Two Dimensional Euler Equations
Transonic Flow Around a NACA 0012 Airfoil

Abhiram Aithal
Hannah Fix

AA 543 - CFD
Homework 5
March 15, 2016

1 Introduction

The Euler equations are quasilinear hyperbolic equations for adiabatic inviscid flow. For a two dimensional airfoil these equations can be solved using a finite volume method. However, due to the non-monotonicity preserving nature of explicit schemes for hyperbolic (convection) problems, artificial viscosity is introduced to damp spurious oscillations in the solution. These oscillations appear near regions of high pressure differences and velocity jumps, such as shocks. Transonic flow and the appearance of shocks was studied over a NACA 0012 airfoil. Jameson's artificial diffusivity terms both a second order and a fourth order term are used [1]. This allows more accurate capturing of the shock and prevents the solution from becoming unstable around the region of the shock.

The CFD results for two dimensional transonic flow around a NACA 0012 airfoil using the Euler equations are presented below for two different angle of attacks and with two different levels of mesh refinement.

2 Numerical Method

The two dimensional Euler equations are solved on a curvilinear grid using a finite volume method. The equations are solved for a NACA 0012 airfoil in transonic flow. The initial conditions and boundary conditions are prescribed as defined below. Local time stepping is used with a 4th order Runge-Kutta method for time integration until steady state is reached. Jameson artificial diffusivity is applied to damp out the oscillations due to the convective terms.

2.1 Euler Equations

The two dimensional Euler equations are defined as,

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho u}{\partial x} + \frac{\partial \rho v}{\partial y} = 0, \tag{1}$$

$$\frac{\partial \rho u}{\partial t} + \frac{\partial (\rho u^2 + p)}{\partial x} + \frac{\partial (\rho uv)}{\partial y} = 0, \tag{2}$$

$$\frac{\partial \rho v}{\partial t} + \frac{\partial (\rho uv)}{\partial x} + \frac{\partial (\rho v^2 + p)}{\partial y} = 0, \tag{3}$$

$$\frac{\partial \rho E}{\partial t} + \frac{\partial (\rho uH)}{\partial x} + \frac{\partial (\rho vH)}{\partial y} = 0. \tag{4}$$

For this problem the assumption is no external forces are applied. This can be written as a vector equation as follows,

$$\frac{\partial U}{\partial t} + \frac{\partial F}{\partial x} + \frac{\partial G}{\partial y} = 0, \tag{5}$$

where the vectors U, F, and G are defined as follows,

$$U = \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{bmatrix} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho E \end{bmatrix}, \quad F = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ \rho uH \end{bmatrix} = \begin{bmatrix} U_2 \\ \frac{U_2^2}{U_1} + (\gamma - 1)\left(U_4 - \frac{(U_2^2 + U_3^3)}{2U_1}\right) \\ \frac{U_2 U_3}{U_1} \\ U_2\left(\gamma \frac{U_4}{U_1} - \frac{(\gamma-1)}{2U_1^2}(U_2^2 + U_3^2)\right) \end{bmatrix}, \tag{6}$$

$$G = \begin{bmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ \rho vH \end{bmatrix} = \begin{bmatrix} U_3 \\ \frac{U_2 U_3}{U_1} \\ \frac{U_3^2}{U_1} + (\gamma - 1)\left(U_4 - \frac{(U_2^2 + U_3^3)}{2U_1}\right) \\ U_3\left(\gamma \frac{U_4}{U_1} - \frac{(\gamma-1)}{2U_1^2}(U_2^2 + U_3^2)\right) \end{bmatrix}. \tag{7}$$

Note, the flux vectors can be written in terms of U variables, using the Ideal Gas relations with enthalpy H, energy E, and pressure p,

$$H = E + \frac{p}{\rho}, \qquad E = e + \frac{1}{2}(u^2 + v^2), \qquad p = (\gamma - 1)\rho e. \tag{8}$$

Equations for Mach number and entropy were also used,

$$M = \frac{|\mathbf{v}|}{c}, \qquad c = \sqrt{\gamma \frac{p}{\rho}}, \qquad \frac{S + S_0}{R} = \frac{3}{2} \ln\left(\frac{p}{\rho}\right), \tag{9}$$

where R is the ideal gas constant and $|\mathbf{v}|$ is the magnitude of the velocity. The equations are solved by first setting up the domain, the grid, and the initial conditions. Then the discretized equations are solved over the domain for all the time steps until steady state is achieved.

The conservative finite volume formulation of the two dimensional Euler equations (Eq. 5) is

$$\frac{d}{dt}\left(U_{i,j}\Omega_{i,j}\right) + \sum_{\texttt{faces } i,j} \mathbf{F}^{*(AV)} \cdot \Delta\mathbf{s} = 0. \tag{10}$$

2.2 Initial Conditions

The initial Mach was chosen as $M_\infty = 0.85$ for transonic flow. The initial conditions for density, pressure, etc. were determined based on standard atmospheric conditions at an altitude of $h = 1000m$ above sea level, defined as

$$p_\infty = 89875Pa, \quad \rho_\infty = 1.112kg/m^3, \quad T_\infty = 281.6K. \tag{11}$$

The velocities were found from Mach as a function of angle of attack,

$$u = M_\infty c_\infty \cos(\alpha), \quad v = M_\infty c_\infty \sin(\alpha), \tag{12}$$

where c_∞ is the speed of sound as defined in Eq. 9 and α is the angle of attack defined in radians. From these values and Eq. 8 the E_∞ can be found and the U vector can be defined.

2.3 Boundary Conditions

Characteristics were used to define the boundary conditions and determine which boundary conditions were physical and which were numerical. The sign of eigenvalues of the characteristics determine whether the boundary conditions are physical or not. The inlet region is $\mathbf{u} \cdot \mathbf{n} > 0$, the velocity vector dotted with the normal pointing into the domain. The outlet region is the reverse $\mathbf{u} \cdot \mathbf{n} < 0$. The boundary conditions in the i direction, around the airfoil were periodic at the trailing edge where the domain is "cut" to allow indexing around.

2.3.1 Inlet Boundary Conditions

The inlet has the following eigenvalues,

$$\lambda_1 = \lambda_2 = u_n, \quad \lambda_3 = u_n + c, \quad \lambda_4 = u_n - c. \tag{13}$$

λ_1, λ_2, and λ_3 are all positive and thus three physical boundary conditions need to be applied. The boundary condition corresponding to λ_4 is numerical. The boundary conditions for the inlet are defined as,

$$R^+_{nB} = u_{nB} + \frac{2c_B}{\gamma - 1} = u_{n\infty} + \frac{2c_\infty}{\gamma - 1}, \tag{14}$$

$$R^-_{nI} = u_{nB} - \frac{2c_B}{\gamma - 1} = u_{nI} - \frac{2c_I}{\gamma - 1}, \tag{15}$$

where the subscript B represents the ghost cell and the subscript I represents the interior cell that is adjacent to the boundary. These two conditions can be combined to solve for the normal velocity and the speed of sound for the ghost cell,

$$u_{nB} = \frac{R^+_{nB} + R^-_{nI}}{2} \tag{16}$$

$$c_B = (R^+_{nB} - R^-_{nI})\frac{\gamma - 1}{4}. \tag{17}$$

The last two conditions used are both physical boundary conditions,

$$u_{tB} = u_{t\infty} \tag{18}$$

where u_t is the tangential velocity.

A relation for entropy can be applied

$$S_B = S_\infty \tag{19}$$

alternatively a relation for enthalpy can be used

$$H_B = H_\infty. \tag{20}$$

We thus obtain four conditions for four unknowns ρ_B, u_B, v_B and E_B. The x- and y- components of velocity can be obtained by solving two linear equations,

$$u_{nB} = u_B n_x + v_B n_y \quad \text{and} \quad u_{tB} = u_B t_x + v_B t_y \tag{21}$$

where (n_x, n_y) and (t_x, t_y) are the normal and tangential unit vectors. ρ_B and E_B can then be found using the other two conditions on c_B and S_B.

2.3.2 Outlet Boundary Conditions

The outlet boundary conditions are also applied based on the characteristics. At the outlet only one eigenvalue is positive and three are negative, thus one physical and three numerical boundary conditions are applied for the outlet as follows.

$$R^-_{nB} = -|u_{nB}| - \frac{2c_B}{\gamma - 1} = R^-_{n\infty} = -|u_{n\infty}| - \frac{2c_\infty}{\gamma - 1} \tag{22}$$

$$R^+_{nB} = -|u_{nB}| + \frac{2c_B}{\gamma - 1} = R^+_{nI} = -|u_{nI}| - \frac{2c_I}{\gamma - 1} \tag{23}$$

$$u_{tB} = u_{tI} \tag{24}$$

$$S_B = S_I, \quad \texttt{or} \quad \left.\frac{p}{\rho^\gamma}\right|_B = \left.\frac{P}{\rho^\gamma}\right|_I. \tag{25}$$

As with the inlet the B cell represents the ghost cell which lies outside the domain. The I cell is on the interior of the domain next to the boundary.

2.3.3 Airfoil Boundary Conditions

The airfoil boundary represents a solid wall, which for inviscid flow uses a zero normal velocity. The no-slip condition at walls can only be applied for viscous flow. The boundary condition at the wall is

$$u_n = 0. \tag{26}$$

This leads to the flux dotted with the unit normal for the airfoil wall to be

$$\mathbf{F} \cdot \Delta\mathbf{s} = \begin{bmatrix} 0 \\ p\Delta s_x \\ p\Delta s_y \\ 0 \end{bmatrix}. \tag{27}$$

This equation is used for face the corresponds to the wall during the summation of fluxes around cells adjacent to the wall.

2.4 Spacial Discretization

The grid used was a non-Cartesian body-fitted grid as shown in Fig. 1. The outer boundary of the domain is a circle of radius 10.

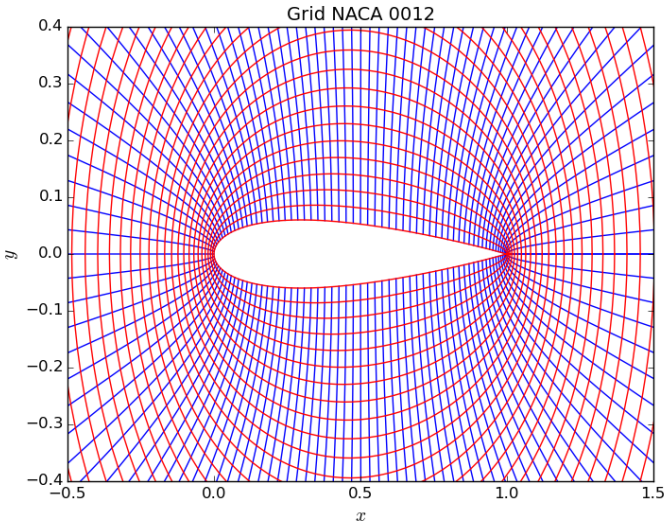

Figure 1: Grid around a NACA 0012 airfoil.

As a comparison the grid was then refined to allow better capturing of the shock and flow features. The refined grid, consisting of 257×129 grid points is shown in Fig. 2

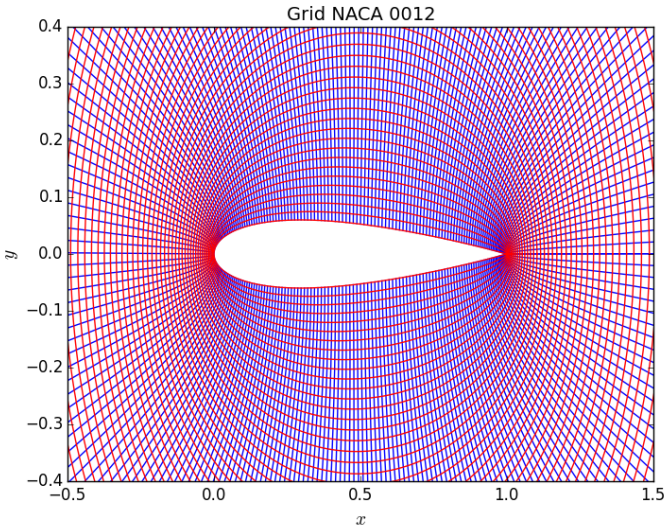

Figure 2: Refined Grid around a NACA 0012 airfoil.

A cell centered finite volume method was used for solving the two dimensional Euler equations. The cell volumes and normals are needed. Then the fluxes are calculated at the faces of the cells using the outward normals. The cell volumes are calculated as,

$$\Omega_{i,j} = \frac{1}{2}|\mathbf{x}_{AC} \times \mathbf{x}_{BD}| = \frac{1}{2}|(x_C - x_A)(y_D - y_B) - (x_D - x_B)(y_C - y_A)|, \tag{28}$$

where A, B, C, and D are the four veticies defining the cell with coordinates (x_A, y_A), (x_B, y_B), etc. The cell centers are defined as an average of the 4 vertices,

$$\mathbf{x}_{i,j} = \frac{1}{4}(\mathbf{x}_A + \mathbf{x}_B + \mathbf{x}_C + \mathbf{x}_D). \tag{29}$$

For the finite volume method, the flux dotted with the outward normal and summed up over the cell faces. Where the flux vector is made up of the fluxes F and G,

$$\begin{aligned}\sum_{\texttt{faces } i,j} \mathbf{F}^{*(AV)} \cdot \Delta\mathbf{s} = (F\Delta s_x + G\Delta s_y)_{i+1/2,j} + (F\Delta s_x + G\Delta s_y)_{i,j+1/2} \\ +(F\Delta s_x + G\Delta s_y)_{i-1/2,j} + (F\Delta s_x + G\Delta s_y)_{i,j-1/2}\end{aligned} \tag{30}$$

The fluxes at the cell faces are defined as an average of the two cells on either side of the face,

$$F_{i+1/2,j} = \frac{1}{2}(F_{i+1} + F_i). \tag{31}$$

The outward normals $\Delta\mathbf{s}$ are calculated for each of the faces as

$$\Delta\mathbf{s}_{i+1/2,j} = \Delta\mathbf{s}_{AB} = \Delta y_{AB}\mathbf{e_x} - \Delta x_{AB}\mathbf{e_y}. \tag{32}$$

2.5 Solver

A 4th order Runge-Kutta method is used for time integration until steady state is reached. The time integration involves for steps and uses the residual. The residual is defined as the sum of the fluxes over the cell faces,

$$R_{i,j} = \sum_{\texttt{faces } i,j} \mathbf{F}^{*(AV)} \cdot \Delta\mathbf{s}. \tag{33}$$

The residual is and used to calculate the U vector at each step of the time integration as follows,

$$\begin{aligned} U_{i,j}^{(1)} &= U_{i,j}^{n} - \frac{\Delta t_{i,j}}{\Omega_{i,j}} \alpha_1 R_{i,j}^{n} \\ U_{i,j}^{(2)} &= U_{i,j}^{n} - \frac{\Delta t_{i,j}}{\Omega_{i,j}} \alpha_2 R_{i,j}^{(1)} \\ U_{i,j}^{(3)} &= U_{i,j}^{n} - \frac{\Delta t_{i,j}}{\Omega_{i,j}} \alpha_3 R_{i,j}^{(2)} \\ U_{i,j}^{n+1} &= U_{i,j}^{n} - \frac{\Delta t_{i,j}}{\Omega_{i,j}} \alpha_4 R_{i,j}^{(3)} \end{aligned} \tag{34}$$

This time integration is derived from the conservative finite volume formulation Eq. 10. For the steady solution of the Euler equations, a local time step can be used which is determined by the stability for each cell. This can be calculated as,

$$\tau_{i,j} = \frac{\Delta t_{i,j}}{\Omega_{i,j}} \leq \frac{CFL\#}{|(\mathbf{u}+c)_{i,j} \cdot \Delta \mathbf{s}_i| + |(\mathbf{u}+c)_{i,j} \cdot \Delta \mathbf{s}_j|}. \tag{35}$$

The CFL # is based off the stability of the scheme and depends on the α values selected for the time integration. The $\Delta \mathbf{s}_i$ and $\Delta \mathbf{s}_j$ are defined as,

$$\Delta \mathbf{s}_i = \frac{1}{2}(\Delta \mathbf{s}_{i+1/2,j} - \Delta \mathbf{s}_{i-1/2,j}), \tag{36}$$

$$\Delta \mathbf{s}_j = \frac{1}{2}(\Delta \mathbf{s}_{i,j+1/2} - \Delta \mathbf{s}_{i,j-1/2}) \tag{37}$$

for outward facing normals.

The α values used for standard 4th Order Runge Kutta are

$$\alpha_1 = \frac{1}{4}, \quad \alpha_2 = \frac{1}{3}, \quad \alpha_3 = \frac{1}{2}, \quad \alpha_4 = 1. \tag{38}$$

The α values for time integration were chosen based off the work by Jameson Ref. [1], as

$$\alpha_1 = \frac{1}{8}, \quad \alpha_2 = 0.306, \quad \alpha_3 = 0.587, \quad \alpha_4 = 1, \tag{39}$$

with these α values the max CFL # for stability is 1.7. Although local time stepping allows for faster convergence to steady state, it is still an explicit scheme and thus there is still a stability limit. Note that this maximum value is smaller than the one prescribed by Jameson [1]. This is due to the fact that the grid used here is finer.

Convergence to steady state is determined by the norm of the residuals. To ensure the solution had converged the max norm of all the residuals need to be less than 10^{-4} times the initial values (a small prescribed tolerance). An alternative method is to use the grid two norm instead of the max norm. The measure of convergence used in Jameson's original results was the root mean square of the residual for density and the root mean square of deviation of total enthalpy from its free stream value [1].

2.6 Jameson Artificial Diffusivity

The non monotonicity preserving second order scheme in two dimensions can cause the solution to become unstable due to the large oscillations in the pressure and velocity terms. To minimize these oscillations artificial diffusivity can be used with both second order and fourth order terms. The fourth order terms are necessary in two dimensions to achieve a stable solution. The Jameson artificial diffusivity is subtracted from the numerical flux as follows

$$\left(\mathbf{F}^{*(AV)} \cdot \Delta \mathbf{s}\right)_{i+1/2,j} = \frac{1}{2}\left(\mathbf{F}_{i,j}^{*} + \mathbf{F}_{i+1,j}^{*}\right) \cdot \Delta \mathbf{s}_{i+1/2,j} - D_{i+1/2,j} \tag{40}$$

where $\mathbf{F}$ is the flux vector and D is the artificial diffusivity term. Similar terms are used for all 4 cell faces. The specifics of the $D_{i+1/2,j}$ and $D_{i-1/2,j}$ artificial diffusivity terms are given below, the terms on the $i, j+1/2$ and $i, j-1/2$ faces are the similar

$$D_{i+1/2,j} = \epsilon^{(2)}_{i+1/2,j} \left(U_{i+1,j} - U_{i,j}\right) - \epsilon^{(4)}_{i+1/2,j} \left(U_{i+2,j} - 3U_{i+1,j} + 3U_{i,j} - U_{i-1,j}\right), \tag{41}$$

$$D_{i-1/2,j} = \epsilon^{(2)}_{i-1/2,j} \left(U_{i,j} - U_{i-1,j}\right) - \epsilon^{(4)}_{i-1/2,j} \left(U_{i+1,j} - 3U_{i,j} + 3U_{i-1,j} - U_{i-2,j}\right). \tag{42}$$

The $\epsilon^{(2)}_{i+1/2,j}$ can be defined as,

$$\epsilon^{(2)}_{i+1/2,j} = \frac{1}{2}\kappa^{(2)} \left(\mathbf{u} \cdot \Delta\mathbf{s} + c|\Delta\mathbf{s}|\right)_{i+1/2,j} \max(\nu_{i-1}, \nu_i, \nu_{i+1}, \nu_{i+2}) \tag{43}$$

Likewise, $\epsilon^{(2)}_{i-1/2}$ was defined as,

$$\epsilon^{(2)}_{i-1/2,j} = \frac{1}{2}\kappa^{(2)} \left(\mathbf{u} \cdot \Delta\mathbf{s} + c|\Delta\mathbf{s}|\right)_{i-1/2,j} \max(\nu_{i-2}, \nu_{i-1}, \nu_i, \nu_{i+1}) \tag{44}$$

The $\mathbf{u}$ and c at the cell face can just be taken as an average of the values for the two cells the face is between. The ν_i depends on the pressure ratio and is defined as,

$$\nu_i = \frac{|p_{i+1,j} - 2p_{i,j} + p_{i-1,j}|}{p_{i+1,j} + 2p_{i,j} + p_{i-1,j}}, \tag{45}$$

this allows artificial viscosity to damp the regions of high pressure ratio such as regions near shocks.

The higher order coefficient, $\epsilon^{(4)}_{i+1/2}$ is defined as,

$$\epsilon^{(4)}_{i+1/2,j} = \max\left(0, \frac{1}{2}\kappa^{(4)} \left(\mathbf{u} \cdot \Delta\mathbf{s} + c|\Delta\mathbf{s}|\right)_{i+1/2,j} - \epsilon^{(2)}_{i+1/2,j}\right) \tag{46}$$

Based on the artificial diffusivity used by Jameson used the diffusion coefficients,

$$\kappa^{(2)} = \frac{1}{4} \quad and \quad \kappa^{(4)} = \frac{1}{256}. \tag{47}$$

3 Results

3.1 Flow around a NACA 0012 Airfoil at an Angle of Attack of 0 Degrees

The plots for speed, pressure, pressure coefficient, Mach, enthalpy, entropy, and density are shown in Fig. 3 to 9 for a NACA 0012 airfoil at 0° angle of attack. It can be noted for the 0° angle of attack the symmetric airfoil produces symmetric results, the pressure, velocity, etc. are all symmetric at the top and bottom surfaces of the airfoil. The region of the shock can clearly be seen at around an x/c location of 0.7. It can be noted that prior to the shock there is a region of very high velocity and lower pressure which forms the structure of the shock. The shock can also be seen in the enthalpy as an area of slightly higher enthalpy around the shock.

The stagnation point at the leading edge of the airfoil can clearly be seen in the velocity, mach, and pressure coefficients. The pressure coefficient at the leading edge of the airfoil is 1, which is consistent with a stagnation point. The units of velocity, pressure, enthalpy, entropy and density are ms^{-1}, Pa, Jkg^{-1}, $kJkg^{-1}K^{-1}$ and $kg\ m^{-3}$ respectively. Note that this applies to all the plots.

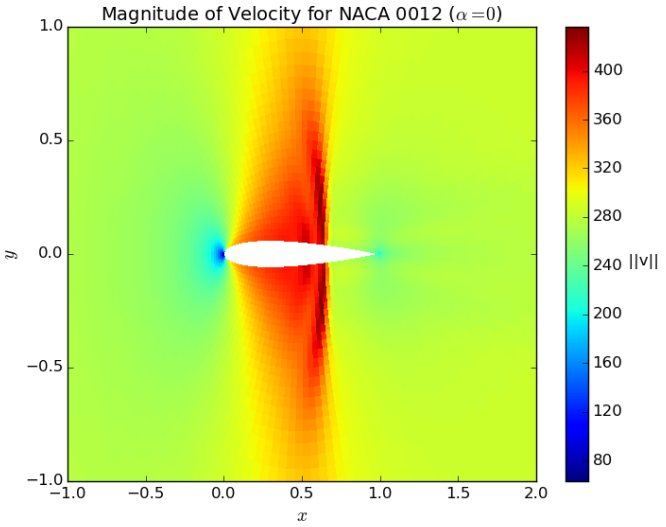

Figure 3: Speed around a NACA 0012 airfoil for 0° angle of attack for Euler 2D Equations.

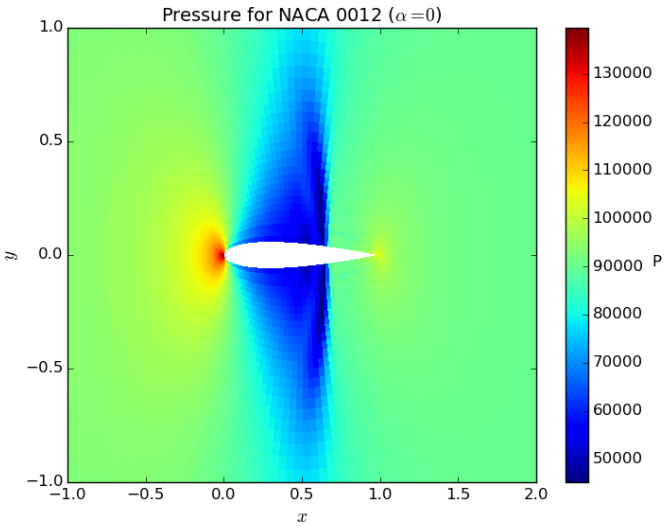

Figure 4: Pressure around a NACA 0012 airfoil for 0° angle of attack for Euler 2D Equations.

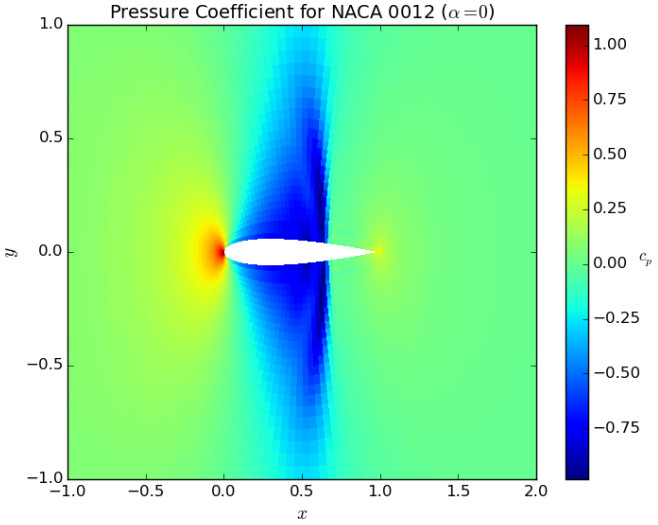

Figure 5: Pressure Coefficient around a NACA 0012 airfoil for 0° angle of attack for Euler 2D Equations.

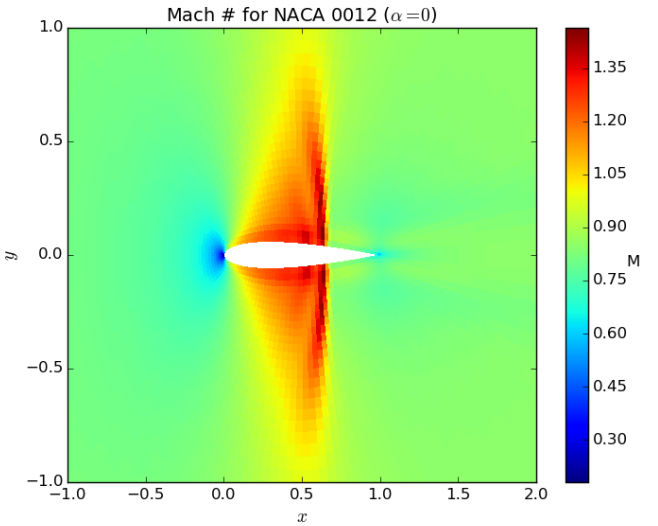

Figure 6: Mach # around a NACA 0012 airfoil for 0° angle of attack for Euler 2D Equations.

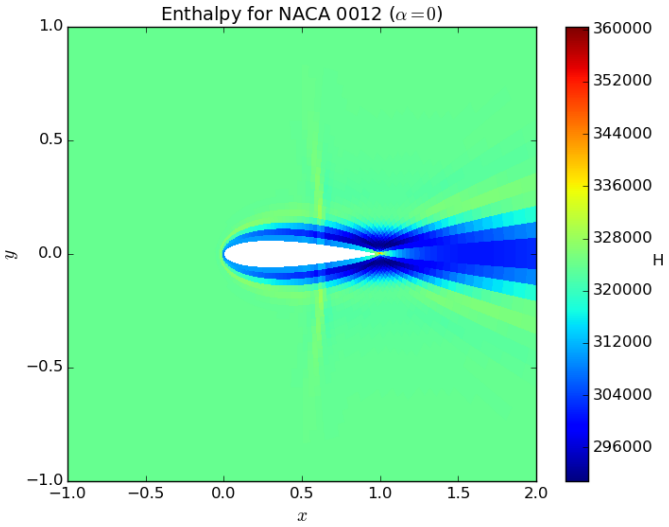

Figure 7: Enthalpy around a NACA 0012 airfoil for 0° angle of attack for Euler 2D Equations.

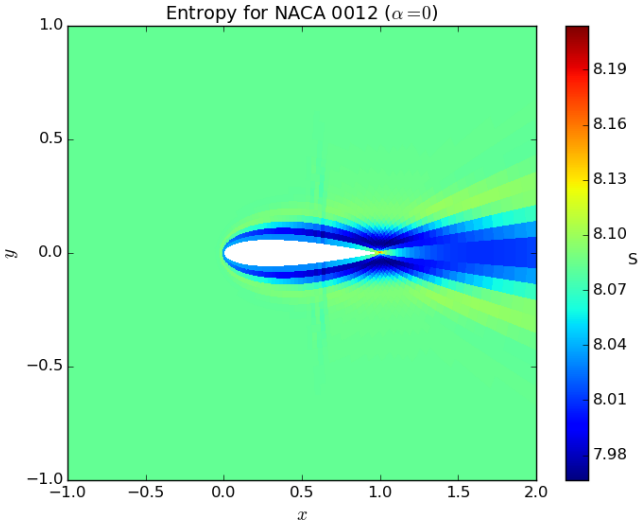

Figure 8: Entropy around a NACA 0012 airfoil for 0° angle of attack for Euler 2D Equations.

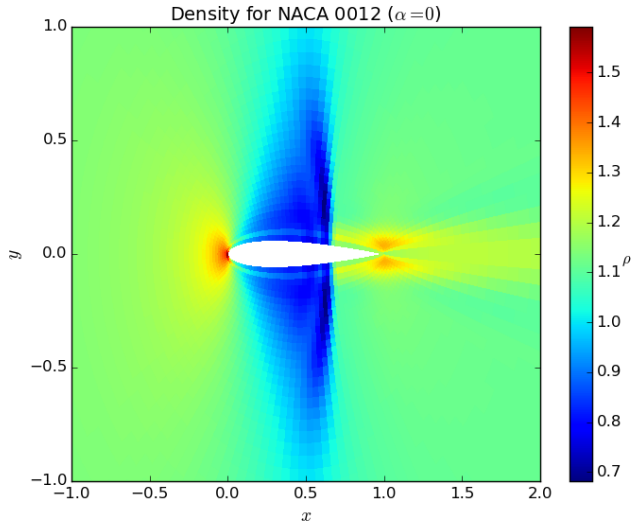

Figure 9: Density around a NACA 0012 airfoil for 0° angle of attack for Euler 2D Equations.

Plots of the pressure coefficient and the velocity on the airfoil surface are shown in Fig. 10 to 12. The shock location can clearly be seen where the pressure coefficient increases, also the velocity drops over the shock. Some small oscillations still remain directly preceding the shock. The oscillations may be damped by using different coefficients for the artificial diffusivity terms. The pressure and x velocity on the top and bottom surfaces of the symmetric airfoil is the same for the zero angle of attack case. This means the airfoil generates no lift.

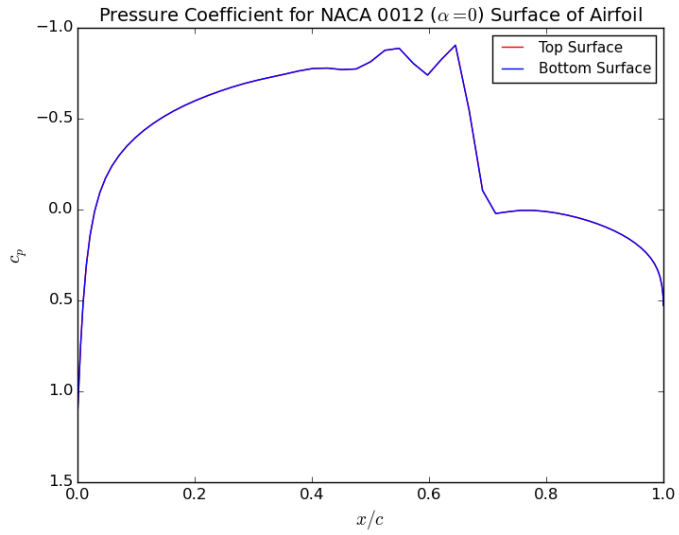

Figure 10: Pressure Coefficient on the surface of a NACA 0012 airfoil for 0° angle of attack for Euler 2D Equations.

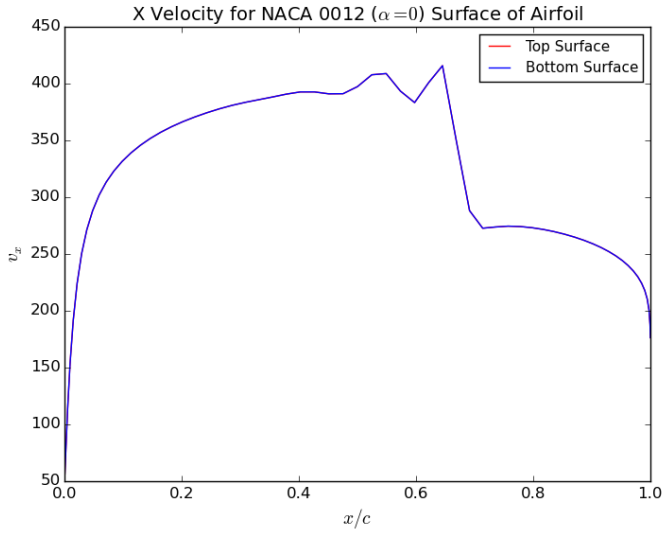

Figure 11: Velocity in X Direction on the surface of a NACA 0012 airfoil for 0° angle of attack for Euler 2D Equations.

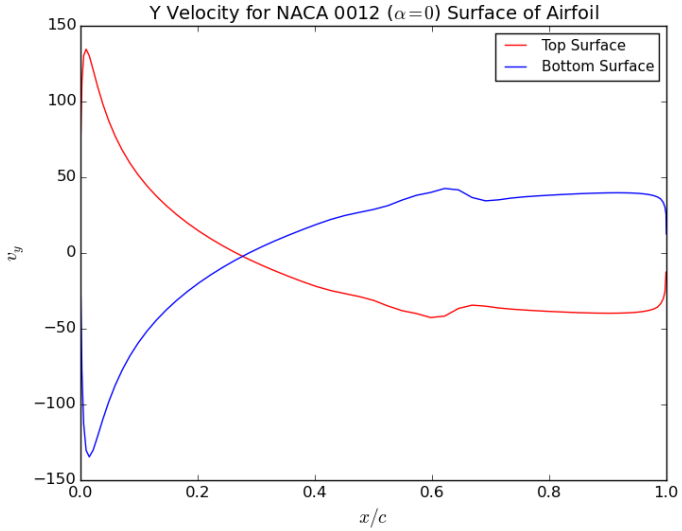

Figure 12: Velocity in Y Direction on the surface of a NACA 0012 airfoil for 0° angle of attack for Euler 2D Equations.

3.2 Flow around a NACA 0012 Airfoil at an Angle of Attack of 2 Degrees

The angle of attack changed the location of the shocks on the top and bottom surfaces. Also the airfoil now as a pressure differential. The shock locations on the top and bottom surfaces can be seen from Fig. 20. The shock on the upper surface occurs at an x/c location of around 0.8 which is shifted back toward the trailing edge slightly from the zero degree angle of attack case. The bottom surface the shock has moved forward and occurs at an x/c location of around 0.5. The intensity of the shocks have also changed, the shock on the top surfaces is much stronger and more well defined which the bottom shock is weaker and

more diffused. Though it should be noted some of the diffusivity in the shocks may be due to the artificial diffusivity introduced into the solution.

The pressure on the bottom surface of the airfoil is much higher then on the top surface as seen in Fig. 14 and 15. This pressure difference is what causes the airfoil to generate lift. Additionally it can be noted the speed on the top surface of the airfoil is much greater than on the bottom surface.

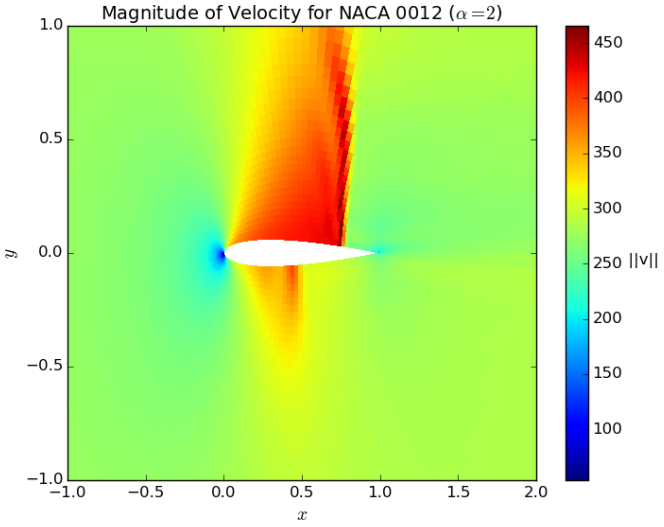

Figure 13: Speed around a NACA 0012 airfoil for 2° angle of attack for Euler 2D Equations.

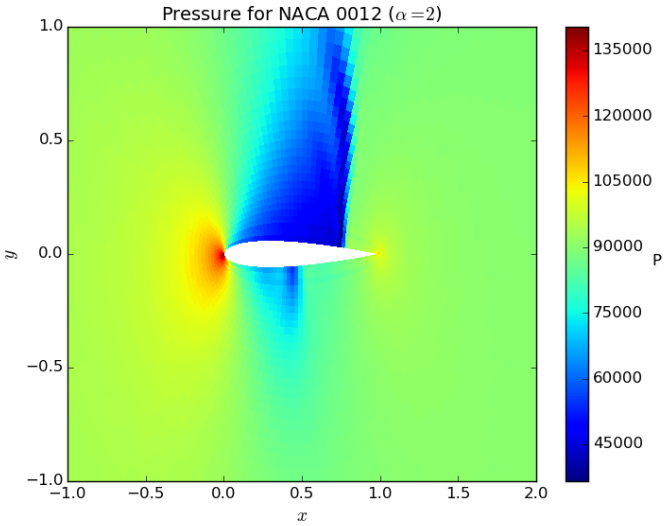

Figure 14: Pressure around a NACA 0012 airfoil for 2° angle of attack for Euler 2D Equations.

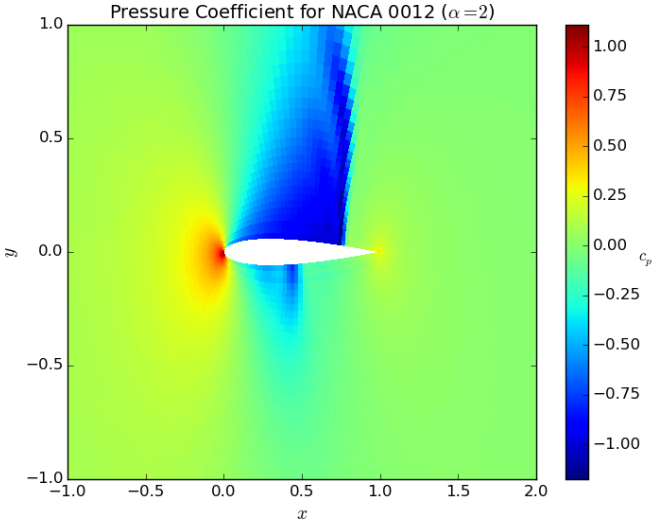

Figure 15: Pressure Coefficient around a NACA 0012 airfoil for 2° angle of attack for Euler 2D Equations.

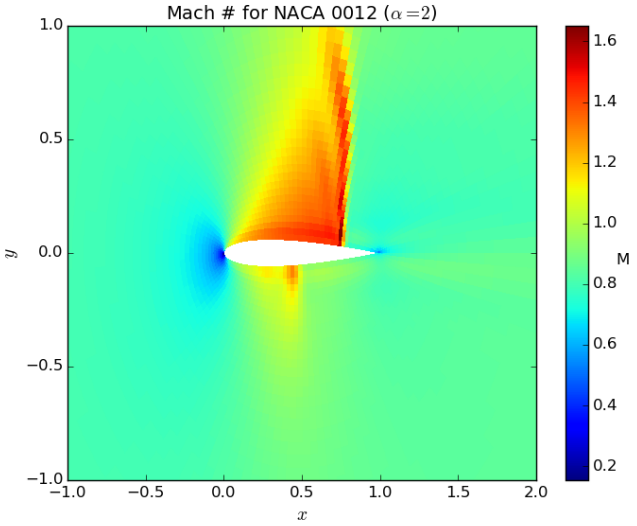

Figure 16: Mach # around a NACA 0012 airfoil for 2° angle of attack for Euler 2D Equations.

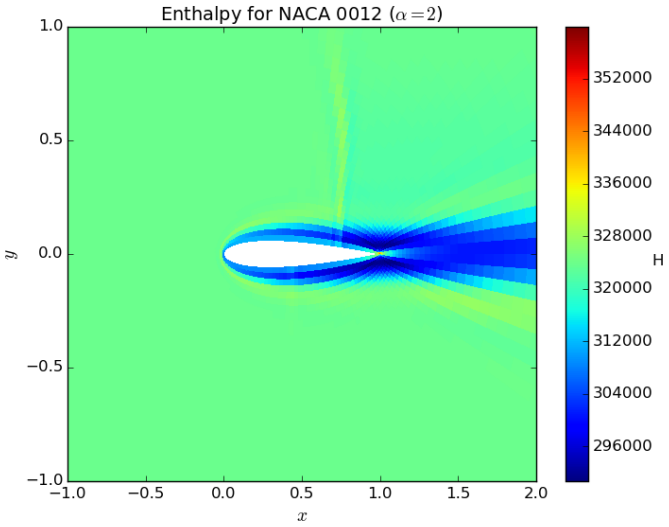

Figure 17: Enthalpy around a NACA 0012 airfoil for 2° angle of attack for Euler 2D Equations.

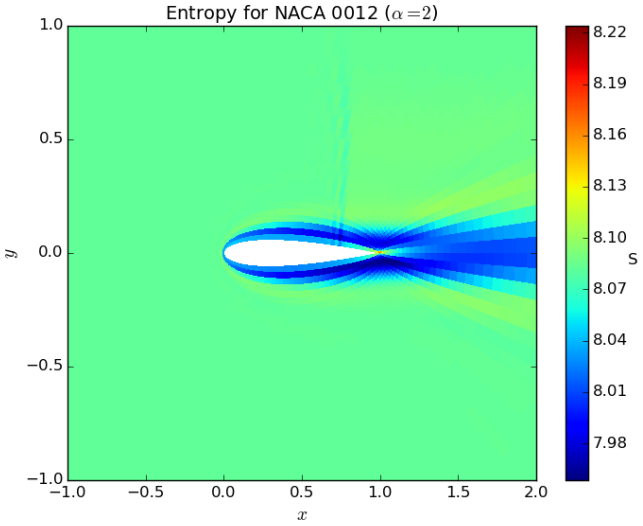

Figure 18: Entropy around a NACA 0012 airfoil for 2° angle of attack for Euler 2D Equations.

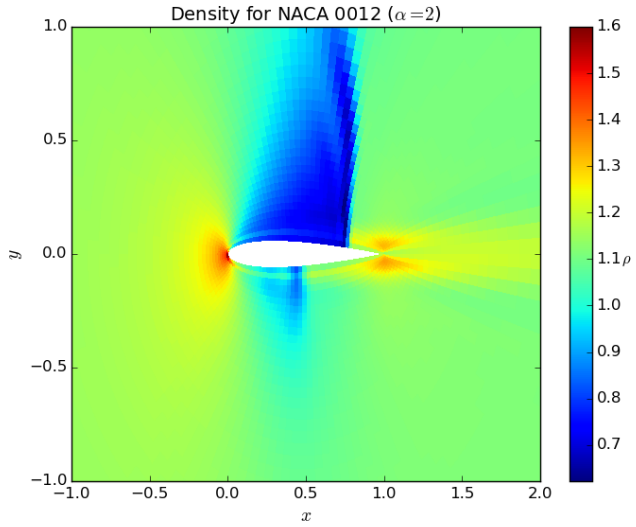

Figure 19: Density around a NACA 0012 airfoil for 2° angle of attack for Euler 2D Equations.

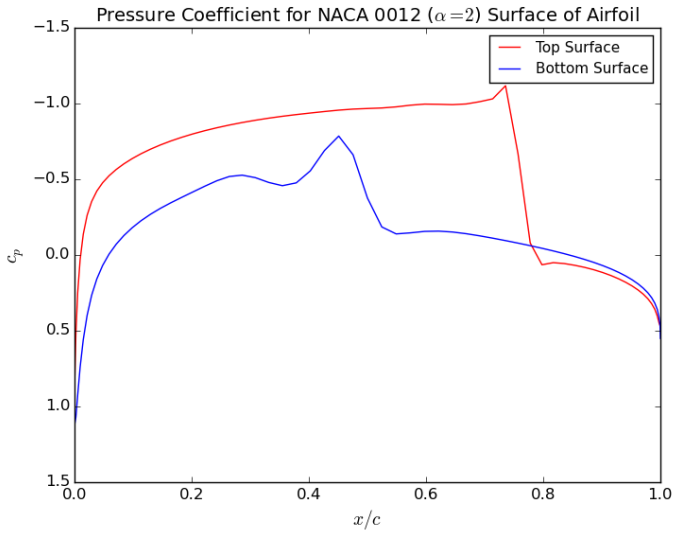

Figure 20: Pressure Coefficient on the surface of a NACA 0012 airfoil for 2° angle of attack for Euler 2D Equations.

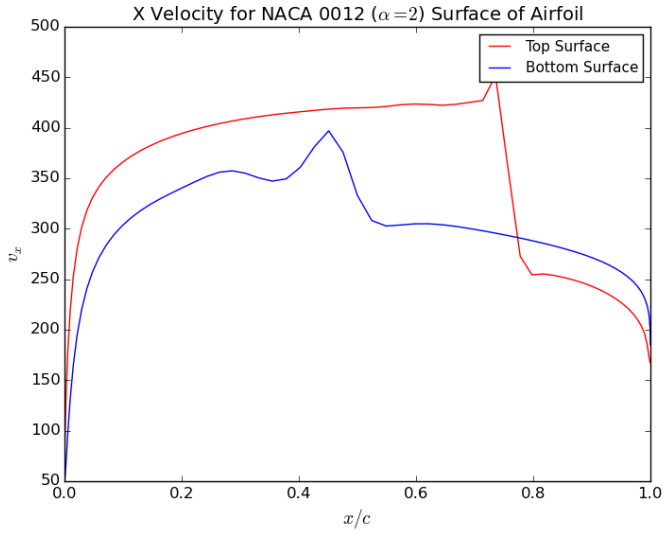

Figure 21: Velocity in X Direction on the surface of a NACA 0012 airfoil for 2° angle of attack for Euler 2D Equations.

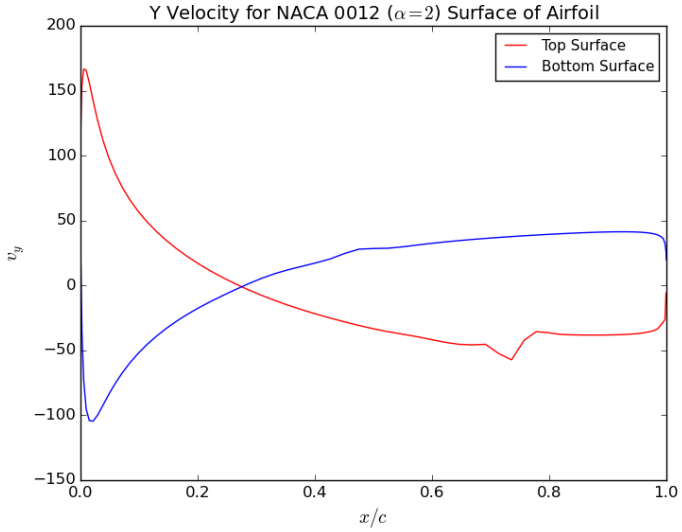

Figure 22: Velocity in Y Direction on the surface of a NACA 0012 airfoil for 2° angle of attack for Euler 2D Equations.

3.3 Flow around a NACA 0012 Airfoil at an Angle of Attack of 0 Degrees using the Finer Grid

Transonic Flow around the NACA 0012 airfoil was also investigated using a finer grid with 257×129 grid points. The results obtained using the finer grid show a marked improvement over the coarse one. The wiggles or non-physical oscillations around the shock seem to have reduced as well. For this run, the CFL# was reduced to 1.0.

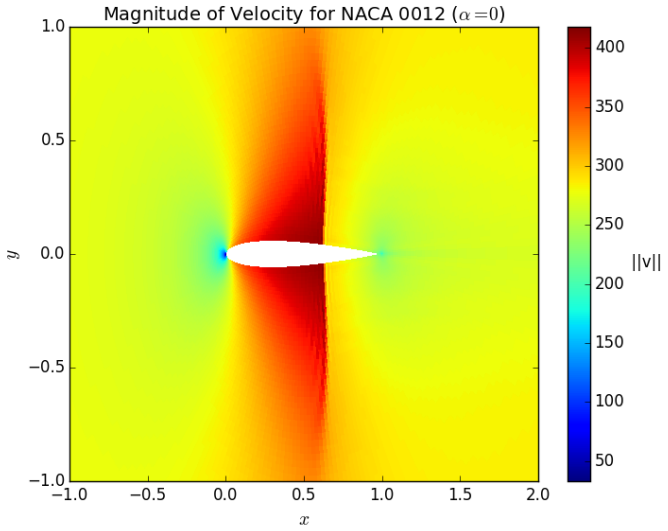

Figure 23: Speed around a NACA 0012 airfoil for 0° angle of attack for Euler 2D Equations on the finer grid.

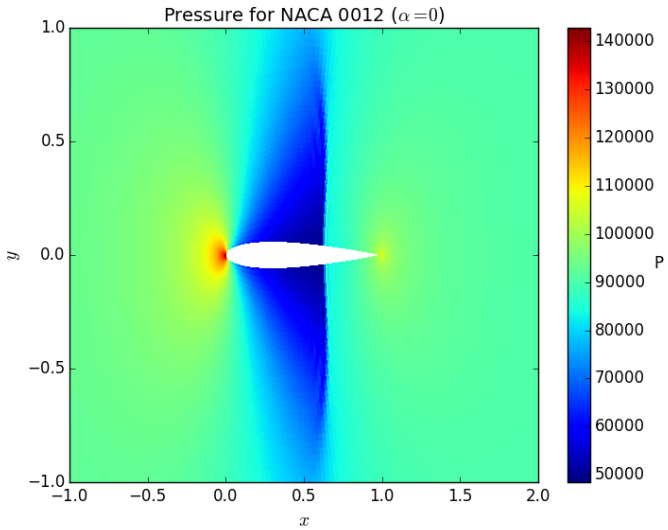

Figure 24: Pressure around a NACA 0012 airfoil for 0° angle of attack for Euler 2D Equations on the finer grid.

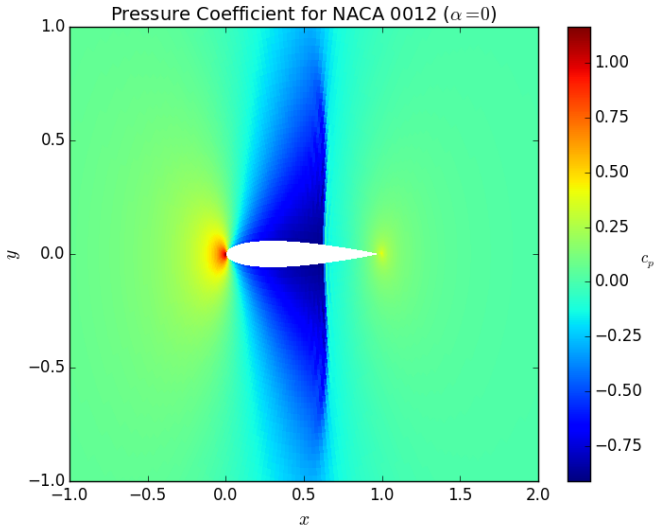

Figure 25: Pressure Coefficient around a NACA 0012 airfoil for 0° angle of attack for Euler 2D Equations on the finer grid.

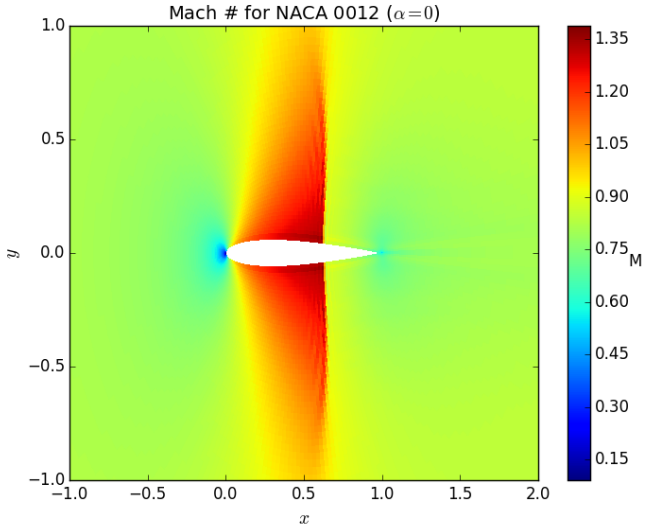

Figure 26: Mach # around a NACA 0012 airfoil for 0° angle of attack for Euler 2D Equations on the finer grid on the finer grid.

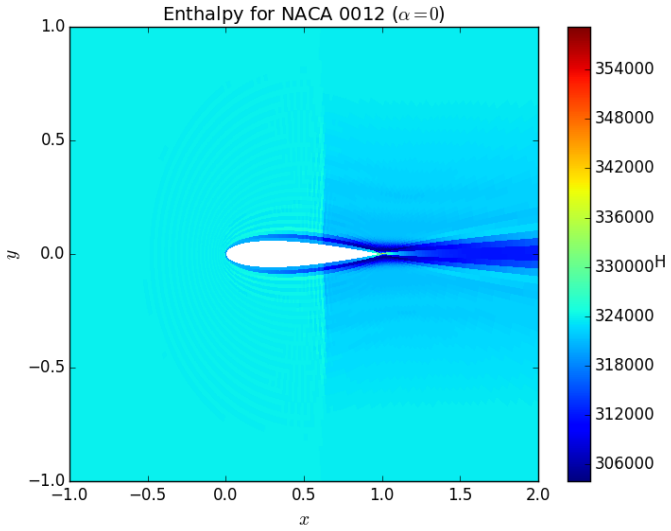

Figure 27: Enthalpy around a NACA 0012 airfoil for 0° angle of attack for Euler 2D Equations on the finer grid.

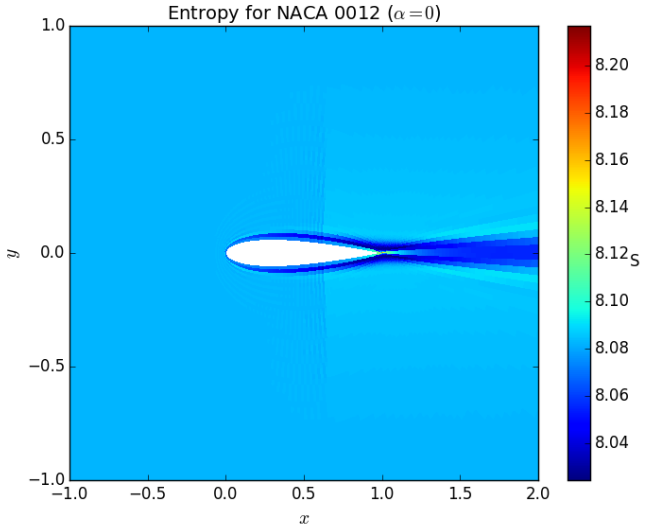

Figure 28: Entropy around a NACA 0012 airfoil for 0° angle of attack for Euler 2D Equations on the finer grid on the finer grid.

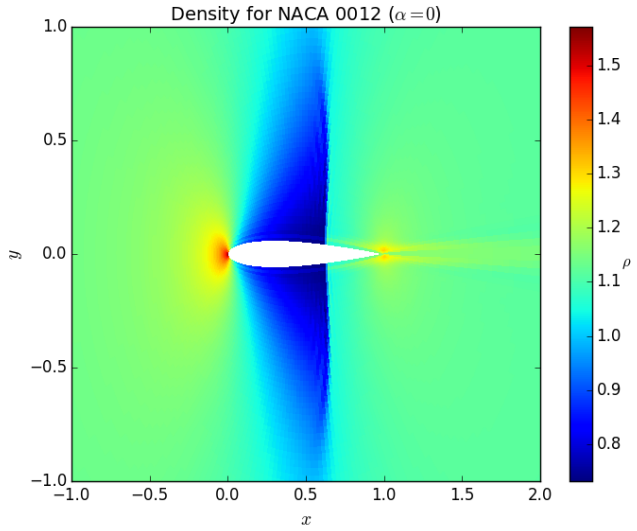

Figure 29: Density around a NACA 0012 airfoil for 0° angle of attack for Euler 2D Equations on the finer grid.

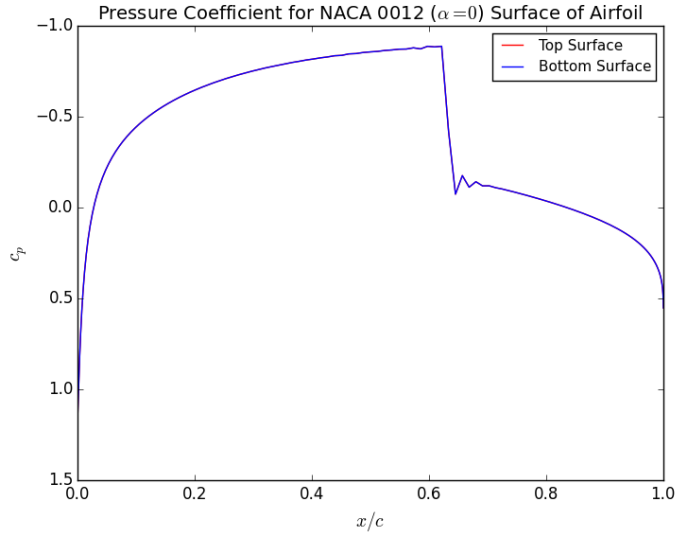

Figure 30: Pressure Coefficient on the surface of a NACA 0012 airfoil for 0° angle of attack for Euler 2D Equations on the finer grid.

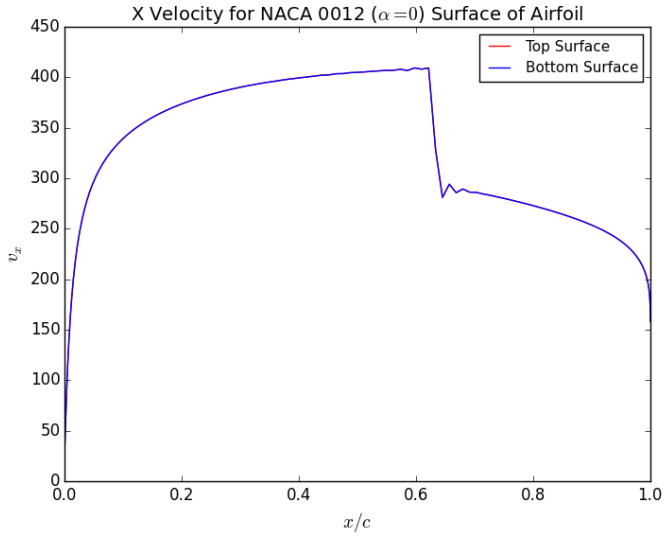

Figure 31: Velocity in X Direction on the surface of a NACA 0012 airfoil for 0° angle of attack for Euler 2D Equations on the finer grid.

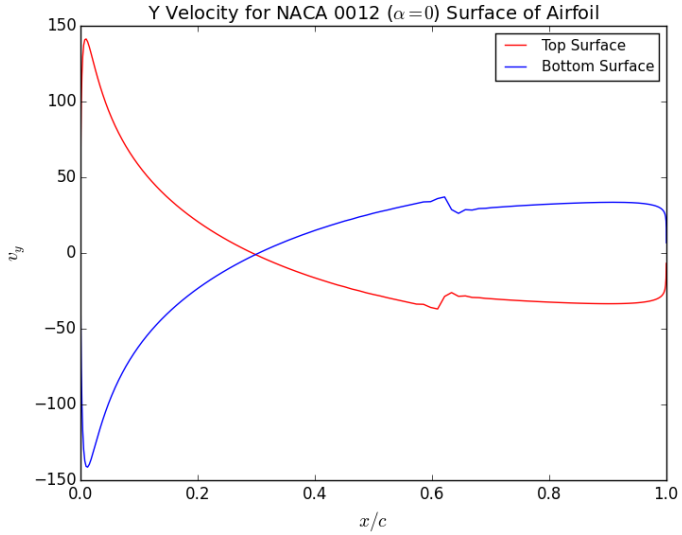

Figure 32: Velocity in Y Direction on the surface of a NACA 0012 airfoil for 0° angle of attack for Euler 2D Equations on the finer grid.

3.4 Flow around a NACA 0012 Airfoil at an Angle of Attack of 2 Degrees using the Finer Grid

The finer grid was also used to study transonic flow around the airfoil at an angle of attack of 2°. The results obtained using the finer grid show a marked improvement over the coarse one. The wiggles or non-physical oscillations around the shock seem to have reduced as well. For this run, the CFL# was reduced to 1.0.

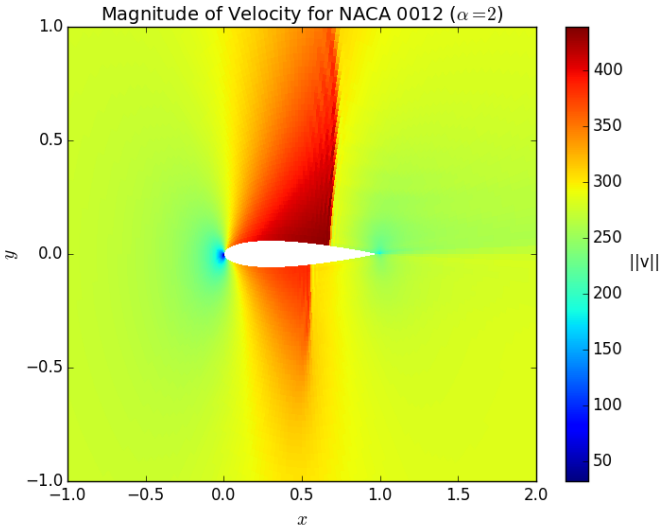

Figure 33: Speed around a NACA 0012 airfoil for 0° angle of attack for Euler 2D Equations on the finer grid.

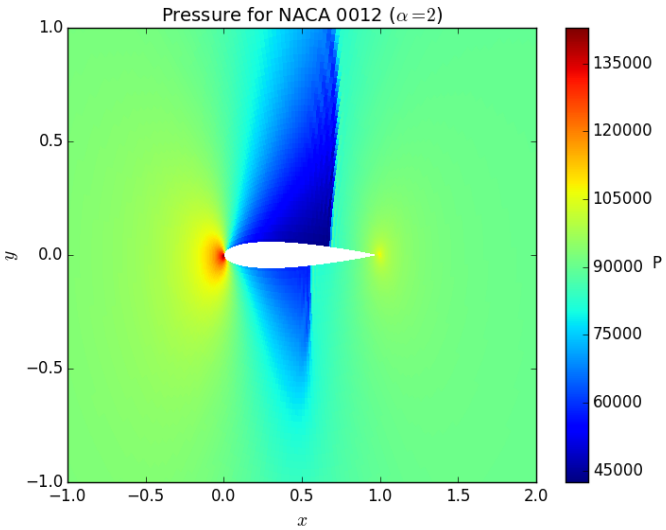

Figure 34: Pressure around a NACA 0012 airfoil for 0° angle of attack for Euler 2D Equations on the finer grid.

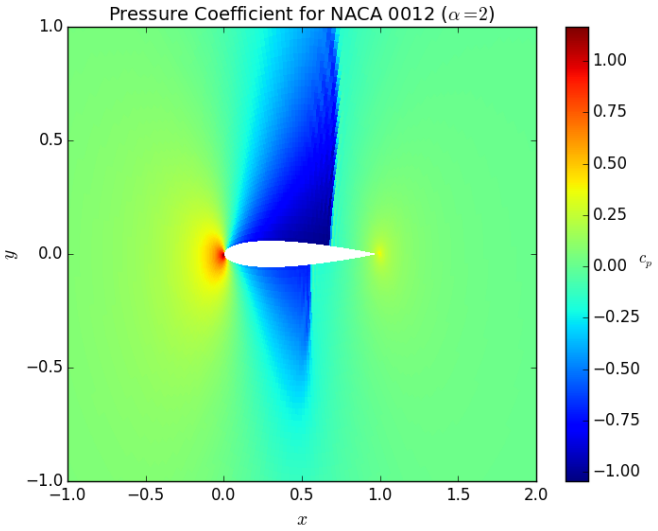

Figure 35: Pressure Coefficient around a NACA 0012 airfoil for 0° angle of attack for Euler 2D Equations on the finer grid.

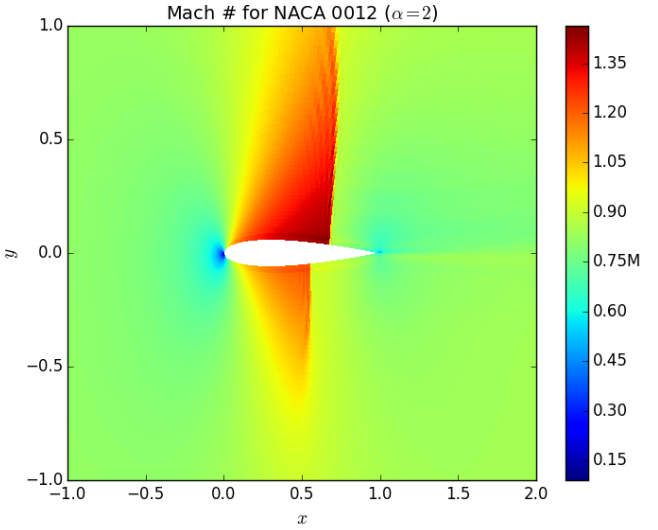

Figure 36: Mach # around a NACA 0012 airfoil for 0° angle of attack for Euler 2D Equations on the finer grid on the finer grid.

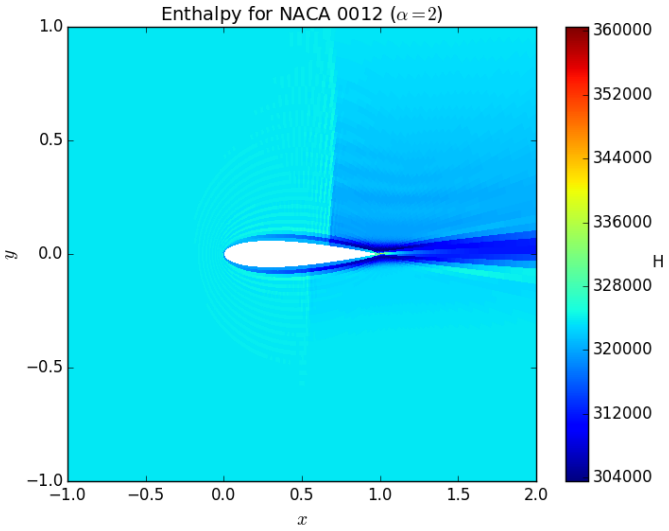

Figure 37: Enthalpy around a NACA 0012 airfoil for 0° angle of attack for Euler 2D Equations on the finer grid.

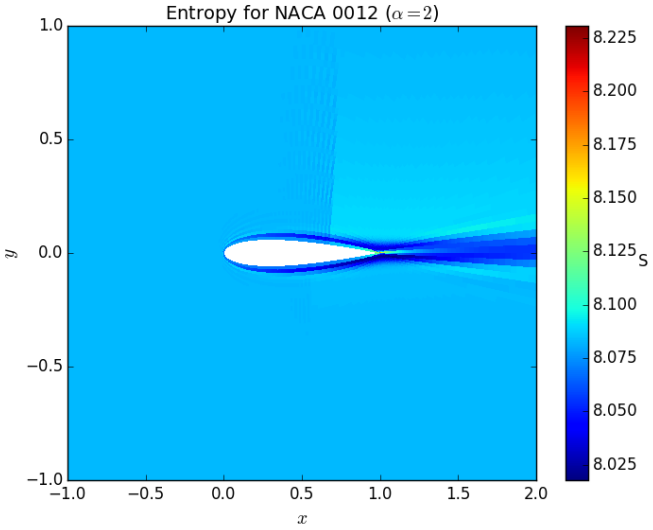

Figure 38: Entropy around a NACA 0012 airfoil for 0° angle of attack for Euler 2D Equations on the finer grid on the finer grid.

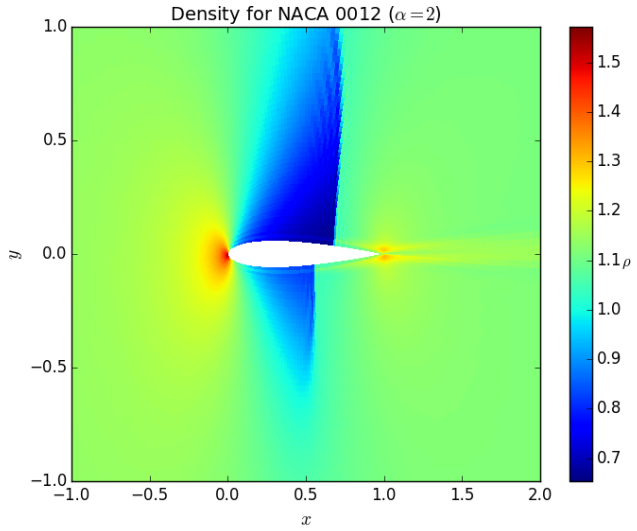

Figure 39: Density around a NACA 0012 airfoil for 0° angle of attack for Euler 2D Equations on the finer grid.

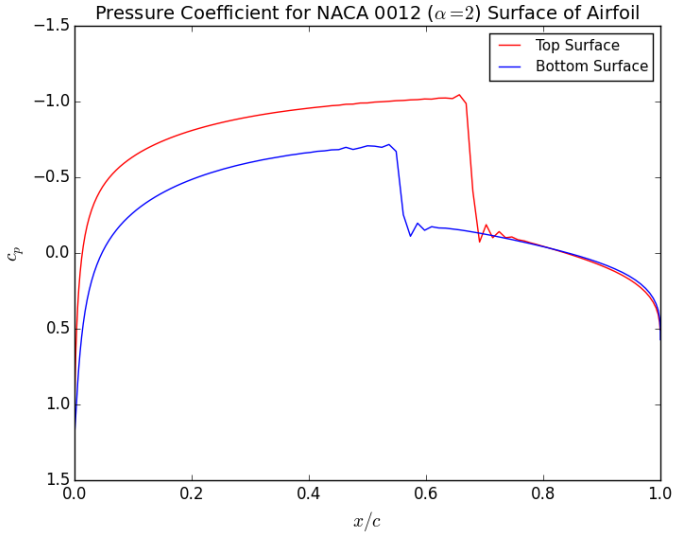

Figure 40: Pressure Coefficient on the surface of a NACA 0012 airfoil for 0° angle of attack for Euler 2D Equations on the finer grid.

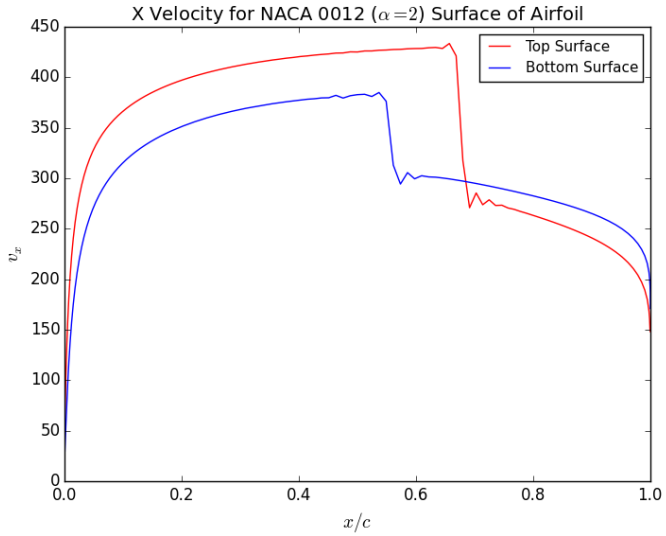

Figure 41: Velocity in X Direction on the surface of a NACA 0012 airfoil for 0° angle of attack for Euler 2D Equations on the finer grid.

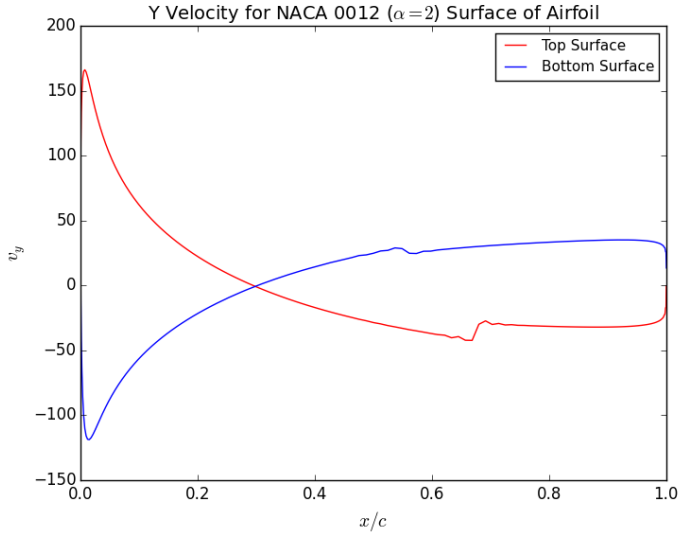

Figure 42: Velocity in Y Direction on the surface of a NACA 0012 airfoil for 0° angle of attack for Euler 2D Equations on the finer grid.

4 Conclusions and Future Work

Transonic flows are inherently complex due to the fact that the system of governing equations is a hybrid mix of elliptic-parabolic. The system of equations are hyperbolic for supersonic flows and parabolic at $M = 1$. The Jameson method [1] can be used to study transonic flows around immersed objects, but the stability of the scheme is an issue. Obtaining the proper values for various coefficients such as CFL#, Runge-Kutta constants and artificial viscosity parameters to ensure stability, accuracy and convergence of the scheme is

tricky. However, meaningful results to reasonable accuracy of such flows can be obtained using this method. Future work could involve conducting a grid independence study.

References

[1] A. Jameson, W. Schmidt, and E. Turkel. Numerical solutions of the euler equations by finite volume methods using runge-kutta time-stepping schemes. *AIAA Paper 81-1259*, 1981.

Appendices

A Two Dimensional Euler

```
program euler2D
    implicit none

    integer :: n, i, j, tmpi, tmpj
!    integer, parameter :: ni = 5-1 !ni = 16 !128 !Number of grid points in x
!    integer, parameter :: nj = 4 !nj = ni !Number of grid points in y
    integer, parameter :: ni = 257-1 !ni = 16 !128 !Number of grid points in x
    integer, parameter :: nj = 129 !nj = ni !Number of grid points in y
!    integer, parameter :: ni = 16-1 !ni = 16 !128 !Number of grid points in x
!    integer, parameter :: nj = 9 !nj = ni !Number of grid points in y
    real, parameter :: gam = 1.4
    real, parameter :: pi = 4.*atan(1.)
    !real, parameter :: cfl = 2.7
    real, dimension(ni,nj+1) :: x, y, x_cell, y_cell, sigma
    real, dimension(2,ni,nj) :: n1, n2, n3, n4 !Cell normals
    real,dimension(ni,nj) :: tau !local timestep
    real, dimension(4,ni,nj) :: U, F, G, Risd, U1, U2, U3, U_next, &
                                Risd_1, Risd_2, Risd_3
    !real, dimension(4,ni,nj) :: Jac, Jinv, Jinv_det
    real :: alt_inf, vx_inf, vy_inf, p_inf, T_inf, rho_inf, M_inf, c_inf, &
            E_inf, H_inf, alpha !Initial Condition Variables  , S_inf

!    real :: pres_func, rho_func, c_func
!    real :: vx, vy !Output variables

    !Runge Kutta alpha values
    real, parameter :: alpha1 = 1./8., alpha2 = 0.306, alpha3 = 0.587, alpha4
       = 1.
    real, parameter :: dt = 0.0001

    !Set Initial Conditions
    alpha = 2.*(pi/180.)
    !alpha = 0.
    write(*,*) pi, alpha
    M_inf = 0.85

    !Set US Standard Atmospheric Conditions
```

```
    ! http://www.engineeringtoolbox.com/standard-atmosphere-d_604.html
    alt_inf = 1000. ! 1000m above sea level
    p_inf = 101325.*(1. - (2.25577e-5)*alt_inf)**5.25588 !Pressure in Pa
    T_inf = 8.50 + 273.15 !Temperature in K
    rho_inf = 11.12e-1 !Density in kg/m^3

!     p_inf = 1.
 !    rho_inf = 1.

    !Velocities are in m/s
    c_inf = (gam*p_inf/rho_inf)**0.5
    vx_inf = M_inf*c_inf*cos(alpha)
    vy_inf = M_inf*c_inf*sin(alpha)

    E_inf = (1./(gam-1.))*(p_inf/rho_inf) + 0.5*(vx_inf**2 + vy_inf**2)
    H_inf = E_inf + p_inf/rho_inf

    !write(*,*) rho_inf*vx_inf, rho_inf*vx_inf*H_inf, rho_inf*vy_inf*H_inf
    !write(*,*) p_inf/(rho_inf*T_inf)

    U(1,:,:) = rho_inf
    U(2,:,:) = rho_inf*vx_inf
    U(3,:,:) = rho_inf*vy_inf
    U(4,:,:) = rho_inf*E_inf


    !Read in the Grid Data :
    open(unit = 2, file = "coords.dat");
    x=0.
    y=0.
    do i=1,ni
    do j=1,nj
        read(unit=2, fmt=*) tmpi, tmpj, x(i,j), y(i,j)
!         write(*,*) i, j, x(i,j), y(i,j)
    enddo
    enddo

    !Calculate the cell volumes
    call cell_volumes(ni,nj,x,y,x_cell,y_cell,sigma,n1,n2,n3,n4)
    call timestep(ni,nj,n1,n2,n3,n4,U,tau)
    !Assign BC's at inlet and outlet
!     call inlet_outlet_bcs(ni,nj,n4,vx_inf,vy_inf,p_inf,rho_inf,c_inf,U)
!     call timestep(ni,nj,cfl,n1,n2,n3,n4,U,tau)
    open(unit = 4, file = "res2alpha.dat");
    do n = 1,20000

!         write(*,*) n
        !Step 1 - Runge-Kutta
!         call timestep(ni,nj,cfl,n1,n2,n3,n4,U,tau)
        call inlet_outlet_bcs(ni,nj,n4,vx_inf,vy_inf,p_inf,rho_inf,c_inf,U) !
           Enforce BC's
        call fluxes(ni,nj,U,F,G)
        call timestep(ni,nj,n1,n2,n3,n4,U,tau)
        call risdual(ni,nj,U,U,F,G,n1,n2,n3,n4,dt,sigma,alpha1,Risd,U1,x_cell,
```

```
     y_cell,tau)


     !Step 2 - Runge-Kutta
     call inlet_outlet_bcs(ni,nj,n4,vx_inf,vy_inf,p_inf,rho_inf,c_inf,U1) !
        Enforce BC's
     call fluxes(ni,nj,U1,F,G)
!     call timestep(ni,nj,n1,n2,n3,n4,U,tau)
     call risdual(ni,nj,U,U1,F,G,n1,n2,n3,n4,dt,sigma,alpha2,Risd_1,U2,
        x_cell,y_cell,tau)


!     if (n .eq. 1) then
!         call output_velocities(ni,nj,x_cell,y_cell,U2,F,G)
!     endif

     !Step 3 - Runge-Kutta
     call inlet_outlet_bcs(ni,nj,n4,vx_inf,vy_inf,p_inf,rho_inf,c_inf,U2) !
        Enforce BC's
     call fluxes(ni,nj,U2,F,G)
!     call timestep(ni,nj,n1,n2,n3,n4,U,tau)
     call risdual(ni,nj,U,U2,F,G,n1,n2,n3,n4,dt,sigma,alpha3,Risd_2,U3,
        x_cell,y_cell,tau)


     !Step 4 - Runge-Kutta
     call inlet_outlet_bcs(ni,nj,n4,vx_inf,vy_inf,p_inf,rho_inf,c_inf,U3) !
        Enforce BC's
     call fluxes(ni,nj,U3,F,G)
!     call timestep(ni,nj,n1,n2,n3,n4,U,tau)
     call risdual(ni,nj,U,U3,F,G,n1,n2,n3,n4,dt,sigma,alpha4,Risd_3,U_next,
        x_cell,y_cell,tau)



     !Output the velocity magnitude and direction
     if (n .eq. 20000) then
         call output_velocities(ni,nj,x_cell,y_cell,U,F,G,n4,p_inf,rho_inf,
            vx_inf,vy_inf)
!     write(*,*) tau
     endif
     if (maxval(abs(Risd_3(1,:,:))) .le. 10.**-5 .or. &
         maxval(abs(Risd_3(2,:,:))) .le. 10.**-5 .or. &
         maxval(abs(Risd_3(3,:,:))) .le. 10.**-5 .or. &
         maxval(abs(Risd_3(4,:,:))) .le. 10.**-5) then

     write(*,*) n, maxval(Risd_3(1,:,:)), maxval(Risd_3(2,:,:)), maxval(
        Risd_3(3,:,:)) &
                , maxval(Risd_3(4,:,:))
!     write(*,*) "Min res", minval(Risd_3)
     endif
     U = U_next
     write(4,*) n, maxval(Risd_3(1,:,:)), maxval(Risd_3(2,:,:)), maxval(
        Risd_3(3,:,:)) &
```

```
                    , maxval(Risd_3(4,:,:))
    enddo


end program euler2D


subroutine cell_volumes(ni, nj, x, y, x_cell, y_cell, sigma, n1, n2, n3, n4)
    implicit none
    integer :: i, j, ip, im
    integer :: ni, nj
    real, dimension(ni,nj) :: x, y, sigma, x_cell, y_cell
    real, dimension(2,ni,nj) :: n1, n2, n3, n4 !Cell normals

    x_cell=0.
    y_cell=0.
    sigma=0.
    n1 = 0.
    n2 = 0.
    n3 = 0.
    n4 = 0.
    do i = 1,ni
    do j = 1,nj
        ip = i+1
        im = i-1
        if (i .eq. 1) then
            im = ni
        else if (i .eq. ni) then
            ip = 1
        endif

        if (j .lt. nj) then
        !Assume x(i,j) is point D (lower left cell point)
        !The i,j of the cell is actually at the cell center (coordinates (
           x_cell,y_cell))
        !Number of cells: (ni-1,nj), includes 'ghost cells' (i,nj)
        x_cell(i,j) = 1./4.*(x(i,j) + x(ip,j) + x(i,j+1) + x(ip,j+1))
        y_cell(i,j) = 1./4.*(y(i,j) + y(ip,j) + y(i,j+1) + y(ip,j+1))
        sigma(i,j) = 0.5*abs((x(ip,j) - x(i,j+1))*(y(i,j) - y(ip,j+1)) &
                        - (x(i,j) - x(ip,j+1))*(y(ip,j) - y(i,j+1)))
        !write(*,*) i, j, x_cell(i,j), y_cell(i,j), sigma(i,j)
        !write(*,*) x(ip,j), x(i,j+1), y(i,j), y(ip,j+1),
        ! x(i,j), y(i,j),


        !Calculate the normals
        n1(1,i,j) =   y(ip,j+1) - y(ip,j)  !ip/2,j Side
        n1(2,i,j) = -(x(ip,j+1) - x(ip,j))
        n2(1,i,j) =   y(i,j+1) - y(ip,j+1)  !i,j+1/2 Side
        n2(2,i,j) = -(x(i,j+1) - x(ip,j+1))
        n3(1,i,j) =   y(i,j) - y(i,j+1)      !im/2,j Side
        n3(2,i,j) = -(x(i,j) - x(i,j+1))
        endif
```

```
        n4(1,i,j) =   y(ip,j) - y(i,j)       !i,j-1/2 Side
        n4(2,i,j) = -(x(ip,j) - x(i,j))

!        if (j .eq. nj-1) then
!        write(*,*) "FOR CELL", x_cell(i,j), y_cell(i,j), i,j, sigma(i,j)
!        write(*,*) "N(i+1/2,j):", x(ip,j+1), y(ip,j+1), x(ip,j), y(ip,j), n1
   (1,i,j), n1(2,i,j)
!        write(*,*) "N(i,j+1/2):", x(i,j+1), y(i,j+1), x(ip,j+1), y(ip,j+1),
   n2(1,i,j), n2(2,i,j)
!        write(*,*) "N(i-1/2,j):", x(i,j), y(i,j), x(i,j+1), y(i,j+1), n3(1,i,
   j), n3(2,i,j)
!        write(*,*) "N(i,j-1/2):", x(ip,j), y(ip,j), x(i,j), y(i,j), n4(1,i,j)
   , n4(2,i,j)
!        endif
    enddo
    enddo
!    write(*,*) 'n1',n1
!    write(*,*) 'n2',n2
!    write(*,*) 'n3',n3
!    write(*,*) 'n4',n4
end subroutine cell_volumes


subroutine fluxes(ni, nj, U, F, G)
    implicit none
    integer :: i, j, ni, nj
    real, parameter :: gam = 1.4
    real, dimension(4,ni,nj) :: U, F, G

!    write(*,*) "FLUXES CALLED"
    do i = 1,ni
    do j = 1,nj

        !Assign F Flux Vector
        F(1,i,j) = U(2,i,j)
        F(2,i,j) = (U(2,i,j)**2/U(1,i,j)) + (gam-1.)*(U(4,i,j) &
                        - 0.5/(U(1,i,j))*(U(2,i,j)**2 + U(3,i,j)**2))
        F(3,i,j) = U(2,i,j)*U(3,i,j)/U(1,i,j)
        F(4,i,j) = U(2,i,j)*(gam*U(4,i,j)/U(1,i,j) &
                    - (gam-1.)/2./(U(1,i,j)**2)*(U(2,i,j)**2 + U(3,i,j)**2))

        !Assign G Flux Vector
        G(1,i,j) = U(3,i,j)
        G(2,i,j) = U(2,i,j)*U(3,i,j)/U(1,i,j)
        G(3,i,j) = (U(3,i,j)**2/U(1,i,j)) + (gam-1.)*(U(4,i,j) &
                        - 1./(2.*U(1,i,j))*(U(2,i,j)**2 + U(3,i,j)**2))
        G(4,i,j) = U(3,i,j)*(gam*U(4,i,j)/U(1,i,j) &
                    - (gam-1.)/2./(U(1,i,j)**2)*(U(2,i,j)**2 + U(3,i,j)**2))


!        write(*,*) i,j, U(:,i,j), F(:,i,j), G(:,i,j)
```

```
    enddo
    enddo

end subroutine


subroutine risdual(ni,nj,U,U_cur,F,G,n1,n2,n3,n4,dt,sigma,alpha,Risd,U_next,
   x_cell,y_cell,tau)
    implicit none

    integer :: i, j, ip, im, ni, nj, ip2, im2
    real, dimension(4) :: Fdotn_bc
    real, dimension(ni,nj) ::sigma, x_cell, y_cell
    real, dimension(4,ni,nj) :: U, U_cur, F, G, Risd, U_next
    real, dimension(2,ni,nj) :: n1, n2, n3, n4 !Cell normals
    real :: alpha, dt, pres_func, vx_func, vy_func, c_func, &
            u_plus_c_ds_func
    real, dimension(ni,nj) :: tau
    !real, dimension(ni) :: nui
    !real, dimension(nj) :: nuj
    real, dimension(4) :: u_plus_c_ds, eps2_ds, eps4_ds
    real, dimension(4,4) :: D_ds
    real :: alpha_diff2, alpha_diff4
    real, dimension(ni,nj) :: nui
    real, dimension(ni,nj+1) :: nuj

    Risd = 0.
    alpha_diff2 = 1./1.5
    alpha_diff4 = 1./50.

    U_next = U_cur
    D_ds = 0.

    call nui_diff(ni,nj,U,nui)
    call nuj_diff(ni,nj,U,nuj)

    !write(*,*) "RISIDUAL CALLED"
    do i = 1,ni
    do j = 1,nj-1 !Don't include ghost cell

!         write(*,*) "FOR CELL", x_cell(i,j), y_cell(i,j), i,j

         if (isnan(U(1,i,j))) then
             write(*,*) "U1 is nan",i,j
             stop
         endif

         if (isnan(U(2,i,j))) then
             write(*,*) "U2 is nan",i,j
             stop
         endif

         if (isnan(U(3,i,j))) then
             write(*,*) "U3 is nan",i,j
```

```
    stop
endif

if (isnan(U(4,i,j))) then
    write(*,*) "U4 is nan",i,j
    stop
endif

ip = i+1
im = i-1
ip2 = i+2
im2 = i-2
if (i .eq. 1) then
    im = ni
    im2 = ni-1
else if (i .eq. ni) then
    ip = 1
    ip2 = 2
else if (i .eq. 2) then
    im2 = ni
else if (i .eq. ni-1) then
    ip2 = 1
endif


!For each cell sum up the 4 flux contributions
!(Note the appropriate minus sign is already included in the normal n#
   y)
Risd(:,i,j) = 0.5*(F(:,ip,j) + F(:,i,j)) * n1(1,i,j) & !i+1/2,j Side
            + 0.5*(G(:,ip,j) + G(:,i,j)) * n1(2,i,j) &
            + 0.5*(F(:,i,j+1) + F(:,i,j)) * n2(1,i,j) & !i,j+1/2 Side
            + 0.5*(G(:,i,j+1) + G(:,i,j)) * n2(2,i,j) &
            + 0.5*(F(:,im,j) + F(:,i,j)) * n3(1,i,j) & !i-1/2,j Side
            + 0.5*(G(:,im,j) + G(:,i,j)) * n3(2,i,j)


!i+1/2 and i-1/2 sides, artificial diffusivity
u_plus_c_ds(1) = u_plus_c_ds_func(ni,nj,U,n1,i,ip,j,j)
u_plus_c_ds(3) = u_plus_c_ds_func(ni,nj,U,n3,i,im,j,j)

eps2_ds(1) = 0.5 * alpha_diff2 * u_plus_c_ds(1) &
        * max(nui(im,j),nui(i,j),nui(ip,j),nui(ip2,j))
eps2_ds(3) = 0.5 * alpha_diff2 * u_plus_c_ds(3) &
        * max(nui(im2,j),nui(im,j),nui(i,j),nui(ip,j))

eps4_ds(1) = max(0., 0.5*alpha_diff4*u_plus_c_ds(1)-eps2_ds(1))
eps4_ds(3) = max(0., 0.5*alpha_diff4*u_plus_c_ds(3)-eps2_ds(3))

D_ds(:,1) = eps2_ds(1)*(U(:,ip,j) - U(:,i,j)) &
    -eps4_ds(1)*(U(:,ip2,j)-3.*U(:,ip,j)+3.*U(:,i,j)-U(:,im,j))

D_ds(:,3) = eps2_ds(3)*(U(:,i,j) - U(:,im,j)) &
    -eps4_ds(3)*(U(:,ip,j)-3.*U(:,i,j)+3.*U(:,im,j)-U(:,im2,j))
```

```
        Risd(:,i,j) = Risd(:,i,j) - (D_ds(:,1) - D_ds(:,3))

        !Wall Boundary Condition
        !Set F.n = [0 p*nx p*ny 0] at j = 1 at edge (i,j-1/2) , i = 1,2, ... ,
            ni-1 (?)
        if (j.eq.1) then

            !May need to invert the components of normal
            Fdotn_bc(1) = 0.
            Fdotn_bc(2) = pres_func(U_cur(:,i,j))*n4(1,i,j)
            Fdotn_bc(3) = pres_func(U_cur(:,i,j))*n4(2,i,j)
            !Fdotn_bc(2) = 0.5*(pres_func(U_cur(:,i,j+1))+pres_func(U_cur(:,i,
                j)))*n4(1,i,j)
            !Fdotn_bc(3) = 0.5*(pres_func(U_cur(:,i,j+1))+pres_func(U_cur(:,i,
                j)))*n4(2,i,j)
            Fdotn_bc(4) = 0.
!           write(*,*) x_cell(i,j), y_cell(i,j), n4(1,i,j), n4(2,i,j)
            !write(*,*) x_cell(i,j), y_cell(i,j), pres_func(U_cur(:,i,j))

            Risd(:,i,j) = Risd(:,i,j) + Fdotn_bc(:) !i,j-1/2 Side

        else
            Risd(:,i,j) = Risd(:,i,j) &
                        + 0.5*(F(:,i,j-1) + F(:,i,j)) * n4(1,i,j) & !i,j-1/2
                            Side
                        + 0.5*(G(:,i,j-1) + G(:,i,j)) * n4(2,i,j)

            u_plus_c_ds(2) = u_plus_c_ds_func(ni,nj,U,n2,i,i,j,j+1)
            u_plus_c_ds(4) = u_plus_c_ds_func(ni,nj,U,n4,i,i,j,j-1)

            eps2_ds(2) = 0.5 * alpha_diff2 * u_plus_c_ds(2) &
                    * max(nuj(i,j-1),nuj(i,j),nuj(i,j+1),nuj(i,j+2))
            eps4_ds(2) = max(0., 0.5*alpha_diff4*u_plus_c_ds(2)-eps2_ds(2))

            if (j .eq. 2) then
                eps2_ds(4) = 0.5 * alpha_diff2 * u_plus_c_ds(4) &
                        * max(nuj(i,j-1),nuj(i,j),nuj(i,j+1))
                eps4_ds(4) = max(0., 0.5*alpha_diff4*u_plus_c_ds(4)-eps2_ds(4)
                    )
                D_ds(:,4) = eps2_ds(4)*(U(:,i,j) - U(:,i,j-1)) &
                    -eps4_ds(4)*(U(:,i,j+1)-3.*U(:,i,j)+3.*U(:,i,j-1))
            else
                eps2_ds(4) = 0.5 * alpha_diff2 * u_plus_c_ds(4) &
                        * max(nuj(i,j-2),nuj(i,j-1),nuj(i,j),nuj(i,j+1))
                eps4_ds(4) = max(0., 0.5*alpha_diff4*u_plus_c_ds(4)-eps2_ds(4)
                    )

                D_ds(:,4) = eps2_ds(4)*(U(:,i,j) - U(:,i,j-1)) &
                    -eps4_ds(4)*(U(:,i,j+1)-3.*U(:,i,j)+3.*U(:,i,j-1)-U(:,i,j
                        -2))
            endif


            if (j .eq. nj-1) then
```

```
                D_ds(:,2) = eps2_ds(2)*(U(:,i,j+1) - U(:,i,j))
            else
                D_ds(:,2) = eps2_ds(2)*(U(:,i,j+1) - U(:,i,j)) &
                    -eps4_ds(2)*(U(:,i,j+2)-3.*U(:,i,j+1)+3.*U(:,i,j)-U(:,i,j
                        -1))
            endif

            Risd(:,i,j) = Risd(:,i,j) - (D_ds(:,2) - D_ds(:,4))

        endif

        !Calculate the U value using given risdual
        !U_next(:,i,j) = U(:,i,j) - dt/sigma(i,j)*alpha*Risd(:,i,j)
        U_next(:,i,j) = U(:,i,j) - tau(i,j)*alpha*Risd(:,i,j)

!!          !write(*,*) Risd(:,i,j)
        if (isnan(U(1,i,j))) then
            write(*,*) "U1 is nan",i,j
            stop
        endif

        if (isnan(U(2,i,j))) then
            write(*,*) "U2 is nan",i,j
            stop
        endif

        if (isnan(U(3,i,j))) then
            write(*,*) "U3 is nan",i,j
            stop
        endif

        if (isnan(U(4,i,j))) then
            write(*,*) "U4 is nan",i,j
            stop
        endif
    enddo
    enddo


end subroutine risdual

subroutine timestep(ni,nj,n1,n2,n3,n4,U,tau)

    integer :: ni,nj,i,j
    real :: cfl
    real, dimension(ni,nj) :: tau
    real, dimension(2,ni,nj) :: n1, n2, n3, n4,dsi,dsj !Cell normals
    real, dimension(4,ni,nj) :: U
    real :: clocal,ulocal,vlocal,plocal,rholocal
    real, parameter :: gam = 1.4

    cfl = 1.0 !Max value
    dsi = 0.5*(n1-n3)
    dsj = 0.5*(n2-n4)
```

```
    do i =1,ni
    do j = 1,nj-1
        plocal = pres_func(U(:,i,j))
        rholocal = U(1,i,j)
        clocal = sqrt(abs(gam*plocal/rholocal))
        ulocal = U(2,i,j)/U(1,i,j)
        vlocal = U(3,i,j)/U(1,i,j)
        tau(i,j) = cfl/(abs((ulocal+clocal)*dsi(1,i,j)+(vlocal+clocal)*dsi(2,i
            ,j)) &
                    +abs((ulocal+clocal)*dsj(1,i,j)+(vlocal+clocal)*dsj(2,i
                        ,j)))
!        write(*,*) i,j,dsi(1,i,j),dsi(2,i,j),dsj(1,i,j),dsj(2,i,j) &
!                    ,(abs((ulocal+clocal)*dsi(1,i,j)+(vlocal+clocal)*dsi(2,
    i,j)) &
!                     +abs((ulocal+clocal)*dsj(1,i,j)+(vlocal+clocal)*dsj(2,
    i,j)))
!        write(*,*) i, j, tau(i,j)
    enddo
    enddo
end subroutine timestep

!Computes pressure at single grid location using components of U
real function pres_func(U_vec)
    implicit none
    real, parameter :: gam = 1.4
    real, dimension(4) :: U_vec

    pres_func = (gam-1.)*(U_vec(4) &
            - 1./(2.*U_vec(1))*(U_vec(2)**2 + U_vec(3)**2))
    return
end function pres_func

!Computes density at single grid location using components of U
real function rho_func(U_vec)
    implicit none
    real, dimension(4) :: U_vec

    rho_func = U_vec(1)
    return
end function rho_func

!Computes speed of sound at single grid location using components of U
real function c_func(U_vec)
    implicit none
    real, parameter :: gam = 1.4
    real, dimension(4) :: U_vec
    real :: pres_func, rho_func

    c_func = (gam*pres_func(U_vec)/rho_func(U_vec))**0.5
    return
end function c_func

real function vx_func(U_vec)
    implicit none
```

```
    real, dimension(4) :: U_vec
    vx_func = U_vec(2)/U_vec(1)
    return
end function vx_func

real function vy_func(U_vec)
    implicit none
    real, dimension(4) :: U_vec
    vy_func = U_vec(3)/U_vec(1)
    return
end function vy_func

real function E_func(U_vec)
    implicit none
    real, parameter :: gam = 1.4
    real, dimension(4) :: U_vec
    real :: pres_func, rho_func, vx_func, vy_func
    real :: e
    e = pres_func(U_vec)/(rho_func(U_vec)*(gam-1.))
    E_func = e + 0.5*(vx_func(U_vec)**2 + vy_func(U_vec)**2)
    return
end function E_func

real function H_func(U_vec)
    implicit none
    real, dimension(4) :: U_vec
    real :: pres_func, rho_func, E_func
    H_func = E_func(U_vec) + pres_func(U_vec)/rho_func(U_vec)
    return
end function H_func

real function M_func(U_vec)
    implicit none
    real, dimension(4) :: U_vec
    real :: vx_func, vy_func, c_func
    M_func = (vx_func(U_vec)**2+vy_func(U_vec)**2)**0.5/c_func(U_vec)
    return
end function M_func

real function S_func(U_vec)
    implicit none
    real, parameter :: c_v = 0.718
    real, parameter :: gam = 1.4
    real, dimension(4) :: U_vec
    real :: pres_func, rho_func
    S_func = c_v * log(pres_func(U_vec)/(rho_func(U_vec))**gam)
    return
end function S_func

real function cp_func(U_vec,p_inf,rho_inf,vx_inf,vy_inf)
    implicit none
    real, dimension(4) :: U_vec
    real :: pres_func
    real :: p_inf, rho_inf, vx_inf, vy_inf, vmag_inf
```

```
    vmag_inf = (vx_inf**2 + vy_inf**2)**0.5
    cp_func = (pres_func(U_vec) - p_inf)/(0.5*rho_inf*vmag_inf**2)
    return
end function cp_func



!Artificial Diffusivity Subroutines
subroutine nui_diff(ni,nj,U,nui)
    implicit none
    integer :: i,j,ni,nj,ip,im
    real, dimension(4,ni,nj) :: U
    real, dimension(ni,nj) :: nui
    real :: pres_func

    nui = 0.
    do i = 1,ni
    do j = 1,nj
        ip = i+1
        im = i-1
        if (i .eq. 1) then
            im = ni
        else if (i .eq. ni) then
            ip = 1
        endif

        nui(i,j) = abs(pres_func(U(:,ip,j))-2.*pres_func(U(:,i,j))+pres_func(U
            (:,im,j))) &
            /(pres_func(U(:,ip,j))+2.*pres_func(U(:,i,j))+pres_func(U(:,im,j))
                )
    enddo
    enddo
end subroutine nui_diff

subroutine nuj_diff(ni,nj,U,nuj)
    implicit none
    integer :: i,j,ni,nj
    real, dimension(4,ni,nj) :: U
    real, dimension(ni,nj+1) :: nuj !Add additional point for j+2
    real :: pres_func

    nuj = 0.
    do i = 1,ni
    do j = 2,nj-1

        nuj(i,j) = abs(pres_func(U(:,i,j+1))-2.*pres_func(U(:,i,j))+pres_func(
            U(:,i,j-1))) &
            /(pres_func(U(:,i,j+1))+2.*pres_func(U(:,i,j))+pres_func(U(:,i,j
                -1)))
    enddo
    enddo
end subroutine nuj_diff
```

```
real function u_plus_c_ds_func(ni,nj,U,n_out,i,i2,j,j2)
    implicit none
    integer :: ni, nj, i, i2, j, j2
    real, dimension(4,ni,nj) :: U
    real, dimension(2,ni,nj) :: n_out !Outward normal
    real :: vx_func, vy_func, c_func
    real :: vx_face, vy_face, c_face, ds_mag

    vx_face = 0.5*(vx_func(U(:,i,j))+vx_func(U(:,i2,j2)))
    vy_face = 0.5*(vy_func(U(:,i,j))+vy_func(U(:,i2,j2)))
    c_face  = 0.5*(c_func(U(:,i,j)) + c_func(U(:,i2,j2)))
    ds_mag = (n_out(1,i,j)**2+n_out(2,i,j)**2)**0.5

    u_plus_c_ds_func = vx_face*n_out(1,i,j)+vy_face*n_out(2,i,j) &
                       + c_face*ds_mag
    return
end function u_plus_c_ds_func

subroutine output_velocities(ni,nj,x_cell,y_cell,U,F,G,n4,p_inf,rho_inf,vx_inf
    ,vy_inf)
    implicit none
    integer :: i, j, ni, nj
    real :: vx, vy, speed
    real, dimension(ni,nj) :: x_cell,y_cell
    real, dimension(4,ni,nj) :: U, F, G
    real, dimension(2,ni,nj) :: n4 !Cell normals
    real :: e_nx, e_ny
    real :: pres_func, M_func, H_func, S_func, rho_func, cp_func
    real :: p_inf,rho_inf,vx_inf,vy_inf !Used in cp calculation

    !open(unit = 3, file = "v.dat");
    open(unit = 3, file = "v_2alpha.dat");
    !open(unit = 4, file = "p.dat");

    !Output the velocity magnitude and direction
    do i = 1,ni
    do j = 1,nj !Don't want to include the outer ghost cell
        vx = U(2,i,j)/U(1,i,j)
        vy = U(3,i,j)/U(1,i,j)
        speed = (vx**2+vy**2)**0.5
        e_nx = n4(1,i,j)/(n4(1,i,j)**2+n4(2,i,j)**2)**0.5
        e_ny = n4(2,i,j)/(n4(1,i,j)**2+n4(2,i,j)**2)**0.5

        !write(*,*) "PRES", i, j, pres_func(U(:,i,j))

        if (j .eq. nj) then
            !write(*,*) i,j, F(:,i,j)
            !Check inlet BC's
            !if (vx*n4(1,i,j) + vx*n4(2,i,j) .gt. 0.) then
            !   vx

        endif
        !if (j .eq. 1) then
```

```
        !     write(*,*) i,j, x_cell(i,j), y_cell(i,j), n4(1,i,j), n4(2,i,j), &
        !      vx, vy, vx*e_nx+vy*e_ny
        !endif
        if (j .le. nj-1) then !Don't include outer ghost cell
            write(3,*) i, j, x_cell(i,j), y_cell(i,j), vx, vy, speed, &
                pres_func(U(:,i,j)),cp_func(U(:,i,j),p_inf,rho_inf,vx_inf,
                    vy_inf), &
                M_func(U(:,i,j)),H_func(U(:,i,j)),S_func(U(:,i,j)), &
                rho_func(U(:,i,j))

            !write(4,*) i, j, x_cell(i,j), y_cell(i,j)
        endif
    enddo
    enddo
end subroutine output_velocities



subroutine inlet_outlet_bcs(ni,nj,n4,vx_inf,vy_inf,p_inf,rho_inf,c_inf,U)

    implicit none
    integer :: i,j,ni,nj

    real, dimension(4,ni,nj) :: U, F, G
    real, dimension(2,ni,nj) :: n4 !Outer Ghost Cell normal
    real, parameter :: gam = 1.4

    real :: vx_inf, vy_inf, p_inf, rho_inf, c_inf
    real :: pres_func, rho_func, c_func

    !Variables for BC's
    real :: v_inf_dot_n, e_nx, e_ny, e_tx, e_ty, p_i, rho_i, c_i, u_n_i, rni,
       rnb
    real :: u_n_b, c_b, u_t_b, rho_b, p_b, E_b

    j = nj
    do i = 1,ni
        v_inf_dot_n = vx_inf*n4(1,i,j)+vy_inf*n4(2,i,j)

        !Unit normal : only n4(1,i,nj) and n4(2,i,nj) are required
        e_nx = n4(1,i,j)/sqrt(n4(1,i,j)**2+n4(2,i,j)**2)
        e_ny = n4(2,i,j)/sqrt(n4(1,i,j)**2+n4(2,i,j)**2)
        e_tx =  e_ny     !tangential, 90 deg clockwise rotation
        e_ty = -e_nx     !tangential, 90 deg clockwise rotation

        !Calculate variables at interior cell : (k,i,nj-1), k = 1,2,3,4
        p_i = pres_func(U(:,i,j-1))
        rho_i = rho_func(U(:,i,j-1))
        c_i = c_func(U(:,i,j-1))
        u_n_i = U(2,i,j-1)/U(1,i,j-1)*e_nx + U(3,i,j-1)/U(1,i,j-1)*e_ny

        !Inlet BC's
        if(v_inf_dot_n > 0.) then
```

```
            rni = u_n_i - 2.*c_i/(gam-1.)                          !Eq 4, lec 14
            rnb = vx_inf*e_nx+vy_inf*e_ny + 2.*c_inf/(gam-1.) !Eq 5, lec 14

                        !Calculate variables at boundary ('ghost') cell : (k,i
                           ,nj), k = 1,2,3,4
            u_n_b = 0.5*(rnb+rni)                    !Eq A, lec 14
            c_b = (rnb-rni)*(gam-1.)*0.25            !Eq B, lec 14
            u_t_b = vx_inf*e_tx + vy_inf*e_ty        !Eq C, lec 14
            rho_b = (c_b**2*rho_inf**gam/p_inf/gam)**(1./(gam-1.))  !Derived
               using S_B = S_infinity
            p_b = c_b**2*rho_b/gam

        !Outlet BC's
        else
            rni = -abs(u_n_i) - 2.*c_i/(gam-1.)                          !Eq B,
               lec. 14
            !rnb = -abs(vx_inf*e_nx+vy_inf*e_ny) + p_inf/(rho_inf*c_inf)  !Eq
               A, lec. 14
            rnb = -abs(vx_inf*e_nx+vy_inf*e_ny) + 2.*c_inf/(gam-1.)   !Eq A,
               lec. 14

            !rni = -abs(u_n_i) + 2.*c_i/(gam-1.)                          !Eq B,
               lec. 20
            !rnb = -abs(vx_inf*e_nx+vy_inf*e_ny) - 2.*c_inf/(gam-1.)     !Eq A,
                lec. 20

                        !Calculate variables at boundary ('ghost') cell : (k,i
                           ,nj), k = 1,2,3,4
            u_n_b = 0.5*(rnb+rni)                                        !Eq A,
               lec 14
            c_b = (rnb-rni)*(gam-1.)*0.25                                !Eq B,
               lec 14
            !c_b = (rni-rnb)*(gam-1.)*0.25                               !Eq B,
               lec 14
            u_t_b = U(2,i,j-1)/U(1,i,j-1)*e_tx + U(3,i,j-1)/U(1,i,j-1)*e_ty
                     !Eq C, lec 14
            rho_b = (c_b**2*rho_i**gam/p_i/gam)**(1./(gam-1.))           !Derived
                using S_B = S_I
            p_b = c_b**2*rho_b/gam

        endif

        !Assign BC's
        U(1,i,j) = rho_b
        U(2,i,j) = rho_b*(u_n_b*e_ty - u_t_b*e_ny)/(e_nx*e_ty - e_ny*e_tx) !
            Solve linear equations: vn = vx*nx +vy*ny; vt = vx*tx +vy*ty
        U(3,i,j) = rho_b*(u_n_b*e_tx - u_t_b*e_nx)/(e_ny*e_tx - e_nx*e_ty) !
            Solve linear equations: vn = vx*nx +vy*ny; vt = vx*tx +vy*ty

        E_b = (1./(gam-1.))*(p_b/rho_b) + 0.5*((U(2,i,j)/rho_b)**2 + (U(3,i,j)
            /rho_b)**2)
        U(4,i,j) = rho_b*E_b
!           write(*,*) x_cell(i,j), y_cell(i,j), i,j, n4(1,i,j), n4(2,i,j), U
```

```
   (2,i,j)/rho_b, &
!                         U(3,i,j)/rho_b, U(1,i,j), rho_inf, U(4,i,j), rho_inf*
   E_inf, p_b, p_inf
!         write(*,*) i,j, n4(1,i,j), n4(2,i,j), U(2,i,j)/rho_b    &
!         ,U(3,i,j)/rho_b, U(1,i,j), rho_inf, U(4,i,j), p_b, p_inf

    enddo

end subroutine inlet_outlet_bcs
```

B Grid Generation

```
program hw3_1

!Grid generation: NACA 0012

integer, parameter ::nni = 129, nnj = 65  !nni = 129, nnj = 65  !
!integer, parameter ::nni = 16, nnj = 9  !nni = 129, nnj = 65  !

real, dimension(:), allocatable :: xb,yb,xbound,ybound,theta,l1,l1j,l2,l2j
real, dimension(:,:), allocatable :: x,y,Ax,Ay,Bx,By,Tx,Ty,xjnew,yjnew,xj,yj
real, parameter :: pi = 4.*atan(1.)
real :: alpha,beta,gamma1,delta,p,q,r,rx,ry,rbar,xold,yold,rjbar,ap,bp,cp,dp,
   dap,dcp,aq,bq,cq,dq,daq,dcq
real, parameter :: w = 1.2 !w used for SOR
integer :: n,i,j,l

!real :: x,y

allocate(xb(nni),yb(nni),x(nni+1,nnj+1),y(nni+1,nnj+1),xjnew(nni+1,nnj+1),
   yjnew(nni+1,nnj+1),xj(nni+1,nnj+1),yj(nni+1,nnj+1))
allocate(theta(nni))
allocate(xbound(nni),ybound(nni))
allocate(Ax(nni,nni),Ay(nni,nni),Bx(nni,nni),By(nni,nni),Tx(nni,nni),Ty(nni,
   nni))
allocate(l1(nni),l2(nni),l1j(nni),l2j(nni))

theta(1) = 0.
theta(nni) = 2.*pi
xbound(1) = 10.5
xbound(nni) = 10.5
ybound(1) = 0.
ybound(nni) = 0.

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!generate airfoil(body)

    !Get Body x and y coordinates
    call body(0012,nni,xb,yb);

!open(10,file = 'body.dat')
!do i = 1,nni
```

```
!  read(10,*) n, xb(i) , yb(i)
!enddo

close(10)
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!generate boundary
do i = 2,nni-1
  theta(i) = (i-1.)/(nni-1.)*2.*pi
  xbound(i) = 0.5 + 10.*cos(theta(i))
  ybound(i) = -10.*sin(theta(i))
enddo
!write(*,*) ybound
!write(*,*) xb
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!Assign initial conditions
x(1:nni,1) = xb
y(1:nni,1) = yb

x(1:nni,nnj) = xbound
y(1:nni,nnj) = ybound
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!cut
do j = 2,nnj-1
  x(1,j) = 1+9.5/64.*(j-1)
enddo

x(nni,1:nnj-1) = x(1,:)
y(1,1:nnj) = 0.
y(nni,1:nnj) = 0.

!write(*,*) y(:,1)
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!TFI
do j = 2,nnj-1
  do i = 2,nni-1

    l1(i) = (i-nni)/(1.-nni)

    l2(i) = (i-1.)/(nni-1.)

    l1j(j) = (j-nnj)/(1.-nnj)

    l2j(j) = (j-1.)/(nnj-1.)

    Ax(i,j) = l1(i)*x(1,j)+l2(i)*x(nni,j)
    Ay(i,j) = l1(i)*y(1,j)+l2(i)*y(nni,j)

    Bx(i,j) = l1j(j)*x(i,1)+l2j(j)*x(i,nnj)
    By(i,j) = l1j(j)*y(i,1)+l2j(j)*y(i,nnj)

    Tx(i,j) = l1(i)*l1j(j)*x(1,1)+l2(i)*l2j(j)*x(nni,nnj)+l1(i)*l2j(j)*x(1,nnj
        )+l2(i)*l1j(j)*x(nni,1)
```

```
    Ty(i,j) = l1(i)*l1j(j)*y(1,1)+l2(i)*l2j(j)*y(nni,nnj)+l1(i)*l2j(j)*y(1,nnj
        )+l2(i)*l1j(j)*y(nni,1)

    x(i,j) = Ax(i,j)+Bx(i,j)-Tx(i,j)
    y(i,j) = Ay(i,j)+By(i,j)-Ty(i,j)

  enddo
enddo

x(nni+1,:) = x(2,:)
y(nni+1,:) = y(2,:)

!!END INITIAL CONDITION
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!open(unit=1, file='x_init.txt', ACTION="write", STATUS="replace")

!do i = 1,nni
!  WRITE(1,*)(x(i,j),j = 1,nnj)
!ENDDO

!close(1)


!open(unit=1, file='y_init.txt', ACTION="write", STATUS="replace")

!do i = 1,nni
!  WRITE(1,*)(y(i,j),j = 1,nnj)
!ENDDO

!close(1)
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!ASSIGN INITIAL CONDITION - Jacobi Method
!xj = x
!yj = y
!xjnew = xj
!yjnew = yj
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!P and Q
!1. No clustering.
!2. Boundary Layer mesh: clustering in j; no clustering in i.
!3. Clustering about a point (i,j) = (65,33)
p = 0.
q = 0.
!1.
!ap = 0.; bp = 0.; cp = 0.; dp = 0.
!aq = 0.; bq = 1.; cq = 0.; dq = 0.25
!dap = 0.; dcp = 0.
!daq = 200.; dcq = 0.
!2.
!ap = 0.; bp = 0.; cp = 0.; dp = 0.
!aq =3000.; bq = 0.5; cq = 0.; dq = 0.
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!Grid generation

!xold = x
!yold = y

l = 0
rbar = 1.
rjbar = 1.
do while(rbar.gt.1e-6)
!do k = 1,1116
! ap = ap + dap !; cp = cp + dcp
  aq = aq + daq !; cq = cq + dcq
  l = l + 1
  do j = 2,nnj-1
    do i = 2,nni

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!P and Q
!      q = -300.*sign(1,(j-1))*exp(-0.15*(j-1))
!     p = -ap*sign(1.,i-65.)*exp(-bp*(abs(i-65.)))-cp*sign(1.,(i-65.))*exp(-dp
    *(sqrt((i-65.)**2+(j-33.)**2)))
!     q = -aq*sign(1.,(j-1.))*exp(-bq*(abs(j-1.)))-cq*sign(1.,(j-33.))*exp(-dq
    *(sqrt((i-65.)**2+(j-33.)**2)))
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!


!       xjnew(nni+1,j) = xjnew(2,j)          !pe!od!B  !Jacobi
!       yjnew(nni+1,j) = yjnew(2,j)          !ri!ic!C

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!       xold = x
!       yold = y
!SOR!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
      xold = x(i,j)
      yold = y(i,j)

      alpha  = 0.25*((x(i,j+1)-x(i,j-1))**2+(y(i,j+1)-y(i,j-1))**2)
      beta   = 0.25*((x(i+1,j)-x(i-1,j))*(x(i,j+1)-x(i,j-1))+(y(i+1,j)-y(i-1,j
         ))*(y(i,j+1)-y(i,j-1)))
      gamma1 = 0.25*((x(i+1,j)-x(i-1,j))**2+(y(i+1,j)-y(i-1,j))**2)
      delta  = 0.0625*((((x(i+1,j)-x(i-1,j))*(y(i,j+1)-y(i,j-1)))-((x(i,j+1)-x
         (i-1,j))   &
      * (y(i+1,j)-y(i-1,j)))))**2)

      x(i,j) = 0.5*((alpha*(x(i-1,j)+x(i+1,j)))-(0.5*beta*(x(i+1,j+1)-x(i-1,j
         +1)   &
      -x(i+1,j-1)+x(i-1,j-1)))+(gamma1*(x(i,j-1)+x(i,j+1)))+(0.5*delta*p*(x(i
         +1,j)  &
      -x(i-1,j)))+(0.5*delta*q*(x(i,j+1)-x(i,j-1))))/(alpha+gamma1)

      y(i,j) = 0.5*((alpha*(y(i-1,j)+y(i+1,j)))-(0.5*beta*(y(i+1,j+1)-y(i-1,j
         +1)   &
```

```
      -y(i+1,j-1)+y(i-1,j-1)))+(gamma1*(y(i,j-1)+y(i,j+1)))+(0.5*delta*p*(y(i
          +1,j)  &
      -y(i-1,j)))+(0.5*delta*q*(y(i,j+1)-y(i,j-1))))/(alpha+gamma1)

       x(i,j) = xold + w*(x(i,j)-xold) !Comment for Gauss Seidel
       y(i,j) = yold + w*(y(i,j)-yold) !Comment for Gauss Seidel
       !SOR!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
       x(1,j) = x(nni,j)                  !pe!od!B
       y(1,j) = y(nni,j)                  !ri!ic!C
       x(nni+1,j) = x(2,j)                      !pe!od!B  !SOR
       y(nni+1,j) = y(2,j)                      !ri!ic!C
      if (isnan(y(i,j))) stop '"y" is a NaN'
      if (isnan(x(i,j))) stop '"x" is a NaN'

       enddo
  enddo

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

  !Residual!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  r = 0.
!   rj = 0.
  do j = 2,nnj-1
    do i = 2,nni-1
!      if (isnan(x(i,j))) stop '"x" is a NaN'
!      if (isnan(y(i,j))) stop '"y" is a NaN'
!      if (isnan(q)) stop '"q" is a NaN'
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!P and Q
!      q = -300.*sign(1,(j-1))*exp(-0.15*(j-1))

!     p = -ap*sign(1.,i-65.)*exp(-bp*(abs(i-65.)))-cp*sign(1.,(i-65.))*exp(-dp
    *(sqrt((i-65.)**2+(j-33.)**2)))
!     q = -aq*sign(1.,(j-1.))*exp(-bq*(abs(j-1.)))!-cq*sign(1.,(j-33.))*exp(-dq
    *(sqrt((i-65.)**2+(j-33.)**2)))
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!SOR
      alpha  = 0.25*((x(i,j+1)-x(i,j-1))**2+(y(i,j+1)-y(i,j-1))**2)
      beta   = 0.25*((x(i+1,j)-x(i-1,j))*(x(i,j+1)-x(i,j-1))+(y(i+1,j)-y(i-1,j
          ))*(y(i,j+1)-y(i,j-1)))
      gamma1 = 0.25*((x(i+1,j)-x(i-1,j))**2+(y(i+1,j)-y(i-1,j))**2)
      delta  = 0.0625*((((x(i+1,j)-x(i-1,j))*(y(i,j+1)-y(i,j-1)))-((x(i,j+1)-x
          (i-1,j))   &
      * (y(i+1,j)-y(i-1,j)))))**2)
      !LHS!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
      rx =  (alpha*(x(i-1,j)-2.*x(i,j)+x(i+1,j)))-(0.5*beta*(x(i+1,j+1)-x(i-1,
          j+1)   &
      -x(i+1,j-1)+x(i-1,j-1)))+(gamma1*(x(i,j-1)-2.*x(i,j)+x(i,j+1)))+(0.5*
          delta*p*(x(i+1,j)  &
      -x(i-1,j)))+(0.5*delta*q*(x(i,j+1)-x(i,j-1)))

      ry =  (alpha*(y(i-1,j)-2.*y(i,j)+y(i+1,j)))-(0.5*beta*(y(i+1,j+1)-y(i-1,
```

```
          j+1)     &
        -y(i+1,j-1)+y(i-1,j-1)))+(gamma1*(y(i,j-1)-2.*y(i,j)+y(i,j+1)))+(0.5*
          delta*p*(y(i+1,j)   &
        -y(i-1,j)))+(0.5*delta*q*(y(i,j+1)-y(i,j-1)))

        r = (abs(rx)+abs(ry))*0.5 + r
      enddo
    enddo
    rbar = r/(nni-1.)/(nnj-1.)
!   rjbar = r/(nni-1)/(nnj-1)
!open(unit = 10,file = 'res_pq_j1.txt',action = "write")
!   write(*,*)l,rbar

! close(10)
enddo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!



!close(10)

open(unit=1, file='x.txt', ACTION="write", STATUS="replace")
do i = 1,nni
  WRITE(1,*)(x(i,j),j = 1,nnj)
ENDDO
close(1)

!open(unit=1, file='y.txt', ACTION="write", STATUS="replace")
open(unit=1, file='y.txt', ACTION="write", STATUS="replace")
do i = 1,nni
  WRITE(1,*)(y(i,j),j = 1,nnj)
ENDDO
close(1)

!Write x and y data that is read in by euler2D
open(unit = 2, file = "coords.dat");
do i = 1,nni
do j = 1,nnj
    write(2,*) i, j, x(i,j), y(i,j)
enddo
enddo



end program hw3_1
```

C Post Processing

```
#!/usr/bin/python
import re
#import matplotlib.pyplot as plt
import numpy as np
import matplotlib.pyplot as plt
```

```
#ni = 5-1
#nj = 4
ni = 129-1
nj = 65
#ni = 16-1
#nj = 9

direc = './'
filename = direc + 'v.dat'
filename2 = direc + 'coords.dat'
f = open(filename, 'r')

x_cell = np.zeros([ni,nj])
y_cell = np.zeros([ni,nj])
vx = np.zeros([ni,nj])
vy = np.zeros([ni,nj])
pres = np.zeros([ni,nj])
cp = np.zeros([ni,nj])
Mach = np.zeros([ni,nj])
Enthalpy = np.zeros([ni,nj])
Entropy = np.zeros([ni,nj])
rho = np.zeros([ni,nj])
speed = np.zeros([ni,nj])

linenum = 0
for line in f:
    words = line.split()
    i = int(words[0])-1
    j = int(words[1])-1
    x_cell[i,j] = words[2]
    y_cell[i,j] = words[3]
    vx[i,j] = words[4]
    vy[i,j] = words[5]
    speed[i,j] = words[6]
    pres[i,j] = words[7]
    cp[i,j] = words[8]
    Mach[i,j] = words[9]
    Enthalpy[i,j] = words[10]
    Entropy[i,j] = words[11]
    rho[i,j] = words[12]

f = open(filename2, 'r')
#print f.readline()

ni = ni+1
x = np.zeros([ni,nj])
y = np.zeros([ni,nj])
i = 0
j = 0
linenum = 0
for line in f:
    words = line.split()
```

```
    i = int(words[0])-1
    j = int(words[1])-1
    x[i,j] = words[2]
    y[i,j] = words[3]



plt.figure(num=1, figsize=(8, 6))
plt.title(r'Magnitude of Velocity for NACA 0012 ($\alpha=0$)', size=14)
#plt.title('Euler 2D Magnitude of Velocity', size=14)
plt.ylim([-1,1])
plt.xlim([-1,2])
plt.xlabel(r'$x$', size=14)
plt.ylabel(r'$y$', size=14)
plt.pcolormesh(x,y,speed)
cbar = plt.colorbar()
cbar.set_label('      ||v||',rotation=0);
plotname = './plot_speed_zoom.png'
plt.savefig(plotname, format='png')
plt.clf()

plt.figure(num=1, figsize=(8, 6))
plt.title(r'Pressure for NACA 0012 ($\alpha=0$)', size=14)
plt.ylim([-1,1])
plt.xlim([-1,2])
plt.xlabel(r'$x$', size=14)
plt.ylabel(r'$y$', size=14)
plt.pcolormesh(x,y,pres)
cbar = plt.colorbar()
cbar.set_label('P',rotation=0);
plotname = './plot_pressure_zoom.png'
plt.savefig(plotname, format='png')
plt.clf()

plt.figure(num=1, figsize=(8, 6))
plt.title(r'Pressure Coefficient for NACA 0012 ($\alpha=0$)', size=14)
#plt.title('Euler 2D Pressure Coefficient', size=14)
plt.ylim([-1,1])
plt.xlim([-1,2])
plt.xlabel(r'$x$', size=14)
plt.ylabel(r'$y$', size=14)
plt.pcolormesh(x,y,cp)
cbar = plt.colorbar()
cbar.set_label('$c_p$',rotation=0);
plotname = './plot_cp_zoom.png'
plt.savefig(plotname, format='png')
plt.clf()

plt.figure(num=1, figsize=(8, 6))
plt.title(r'Mach # for NACA 0012 ($\alpha=0$)', size=14)
#plt.title('Euler 2D Mach', size=14)
plt.ylim([-1,1])
plt.xlim([-1,2])
plt.xlabel(r'$x$', size=14)
```

```
plt.ylabel(r'$y$', size=14)
plt.pcolormesh(x,y,Mach)
cbar = plt.colorbar()
cbar.set_label('M',rotation=0);
plotname = './plot_Mach_zoom.png'
plt.savefig(plotname, format='png')
plt.clf()

plt.figure(num=1, figsize=(8, 6))
plt.title(r'Enthalpy for NACA 0012 ($\alpha=0$)', size=14)
#plt.title('Euler 2D Enthalpy', size=14)
plt.ylim([-1,1])
plt.xlim([-1,2])
plt.xlabel(r'$x$', size=14)
plt.ylabel(r'$y$', size=14)
plt.pcolormesh(x,y,Enthalpy)
cbar = plt.colorbar()
cbar.set_label('H',rotation=0);
plotname = './plot_Enthalpy_zoom.png'
plt.savefig(plotname, format='png')
plt.clf()

plt.figure(num=1, figsize=(8, 6))
plt.title(r'Entropy for NACA 0012 ($\alpha=0$)', size=14)
#plt.title('Euler 2D Entropy', size=14)
plt.ylim([-1,1])
plt.xlim([-1,2])
plt.xlabel(r'$x$', size=14)
plt.ylabel(r'$y$', size=14)
plt.pcolormesh(x,y,Entropy)
cbar = plt.colorbar()
cbar.set_label('S',rotation=0);
plotname = './plot_Entropy_zoom.png'
plt.savefig(plotname, format='png')
plt.clf()

plt.figure(num=1, figsize=(8, 6))
plt.title(r'Density for NACA 0012 ($\alpha=0$)', size=14)
#plt.title('Euler 2D Density', size=14)
plt.ylim([-1,1])
plt.xlim([-1,2])
plt.xlabel(r'$x$', size=14)
plt.ylabel(r'$y$', size=14)
plt.pcolormesh(x,y,rho)
cbar = plt.colorbar()
cbar.set_label(r'$ \rho$',rotation=0);
plotname = './plot_rho_zoom.png'
plt.savefig(plotname, format='png')
plt.clf()

#Plot of variables on the surface of the airfoil
plt.figure(num=1, figsize=(8, 6))
plt.title(r'Pressure Coefficient for NACA 0012 ($\alpha=0$) Surface of Airfoil
    ', size=14)
```

```
#plt.title('Euler 2D $c_p$ Surface of Airfoil', size=14)
plt.xlabel(r'$x/c$', size=14)
plt.ylabel(r'$c_p$', size=14)
plt.plot(x[0:ni/2,0],cp[0:ni/2,0],'b-',label='Top Surface')
plt.plot(x[ni/2+1:ni-1,0],cp[ni/2+1:ni-1,0],'r-',label='Bottom Surface')
plt.gca().invert_yaxis()
plt.legend(prop={'size':11})
plotname = './plot_cp_airfoil_surf.png'
plt.savefig(plotname, format='png')
plt.clf()

##plt.figure(num=1, figsize=(8, 6))
##plt.title(r'Density for NACA 0012 ($\alpha=0$) Surface of Airfoil', size=14)
###plt.title('Euler 2D Density Surface Airfoil', size=14)
###plt.xlim([0,2])
##plt.xlabel(r'$x/c$', size=14)
##plt.ylabel(r'$\rho$', size=14)
##plt.plot(x[1:(ni/2),0],rho[1:(ni/2),0],'b-',label='Top Surface')
##plt.plot(x[(ni/2+1):(ni-2),0],rho[(ni/2+1):(ni-2),0],'r-',label='Bottom
    Surface')
###plt.plot(x[0:(ni/2),0],rho[0:(ni/2),0],'b-')
###plt.plot(x[(ni/2+1):(ni-1),0],rho[(ni/2+1):(ni-1),0],'r-')
##plt.legend(prop={'size':11})
##plotname = './plot_rho_airfoil_surf.png'
##plt.savefig(plotname, format='png')
##plt.clf()

plt.figure(num=1, figsize=(8, 6))
plt.title(r'X Velocity for NACA 0012 ($\alpha=0$) Surface of Airfoil', size
    =14)
#plt.title('Euler 2D x velocity on Surface of Airfoil', size=14)
plt.xlabel(r'$x/c$', size=14)
plt.ylabel(r'$v_x$', size=14)
plt.plot(x[0:ni/2,0],vx[0:ni/2,0],'b-',label='Top Surface')
plt.plot(x[ni/2+1:ni-1,0],vx[ni/2+1:ni-1,0],'r-',label='Bottom Surface')
plt.legend(prop={'size':11})
plotname = './plot_vx_airfoil_surf.png'
plt.savefig(plotname, format='png')
plt.clf()

plt.figure(num=1, figsize=(8, 6))
plt.title(r'Y Velocity for NACA 0012 ($\alpha=0$) Surface of Airfoil', size
    =14)
#plt.title('Euler 2D y velocity on Surface of Airfoil', size=14)
plt.xlabel(r'$x/c$', size=14)
plt.ylabel(r'$v_y$', size=14)
plt.plot(x[0:ni/2,0],vy[0:ni/2,0],'b-',label='Top Surface')
plt.plot(x[ni/2+1:ni-1,0],vy[ni/2+1:ni-1,0],'r-',label='Bottom Surface')
plt.legend(prop={'size':11})
plotname = './plot_vy_airfoil_surf.png'
plt.savefig(plotname, format='png')
plt.clf()
```