# Computational Fluid Dynamics
# The Fundamentals

By

**Nicholas Cheong**

APRIL 22, 2018

UNIVERSITY OF GLASGOW

**Dr George Barakos**

# Contents

# Nomenclature

| | |
|---|---|
| $u, v, w$ | Cartesian velocities |
| $u_r$ | Radial velocity |
| $u_\theta$ | Tangential velocity |
| $u_{mag}$ | Velocity magnitude |
| $N$ | Arbitrary limit number |
| $\Delta x, x_s$ | Grid spacing in $x$ |
| $\Delta y, y_s$ | Grid spacing in $y$ |
| $\sigma$ | Truncation error |
| $i, j, k$ | Spatial dimension indexes |
| $n$ | Temporal index |
| $\vec{V}$ | Velocity vector |
| $\hat{n}$ | Unity normal |
| $\Psi$ | Stream function |
| $a, b, c, d$ | Arbitrary distances between grid point and circular geometry |
| $c_i$ | Circle centre in $x$ |
| $c_j$ | Circle centre in $y$ |
| $r$ | Current radial position |
| $R$ | radius of circle |
| $NP$ | Number of grid points |
| $\theta$ | Angle in relation to the flow direction |
| $A_{ratio}$ | Ratio of exact and scheme area |
| $A_{scheme}$ | Finite difference scheme area |

| | |
|---|---|
| $A_{exact}$ | Exact solution area |
| $a\frac{\Delta t}{\Delta x}$ | Courant Friedrichs-Lewy factor |
| $a$ | Speed of wave propagation |
| $\Delta t$ | Time step |
| $\lvert G_f \rvert$ | Amplification factor |
| $\beta$ | Wave number |
| $\lambda_m$ | Wavelength |
| $I$ | Imaginary component |
| $L$ | Length of discrete domain |
| $\rho$ | Density |
| $\rho u$ | Momentum in $x$ |
| $\rho v$ | Momentum in $y$ |
| $\rho w$ | Momentum in $z$ |
| $p$ | Pressure |
| $E$ | Total energy |
| $A$ | Jacobian Matrix |
| $\lambda$ | Eigenvalue |
| $W$ | Conservative variables |
| $\vec{Q}$ | Flux tensor |
| $\vec{n}$ | Unit normal vector |
| $E, F, G$ | The flux vectors with associated cartesian directions |
| $\gamma$ | Specific heat coefficient |
| $a$ | Speed of sound |
| $S_i$ | Runga-kutta coefficients |

| | |
|---|---|
| $S_A$ | Area of the cell |
| $Q_{FV,K}$ | Flux velocity across edge |
| $D$ | Dissipative term |
| $d^i$ | Dissipative term part |
| $v_i$ | Shock sensor |
| $k^i$ | Empirical constant based on finite computer precision |
| $K$ | Cell reference |
| $P$ | Cell reference |
| $IA$ | Cell vertex reference |
| $IB$ | Cell vertex reference |
| $e$ | Edge reference |

# Chapter 1:

# Potential flow around a two-dimensional continuum

## 1.1 Objectives and overview

The objective of the assignment was gain insight into constructing Computational Fluid Dynamic codes and to simulate potential flow over a two-dimensional fluid continuum entailing a circle surrounded by walls with an inlet. Figure 1.1 depicts the fluid domain required to simulate.

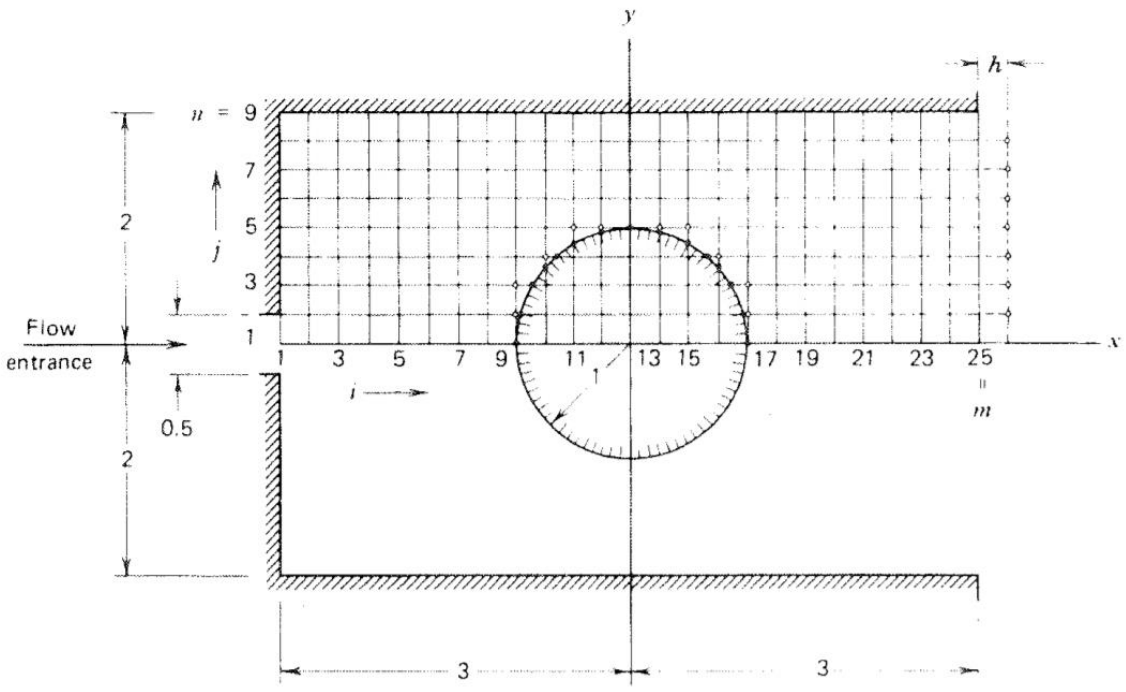



*Figure 1.1 – Fluid domain for problem with an example of a cartesian structured grid over the domain. Image courtesy of Dr Barakos [14].*

A symmetric boundary condition can be employed at the centre line of the domain to reduce the computational time. The flow was expected to have three stagnation points, before/after the circle at the furthest $x$ locations and at the wall corner adjacent to the inlet.

## 1.2 Finite difference schemes and grid generation

### 1.2.1 Approximating derivatives and accuracy considerations

The flow over the circle was solved using a finite difference scheme. For this type of discretization, the grid is generated over a fluid continuum consisting of grid lines and nodes which are indexed typically with *i, j* and *k* which represent the spatial dimensions (see figure 2.1). The overall objective of employing this numerical method is to discretise a continuous domain of PDE(s) into a solvable discrete domain. The nodes, or so-called evaluation points, are where the selected governing equations of fluid dynamics are solved relative to the neighbouring nodes and with an appropriate finite difference scheme [1]. Figure 1.2 displays an example of a one-dimensional continuous system (sine curve) discretized in the spatial dimension.

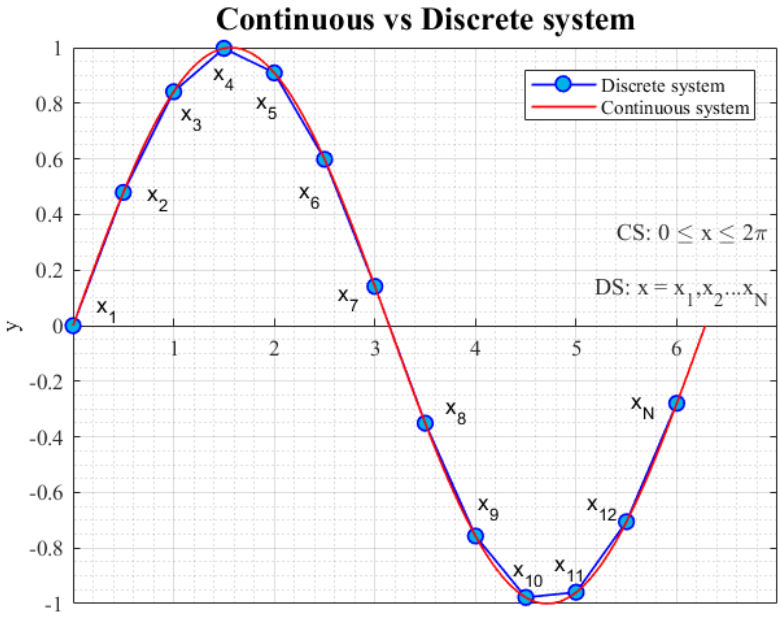



*Figure 1.2 - Continuous versus discrete system of a representative sin(x) curve. Note, the discretization for the discrete system is elaborately coarse to highlight the concept.*

The mathematical definition of a first order derivative of function is the gradient. In a finite difference scheme, the gradient at the evaluation point is evaluated using the neighbouring grid points. There are three fundamental schemes: forward, backwards and central difference. For a first order derivative, the forward and backwards estimate the gradient with the evaluation point and one other neighbouring grid point. The central difference differs and uses two

neighbouring points. The discretized equation used for the first derivative is found using the Taylor series expansion. Using the forwards difference as an example, equations 1 – 3 depict the derivation of the 1st derivative in one dimension. Equations 3,4 and 5 show the forward, backwards and central difference respectively.

Forward difference Taylor series expansion:

$$u_{i+1} = u_i + \Delta x \frac{\partial u}{\partial x} + \frac{(\Delta x)^2}{2!}\frac{\partial^2 u}{\partial x^2} + \cdots \frac{(\Delta x)^N}{N!}\frac{\partial^N u}{\partial x^N} \quad (1)$$

$$\frac{\partial u}{\partial x} = \frac{u_{i+1} + u_i}{\Delta x} + \frac{\Delta x}{2!}\frac{\partial^2 u}{\partial x^2} + \cdots \frac{(\Delta x)^{N-1}}{N!}\frac{\partial^N u}{\partial x^N} \quad (2)$$

$$\frac{\partial u}{\partial x} = \frac{u_{i+1} - u_i}{\Delta x} + \sigma(\Delta x) \quad \text{Forward difference scheme} \quad (3)$$

$$\frac{\partial u}{\partial x} = \frac{u_i - u_{i-1}}{\Delta x} + \sigma(\Delta x) \quad \text{Backward difference scheme} \quad (4)$$

$$\frac{\partial u}{\partial x} = \frac{u_{i+1} - u_{i-1}}{2\Delta x} + \sigma(\Delta x^2) \quad \text{Central difference scheme} \quad (5)$$

Where $N \to \infty$ and $\sigma(\Delta x)$ is the truncation error and the order of which is defined by the order of the lowest derivative which is eliminated from the total solution [1]. The power of the truncation error defines the order of error, where for a forward difference scheme of the 1st derivative, is a first-order accurate representation of the continuous system. The central difference scheme results in a 2nd order accurate representation and is derived by combining the backwards and forwards difference equations.

Furthermore, the accuracy of these schemes is highly dependent on the function under evaluation, local curvature and magnitude of discretization. Graphically representing and referring to figure 1.2, the gradient approximation of the continuous sine curve between approximately $2.5 \leq x \leq 3.5$ is well represented at point $x_7$ for all finite difference schemes, however, when strong curvature is present between $0.5 \leq x \leq 2.5$ and at point $x_4$, the

estimation of the gradient is significantly inaccurate for forwards and backwards but more for central difference. Figure 1.3 further highlights the difference in accuracy of gradient approximation between the schemes.

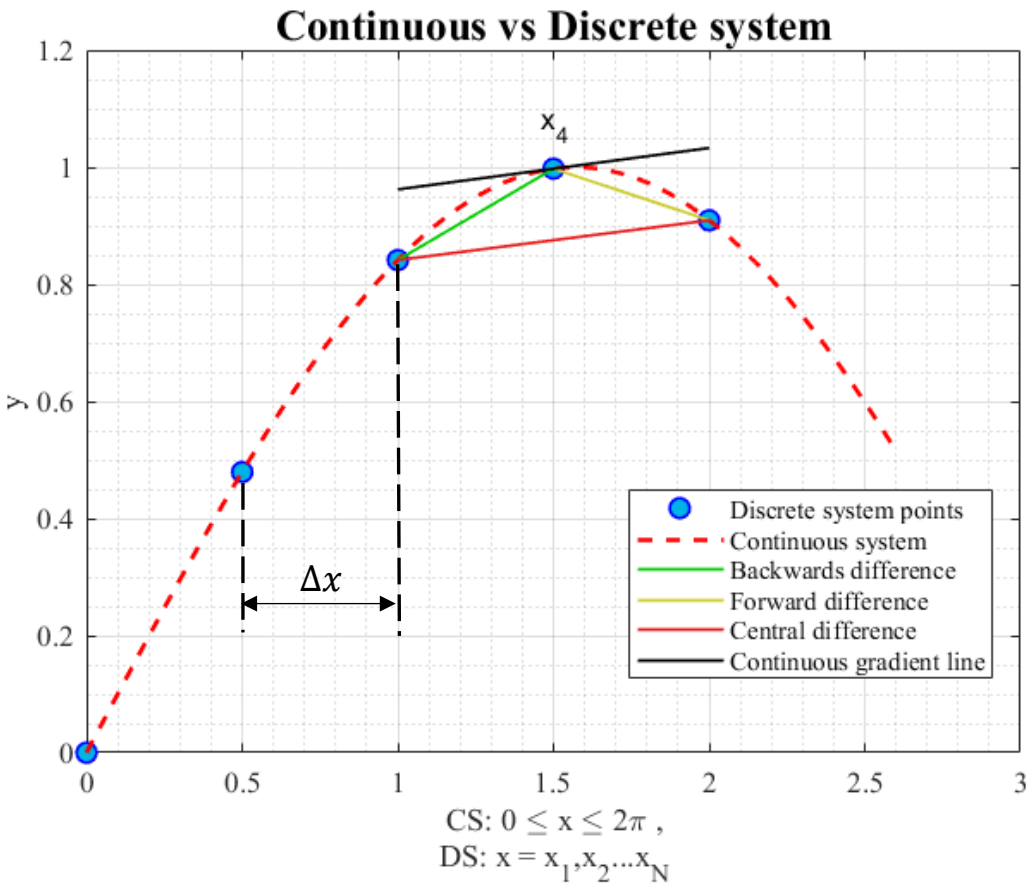



*Figure 1.3 - Forwards, backwards and central difference example of a continuous sin(x) curve showing the difference between accuracies of gradient approximation for $x_4$*

With consideration of this, it is also clear that with decrease in grid spacing, the accuracy will increase as $\Delta x \to 0$, where $\Delta x = 0$ represents the true continuous system. However, with the increase in accuracy introduces an increase in computational costs as a finite difference scheme for the fluid continuum calculates the governing equations at each evaluation point, which consequently requires more calculations with increase in nodes. On a further note, as $\Delta x \to 0$, the computer round-off error starts to become significant as computers have finite precision for floating-point numbers.

### 1.2.2 Grid generation

The grid generated was of a cartesian structured type and different levels of refinement were explored for accuracy considerations on the numerical solution. Figure 1.4 and 1.5 depict the grid for the grid likewise to figure 1.1 and an example of further refinement. Note, the colours indicate boundary values which will be discussed in later sections.

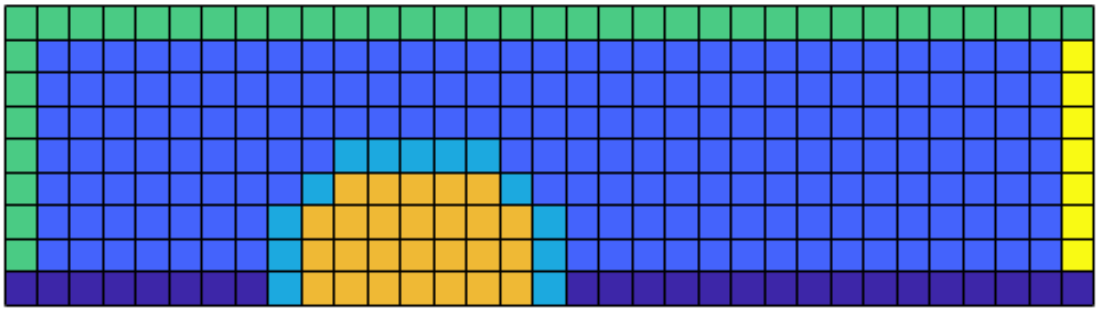


*Figure 1.4 – Course grid for flow domain.*

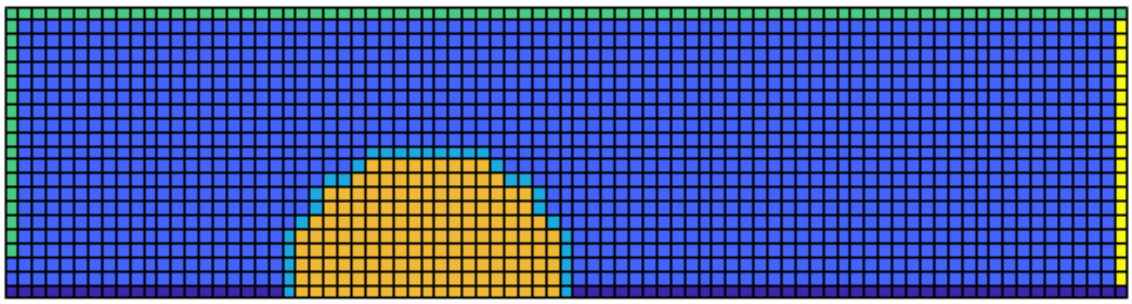


*Figure 1.5 – Grid refined, however, still slightly course*

The grid refinement is important to enable an accurate depiction of the circle geometry and inlet. Furthermore, it is important to note that an odd number of points should be specified for each spatial dimension as the spacing is calculated with one point less than the specified as there are points located on the boundaries. Therefore, using an odd number of points allows the calculate of spacing to be of division of an even number by even; this avoids large floating-point numbers for the grid spacing which allows conditional statements to be executed more accurately. This is clear in figure 1.1, where the $y$ spatial dimension has 9 points but 8 segments. Additionally, grid points must lie on the circle outer $x$ and top boundaries in order to allow

consistency of grid refinement for $N^{th}$ degree of refinement and depiction of key flow characteristics of the stagnation points and maximum velocity on the top of the circle.

## 1.3 Simulating the flow

Structured matrices were employed to hold all variables (excluding index variables) where the benefit is that it can be used to sort variables into different categories which increases clarity of the code and workspace. In the code created, four different structured matrices were used:

- Real Value Parameter (RVP.), used to store values of physical parameters such as fluid continuum height/width, circle coordinates etc.
- Normalized Parameter (NP.), used to store normalized values of the real value parameters. These values were used within the computations as computers operate more efficiently and precisely with numbers between 0 and 1.
- Grid Parameter (GP.), used to store grid parameters such as index spacing $(i, j)$ and the physical geometry in terms of index values.
- Working Variable (WV.), used for variables within the main computation loops and primarily for clarity of workspace and speed.

### 1.3.1 Defining boundary conditions of the fluid continuum

There are five main boundary conditions to consider for this problem. Each condition was set an identification number to be stored in a so-called identity matrix. The boundary conditions and identification numbers are displayed in table 1.1:

*Table 1.1 - Boundary conditions and Identification numbers*

| Boundary condition | Identification number (ID) |
|---|---|
| Near circle points | 0 |
| Outer walls | 1 |
| Interior circle points | 2 |
| Outlet | 3 |
| Symmetry line | -2 |
| Additionally: Inner fluid | -1 |

The identification numbers were set during the grid generation and are employed to classify which calculations are executed for each grid point in addition to other conditional statements. They are the foundation of the code as they define the grid layout. The identification points were established using the known geometry of the fluid continuum with use of conditional “if” statements, e.g. ““if” the current grid point is located within the circle…”. On an additional note, if the identification number was set to that of an interior circle point, no calculations are executed as it is nonsensical to calculate within the geometry; this also saves computational time. Figure 1.6 below depicts the identification matrix for an elaborately course (unrefined) grid to portray the concept. Furthermore, figure 1.7, 1.8 displays the identification matrix of a refined grid, highlighting the adaptability of the code to decreasing grid spacing.

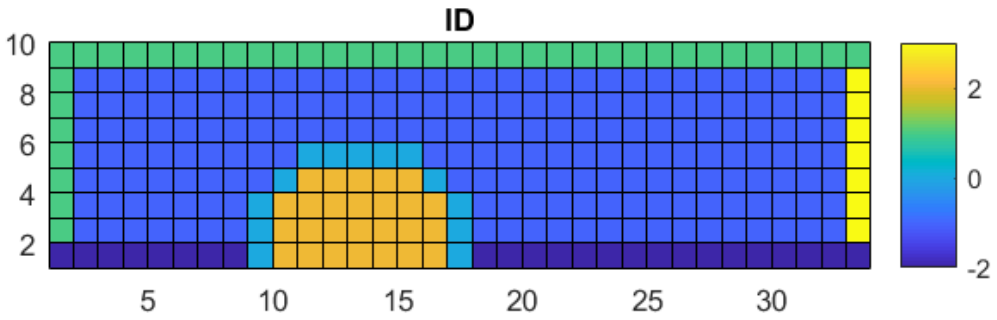



*Figure 1.6 - ID number positions of an elaborately course grid*

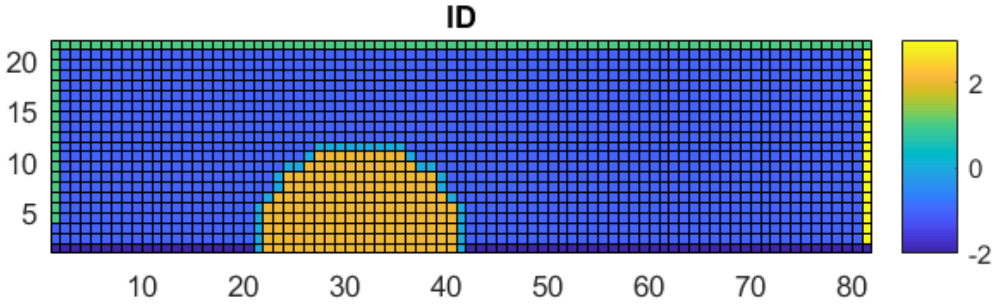



*Figure 1.7 – ID number positions of a further refined grid*

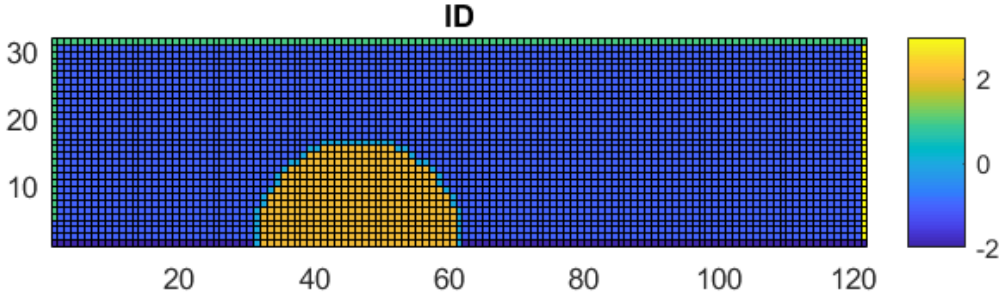



*Figure 1.8 – ID number positions of an increase of refinement*

## 1.4 Overview of potential flow theory

Potential flow is a special, idealized type of fluid which is the fundamental basis of panel and vortex methods. The flow is characterized as:

- Irrotational, $\nabla \times \vec{V} = 0$
- Incompressible, inviscid $\nabla \cdot \vec{V} = 0$
- Steady, $\frac{\partial V}{\partial t} = 0$

For flows at the boundary surfaces the condition known as the Kutta-condition must be satisfied [1]. This is where the normal component of velocity is zero, therefore, the total velocity vector must be completely tangential to the surface.

$$\vec{V} \cdot \hat{n} = 0 \tag{6}$$

This is known as a boundary condition which must be met in order to find a physically realistic solution [1]. Satisfying this boundary condition will be discussed in the results section.

### 1.4.1 Calculating the streamline function

The potential flow over the cylinder was solved using a central difference scheme for the Laplace equation of the stream function which is the governing equation for this analysis.

$$\nabla^2 \Psi = 0 \tag{7}$$

$$2D: \quad \frac{\partial^2 \Psi}{\partial x^2} + \frac{\partial^2 \Psi}{\partial y^2} = 0 \tag{8}$$

Equating the both spatial derivatives and solving for the central finite difference scheme leads:

$$\Psi_{i,j} = \frac{\Psi_{i-1,j} + \Psi_{i+1,j} + \Psi_{i,j+1} + \Psi_{i,j-1}}{4} \tag{9}$$

Boundary conditions were set using the identification matrix, which as mentioned previously, dictated which calculation was executed for the identified grid point. Furthermore, for the wall and symmetry line boundary condition, the $\Psi$ values were set at 1 and 0 respectively in accordance with potential flow theory. The different calculations are displayed below with their associated boundary condition and the near circle calculations are discussed in section 1.4.2. The values of $\Psi$ for every grid point were stored within a $\Psi$ matrix.

$$Outlet\ and\ inlet\ boundary: \quad \Psi_{i+1,j} = \Psi_{i-1,j} \tag{10}$$

$$Inlet: \qquad \Psi_{i,j} = \frac{2\Psi_{i+1,j} + \Psi_{i,j+1} + \Psi_{i,j-1}}{4} \tag{11}$$

$$Outlet: \qquad \Psi_{i,j} = \frac{2\Psi_{i-1,j} + \Psi_{i,j+1} + \Psi_{i,j-1}}{4} \tag{12}$$

### 1.4.2 Near circle calculations

For neighbouring points to the circle, the governing equation calculations were solved using a generalized equation (13) which accounts for the difference in grid spacing between the point and the circular geometry.

$$Near\ circle: \ \Psi_{i,j} = \frac{\frac{\Psi_a}{a(a+b)} + \frac{\Psi_b}{b(a+b)} + \frac{\Psi_c}{c(c+d)} + \frac{\Psi_d}{d(c+d)}}{\left(\frac{1}{ab} + \frac{1}{cd}\right)} \tag{13}$$

Where *a*, *b*, *c* and *d* are the arbitrary distances between the grid point and circular geometry in which related distances can be seen in figure 1.9. For the equation above, a significant error can occur due to the computer round off error and grid refinement. With increase in grid refinement, some of the arbitrary distances may become close enough to the circle circumference; where spacing becomes $(10^{-N})$ where $N \rightarrow \infty$ as grid refinement $\Delta x \rightarrow 0$. If any of the distances becomes zero, the solution will become infinite and in Matlab, produce "NaN" (Not a Number) for the associated grid point in the $\Psi$ matrix. Due to the nature of finite difference schemes, where values are calculated with respect to neighbouring values, not accounting for this error will essentially spread "NaN" values across the grid resulting in no solution at all. A conditional statement was added to avoid any division by zero and setting the respective point $\Psi$ value to 1, assuming that the point is close enough to be considered on the wall and therefore in accordance to the wall boundary condition.

The basis of calculations for *a*, *b*, *c* and *d* is Pythagoras theorem of the grid related parameters of the circle centre and others. This can be seen in figure 1.9 and the equations are displayed bellow:

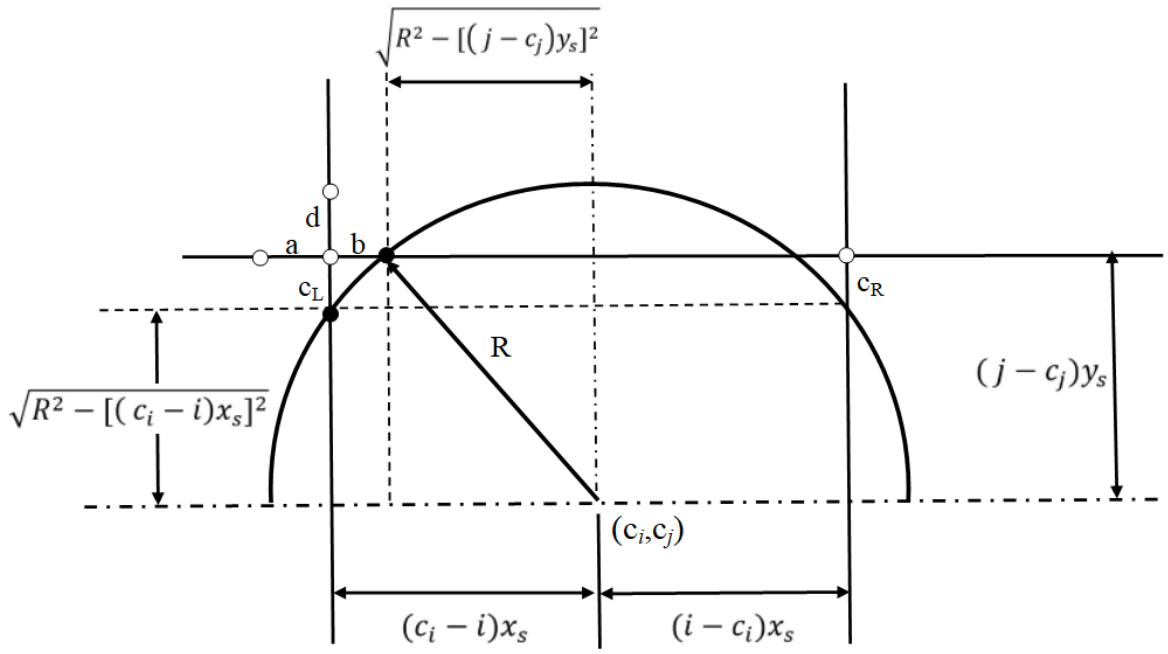



*Figure 1.9 – Circle geometry with characteristic distances for the arbitrary distance points $a, b, c, d$.*

$$a = (i - c_i)x_s - \sqrt{R^2 - \left[\left(j - c_j\right)y_s\right]^2} \tag{14}$$

$$b = (c_i - i)x_s - \sqrt{R^2 - \left[\left(j - c_j\right)y_s\right]^2} \tag{15}$$

$$c_L = \left(j - c_j\right)y_s - \sqrt{R^2 - [(\,c_i - i)x_s]^2} \tag{16}$$

$$c_R = \left(j - c_j\right)y_s - \sqrt{R^2 - [(\,i - c_i)x_s]^2} \tag{17}$$

$$d = y_s \tag{18}$$

Where $x_s, y_s$ are the grid spacing in the respective spatial dimension, $c_i, c_j$ are the coordinates for the circle centre, and $R$ the radius of the circle.

### 1.4.3 Calculating velocity from the stream function

In accordance with potential flow theory, calculating the velocity from the stream function consisted of using the equations (19) and (20) for the horizontal and vertical velocities:

$$u = \frac{\partial \Psi}{\partial y} \tag{19}$$

$$v = -\frac{\partial \Psi}{\partial x} \tag{20}$$

The forwards, backwards and central difference schemes for the horizontal velocity, $u$, are displayed below in equations (21), (22), and (23), which is dependent on the spatial dimension $j$. The vertical velocity, $v$, is likewise to these equations, however, is dependent on spatial dimension $i$.

$$u_{i,j,Central} = \frac{\Psi_{i,j+1} - \Psi_{i,j-1}}{2\Delta y} + \sigma(\Delta x^2) \tag{21}$$

$$u_{i,j,Forward} = \frac{\Psi_{i,j+1} - \Psi_{i,j}}{\Delta y} + \sigma(\Delta x) \tag{22}$$

$$u_{i,j,Backwards} = \frac{\Psi_{i,j} - \Psi_{i,j-1}}{\Delta y} + \sigma(\Delta x) \tag{23}$$

The central difference scheme was used for the velocity calculations in both spatial dimensions as it entails a higher order truncation error and therefore will provide results closer to the physical solution [2]. However, this excludes boundaries such as the symmetry line, wall etc. as there is no points outside the boundary in which the central difference scheme required. Instead, the forwards or backwards scheme were used in these locations.

## 1.5 Convergence

The root mean square of the difference between Ψ values of each cell were calculated and the convergence criterion for the simulation was deemed when this root mean square fell below a certain value, in this case, $10^{-3}$.

$$Residuals\ RMS = \sqrt{\frac{\sum \Psi_{i,j}^{n} - \Psi_{i,j}{}^{n-1}}{NP}} \tag{24}$$

Where $n$ denotes the iteration and $NP$ the total number of grid points. Furthermore, for the solution to be deemed converged, the change in the vertical velocity, $v$, must be zero at the outlet boundary; this was achieved by extending the outlet region.

## 1.6 Implementation into Matlab

The implementation into Matlab can be seen in the following flowchart in figure 1.10. Note, only the highest significant conditional statements are included.

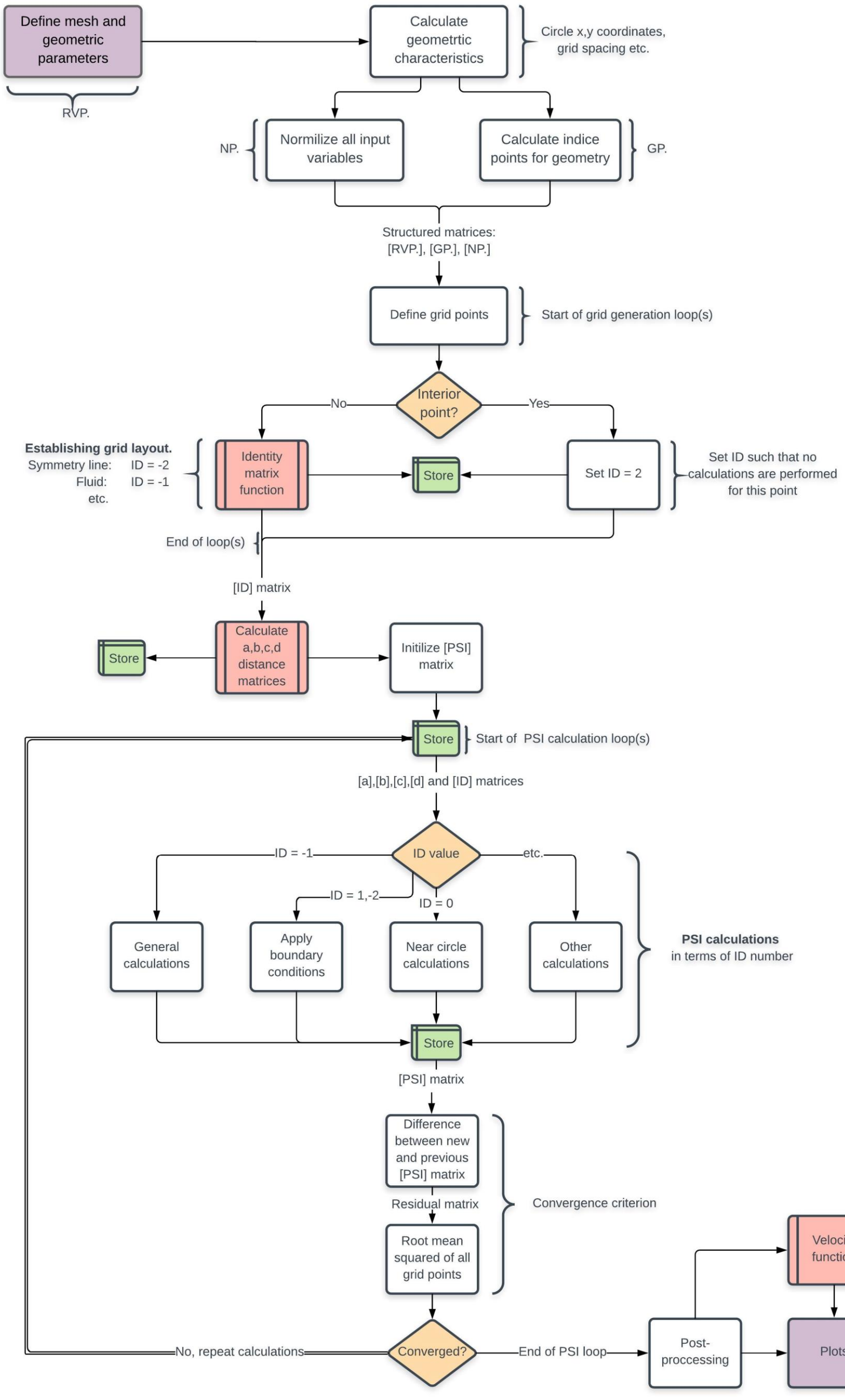



*Figure 1.10 – Flow chart of computational code used to simulate flow over a two-dimensional circle with potential flow.*

## 1.7 Results and comparison with potential flow theory

Three different grid refinements were considered for simulating the flow; the respective grid spacing, iterations required, and simulation times are displayed in table 1.2 below. In terms of computational expense, the increase in grid refinement drastically after the grid refinement of 0.1.

*Table 1.2 – Grid spacing, iterations and simulation time used*

| Grid spacing: $\Delta x, \Delta y$ | Iterations | Simulation time (s) |
|---|---|---|
| 0.25 | 113 | 2.4 |
| 0.1 | 551 | 5.7 |
| 0.05 | 1769 | 36.3 |

### 1.7.1 Physicality of the solution

For the simulation to be in accordance with potential flow theory, three primary characteristics of potential flow must be satisfied.

(1) Stagnation points, where the resultant velocity equates to zero, must be located before and after the circle on the furthest $x$ locations, and additionally at the external wall corner.
(2) The conservative nature of potential flow.
(3) The velocity at the solid boundaries must have normal component of velocity of zero ($v = 0$) in accordance with the Kutta-condition.
(4) The velocity at the top point is where the maximum velocity magnitude should occur.

*(1) Stagnation points*

The potential theory for the normal (or radial) and tangential components of velocity around a circle is defined below:

$$u_r = V_\infty cos\theta \left(1 - \frac{R^2}{r^2}\right) \tag{25}$$

$$u_\theta = -V_\infty \sin\theta \left(1 + \frac{R^2}{r^2}\right) \tag{26}$$

Where $\theta$ is the angle in relation to the flow direction, $R$ the radius of the circle, and $r$ the current radial position. On the circle surface, i.e. $r = R$ , the equations become:

$$u_r = 0 \tag{27}$$

$$u_\theta = -2V_\infty sin\theta \tag{28}$$

This satisfies the Kutta-condition where no normal component of velocity is present on the circle surface. Furthermore, the furthest point upstream and downstream of the circle is denoted by $0^o$ and $180^o$ in accordance with the flow direction (flow direction is parallel to the circle centreline). The stagnation points should lie on these points as $\sin(0^o) = \sin(180^o) = 0$, rendering $u_\theta = 0$.

The stagnation points before and after the circle were considered in terms of grid refinement. Figures 1.11, 1.12 and 1.13 display the velocity profiles with respect to the $y$ direction for each of the grid refinements before and after the circle through the theoretical stagnation point.

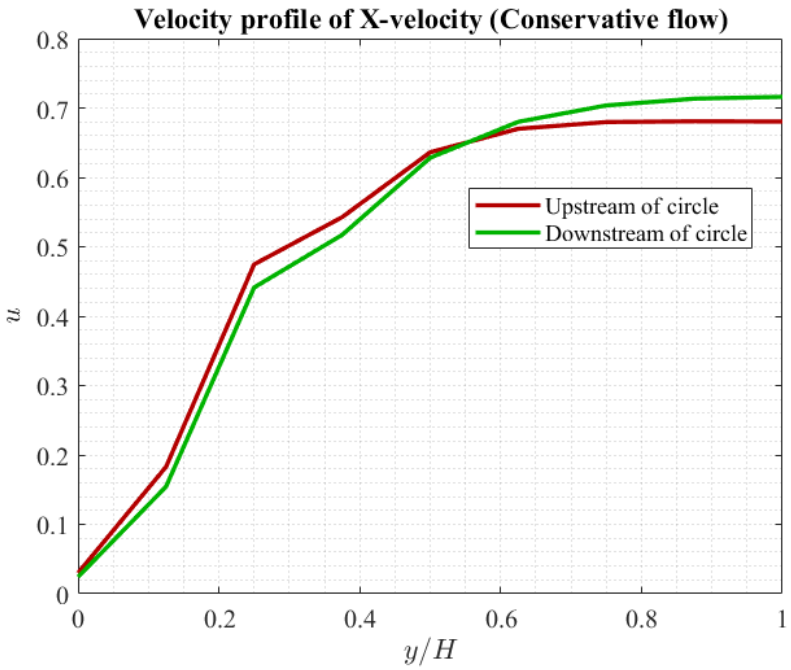



*Figure 1.11 – Velocity profile before and after circle for the horizontal velocity, $u$, for a grid spacing of 0.25.*

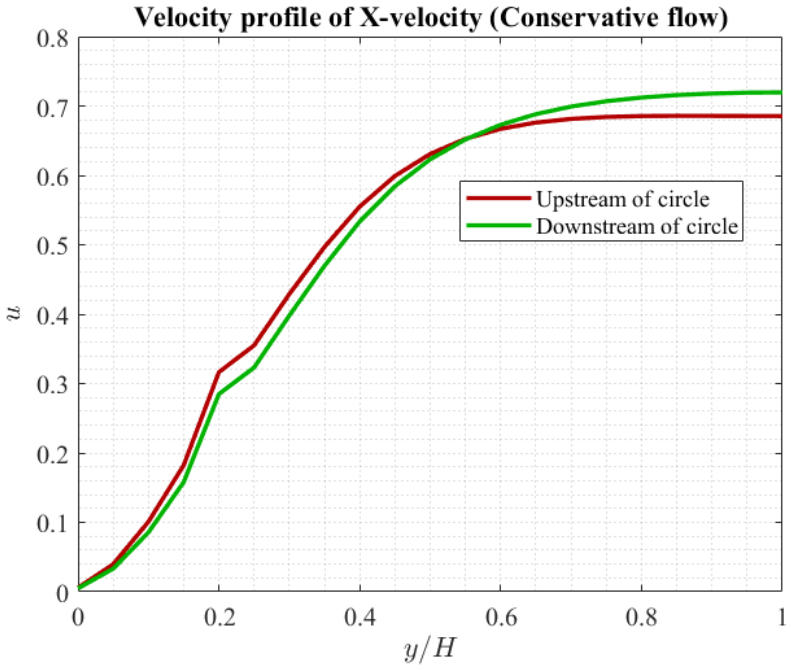



*Figure 1.12 – Velocity profile before and after circle for the horizontal velocity, $u$, for a grid spacing of 0.1.*

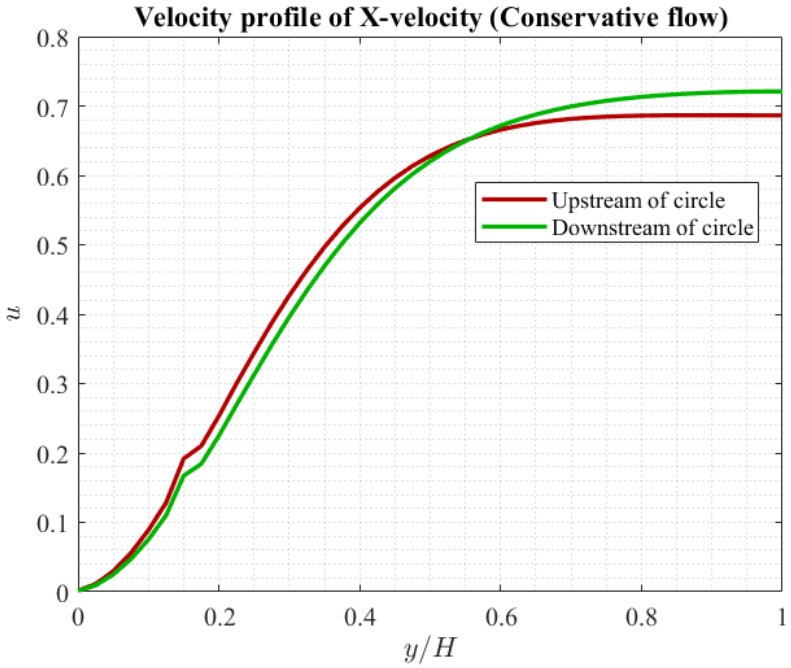



*Figure 1.13 – Velocity profile before and after circle for the horizontal velocity, $u$, for a grid spacing of 0.05.*

Significantly, the lower grid refinements do not produce zero velocity for the stagnation points, however, with increase in grid refinement the velocity tends closer to zero and therefore can be deemed to be approaching the correct physical solution with increase in grid refinement. This is due to the representation of the circular geometry becoming more accurate with the increase in refinement. The vertical velocity should also be zero at the stagnation points and since it is defined by the symmetry line which has the exact same boundary conditions as the surface cylinder, this was zero for all grid refinement settings. Furthermore, the stagnation point at the corner must be satisfied; figure 1.14 displays the vertical velocity profile at the inlet/wall over $y$ positions.

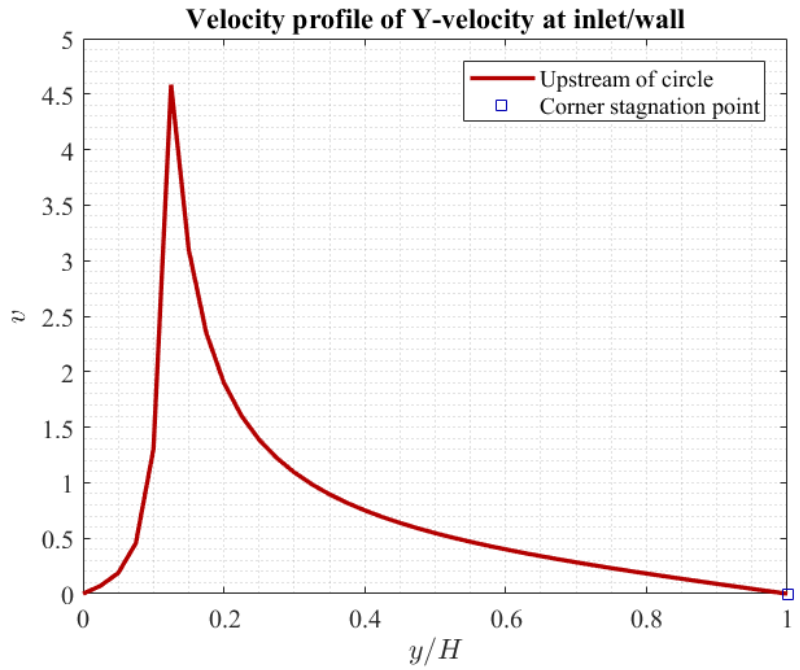



*Figure 1.14 – Velocity profile of vertical velocity at inlet/wall displaying the stagnation point at wall corner.*

On an additional note on the velocity profiles, sharp discontinuities in the velocity curves can be seen in the in the lower half of the discretized points in the $y$ direction. For a continuous circular curve defining the circle, the velocity profile plots will be continuous with no discontinuities. However, due to the grid refinement unable to account for the high gradients of the circular geometry in relation to the $y$ axis, the discretization in the $x$ direction is unable to capture the gradient accurately. This is less prominent with increase in grid refinement as can be seen in the velocity profile figures.

*(2) Conservative nature of potential flow*

Further in terms of satisfying potential flow theory, the flow can be seen to reduce in velocity downstream and in the circle "shadow". Since potential flow is conservative, i.e. no energy is lost, the decrease in flow velocity has a reactionary effect where the flow speeds above and downstream of the circle. If no energy is lost, the area under the velocity curve forward and aft of the circle should be equivalent. The percentage difference in areas can be seen in table 1.3 and with decrease in grid spacing, is evident that the solution approaches the conservative nature of potential flow.

*Table 1.3 – Grid spacing and difference in area under velocity-curves.*

| Grid spacing | Difference in area |
|---|---|
| 0.25 | 0.33% |
| 0.1 | 0.15% |
| 0.05 | 0.06% |

*(3) Kutta-condition*

In terms of the Kutta condition, the Ψ values at the point on top of the circle, before and after should be equivalent in order to produce a normal component of velocity of zero using either forwards, backwards or central difference. Table 1.4 shows the Ψ values of the points before and after the circle boundary point and their respective percentage difference; this displays the degree in which the simulation satisfies the Kutta-condition using the central difference scheme for normal velocity. Note, the Ψ values decrease with refinement as the points become closer to the circular boundary condition, $\Psi = 0$.

Kutta-condition for central difference in velocity at circle top:

$$v = -\frac{\Psi_{i+1,j} - \Psi_{i-1.j}}{2\Delta y} \approx 0 \tag{29}$$

*Table 1.4 – Central difference inspection on satisfying the Kutta-condition*

| Grid spacing | $\Psi_{i-1,j}$ | $\Psi_{i+1,j}$ | Difference |
|---|---|---|---|
| 0.25 | 0.06133 | 0.05918 | 3.72% |
| 0.1 | 0.01141 | 0.01126 | 1.33% |
| 0.05 | 0.00241 | 0.00240 | 0.42% |

Iterating on the physicality of the solution, the increase in grid refinement leads to a smaller normal velocity component, hence a closer approximation to the physical solution, however, a slight percentage difference in the $\Psi$ values will always be present as the discretization of the circle will never fully represent the continuous geometry.

*(4) Maximum velocity on top of the circle*

The highest velocity around the circle occurs at $\theta = 90^o$ which is accordance to potential flow theory. This is depicted in figure 1.15 of the velocity magnitude of the fluid domain Additionally, figures 1.16 → 1.19 display the colour plots of the horizontal, vertical velocity and stream function, further demonstrating the satisfaction maximum velocity and Kutta-condition.

$$u_{mag} = \left((u^2 + v^2)^{\frac{1}{2}}\right) \tag{30}$$

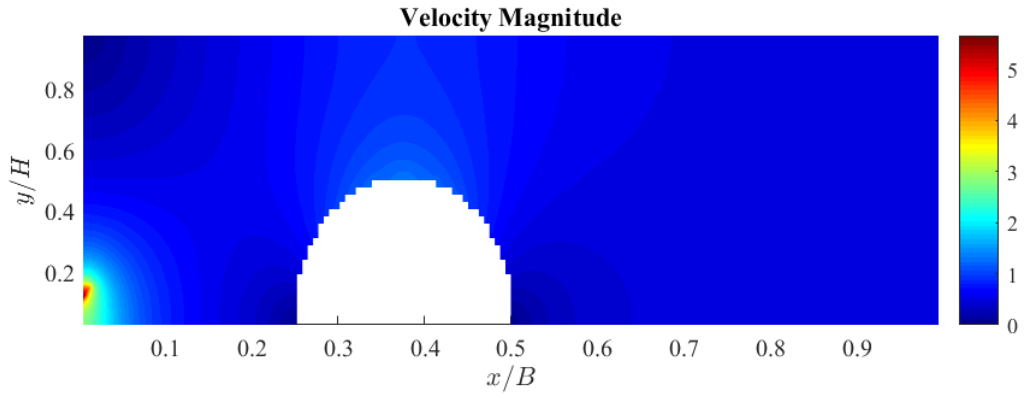



*Figure 1.15 – Velocity magnitude colour plot displaying flow over circle, where the maximum occurs at the circle top.*

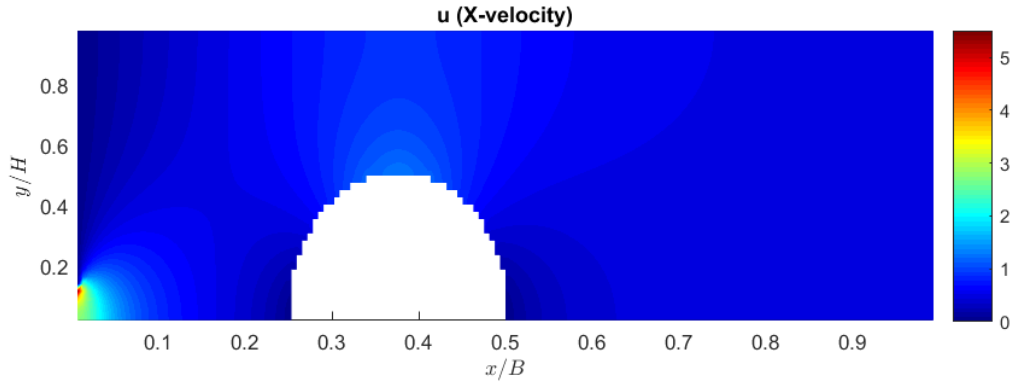



*Figure 1.17 – Horizontal velocity colour plot displaying flow over circle, where the maximum occurs at the circle top.*

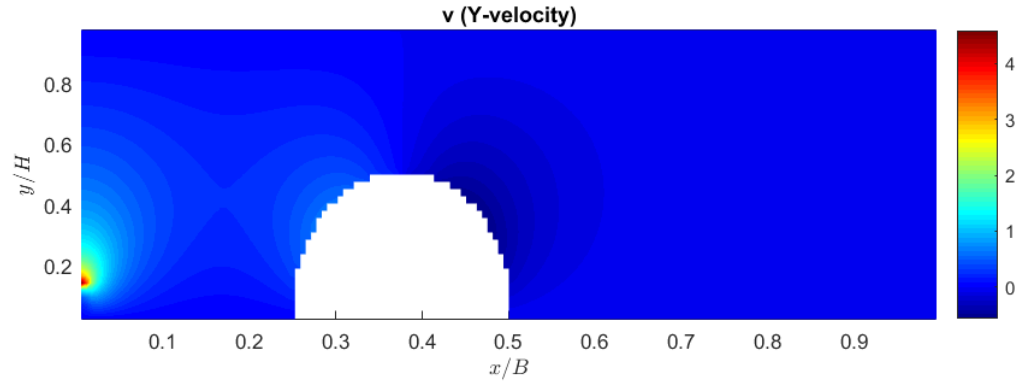



*Figure 1.18 – Vertical velocity colour plot displaying flow over circle, where the zero occurs at the circle top.*

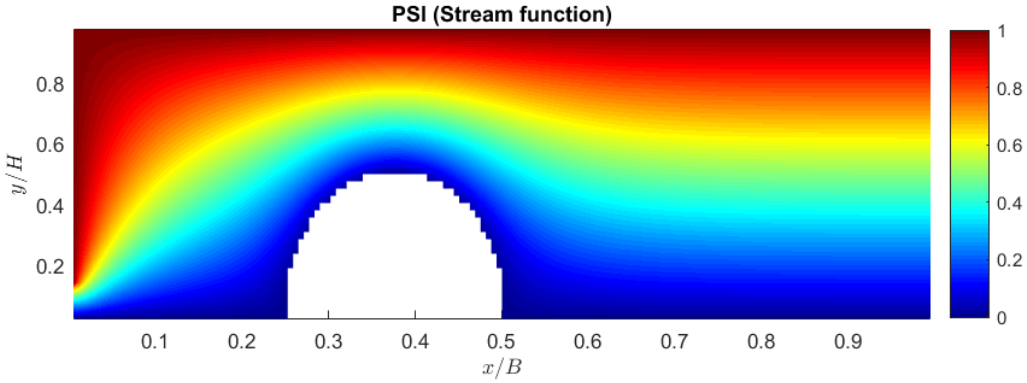



*Figure 1.19 – Stream function over entire domain.*

# Chapter 2:

# Investigating Finite Difference methods

## 2.1 One-sided second order finite difference approximation (Q1A)

*Derive a one-sided second order finite difference approximation for (ux)i using ui, ui+1, and ui+2. Assume equal spacing ∆x. Expand ui+1 and ui+2 about xi by Taylor series to the third order and manipulate the equations to cancel the second order terms and solve for (ux)i .*

The definition of a one-sided difference scheme, as the name suggests, is the usage of only points located on one side of an axis. The solution to this starts off with Taylor series expansion around the $u_{i+1}$ and $u_{i+2}$ and multiplying each equation by separate constant, *a* and *b*. Each Taylor series expansion has $1 \rightarrow N$ derivative of terms as can be seen in equation (31); the two expansions are added together and all of the *N* order terms combined to produce arbitrary equations of *a* and *b* before each order term.

$$u_{i+1} = a\left(u_i + \Delta x\frac{\partial u}{\partial x} + \frac{(\Delta x)^2}{2!}\frac{\partial^2 u}{\partial x^2} + \frac{(\Delta x)^3}{3!}\frac{\partial^3 u}{\partial x^3} + \sigma(\Delta x^4)\ldots\right) \tag{31}$$

$$u_{i+2} = b\left(u_i + 2\Delta x\frac{\partial u}{\partial x} + \frac{(2\Delta x)^2}{2!}\frac{\partial^2 u}{\partial x^2} + \frac{(2\Delta x)^3}{3!}\frac{\partial^3 u}{\partial x^3} + \sigma(\Delta x^4)\ldots\right) \tag{32}$$

The addition of both equations yields equation (33) below with all terms resolved and ignoring the 4th derivative:

$$a(u_{i+1}) + b(u_{i+2}) =$$

$$(a+b)u_i + (a+2b)\Delta x\frac{\partial u}{\partial x} + \left(\frac{a}{2} + 2b\right)(\Delta x)^2\frac{\partial^2 u}{\partial x^2} + \left(\frac{a}{6} + \frac{4b}{3}\right)(\Delta x)^3\frac{\partial^3 u}{\partial x^3} \tag{33}$$

The solution to creating a second order finite difference scheme arises by setting the constants such that the 2nd derivative is cancelled out. The results in:

$$\frac{a}{2} + 2b = 0 \tag{34}$$

$$a = -4b \tag{35}$$

Solving equation (33) for the 1st derivative and relative values constant equations and substituting in $a = -4b$ yields:

$$\frac{\partial u}{\partial x} = \frac{a(u_{i+1}) + b(u_{i+2}) - (a+b)u_i}{(a+2b)\Delta x} - \frac{\left(\frac{a}{2} + 2b\right)(\Delta x)^2}{(a+2b)\Delta x}\frac{\partial^2 u}{\partial x^2} - \frac{\left(\frac{a}{6} + \frac{4b}{3}\right)(\Delta x)^3}{(a+2b)\Delta x}\frac{\partial^3 u}{\partial x^3} \tag{36}$$

$$\frac{\partial u}{\partial x} = \frac{-4b(u_{i+1}) + b(u_{i+2}) + 3bu_i}{(-2b)\Delta x} - \frac{\left(\frac{-4b}{2} + \frac{4b}{2}\right)(\Delta x)^2}{(-2b)\Delta x}\frac{\partial^2 u}{\partial x^2} - \frac{\left(\frac{-4b}{6} + \frac{8b}{6}\right)(\Delta x)^3}{(-2b)\Delta x}\frac{\partial^3 u}{\partial x^3} \tag{37}$$

Setting arbitrary constant to $b = 1$ results in the final solution:

$$\frac{\partial u}{\partial x} = \frac{4(u_{i+1}) - (u_{i+2}) - 3u_i}{2\Delta x} + \sigma(\Delta x^2) \tag{38}$$

## 2.2 Numerical parameters of finite difference (Q1B)

*Given the function u(x)=cos(πx), find ux(0.25) using first order forward difference and second order central difference with equal spaces of 0.25, 0.1 and 0.01. Compare with the exact result and discuss the influence of the scheme order and the space size on accuracy.*

Similar to the discussion in section 1.2.1, this question entails the comparison of gradients created by each scheme and the exact solution. However, the effect of grid spacing will be also be explored. The function under consideration is the cosine function, which, similarly to the sine function, exhibits strong geometric curvature and linearity in different locations of $x$ and therefore forms a good basis of finite difference scheme performance analysis. The exact solution (continuous) can be seen in equation (39), and the discretized points for $u_i$ , $u_{i+1}$ and $u_{i-1}$ are displayed in equations (40), (41) and (42) respectively.

$$u(x) = \cos(\pi x) \tag{39}$$

$$u_i = \cos(\pi . dx_i) \tag{40}$$

$$u_{i+1} = \cos(\pi . dx_{i+1}) \tag{41}$$

$$u_{i-1} = \cos(\pi . dx_{i-1}) \tag{42}$$

Where $dx$ is the discretized locations of $x$ with respect to the grid spacing under analysis ($dx = 0, \Delta x, 2\Delta x, \dots, 2\pi$) and $i$ is the index number which establishes the current evaluation point and associated value of discretized $x$. The exact equation, forward and central scheme equations for the 1st derivative (gradient) are depicted below:

$$\frac{\partial u}{\partial x} = -\pi \ \sin(\pi x) \qquad \text{Exact solution} \tag{43}$$

$$\frac{\partial u}{\partial x} = \frac{u_{i+1} - u_i}{\Delta x} + \sigma(\Delta x) \qquad \text{Forward difference scheme} \tag{44}$$

$$\frac{\partial u}{\partial x} = \frac{u_{i+1} - u_{i-1}}{2\Delta x} + \sigma(\Delta x^2) \qquad \text{Central difference scheme} \tag{45}$$

Figure 1.3 shows an exemplary domain with the exact solution and the central and forwards difference scheme gradients and table 2.1 shows the gradient resultants of central, forwards and exact for $u(0.25)$.

For grid spacing of $\Delta x = 0.1$, interpolation was needed to find the gradient at $u(0.25)$. This resulted in a small error in the values as interpolation is an approximation method; since interpolation between two points is linear, the error was assumed to be linear and scaled out by using the known exact value over the errored exact value multiplied by the scheme values to give a more accurate answer.

$$\frac{(du/dx)_{Exact}}{(du/dx)_{Exact,errored}} (du/dx)_{Exact,errored} = \left(\frac{du}{dx}\right)_{Exact} \tag{46}$$

$$\frac{(du/dx)_{Exact}}{(du/dx)_{Exact,errored}} (du/dx)_{Scheme} = \left(\frac{du}{dx}\right)_{Scheme,corrected} \tag{47}$$

*Table 2.1 – u (0.25) gradient values for forwards, central difference schemes and exact for different grid spacings.*

| $\Delta x$ | $(du/dx)_{Forward}$ | $(du/dx)_{Central}$ | $(du/dx)_{Exact}$ |
|---|---|---|---|
| 0.25 | -2.8284 | -2.0000 | -2.2214 |
| 0.1 | $-2.5311^{C}$ | $-2.1850^{C}$ | $-2.2214^{C}$ |
| 0.01 | -2.2560 | -2.2211 | -2.2214 |

*subscript $^{C}$ denotes corrected

There is a significant discrepancy for both schemes with a course discretization, however, the finite difference approximation improves as grid spacing decreases. The higher discrepancy of the forward scheme to the exact solution will be due to the lower order of the truncation error of the scheme; where the truncation is characterises the measure of discrepancy of the discrete to the exact solution [2].

### 2.2.1 Additional physical considerations of the finite difference schemes

Figure 2.1 and 2.2 below depicts the gradients of the central and forward difference schemes respectively for grid spacing $\Delta x = 0.25, 0.1$. The graphs further depict the area under the gradient curves and the substantial differences in areas under the curves. This is physically important as, for example, the area under a velocity-time graph depicts the distance travelled. This example is suited for this problem as the 1st derivative of distance is of course, velocity, and this can be discretized in time; therefore, the under or over predictions of area between discretized points is significant.

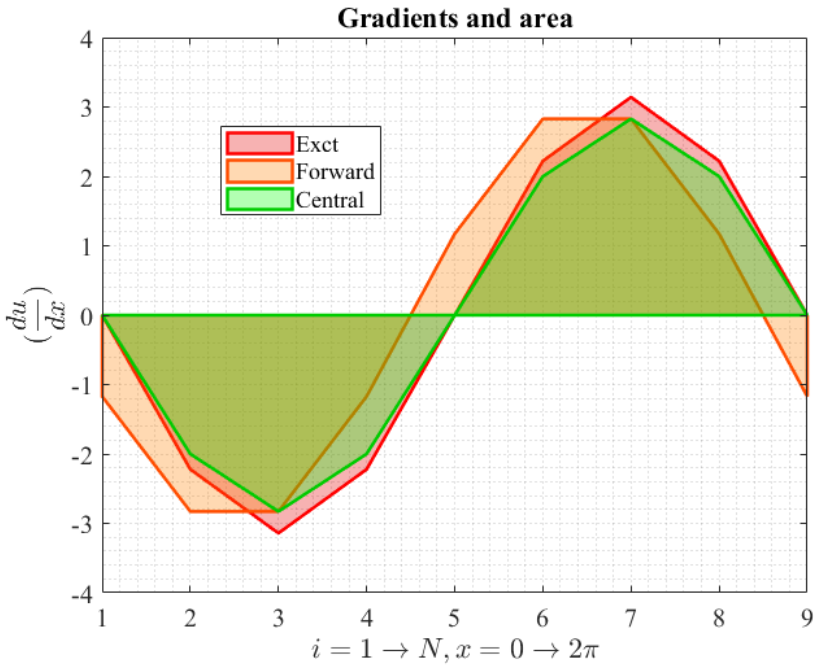



*Figure 2.1 – Gradient and area of exact, forward and central difference schemes, where the exact values are relative to the discretized points with grid spacing,* $\Delta x = 0.25$.

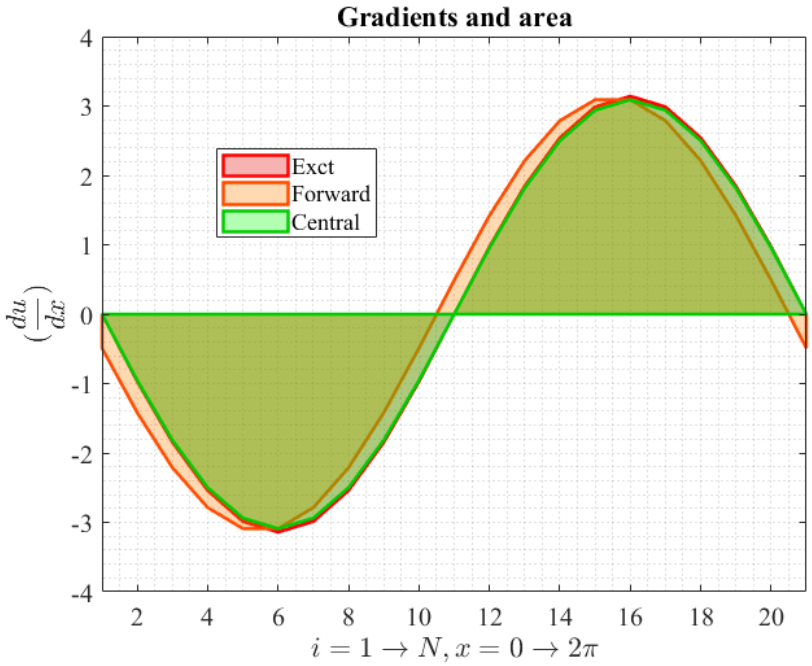



*Figure 2.2 – Gradient and area of exact, forward and central difference schemes, where the exact values are relative to the discretized points with grid spacing,* $\Delta x = 0.1$.

Referring to figure 2.2 and considering $\frac{\partial u}{\partial x}$ to be velocity and $i$ to be discretization in time, using the forwards difference in time $(i = t)$ with a grid spacing of 0.25 results in the distance travelled to be significantly overpredicted between $1 < i < 2$ and underpredicted between points $4 < i < 5$. Numerically, for the scheme and grid spacing used, the area between $4 < i < 5$ is approximately zero which suggests the velocity is zero despite a non-zero, sign-symmetric gradient for the evaluation points. This is displayed in figure 2.3.

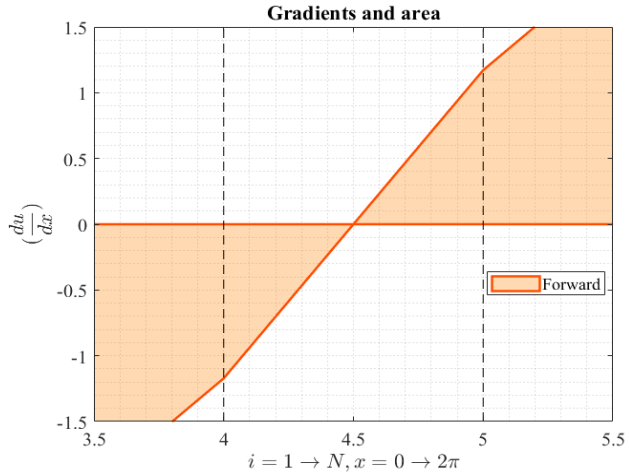



*Figure 2.3 – Area between two evaluation points, 4 and 5, for the forwards difference scheme*

This error creates a non-physicality and is a result of phase difference between the exact and approximated solution which is known as the dispersion error and is essentially the phase difference between the exact solution and the approximate [1]. This error causes the function to intersect the x-axis between the evaluation point and it's used neighbour.

As grid spacing decreases, the phase difference decreases; however, the 1st derivative function will always intersect the x-axis between two evaluation points for the forward difference. This is highlighted in figure 2.4, which entails the segmented area ratio between the evaluation points of the forwards difference and exact gradients for decreasing grid spacings $\Delta x = 0.25, 0.1, 0.01$ and $0.001$ to highlight that the physicality is independent of grid spacing.

$$A_{ratio} = \frac{A_{Exact} - A_{scheme}}{A_{Exact}} \tag{48}$$

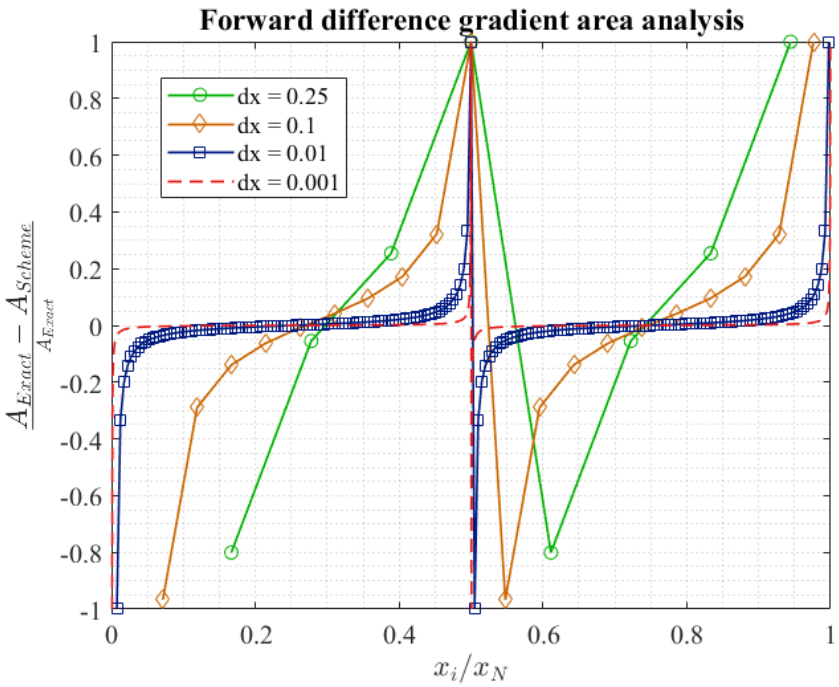



*Figure 2.4 – Area ratio between the exact and forward difference scheme solution, where $A_{ratio} = 1$ indicates the $A_{scheme} = 0$. This graph is for multiple grid refinements $\Delta x = 0.25, 0.1, 0.01$ and $0.001$.*

Where $A_{ratio}$ equals unity when the area between the points results in zero due to the intersection between the points. This conclusion highlights the constraints on usage of the forwards difference scheme on certain functions or on a more physical note, on physical systems exhibiting sign-changing characteristics such as flow separation/reversal. The central difference however, has no phase change and therefore $A_{Scheme}$ never equals zero as seen in figure 2.5.

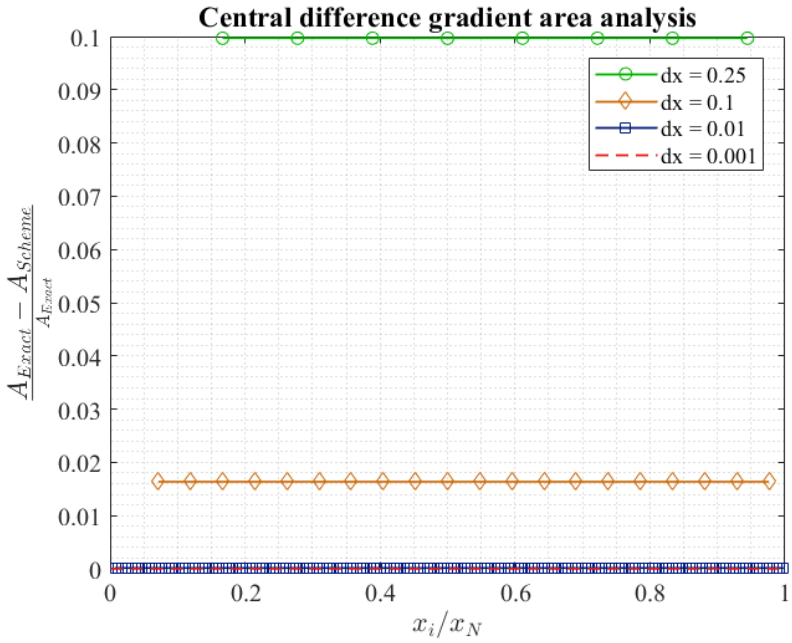



*Figure 2.5 – Area ratio between the exact and central difference scheme solution, where $A_{ratio} = 1$ indicates the $A_{scheme} = 0$. This graph is for multiple grid refinements $\Delta x = 0.25, 0.1, 0.01$ and $0.001$.*

Note, each point of the larger grid spacing equals multiple points of the smaller grid spacings, hence the difference in starting points of each line.

## 2.3 Stability analysis of the Upwind scheme (Q1C)

*Derive an upwind scheme using forward-time forward-space discretisation for the linear convection equation when a < 0. What is the truncation error, the order and the stability condition of the numerical scheme?*

The linear advection equation is a factorised form of the partial differential wave equation in one-dimensional space. The wave equation is used

$$\frac{\partial^2 u}{\partial t^2} = a^2 \nabla^2 u \qquad \text{Wave equation} \qquad (49)$$

$$\frac{\partial^2 u}{\partial t^2} = a^2 \frac{\partial^2 u}{\partial x^2} \qquad \text{Wave equation (1D)} \qquad (50)$$

The objective of this question is to establish the truncation error and stability of the system with $a < 0$. The wave equation is a partial differential equation (PDE) and the type of which can be established by considering the general second order PDE equation [3]:

$$A(x,t)\frac{\partial^2 u}{\partial x^2} + B(x,t)\frac{\partial^2 u}{\partial x \partial t} + C(x,t)\frac{\partial^2 u}{\partial t^2} = F\left(x, t, \frac{\partial u}{\partial x}, \frac{\partial u}{\partial t}\right) \qquad (51)$$

The conditions for which type of PDE are displayed below [3], [4].

$B^2 - 4AC = 0$ Parabolic (52)

$B^2 - 4AC < 0$ Elliptic (53)

$B^2 - 4AC > 0$ Hyperbolic (54)

The wave equation is hyperbolic regardless of the value of $a$ since it is in the form $a^2$, therefore, the solution to the equation at a point $(x, t)$ will have two real characteristics [3], one of which is the solution to the linear advection equation. Solving for the linear advection equation:

$$\frac{\partial^2 u}{\partial t^2} - a^2 \frac{\partial^2 u}{\partial x^2} = 0 \tag{55}$$

$$\left(\frac{\partial}{\partial t} - a\frac{\partial}{\partial x}\right)\left(\frac{\partial}{\partial t} + a\frac{\partial}{\partial x}\right)u = 0 \tag{56}$$

$$\therefore \ \frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} = 0 \quad \text{Linear advection equation} \tag{57}$$

The Upwind scheme of the Linear advection equation is discretized equation based on the sign of the propagation speed of the wave, $a$, in which determines the direction of the wave; on a physical note, the flow direction. The scheme uses the neighbouring point is "ahead" or "upwind" of the evaluation point, which results in the scheme having two forms depending on the sign of $a$.

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \begin{Bmatrix} -a\dfrac{u_i^n - u_{i-1}^n}{\Delta x}, a > 0 \\ -a\dfrac{u_{i+1}^n - u_i^n}{\Delta x}, a < 0 \end{Bmatrix} \tag{58}$$

This assignment will focus on $a < 0$, i.e. the wave will traverse right to left, and the resultant equation for the proceeding function value $u_i^{n+1}$ is displayed below:

$$u_i^{n+1} = u_i^n - a\frac{\Delta t}{\Delta x}(u_{i+1}^n - u_i^n) \tag{59}$$

Where $a\frac{\Delta t}{\Delta x}$ is the Courant Friedrichs-Lewy factor (CFL) [1], which is an important constant for stability analysis.

This is an explicit scheme in time as the next time layer is based only on known previous function quantities in time. The opposite to this would be an implicit scheme, which would require simultaneous equations to solve the multiple unknown variables in the proceeding time layer [5]. The difference between the two schemes is the simplicity and stability, where an explicit scheme requires significantly smaller time steps than the implicit in order to maintain stability, however, it also significantly easier to code [1]. This scheme is also an example of a time-marching scheme, where the solution is moved forward in time by a time step selected in accordance to the stability criterion which is expected to be small due to the explicit nature of the scheme.

### 2.3.1 Stability analysis

A numerical scheme is considered stable when the errors such as truncation and computer round off errors do not grow per iteration. Stability analysis is common in time marching schemes where the scheme is sequenced, or marched, in time towards the converged solution [1]. Von-Neumann stability analysis was used to establish the stability criterion of the Upwind scheme. The objective of this analysis is to establish the amplification factor of the numerical scheme, $G_f$, which is equivalent to equation (60) and should be less than 1 for stability, i.e. the errors do not grow.

$$|G_f| = -1 \leq \left|\frac{G^{n+1}}{G^n}\right| \leq 1 \tag{60}$$

Transforming the discrete equation from the real space to Fourier space, $u_i^n$ becomes:

$$u_i^n = G(t)e^{I\beta x_i} \tag{61}$$

Where $G(t)$ is the amplitude at a certain time, $\beta$ the wave number in which is defined as $\frac{2\pi}{\lambda_m}$ where $\lambda_m$ is the wavelength, $I$ the imaginary component $\sqrt{-1}$ , and $x_i$ is the spatial evaluation point. Substituting this form into the discrete equation (59) results in:

$$G^{n+1}e^{I\beta x_i} = G^n e^{I\beta x_i} - a\frac{\Delta t}{\Delta x}\left(G^n e^{I\beta(x_i+\Delta x)} - G^n e^{I\beta x_i}\right) \tag{62}$$

Factorising and re-arranging for the amplification factor form:

$$\left|\frac{G^{n+1}}{G^n}\right| = 1 - a\frac{\Delta t}{\Delta x}\left(e^{I\beta(\Delta x)} - 1\right) \tag{63}$$

In order to solve this equation in the real space, one must consider the square of the amplification factor [4] since the trigonometric identity of $e^{I\beta(\Delta x)}$ is within the complex plane, $e^{I\beta(\Delta x)} = \cos(\beta\Delta x) + isin(\beta\Delta x)$. Since the limits of the unsquared amplification ratio is between $-1 \leq \left|\frac{G^{n+1}}{G^n}\right| \leq 1$; negative amplifications factors can occur and the square of any negative value becomes positive, therefore, the limits of $\left|G_f\right|^2$become:

$$0 \leq \left|\frac{G^{n+1}}{G^n}\right| \leq 1 \tag{64}$$

Re-arranging and squaring equation (64):

$$\left|G_f\right|^2 = \left[\left(1 + a\frac{\Delta t}{\Delta x}\right) - a\frac{\Delta t}{\Delta x}e^{I\beta(\Delta x)}\right]^2$$

$$= \left[\left(1 + a\frac{\Delta t}{\Delta x}\right) - a\frac{\Delta t}{\Delta x}e^{I\beta(\Delta x)}\right]\left[\left(1 + a\frac{\Delta t}{\Delta x}\right) - a\frac{\Delta t}{\Delta x}e^{-I\beta(\Delta x)}\right]$$

$$= \left(1 + a\frac{\Delta t}{\Delta x}\right)^2 - a\frac{\Delta t}{\Delta x}(e^{I\beta(\Delta x)} + e^{-I\beta(\Delta x)}) + \left(a\frac{\Delta t}{\Delta x}\right)^2 (e^{I\beta(\Delta x)}e^{-I\beta(\Delta x)})$$

Which reduces to equation (65) by substituting the trigonometric expression, $2\cos(\beta\Delta x) = (e^{I\beta(\Delta x)} + e^{-I\beta(\Delta x)})$ and factorising:

$$\left|G_f\right|^2 = 1 + 2a\frac{\Delta t}{\Delta x}\left(1 + a\frac{\Delta t}{\Delta x}\right)\left[1 - \cos\left(\beta(\Delta x)\right)\right] \tag{66}$$

Considering that for this form of the Upwind scheme, $a = -a_{absolute}\ (= -a_{abs})$, the equation becomes:

$$\left|G_f\right|^2 = 1 - 2a_{abs}\frac{\Delta t}{\Delta x}\left(1 - a_{abs}\frac{\Delta t}{\Delta x}\right)\left[1 - \cos\left(\beta(\Delta x)\right)\right] \tag{67}$$

Therefore,

$$0 \leq \left[2a_{abs}\frac{\Delta t}{\Delta x}\left(1 - a_{abs}\frac{\Delta t}{\Delta x}\right)\left[1 - \cos\left(\beta(\Delta x)\right)\right]\right] \leq 1 \tag{68}$$

The term $\left[1 - \cos\left(\beta(\Delta x)\right)\right]$ has limits where $\cos\left(\beta(\Delta x)\right)$ will never surpass unity [6], thus the stability condition is:

$$0 \leq \left(1 - a_{abs}\frac{\Delta t}{\Delta x}\right) \leq 1 \tag{69}$$

$$\therefore\ a_{abs}\frac{\Delta t}{\Delta x} \leq 1 \tag{70}$$

This is known as the Courant condition for hyperbolic equations [1]. Further noting, that if $a$ was positive instead of negative ($a = a_{absolute}$ ), the scheme would become a "downwind" scheme. This scheme would become unconditionally unstable as the condition would become:

$$0 \leq \left(1 + a_{abs}\frac{\Delta t}{\Delta x}\right) \leq 1 \tag{71}$$

Since $\Delta t$ or $\Delta x$ can never be negative, the scheme would have no stable stability condition. This would also be the case for the other form of upwind scheme where if the direction of the wave inversed, therefore, this suggests that the upwind technique is essential for stability.

### 2.3.2 Validation of stability criterion

Testing of this scheme was performed in Matlab and in order to test the scheme, initial conditions were required. Only one initial condition is required as the time derivative is in the first order [4] and the Gaussian pulse was used due to the smooth and well-defined characteristics. Figure 2.6 displays a two-dimensional example of the Gaussian pulse. No constant source was applied, therefore, for stable stability the Gaussian pulse is expected to decrease in amplitude over time.

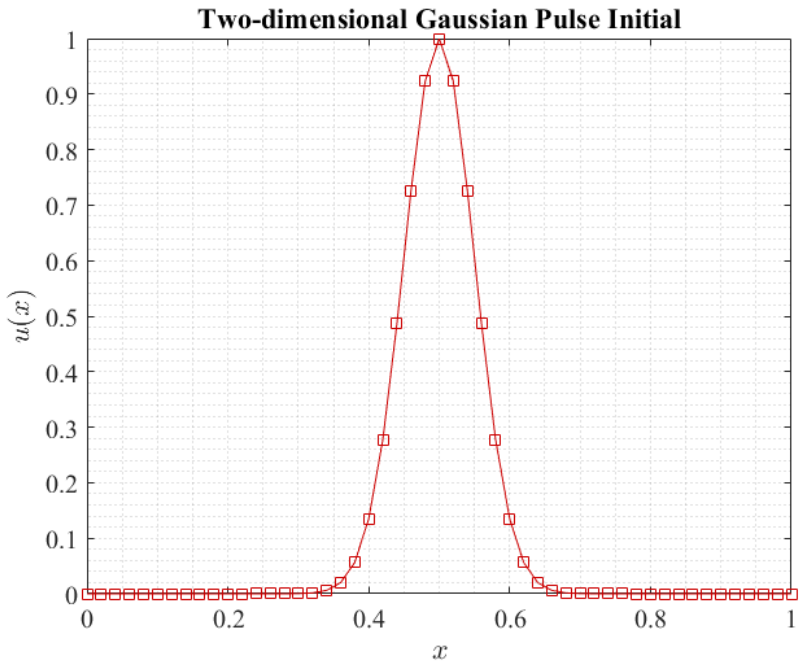



*Figure 2.6 – Two-dimensional Gaussian pulse for initial condition*

### 2.3.3 Boundary conditions

Periodic boundary conditions were applied to the domain to enable the development of the wave to be analysed in a fraction of the grid that would be required without the boundary conditions, in which in terms of computational costs, is significantly more efficient compared to storing a large quantity of grid points. The domain is $0 \rightarrow L$ where $L = 1$ and therefore the periodic boundary conditions become:

$$u(L,t) = u(0,t) \tag{73}$$

Furthermore, since the wave is travelling in one direction, only one periodic boundary condition is required. The discretized form of the boundary condition entails the finite difference scheme applied with modified point relations; this is displayed in equation (74) below with figure 2.7 highlighting the point relations.

$$u_N^{n+1} = u_1^n - a\frac{\Delta t}{\Delta x}(u_2^{n-1} - u_1^{n-1}) \tag{74}$$

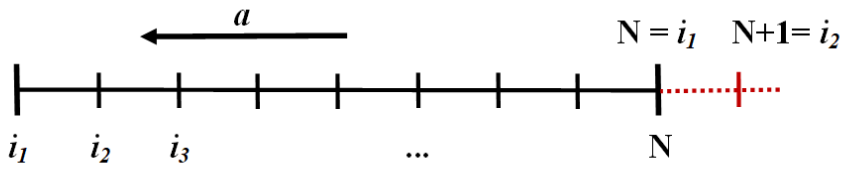



*Figure 2.7 – Point relations for discretized periodic boundary condition*

### 2.3.4 Convection equation stability analysis results

Contour and three-dimensional plots were used to clearly depict the wave development over time which were of the following stability criterion:

- Stable stability criterion of $a\frac{\Delta t}{\Delta x} \leq 1$:
- Verge of instability, $a\frac{\Delta t}{\Delta x} = 1$:
- Stability criterion over the stability threshold, $a\frac{\Delta t}{\Delta x} = 1.1$
- Stability criterion further over stability threshold, $a\frac{\Delta t}{\Delta x} = 1.2$

For stable stability criterion of $a\frac{\Delta t}{\Delta x} \leq 1$:

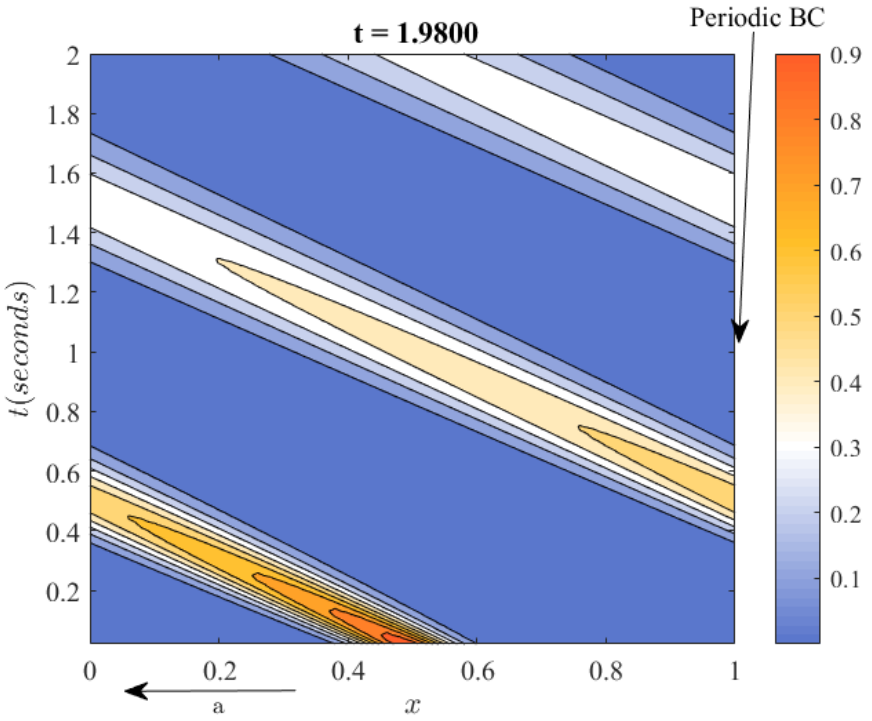



*Figure 2.8 – Contour plot displaying the development of the Gaussian pulse over time for* $a\frac{\Delta t}{\Delta x} \leq 1$

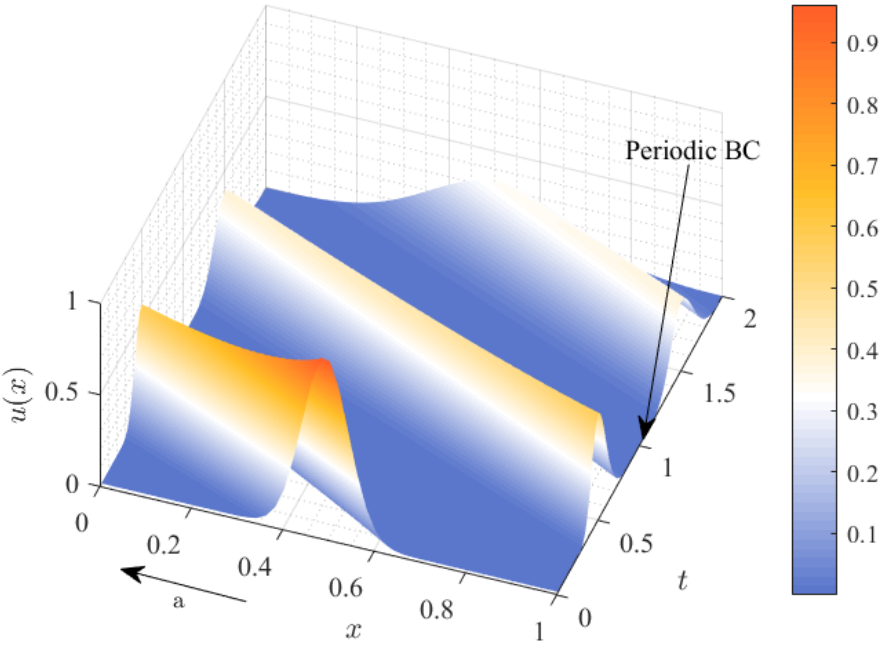



*Figure 2.9 – Three-dimensional plot displaying the development of the Gaussian pulse over time for* $a\frac{\Delta t}{\Delta x} \leq 1$

Stability criterion which can be considered on the verge of instability, $a\frac{\Delta t}{\Delta x} = 1$:

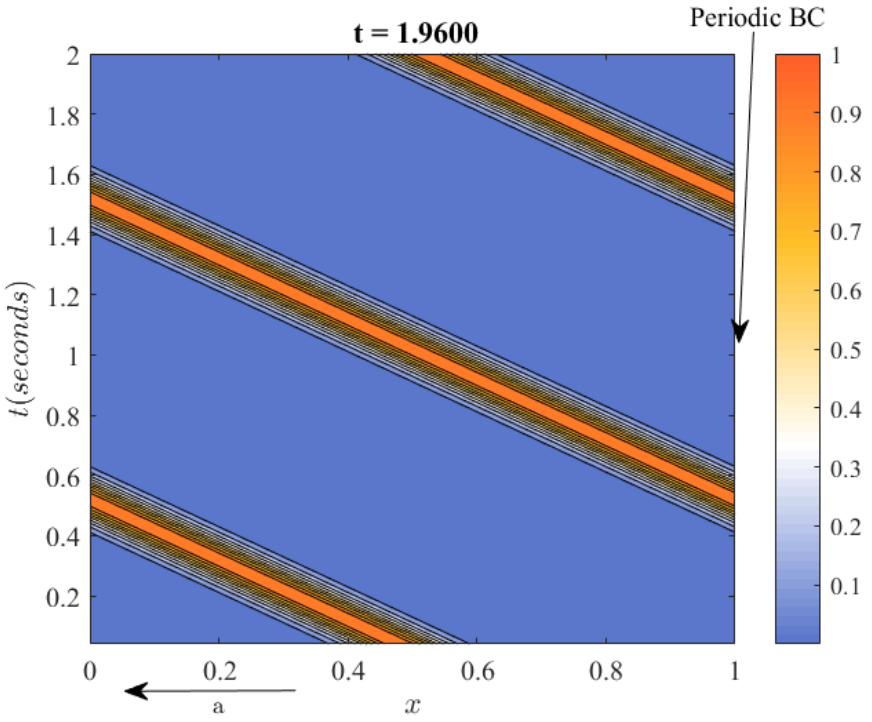



*Figure 2.10 – Contour plot displaying the development of the Gaussian pulse over time for* $a\frac{\Delta t}{\Delta x} = 1$

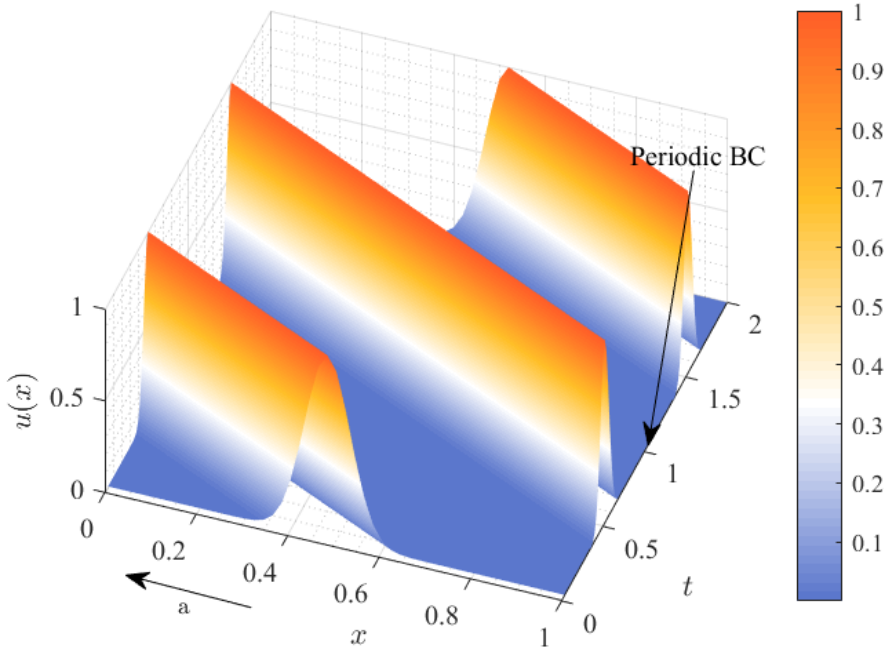



*Figure 2.11 – Three-dimensional plot displaying the development of the Gaussian pulse over time for* $a\frac{\Delta t}{\Delta x} = 1$

Stability criterion over the stability threshold, $a\frac{\Delta t}{\Delta x} = 1.1$

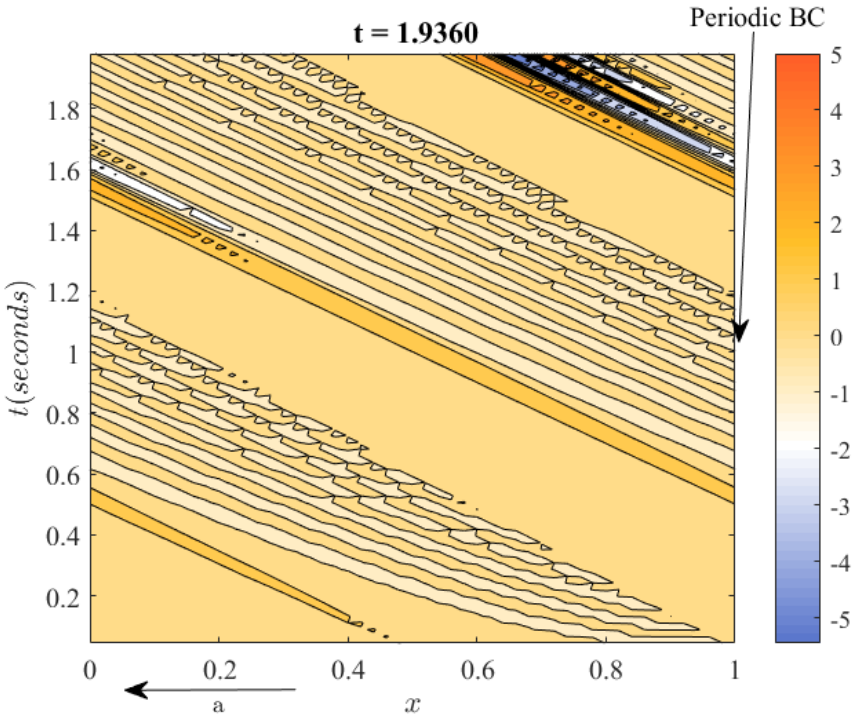



*Figure 2.12 – Contour plot displaying the development of the Gaussian pulse over time for* $a\frac{\Delta t}{\Delta x} = 1.1$

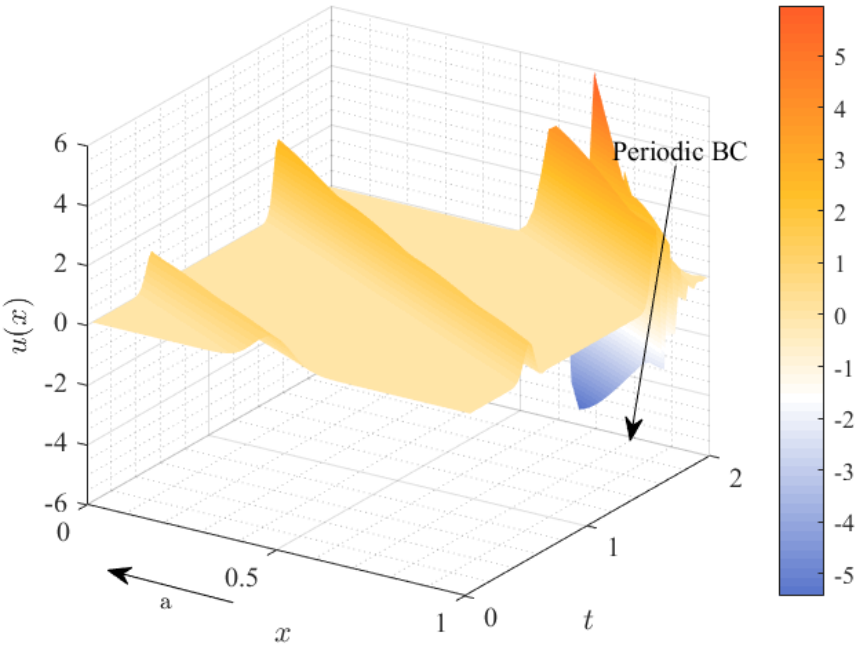



*Figure 2.13 – Three-dimensional plot displaying the development of the Gaussian pulse over time for* $a\frac{\Delta t}{\Delta x} = 1.1$

Stability criterion further over stability threshold, $a\frac{\Delta t}{\Delta x} = 1.2$

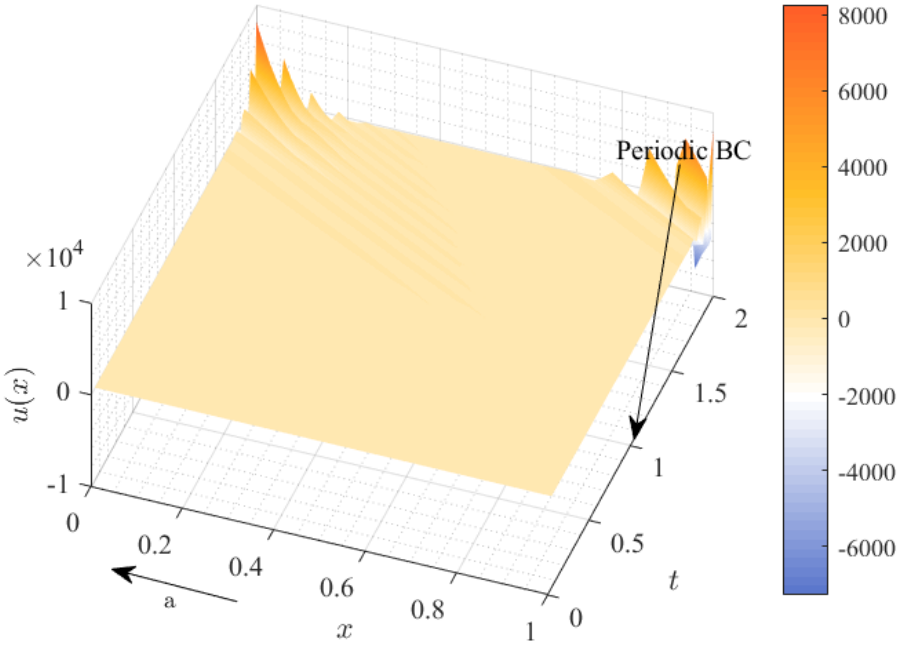



*Figure 2.14 – Three-dimensional plot displaying the development of the Gaussian pulse over time for* $a\frac{\Delta t}{\Delta x} = 1.2$.

Each stability condition follows the expected results:

- Stable condition of $a\frac{\Delta t}{\Delta x} \leq 1$ constantly decreasing in amplitude as the initial condition is not a source, and therefore the wave dissipates over time.
- Verge condition of $a\frac{\Delta t}{\Delta x} = 1$ maintaining a constant amplitude over time
- Unstable condition of $a\frac{\Delta t}{\Delta x} \geq 1$ constantly increasing and eventually decomposing the wave form, even at seemingly insignificant values above the stability criterion.

Therefore, the stability criterion derived from Von Neumann's theory can be considered valid for this case.

## 2.4 Forwards time, central space schemes (Q1D)

*For the linear convection equation, derive a numerical scheme using forward-time central space finite difference discretisation. What is the order of the scheme? Prove that the scheme is unconditionally unstable.*

The forwards difference, central time scheme is typically used in textbooks and papers to model instability of numerical schemes. Discretizing the linear convective equation with the respective schemes and solving for the next time layer results in:

$$u_i^{n+1} = u_i^n - a\frac{\Delta t}{\Delta x}\frac{(u_{i+1}^n - u_{i-1}^n)}{2} \qquad \text{FTCS} \qquad (75)$$

Using Von Neumann's stability criterion as in section 2.3.1, the stability derives as:

$$G^{n+1}e^{I\beta x_i} = G^n e^{I\beta x_i} - a\frac{\Delta t}{\Delta x}\frac{\left(G^n e^{I\beta(x_i+\Delta x)} - G^n e^{I\beta(x_i-\Delta x)}\right)}{2}$$

$$G^{n+1} = G^n\left(1 - a\frac{\Delta t}{\Delta x}\frac{\left(e^{I\beta(\Delta x)} - e^{I\beta(-\Delta x)}\right)}{2}\right)$$

$$|G_f| = \left(1 - a\frac{\Delta t}{\Delta x}\frac{\left(e^{I\beta(\Delta x)} - e^{I\beta(-\Delta x)}\right)}{2}\right)$$

$$|G_f| = \left(1 - a\frac{\Delta t.\, i}{\Delta x}\sin(\beta\Delta x)\right) \qquad (76)$$

To transform $|G_f|$ to the real space from the complex plane, the amplification factor is again, squared, and the limits of stability modified as in the previous section.

$$|G_f|^2 = \left(1 - a\frac{\Delta t.\, i}{\Delta x}\sin(\beta\Delta x)\right)^2 \tag{77}$$

Which results in:

$$|G_f|^2 = 1 + a^2\left(\frac{\Delta t}{\Delta x}\right)^2 \sin^2(\beta\Delta x) \tag{78}$$

Since $a^2$, $\Delta t$ and $\Delta x$ will always be positive regardless of the wave direction, $|G_f|^2$ will always be greater than one since $a^2\left(\frac{\Delta t}{\Delta x}\right)^2 \sin^2(\beta\Delta x)$ is always positive. Figure2.15 displays a three-dimensional plot of the FTCS marched in time for a typical stable Courant number, $a\frac{\Delta t}{\Delta x} < 1$.

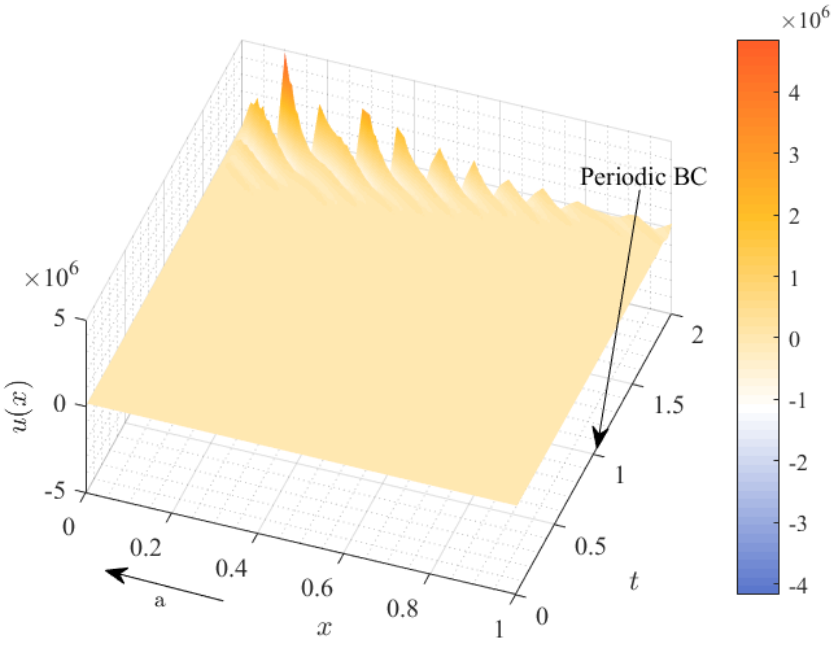



*Figure 2.15 – Three-dimensional plot of the FTCS displaying the development of the Gaussian pulse over time for* $a\frac{\Delta t}{\Delta x} < 1$.

## 2.5 Euler equation (Q2A)

*The 1D Euler equations for compressible gas in a straight pipe of constant section can be written as:*

$$\frac{\partial}{\partial t}\begin{bmatrix}\rho\\ \rho u\\ E_t\end{bmatrix}+\frac{\partial}{\partial x}\begin{bmatrix}\rho u\\ \rho u^2+p\\ u(E+p)\end{bmatrix}=\vec{0}$$

*Where* $\gamma=\frac{C_p}{C_v}$ *is the ratio of specific heats, the speed of sound is given by* $a^2=\gamma\frac{p}{\rho}$ *and E=-0.5ρu*$^2$*+e is the total energy.*

*Q2A) Show that equation 1 can also be written in vector-matrix format as:*

$$\frac{\partial U}{\partial t}+A(U)\frac{\partial U}{\partial x}=0$$

*Q2B) Compute the eigenvalues of A of equation 2 above. Recall that the trajectories C1, C2, C3 in the (x,t) plane can be defined from these eigenvalues. Along C1 defined by:*

$$\frac{dx}{dt}=\lambda_1=u-a$$

### 2.5.1 Background of the Euler equation

The integral form of the Navier Stokes equations entails:

$$\underbrace{\frac{\partial}{\partial t}\int_V W dV}_{(1)} + \underbrace{\int_s (\vec{Q}\cdot\vec{n}) dS}_{(2)} = 0 \tag{79}$$

Where term (1) is the change in conservative variable volume in time within an arbitrary control volume and $W$ is the conservative variable vector which dictates which governing equations of fluid dynamics are to be solved within a fluid continuum. For a three-dimensional fluid continuum and solving for the conservation of mass, momentum and energy, the conservation variable vector would surmount to:

$$W = [\rho, \rho u, \rho v, \rho w, E_t]^T \tag{80}$$

Where $\rho, u, v, w$ and $E_t$ are the density, cartesian velocities and energy respectively. Term (2) is the flux of the conservative variables where $\vec{Q}$ is the flux tensor which can be decomposed into inviscid and viscous flux vectors:

$$\vec{Q} = (E^I - E^V)i + (F^I - F^V)j + (G^I - G^V)k \tag{81}$$

Where $i, j$ and $k$ are the respective cartesian directions, $I$ and $V$ the inviscid and viscous, and letter synonyms, $E, F$ and $G$, for categorizing the flux vectors with associated cartesian directions. The integral equation of the Navier stokes equation can be put into a vector form which is useful for computational fluid dynamics as it is compact [1] and therefore provides easier data handling:

$$\frac{\partial W}{\partial t}+\frac{\partial E}{\partial x}+\frac{\partial F}{\partial y}+\frac{\partial G}{\partial z}=0 \tag{82}$$

The Euler equation is a reduced form of the Navier Stokes equations where the viscous terms are neglected resulting in a reduced flux tensor, $\vec{Q}=(E^I)i+(F^I)j+(G^I)k$, and in one dimension, the Navier Stokes equation reduces to:

$$\frac{\partial W_{1D}}{\partial t}+\frac{\partial E^I}{\partial x}=0 \qquad \text{One-dimensional Euler equation} \tag{83}$$

Where the conservative variables, $W_{1D}$, and inviscid flux vector, $E^I$ results in equation (84) in which the flux vector is composed of the fluxes for mass, momentum and energy corresponding to each rows of the vector:

$$W_{1D}=\begin{bmatrix}\rho\\ \rho u\\ E_t\end{bmatrix} \qquad E^I=\begin{bmatrix}\rho u\\ \rho u^2+p\\ (E_t+p)u\end{bmatrix} \tag{84}$$

### 2.5.2 Euler equation with Jacobian matrix (Q2A)

Reconsidering the one-dimensional Euler equation where:

$$\frac{\partial W}{\partial t}+\frac{\partial E}{\partial x}=0 \tag{83}$$

Rearranging and differentiating by time:

$$\frac{\partial^2 W}{\partial t^2}=-\frac{\partial^2 E}{\partial x dt}$$

Multiply by $\frac{\partial W}{\partial W}$ $(by\ 1)$:

$$\frac{\partial^2 W}{\partial t^2} = -\frac{\partial}{\partial t}\frac{\partial W}{\partial x}\frac{\partial E}{\partial W}$$

Integrating by time and $A = \frac{\partial E}{\partial W}$:

$$\frac{\partial W}{\partial t} + A\frac{\partial W}{\partial x} = 0 \tag{85}$$

Where *A* is flux vector Jacobian matrix. Jacobian matrices are formed of first-order partial differential equations.

## 2.6 Eigenvalues of the Jacobian matrix (Q2B)

*Compute the eigenvalues of A of equation 2 above. Recall that the trajectories C1, C2, C3 in the (x,t) plane can be defined from these eigenvalues.*

Consider the equations for the conservative variables (equation (84)) of the one-dimensional Euler equation and substitute $u_1, u_2$ and $u_3$ for each conservative variable:

$$W = \begin{bmatrix} \rho \\ \rho u \\ E_t \end{bmatrix} = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} \qquad E^I = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ (E_t + p)u \end{bmatrix} = \begin{bmatrix} u_2 \\ u_1\left(\frac{u_2}{u_1}\right)^2 + p \\ (u_3 + p)\frac{u_2}{u_1} \end{bmatrix} \tag{86}$$

Where, $p = (\gamma - 1)\left(u_3 - \frac{1}{2}\frac{u_2^2}{u_1}\right)$ and the second term in the flux tensor, $\rho u^2 + p$, is the source of the Euler equations non-linearity [7]. The Jacobian matrix is equivalent to $\frac{\partial E}{\partial W}$, which results in:

$$A = \frac{\partial E}{\partial W} = \begin{bmatrix} 0 & 1 & 0 \\ -\frac{1}{2}(3-\gamma)\frac{u_2^2}{u_1^2} & (3-\gamma)\frac{u_2}{u_1} & \gamma - 1 \\ -\gamma\frac{u_2 u_3}{u_1^2} + (\gamma-1)\frac{u_2^3}{u_1^3} & \gamma\frac{u_3}{u_1} - \frac{3}{2}(\gamma-1)\frac{u_2^2}{u_1^2} & \gamma\frac{u_2}{u_1} \end{bmatrix} \tag{87}$$

From this, the eigenvalues can be extracted by using the classical approach:

$$(A - \lambda I)\boldsymbol{x} = 0 \tag{88}$$

Where $\lambda$ and $x$ are the eigenvalue and eigen vectors and is solved by taking the determinant of the system which is equivalent to zero. The result should correspond to the form $A\boldsymbol{x} = \lambda\boldsymbol{x}$ and since the Jacobian matrices is a size of $3 \times 3$, three eigenvalues should be obtained. The eigenvalues obtained are displayed below:

$$\lambda_1 = u - a \tag{88}$$

$$\lambda_2 = u \tag{89}$$

$$\lambda_3 = u + a \tag{90}$$

Symbolic maths was used in Matlab to replicate these results and the code for which can be found in appendix 6.3.3.

# Chapter 3:

# Compressible flow around an aerofoil

## 3.1 Objectives and overview

The objective of this assignment entailed the simulation of the NACA0012 aerofoil using Finite Volume Method using the fourth-order Runga-Kutta method with a triangular, unstructured grid. Artificial dissipation was used to damp any unwanted oscillations which could invoke numerical divergence; the numerical order of the dissipation was interactive with the size of oscillations and mimicked the effects of natural viscosity. Local time stepping was performed to speed up convergence and Reimann boundary conditions were used at the outer boundaries.

The Runga-Kutta method was designed to enhance and quicken the convergence of the Euler method [8] which is significant since Computational Fluid Dynamics can be computationally expensive in time and resources. Furthermore, time-derivatives are likewise to spatial derivatives except for one key factor; time cannot be use the forward neighbouring point with respect to the evaluation point since the value is "in the future" [1]. In the past, this resulted in less order of accuracy for the time-derivatives where the 1st order accurate backwards difference ($\sigma(\Delta t)$) was typically used. Jameson, Schmidt and Turkel introduced the fourth-order Runga-Kutta method which uses four Runga-Kutta steps in the time integration, yielding a fourth-order accurate temporal scheme ($\sigma(\Delta t^4)$).

### 3.1.1 Grid generation

The grid used to simulate the NACA0012 aerofoil was an unstructured, C-type grid converted from a structured grid. C-type grids are named after their characteristic shape of a "C" and are widely used for aerofoil analysis [ref helicopter]. Unlike structured grids in which grid cells can be defined from neighbouring grid cells, an unstructured grid requires a so-called connectivity matrix [9] to define the layout and cell interactions. Unstructured grids have risen in popularity as they are relatively easier to design than structured grids [1]. However, the

drawbacks are that the grids are prone to adverse cell quality such as cell skewness, zero-volume cells; furthermore, it is harder to apply turbulence models (viscous computations) [1]. Figure 3.1 displays the difference between the structured and unstructured connectivity.

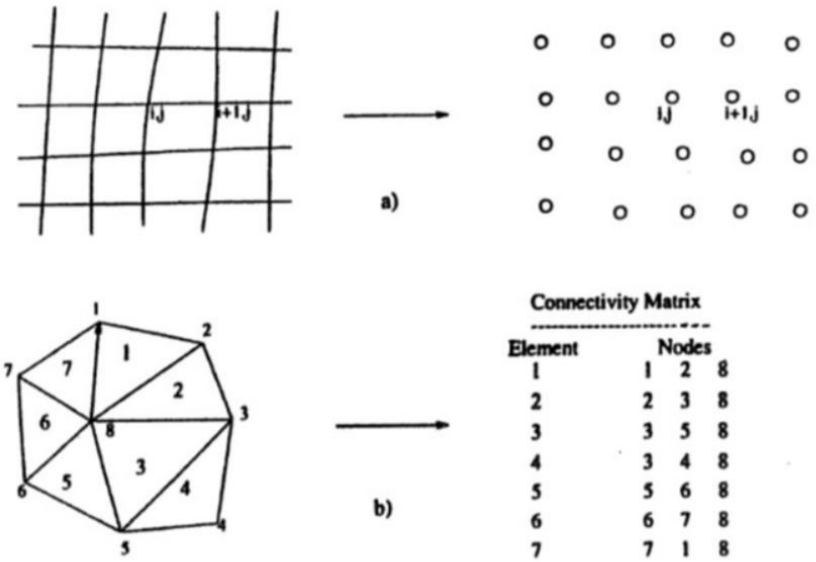



*Figure 3.1 - Difference between unstructured and structured connectivity, courtesy of Dr Barakos [9]*

Computations between the cells entail two cell numbers, two vertices and an edge shared by both cells. These are labelled $K, A, B, P$ and $e$, in which $K$ and $P$ are the cell numbers and $A$ and $B$ the vertices and $e$ the edge. The cell numbers are used to index the cell connectivity and relate the cell to its according conservative variables, relative flux and boundary conditions.

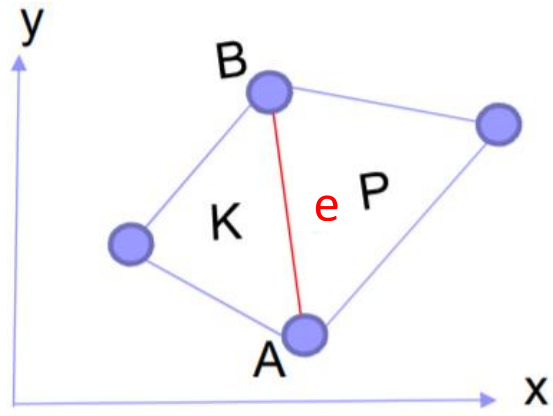



*Figure 4.2 – Cell layout. Image courtesy of Dr Barakos [9]*

The connectivity matrix further separates the grid into convenient sections each containing the grid boundaries such as the aerofoil surface and outer boundaries. This allows for easier indexing on which calculations are used. Figure 4.3 displays the mesh used with dark green, red and black indicating the aerofoil surface, outer boundary and inner fluid.

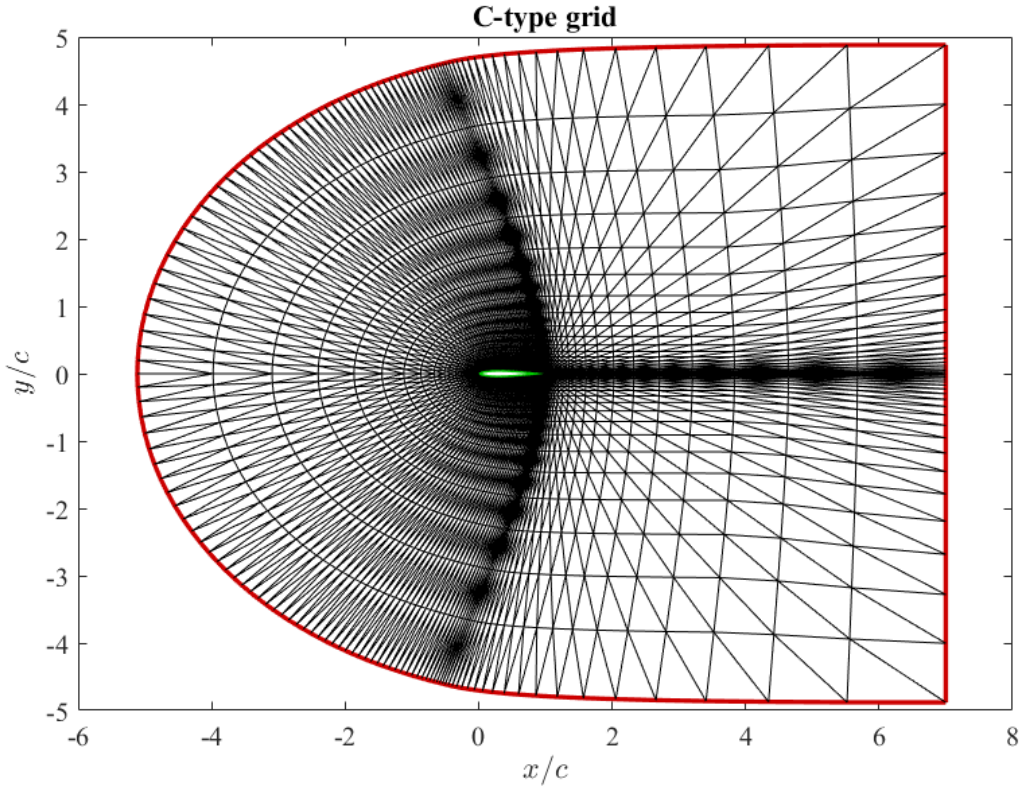



*Figure 4.3 – C-type unstructured grid designed by Dr Barakos. Dark green, red and black indicating the aerofoil surface, outer boundary and inner fluid.*

### 3.1.2 The Runga-kutta method

The Runga-Kutta method for time integration is used for higher orders of accuracy in time [1] and has displayed robustness and accuracy [8]. For this assignment, a 4$^{th}$ order Runga-Kutta method is used which yields an order of accuracy in the spatial dimensions and time of the second and fourth order, $\sigma(\Delta x^2, \Delta y^2, \Delta t^4)$ [1]. The coefficients of the method are displayed below [9] where CFL indicates the Courant condition:

$$S_1 = 0.25\, CFL \tag{91}$$

$$S_2 = \frac{1}{3} CFL \tag{92}$$

$$S_3 = 0.5\ CFL \tag{93}$$

$$S_4 = CFL \tag{94}$$

### 3.1.3 Governing equations of fluid

The integral form of the two-dimensional, inviscid Navier Stokes, or Euler equation, surmounts to:

$$\frac{\partial}{\partial t}\int\int_{S_A} W dxdy + \int_{\partial S_A} \left(\vec{Q}\cdot\vec{n}\right) = 0 \tag{95}$$

In vector form:

$$\frac{\partial W}{dt} + \frac{\partial E^I}{dx} + \frac{\partial F^I}{dy} = 0$$

Where $S_A$ and $\partial S_A$ are the area and boundary of the cell under analysis. $W$ and $\vec{Q}$,as previously stated, are the conservative variables and flux tensor in which for the 2D Euler equation in vector form are [1]:

$$W_{2D} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ E_t \end{bmatrix} = \begin{bmatrix} Mass \\ X-momentum \\ Y-momentum \\ Energy\ per\ unit\ area \end{bmatrix} \tag{96}$$

$$\vec{Q} = E^I + F^I = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ (E_t + p)u \end{bmatrix} + \begin{bmatrix} \rho v \\ \rho vu \\ \rho v^2 + p \\ (E_t + p)v \end{bmatrix} \tag{97}$$

Where, the energy per unit area and pressure are:

$$E_t = \rho\left(\frac{p}{(\gamma-1)\rho} + \frac{1}{2}(u^2+v^2)\right) \quad \text{Energy} \quad (98)$$

$$p = (\gamma-1)\left(E_t - \frac{1}{2}\rho(u^2+v^2)\right) \quad \text{Pressure} \quad (99)$$

Substituting in the flux tensor decomposition, equation (95) becomes:

$$\frac{\partial}{\partial t}\int\int_{S_A} W dxdy + \int_{\partial S_A} E^I\, dy - F^I\, dx = 0 \quad (100)$$

The governing equation are solved as a system of vectors which can be easily implemented into any numerical computing environment. For cell $K$, equation (100) can be displayed as:

$$\frac{\partial}{\partial t} S_A (W)_K + \sum_{k=1} (E^I)_K dy - (F^I)_K dx = 0 \quad (101)$$

For the implementation and as an example, the Y-momentum has been derived fully in equations $(102) \rightarrow (105)$, and equations (107), (108), (109) and (110) display the resulting governing equations for conservation of X-momentum, mass, energy and pressure.

***Y-Momentum:***

The associated values from equation (97) for the y-momentum are inputted into equation (101) for cell $K$:

$$\frac{\partial}{\partial t}S_A(\rho v) + \sum_{k=1}(\rho uv)_K dy - (\rho v^2 + p)_K dx = 0 \quad (102)$$

Multiplying out the brackets and rearranging such that flux velocity, $Q_K$, can be substituted in:

$$\frac{\partial}{\partial t}S_A(\rho v) + \sum_{k=1}(\rho uv)_K dy - (\rho v^2)_K dx - p_K dx = 0 \quad (103)$$

$$\frac{\partial}{\partial t}S_A(\rho v) + \sum_{k=1}(\rho v)_K (u_K dy - v_K dx) - p_K dx = 0 \quad (104)$$

$$\frac{\partial}{\partial t}S_A(\rho v) + \sum_{k=1} Q_{FV,K}(\rho v)_K - p_K dx = 0 \quad (105)$$

Where, $Q_{FV,K}$, the resultant flux velocity across the edge of cell $K$:

$$Q_{FV,K} = u_K dy - v_K dx \quad (106)$$

The generalized governing equations for x-momentum, mass and energy per unit volume are calculated displayed below where $j$ denotes cell $K$ or $P$:

$\frac{\partial}{\partial t}S_A(\rho u) + \sum Q_{FV,e}(\rho u)_j + p_j dy = 0$ ***X-momentum*** (107)

$\frac{\partial}{\partial t}S_A(\rho v) + \sum Q_{FV,e}(\rho v)_j - p_j dx = 0$ ***Y-momentum*** (108)

$\frac{\partial}{\partial t}S_A(\rho) + \sum \rho_j Q_{FV,e} = 0$ ***Continuity*** (109)

$\frac{\partial}{\partial t}S_A(E_t) + \sum((E_t)_j + p_j)Q_{FV,e} = 0$ ***Energy*** (110)

### 3.1.4 Flux calculations in the simulation

The fluxes are calculated across the edges of primary cells ($K$) and secondary cells ($P$). Furthermore, for computational efficiency the fluxes were only calculated once and applied as positive for $K$ and negative for $P$ in accordance to flux direction and conservation, i.e. the fluxes balancing between cells.

$$Q_{K,W_i} = Q_{K,W_i} + F_{i,W_i} \tag{111}$$

$$Q_{P,W_i} = Q_{P,W_i} - F_{i,W_i} \tag{112}$$

Where $W_i$ denotes which conservative variable flux is being calculated. The flux equations for each conservative variable are formed from the flux velocity and averaged conservative variables between the primary and secondary cells:

$$F_{i,W_1} = \frac{W_{1,K}+W_{1,P}}{2} Q_{FV,e} \qquad \text{Continuity} \tag{113}$$

$$F_{i,W_2} = \frac{W_{2,K}+W_{2,P}}{2} Q_{FV,e} + \frac{p_K+p_p}{2} dy \qquad \text{X – momentum} \tag{114}$$

$$F_{i,W_3} = \frac{W_{3,K}+W_{3,P}}{2} Q_{FV,e} - \frac{p_K+p_p}{2} dx \qquad \text{Y – momentum} \tag{115}$$

$$F_{i,W_4} = \left(\frac{W_{4,K}+W_{4,P}}{2} + \frac{p_K+p_p}{2}\right) Q_{FV,e} \qquad \text{Energy} \tag{116}$$

Where in accordance to the vector form of $W$ and $j$ denoting cell $K$ or $P$,

$$W_{1,j} = \rho_j\,, \qquad W_{2,j} = (\rho u)_j \qquad W_{3,j} = (\rho v)_j \qquad W_{4,j} = (E_t)_j \tag{117}$$

Furthermore, the velocities, $u$ and $v$, in $Q_{FV,e}$ are calculated using the conservative variables:

$$u = \frac{1}{2}\left(\frac{W_{2,K}}{W_{1,K}} + \frac{W_{2,P}}{W_{1,P}}\right) = \frac{1}{2}\left(\frac{(\rho u)_K}{\rho_K} + \frac{(\rho u)_P}{\rho_P}\right) \tag{118}$$

$$v = \frac{1}{2}\left(\frac{W_{3,K}}{W_{1,K}} + \frac{W_{3,P}}{W_{1,P}}\right) = \frac{1}{2}\left(\frac{(\rho v)_K}{\rho_K} + \frac{(\rho v)_P}{\rho_P}\right) \tag{119}$$

The flux calculations varied at different sections within the grid, mainly, the aerofoil surface, inner fluid and outer boundary. The inner fluid flux calculations are the general flux equations above. At the aerofoil surface, there is no secondary cell ($P$) as it is a boundary and no flow can pass through the edge since the boundary is classified as solid. This results in the flux equation only being applied for two conservative variables, the momentums; continuity and energy are neglected since there is no mass flow rate, which render the flux equations for the momentums as only the pressure contribution.

$F_{i,W_2} = +\frac{p_K + p_p}{2} dy$ Aerofoil X-momentum (120)

$F_{i,W_3} = -\frac{p_K + p_p}{2} dx$ Aerofoil Y-momentum (121)

At the boundary, the cells are considered far enough from the aerofoil such that free-stream conditions apply; the flux calculations use the free-stream conditions instead of the conservative variables which invoke a free-stream boundary.

$F_{iB,W_1} = W_{2,B} Q_{B,FV,e}$ Continuity at boundary (122)

$F_{iB,W_2} = W_{2,B} Q_{B,FV,e} + p_B dy$ X – momentum at boundary (123)

$F_{iB,W_3} = W_{3,B} Q_{B,FV,e} - p_B dx$ Y – momentum at boundary (124)

$F_{iB,W_4} = (W_{4,B} + p_B)\, Q_{B,FV,e}$ Energy (125)

### 3.1.5 Artificial viscosity

Artificial viscosity was added to the conservative variable calculations and was required to damp out undesired oscillations within each conservative variable, which in turn, could lead to inaccurate results or no solution at all. This parameter damps out oscillations as it introduces dissipative effects, analogous to physical viscosity, which smooths out gradients [1]. An example of a gradient discontinuity that artificial viscosity can damp out is Gibbs phenomenon, as shown in figure 4.3. Image courtesy of authors in [1].

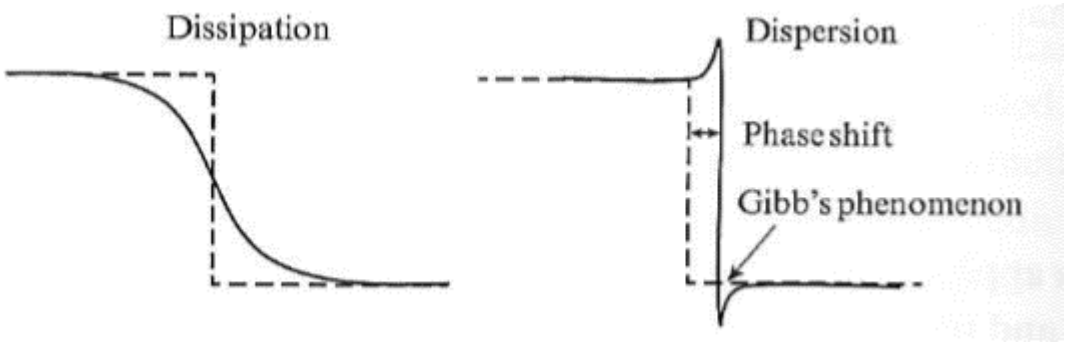



*Figure 4.4 – Effects of artificial viscosity on Gibbs phenomenon. Image courtesy of authors in [1].*

Applying the dissipative terms to the conservative variables leads to [8]:

$$\frac{\partial}{\partial t} S_A W + Q - D = 0 \tag{126}$$

Where the dissipative term, $D$, is equivalent to:

$$D = \sum_{i=1}^{e} d_i^{(2)} + \sum_{i=1}^{e} d_i^{(4)} \tag{127}$$

The dissipative term is calculated differently at different regions of the grid: near and far of the aerofoil surface which correspond to $d_i^{(2)}$ and $d_i^{(4)}$. This is different values are required due to

the addition of the pressure field of the aerofoil. A shock sensor equation is used to determine which dissipative term to use as it accounts for the pressure field of the aerofoil:

$$v_i = \frac{|p_P - p_K|}{|p_P + p_K|} \tag{128}$$

Which relates to the dissipative terms:

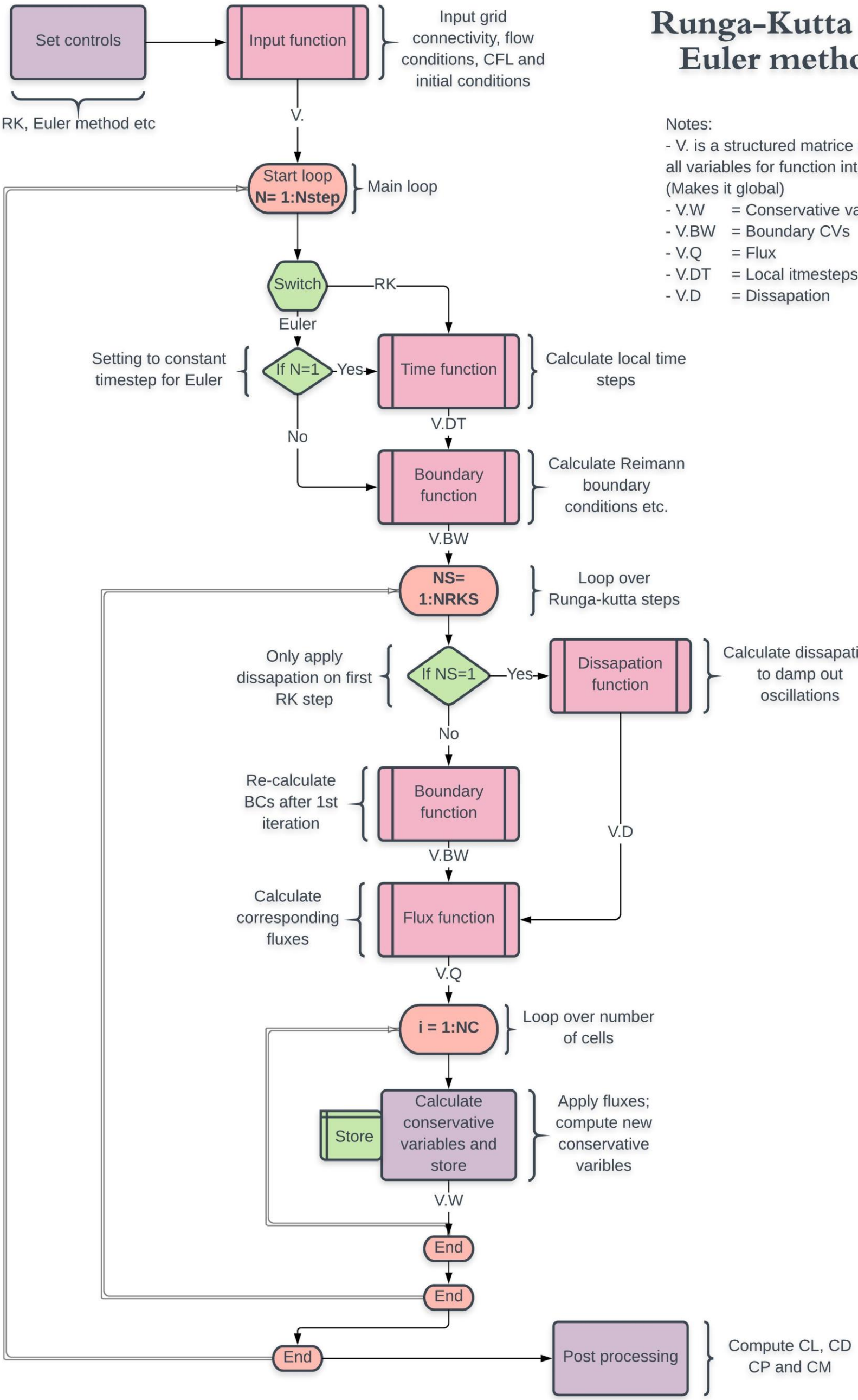
$$d_i^{(2)} = \lambda_i \left(W_p - W_K\right)\left(k^{(2)} v_i\right) \tag{129}$$

$$d_i^{(4)} = -\lambda_i (\nabla^2 W_P - \nabla^2 W_K)\left(\max\left(0, k^{(4)} - k^{(2)} v_i\right)\right) \tag{130}$$

Where $\lambda_i$ is scaling factor correlating to the maximum eigenvalue, $u + a$, of the Jacobian matrix [9]. $k^{(2)}$ and $k^{(4)}$ are empirical constants related to the finite computer precision. $d_i^{(2)}$ is the dissipative term used near the aerofoil surface; when the shock sensor is relatively large, equation (129) tends to zero since the maximum of zero and a negative value is zero. The vice versa occurs away from the aerofoil surface, where the shock sensor will be near zero since the pressure begins to equalize to freestream values.

## 3.2 Flow chart of code for Runga-Kutta and Euler method

Figure 4.5 displays a flowchart of the main function with its child functions of the flux, dissipation, time, and boundary conditions.




# Runga-Kutta and Euler method

Notes:
- V. is a structured matrice containing all variables for function interaction (Makes it global)
- V.W = Conservative variables
- V.BW = Boundary CVs
- V.Q = Flux
- V.DT = Local itmesteps
- V.D = Dissapation

*Figure 4.5 – Flow chart of the main code with child functions: Time, Boundary conditions, Flux and Dissipation.*

## 3.5 Results of the Runga-Kutta method

The resultant coefficient of lift, drag and moment compared with Dr Barakos's results are displayed in table 3.2 Below:

*Table 3.2 – Coefficient of lift, drag and moment compared*

| | The code | Dr Barakos |
|---|---|---|
| $C_L$ | 0.2567 | 0.2581 |
| $C_D$ | 0.0024 | 0.00236 |
| $C_M$ | -0.0021 | -0.00217 |

Figure 4.6 → 4.7 Displays the results of the simulation for the distribution of pressure over the aerofoil surface and convergence history for the coefficient of lift and drag.

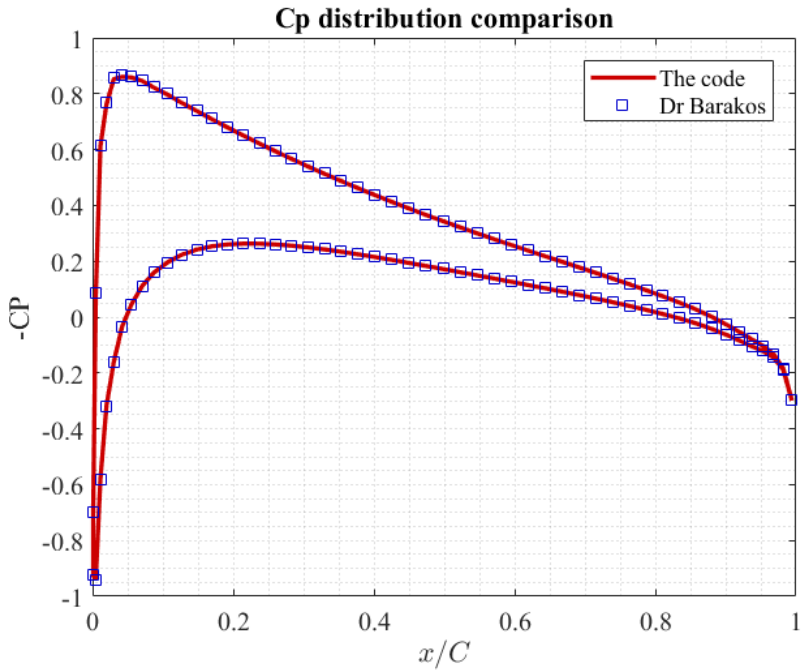



*Figure 4.6 – Distribution of coefficient of pressure over aerofoil surface, compared with Dr Barakos's results*

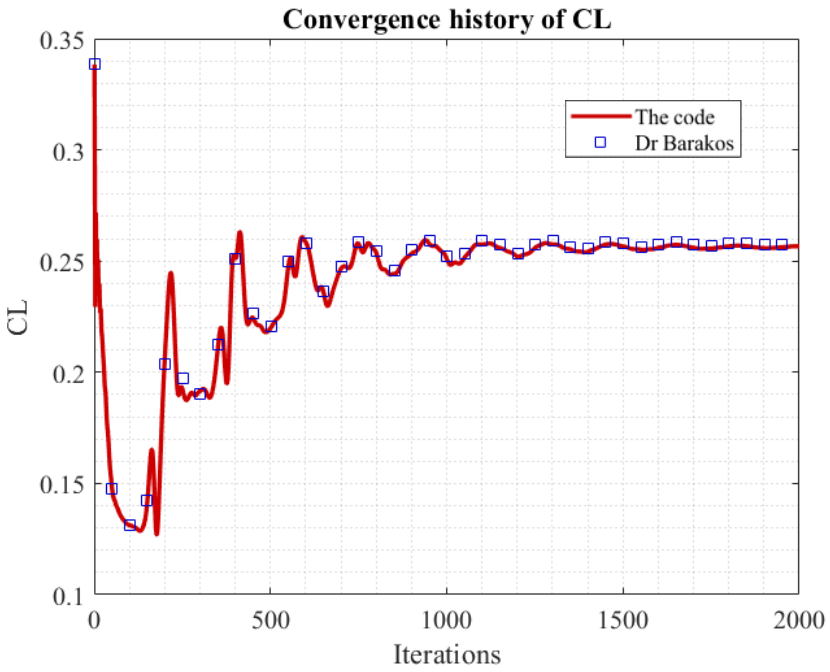



*Figure 4.7 – Convergence history of the lift coefficient, compared with Dr Barakos's results.*

Clear cohesion between the results of Dr Barakos indicate the code is correct and will be used to implement the Euler method in the next assignment. Furthermore, considering the code *without* the dissipation function shows clearly that the oscillations deteriorate the solution over the iterations. This displayed in figure 4.8 and lastly, figure 4.9 displays a velocity vector plot of the flow over the NACA0012 aerofoil.

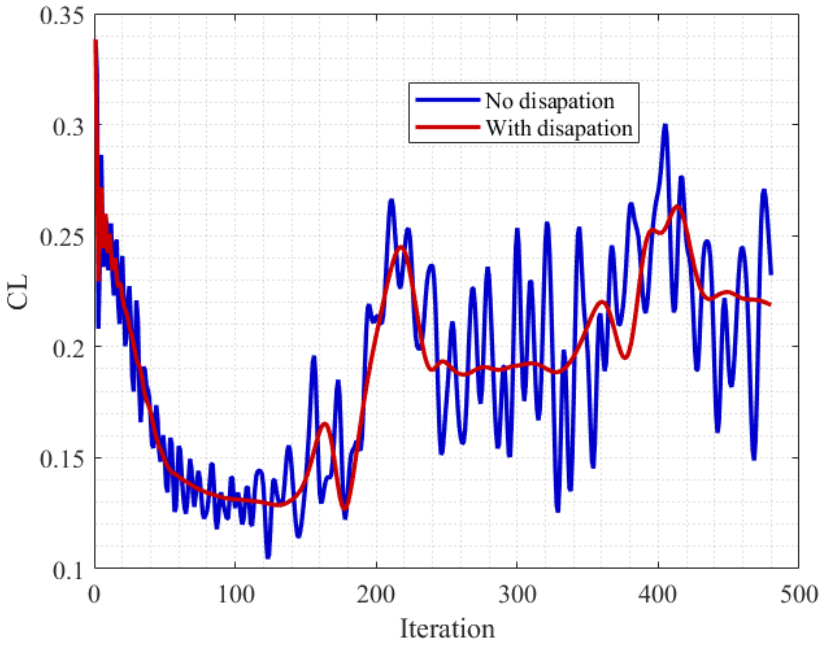



*Figure 4.8 – Comparison of code with and without the dissipation function over iterations.*

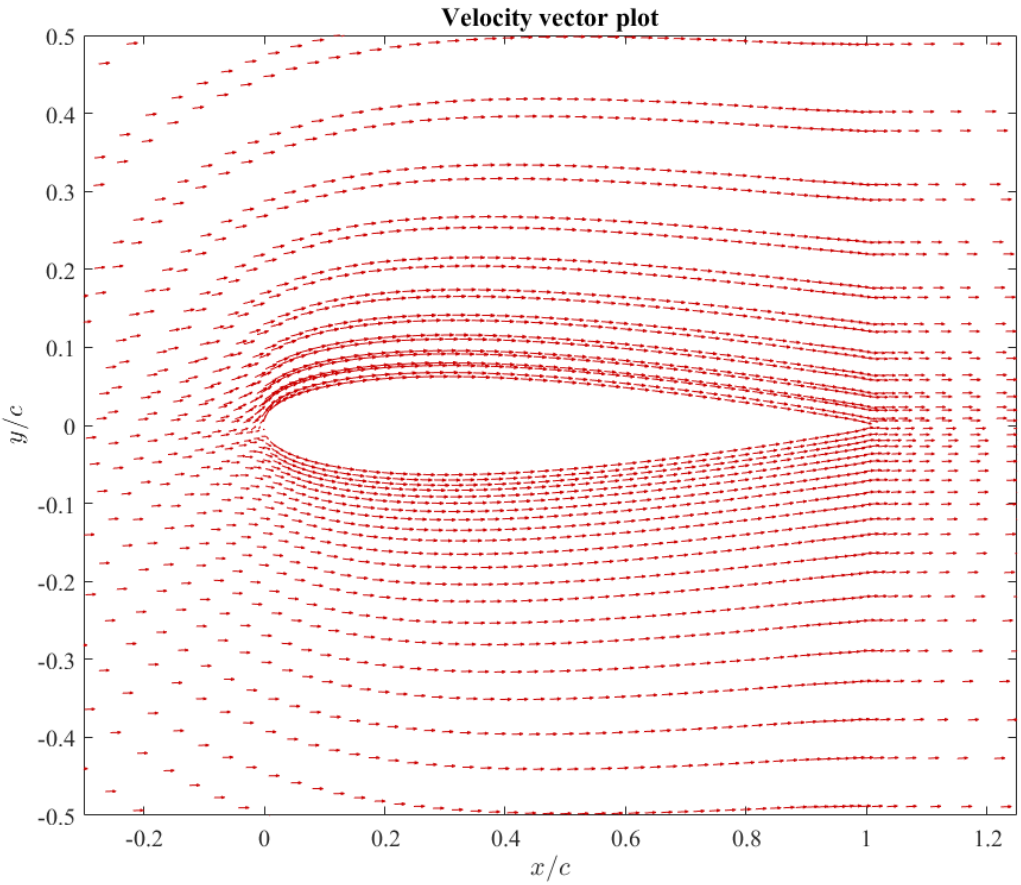



*Figure 4.9 – Velocity vector plot of flow over NACA0012 aerofoil.*

Where the velocity vectors are based on the corresponding velocity values of every cell, $K$, since each cell is $K$ at one point in the connectivity matrix. Since a cell can be applied multiple times throughout the connectivity matrix, the matrix required to be processed, where each value of $K$ was ordered and linked to its associated vertices. The position of the velocity vectors were established by averaging the vertices of the cell.

# Chapter 4:

# Convergence of the Euler method

The Euler method is the 1st order Runga-Kutta method where only the first coefficient is used for the time integration which yields a first-order accuracy in the temporal scheme ($\sigma(\Delta t)$). Furthermore, the local time-step was set to constant, as in accordance to the flowchart in figure 4.5.

## 4.1 Differences between the R-K and Euler method (4A, 4B, 4C)

### 4.1.1 Computational time difference

As mentioned previously, the Runga-Kutta method was designed to enhance and quicken the convergence of the Euler method where computational time and rate of convergence. This is of significant importance for Computational Fluid Dynamics as it is often a computationally expensive method in time and resources. Typically, computers are required to have high workspace memory (Random-Access-Memory) and processing speeds in order to execute CFD. The high memory is typically required to store large system of equations and the method used for grid definition; for example, assignment 1 required the so-called identity matrix to be stored to define the grid and fluid continuum, whereas for assignment 3, the grid was defined by a connectivity matrix. The computational resource ceiling is rising with further developments to the technological area, however, CFD codes should still be created for maximum efficiency in terms of coding and method used [1]. CFD is typically an iterative process, where the solution is marched towards a converged state, therefore, any inefficiencies in the code or method can produce significant increase in the time taken; therein lies the other reason for developing the Runga-Kutta method to simulate the Euler equation. For an example of code inefficiency, consider a loop of 10000 iterations; if an inefficient code statement increased the time taken per iteration by 0.001 seconds, this would result in an increase of 10 seconds overall.

For both simulations, the computational power used was an Intel i7-7700HQ CPU @2.80 GHz with 16 GB RAM (15.8 useable) and were run at $M = 0.5$. For the Runga-Kutta method, the

simulation was run for 2000 iterations at a CFL of 5.4 and local time stepping with the purpose to reduce the simulation time required, i.e. speed up convergence. The Euler method required 200,000 iterations to converged and the CFL used was 0.15, where the CFL must be $< 0.8$ for stability. Table 4.1 and 4.2 below shows the time taken for each function for the Runga-Kutta and Euler method respectively. Note, Matlab employs only one processor if parallel computing is not selected [10], therefore, the times indicate CPU times.

*Table 4.1 – Time taken for simulation for R-K method at Mach 0.5*

| Function | Full-time (s) | Self-time (s) | Calls | Self-time/call (s) |
|---|---|---|---|---|
| Main script | 822.4 | 244.7 | 1 | 244.70 |
| Flux | 382.2 | 382.2 | 8000 | 0.0478 |
| Boundary conditions | 8.7 | 8.7 | 8000 | 0.0011 |
| Dissipation | 148.1 | 148.1 | 2000 | 0.0740 |
| Time | 43.2 | 43.2 | 2000 | 0.0216 |
| Input | 4.8 | 4.8 | 1 | 4.8 |

*Table 4.2 – Time taken for simulation for Euler method at Mach 0.5*

| Function | Full-time (s) | Self-time (s) | Calls | Self-time/call (s) |
|---|---|---|---|---|
| Main script | 39439.8 | 9060.5 | 1 | 9060.5 |
| Flux | 11991.2 | 11991.2 | 200000 | 0.06 |
| Boundary conditions | 299.2 | 299.2 | 200000 | 0.0015 |
| Dissipation | 18079.9 | 18079.9 | 200000 | 0.0904 |
| Time | 0.0246 | 0.0246 | 1 | 0.0246 |
| Input | 4.8 | 4.8 | 1 | 4.8 |

Where full-time encompasses the time taken for the function itself and the so-called child functions within. Self-time is the time taken for the function individually.

In terms of iterations required for convergence, the Euler equation requires an order of magnitude $10^2$ more than the Runga-Kutta method. This portrays the ingenuity and effectiveness of the Runga-Kutta method created by Jameson, Schmidt and Turkel. Furthermore, the Runga-Kutta method is able to use a CFL of 5.4 however the Euler required $< 0.8$ in order to remain stable, this in turn, is one of the reasons for the longer CPU times.

### 4.1.2 Result comparison of the R-K and Euler method

The comparison of the coefficient of pressure over the aerofoil surface and coefficient of lift convergence history is displayed in figure 4.10 and 4.11. Note, the iterations in the convergence history have been normalized by the maximum iterations to enable the comparison.

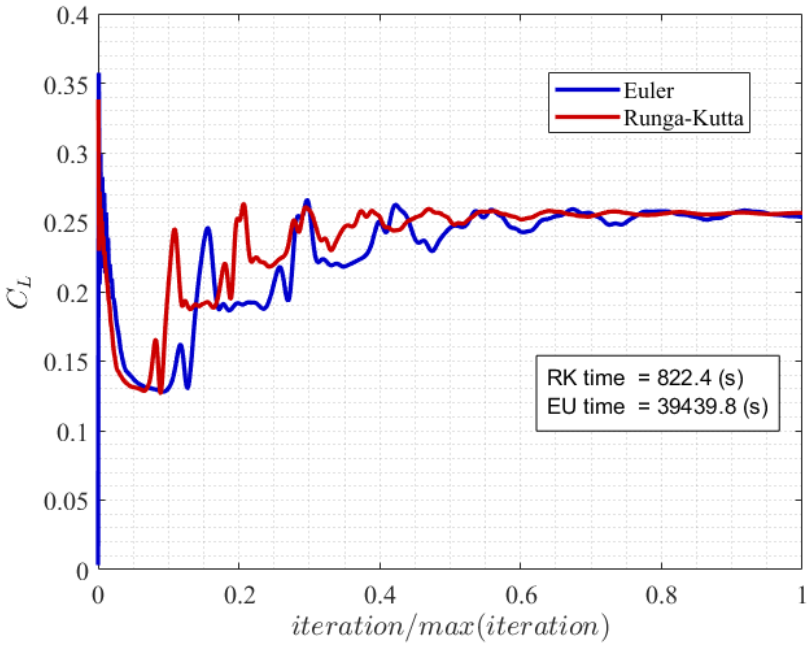



*Figure 4.10 – Comparison of $C_L$ convergence history for the Euler and Runga-Kutta method.*

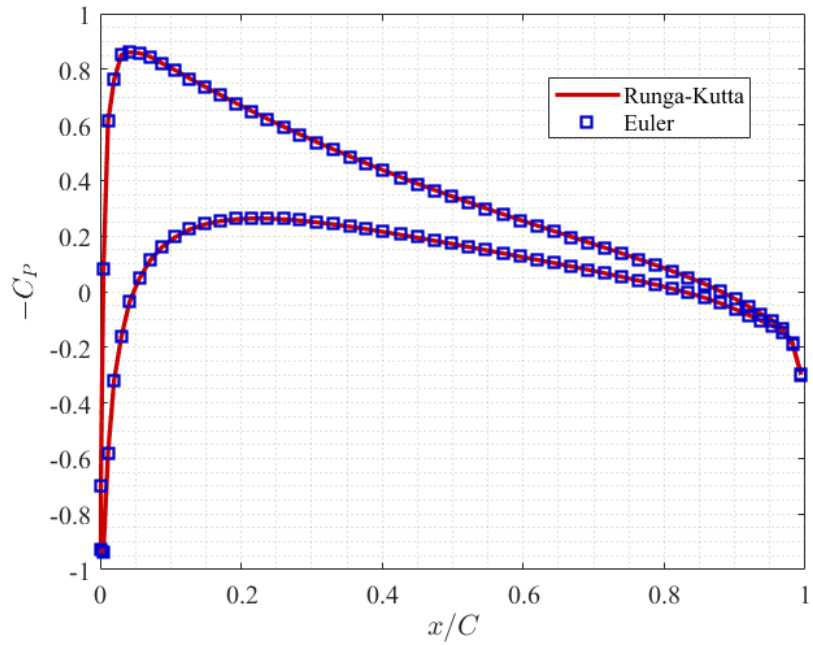



*Figure 4.11 – Comparison of $C_P$ over aerofoil surface between the Euler and Runga-Kutta method.*

The convergence in the $C_L$ is similar and follows analogous numerical trends towards the converged solution. Most significantly, the values at the converged state are likewise. Furthermore, the coefficient of pressure distribution is in agreement and figure 4.12 displays the velocity vector plot for the Euler method in which is identical to the Runga-Kutta method.

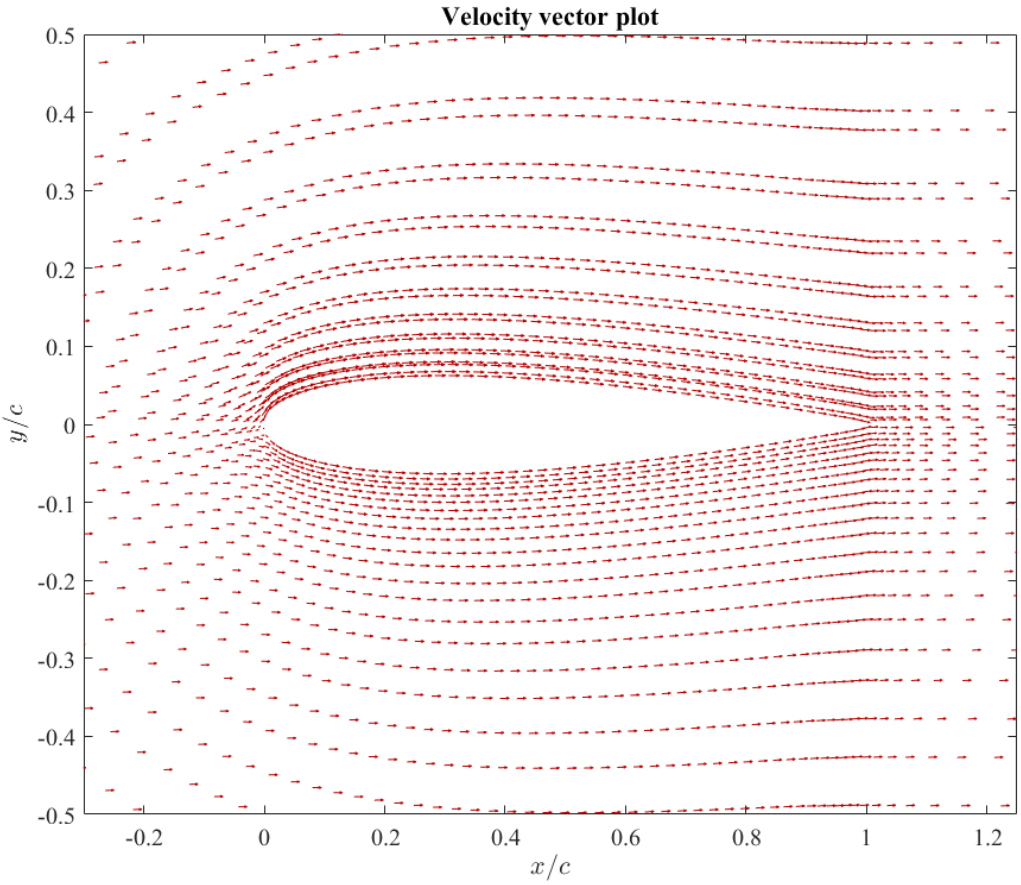



*Figure 4.12 – Velocity vector plot of flow over NACA0012 aerofoil for the Euler method.*

## 4.2 Comparing the Runga-Kutta method to panel methods (4D)

Comparison between the Runga-Kutta and panel method was executed for a Mach number equivalent to 0.1 at an angle of incidence of $2^o$. The panel method was executed on Q-Blade, which employs XFOIL for the aerofoil polars. Panel methods typically use an inviscid, incompressible and irrotational flowfield, however, XFOIL is combined with a fully coupled viscous-inviscid interaction method [11] which allows for viscous effects to be included. For

the comparison and since the Euler equation is inviscid, the Reynolds number inputted was not significant and the inviscid curve feature was selected on Q-Blade to obtain the coefficient of pressure distribution for the panel method. However, the viscid curve for an $Re = 1 \times 10^6$ was also taken for comparison as the artificial viscosity is synonymous to physical viscosity, and therefore will be impactful on the inviscid results. Figure 4.13 and 4.14 display the panelled aerofoil used in Q-Blade and the comparison between the coefficient of pressure values.

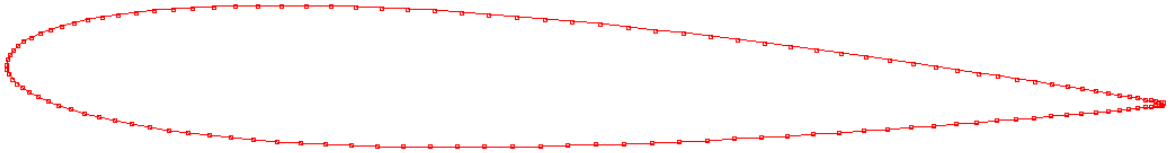


*Figure 4.13 – Panelled aerofoil created on Q-Blade for the panel method simulation.*

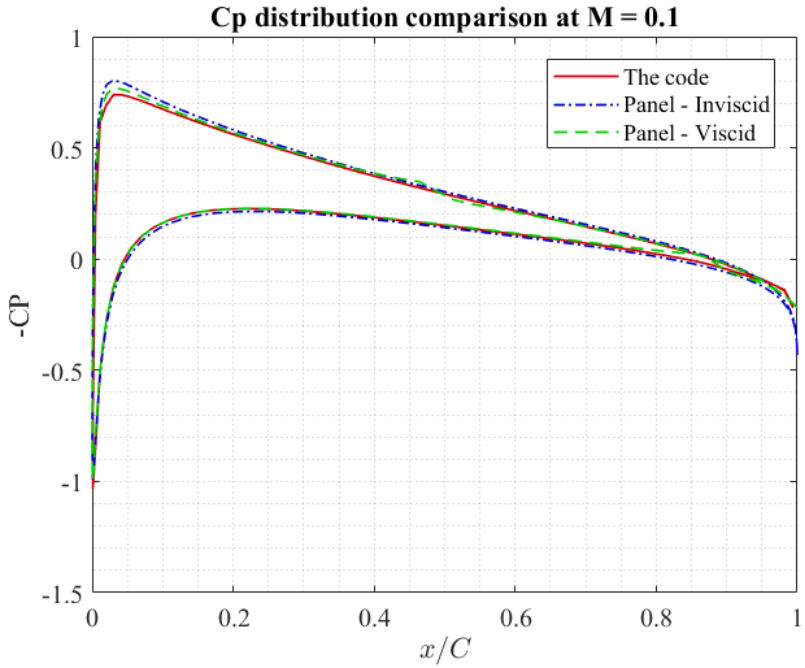



*Figure 4.13 – Comparison of pressure coefficient over aerofoil surface between the Runga-kutta, panel inviscid and viscid methods.*

The coefficient of pressure of the results display interesting characteristics; at the leading and trailing edge of the aerofoil, the Runga-kutta method agrees significantly better with the viscid panel method than the inviscid, despite the Euler method being inviscid by nature. This is due to the artificial viscosity and it is likely that a Reynolds number can be found for the viscid panel methods to closely mimic the Runga-kutta method. On comparison of the inviscid panel method, the Runga-kutta method agrees well in the mid regions of the aerofoil on both upper and lower surfaces.

## 4.3 Validity of the Runga-Kutta method on complex aerofoils (4E)

The two-element NLR/Boeing aerofoil and SKF aerofoil were also simulated using the Runga-Kutta method likewise to the NACA0012. The objective of the simulation was to investigate the validity of the Runga-Kutta results on complex aerofoils by inspection of plots of the flowfield and surface coefficient of pressure. All computational grids are courtesy of Dr George Barakos.

The second element of the aerofoil is known as a high-lift device and on the trailing edge have the effect of shifting the lift curve of the aerofoil upwards [12], thus resulting in an overall increase of lift coefficient as camber is increased. Consequently however, they come with a drag penalty. Nonetheless, they are significantly effective for take-off and landing as they can be extended during these periods [12] and retracted during cruise conditions, thus minimizing the drag penalty and optimizing take-off, cruise and landing.

### 4.3.1 The two-element NLR/Boeing aerofoil

The NLR Boeing aerofoil required 15,000 iterations in order to converged and the stability criterion was a $CFL$ equivalent to 1. The flow conditions used was a Mach number of 0.5 at an angle of incidence of $1^o$. Figure 4.14 And 4.15 displays the NLR Boeing aerofoil unstructured grid and a close-up of the mesh at the near-flap region.

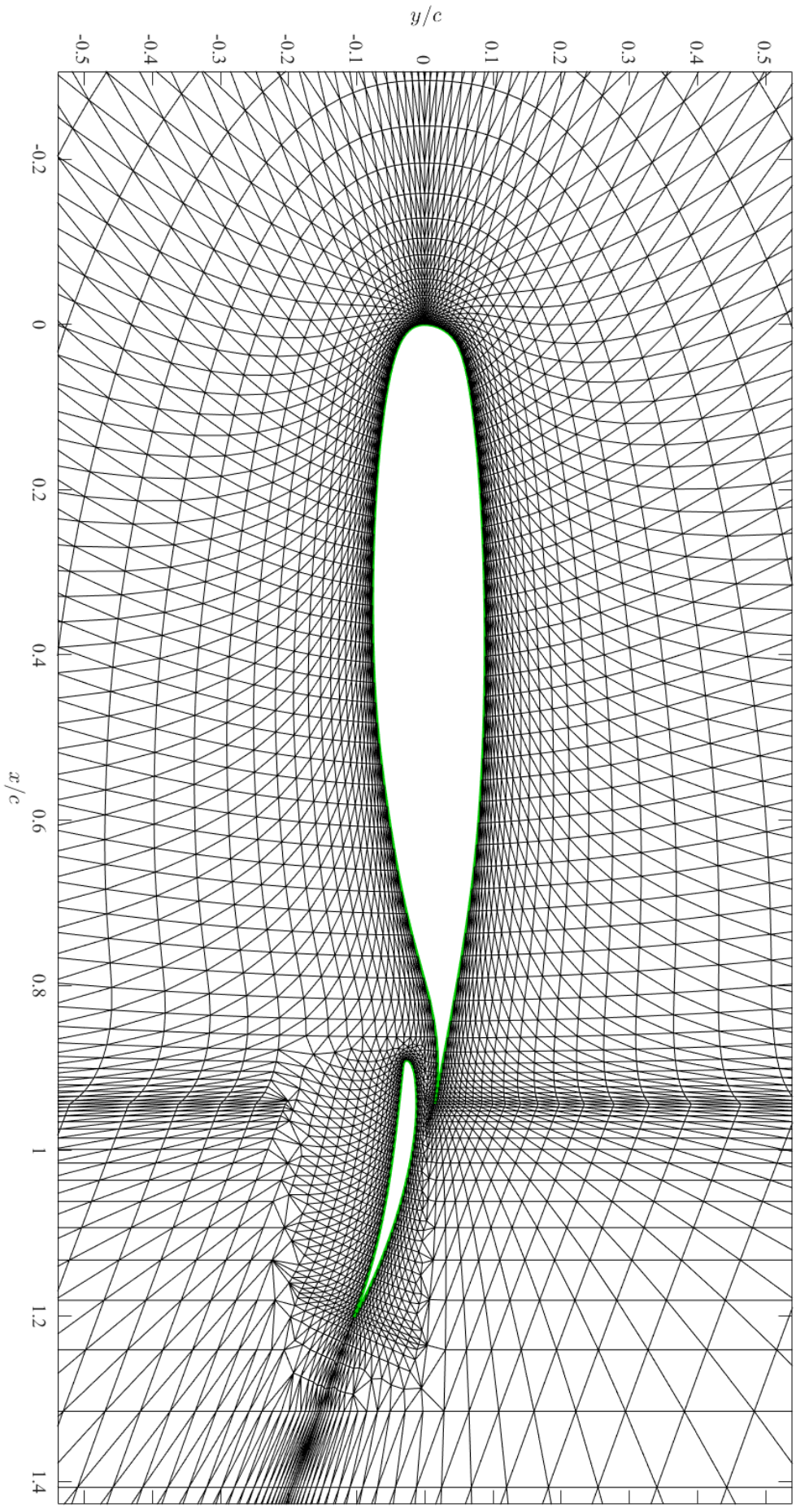



*Figure 4.14 – Two-element NLR Boeing computational grid, where the dark green lines indicate the aerofoil surfaces. Grid courtesy of Dr George Barakos.*

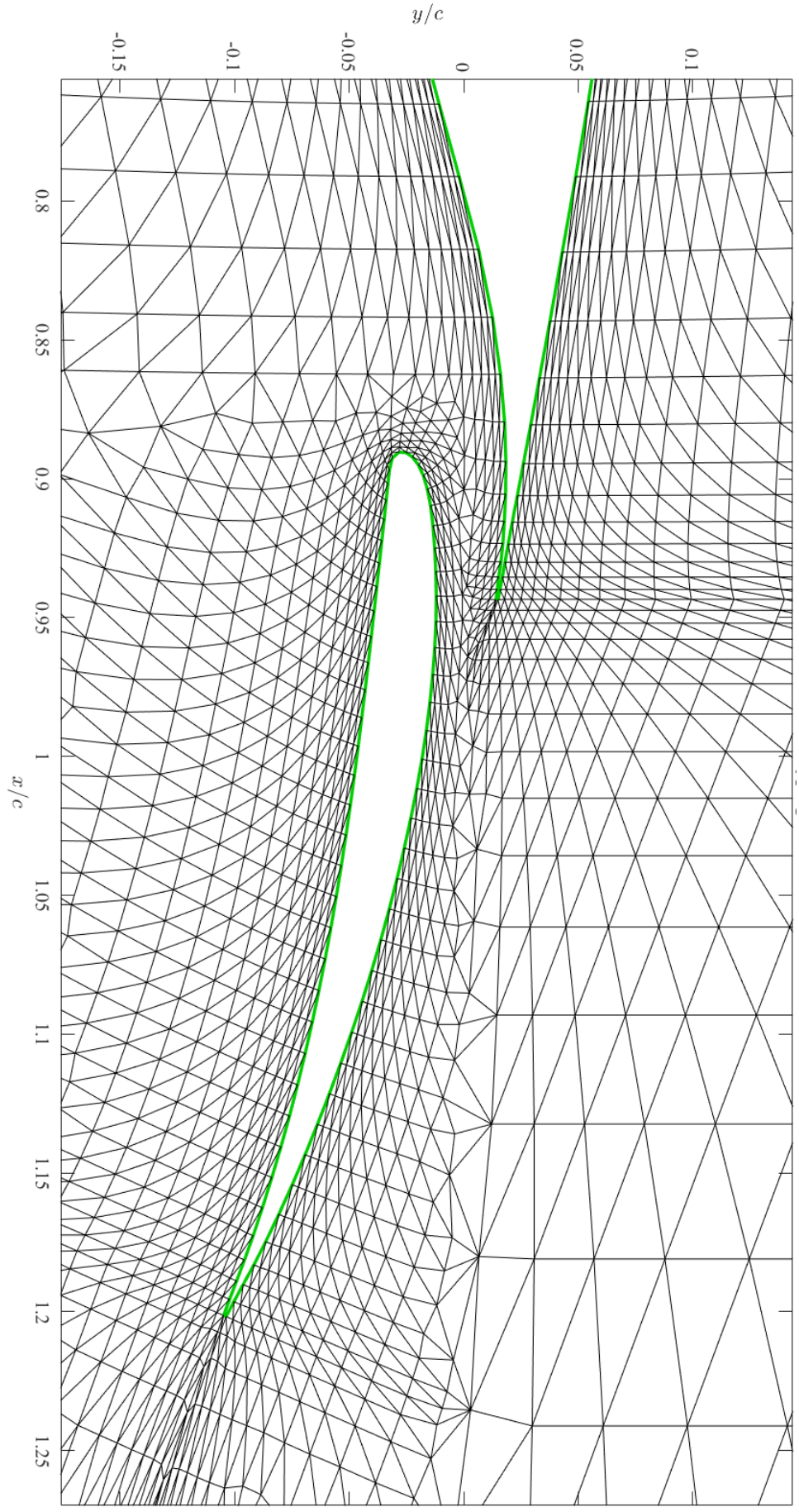



*Figure 4.15 – Close up of the mesh at the near-flap region.*

The results for the NLR Boeing mesh for the coefficient of lift, drag and moment are displayed in table 4.3.

*Table 4.3 – NLR Boeing aerofoil coefficients*

| Coefficients | R-K code |
|---|---|
| $C_L$ | 2.0783 |
| $C_D$ | 0.0477 |
| $C_M$ | -0.5348 |

Figure 4.16, 4.17 and 4.18 displays the surface $C_P$ and flowfield of the entire aerofoil and near flap region respectively. The flowfield of the NLR/Boeing and SKF aerofoil will be discussed together in section 4.3.3.

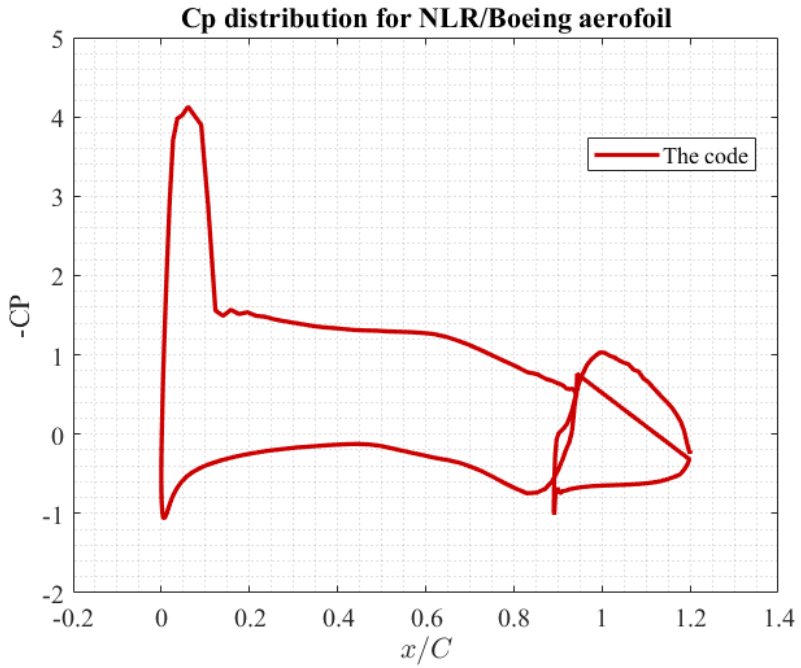



*Figure 4.16 – Coefficient of pressure distribution over the NLR/Boeing aerofoil.*

The pressure distribution is as expected for a two-element aerofoil where the cross-over of the lower surface pressure coefficient is due to the gap between the aerofoil and high-lift device. The high pressure on the lower side of the aerofoil accelerates the fluid through the gap, thus causing the absolute pressure coefficient to drop and re-establish once passed the gap.

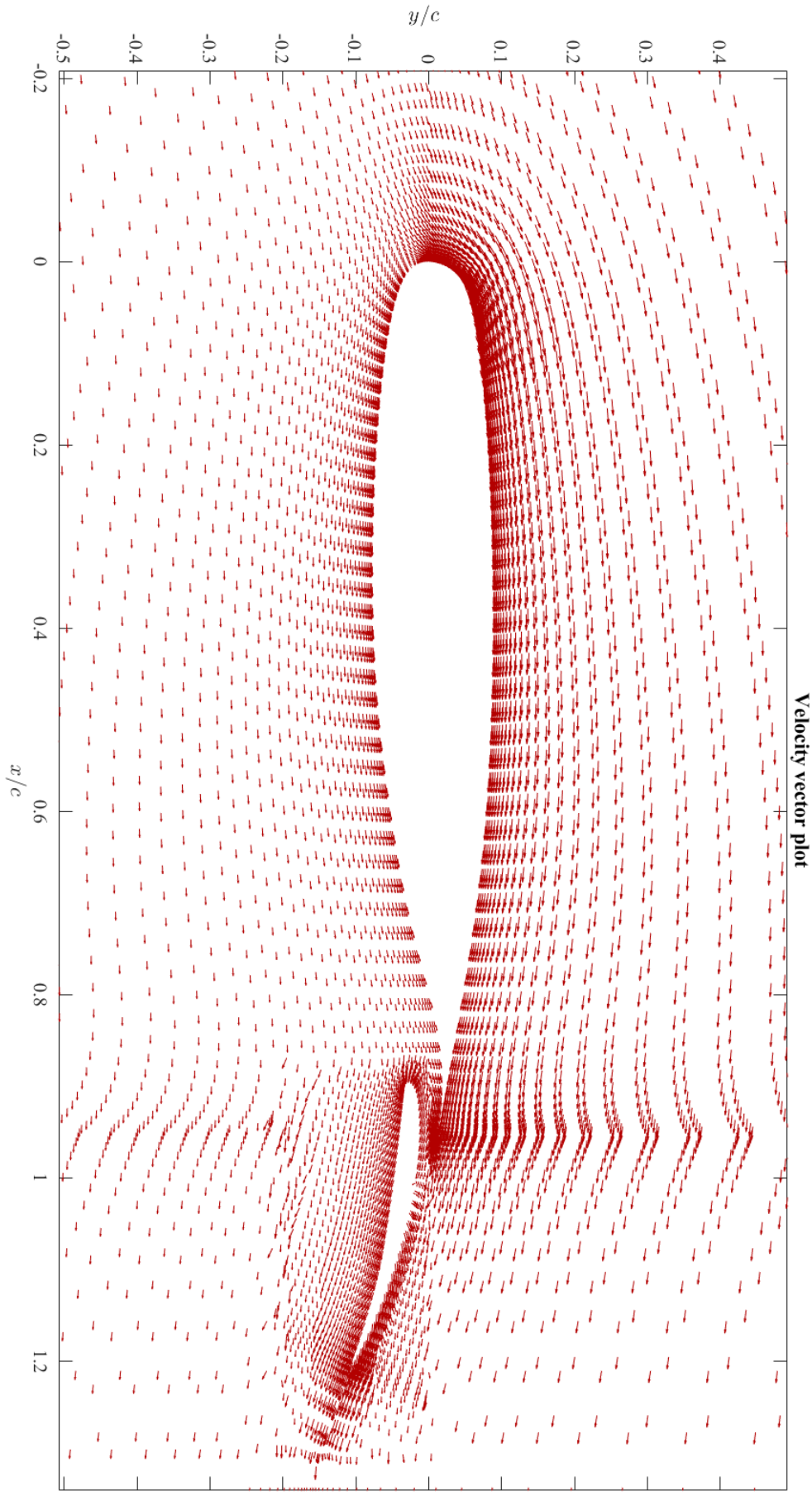



*Figure 4.17 – Velocity vector plot of flowfield over the entire NLR/Boeing aerofoil*

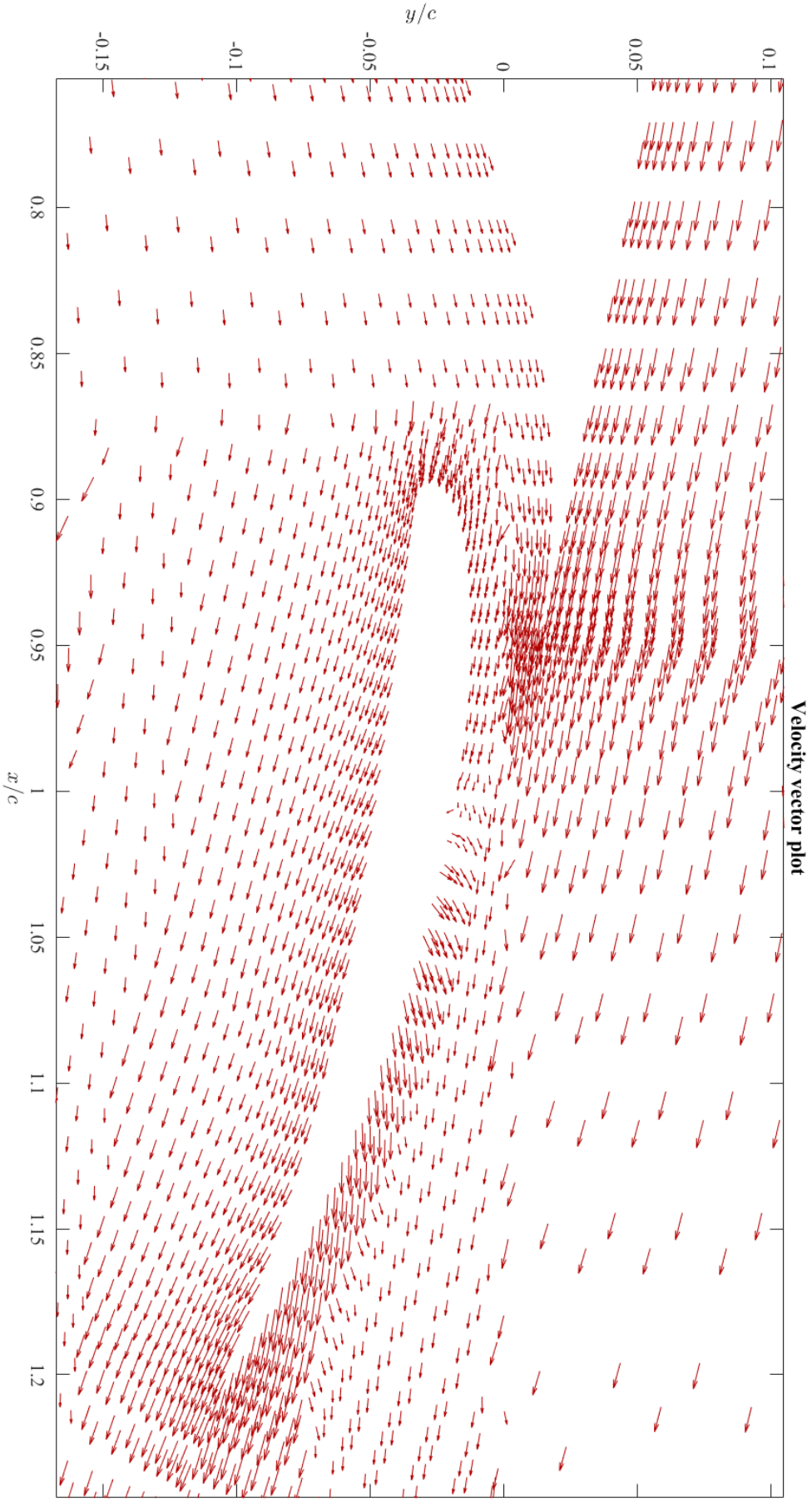



*Figure 4.18 – Close up of velocity vector plot of flowfield over the near flap region of the NLR/Boeing aerofoil*

### 4.3.2 The two-element SKF aerofoil

The two-element SKF aerofoil required 25,000 iterations in order to converged and the stability criterion was a $CFL$ equivalent to 1. The flow conditions used was a Mach number of 0.6 at an angle of incidence of $2^o$. Figure 4.19 And 4.20 displays the NLR Boeing aerofoil unstructured grid and a close-up of the mesh at the near-flap region.

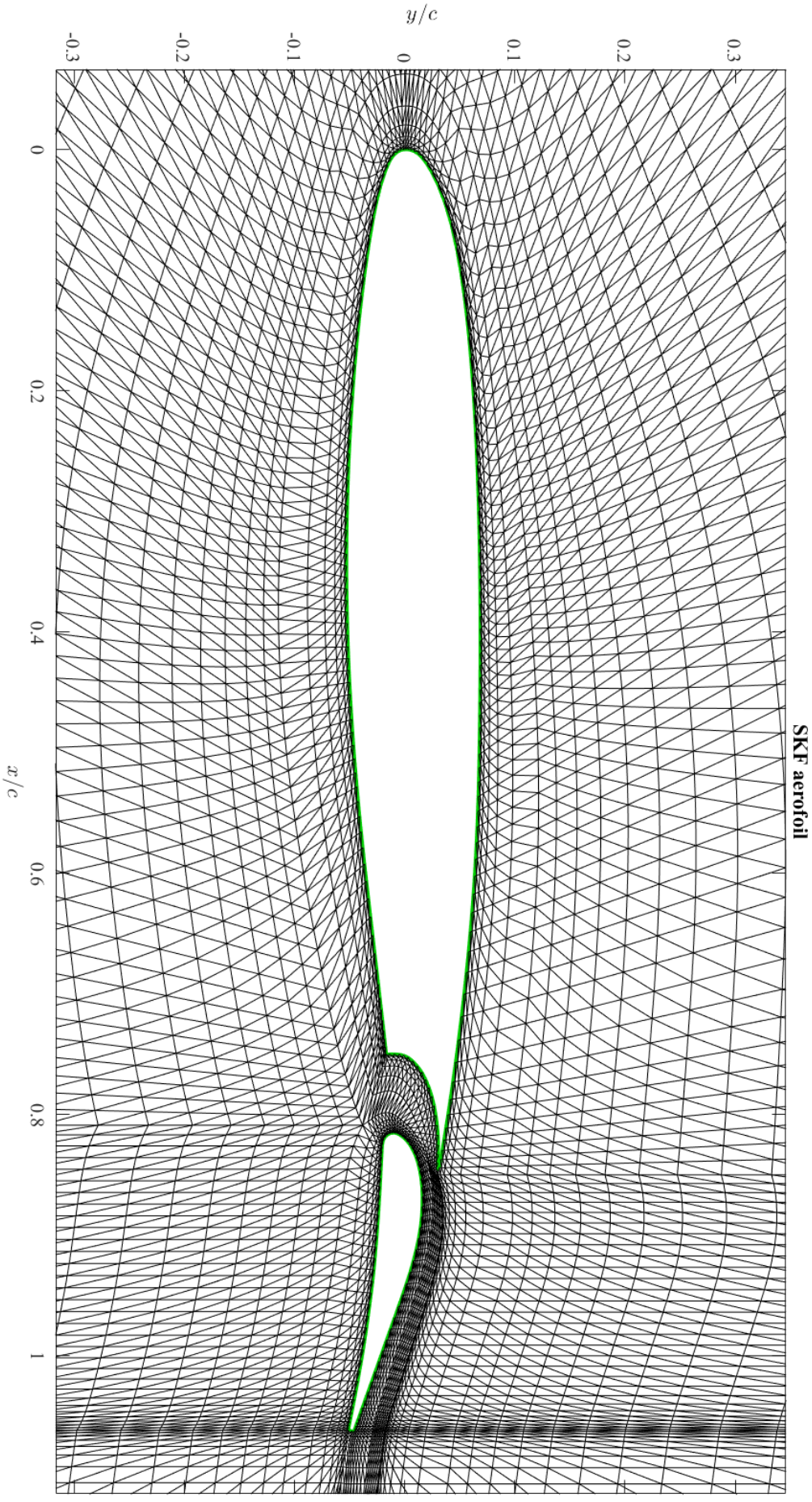



*Figure 4.19 - Two-element SKF computational grid, where the dark green lines indicate the aerofoil surfaces. Grid courtesy of Dr George Barakos.*

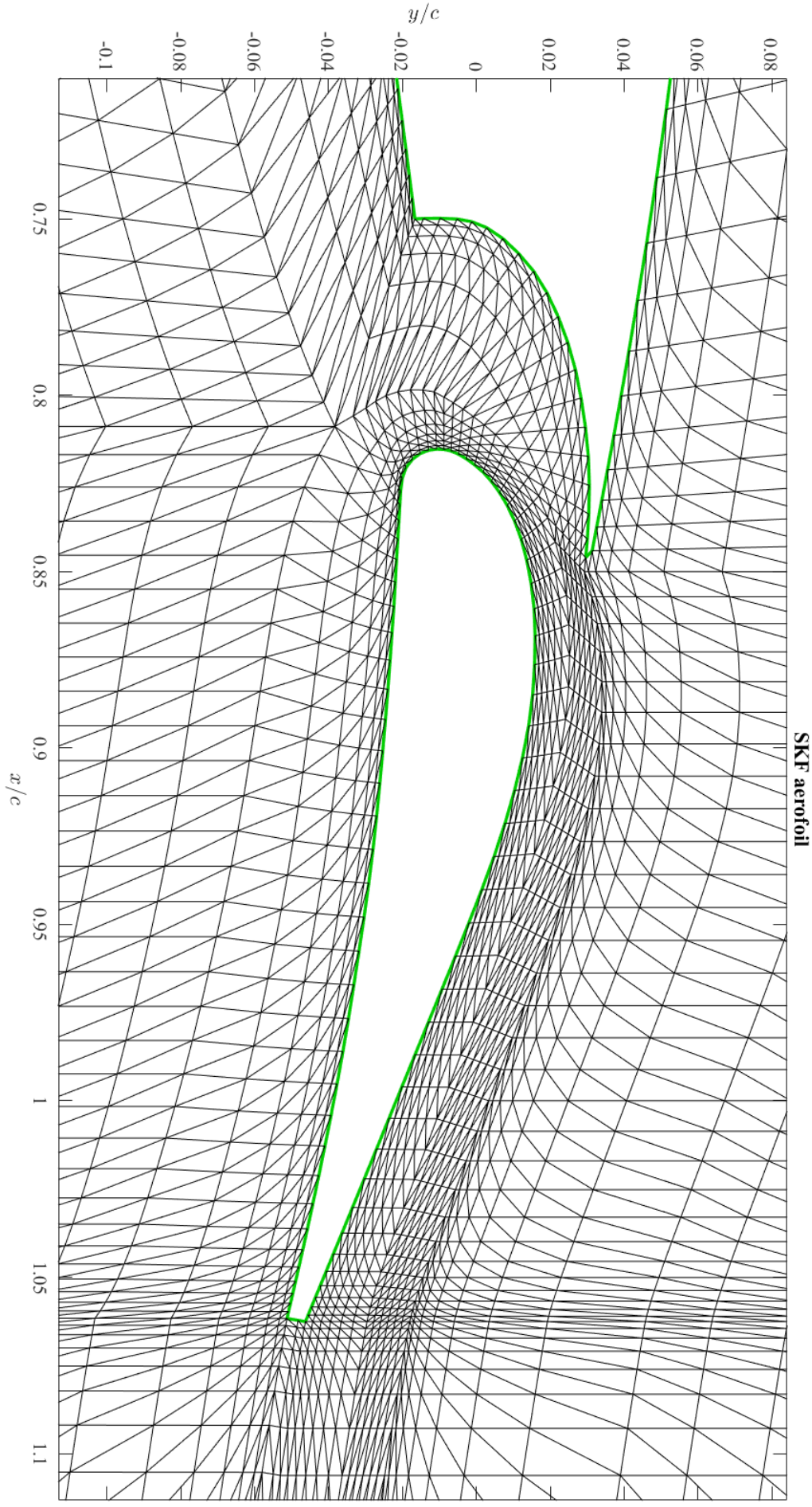



*Figure 4.20 – Close up of the mesh at the near-flap region*

The results for the two-element SKF aerofoil for the coefficient of lift, drag and moment are displayed in table 4.4.

*Table 4.4 – SKF aerofoil coefficients*

| Coefficients | R-K code |
|---|---|
| $C_L$ | 1.5302 |
| $C_D$ | 0.0300 |
| $C_M$ | -0.2855 |

Figure 4.21, 4.22 and 4.23 display the surface $C_P$ and flowfield of the entire aerofoil and near flap region respectively.

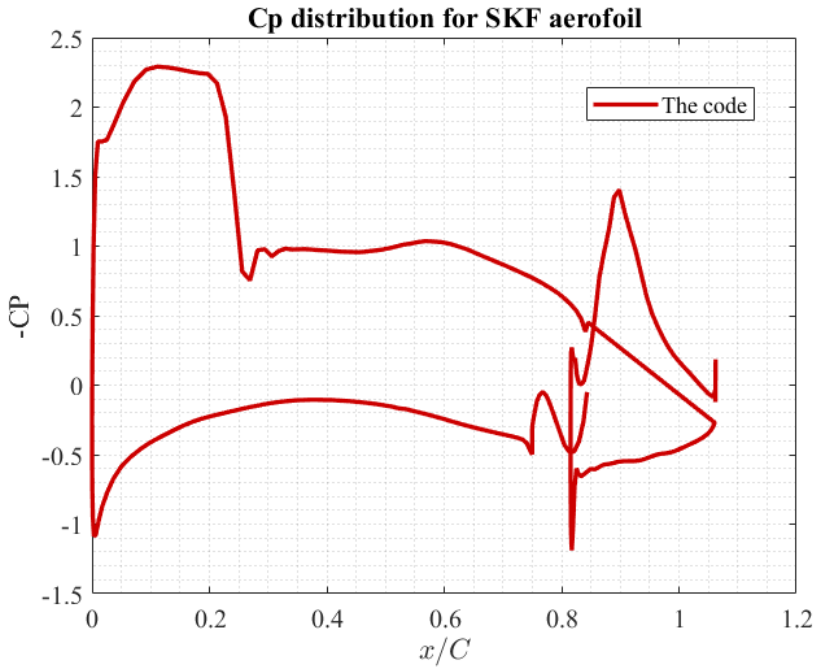



*Figure 4.21 – Coefficient of pressure distribution over the SKF aerofoil.*

This is likewise to the NLR/Boeing aerofoil, however, the pressure coefficient at the gap region is sporadic which is due to the circulating flow in between the aerofoil and high-lift device as can be seen in figure 4.23 of the velocity vector plot in the near-flap region.

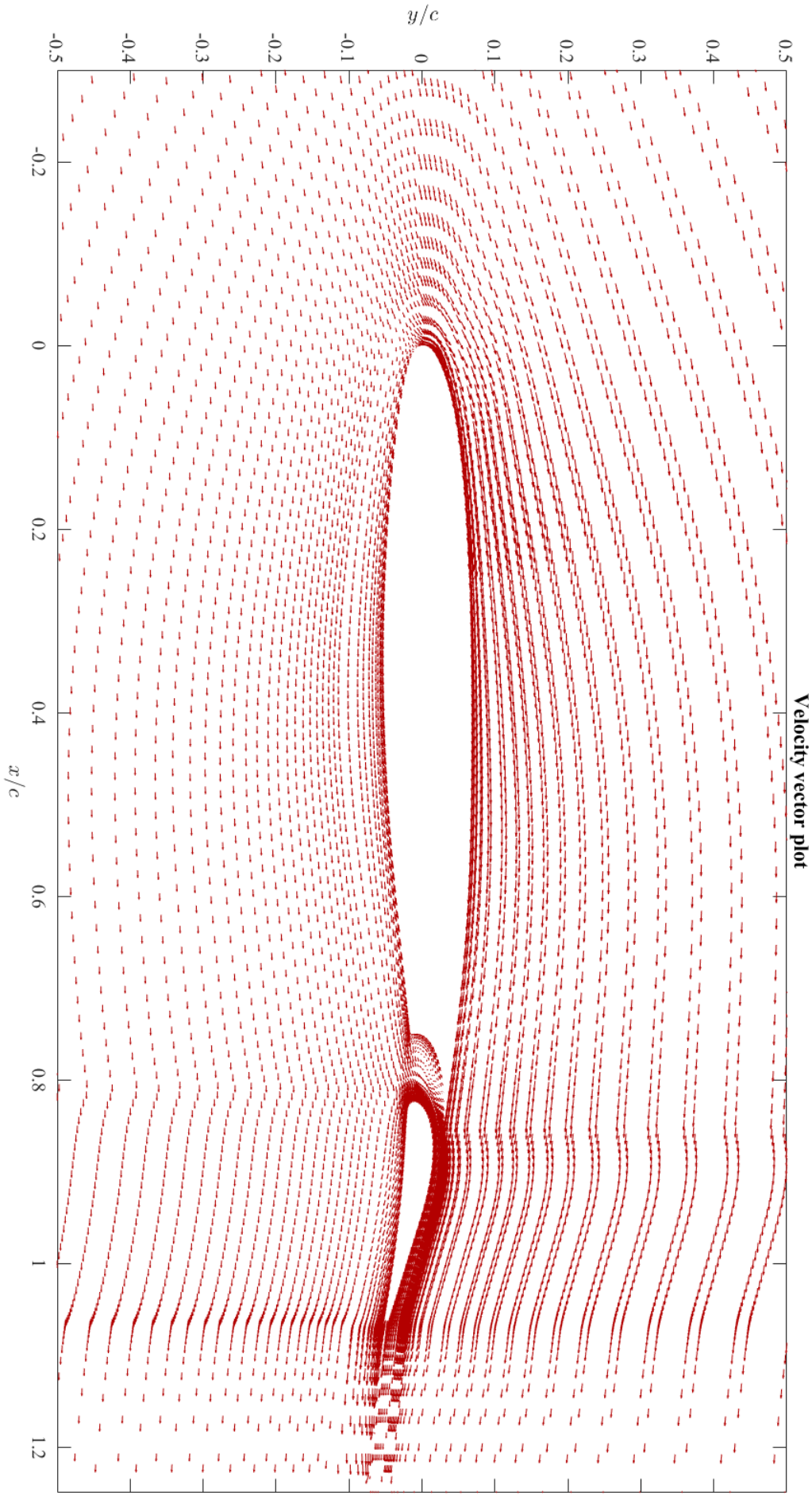



*Figure 4.22 – Velocity vector plot of flowfield over the entire SKF aerofoil.*

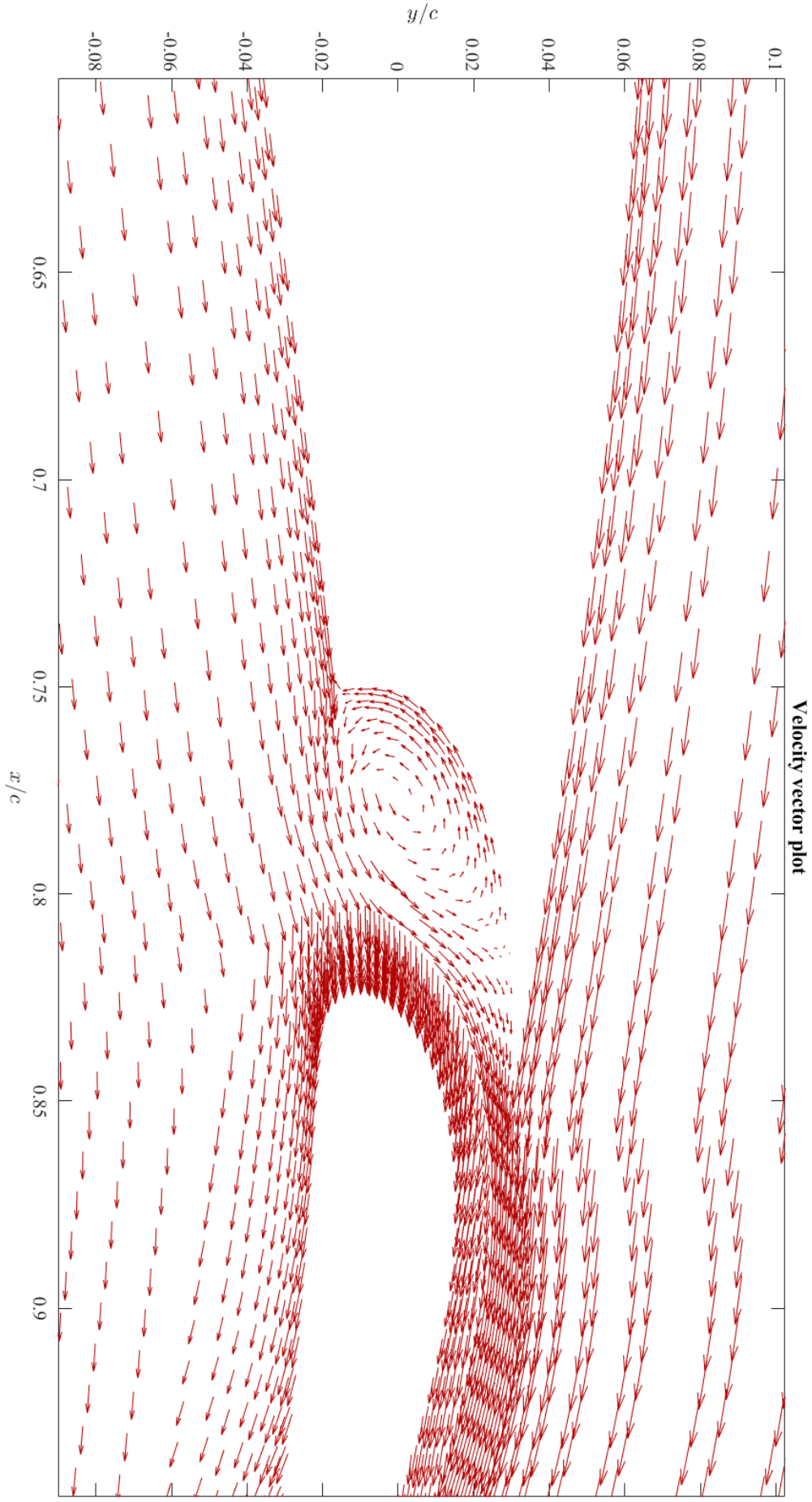



*Figure 4.23 – Close up of velocity vector plot of flowfield in near-flap region of the SKF aerofoil*

### 4.3.3 Discussion of the flowfield of the NLR/Boeing and SKF aerofoil

From both the flow fields, the most notable feature is the rotational flow within the gap of the SKF aerofoil and high-lift device. This is questionable as the Euler equation does not permit the production or destruction of rotational flows; vorticity is conserved and should convect freely if a rotational flow was originally inputted [13]. This conservation is due to the inviscid nature of the flow where no diffusion or dissipation should occur as they are viscous effects. However, the artificial viscosity added, and the inherent numerical diffusion and dissipation caused by the spatial discretization of the Euler equation [13], causes the Euler equation to be analogous to reality. This hypothesis applied to the aerofoils simulated is in accordance.

From the NLR/Boeing flowfield (see figure 4.18), the velocity vectors are not tangent to the surface of the high-lift device. For potential flows, which are irrotational, the flow vectors must be tangent to the surface in accordance to the Kutta-condition. Since no rotational component was originally inputted, the flow should behave like a potential flow at the surfaces due to no production of rotational flows being permitted in the Euler equation. At the solid boundaries, the presence of a normal component within the velocity vectors indicates that a rotational flow has been produced, and therefore, confirms the presence of viscosity. From the SKF flowfield (see figure 4.23), a vortex is clearly formed within the gap between the aerofoil and high-lift device. Likewise to the NLR/Boeing aerofoil, this is also due to the artificial viscosity and therefore the Eulerian flow eludes to a real flow.

In conclusion, a rotational flow is clearly produced for both cases despite no rotational flow component being originally inputted, therefore, the Eulerian flow behaves like a real flow due to the artificial viscosity.

# 5. References


[1] R. M. Cummings, W. H. Mason, S. A. Morton and D. R. McDaniel, Applied Computational Aerodynamics - A Modern Engineering Approach, Cambridge: Cambridge University Press, 2015.

[2] A. Syrakos and A. Goulas, "ESTIMATE OF THE TRUNCATION ERROR OF FINITE VOLUME DISCRETISATION OF THE NAVIER-STOKES EQUATIONS ON COLOCATED GRIDS," *International Journal for Numerical Methods in Fluids,* vol. 50, pp. 103-130, 2006.

[3] A. Busse, "Computational Fluid Dynamics lecture notes - Chapter 3," University of Glasgow, Glasgow, 2017.

[4] L. Lynch, "Numerical Integration of Linear and Nonlinear Wave Equations," *Dissertations, Theses, and Student Research Papers in Mathematics,* vol. 16, 2004.

[5] A. Busse, "Computational Fluid Dynamics Lecture notes - Chapter 5," University of Glasgow, Glasgow, 2017.

[6] A. Busse, "Computational Fluid Dynamic Lecture notes - Chapter 6," University of Glasgow, Glasgow, 2017.

[7] G. Barakos, "Fundamental Central Difference Methods - Chapter 5 Lecture notes," University of Glasgow, Glasgow, 2018.

[8] A.Jameson and D. Mavriplis, "Finite Volume Solution of the Two-DimensionaI Euler Equations on a Regular Triangular Mesh," *Aerospace Sciences Meeting,* vol. 23, 1985.

[9] G. Barakos, "Implementation of the R-K method," University of Glasgow, Glasgow, 2018.

[10] Mathworks, "Mathworks," Matlab, 16 December 2016. [Online]. Available: https://uk.mathworks.com/matlabcentral/answers/317128-does-matlab-use-all-cores-by-default-when-running-a-program. [Accessed 20 May 2018].

[11] D. Marten and J. Wendler, “QBlade guidelines v0.6,” Qblade, Berlin, 2013.

[12] C. v. Dam, “The aerodynamic design of multi-element high-lift systems for transport airplanes,” *Progress in Aerospace Sciences,* vol. 38, pp. 101-144, 2002.

[13] J. G. Leishman, Principles of Helicopter Aerodynamics, 2nd ed., Cambridge: Cambridge University Pres., 2006, pp. 776-777.

[14] G. Barakos, “Project 1,” University of Glasgow, Glasgow, 2018.

# 6. Appendices

## 6.1 Assignment 1 code

### 6.1.1 Main code

```
%% Assignment 1: Nicholas Cheong

% NOTE: init must be set to 0 to start or reset the code.

switch init
    case 0
clc, clear, close all
%% Controls
% Numerical controls
CTRL.RF = 0.1;      % Relaxation factor
CTRL.OC = 10^-3;    % Order of convergence of the root mean square of residuals.

% Note, any change to controls needs reinitilizing
% Visual controls: 'on' or 'off'
CTRL.residualvisualization = 'off';     % Show residual progression. Greatly increases
computation time
CTRL.streamlinevis         = 'on';      % Show streamline plot
CTRL.colorplotPSIvis       = 'on';      % Show PSI colour plot
CTRL.colorplotUvis         = 'on';      % Show U velocity colour plot
CTRL.colorplotVvis         = 'on';      % Show V velocity colour plot
CTRL.colorplotvelmagvis    = 'on';      % Show Velocity mag colour plot
CTRL.IDvis                 = 'on';      % Show ID matrix colour plot
CTRL.vectorvis             = 'on';      % Show velocity vector plot
CTRL.stagpointvis          = 'on';      % Display stag points in vel vector
CTRL.gridplot              = 'no';      % Plot the grid will generating ID matrix

% Inner fluid finite difference scheme % 1 = Forward diff, 2 = Central diff,
% For velocity                          3 = backward diff
CTRL.velxDS = 2;    % x velocity
CTRL.velyDS = 2;    % y velocity

% Grid settings
CTRL.NpY = 41;      % Number of points in y direction,
```

```
                    % This is used to calculate X points within the code to ensure
                    % points always lie on circle x lower and upper limits
                    % and top

%% Notes on code
% Structured matrices:
% GP. = Grid parameter
% NP. = Normilized parameter
% RVP = Real Value parameter
%% Discretization of spatial domain and main loop for calculating potential flow

% Since using cartesian grid, the spacing is similar and therefore
% coordinates of neighbouring points can be found using a known point.

% Geometric parameters (Physical domain)
RVP.h  = 2;      % Height of domain
RVP.br = 8;      % Width of domain
RVP.IH = 0.25;   % Inlet height

% Define mesh parameters
% THIS MUST BE ODD to have point on circle top, and side centres.
% An odd number also gives a less floating points. eg. 21 is divides the
% grid by 20, an even number. Thus reducing errors.
GP.Yp = CTRL.NpY;     % Grid points

% Define mesh spacing
% Since before circle = 2 and height, and inside circle is = 2;
RVP.Yspace =  RVP.h/(GP.Yp - 1);
RVP.Xspace =  RVP.Yspace;
GP.Xp      = (RVP.br/RVP.Xspace) + 1;
%% Boundary initilization
% Need to define the boundaries in which dictates the value of flow
% potential, psi:
% At cylinder surface: psi = 0.
% At walls, psi = 1;
% At outlet, -dPsi/dx = 0; (zero y velocity)

% Assume an initial value of psi for every grid point
% Initilizing PSI matrix
psi_initial = 0.25;
PSImatrix = repmat(psi_initial,GP.Xp,GP.Yp);
%% GEOMETRY
%% Predefine circular geometry
% Need to define circular geometry to then define the boundary along the cylinder.
RVP.cir_xo  = 3;    %a (Centre,x) (3 m)
RVP.cir_yo  = 0;    %b (Centre,y)
RVP.cir_rad = 1;    %r (Radius,r)

% Circular bounds in x direction
RVP.cir_xLB = RVP.cir_xo - RVP.cir_rad;
RVP.cir_xUB = RVP.cir_xo + RVP.cir_rad;
%% Calculating circle geometry as physical coordinates
% This is for locating points ALLIGNED WITH THE GRID POINTS
RVP.x_cir = RVP.cir_xLB: RVP.Xspace: RVP.cir_xUB;

% Calculating alligned point y coords
for i =  1:length(RVP.x_cir)
    RVP.y_cir(1,i) = ( (((RVP.cir_rad^2)- ((RVP.x_cir(i) - RVP.cir_xo)^2))^(1/2)) ) +
RVP.cir_yo;
end

% Fine circle to show a "representative" true circle in plot
RVP.x_cirfine = RVP.cir_xLB: 0.01: RVP.cir_xUB;
% Calculating fine curve y coords
for i = 1:length(RVP.x_cirfine)
RVP.y_cirfine(1,i) = ( (((RVP.cir_rad^2)- ((RVP.x_cirfine(i) - RVP.cir_xo)^2))^(1/2)) )
+ RVP.cir_yo;
end

%% Circle in terms of discrete points
GP.cir_xo   = (RVP.cir_xo/RVP.Xspace) + 1;  % Circle centre grid point x
GP.cir_yo   = 1;                            % Circle centre grid point y
```

```
GP.cir_rad  = RVP.cir_rad/RVP.Xspace;       % Number of grid points in quater circle in
x (same as y)
GP.cir_xLB  = GP.cir_xo - GP.cir_rad;       % LB of circle in grid points
GP.cir_xUB  = GP.cir_xo + GP.cir_rad;       % Upper bound of circle in grid points

GP.x_cir  = GP.cir_xLB:1:GP.cir_xUB;
for i = 1:length(GP.x_cir)
     GP.y_cir(1,i) = ( (((GP.cir_rad^2)- ((GP.x_cir(i) - GP.cir_xo)^2))^(1/2)) ) +
GP.cir_yo;
end

%% Pre-Normilizing variables for input
% NORMALIZED circular x bounds
NP.cir_xLB = RVP.cir_xLB/RVP.br;
NP.cir_xUB = RVP.cir_xUB/RVP.br;
% NORMILIZED alligned points of circle
NP.x_cir = RVP.x_cir./RVP.br;
NP.y_cir = RVP.y_cir./RVP.h;
% NORMILIZED circle centre
NP.cir_xo = RVP.cir_xo/RVP.br;
NP.cir_yo = RVP.cir_yo/RVP.h;
% NORMALIZED fine curve coordinates
NP.x_cirfine = RVP.x_cirfine./RVP.br;
NP.y_cirfine = RVP.y_cirfine./RVP.h;
% NORMILIZED Xspace and Yspace
NP.Xspace = RVP.Xspace/RVP.br;
NP.Yspace = RVP.Yspace/RVP.h;

%% Predefining matrices
ID = ones(GP.Xp,GP.Yp);

%%  Looping for each respective point starting from origin and moving up the y, then
along the x.
% Note, not indexing as this would store every value to therefore REDUCE
% MEMORY USAGE

%Start from origin
WV.Cxo = 0;
WV.Cyo = 0;
for i = 1:GP.Xp     %Along ith direction

    if i == 1
    %Initilizing grid from origin [0,0]
        WV.Cx = WV.Cxo;
    else
        %Overwriting data for next coordinate
        WV.Cx = WV.Cx + NP.Xspace;
    end
    % Initilizing upwards y coordinates for each x coordinate
    % Note, not indexing as this would store every value of coordinate
    for j = 1:GP.Yp
        if j == 1
            %Initilizing grid from origin [0,0]
            WV.Cy = WV.Cyo;
        else
            %Overwriting data for next coordinate
            WV.Cy = WV.Cy + NP.Yspace;
        end
        % Need a statement which detects if the points are within the circlular x
bounds.
        if WV.Cx > NP.cir_xLB && WV.Cx < NP.cir_xUB

            % Get circlular geometry (y coords)
            % The limitations on either side of the Cx coord is needed to
            % avoid round error which is making the code unable to read e.g
            % 0.2632 = 0.2632 ...
            WV.Cyind = find(NP.x_cir > (WV.Cx - 0.5*NP.Xspace) & NP.x_cir < (WV.Cx +
0.5*NP.Xspace));
            WV.Cy_upbound = NP.y_cir(WV.Cyind);

            if WV.Cy_upbound > WV.Cy + 0.1*NP.Yspace
                skippoint = 1;  % No need to analyse within the geometry itself.
```

```
            else
                skippoint = 0;
            end
        else
            skippoint = 0;
        end

        %% Sorting grid points into an ID matrix to identify boundaries
        switch skippoint
            case 0
        %  Defining boundaries psi values
            [WV] = IdentityGP(i,j,WV,RVP,NP,GP,ID);
            ID(i,j) = WV.IDval;
            case 1
            % Setting ID for this grid point to a reference value the ID
            % function can read and prep neighbouring cylinder point.
            ID(i,j) = 2;
            % Overide of left side of cylinder ID values (Cannot impliment
            % ID(i+1,j) as i+1 is not created yet
            % Defining points left of the circle = 0 if they are = to -1.
            if i > 1 && ID(i,j) == 2 && ID(i-1,j) == -1 || ID(i-1,j) == -2
                    ID(i-1,j) = 0;
            end
        end
        % Plotting the grid to confirm it is working
        switch CTRL.gridplot
            case 'on'
        if skippoint == 0
        figure(1)
        if ID(i,j) == -1
            plot(WV.Cx*RVP.br,WV.Cy*RVP.h,'-ob','MarkerSize',3,'LineWidth',3);
        elseif ID(i,j) == -2
            plot(WV.Cx*RVP.br,WV.Cy*RVP.h,'-oy','MarkerSize',3,'LineWidth',3);
        elseif ID(i,j) == 3
            plot(WV.Cx*RVP.br,WV.Cy*RVP.h,'-om','MarkerSize',3,'LineWidth',3);
        elseif ID(i,j) == 0
            plot(WV.Cx*RVP.br,WV.Cy*RVP.h,'-or','MarkerSize',3,'LineWidth',3);
        elseif ID(i,j) == 1
            plot(WV.Cx*RVP.br,WV.Cy*RVP.h,'-og','MarkerSize',3,'LineWidth',3);
        elseif ID(i,j) == 2
            plot(WV.Cx*RVP.br,WV.Cy*RVP.h,'-ok','MarkerSize',3,'LineWidth',3);
        end
        hold on; title('Grid Generation'); xlabel('X'); ylabel('Y'); pause(0.001)
        end
            case 'off'
        end
    % End of jth line
    end
% End of ith line
end
% Plotting circle in mesh generate figure
switch CTRL.gridplot
    case 'on'
    figure(1)
    plot(NP.x_cir,NP.y_cir);
    hold on
    plot(NP.x_cirfine,NP.y_cirfine,'r');
    case 'off'
end
% New loop as you do not need to compute the coordinates of the mesh again.
% Saves computational time. This is where the iterations start.
% Reminder:   ID = 1  => Wall boundary
%             ID = 2  => Inside cylinder
%             ID = 0  => Cylinder neighbouring points
%             ID = -1 => General fluid
%             ID = -2 => Sym line
% Each has its own equation to solve for psi.

% Need to predefine different values of a,b,c,d for ID = 0;

% Function to calcualte
[a,b,c,d] = ABCDdistances(RVP,GP,NP,ID);
```

```
% Now that the arbitrary distances and ID matrix have been developped,
% calculating for psi can now be executed.

% Looping. Three for loops: (1) iterations, (2) ith direction of points,
% (3) jth direction of points.
% create fictous point after i = 25. or create new ID for boundary.

clear i j
% Code will now start from case 1 below, to "re-mesh", re-input init = 0;
init = 1;
    case 1
        clear PSImatrix u_matrix v_matrix PSImatrix_tm1 Res ResRMS CMUplot CMVplot
        close all
        %Re-initializing psi matrix
        PSImatrix = repmat(psi_initial,GP.Xp,GP.Yp);
end

% Initilizing while loop
[WV.IDrow,WV.IDcol] = size(ID);
it = 0;
convergence = 0;
while convergence < 1

    it = it + 1;
    fprintf('\nIteration: %0.1f',it)
    for i = 1:WV.IDrow
        for j = 1:WV.IDcol
            % Need to sort or which psi equation will be calculated using
            % the identities.
            % Switching can be generally more efficient in this case as the code does
            % not check every conditional statement.
            switch ID(i,j)
                case 1 % Wall boundary
                    PSImatrix(i,j) = 1;

                case -1 % General fluid
                    % Define i+1, i-1, j+1, j-1

                    WV.PSI_ip1_j = PSImatrix(i+1,j);
                    if i == 1 % Inlet
                        WV.PSI_im1_j = WV.PSI_ip1_j;   %for now % This cold maybe
control velocity?
                    else
                        WV.PSI_im1_j = PSImatrix(i-1,j);
                    end
                    WV.PSI_i_jp1 = PSImatrix(i,j+1);
                    WV.PSI_i_jm1 = PSImatrix(i,j-1);
                    %Calculating new value of PSI for matrix
                    PSImatrix(i,j) = (WV.PSI_im1_j + WV.PSI_ip1_j + WV.PSI_i_jm1 +
WV.PSI_i_jp1)/4;
                case 0  % Circle boundary
                    % If arbitrary distances are less than the grid spacing
                    % (constant value) then grid line is where psi should =
                    % 0. if not, then the grid line is not connected to the
                    % circle.
                        if a(i,j) < NP.Xspace % Psi A
                            WV.PSI_im1_j = 0;
                        else
                            WV.PSI_im1_j = PSImatrix(i-1,j);
                        end

                        if b(i,j) < NP.Xspace % Psi B
                            WV.PSI_ip1_j = 0;
                        else
                            WV.PSI_ip1_j = PSImatrix(i+1,j);
                        end

                        if c(i,j) < NP.Yspace % Psi C
                            WV.PSI_i_jm1 = 0;
                        else
                            WV.PSI_i_jm1 = PSImatrix(i,j-1);
```

```
                        end

                        if d(i,j) < NP.Yspace  % Psi D
                            WV.PSI_i_jp1 = 0;
                        else
                            WV.PSI_i_jp1 = PSImatrix(i,j+1);
                        end

                        % a,b,c,d need to be ensured not to be zero when refining grid
such
                        % that NaNs dont occur
                        if c(i,j) ~= 0 && a(i,j) ~= 0 && b(i,j) ~=0 && c(i,j) ~= 0 &&
d(i,j) ~= 0
                        WV.DEN = ((1/(a(i,j)*b(i,j))) + (1/(c(i,j)*d(i,j))));
                        WV.PSI_A = (WV.PSI_im1_j/(a(i,j)*(a(i,j)+b(i,j))));
                        WV.PSI_B = (WV.PSI_ip1_j/(b(i,j)*(a(i,j)+b(i,j))));
                        WV.PSI_D = (WV.PSI_i_jp1/(d(i,j)*(c(i,j)+d(i,j))));
                        WV.PSI_C = (WV.PSI_i_jm1/(c(i,j)*(c(i,j)+d(i,j))));
                        WV.NUM = WV.PSI_A + WV.PSI_B + WV.PSI_C + WV.PSI_D;
                        PSImatrix(i,j) = WV.NUM/WV.DEN;

                        % Considering centre point where c = 0
                        else
                            PSImatrix(i,j) = 0;
                        end

                case  2  % Inside circle (Leave blank)
                    PSImatrix(i,j) = NaN;

                case -2  % Symmetry line
                    PSImatrix(i,j) = 0;  % Psi = Uy ;  y  = 0.

                case  3  % Outlet
                    WV.PSI_im1_j = PSImatrix(i-1,j);
                    WV.PSI_ip1_j = WV.PSI_im1_j;
                    WV.PSI_i_jp1 = PSImatrix(i,j+1);
                    WV.PSI_i_jm1 = PSImatrix(i,j-1);
                    PSImatrix(i,j) = (2*WV.PSI_im1_j + WV.PSI_i_jp1 + WV.PSI_i_jm1)/4;
            end
        end
    end

    %% Convergence criterion
    [WV.rows,WV.cols] = size(PSImatrix);
    if it == 1
        PSImatrix_tm1 = repmat(10,WV.rows,WV.cols);
    end
    % Storing value of PSImatrix to compare with next PSImatrix values.
    % Conv is the absolute value of all grid points.
    WV.conv = abs(PSImatrix_tm1 - PSImatrix);
    PSImatrix_tm1 = PSImatrix;

    for iind = 1:WV.rows
        for jind = 1:WV.cols
            % --Residuals--
            WV.Res(iind,jind) = WV.conv(iind,jind);
            if isnan(WV.Res(iind,jind)) % Setting interior point of circle residuals to
zero
                WV.Res(iind,jind) = 0;  % They are not analysed
            end
        end
    end
    % Root mean square of residuals
    ResRMS = ((sum(sum(WV.Res)))/(GP.Xp*GP.Yp))^(1/2);
    switch CTRL.residualvisualization
        case 'on'
    figure(5)
    semilogy(it,ResRMS,'or')
    hold on
    title('Residuals')
    xlabel('Iteration')
    ylabel('Absolute error')
```

```
    pause(0.001)
        case 'off'
    end

    if ResRMS < (CTRL.OC)
        convergence = 1;
    end
    if it >4000
        if it > 8000
        error('Max iterations')
        end
        if it > 6000
            CTRL.OC = 10^-2;
        end
    end
end
clear WV.IDval
%% Post-processing:
% Velocity calculation:
for i = 1:GP.Xp
    for j = 1:GP.Yp
            WV.IDval = ID(i,j);
            [u,v]=Velocity2(i,j,WV,RVP,GP,NP,CTRL,PSImatrix);
            u_matrix(i,j) = u;
            v_matrix(i,j) = v;
    end
end

%% Plotting
plt.CMx = 1:GP.Xp+1;
plt.CMy = 1:GP.Yp+1;
% Adding NaN line to data as pcolor decides to delete data when plotting
% (Upper and right line)
plt.CMUplot   = NaN(GP.Yp+1,GP.Xp+1);
plt.CMVplot   = NaN(GP.Yp+1,GP.Xp+1);
plt.IDplott   = NaN(GP.Yp+1,GP.Xp+1);
plt.CMPSIplot = NaN(GP.Yp+1,GP.Xp+1);

clear i j
% Rotating matrices for plot visulization
for i = 1:WV.rows
    for j = 1:WV.cols
        plt.CMPSIplot(j,i) = PSImatrix(i,j);
        plt.CMUplot  (j,i) = u_matrix(i,j);
        plt.CMVplot  (j,i) = v_matrix(i,j);
        plt.IDplott  (j,i) = ID(i,j);
    end
end
% Velocity magnitude
plt.VMplot = ((plt.CMUplot.^2)+(plt.CMVplot.^2)).^(1/2);

switch CTRL.colorplotPSIvis
    case 'on'
figure(10)
set(gcf,'color','w')
pcolor(plt.CMx(1:end-1)/length(plt.CMx),plt.CMy(1:end-1)/length(plt.CMy),
plt.CMPSIplot(1:end-1,1:end-1))
colormap jet; colorbar;
shading interp
 xlabel('$$x/B$$','Interpreter','Latex');ylabel('$$y/H$$','Interpreter','Latex')
title('PSI (Stream function)')
daspect([0.35 1 1])
    case 'off'
end
switch CTRL.colorplotUvis
    case 'on'
figure(11)
set(gcf,'color','w')
pcolor(plt.CMx(1:end-1)/length(plt.CMx),plt.CMy(1:end-
1)/length(plt.CMy),plt.CMUplot(1:end-1,1:end-1))
colormap jet; colorbar
shading interp
```

```
xlabel('$$x/B$$','Interpreter','Latex');ylabel('$$y/H$$','Interpreter','Latex')
daspect([0.35 1 1])
title('u (X-velocity)')
    case 'off'
end
switch CTRL.colorplotVvis
    case 'on'
figure(12)
set(gcf,'color','w')
pcolor(plt.CMx(1:end-1)/length(plt.CMx),plt.CMy(1:end-
1)/length(plt.CMy),plt.CMVplot(1:end-1,1:end-1))
colormap jet; colorbar
shading interp
 xlabel('$$x/B$$','Interpreter','Latex');ylabel('$$y/H$$','Interpreter','Latex')
daspect([0.35 1 1])
title('v (Y-velocity)')
    case 'off'
end
switch CTRL.IDvis
    case 'on'
figure(13)
set(gcf,'color','w')
pcolor(plt.CMx,plt.CMy,plt.IDplott)
title('ID')
colorbar; daspect([1 1 1])
    case 'off'
end
switch CTRL.colorplotvelmagvis
    case 'on'
figure(15)
set(gcf,'color','w')
pcolor(plt.CMx(1:end-1)/length(plt.CMx),plt.CMy(1:end-
1)/length(plt.CMy),plt.VMplot(1:end-1,1:end-1))
colormap jet; colorbar
shading interp
daspect([0.35 1 1])
title('Velocity Magnitude');
xlabel('$$x/B$$','Interpreter','Latex');ylabel('$$y/H$$','Interpreter','Latex')
set(gca,'FontName','Times New Roman','FontSize',12);
    case 'off'
end
switch CTRL.vectorvis
    case 'on'
figure(14)
set(gcf,'color','w')
[plt.x,plt.y] = meshgrid((1:1:GP.Xp+1),(1:1:GP.Yp+1));
hold on
quiver(plt.x,plt.y,plt.CMUplot,plt.CMVplot)
hold on
daspect([1 1 1])
    switch CTRL.stagpointvis
        case 'on'
    plot(GP.x_cir(1),GP.y_cir(1),'or','LineWidth',1,'MarkerSize',7)
    hold on
    plot(GP.x_cir(end),GP.y_cir(end),'ob','LineWidth',1,'MarkerSize',7)
    hold on
    plot(1,GP.Yp,'og','LineWidth',1,'MarkerSize',7)
    hold on
    plot(GP.x_cir,GP.y_cir,'r')
    legend('Velocity vector','Upstream stagnation point','Downstream stagnation
point','Corner stagnation point')
    title('Velocity vector plot')
        case 'off'
    plot(GP.x_cir,GP.y_cir,'r')
    title('Velocity vector plot')
    legend('Velocity vector')
    end
    case 'off'
end
switch CTRL.streamlinevis
    case 'on'
    legend('off')
```

```
    hold on
    plt.starty = 1:0.1:2;
    plt.startx = ones(size(plt.starty));
    streamline(plt.x,plt.y,plt.CMUplot,plt.CMVplot,plt.startx,plt.starty)
    case 'off'
end

% Plotting velocity profile before and after circle in y and x
plt.CMy2 = (plt.CMy)'; % Reorientating matrix for plot
figure(16)
% Lower bound circle plot
plot(plt.CMy2(1:GP.Yp),plt.CMVplot(1:GP.Yp,GP.cir_xLB),'r')
hold on
% Ypper bound circle plot
plot(plt.CMy2(1:GP.Yp),plt.CMVplot(1:GP.Yp,GP.cir_xUB),'b')
legend('Upstream of circle','Downstream of circle')
grid minor
title('Velocity profile of Y-velocity (Conservative flow)')
xlabel('Discretized points')
ylabel('Y-velocity')

figure(17)
set(gcf,'color','w')
plot(((plt.CMy2(1:GP.Yp)-
1).*RVP.Yspace/RVP.h),plt.CMUplot(1:GP.Yp,GP.cir_xLB),'color',[0.7 0 0],'Linewidth',2)
hold on
plot((plt.CMy2(1:GP.Yp)-1).*RVP.Yspace/RVP.h,plt.CMUplot(1:GP.Yp,GP.cir_xUB),'color',[0
0.7 0],'Linewidth',2)
legend('Upstream of circle','Downstream of circle')
grid minor
title('Velocity profile of X-velocity (Conservative flow)')
xlabel('$$y/H$$','Interpreter','Latex'); ylabel('$$u$$','Interpreter','Latex')
set(gca,'FontName','Times New Roman','FontSize',12);

figure(18); clf
set(gcf,'color','w')
plot((plt.CMy2(1:GP.Yp)-1).*RVP.Yspace/RVP.h,plt.CMVplot(1:GP.Yp,1),'color',[0.7 0
0],'Linewidth',2)
hold on
plot((GP.Yp-1)*RVP.Yspace/RVP.h,0,'s','color',[0 0 0.7],'Markersize',7)
legend('Upstream of circle','Corner stagnation point')
grid minor
title('Velocity profile of Y-velocity at inlet/wall')
xlabel('$$y/H$$','Interpreter','Latex'); ylabel('$$v$$','Interpreter','Latex')
set(gca,'FontName','Times New Roman','FontSize',12);

areaBC = trapz(plt.CMy2(1:GP.Yp),plt.CMUplot(1:GP.Yp,GP.cir_xLB));
areaAC = trapz(plt.CMy2(1:GP.Yp),plt.CMUplot(1:GP.Yp,GP.cir_xUB));
```

### 6.1.2 Identity matrix function

```
function [WV] = IdentityGP(i,j,WV,RVP,NP,GP,ID)

   % Walls    => ID =  1
   % Cylinder => ID =  0
   % Nfluid   => ID = -1
   % Sym line => ID = -2
   % In cylin => ID =  2
   % Outlet   => ID =  3

   % Conditional statements to seperate coordinates into related boundary
   % conditions

   %% Outer wall conditions and general fluid
   % Outer wall on upper surface and partial front wall.

   if WV.Cx == 0 && WV.Cy < RVP.IH/RVP.h  % Inlet part
```

```
        WV.IDval = -1;    % Fluid boundary

   elseif WV.Cx == 0 && WV.Cy >= RVP.IH/RVP.h  || WV.Cy ==1  % Front wall

        WV.IDval = 1;
   end
   if j > 1
   if WV.Cx > 0 && WV.Cy < 1 && WV.Cy > 0 && ID(i,j-1) ~=2 && WV.Cx ~= 1    % General
fluid
       WV.IDval = -1;
   end
   end
   % Outlet boundary
   if WV.Cx > 1 - 0.1*NP.Xspace && WV.Cx < 1 + 0.1*NP.Xspace &&...
      WV.Cy > 0 + 0.1*NP.Yspace && WV.Cy < 1 + 0.1*NP.Yspace
       WV.IDval = 3;
   end
   if WV.Cy > 1 - 0.01*NP.Yspace && WV.Cy < 1 + 0.01*NP.Yspace        % Upper wall
       WV.IDval = 1;
   end
   if WV.Cy == 0                                                    % Sym line
       WV.IDval = -2;
   end
   %% Cylinder boundary condition
   %ID is set at 2 for points within the cylinder (skipt points and set outside this
function)
   if j > 1
       if ID(i,j-1) == 2 % If the ID value bellow is set to "Inside circle"
            WV.IDval = 0;
       end
   end

   % Right side of cylinder
   if i > 1 && WV.Cx > NP.cir_xo
       if ID(i-1,j) == 2 % If the ID value to the left is set to "Inside circle"
           WV.IDval = 0;
       end
   end
   % NOTE: Left side considered in in skippoint case

end
```

### 6.1.3 a,b,c,d arbitrary distance function

```
function [a,b,c,d] = ABCDdistances(RVP,GP,NP,ID)

% Need ith and jth loop
[IDrow,IDcol] = size(ID);
for i = 1:IDrow

    for j = 1:IDcol

        if ID(i,j) == 0

            % left and right circle conditions
            if i < GP.cir_xo      % Left side
            a(i,j) = RVP.Xspace;
            b(i,j) = ((GP.cir_xo - i)*RVP.Xspace)-(((RVP.cir_rad^2)-...
                     ((j - GP.cir_yo)*RVP.Yspace)^2)^(1/2));
            c(i,j) = ((j - GP.cir_yo)*RVP.Yspace)-(((RVP.cir_rad^2)-...
                     ((GP.cir_xo - i)*RVP.Xspace)^2)^(1/2));
            d(i,j) = RVP.Yspace;

            elseif i == GP.cir_xo  % Centre
            a(i,j) = RVP.Xspace;
            b(i,j) = RVP.Xspace;
```

```
            c(i,j) = 0;
            d(i,j) = RVP.Yspace;

            elseif i > GP.cir_xo   % Right side
            a(i,j) = ((i - GP.cir_xo)*RVP.Xspace)-(((RVP.cir_rad^2)-...
                     ((j - GP.cir_yo)*RVP.Yspace)^2)^(1/2));
            b(i,j) = RVP.Xspace;
            c(i,j) = ((j - GP.cir_yo)*RVP.Yspace)-(((RVP.cir_rad^2)- ...
                     (((i - GP.cir_xo)*RVP.Xspace)^2))^(1/2));
            d(i,j) = RVP.Yspace;
            end

        % Inside fluid
        else
            a(i,j) = RVP.Xspace;
            b(i,j) = RVP.Xspace;
            c(i,j) = RVP.Yspace;
            d(i,j) = RVP.Yspace;
        end

        % Conditional statements which considers points where arbitrary
        % distances should be equal to grid spacing (e.g. point located on
        % the tangential line of the circle where grid refinement cannot account for
very sharp gradient
        % at circle tangent line parralel to grid lines)
          if a(i,j) > RVP.Xspace
             a(i,j) = RVP.Xspace;
          end
          if b(i,j) > RVP.Xspace
             b(i,j) = RVP.Xspace;
          end
          if c(i,j) > RVP.Yspace
             c(i,j) = RVP.Yspace;
          end
          if d(i,j) > RVP.Yspace
             d(i,j) = RVP.Yspace;
          end
    end
end

% Normilize arbitraty distance matrice
a = a./RVP.br;
b = b./RVP.br;
c = c./RVP.h;
d = d./RVP.h;

end
```

### 6.1.4 Velocity function

```
function [u,v] = Velocity2(i,j,WV,RVP,GP,NP,CTRL,PSImatrix)

%% Converting Psi values to velocity.
% Need to use quiver to plot which requires u,v components of the potential
% flow.

% Forward, backwards or central difference scheme for velocity calculation
% of d psi/dx
% Forward and backwards difference schemes have a truncation error of order
% 1, central difference has order of 2.
%
% u = jth direction = dpsi /dy (use j)
% v = ith direction = -dpsi/dx (use i)

% 1 forward dif
% 2 central
% 3 backwars
```

```
% Chose scheme based on the geometry of known points. Eg, at outlet:
% backwards difference.
% DS for u = DSY;
% DS for v = DSX
% Using ID points
clear DSX DSY

% Wall boundaries
if     WV.IDval ==  1  % Wall boundaries
            if i == 1 && j < GP.Yp     % Along front wall
                DSX = 1;
                DSY = 1;
            elseif i ==1 && j == GP.Yp % Corner
                DSX = 1; %1
                DSY = 3; %3
            elseif i > 1 && j == GP.Yp % Along top wall
                DSX = 3;
                DSY = 3;
            end

elseif WV.IDval == -1 % Fluid
    if i == 1
            DSX = 1;
    else
            DSX = CTRL.velyDS;
    end
            DSY = CTRL.velxDS;

elseif WV.IDval == 0 % Circle boundary
            DSY = 1;
            if i < GP.cir_xo
                DSX = 3;
            elseif i > GP.cir_xo
                DSX = 1;
            elseif i == GP.cir_xo
                DSX = 2;
            end
            if i == GP.cir_xLB && j == 1
                DSY = 1;

            elseif i == GP.cir_xUB && j == 1
                DSY = 1;
            end
    elseif WV.IDval == 3  % Outlet
            DSX = 3;
            if j < GP.Yp
                DSY = 1;
            else
                DSY = 3;
            end
    elseif WV.IDval == -2 % Sym line

            DSY = 1;
            if i == GP.Xp && j == 1
                DSX = 3;
            else
                DSX = 1;
            end

    elseif WV.IDval == 2 % Inside circle
            u = NaN;
            v = NaN;
            DSX = 4;
            DSY = 4;
end

PSI_i_j = PSImatrix(i,j);

switch DSY
    case 1 % Forwards difference
            PSI_i_jp1 = PSImatrix(i,j+1);
```

```
            u =   (PSI_i_jp1 - PSI_i_j)/RVP.Yspace;
    case 2  % Central difference
            PSI_i_jp1 = PSImatrix(i,j+1);
            PSI_i_jm1 = PSImatrix(i,j-1);
            u =   (PSI_i_jp1 - PSI_i_jm1)/(2*RVP.Yspace);
    case 3 % Backwards difference
            PSI_i_jm1 = PSImatrix(i,j-1);
            u =   (PSI_i_j - PSI_i_jm1)/RVP.Yspace;
    case 4 % Null zone
end

switch DSX
    case 1 % Forwards difference
            PSI_ip1_j = PSImatrix(i+1,j);
            v = - (PSI_ip1_j - PSI_i_j)/RVP.Xspace;
    case 2 % Central difference
            PSI_ip1_j = PSImatrix(i+1,j);
%             if i == 1 % Inlet
                PSI_im1_j = PSI_ip1_j;
%             else
                PSI_im1_j = PSImatrix(i-1,j);
%             end
            v = - (PSI_ip1_j - PSI_im1_j)/(2*RVP.Xspace);
    case 3 % backwards difference
            PSI_im1_j = PSImatrix(i-1,j);
            v = - (PSI_i_j - PSI_im1_j)/RVP.Xspace;
    case 4 % Null zone
end

end
```

## 6.2 Assignment 2 code

### 6.2.1 Q1B

```
%% CFD Assignment 2: Numerical methods Q 1B
clear, clc, close all


% Exact solution
x = 0:0.01:2;
ux = cos(pi*x);
dux = -sin(pi*x);

% Numerical methods
dx = 0.25;              % Grid settings
nmx = 0:dx:2;

for i = 1:length(nmx)
    % Points

    % Evaluation point
    ui(i)   = cos(pi*nmx(i));

    % i+1 point
    if i == length(nmx)
        uip1(i) = cos(pi*(nmx(i)+dx));
    else
        uip1(i) = cos(pi*nmx(i+1));
    end

    % i -1 point
    if i == 1
        uim1(i) = cos(pi*-nmx(i+1));
    else
        uim1(i) = cos(pi*nmx(i-1));
    end
```

```
    % 1st derivative calculation
    dudxF(i) =  (uip1(i) - ui(i))/dx;               % Forward
    dudxC(i) =  (uip1(i) - uim1(i))/(2*dx);         % Central
    dudxE(i) =  -pi*sin(pi*nmx(i));                 % Exact!
end

% Find dudx for each scheme/exact for u(0.25)
% Need interpolation for 0.1
if isreal(0.25/dx) && rem((0.25/dx),1)==0
    ind_025 = find(nmx == 0.25);
    dudxF025 = dudxF(ind_025);
    dudxC025 = dudxC(ind_025);
    dudxE025 = dudxE(ind_025);
elseif dx == 0.1
    ind03 = find(nmx < 0.31 & nmx > 0.29); % Thing cant find 0.3 = 0.3
    ind02 = find(nmx < 0.21 & nmx > 0.19);
    dudxF03  = dudxF(ind03);
    dudxF01  = dudxF(ind02);
    dudxF025 = (dudxF03 + dudxF01)/2;   % Interp will give same answer

    dudxC03  = dudxC(ind03);
    dudxC01  = dudxC(ind02);
    dudxC025 = (dudxC03 + dudxC01)/2;

    dudxE03  = dudxE(ind03);
    dudxE01  = dudxE(ind02);
    dudxE025 = (dudxE03 + dudxE01)/2;

    % Interpolation error corrected
    CdudxE025 = (-2.2214/dudxE025)*dudxE025;
    CdudxF025 = (-2.2214/dudxE025)*dudxF025;
    CdudxC025 = (-2.2214/dudxE025)*dudxC025;
else
    error('No code for base sizes resulting a remainder in 0.25/dx other than 0.1!')
end

%% Percentage difference of area


for i = 1:length(dudxE)-1
    aF(i) = trapz( [nmx(i) nmx(i+1)],[(dudxF(i)) (dudxF(i+1))] );
    aC(i) = trapz( [nmx(i) nmx(i+1)],[(dudxC(i)) (dudxC(i+1))] );
    aE(i) = trapz( [nmx(i) nmx(i+1)],[(dudxE(i)) (dudxE(i+1))] );
end

% Percentages
aFaE = aF./aE;
aCaE = aC./aE;
pdF = 1 - aFaE;
pdC = 1 - aCaE;
pdX = (1.5:1:(length(nmx)-0.5))./length(nmx);

figure(3); clf
set(gcf,'color','w');
area(dudxE,'FaceColor',[0.9 0
0],'FaceAlpha',.3,'EdgeAlpha',1,'EdgeColor','r','Linewidth',1.5); hold on
area(dudxF,'FaceColor',[1 0.5 0],'FaceAlpha',.3,'EdgeAlpha',1,'EdgeColor',[1 0.3
0],'Linewidth',1.5); hold on
area(dudxC,'FaceColor','g','FaceAlpha',.3,'EdgeAlpha',1,'EdgeColor',[0 0.8
0],'Linewidth',1.5); hold on

ax = gca;
ax.XAxisLocation = 'bottom';
xlabel('$$i = 1 \rightarrow N  ,  x = 0\rightarrow 2 \pi$$','Interpreter', 'Latex');
ylabel('$$({du\over dx})$$','Interpreter', 'Latex');
title('Gradients and area','FontSize',12)
set(gca,'FontName','Times New Roman','FontSize',12);grid minor
legend('Exact','Forward','Central')

% plot(repmat(4,1,10),(-1.5:(3/(length(repmat(4,1,10))-1)):1.5),'--k')
% plot(repmat(5,1,10),(-1.5:(3/(length(repmat(4,1,10))-1)):1.5),'--k')
% legend('Forward')
```

```
% xlim([3.5 5.5])
```

### 6.2.2 Q1C and Q1D

```
%%  Q1C and Q1D
clear, clc, close all
%% Controls
% 5 different types of stability settings:
% stable    - Will always dissapate.
% explosion - unstable (Recommend 3D for this)
% verge     - On verge of stability, function should never increase or decrease
% SO verge  - Slightly over verge, slightly increases in amplitude.
% MSO verge - More slightly over verge, faster representation of above.
% FTCS      - Forwards time, central space (Q1D)

CTRL.analysis = 'explosion';
CTRL.scheme   = 'upwind';       % upwind or FTCS (Q1D)
CTRL.BCs      = 'periodic';     % periodic or Neumann
CTRL.plot     = '3D';           % 3D, 2D, contour
%% Numerical parameters
xmin = 0;
xmax = 1;
N    = 51;          % Number of nodes
t    = [0 2];        % Time min and max
a    = -1;           % Speed of propegation wave

% Spatial discretization
dx = (xmax - xmin)/(N-1);
switch CTRL.BCs
    case 'periodic'
        x  = xmin: dx: xmax;
        k = 2;
    case 'neumann'
        % Inserting ghost nodes for boundary conditions Neuman
        x  = xmin - dx: dx: xmax + dx;
        k = 1;
end

%% Stability criterion
switch CTRL.analysis
    case 'explosion'
        dt = 1.2*dx/abs(a);
    case 'stable'
        dt = 0.5*dx/abs(a);
    case 'verge'
        dt = dx/abs(a);
    case 'SO verge'
        dt = 1.05*dx/abs(a);
    case 'MSO verge'
        dt = 1.1*dx/abs(a);
end

% Set initial conditions
u_init  = exp(-200*(x  - (xmax/2)).^2); % Gaussian pulse
u       = u_init;
unp1    = u_init;

figure(3)
plot(x,u_init,'s-','color',[0.8 0 0]);
axis([0 1 0 1])
set(gcf,'color','w')
grid minor
title('Two-dimensional Gaussian Pulse Initial')
xlabel('$$x$$','Interpreter', 'Latex')
ylabel('$$u(x)$$','Interpreter', 'Latex')
set(gca,'FontName','Times New Roman','FontSize',12);

% Time marching loop
```

```
nsteps = t(2)/dt;
ta = t(1);
for n = k:nsteps % was 1:nsteps, trying periodic)

    switch CTRL.BCs
        case 'neumann'
        % Calculate Neuman boundary conditions
        u(1) = u(3); % This ensures zero gradient accros boundayr
        u(N+3) = u(N+1);   % Flow just flows out through boundary (NEUMAN BOUNDARY)
        % Calcualte the upwind scheme
        for i = 2: N+2      % Not including boundary point
            unp1(i) = u(i) - (a*dt/dx)*((u(i+1) - u(i)));
        end
%-------------------------------------------------------------------------------------------
---------
        case 'periodic'
        % Periodic boundary conditions
         % Calcualte the upwind scheme
    switch CTRL.scheme
    case 'upwind'

            for j = 1: N-1   % Not including boundary point
                    unp1(n,j) = u(n-1,j) - (a*dt/dx)*((u(n-1,j+1) - u(n-1,j)));
                    uplot(n-1,j) = unp1(n,j);
            end
            % Boundary conditions
            unp1(n,N)   = u(n-1,1) - (a*(dt/dx))*(u(n-1,2) - u(n-1,1));
            tplot(1,n-1) = n*dt;
            uplot(n-1,N) = unp1(n,N);

     case 'FTCS'
            for j = 2: N-1
            unp1(n,j) = u(n-1,j) - ((a*dt/dx)*((u(n-1,j+1) - u(n-1,j-1))/2));
            uplot(n-1,j) = unp1(n,j);
            end
            % Boundary condition
            unp1(n,N) = u(n-1,1) - ((a*dt/dx)*((u(n-1,2) - u(n-1,N))/2));
            tplot(1,n-1) = n*dt;
            uplot(n-1,N) = unp1(n,N);
     end

            switch CTRL.plot
                case '3D'
                    if n >3
                        figure(4); clf
                        [X,Y] = meshgrid(x,tplot);
                        surf(X,Y,uplot)
                        shading interp
                        colormap(esa)
                        xlabel('x');ylabel('t');zlabel('u');
                        pause(dt)
                    end
                case 'contour'
                    if n>3
                        figure(4);clf
                        [X,Y] = meshgrid(x,tplot);
                        contourf(X,Y,uplot)
                        colormap(esa)
                        title(sprintf(' t = %0.4f', ta))
                        xlabel('x');ylabel('t');zlabel('u');
                        pause(dt)
                    end
            end

    end

    % Update time
    ta = ta + dt;
    u = unp1;

    % Plot solution
    switch CTRL.BCs
```

```
        case 'periodic'
        switch CTRL.plot
            case '2D'
            figure(1)
            plot(x,u(n,:),'rs-');
            axis([0 1 0 1])
            title(sprintf(' t = %0.4f', ta))
            pause(dt);
            set(gcf,'color','w')
            grid minor
            xlabel('$$x$$','Interpreter', 'Latex')
            ylabel('$$u(x)$$','Interpreter', 'Latex')
            set(gca,'FontName','Times New Roman','FontSize',12);

        end
        case 'neumann'
        figure(1)
        plot(x,u,'rs-');
        axis([0 1 0 1])
        title(sprintf(' t = %0.4f', ta))
        pause(dt);
    end
end

%% Plotting

switch CTRL.plot
    case '3D'
figure(4)
set(gcf,'color','w');
colorbar
xlabel('$$x$$','Interpreter', 'Latex')
ylabel('$$t$$','Interpreter', 'Latex')
zlabel('$$u(x)$$','Interpreter', 'Latex')
        switch CTRL.analysis
            case 'stable'
                 view([20 60])
            case 'explosion'
                 view([20 60])
            case 'SO verge'
                 view([20 60])
            case 'verge'
                 view([20 60])
            case 'MSO verge'
                 view([33 25])
        end
    grid minor
    annotation(figure(4),'textarrow',[0.71892856476399 0.671785714285712],...
    [0.705190480405892 0.325238095238101],'String',{'Periodic BC'},...
    'FontSize',12,...
    'FontName','Times New Roman');
    annotation(figure(4),'arrow',[0.26035714285714 0.130357142857143],...
    [0.101380952380956 0.146190476190476]);
    annotation(figure(4),'textbox',...
    [0.172785714285714 0.0795238095238096 0.0461428571428572 0.0495238095238096],...
    'String','a',...
    'LineStyle','none',...
    'Interpreter','latex',...
    'FontWeight','bold',...
    'FitBoxToText','off');
    case 'contour'
        figure(4)
        set(gcf,'color','w');
        colorbar
        xlabel('$$x$$','Interpreter', 'Latex')
        ylabel('$$t (seconds)$$','Interpreter', 'Latex')
        annotation(figure(4),'textarrow',[0.800550206327373 0.785419532324622],...
        [0.955989247311828 0.526881720430108],'String',{'Periodic BC'},...
        'FontSize',12,...
        'FontName','Times New Roman');
        annotation(figure(4),'textbox',...
        [0.227960110041265 0.0268817204301075 0.0719023383768913 0.0250896057347669],...
```

```
        'String',{'a'},...
        'LineStyle','none',...
        'Interpreter','latex',...
        'FitBoxToText','off');
        annotation(figure(4),'arrow',[0.326071428571429 0.148928571428571],...
        [0.0461428571428571 0.0452380952380953]);
end
set(gca,'FontName','Times New Roman','FontSize',12);
```

### 6.2.3 Q2B

```
%% Assignment 2 Q2B
clear, clc, close all

% Establish symbolic variables
syms Q rho rhou E p u Et gamma u1 u2 u3 e

p = (gamma - 1)*(u3 - 0.5*((u2^2)/u1));
e = p/((gamma-1)*u1);
Et = u1*(e + 0.5*(u2/u1)^2);

Q = [u1 u2 u3].';    E = [u2;
                          (u1*(u2/u1)^2)+p;
                          (Et + p)*(u2/u1)];

% Matrix differenciation to form Jacobs mat
for i = 1:length(Q')
    for j = 1:length(E')
        A(j,i) = diff(E(j),Q(i));
    end
end
A = simplify(A);

[V,D] = eig(A);

LAM = diag(D);
EV1 = LAM(1); % u
EV2 = LAM(2); % u+a
EV3 = LAM(3); % u-a


%% Checks for eigenvalues and vectors with respect to notes!
% Instead of working through absolutely massive equations, subbing in value
% to determine if eigen values are infact, u, u-a, u+a etc.
gammaV = 1.4;
u1V = 1;
u2V = 1;    % Normalized conservative variables
u3V = 1;
pV = double(vpa(subs(p,[gamma u1 u2 u3], [gammaV u1V u2V u3V]),20));        % Pressure
aV = (gammaV*pV/u1V)^(1/2);                                                 %
Speed of sound

% Theoretical answers of eigenvalues should be:
% u     = u2/u1          = 1
% u + a = u2/u1 + 0.5292  = 1.5292;
% u - a = u2/u1 - 0.5292  = 1 - 0.5292;
u   = u2V/u1V;
upa = u + aV;
uma = u - aV;

EV1_V = double(vpa(subs(EV1,[u1 u2],[u1V u2V]),20));
EV2_V = double(vpa(subs(EV2,[gamma u2 u1 u3],[gammaV u2V u1V u3V]),20));
EV3_V = double(vpa(subs(EV3,[gamma u2 u1 u3],[gammaV u2V u1V u3V]),20));
if EV1_V == u
    fprintf('\n lam 1 = u')
end
```

```
if EV2_V == upa
    fprintf('\n lam 2 = u+a')
end
if EV3_V > uma - 0.01*uma && EV3_V < uma + 0.01*uma
    fprintf('\n lam 3 = u - a')
end
```

### 6.2.4 Assignment 2 plots

```
clear, clc, close all

%% Plotting random things for A2

% dx = 0.25
pdC025 = load('C_Ydx025.mat');
pdF025 = load('F_Ydx025.mat');
X025   = load('Xdx025.mat');
% dx = 0.1;
pdC01 = load('C_Ydx01.mat');
pdF01 = load('F_Ydx01.mat');
X01   = load('Xdx01.mat');
% dx = 0.01
pdC001 = load('C_Ydx001.mat');
pdF001 = load('F_Ydx001.mat');
X001   = load('Xdx001.mat');
% dx = 0.001
pdC0001 = load('C_Ydx0001.mat');
pdF0001 = load('F_Ydx0001.mat');
X0001   = load('Xdx0001.mat');

% Plot
figure; set(gcf,'color','w')
plot (X025.pdX,pdF025.pdF,'o-','Color',[0 0.7 0],'LineWidth',1); hold on;
plot  (X01.pdX,pdF01. pdF,'d-','Color',[0.8 0.4 0],'LineWidth',1); hold on;
plot (X001.pdX,pdF001.pdF,'s-','Color',[0 0.1 0.5],'LineWidth',1); hold on;
plot(X0001.pdX,pdF0001.pdF,'--','Color',[0.9 0.1 0.1],'LineWidth',1)
ax = gca;
ax.XAxisLocation = 'bottom';
ax.YAxisLocation = 'origin';
xlabel('$$x_{i}/ x_{N}$$','Interpreter', 'Latex')
ylabel('$$A_{Exact} - A_{Scheme}\over A_{Exact}$$','Interpreter', 'Latex')
ax.FontSize = 12;
set(gca,'FontName','Times New Roman')
legend('dx = 0.25','dx = 0.1','dx = 0.01','dx = 0.001')
title('Forward difference gradient area analysis','FontSize',14)
grid minor

% Central
figure; set(gcf,'color','w')
plot (X025.pdX,pdC025.pdC,'o-','Color',[0 0.7 0],'LineWidth',1); hold on;
plot  (X01.pdX,pdC01. pdC,'d-','Color',[0.8 0.4 0],'LineWidth',1); hold on;
plot (X001.pdX,pdC001.pdC,'s-','Color',[0 0.1 0.5],'LineWidth',1); hold on;
plot(X0001.pdX,pdC0001.pdC,'--','Color',[0.9 0.1 0.1],'LineWidth',1)
ax = gca;
ax.XAxisLocation = 'bottom';
ax.YAxisLocation = 'origin';
xlabel('$$x_{i}/ x_{N}$$','Interpreter', 'Latex')
ylabel('$$A_{Exact} - A_{Scheme}\over A_{Exact}$$','Interpreter', 'Latex')
ax.FontSize = 12;
set(gca,'FontName','Times New Roman')
legend('dx = 0.25','dx = 0.1','dx = 0.01','dx = 0.001')
title('Central difference gradient area analysis','FontSize',14)
grid minor
```

## 6.3 Assignment 3 and 4 code

### 6.3.1 Main code (Jameson)

```
%% RK method used for compressible flow over aerofoil.
% JAMESON: Main function


% NOTE: - V is a structured array which contains ALL variables which are
%         required to be in EVERY function; not to be confused with
%         vertical velocity.
%       - Vv is the vertical velocity.

%% Input function

clear, clc, close all
%% Controls

CTRL.gridplot    = 'no';            % Note, grid is plot for every edge because I
thought it looked super cool seeing it develop
CTRL.debugFIT    = 'no';            % Debug first iteration, plots
CTRL.debugEIT    = 'no';            % Debug last iteration, plots grid and colored lines
on cells unequivalent to your results
CTRL.euler       = 'no';            % Euler method
CTRL.convhist    = 'yes';            % Display CL while iterating
CTRL.aerofoil    = 'naca0012' ;     % 'naca0012' 'NLRBoeing' 'SKF'
CTRL.panelcomp   = 'no';            % Compare with panel method (auto switches FMACH to
0.1)
CTRL.CLcompare   = 'yes';            % Only good for naca0012

%% Input function
V.input = 1;
[V] = INIT(V,CTRL);

switch CTRL.gridplot
    case 'yes'
    [V] = gridplot(V);
    case 'no'
end

% Debugging using first iteration
switch CTRL.debugFIT
    case 'yes'
    V.NCYC = 1; % First iteration
    CDat = xlsread('output1iteration.xlsx');
    case 'no'
end

%% Debug end iteration and comparison data
EC_dat = xlsread('outputENDIT.xlsx');
Cl_EI = EC_dat(1:2000,2);
Cd_EI = EC_dat(1:2000,3);
Cm_EI = EC_dat(1:2000,4);

%% Start of program
for N = 1:1:V.NCYC   % Iterating through time steps

    % Call BCON (Boundary condition subroutine) (Apply boundary conditions)
    % Call TIME (time setting subroutine)       (Calculate time steps)
    [V] = BCON(V);
    switch CTRL.euler
        case 'yes'
            if N ==1
                [V] = TIME(V);
            end
```

```
        case 'no'
            [V] = TIME(V);
    end

    for NS = 1:V.NRKS  % Iterating through orders of RK (START RK STEPS)

        % Could combine BCON with flux for speed.
        if NS ~= 1
        [V] = BCON(V);
        elseif  NS == 1
        [V] = DISS(V);      % Add dissipation and only calculate it once for ALL RK
steps
%         V.D = zeros(V.NC,4);
        end
        [V] = FLUX(V);      % Calculate flux
        COEFF = V.ST(NS);   % Set RK coefficient in correspondence to loop

        for i = 1:V.NC  % Updating solution, calculating pressure etc
            COEFF2   = COEFF*(V.DT(i));
            V.W(i,1) = abs(V.WS(i,1) - COEFF2*(V.Q(i,1) - V.D(i,1))); % abs cause you
cannot get -ve density and energy
            V.W(i,2) =     V.WS(i,2) - COEFF2*(V.Q(i,2) - V.D(i,2));
            V.W(i,3) =     V.WS(i,3) - COEFF2*(V.Q(i,3) - V.D(i,3));
            V.W(i,4) = abs(V.WS(i,4) - COEFF2*(V.Q(i,4) - V.D(i,4)));
            V.U      = V.W(i,2)/V.W(i,1);
            V.Vv     = V.W(i,3)/V.W(i,1);
            V.P(i)   = V.GMm1*(V.W(i,4)- (0.5*V.W(i,1))*(V.U*V.U + V.Vv*V.Vv));
        end
    end  % End of RK steps

    % Storing old solution
    for j = 1:V.NC
            V.WS(j,1) = V.W(j,1);
            V.WS(j,2) = V.W(j,2);
            V.WS(j,3) = V.W(j,3);
            V.WS(j,4) = V.W(j,4);
    end
    CX = 0;
    CY = 0;
    CM = 0;
    %% Post/mid proccessing (?)
    % Calculating integrarted loads from simuation
    for i = 1:(V.ICU - V.ICL)
       K     = V.JLIST(i,2);      % thought this was i,3 for nc1
       CP(i) = (V.P(K)- 1)/V.CPD;
       IA    = V.JLIST(i,3);
       IB    = V.JLIST(i,4);
       XCM   = 0.5*(V.XP(IB) + V.XP(IA))  - 0.25;
       YCM   = 0.5*(V.YP(IB) + V.YP(IA));
       DCX   =  CP(i)*(V.YP(IB) - V.YP(IA));
       DCY   = -CP(i)*(V.XP(IB) - V.XP(IA));
       CM    = CM - (DCY*XCM) + (DCX*YCM);
       CX = CX + DCX;
       CY = CY + DCY;
       XF(i) = XCM + 0.25;
    end
    CL(N)  = (CY*V.CA) - (CX*V.SA);
    CD(N)  = (CY*V.SA) + (CX*V.CA);
    formatspec = 'Iteration %f, Cl = %0.5f, Cd = %0.5f, Cm = %0.5f\n';
    fprintf(formatspec,N,CL(N),CD(N),CM)

    switch CTRL.convhist
        case 'yes'
        if N > 1
        plot(N-1:N,CL(N-1:N),'-','color',[0.8 0 0],'LineWidth',2) ; hold on
        switch CTRL.CLcompare
            case 'yes'
            plot(N-1:N,Cl_EI(N-1:N),'s','color',[0 0 0.8],'LineWidth',0.5);  hold on
            case 'no'
        end
        pause(0.001)
        end
```

```
        case 'no'
    end

end   % End of time steps

figure(1), clf; set(gcf,'color','w')
plot(CL,'-','color',[0.8 0 0],'LineWidth',2); hold on
switch CTRL.CLcompare
    case 'yes'
    ClCx = 1:50:2000;
    plot(ClCx,Cl_EI(ClCx),'s','color',[0 0 0.8])
    case 'no'
end
xlabel('Iterations'); ylabel('CL'); title('Convergence history of CL')
set(gca,'FontName','Times New Roman','FontSize',12);
grid minor

%% Plotting Cp surface and compare
figure
set(gcf,'color','w')
plot(XF,-CP,'-','color',[0.8 0 0],'Linewidth',2); hold on
xlabel('$$x/C$$','Interpreter','Latex'); ylabel('-CP')
title('Cp distribution for SKF aerofoil')
switch CTRL.CLcompare
    case 'yes'
    XF_EIT = EC_dat(2004:2103,1);
    CP_EIT = EC_dat(2004:2103,2);
    plot(XF_EIT,-CP_EIT,'s','color',[0 0 0.8])
    case 'no'
end
legend('The code','Dr Barakos')
set(gca,'FontName','Times New Roman','FontSize',12) ;grid minor

% DEBUG: Comparing 1st iteration data with first iteration of timestep.
switch CTRL.debugFIT
    case 'yes'
        [DB]=debugCFDA3(V,CDat);
    case 'no'
end

% Debug: comparing end iteration data with Dr Barakos's code
% Difference in data
switch CTRL.debugEIT
    case 'yes'
        [DB] = debugCFDA3ENDIT(V,EC_dat);
    case 'no'
end

%% Plotting Velocity!
% Every cell is K at one point so using K

Kstore  = zeros(V.NEDGE,1);
Kmat    = zeros(V.NC,1);
IAstore = zeros(V.NC,3);
IBstore = zeros(V.NC,3);
IAtempstore = zeros(1,3);
IBtempstore = zeros(1,3);
ind = 0;
for i = 1:V.NEDGE
    K = V.JLIST(i,2);
    if K ~= Kstore                              % If K is not equal to any previous
values of K
            ind = ind + 1;                      % Indice if statement calls, should
result in 5600 for K number of cells
            Kmat(ind,1) = K;                    % Store this value of K
            Kind = find(K == V.JLIST(:,2));     % Find all connectivity rows with K
value
            for j = 1:length(Kind)              % For the number of rows...
                IA   = V.JLIST(Kind(j),3);      % Get corresponding IA value
                IB   = V.JLIST(Kind(j),4);      % Get corresponding IB value
                IAtempstore(1,j) = IA;          % Store value of IA
                IBtempstore(1,j) = IB;          % Store value of IB
```

```
            end

            IAstore(ind,:) = IAtempstore;
            IBstore(ind,:) = IBtempstore;
            clear IAtempstore IBtempstore
            IAtempstore      = zeros(1,3);
            IBtempstore      = zeros(1,3);
    end

    Kstore(i,1) = K;  % Store untreated K index for referencing K values

end

% NOTE ON DATA OF FMAT
% Col 1,   2,    3,    4,   5,   6,    7
%     K    IA1   IA2   IA3  IB1  IB2   IB3

% Where IAi, IBi are from the same connectivity row and 0 indicates no value
%Ordering 1 - V.NC
[Korder,Ind_order] = sort(Kmat);
for i = 1:length(Ind_order)
    IAstore_order(i,:) = [IAstore(Ind_order(i),1) , IAstore(Ind_order(i),2) ,
IAstore(Ind_order(i),3)];
    IBstore_order(i,:) = [IBstore(Ind_order(i),1) , IBstore(Ind_order(i),2) ,
IBstore(Ind_order(i),3)];
end

% ORDERED CELL DATA FOR K

switch CTRL.aerofoil
    case 'naca0012'
    FMAT = [Korder, IAstore_order, IBstore_order];
    plop = 0;
    case 'NLRBoeing'
    FMAT1= [Korder, IAstore_order, IBstore_order];
    FMAT = FMAT1(83:end,:);
    plop = 82;   % Some reason data is giving zeros for K at start of FMAT matrix,
likely from the zeros init
    case 'SKF'
    FMAT1= [Korder, IAstore_order, IBstore_order];
    FMAT = FMAT1(65:end,:);
    plop = 64;   % Some reason data is giving zeros for K at start of FMAT matrix,
likely from the zeros init
end



for i = 1:V.NC-plop
    % Define cell index and geometrics
    K   = FMAT(i,1);    IA1 = FMAT(i,2);    IB1 = FMAT(i,5);
                        IA2 = FMAT(i,3);    IB2 = FMAT(i,6);
                        IA3 = FMAT(i,4);    IB3 = FMAT(i,7);

    % Get conservative variables related to K
    Pr(i,1) = V.P(K);
    rho(i,1)= V.W(K,1);
    U(i,1)  = V.W(i,2)/V.W(i,1);       % Hor vel
    Vv(i,1) = V.W(i,3)/V.W(i,1);       % Ver vel
    C(i,1) = (abs(V.GM*Pr(i,1)/rho(i,1)))^(1/2);

    % If there is 1,2 or 3 values of IA,IB for a K value. Logical is 1 for
    % actual values and 0 for 0 values.
    if sum(logical([IA1 IA2 IA3])) == 1
        %Only contains IA1, IB1
        XP_IA1 = V.XP(IA1);    XP_IB1 = V.XP(IB1);
        YP_IA1 = V.YP(IA1);    YP_IB1 = V.YP(IB1);
        Cen(i,1) = (XP_IA1 + XP_IB1)/2;
        Cen(i,2) = (YP_IA1 + YP_IB1)/2;
    elseif sum(logical([IA1 IA2 IA3])) == 2
        % Contains IA1 IA2, IB1 IB2 (two cell interactions)
        XP_IA1 = V.XP(IA1);    XP_IB1 = V.XP(IB1);
        YP_IA1 = V.YP(IA1);    YP_IB1 = V.YP(IB1);
```

```
        XP_IA2 = V.XP(IA2);    XP_IB2 = V.XP(IB2);
        YP_IA2 = V.YP(IA2);    YP_IB2 = V.YP(IB2);
        Cenx1 = (XP_IA1 + XP_IB1)/2;
        Ceny1 = (YP_IA1 + YP_IB1)/2;
        Cenx2 = (XP_IA2 + XP_IB2)/2;
        Ceny2 = (YP_IA2 + YP_IB2)/2;
        Cen(i,1) = (Cenx1+Cenx2)/2;
        Cen(i,2) = (Ceny1+Ceny2)/2;
    elseif sum(logical([IA1 IA2 IA3])) == 3
        % Has all three cell interactions, IA1, IA2, IA3...
        XP_IA1 = V.XP(IA1);    XP_IB1 = V.XP(IB1);
        YP_IA1 = V.YP(IA1);    YP_IB1 = V.YP(IB1);
        XP_IA2 = V.XP(IA2);    XP_IB2 = V.XP(IB2);
        YP_IA2 = V.YP(IA2);    YP_IB2 = V.YP(IB2);
        XP_IA3 = V.XP(IA3);    XP_IB3 = V.XP(IB3);
        YP_IA3 = V.YP(IA3);    YP_IB3 = V.YP(IB3);
        Cenx1 = (XP_IA1 + XP_IB1)/2;
        Ceny1 = (YP_IA1 + YP_IB1)/2;
        Cenx2 = (XP_IA2 + XP_IB2)/2;
        Ceny2 = (YP_IA2 + YP_IB2)/2;
        Cenx3 = (XP_IA3 + XP_IB3)/2;
        Ceny3 = (YP_IA3 + YP_IB3)/2;
        Cen(i,1) = (Cenx1+Cenx2+Cenx3)/3;
        Cen(i,2) = (Ceny1+Ceny2+Ceny3)/3;
    end

end

% Plot velocity vectors
figure(10), clf
set(gcf,'color','w')
quiver(Cen(:,1),Cen(:,2),U(:,1),Vv(:,1),0.07,'color',[0.7 0 0]) % Quiver can use vectors
and doesnt need a matrice!
xlim([-0.3 1.25]); ylim([-0.5 0.5])
xlabel('$$x/c$$','Interpreter', 'Latex');
ylabel('$$y/c$$','Interpreter', 'Latex');
title('Velocity vector plot')
set(gca,'FontName','Times New Roman','FontSize',12);


% Load panel method data
switch CTRL.panelcomp
    case 'yes'
inputpanel = xlsread('CPpanel.xlsx');
CpV = inputpanel(:,2);
CpI = inputpanel(:,4);
xI  = inputpanel(:,3);
xV  = inputpanel(:,1);

% Panel comparison
figure(3); clf
set(gcf,'color','w')
plot(XF,-CP,'-','color',[0.8 0 0],'Linewidth',1); hold on
plot(xI,-CpI,'-.','color',[0 0 0.8],'Linewidth',1); hold on
plot(xV,-CpV,'-','color',[0 0.8 0],'Linewidth',1);

legend('The code','Panel - Inviscid','Panel - Viscid')
xlabel('$$x/C$$','Interpreter','Latex'); ylabel('-CP')
title('Cp distribution comparison at M = 0.1')
set(gca,'FontName','Times New Roman','FontSize',12) ;grid minor

    case 'no'
end
```

### 6.3.2 Input function

```
%% INPUT FUNCTION FOR RK
function [V] = INIT(V,CTRL)
% Reads data
% Initilizes variables
% Structured matrix V for globilization of parameters instead of "global"


% Need to read in free stream mach (FMACH), AL (Incidence),CFL
% ,NCYC (number of iterations/cycles).

switch CTRL.aerofoil
    case 'naca0012'
    inputdata = xlsread('grid.xlsx');
    case 'NLRBoeing'
    inputdata = xlsread('NLRBoeing.xlsx');
    case 'SKF'
    inputdata = xlsread('skf.xlsx');
end

% Input settings
V.FMACH = inputdata(1,1);
V.AL    = inputdata(1,2);
V.CFL   = inputdata(1,3);
V.NCYC  = inputdata(1,4);

% For panel methods.
switch CTRL.panelcomp
    case 'yes'
    V.FMACH = 0.1;
    case 'no'
end


% Grid data
V.ICL   = inputdata(2,1);   % Aerofoil inner boundary (1st point)   |
V.ICU   = inputdata(2,2);   % Aerofoil inner boundary (Last point)  | - > Points around
aerofoil
V.NBD   = inputdata(2,3);  % Outer boundary around aerofoil (Number of points)
V.NPTI  = inputdata(2,4);  % Total number of grid points
V.NC    = inputdata(2,5);  % Total number of cells
V.NEDGE = inputdata(2,6);  % Total number of edges

% Runga-kutta parameters
V.NRKS    = 4;                              % Order of rungakutta method

% % Euler method
switch CTRL.euler
    case 'yes'
    V.NRKS = 1;
    V.NCYC = 200000;
    V.CFL  = 0.15;
    case 'no'
end

V.ST(1,1) = 0.25*(abs(V.CFL));                % First order coeff etc et.
V.ST(1,2) = (0.33333)*(abs(V.CFL));
V.ST(1,3) = 0.5*(abs(V.CFL));
V.ST(1,4) = 1.0*(abs(V.CFL));
V.RK2     = 0.5;                              % 1st dissapation parameter, k2
V.RK4     = 0.008;                            % 2nd dissapation parameter, k4
% V.RK2     = 1;
% V.RK4     = 1/32;

% Aerodynamics
V.GM = 1.4; %Gamma of air
V.GMm1 = V.GM - 1;     % Gamma - 1, this is precalculated as it needs to be calculated
alot
V.AL = V.AL*pi/180;    % Converting incidence to radians
V.SA = sin(V.AL);      % Precalculating sin's
V.CA = cos(V.AL);      % Precalculating cos's
V.Rho0 = 1;            % Free stream density
V.P0   = 1;            % Free stream pressure
```

```
V.C0   = (V.GM*V.P0/V.Rho0)^(1/2);  % Freestream speed of sound
V.U0   = V.FMACH*V.C0*V.CA;         % Freesteam horizontal velocity scaled with mach
V.V0   = V.FMACH*V.C0*V.SA;         % Freestream vertical velocity scaled with mach
V.E0   = ((V.P0)/(V.Rho0*V.GMm1)) + (0.5*((V.U0*V.U0)+(V.V0*V.V0)));  % Freestream
energy
V.CPD  = V.GM*V.FMACH*V.FMACH*0.5;  % Freestream dynamic head

% Establishing connectivity (JLIST)
for i = 1:V.NEDGE           % Edge defines entire connectivity index total
    % Note, data starts at row 3
        V.JLIST(i,1) = inputdata(2+i,2);    % Current
        V.JLIST(i,2) = inputdata(2+i,3);    % JLIST(:,2) = K
        V.JLIST(i,3) = inputdata(2+i,4);    % JLIST(:,3) = IA
        V.JLIST(i,4) = inputdata(2+i,5);    % JLIST(:,4) = IB
        V.JLIST(i,5) = inputdata(2+i,6);    % JLIST(:,5) = IP
end

% Establishing XP and YP
for i = 1:V.NPTI
    V.XP(1,i) = inputdata((V.NEDGE+2)+i,2);
    V.YP(1,i) = inputdata((V.NEDGE+2)+i,3);
end
% Establishing solutions from initial conditions for EACH CELL
for i = 1:V.NC
    V.W(i,1) = V.Rho0;        % Setting density for each cell
    V.W(i,2) = V.Rho0*V.U0;   % Setting horizontal velocity
    V.W(i,3) = V.Rho0*V.V0;   % Setting vertical velocity
    V.W(i,4) = V.Rho0*V.E0;   % Setting energy for each cell
    V.P(i)   = V.P0;          % Setting pressure for each cell

    % Initilizing store of variables as initial solution
    % This will be overwritten during simulation
    V.WS(i,1)= V.W(i,1);
    V.WS(i,2)= V.W(i,2);
    V.WS(i,3)= V.W(i,3);
    V.WS(i,4)= V.W(i,4);
end
% Initilizattion of boundary values for each variable (Free stream
% conditions
for i = 1:V.NBD
    V.BW(i,1) = V.Rho0;
    V.BW(i,2) = V.Rho0*V.U0;
    V.BW(i,3) = V.Rho0*V.V0;
    V.BW(i,4) = V.Rho0*V.E0;
    V.BW(i,5) = V.P0;
end

clear inputdata

end
```

### 6.3.3 Flux function

```
%% FLUX
function [V] = FLUX(V)
% Calculates flux between each cell

% Debugging
% clear
% clc
% V.input = 1;
% [V] = INIT(V);

V.Q  = zeros(V.NC,4); % RESET FLUXES EVERY ITERATION
%% Calculating flux contribution of aerofoil surface (Pressure only)
for i = 1:(V.ICU - V.ICL)  % (Aerofoil upper point 101 - aerofoil lower point 1 = 100 =>
1:100)
```

```
    K  = V.JLIST(i,2);
    IA = V.JLIST(i,3);
    IB = V.JLIST(i,4);
    DX = V.XP(IB) - V.XP(IA);
    DY = V.YP(IB) - V.YP(IA);
    V.Q(K,2) =   DY* V.P(K);  % Note only pressure
    V.Q(K,3) = - DX* V.P(K);
end

%% Calculate fluxes of OUTER BOUNDARY
% Uses BW as these are the BOUNDARY CONDITIONS (e.g. free stream)
for i = ((V.ICU - V.ICL)+1):V.NBD
     K = V.JLIST(i,2);
    IA = V.JLIST(i,3);
    IB = V.JLIST(i,4);
     U = V.BW(i,2)/V.BW(i,1);       % Hor vel
    Vv = V.BW(i,3)/V.BW(i,1);       % Ver vel
    DX = V.XP(IB) - V.XP(IA);       % deltax dist?
    DY = V.YP(IB) - V.YP(IA);       % deltay dist?
    QK = DY*U - DX*Vv;              % Flux going out of cell? What is this
    DP = V.BW(i,5);                  % Pressure of cell

    % Updating fluxes for cells
    F1 = QK*V.BW(i,1);                % Flux(?) * density
    V.Q(K,1) = V.Q(K,1) + F1;

    F2 = (QK*V.BW(i,2)) + (DP*DY);
    V.Q(K,2) = V.Q(K,2) + F2;

    F3 = (QK*V.BW(i,3)) - (DP*DX);
    V.Q(K,3) = V.Q(K,3) + F3;

    F4 = QK*(V.BW(i,4) + DP);
    V.Q(K,4) = V.Q(K,4) + F4;
end

%% Calculate fluxes INSIDE DOMAIN
for i = (V.NBD + 1): V.NEDGE
      K = V.JLIST(i,2);
     IA = V.JLIST(i,3);
     IB = V.JLIST(i,4);
     IP = V.JLIST(i,5);
      U = 0.5*((V.W(K,2)/V.W(K,1)) + (V.W(IP,2)/V.W(IP,1)));
     Vv = 0.5*((V.W(K,3)/V.W(K,1)) + (V.W(IP,3)/V.W(IP,1)));
     DX = V.XP(IB) - V.XP(IA);
     DY = V.YP(IB) - V.YP(IA);
     QK = (DY*U) - (DX*Vv);
     DP = V.P(K) + V.P(IP);

     % Updating fluxes
     F1 =  0.5*QK*(V.W(K,1) + V.W(IP,1));
     V.Q(K,1) = V.Q(K,1)  + F1;      % |
     V.Q(IP,1)= V.Q(IP,1) - F1;      % |> Flux's must balance.

     F2 = 0.5*(QK*(V.W(K,2) + V.W(IP,2)) + DP*DY);
     V.Q(K,2) = V.Q(K,2)  + F2;
     V.Q(IP,2)= V.Q(IP,2) - F2;

     F3 = 0.5*(QK*(V.W(K,3) + V.W(IP,3)) - DP*DX);
     V.Q(K,3) = V.Q(K,3)  + F3;
     V.Q(IP,3)= V.Q(IP,3) - F3;

     F4 = 0.5*QK*(V.W(K,4) + V.W(IP,4) + DP);
     V.Q(K,4) = V.Q(K,4)  + F4;
     V.Q(IP,4)= V.Q(IP,4) - F4;
end


end   % End of function 'end'
```

### 6.3.4 Dissipation function

```
%% DISS function
% Adds dissapation to each cell in order for the RK method to work.
function [V] = DISS(V)

% clear
% clc
% V.input = 1;
% [V] = INIT(V);

% RK2 and RK4 = empirical constants where K4 depends on machine precision
RK2 = V.RK2;
RK4 = V.RK4;

% Initilization of variables ( Could use repmat or zeros)
% Preallocating memory
DW2 = zeros(V.NC,4);
V.D = zeros(V.NC,4);    % NEED TO RESSET DISSAPATION

% Not considering around aerofoil as it is a BC.
for i = V.NBD + 1: V.NEDGE
   K  = V.JLIST(i,2);
   IP = V.JLIST(i,5);
   % Calculating differences in variables
   DELW1 = (V.W(IP,1) - V.W(K,1));
   DELW2 = (V.W(IP,2) - V.W(K,2));
   DELW3 = (V.W(IP,3) - V.W(K,3));
   DELW4 = (V.W(IP,4) - V.W(K,4) + V.P(IP) - V.P(K));

   % For every edge, this adds the difference
   DW2(K,1)  = DW2(K,1) + DELW1;
   DW2(K,2)  = DW2(K,2) + DELW2;
   DW2(K,3)  = DW2(K,3) + DELW3;
   DW2(K,4)  = DW2(K,4) + DELW4;
   DW2(IP,1) = DW2(IP,1) - DELW1;
   DW2(IP,2) = DW2(IP,2) - DELW2;
   DW2(IP,3) = DW2(IP,3) - DELW3;
   DW2(IP,4) = DW2(IP,4) - DELW4;
end
% Summary of loop: was to calculate (1) in notebook. Calculates differences
% in variables

% This is where dissapation is calculated
for i = V.NBD + 1: V.NEDGE
   K  = V.JLIST(i,2);
   IA = V.JLIST(i,3);
   IB = V.JLIST(i,4);
   IP = V.JLIST(i,5);
   C2 = V.GM*(V.P(K) + V.P(IP))/(V.W(K,1) + V.W(IP,1));
   U  = ((V.W(K,2)/V.W(K,1)) + (V.W(IP,2)/V.W(IP,1)))*0.5;      % Average velocity
between each cell
   Vv = ((V.W(K,3)/V.W(K,1)) + (V.W(IP,3)/V.W(IP,1)))*0.5;
   DX = V.XP(IB) - V.XP(IA);                                    % Calculating projection
of edge in X
   DY = V.YP(IB) - V.YP(IA);
   DL2   = DX*DX + DY*DY;                                       % For eig calcululation
component
   A     = abs(DY*U - DX*Vv)  + (C2*DL2)^(1/2);                 % Estimate of the
biggest eigenvalue (see notes)
   EPS2  = RK2*abs((V.P(IP) - V.P(K))/(V.P(IP) + V.P(K)));      % See notes with
epsilon, for order of mag 2
   EPS4  = max(0,(RK4 - EPS2));                                 % DMAX1 is just max.
   DELW1 = (V.W(IP,1) - V.W(K,1));
   DELW2 = (V.W(IP,2) - V.W(K,2));
   DELW3 = (V.W(IP,3) - V.W(K,3));
   DELW4 = (V.W(IP,4) - V.W(K,4) + V.P(IP) - V.P(K));
```

```
    T1 = EPS2*DELW1;
    T2 = EPS2*DELW2;
    T3 = EPS2*DELW3;
    T4 = EPS2*DELW4;
    S1 = EPS4*(DW2(IP,1) - DW2(K,1));
    S2 = EPS4*(DW2(IP,2) - DW2(K,2));
    S3 = EPS4*(DW2(IP,3) - DW2(K,3));
    S4 = EPS4*(DW2(IP,4) - DW2(K,4));
    T1 = A*(T1 - S1);
    T2 = A*(T2 - S2);
    T3 = A*(T3 - S3);
    T4 = A*(T4 - S4);

    V.D(K,1)  = V.D(K,1) + T1;
    V.D(K,2)  = V.D(K,2) + T2;
    V.D(K,3)  = V.D(K,3) + T3;
    V.D(K,4)  = V.D(K,4) + T4;
    V.D(IP,1) = V.D(IP,1) - T1;
    V.D(IP,2) = V.D(IP,2) - T2;
    V.D(IP,3) = V.D(IP,3) - T3;
    V.D(IP,4) = V.D(IP,4) - T4;

%     %Debug
%     V.EPS4store(i,1) = EPS4;
end

end
```

### 6.3.5 Time function

```
%% TIME
% Calculates local time step of EACH CELL
function [V] = TIME(V)

% Notes:
% - Can make approximation as it DOES NOR EFFECT SOLUTION but rather the
%   performance of the code.

% - SQRT is 20-30 multiplications

% - There is alot of approximations with averages etc. But with increase
%   grid refinement, this decreases.

% % Debugging
% clear
% clc
% V.input = 1;
% [V] = INIT(V);

% Preallocating
DT = zeros(1,V.NC);
V.DT = zeros(1,V.NC);
%% Around aerofoil
for i = 1:(V.ICU - V.ICL)
    % Get connectivity
     K  = V.JLIST(i,2);
    IA  = V.JLIST(i,3);
    IB  = V.JLIST(i,4) ;
    C2  = V.GM*V.P(K)/V.W(K,1);    % gamma*p/rho  = speed of sound ^2
    DX  = V.XP(IB) - V.XP(IA);
    DY  = V.YP(IB) - V.YP(IA);
    DL2 = DX*DX + DY*DY;
    T2  = (abs(C2*DL2))^(1/2);
    U   = V.W(K,2)/V.W(K,1);
    Vv  = V.W(K,3)/V.W(K,1);
    T1  = abs(DY*U - DX*Vv);
```

```
    DT(K) = DT(K) + T1 + T2;
end
%% Around boundary
for i = (V.ICU - V.ICL)+1 : V.NBD
    K   = V.JLIST(i,2);
    IA  = V.JLIST(i,3);
    IB  = V.JLIST(i,4);
    C2  = V.GM*V.BW(i,5)/V.BW(i,1);
    DX  = V.XP(IB) - V.XP(IA);
    DY  = V.YP(IB) - V.YP(IA);
    DL2 = DX*DX + DY*DY;
    T2  = (abs(C2*DL2))^(1/2);
    U   = V.BW(i,2)/V.BW(i,1);
    Vv  = V.BW(i,3)/V.BW(i,1);
    T1  = abs(DY*U - DX*Vv);

    DT(K) = DT(K) + T1 + T2;
end
% This does nto include sqrt(C2*DL2) as it is very small and do not want to
% calculate it as it wastes time.

%% INSIDE DOMAIN CELLS
% Visits every edge and calculates velocity which corresponds to the
% fastest eigenvalue. (largest)
for i = V.NBD + 1: V.NEDGE
    K   = V.JLIST(i,2);
    IA  = V.JLIST(i,3);
    IB  = V.JLIST(i,4);
    IP  = V.JLIST(i,5);
    C2K = V.GM*(V.P(K) + V.P(IP))/(V.W(K,1) + V.W(IP,1));
    UK  = ((V.W(K,2)/V.W(K,1)) + (V.W(IP,2)/V.W(IP,1)))*0.5;    % Average velocity
    VK  = ((V.W(K,3)/V.W(K,1)) + (V.W(IP,3)/V.W(IP,1)))*0.5;
    DX  = V.XP(IB) - V.XP(IA);
    DY  = V.YP(IB) - V.YP(IA);
    DL2 = DX*DX + DY*DY;
    DTTT   = abs(UK*DY - VK*DX) + (C2K*DL2)^(1/2);
    DT(K)  = DT(K) + DTTT;                                  % Both adding as it is time
not flux
    DT(IP) = DT(IP) + DTTT;

%     % Debug
%     C2Kstore(i) = C2K;
end

%% STORING VARIABLES
for i = 1:V.NC
    if DT(i) > 0
        V.DT(i) = 1/DT(i);
    else
        error('Negative or zero time')
    end
end

end
```

### 6.3.6 Boundary condition function

```
%% BCON
% Calculates boundary condtions for each cell
function [V] = BCON(V)

%Debug
% clear
% clc
% V.input = 1;
% [V] = INIT(V);
```

```
for i = (V.ICU - V.ICL) + 1 : V.NBD
    K   = V.JLIST(i,2);
    IA  = V.JLIST(i,3);
    IB  = V.JLIST(i,4);
    DX  = V.XP(IB) - V.XP(IA);
    DY  = V.YP(IB) - V.YP(IA);
    DH  = (DX*DX + DY*DY)^(1/2); %(3)
   STH  = DY/DH;      % Sin
   CTH  = DX/DH;      % Cos
    RE  = V.W(K,1);
    U   = V.W(K,2)/RE;
    Vv  = V.W(K,3)/RE;
    PE  = V.P(K);
    CE  = (abs(V.GM*PE/RE))^(1/2);  % Speed of sound
    XMACH  = ((U*U + Vv*Vv)^(1/2))/CE;    % MACH


 % Freestream variables
    S0  = (V.P0/V.Rho0)^V.GM;       % Entropy of freesteam,  near boundary must stay
constant
   QN0  = (V.U0*STH) - (V.V0*CTH);
   QT0  = (V.U0*CTH) + (V.V0*STH);
   RI0  = QN0 - (2*V.C0/V.GMm1);           % Internal energy

   % Values inside domain
    SE  = (PE/RE)^V.GM;                % LOCAL PROPERTIES, Also entropy but of boundary
   QNE  = (U*STH) - (Vv*CTH);              % E = local properties,
   QTE  = (U*CTH) + (Vv*STH);
   RIE  = QNE + (2*CE/V.GMm1);

   if XMACH < 1 % Subsonic flow

       % Averaging freestreaming and data from last iteration
       QN =         0.5*(RIE + RI0);
       C  = V.GMm1*0.25*(RIE - RI0);
       % If, then it calculates from outside of freestream
       if QN < 0
           QTT = QT0;
           SB  = S0;
       else % Calculates from inside
           QTT = QTE;
           SB  = SE;
       end

   else    % Supersonic flow
       if QN0 < 0
           QN  = QN0;
           C   = V.C0;
           QTT = QT0;
           SB  = S0;
       else
           QN = QNE;
           C  = CE;
          QTT = QTE;
          SB  = SE;
       end
   end

  % Calculate primary variables
   UB  = (QN*STH) + (QTT*CTH);
   VB  = (QTT*STH) - (QN*CTH);
   RB  = (C*C/V.GM/SB)^(1/V.GMm1);
   PB  = C*C*RB/V.GM;
   REB = PB/V.GMm1 + (0.5*RB*(UB*UB + VB*VB));

   % Calculate conservative variables
   V.BW(i,1) = RB;
   V.BW(i,2) = RB*UB;
   V.BW(i,3) = RB*VB;
   V.BW(i,4) = REB;
   V.BW(i,5) = PB;
end
```

```
end
```

### 6.3.7 Grid plot function

```
function [V] = gridplot(V)

% Plot edges on grid for full grid.
% plot inbetween nv1 and nv2

%% Aerofoil grid plot
for i = 1:(V.ICU - V.ICL)  % (Aerofoil upper point 101 - aerofoil lower point 1 = 100 =>
1:100)
    IA = V.JLIST(i,3);
    IB = V.JLIST(i,4);
    XP_AFIA(i) = V.XP(IA);
    XP_AFIB(i) = V.XP(IB);
    YP_AFIA(i) = V.YP(IA);
    YP_AFIB(i) = V.YP(IB);
    plot([XP_AFIA(i) XP_AFIB(i)],[YP_AFIA(i) YP_AFIB(i)],'-','color',[0 0.8
0],'LineWidth',2)
    hold on
    pause(0.001);
end

%% OUTER BOUNDARY
for i = ((V.ICU - V.ICL)+1):V.NBD
    IA = V.JLIST(i,3);
    IB = V.JLIST(i,4);
    XP_OBIA(i) = V.XP(IA);
    XP_OBIB(i) = V.XP(IB);
    YP_OBIA(i) = V.YP(IA);
    YP_OBIB(i) = V.YP(IB);
    plot([XP_OBIA(i) XP_OBIB(i)],[YP_OBIA(i) YP_OBIB(i)],'-','color',[0.8 0
0],'LineWidth',2)
    hold on
    pause(0.001)
end

%%  INSIDE DOMAIN
for i = (V.NBD + 1): V.NEDGE
    IA = V.JLIST(i,3);
    IB = V.JLIST(i,4);
    XP_IDIA(i) = V.XP(IA);
    XP_IDIB(i) = V.XP(IB);
    YP_IDIA(i) = V.YP(IA);
    YP_IDIB(i) = V.YP(IB);
    plot([XP_IDIA(i) XP_IDIB(i)],[YP_IDIA(i) YP_IDIB(i)],'k-')
    hold on
    pause(0.001)
end

figure(1)
set(gcf,'color','w');
xlabel('$$x/c$$','Interpreter', 'Latex');
ylabel('$$y/c$$','Interpreter', 'Latex');
title('C-type grid','FontSize',12)
set(gca,'FontName','Times New Roman','FontSize',12);
end
```

### 6.3.8 Debug functions for first and last iteration

```
function [DB]=debugCFDA3(V,CDat)
```

```
% Establishing data
DB.CW1 = CDat(108:end,1);
DB.CW2 = CDat(108:end,2);
DB.CW3 = CDat(108:end,3);
DB.CW4 = CDat(108:end,4);

% Difference in data
DB.DW1 = V.W(:,1) - DB.CW1;
DB.DW2 = V.W(:,2) - DB.CW2;
DB.DW3 = V.W(:,3) - DB.CW3;
DB.DW4 = V.W(:,4) - DB.CW4;

% Conditions to find indices of wrong data (filters out very small
% differences. Grid parameters in indices, eg, aerofoil surface is incice
% 1:101.
DB.DW1ind = find(DB.DW1 > 1e-5);
DB.DW2ind = find(DB.DW2 > 1e-5);
DB.DW3ind = find(DB.DW3 > 1e-5);
DB.DW4ind = find(DB.DW4 > 1e-5);


% Find grid connectivity points which correspond to wrong data
% Note, could be an over prediction but there is no way to establish if
% wrong data was from K or P cell using indices.

% W1 for K and P
figure(2)
plot(V.XP,V.YP,'k'); hold on
for i = 1:length(DB.DW1ind)
DB.K_W1 = find(V.JLIST(:,2) == DB.DW1ind(i,1));
DB.P_W1 = find(V.JLIST(:,5) == DB.DW1ind(i,1));
   IAK = V.JLIST(DB.K_W1,3);
   IBK = V.JLIST(DB.K_W1,4);
   IAP = V.JLIST(DB.P_W1,3);
   IBP = V.JLIST(DB.P_W1,4);
   plot([V.XP(IAK) V.XP(IBK)],[V.YP(IAK),V.YP(IBK)],'r','LineWidth',2)
   hold on
   plot([V.XP(IAP) V.XP(IBP)],[V.YP(IAP),V.YP(IBP)],'g','LineWidth',2); hold on
   pause(0.01)
end; hold off

figure(3)
plot(V.XP,V.YP,'k'); hold on
for i = 1:length(DB.DW2ind)
DB.K_W2 = find(V.JLIST(:,2) == DB.DW2ind(i,1));
DB.P_W2 = find(V.JLIST(:,5) == DB.DW2ind(i,1));
   IAK = V.JLIST(DB.K_W2,3);
   IBK = V.JLIST(DB.K_W2,4);
   IAP = V.JLIST(DB.P_W2,3);
   IBP = V.JLIST(DB.P_W2,4);
   plot([V.XP(IAK) V.XP(IBK)],[V.YP(IAK),V.YP(IBK)],'r','LineWidth',2)
   hold on
   plot([V.XP(IAP) V.XP(IBP)],[V.YP(IAP),V.YP(IBP)],'g','LineWidth',2); hold on
   pause(0.01)
end; hold off

figure(4)
plot(V.XP,V.YP,'k'); hold on
for i = 1:length(DB.DW3ind)
DB.K_W3 = find(V.JLIST(:,2) == DB.DW3ind(i,1));
DB.P_W3 = find(V.JLIST(:,5) == DB.DW3ind(i,1));
   IAK = V.JLIST(DB.K_W3,3);
   IBK = V.JLIST(DB.K_W3,4);
   IAP = V.JLIST(DB.P_W3,3);
   IBP = V.JLIST(DB.P_W3,4);

   plot([V.XP(IAK) V.XP(IBK)],[V.YP(IAK),V.YP(IBK)],'r','LineWidth',2)
   hold on
   plot([V.XP(IAP) V.XP(IBP)],[V.YP(IAP),V.YP(IBP)],'g','LineWidth',2); hold on
   pause(0.01)
end; hold off
```

```
figure(5)
plot(V.XP,V.YP,'k'); hold on
for i = 1:length(DB.DW4ind)
DB.K_W4 = find(V.JLIST(:,2) == DB.DW4ind(i,1));
DB.P_W4 = find(V.JLIST(:,5) == DB.DW4ind(i,1));
   IAK = V.JLIST(DB.K_W4,3);
   IBK = V.JLIST(DB.K_W4,4);
   IAP = V.JLIST(DB.P_W4,3);
   IBP = V.JLIST(DB.P_W4,4);
   plot([V.XP(IAK) V.XP(IBK)],[V.YP(IAK),V.YP(IBK)],'r','LineWidth',2); hold on
   plot([V.XP(IAP) V.XP(IBP)],[V.YP(IAP),V.YP(IBP)],'g','LineWidth',2); hold on
   pause(0.01)
end

end



function [DB] = debugCFDA3ENDIT(V,EC_dat)

W1_EI = EC_dat(2107:end,1);
W2_EI = EC_dat(2107:end,2);
W3_EI = EC_dat(2107:end,3);
W4_EI = EC_dat(2107:end,4);

DB.DW1 = abs(V.W(:,1) - W1_EI);
DB.DW2 = abs(V.W(:,2) - W2_EI);
DB.DW3 = abs(V.W(:,3) - W3_EI);
DB.DW4 = abs(V.W(:,4) - W4_EI);

% Conditions to find indices of wrong data (filters out very small
% differences. Grid parameters in indices, eg, aerofoil surface is incice
% 1:101.
DB.DW1ind = find(DB.DW1 > 1e-2);
DB.DW2ind = find(DB.DW2 > 1e-2);
DB.DW3ind = find(DB.DW3 > 1e-2);
DB.DW4ind = find(DB.DW4 > 1e-2);


% Find grid connectivity points which correspond to wrong data
% Note, could be an over prediction but there is no way to establish if
% wrong data was from K or P cell using indices.

% W1 for K and P
figure(2)
plot(V.XP,V.YP,'k'); hold on
for i = 1:length(DB.DW1ind)
DB.K_W1 = find(V.JLIST(:,2) == DB.DW1ind(i,1));
DB.P_W1 = find(V.JLIST(:,5) == DB.DW1ind(i,1));
   IAK = V.JLIST(DB.K_W1,3);
   IBK = V.JLIST(DB.K_W1,4);
   IAP = V.JLIST(DB.P_W1,3);
   IBP = V.JLIST(DB.P_W1,4);
   plot([V.XP(IAK) V.XP(IBK)],[V.YP(IAK),V.YP(IBK)],'r','LineWidth',2)
   hold on
   plot([V.XP(IAP) V.XP(IBP)],[V.YP(IAP),V.YP(IBP)],'g','LineWidth',2); hold on
   pause(0.01)
end; hold off

figure(3)
plot(V.XP,V.YP,'k'); hold on
for i = 1:length(DB.DW2ind)
DB.K_W2 = find(V.JLIST(:,2) == DB.DW2ind(i,1));
DB.P_W2 = find(V.JLIST(:,5) == DB.DW2ind(i,1));
   IAK = V.JLIST(DB.K_W2,3);
   IBK = V.JLIST(DB.K_W2,4);
   IAP = V.JLIST(DB.P_W2,3);
   IBP = V.JLIST(DB.P_W2,4);
   plot([V.XP(IAK) V.XP(IBK)],[V.YP(IAK),V.YP(IBK)],'r','LineWidth',2)
   hold on
   plot([V.XP(IAP) V.XP(IBP)],[V.YP(IAP),V.YP(IBP)],'g','LineWidth',2); hold on
```

```
   pause(0.01)
end; hold off

figure(4)
plot(V.XP,V.YP,'k'); hold on
for i = 1:length(DB.DW3ind)
DB.K_W3 = find(V.JLIST(:,2) == DB.DW3ind(i,1));
DB.P_W3 = find(V.JLIST(:,5) == DB.DW3ind(i,1));
   IAK = V.JLIST(DB.K_W3,3);
   IBK = V.JLIST(DB.K_W3,4);
   IAP = V.JLIST(DB.P_W3,3);
   IBP = V.JLIST(DB.P_W3,4);

   plot([V.XP(IAK) V.XP(IBK)],[V.YP(IAK),V.YP(IBK)],'r','LineWidth',2)
   hold on
   plot([V.XP(IAP) V.XP(IBP)],[V.YP(IAP),V.YP(IBP)],'g','LineWidth',2); hold on
   pause(0.01)
end; hold off

figure(5)
plot(V.XP,V.YP,'k'); hold on
for i = 1:length(DB.DW4ind)
DB.K_W4 = find(V.JLIST(:,2) == DB.DW4ind(i,1));
DB.P_W4 = find(V.JLIST(:,5) == DB.DW4ind(i,1));
   IAK = V.JLIST(DB.K_W4,3);
   IBK = V.JLIST(DB.K_W4,4);
   IAP = V.JLIST(DB.P_W4,3);
   IBP = V.JLIST(DB.P_W4,4);
   plot([V.XP(IAK) V.XP(IBK)],[V.YP(IAK),V.YP(IBK)],'r','LineWidth',2); hold on
   plot([V.XP(IAP) V.XP(IBP)],[V.YP(IAP),V.YP(IBP)],'g','LineWidth',2); hold on
   pause(0.01)
end
end
```