1 Introduction and the numerical method

In this report the flow over a NACA0012 airfoil is studied numerically by using the potential theory based panel method. Since experimental data of the pressure distribution and lift of the chosen airfoil profile is widely available[1], it is an excellent geometry to study the implementation of a numerical code.

By assuming that the flow field is incompressible, invicid and irrotational, the continuity equation can be reduced to a very solvable form known as the Laplace's equation

$$\nabla^2 \Phi = 0, \tag{1}$$

where Φ is the velocity potential related to velocity by the equation

$$\mathbf{u} = \nabla \Phi. \tag{2}$$

Since eq. 1 is linear, it is possible to use the principle of superposition to construct the solution to the equation from the sum of simple known functions (basic elements) that also satisfy the equation

$$\Phi = \sum_{k=1}^{n} c_k \Phi_k, \tag{3}$$

where c_k are some arbitrary constants.

In this report the Laplace's equation is solved using the Hess-Smith panel method approach, where the solution is constructed as a sum of three basic elements; sources, vortices and the free-stream potential

$$\Phi = \Phi_s + \Phi_v + \Phi_\infty. \tag{4}$$

Using the notation in figure 1, the basic elements can be defined as

$$\begin{aligned} \Phi_s &= \int \frac{q(s)}{2\pi} \ln r ds \\ \Phi_v &= -\int \frac{\gamma(s)}{2\pi} \theta ds \\ \Phi_\infty &= V_\infty (x \cos(\alpha) + y \sin(\alpha)), \end{aligned} \tag{5}$$

where $q(s)$ and $\gamma(s)$ denote the *unknown* source and vortex strengths respectively and they will be solved by a numerical algorithm.

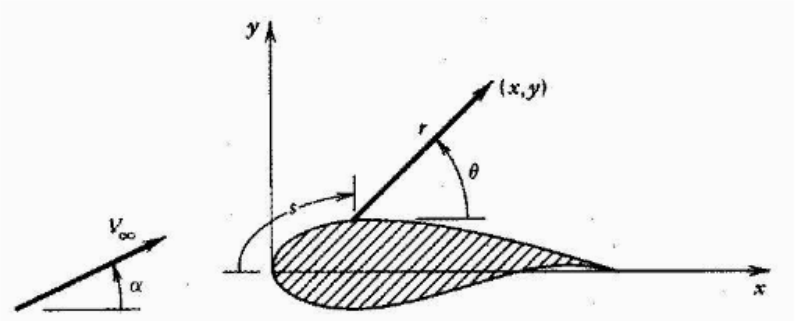

Figure 1: Notation and the coordinate system used. [2]

In the panel method used here, the airfoil geometry is discretized into straight line panel elements and nodal points, where the values of q and γ are located. An example discretization of the geometry is presented in figure 2. It should be noted that the amount of panels in the figure is significantly less than the amount used in the actual simulations.

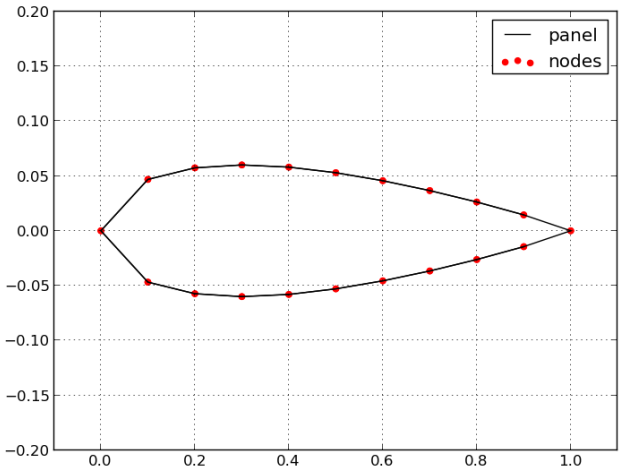

Figure 2: Discretized geometry. The panel numbering in this report begins from the trailing edge so that the beginning of the first panel is at $x, y = 1.0, 0$ and the numbering is in the clockwise direction.

In the Hess-Smith panel method the vortex strength is assumed constant $\gamma(s) = \gamma$ and we use the Kutta-Zhukhovsky (K-Z) condition

$$u_1 \cos(\theta_1) + v_1 \sin(\theta_1) = u_N \cos(\theta_N) + v_N \sin(\theta_N) \tag{6}$$

at the trailing edge to fix its value. Furthermore, the source strength is allowed to vary from panel to panel, so that together with the constant vortex strength value the flow tangency condition

$$-u_i \sin(\theta_i) + v_i \cos(\theta_i) = 0, \quad \text{i} = 1...\text{N} \tag{7}$$

is satisfied everywhere. With these assumptions eq. 4 can be written as

$$\Phi = V_\infty(x\cos(\alpha) + y\sin(\alpha)) + \sum_{j=1}^{N} \int_{panel} \left[\frac{q(s)}{2\pi} \ln r - \frac{\gamma}{2\pi}\theta \right] ds. \tag{8}$$

In the Hess-Smith panel method the source strength is allowed to vary form panel to panel, but it is assumed constant inside each panel $q(s) = q(i), i...N$. Because of the definition of the sources and vertices, we can not evaluate the velocites at the same locations of our singularities (nodal points), in the Hess-Smith approach the flow tangency and the K-Z conditions are "forced" at the panel centers instead

$$\begin{aligned} u_i &= u(\bar{x}_i, \bar{y}_i) \\ v_i &= v(\bar{x}_i, \bar{y}_i), \end{aligned} \tag{9}$$

where

$$\bar{x}_i = (x_{i+1} + x_i)/2, \quad \bar{y}_i = (y_{i+1} + y_i)/2.$$

The overbars denote the location of the panel centers and the coordinates without the overbars denote locations of the nodal points. Furthermore, the angles θ_i are evaluated as

$$\begin{aligned} \sin(\theta_i) &= (y_{i+1} - y_i)/l_i \\ \cos(\theta_i) &= (x_{i+1} - x_i)/l_i \end{aligned} \tag{10}$$

,

where the l_i denotes the length of panel i. Now the velocity at each panel center can be evaluated using the combined contribution of every source and vertex of every panel in the system. As the velocity induced by panel j to panel i is directly proportional to the source and vertex strengths at panel j, the panel center velocities can be written as

$$\begin{aligned} u_i &= V_\infty \cos(\alpha) + \sum_{j=1}^{N} q_j u_{Sij} + \gamma \sum_{j=1}^{N} u_{Vij} \\ v_i &= V_\infty \sin(\alpha) + \sum_{j=1}^{N} q_j v_{Sij} + \gamma \sum_{j=1}^{N} v_{Vij} \end{aligned} \tag{11}$$

Furthermore, we can transform the velocity components u and v into local coordinate systems, which are fixed to each panel j. In the local coordinate system, the x^*-axis (u^*) is parallel to the the panel and the y^*-axis (v^*) to the panel normal (see figure 3).

Using the relation between the potential function and velocities (eq. 2), the velocity components u and v induced by the sources can be written as

$$\begin{aligned} u^*_{Sij} &= \frac{1}{2\pi}\int_0^{l_j} \frac{x^* - t\ dt}{(x^*-t)^2 + y^{*2}} \\ v^*_{Sij} &= \frac{1}{2\pi}\int_0^{l_j} \frac{y^*\ dt}{(x^*-t)^2 + y^{*2}} \end{aligned} \tag{12}$$

and further simplified to

$$\begin{aligned} u^*_{Sij} &= \frac{-1}{2\pi}\ln\frac{r_{ij+1}}{r_{ij}} \\ v^*_{Sij} &= \frac{\beta_{ij}}{2\pi}. \end{aligned} \tag{13}$$

Very similarily, the velocities induced by the vortices can be written as

$$\begin{aligned} u^*_{Vij} &= \frac{1}{2\pi}\ln\frac{r_{ij+1}}{r_{ij}} \\ v^*_{Vij} &= \frac{\beta_{ij}}{2\pi}. \end{aligned} \tag{14}$$

In eqs. (13- 14) r_{ij} and r_{ij+1} denote distances from the center of panel i to the nodes j and $j+1$ respectively and the angle β_{ij} is the angle between vectors from nodes $j, j+1$ to the center of panel i. A demonstrating picture of the definitions is presented in figure 3.

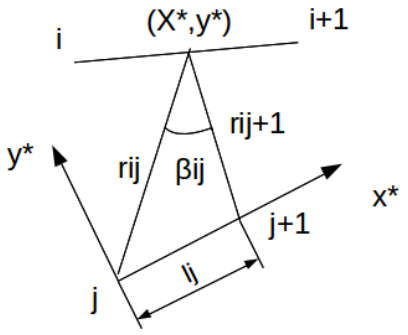

Figure 3: Definitions of β_{ij}, r_{ij} and r_{ij+1} .

Now the velocities (13- 14) can be substituted in the K-Z condition 6 and the flow tangency condition 7 to form a $(N+1)*(N+1)$ matrix equation for the source strengths and vorticity strength. The equation is of form

$$Ax = b, \tag{15}$$

where the indices A_{ij} in the matrix are

$$A_{ij} = \frac{1}{2\pi}\sin(\theta_i - \theta_j)\ln\frac{r_{ij+1}}{r_{ij}} + \cos(\theta_i - \theta_j)\beta_{ij} \tag{16}$$

and the vortex strength coefficients A_{iN+1}

$$A_{iN+1} = \frac{1}{2\pi}\sum_{j=1}^{N}\cos(\theta_i - \theta_j)\ln\frac{r_{ij+1}}{r_{ij}} - \sin(\theta_i - \theta_j)\beta_{ij}. \tag{17}$$

The equation formed from the K-Z condition is set on the last row of the matrix giving the coefficients A_{N+1j}

$$A_{N+1j} = \frac{1}{2\pi}\sum_{k=1,N}\sin(\theta_k - \theta_j)\beta_{kj} - \cos(\theta_k - \theta_j)\ln\frac{r_{kj+1}}{r_{kj}}, \tag{18}$$

where the summation is carried out only over the first ($k = 1$) and last ($k = N$) panels. Additionally, the coefficient A_{N+1N+1} has to be calculated separately

$$A_{N+1N+1} = \frac{1}{2\pi}\sum_{k=1,N}\sum_{j=1}^{N}\sin(\theta_k - \theta_j)\ln\frac{r_{kj+1}}{r_{kj}} + \cos(\theta_k - \theta_j)\beta_{kj}. \tag{19}$$

The corresponding RHS indices b_i are

$$\begin{aligned} b_i &= V_\infty \sin(\theta_i - \alpha) \\ b_{N+1} &= -V_\infty \cos(\theta_1 - \alpha) - V_\infty \sin(\theta_N - \alpha) \end{aligned} \tag{20}$$

Therefore, the full matrix is of form

$$\underbrace{\begin{bmatrix} & & & & & \overbrace{\phantom{A_{1N+1}}}^{\text{vorticity}} \\ A_{11} & \dots & A_{1i} & \dots & A_{1N} & A_{1N+1} \\ \vdots & & \vdots & & \vdots & \vdots \\ A_{i1} & \dots & A_{ii} & \dots & A_{iN} & A_{iN+1} \\ \vdots & & \vdots & & \vdots & \vdots \\ A_{N1} & \dots & A_{Ni} & \dots & A_{NN} & A_{N,N+1} \\ A_{N+1,1} & \dots & A_{N+1,i} & \dots & A_{N+1,N} & A_{N+1,N+1} \end{bmatrix}}_{\text{K-Z condition}} \begin{bmatrix} q_1 \\ \vdots \\ q_i \\ \vdots \\ q_N \\ \gamma \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_i \\ \vdots \\ b_N \\ b_{N+1} \end{bmatrix}. \tag{21}$$

Since the matrix A is "full" the computation of A^{-1} is quite expensive if the amount of panels N is large. Luckily, sufficient accuracy can be achieved with relatively small amount of panels.

When the column vector b is solved from equation 15 the corresponding (tangential)velocities at the panels can be solved by substituting the γ and q_i values back to eq. 11

$$\begin{aligned} V_{ti} =& V_\infty \cos(\theta_i - \alpha) + \frac{1}{2\pi} \sum_{j=1}^{N} q_j \left(\sin(\theta_i - \theta_j)\beta_{ij} - \cos(\theta_i - \theta_j) \ln \frac{r_{ij+1}}{r_{ij}} \right) \\ &+ \frac{\gamma}{2\pi} \sum_{j=1}^{N} \left(\sin(\theta_i - \theta_j) \ln \frac{r_{ij+1}}{r_{ij}} - \cos(\theta_i - \theta_j)\beta_{ij} \right). \end{aligned} \tag{22}$$

The pressure coefficient C_p can be calculated by applying the Bernoulli's equation between any point in the free stream and each panel separately. This results in

$$C_p = 1 - (\frac{V_{ti}}{V_\infty})^2. \tag{23}$$

The lift coefficient C_L on the other hand, can be calculated from Kutta-Zhukhovsky's theorem $L = \rho_\infty \Gamma V_\infty$, where Γ is the total circulation calculated by integrating the vorticity over the airfoil surface

$$\begin{aligned} \Gamma &= \sum_{i=1}^{N} \gamma l_i \\ \rightarrow C_L &= \frac{\sum_{i=1}^{N} \gamma l_i}{1/2 V_\infty c}. \end{aligned} \tag{24}$$

In eq. 24 c is the chord length of the airfoil.

2 Implementation

A diagram of the algorithm used in the code is presented in figure 4. The procedure begins with the initialization of the panels after which the A_{ij} coefficients are calculated. After this, the additional row and columns are added to the matrix followed by the calculation of the RHS column vector. Then the matrix is turned and the velocities are solved using the γ and q values obtained from the matrix equation. Finally the C_p and C_L values are calculated and output to a text file.

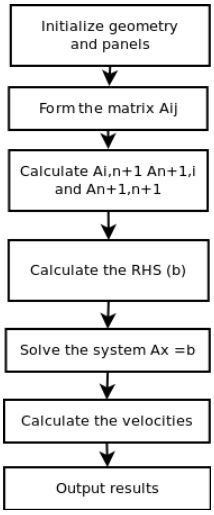

Figure 4: A diagram of the algorithm used.

The initialization of the geometry is based on creating a list of panel objects in a certain order. Each panel object is initialized by giving the initialization method the start and end coordinates of the panel and a rotating angle, so that the coordinates can be rotated to modify the angle of attack of the airfoil. Additionally, the panel object has useful methods (python functions) to calculate the parameters used in forming the matrices. For example, the method

`get_beta(self,neighbour)`

returns the β angle of the panel object i when a another panel j is given as an input parameter. The complete panel object code is given below

```
class Panel(object):

        def __init__(self,start_x,start_z,end_x,end_z,angle):
```

```
        """
        Rotate the points if alfa !=0
        """

        if (angle!=0.):

                theta = math.radians(-angle)
                start_x = start_x*np.cos(theta)-start_z*np.sin(theta)
                end_x   = end_x*np.cos(theta)-end_z*np.sin(theta)


                start_z = start_x*np.sin(theta)+start_z*np.cos(theta)
                end_z = end_x*np.sin(theta)+end_z*np.cos(theta)


        self.length = np.sqrt((end_x-start_x)**2 + (end_z - start_z)**2)
        self.xcoord = (start_x+end_x)/2.
        self.zcoord = (start_z+end_z)/2.

        self.start_x = start_x
        self.start_z = start_z

        self.end_x = end_x
        self.end_z = end_z

        self.sin_theta = (end_z - start_z)/self.length
        self.cos_theta = (end_x - start_x)/self.length



def get_rij(self,neighbour):
        """
        returns the length of rij vector
        """

        x1 = neighbour.start_x
        x2 = self.xcoord

        z1 = neighbour.start_z
        z2 = self.zcoord
```

```
        diff_x = x2 - x1
        diff_z = z2 - z1

        return (np.sqrt(diff_x**2 + diff_z**2))



def get_rij_plus(self,neighbour):
        """
        returns the length of rij+1
        """

        x1 = neighbour.end_x
        x2 = self.xcoord

        z1 = neighbour.end_z
        z2 = self.zcoord

        diff_x = x2 - x1
        diff_z = z2 - z1

        return (np.sqrt(diff_x**2 + diff_z**2))



def get_beta(self,neighbour):

        """
        returns the beta angle as radians
        """

        delta_y = (self.zcoord-neighbour.end_z)*(self.xcoord-neighbour.start_x)\
                -(self.xcoord-neighbour.end_x)*(self.zcoord-neighbour.start_z)

        delta_x = (self.xcoord-neighbour.end_x)*(self.xcoord-neighbour.start_x)\
                +(self.zcoord-neighbour.end_z)*(self.zcoord-neighbour.start_z)

        beta = math.atan2(delta_y, delta_x)

        if (self == neighbour):
                return math.pi
```

```
        else:
                return beta
```

The ordering of the panel objects in the list is the same as in figure 2. The coordinates, which are given for each panel object as an input are calculated in a method "panel_generator()", which first calculates the x-coordinates and then the corresponding y-coordinates (z-coordinates in the code). The positive y-coordinates are calculated from a formula of the symmetric NACA00xx airfoils so that the trailing edge is fixed at $x, y = coord\ length, 0$ and the leading edge at $x, y = 0, 0$. If an angle of attack other than 0 is given to the panel_generator(), the rotation of the points is done in a way that the leading edge stays at $x, y = 0, 0$ and the other points are moved. The negative y-coordinates are calculated by multiplying the positive coordinates by -1. The complete code for the panel_generator() method is given below:

```
def panel_generator(last_digits, coord_l, num_points,angle):


        t=(last_digits/100.)*coord_l
        c=coord_l
        x = np.arange(0.,c,c/num_points)


        x=np.append(x,c)

        #calculates the positive z-coordinates of naca00xx profile
        z_plus = 5.*t*c*(0.2969*np.sqrt(x/c)+(-0.1260)*(x/c)+(-0.3516)*(x/c)**2\
                        +(0.2843)*(x/c)**3+(-0.1015)*(x/c)**4)

        #fix the last index to have exactly zero value
        #and set the lower side z-coordinates to be z_positive*(-1)
        z_plus[-1] = 0.
        z_minus = -z_plus



        panel_list = []


        """
        Generates a list of panel objects.
        The panels are in exactly same order as in the lecture slides.
        Start of the fist panel is at x=coord length, z=0
```

```
"""

#pressure side panels (lower half)
i=0
while (i < len(z_minus)-1):

        start_x = x[len(x)-i-1]
        end_x = x[len(x)-i-2]
        start_z = z_minus[len(z_minus)-i-1]
        end_z = z_minus[len(z_minus)-i-2]
        panel_list.append(Panel(start_x,start_z,end_x,end_z,angle))
        i+=1

#suction side panels (upper half)
i=0
while (i < len(z_plus)-2):

        start_x=x[i]
        end_x = x[i+1]

        start_z = z_plus[i]
        end_z = z_plus[i+1]

        panel_list.append(Panel(start_x,start_z,end_x,end_z,angle))

        i+=1

"""
Closes the system=adds the final panel from the end
of the last panel (at this point)
to the start of the first panel
"""
panel_list.append(Panel(panel_list[-1].end_x,panel_list[-1].end_z,
            \ panel_list[0].start_x,panel_list[0].start_z,0.))

return panel_list
```

The matrix A_{ij} is formed in "matrix_Aij()"-method using a double loop, where for each panel i, a separate loop is used to calculate the coefficients by all the panels j in the panel list. Then the outer loop moves to the panel

$i + 1$ and the inner loop, again, loops over all the panels. This corresponds to eq. 16 in the first section.

```
def matrix_Aij(panels):

    #Initialize the matrix
    A_ij = np.zeros((len(panels)+1,len(panels)+1))


    n=0; m=0;
    # i is now a panel object
    for i in panels:
        #loop over all neighbour panel objects
        for j in panels:


            rij = i.get_rij(j)
            rij_plus = i.get_rij_plus(j)
            beta = i.get_beta(j)
            #sin(0i - 0j)*ln(r_ij+/r_ij)
            eka_termi = (j.cos_theta*i.sin_theta - \
            i.cos_theta*j.sin_theta)*math.log(rij_plus/rij)



            #B*cos(0i-0j)
            toka_termi = beta*(i.cos_theta*j.cos_theta \
                  + i.sin_theta*j.sin_theta)


            aij = 1./(2*np.pi)*(eka_termi + toka_termi)


            A_ij[n,m] = aij


            m+=1


        m=0
        n+=1


    return A_ij
```

The columns additional column and row of the matrix A is formed in method "matrix_Aij_plus()" in a very similar manner and the operations in the method correspond to equations 17-19 in the first section. It should be noted that the summations of the first and last panel in equations 18 and 19 are done in separate loops.

```
def matrix_Aij_plus(panels,A_ij):

    ##########################################################################################
                                        #A_i,n+1
    #Calculates A_i,n+1
    n=0; m=0;


    for i in panels:
            summa = 0.
            for j in panels:
                    #calculates the sum of all neighbour panels
                    rij = i.get_rij(j)
                    rij_plus = i.get_rij_plus(j)
                    beta = i.get_beta(j)
                    eka_termi = math.log(rij_plus/rij)\
                            *(i.cos_theta*j.cos_theta + i.sin_theta*j.sin_theta)

                    toka_termi = beta*(j.cos_theta*i.sin_theta - \
                          i.cos_theta*j.sin_theta)

                    summa += (eka_termi - toka_termi)



            A_ij[n,-1] = 1./(2.*np.pi)*(summa)
            n+=1


    ##########################################################################################
                                        #A_n+1,j

    #Calculates the first summation (k=1) of A_n+1,j
    #k = 1
    m=0
    i = panels[0]
    for j in panels:
            rij = i.get_rij(j)
            rij_plus = i.get_rij_plus(j)
            beta = i.get_beta(j)

            eka_termi = beta*(j.cos_theta*i.sin_theta - i.cos_theta*j.sin_theta)
```

```
        toka_termi = math.log(rij_plus/rij)*\
                   (i.cos_theta*j.cos_theta + i.sin_theta*j.sin_theta)

        temp = (eka_termi - toka_termi)
        A_ij[-1,m] = temp
        m+=1
```

#Adds the second summation to the first summation (k=N)

```
m=0
i = panels[-1]
for j in panels:
        rij = i.get_rij(j)
        rij_plus = i.get_rij_plus(j)
        beta = i.get_beta(j)

        eka_termi = beta*(j.cos_theta*i.sin_theta - i.cos_theta*j.sin_theta)

        toka_termi = math.log(rij_plus/rij)*\
                 (i.cos_theta*j.cos_theta + i.sin_theta*j.sin_theta)

        temp = (eka_termi - toka_termi)
        A_ij[-1,m] += temp
        m+=1
```

*#Both now calculated, divide by 2*pi*

```
A_ij[-1,:]=A_ij[-1,:]/(2*np.pi)
```

###

#A_{n+1,n+1}

#k = 1
#j=1..N

```
summa1 = 0.
i = panels[0]
for j in panels:

        rij = i.get_rij(j)
        rij_plus = i.get_rij_plus(j)
```

```
        beta = i.get_beta(j)

        eka_termi = math.log(rij_plus/rij)*\
        (j.cos_theta*i.sin_theta - i.cos_theta*j.sin_theta)

        toka_termi = beta*(i.cos_theta*j.cos_theta + i.sin_theta*j.sin_theta)

        summa1 += (eka_termi + toka_termi)


#k=N
#j=1..N
summa2 = 0.
i = panels[-1]
for j in panels:

        rij = i.get_rij(j)
        rij_plus = i.get_rij_plus(j)
        beta = i.get_beta(j)

        eka_termi = math.log(rij_plus/rij)*\
              (j.cos_theta*i.sin_theta - i.cos_theta*j.sin_theta)

        toka_termi = beta*(i.cos_theta*j.cos_theta + i.sin_theta*j.sin_theta)

        summa2 += (eka_termi + toka_termi)


summa = summa1+summa2

A_np_np = summa/(2.*np.pi)

A_ij[-1,-1] = A_np_np

return A_ij
```

The RHS of the equation set is formed in method "RHS()". It should be noted that the y-direction velocity W in the method (and other methods as well) is set to zero here, since the angle of attack is modified by rotating the points instead of altering the direction of the free stream velocity. The operations in the method correspond to equation 20 in the first section.

```
def RHS(panels,U,W,alfa):
        b = np.zeros(len(panels)+1)
        n = 0
        for i in panels:
                temp = i.sin_theta*np.cos(alfa)-i.cos_theta*np.sin(alfa)
                b[n] = np.sqrt((U**2 + W**2))*temp
                n+=1

        #b_N+1
        #cos(theta_1 - alfa)
        temp1 = panels[0].cos_theta*np.cos(alfa) + panels[0].sin_theta*np.sin(alfa)

        #cos(theta_N - alfa)
        temp2 = panels[-1].cos_theta*np.cos(alfa) + panels[-1].sin_theta*np.sin(alfa)

        b[-1] = -np.sqrt(U**2 + W**2)*(temp1 + temp2)
        return b
```

The matrix system is turned using the "linalg.solve()"-method from the Python's Numpy library. It proved to be quite difficult to determine which algorithm the method uses in solving the equations system, since the method changes the algorithm based on the type of the coefficient matrix. However, in the documentation of Numpy it was stated that the routines used in the method are from the LAPACK library [3].

The tangential velocities are calculated in method "Velocities()" using the equation 22 of the first section, again, using double loop:

```
def Velocities(panels,x,U,W,alfa):
        velocities = np.zeros(len(x))

        #n corrsepnds to i th index and m to j th.
        #new variables have to be used since i and j are now objects!
        n=0; m=0; #i,j

        for i in panels:
                #set both summations of i to zero
                summa1 = 0.
                summa2 = 0.
                m=0
                for j in panels:
```

```
            rij = i.get_rij(j)
            rij_plus = i.get_rij_plus(j)
            beta = i.get_beta(j)

            #calculates the two summations of i velocity

            eka_termi_sigma = beta*\
             (j.cos_theta*i.sin_theta - i.cos_theta*j.sin_theta)
            toka_termi_sigma = math.log(rij_plus/rij)*\
                (i.cos_theta*j.cos_theta + i.sin_theta*j.sin_theta)

            #sigma(j)*(B*sin(0i - 0j) - ln(rij+/rij)*cos(0i-0j))
            summa1 += x[m]*(eka_termi_sigma - toka_termi_sigma)


                                #ln(rij+/rij)*sin(0i-0j)+B*cos(0i-0j)
            eka_termi_gamma = math.log(rij_plus/rij)*\
           (j.cos_theta*i.sin_theta - i.cos_theta*j.sin_theta)
            toka_termi_gamma = beta*\
               (i.cos_theta*j.cos_theta + i.sin_theta*j.sin_theta)
            summa2 += (eka_termi_gamma + toka_termi_gamma)


            m+=1

        #temp = cos(0i-alfa)
        temp = panels[n].cos_theta*np.cos(alfa)\
             + panels[n].sin_theta*np.sin(alfa)
        velocities[n] = np.sqrt(U**2 + W**2)*temp \
                        + 1./(2.*np.pi)*(summa1) \
                        + x[-1]/(2.*np.pi)*(summa2)



        n+=1

    return velocities
```

The "main()"-method of the code is used for setting the input parameters for the geometry and running the "run_code()" method in a single loop, where the angle of attack is different for each run. Other input parameters are kept constant in the different runs. The structure of the "run_code()"-method is

the same as in the diagram presented in figure 4.

```
def main():

        U = 1.
        W = 0.
        panel_amount = 150
        last_digits = 12
        angle_list = np.arange(-5,16,1)

        for angle in angle_list:
                run_code(U,W,angle,panel_amount,last_digits)

def run_code(U,W,angle,panel_amount,last_digits):
        c = 1.
        #this is the alpha of the free stream velocity
        alfa = 0.

        #create panels
        panels = panel_generator(last_digits,c,panel_amount,angle)

        #forms the Aij matrix and calculates the values 1...N* 1...N
        mat_AIJ = matrix_Aij(panels)

        #calculates values A_i,n+1, A_n+1,j and A_n+1,n+1
        final_A = matrix_Aij_plus(panels,mat_AIJ)

        #calculates the right hand side of the system
        b = RHS(panels,U,W,alfa)

        #solves the system
        x = np.linalg.solve(final_A,b)

        #calculates the tangential velocities from the sigma and gamma values
        v=Velocities(panels,x,U,W,alfa)

        #output results
        post_process(v,panels,U,W,x,c,angle)
```

3 Results and discussion

The simulations were run with the free stream velocity $V_\infty = 1^{\mathrm{m}}/_{\mathrm{s}}$ for NACA0012 airfoil with $c = 1\mathrm{m}$ coord length. The angle of attack was varied from $\alpha \in [-5°, 15°]$ with $1°$ intervals. The negative α values denote the trailing edge of the airfoil being higher than the leading edge and the range of the angle was chosen because of the measurement data being available for that range. The geometry was discretized into $N = 150$ equally spaced panels for all simulations.

The calculated C_p values along the airfoil for $\alpha = 0°, 10°, 15°$ are presented in figure 5. The figure shows the corresponding measurement data from [1] for the upper side of the airfoil. It can be seen that the calculated results are quite close to the measured ones except for the trailing edge, where strange behaviour can be seen. At the trailing edge, the pressure distributions on the upper and lower surfaces should be the same due to the K-Z codition set at panel 1 and N. The strange behaviour at the trailing edge is caused by some small programming error in the code, most likely in the rotation or generation of the panels.

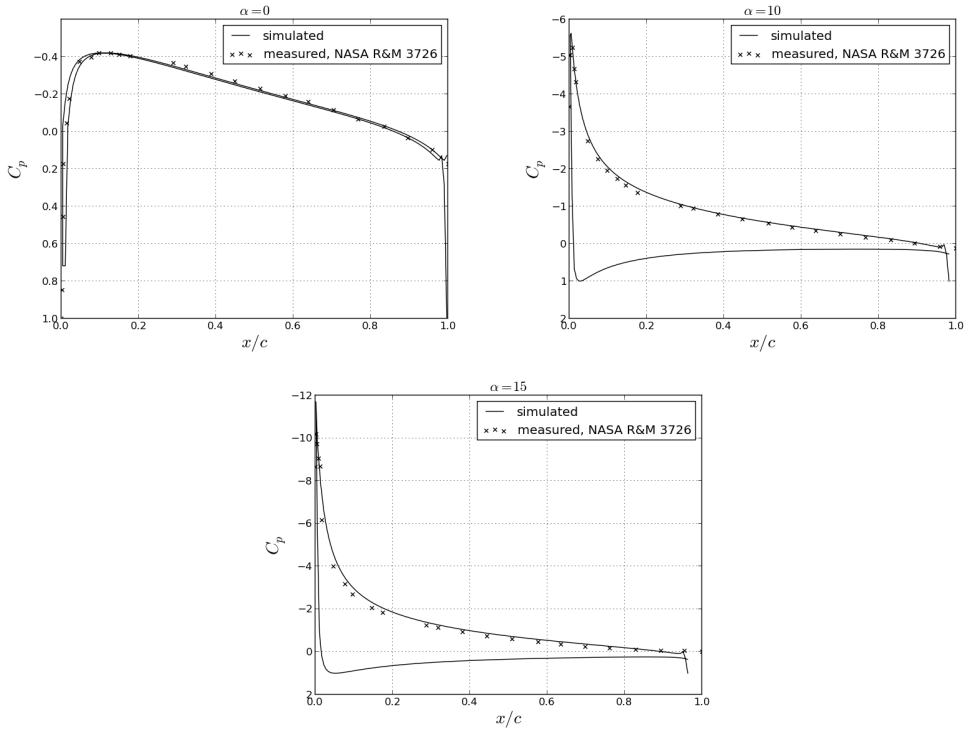

Figure 5: Pressure distributions along the airfoil.

Figure 6 shows the lift coefficient of the airfoil as a function of the angle of attack. The straight line in the figure shows the theoretical *inviscid* C_L value of the airfoil. Again the calculated results are close to the measured ones, except for the $\alpha > 10^\circ$ when the measured data clearly separates from the calculated ones. The sudden decrease in the measured lift value at higher α values can be explained by the viscous boundary layer separation (or stall) on the upper surface of the airfoil. In stalled flow, the upper surface pressure distribution deviates from the inviscid theory, which is used here, resulting in a sudden loss of lift and increase in the drag force of the airfoil[4]. As long as the angle of attack is reasonably small, $\alpha < 10$ in this case, the lift can be calculated using the inviscid theory and the assumptions made in eq. 1 are valid.

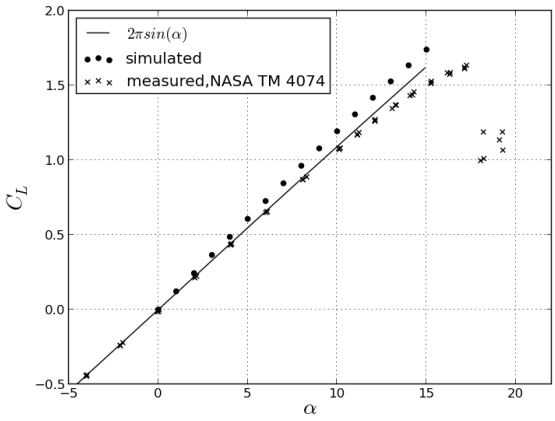

Figure 6: The lift coefficients as a function of α.

To summarize, the code developed and used here seems to give fairly accurate results as long as the viscous effects in the flow play only a minor role. The method used here, without any modifications, can not be used when separation of flow occurs. The computational efficiency of the Hess-Smith panel method is very good and the twenty simulations ran in this study took less than 0.1 CPU hours. The short computational times are extreamely useful for example in optimization of the geometry or angle of attack.

The total time spent on the assignment was approximately 45 hours, most of which was used in trying to find a stupid error in the code. The writing of the report took also around 10 hours. With more care taken in the implementation of the first version of the code, at least 10 hours could've been reduced from the total time spent on the assignment. Cooperation

with Markus was quite substantial in the assignment, especially in forming of the matrix. At first, I had understood the double summations in the K-Z condition wrong and without any help the code would've never been finished.

References

[1] NASA Langley Research Center, Turbulence Modelling Resource, `http://turbmodels.larc.nasa.gov/naca0012_val.html`

[2] Moran J. An Introduction to Theoretical and Computational Aerodynamics Courier Corporation, 1984, ISBN: 0486428796

[3] Anderson, E. and Bai, Z. and Bischof, C. and Blackford, S. and Demmel, J. and Dongarra, J. and Du Croz, J. and Greenbaum, A. and Hammarling, S. and McKenney, A. and Sorensen, D.,
LAPACK Users' Guide, 3rd edition, Society for Industrial and Applied Mathematics, 1999, Philadelphia, PA,
ISBN = 0-89871-447-8

[4] White F. M. Viscous Fluid Flow, Third edition, McGraw-Hill, 2006, ISBN: 007-124493-X