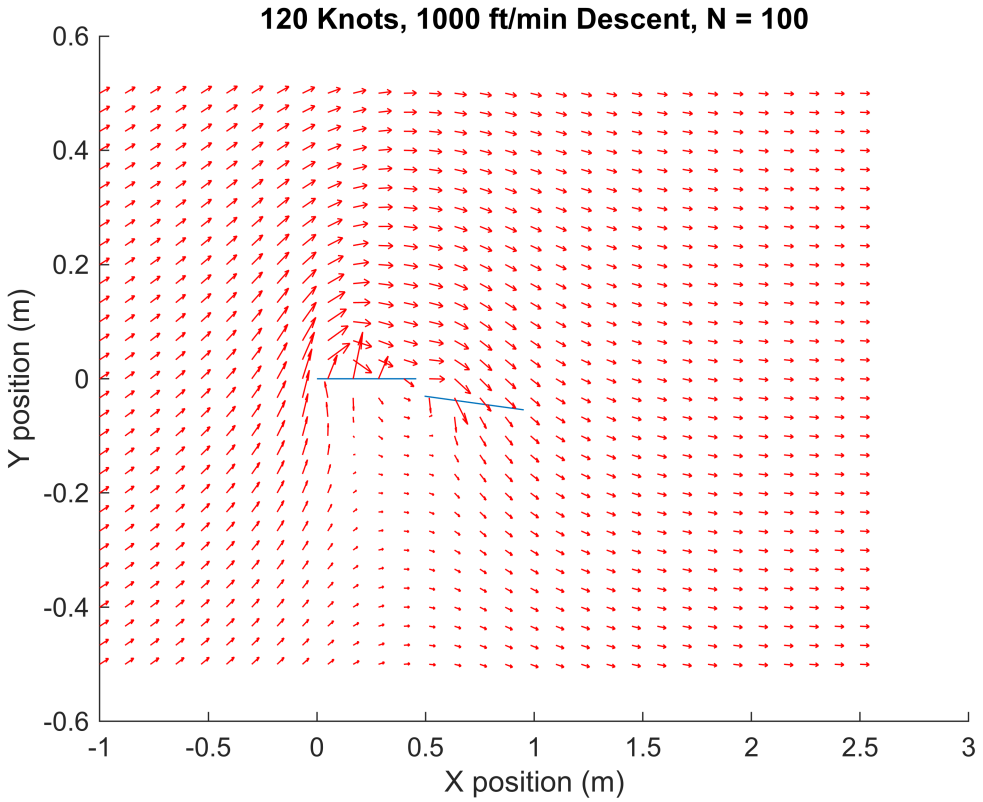



```
120 Knots, 1000 ft/min Descent, N = 10
```

```
Upstream Airfoil Lift: 1102.0675N
Downstream Airfoil Lift: 497.0784N
Total System Lift: 1599.1459N
```




```
120 Knots, 1000 ft/min Descent, N = 50
Upstream Airfoil Lift: 1102.4292N
Downstream Airfoil Lift: 496.0573N
Total System Lift: 1598.4865N
```




```
120 Knots, 1000 ft/min Descent, N = 100
Upstream Airfoil Lift: 1102.4262N
Downstream Airfoil Lift: 496.0319N
Total System Lift: 1598.4581N
```

Reflection

This section is giving us some of the more interesting resutls. The first thing I noticed was the velocity fields. Specifically, there seemed to be flow going through the airfoil. Upon reflection, this is fine and to be expected as we set a no flow through condition at the colocation points, not everywhere along the airfoil. As we increase the number of panels, it will be less likely for flow to go through. The only worry that I had was that the no flow through condition was not satisfied at the colocation points. However, I knew this was not an issue since we built in a check for this given the residual of 1e-10. The reason I did not set this to zero is that floating point errors could set off our warning system which we do not want. 1e-10 is a residual that accounts for this error.

It is also interesting to see that, when flying horizontally, the upstream airfoil was generating lift. This is a surprise since in the horizontal cases, the upstream airfoil has zero angle of attack relative to the freestream. This enforces our belief that, in subsonic flow, information travels everywhere. As a result, the circulation from the aft airfoil forces the upstream airfoil to produce some circulation. Mathematically, this is because our airfoil needs to keep the flow at the colocation points tangent.

Conclusion

In this end, this homework showed the power of panel methods in finding accurate solutions to complex flow issues. Although our cases were not geometrically complex, panel methods can be expanded to deal with any geometry and will likely produce decent results. It should be noted that, when looking at complex, the

accuracy for panel methods will depend heavily on the number of panels used. However, this is the case for any numerical method and should not be seen as a negative of panel methods but just as another fact to remember.

An interesting Idea for future experimentation would be to test panel methods on 2D geometry. We know panel methods work well for flat airfoils, but in real life we will often want to consider 2D airfoils. Once we have established panel methods in 2D, the obvious next step would be to expand our concept of panels to 3D constructs that could be used to analyze entire wings and potentially more complex strutures.

Appendix

Below is a in-text version of all the supporting files. There were 4 main files that made this main possible; airfoil.m, Uniform.m, Vortex.m, VortexPanel.m.

airfoil.m is a function file that took in airfoil shape properties such as chord length, maximum camber, and maximum camber location and returns an array with the mean camber line. This allowed for the generation of our airfoil shape in a quick function. This took advantage of the NACA 4 Series Mean Camber Line equations. This allowed for sufficient robustness to deal with both symmetric and cambered airfoils in one function.

Uniform.m is a simple class that allowed for the creation of a uniform velocity flow pattern.

Vortex.m is another simple class that allowed for the creation of a vortex flow pattern. It should be noted that this class was not used directly in the main class very often. Instead, it was implemented to make the VortexPanel.m class simpler.

VortexPanel.m the primary functional class of this assignment, this class implemented all the functionality that is associated with vortex panel methods. From generating the airfoil in panel form to solving for the strengths of each panels, almost all methods were implemented in this file.

Below you will see the code implementation for all of these files.

```
% Using the NACA 4 Series equations, we can create a line airfoil using the
% mean camber line equation. This equation is robust enough to deal with
% any camber we would like, including none. Although not as useful for this
% homework, this equation also allows us to move the location of max camber
% if we would like.

function P = airfoil(x0,y0,angle,m,p,c,N)
    xset = linspace(0,c,N); % Make set of x coordinates ranging from 0 to
    % end of chord
    yset = zeros(size(xset));
    % Allocate space for airfoil
    P = complex(zeros(size(N)));

    % Create airfoil at origin with proper geometry but zero angle
    for i=1:length(xset)
        xcurr = xset(i);
        if(xcurr<p*c)
            yset(i) = (m*c/p^2)*(2*xcurr*p/c-xcurr^2/c^2);
        else
            yset(i) = (m*c/(1-p)^2)*((1-2*p)+2*p*xcurr/c-xcurr^2/c^2);
        end

        P(i) = complex(xcurr + yset(i)*1i);
    end

    % These few lines now rotate the airfoil about the origin to the proper
    % angle. Remeber that this is with the CCW positive convention, so an
    % increasing angle of attack would be a negative angle here.
    angle = deg2rad(angle);
    rotVec = exp(angle*1i);

    P = P*rotVec;
    % Finally, we translate the airfoil to the proper location
    P = complex(P + x0);
    P = complex(P + y0*1i);
end
```

Published with MATLAB® R2022b

```
% This class is used to generate a freestream velocity that we can use in
% our main file.
classdef Uniform
   properties
      dir;
      vMag;
      v;
   end
   methods
       % Constructor
       function obj = Uniform(vMag,dir)
           obj.vMag = vMag;
           obj.dir = dir;
           obj.v = vMag*cosd(dir)+vMag*sind(dir)*1i;
       end

       % Working Methods
       function ret = velVec(obj,P)
           ret = zeros(size(P))+obj.v;
       end

       function [grid,v] = velField(obj,xmin,ymin,xmax,ymax,Nx,Ny)
           xSet = linspace(xmin,xmax,Nx);
           ySet = linspace(ymin,ymax,Ny);
           [xSet,ySet] = meshgrid(xSet,ySet);
           grid = xSet+ySet*1i;

           v = obj.velVec(grid);
       end

       % Setters

       % This setStrength method will be very important for the homework.
       % To create our ICM, we will create all our vortex panels with a
       % strength fo 1. We can then use a higher level function to get the
       % influence from each vortex panel at each colocation point. Once we
       % have solved for the ICM and found the true strengths, we then need
       % a way to set the strengths to their true values and this will
       % allow us to do that.
       function obj = setStrength(obj,vMag)
           obj.vMag=vMag;
           obj.v = vMag*cosd(obj.dir)+vMag*sind(obj.dir)*1i;
       end

       % Getters
       function ret = getDir(obj)
           ret = obj.dir;
       end

       function ret = getVelocity(obj)
           ret = obj.v;
       end

       function ret = getSpeed(obj)
           ret = obj.vMag;
       end
```

```
    end
end
```

Published with MATLAB® R2022b

```
% This class allows for the creation of a vortex at any given location. It
% will not be used directly in the main, but will be used to enable our
% VortexPanel class.

classdef Vortex
   properties
      x;
      y;
      comCen;
      S;
   end
   methods
       % Constructor
       function obj = Vortex(x,y,S)
           obj.x = x;
           obj.y = y;
           obj.S = S;
           obj.comCen = complex(x+y*1i);
       end

       % Working Methods
       function v = velVec(obj,P)
           dP = P-obj.comCen;
           v = -1i*conj(obj.S./(2*pi*dP));
       end

       function [grid,v] = velField(obj,xmin,ymin,xmax,ymax,Nx,Ny)
           xSet = linspace(xmin,xmax,Nx);
           ySet = linspace(ymin,ymax,Ny);
           [xSet,ySet] = meshgrid(xSet,ySet);
           grid = xSet+ySet*1i;

           v = obj.velVec(grid);
       end

       % Setters

       % This setStrength method will be very important for the homework.
       % To create our ICM, we will create all our vortex panels with a
       % strength fo 1. We can then use a higher level function to get the
       % influence from each vortex panel at each colocation point. Once we
       % have solved for the ICM and found the true strengths, we then need
       % a way to set the strengths to their true values and this will
       % allow us to do that.
       function obj = setStrength(obj,nS)
           obj.S = nS;
       end

       % Getters
       function ret = getX(obj)
           ret = obj.x;
       end
```

```
        function ret = getY(obj)
            ret = obj.y;
        end

        function ret = getStrength(obj)
            ret = obj.S;
        end

        function ret = getCenter(obj)
            ret = obj.comCen;
        end

    end
end
```

Published with MATLAB® R2022b

```
% This is is the primary class our main file will be interfacing with. It
% all the primary functions we expect out of a panel will be implemented in
% this file and will simply be called in our main file.

classdef VortexPanel
   properties
      p1;
      p2;
      Vortex;
      colocation;
   end
   methods(Static)
       function foil = meshFoil(xset, yset)
           N = length(xset)-1;
           for j = 1:N
               p1 = complex(xset(j)+yset(j)*1i);
               p2 = complex(xset(j+1)+yset(j+1)*1i);

               foil(j) = VortexPanel(p1,p2,1);
              % I know this is memory inefficient since I am not allocating space
              % I have to accept this because I don't know a better way.
              % Maybe I fix Later
           end
       end

       function [strengths,cl] = solve(foil,vinfty,Vmag,c,N,res)
           ICM = zeros(N);
           RHS = zeros(1,N);
           for i=1:length(foil) % Current Panel Vortex
               currVortexPanel = foil(i);

               for j = 1:length(foil) % Panel we will calculate influence on.
                   targetPanel = foil(j);

                   inducedVel = currVortexPanel.getInfluence(targetPanel.getColocationPoint());
                   normalVec = targetPanel.getDirVec()*1i;
                   ICM(i,j) = real(inducedVel)*real(normalVec)+imag(inducedVel)*imag(normalVec);

                   if(i==1)
                       RHS(j) = -real(vinfty)*real(normalVec)-imag(vinfty)*imag(normalVec);
                   end
               end
           end

           strengths = RHS/ICM;

           for i = 1:length(foil)
               foil(i) = foil(i).setStrength(strengths(i));
           end

        %Here Let us check the colocation points
           for i = 1:length(foil)
               colocPoint = complex(foil(i).getColocationPoint());
               vel = 0;
               for j = 1:length(foil)
                   vel = vel + foil(j).velVec(colocPoint);
```

```
            end
            vel = vel + vinfty;
            normalVec = foil(i).getDirVec()*1i;

            normalVel = real(vel)*real(normalVec)+imag(vel)*imag(normalVec);
            if(not(normalVel < res))
                disp('Residues Not Met')
            end
        end

        totalCirc = 0;
        for i = 1:length(strengths)
            totalCirc = totalCirc + strengths(i);
        end

        rho = 1.2;

        L = rho*Vmag*totalCirc;

        q = .5*rho*Vmag^2;

        cl = L/(q*c);
    end
end

methods
    % Constructor
    function obj = VortexPanel(p1,p2,S)
        vorPoint = complex(p1+.25*(p2-p1));
        obj.colocation = complex(p1+.75*(p2-p1));
        obj.Vortex = Vortex(real(vorPoint),imag(vorPoint),S);
        obj.p1 = p1;
        obj.p2 = p2;
    end

    % Important Method


    function ret = getInfluence(obj,P)
        ret = obj.Vortex.velVec(P);
    end

    function [grid,v] = velField(obj,xmin,ymin,xmax,ymax,Nx,Ny)
        [grid,v] = obj.Vortex.velField(xmin,ymin,xmax,ymax,Nx,Ny);
    end

    function v = velVec(obj,P)
        v = obj.Vortex.velVec(P);
    end

    function draw(obj)
        line([real(obj.p1) real(obj.p2)],[imag(obj.p1) imag(obj.p2)]);
        plot(obj.p1,'k.');
        plot(obj.p2,'k.');
    end

    function simpleDraw(obj)
        line([real(obj.p1) real(obj.p2)],[imag(obj.p1) imag(obj.p2)]);
```

```
        end

        function detailDraw(obj)
            obj.draw();
            plot(obj.Vortex.getCenter(),'ro');
            plot(obj.colocation,'rx');
        end

        % Setters
        function obj = setStrength(obj,nS)
            obj.Vortex = obj.Vortex.setStrength(nS);
        end

        % Getters
        function ret = getDirVec(obj)
            ret = (obj.p2-obj.p1);
            ret = ret./abs(ret);

        end
        function ret = getColocationPoint(obj)
            ret = obj.colocation;
        end

        function ret = getVortexLocation(obj)
            ret = obj.Vortex.getCenter();
        end

        function ret = getP1(obj)
            ret = obj.p1;
        end

        function ret = getP2(obj)
            ret = obj.p2;
        end

        function ret = getStrength(obj)
            ret = obj.Vortex.getStrength();
        end

    end
end
```

Published with MATLAB® R2022b