Part IIB 4A2, 2021-22

WRITING AN EULER SOLVER FOR INTERNAL FLOW

Introduction

The objective of this exercise is to write a program for solving the Euler equations for two dimensional internal flows using a time marching method. You are provided with the skeleton of a program and are guided by these notes on how to complete it to obtain a simple program that is stable but not very accurate. You will then be given suggestions for methods to extend the program to make it faster and more accurate and you should complete as many of these as you have time for and include them in your final report.

The resulting program will not take long to run on the DPO system (0.5 – 2 minutes) for the simple test cases. The larger test cases which will be tackled later will take considerably longer, so it is worth taking care to code it efficiently. This is always important for big programs and simply means that, if you find yourself calculating the same thing in different parts of the program, then define a variable to hold the result, calculate it once and simply use the variable. These rules are much more important in the main time stepping loop which will be executed thousands of times.

Copying the programs and data

You will work remotely on the computers of the teaching system. To access to them from a **Linux** platform (install **MobaXterm** first if you use Windows platform), use the Linux commands:

ssh -X CRSid@gate.eng.cam.ac.uk

ssh -X CRSid@ts-access

You are in your **home directory** on the teaching system. Copy the skeleton of the program, and all test cases from the module directory and move into the directory **2021_4A2**, using:

cp -r /public/teach/4A2/2021_4A2 .

cd 2021_4A2

Remember that the last **.** is essential in the first command. The main program **euler.f90** and the other **.f90** files in the directory **2021_4A2** form the skeleton of the program that you are going to write. The compilation of these files into an executable code is managed by a Makefile. See the file **README** for the details. You are suggested to work first on **test case 0**. The **.f90** files, the data sets **test0_flow** and **test0_geom** can be edited using any editor with which you are familiar, but there are advantages to using an editor which is "fortran aware" such as "emacs".

The data sets **test0_flow** and **test0_geom** are for you to use to check your program against a standard solution which will be provided.

There are other commands which you will use, **eulplt** and **pltconv**, which are alternative graphics programs that you can use to plot your results and the convergence of your solution. You cannot modify these programs. They should be available to you automatically on the DPO system. You can test this by typing the command

which eulplt

If this returns the answer

/usr/local/bin/eulplt

Geometry to be Calculated and Indexing of the Grid

The geometry we will calculate is a two-dimensional duct bounded by upper and lower solid walls. We will assume that the flow is basically from left to right and the *i* index of the grid increases in the direction of flow whilst the *j* index increases in the cross-flow direction. We also need to establish whether *i* and *j* form a right handed or a left handed coordinate system and will take *j* to increase to the left as we look along an *j* = constant grid line in the direction of the flow. The grid will look something as shown below.

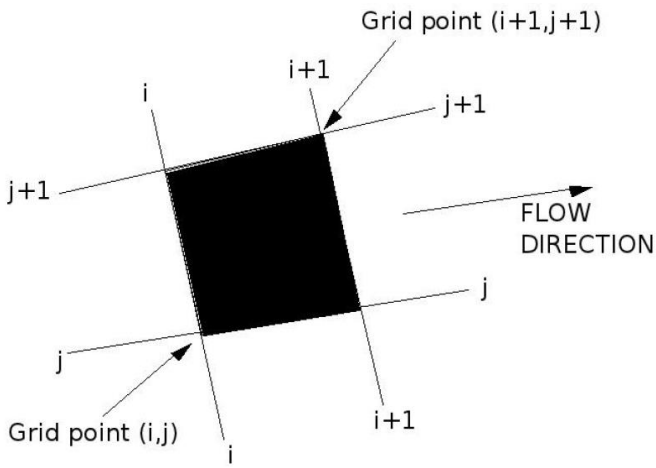

Figure 1: Finite Volume Grid for Duct Flow

We will use cell corner storage for all the flow variables, so that these are stored at all points where the *i*-lines and the *j*-lines cross (nodes) and there will be $ni \times nj$ of these. You will find that we do not need to use or store the average values of the variables in the cells directly.

We will use the finite volume method of discretising the equations. The finite volumes to which the conservation equations are applied are the quadrilateral elements (also often called cells or just

volumes although in your 2D program they are really areas) formed between two *i* lines and two *j* lines in the above mesh. There are (*ni*−1) × (*nj*−1) such elements. Each element will be referenced by the *i, j* values at its *bottom left hand corner* when looking along a constant *i* line in the direction of increasing *j* .

We will use cell corner storage for all the flow variables so that these are stored at all the points where the "*i*" lines and the "*j*" lines cross. You will find that we do not need to use or store the average values of the variables in the cells.

Method of Solution

The solution technique that we will use at first is the simplest possible one. It is analogous to the Lax-Wendroff method that you used in the one-dimensional demonstration last year in 3A3. However, the Lax-Wendroff method proper as applied to the Euler equations is quite complex, and the method we will use is called simply the "Lax" method (or "Lax-Friedrichs" method in Hirsch). In it, all the spatial derivatives are obtained by central differencing, which is inherently unstable, and then the solution is stabilised by adding enough artificial viscosity, or smoothing, to make it stable.

For example, in one dimension and in finite volume form, the continuity equation is

$$\rho_i^{n+1} = \rho_i^n - \frac{\Delta t}{2A_i\Delta x}\left[(\rho VA)_{i+1}^n - (\rho VA)_{i-1}^n\right] + \frac{1}{2}\times SF \times CFL \times \left(\rho_{i+1}^n - 2\rho_i^n + \rho_{i-1}^n\right)$$

A_i is the area of the face of element and the artificial viscosity, *SF*, is chosen by trial and error to be large enough to maintain stability. *CFL* is the CFL number $(U+c)\,\Delta t/\Delta x$. The other conservation equations have similar artificial viscosity terms.

This method requires a lot of artificial viscosity to make it stable and, as a result, the method is not very accurate and is not now widely used. However, we will see later how it can very easily be improved to become a respectable method.

Program Structure

The program **euler.f90** consists of the main program and the following subroutines:

```
program euler
      call read_data
      call inguess
      call generate_grid
      call check_grid
      call crude_guess
!     call flow_guess
      call set_timestep
      do nstep = 1,nsteps
         ro_start = ro
         call set_others
         call apply_bconds
         call set_fluxes
         call sum_fluxes
         ro = ro_start + ro_inc
         call smooth
         call check_conv
      end do
      call output
      stop
```

Subroutine **flow_guess** is commented out in the initial version and is replaced by **crude_guess** which you can use to get started. However, **flow_guess** should be completed later to enable you to get a faster convergence.

Plotting the Output

Subroutine "**output_hg"** writes a file "**euler.plt**" which can be read and plotted by the graphics package "**eulplt**". To plot your results simply type **eulplt** and follow the menu. You will be asked for the name of the plotting file which is "**euler.plt**".

If you wish to plot out any array at any time during the running of the program, you can call **output** with that array as an argument, e.g. call output(dlix) . The array "dlix" can then be plotted as contours or line plots by choosing item 9 on the menu of **eulplt**. This is useful for debugging.

eulplt is a cut down version of a 3D plotting package and so some of the options in it (e.g. to plot radial velocity or efficiency) are not applicable to your 2D solution.

The solver also writes a file ("**euler.log**"), containing the convergence history of the calculation, to Fortran unit 3. After you have completed a run you can plot the convergence, or lack of it, using the program "**pltconv**". Simply type **pltconv** and a plot should appear on the screen. The plot shows the average changes (i.e. averaged over all ***i*** and ***j* grid points**) in each of the primary variables over the last 5 time steps, against time step number.

Approximate Time Schedule for Progress through the module

Once you have the skeleton program you can complete it at your own pace but an approximate schedule would be:

Week 1 (Oct 14-20)	Subroutines: read_data, generate_grid, check_grid, flow_guess.
Week 2 (Oct 21-27)	Subroutines: set_others, set_timestep, apply_bconds, set_fluxes.
Week 3 (Oct 28-Nov 3)	Subroutines: sum_fluxes, smooth, check_conv.
Week 4 (Nov 4-10)	Run test cases and prepare interim report.
Week 5 (Nov 11-17)	Add improvements to the method.
Week 6 (Nov 18-24)	Run more test cases.
Week 7 (Nov 25-Dec 1)	Add more improvements.
Week 8 (Dec 2-9)	Write final report.

Completing the subroutines

subroutne **read_data**

The data for the calculation is provided in two files "**geom**" and "**flow**". (copy test0_geom to geom, etc)

"**geom**" is read from unit 1 and consists of a table of coordinates of the duct in the form:

title	The title of the data set.
ni, nj	The number of grid points in the streamwise (*i*) and cross flow (*j*) directions. Suitable initial values would be *ni*=60, *nj*=20 . The maximum grid point numbers allowed for in the array dimensions are 201 × 51. You can change these by editing the parameters i_max and j_max if necessary but they should be big enough for all your test cases.

Followed by *ni* values of:

xlow(i), ylow(i), xhigh(i), yhigh(i)

the x and y coordinates of the lower and upper boundaries of the duct.

All the above data can be read in free format using: read(1,*) .

The second data file "**flow**" is read from unit 2 and gives the data needed to specify the flow conditions through the channel. It consists of the following data, read sequentially:

rgas	The gas constant in J/kg K (287.5 for air)
gamma	The gas specific heat ratio (1.4 for air)
poin	The inlet stagnation pressure in N/m^2
toin	The inlet stagnation temperature in K.
alpha1	The inlet flow direction in degrees. This must be compatible with the duct geometry.
pdown	The exit static pressure in N/m^2. This determines the mass flow rate and Mach numbers. Do not be too ambitious at first.
cfl	The CFL number. Less than 1, try 0.5 initially.
smooth_fac_in	The smoothing factor which should initially be about 0.5 .
nsteps	The maximum number of time steps allowed. Typically 3000.
conlim	The convergence tolerance. Typically = 0.0001 .

This data can be read using: read(2,*)

You can edit the file "**flow**" as required to change the flow conditions. For your initial solution choose a moderate Mach number level so that p_2/p_{o1} = about 0.9. It is also essential that the inlet flow direction that you specify, alpha1, is compatible with the angles of the walls of the duct at inlet.

After you have written your subroutine, compile it by typing

make

This creates an executable code **Euler**. Make sure that you have generated data sets "**geom**" and "**flow**" then run your program by typing "./**Euler**" . If the program reads the data satisfactorily move on to the next subroutine. You cannot plot any results at this stage because you have not yet generated a grid.

Back up your program before starting on the next subroutine.

subroutine **generate_grid**

This subroutine generates the grid and works out the areas of the elements and the projected lengths of each face in both the coordinate directions. It also finds the minimum side of any element, which is used to set the time step.

The grid points are generated by taking them to be evenly spaced along straight lines joining (xlow,ylow) to (xhigh,yhigh). While it is not essential that the lines are straight or that the points are evenly spaced (and in many applications this would not be the case), we will use this method for simplicity, you can easily change it later on if you are interested. Generate all the x(i,j) and y(i,j) on this basis.

Now evaluate the areas of all the (ni-1)×(nj-1) elements and store them in an array area(i,j) . The area is half the vector cross product of the diagonals but it is up to you to make sure that you take the cross product in such a way as to obtain a positive result.

Also evaluate the projected lengths (or areas in 3D) of the ***left hand*** *i* face and the ***lower*** *j* face of each element in the x and y directions. The projected lengths are the components of a vector of magnitude equal to the length of the face but directed perpendicular to the face. Store these as dlix(i,j), dliy(i,j), dljx(i,j), dljy(i,j). Because every face is common to two elements you only need to do this for two of the faces of each element. The values for the other faces can be obtained from those stored for adjacent elements when needed.

dli is the vector representing the “area” of the "i = const" face of the element (i,j) and dlix(i,j), dliy(i,j) are the components of this vector in the x and y directions respectively. Similarly for dlj.

When evaluating these projected areas try not to use any angles as this can lead to confusion as to which quadrant the angle is in. It is not at all necessary or even the most convenient way.

In the same "do loops" it is convenient to evaluate the length of the smallest side for any element as we shall need this to set the time step. Store this as "dmin".

Back up your program before starting on the next subroutine.

subroutine **check_grid**

It is *very important* that you have generated the grid and its projected lengths and areas correctly, as you will have lots of problems later if not. Hence, you must check the grid areas and projected areas v*ery carefully. Experience shows this can save you a great deal of effort later on.*

After generating the grid you can call subroutine "**output**" to generate a plotting file and then plot the grid using "**eulplt**" or matlab, or paraview if you prefer. Option 14 of the menu plots the grid. You can also plot the areas you have calculated by using: **call output(area)** and selecting item 9 on the menu of **eulplot**. At this stage you cannot plot any of the flow variables. Do this for both a "**bump**" and a "**bend**" data set (other test cases in /public/tech/4A2 to ensure that your subroutine works generally.

You should also check that the areas you calculated are correct. Print out some areas and compare with hand calculations. Above all, check that for every cell the sum of the projected lengths of the 4 faces of every cell, in both the x and y directions, is effectively zero, i.e. less than about 0.000001 * dmin. If this is not the case then a uniform pressure acting around the element will produce a force on it and the fluid in it will accelerate when there is no flow and no pressure gradient.

Back up your program before starting on the next subroutine.

subroutine **flow_guess**

If you wish to get started quickly you can leave this subroutine until later and just use the completed subroutine **crude_guess.** Your program will take longer to converge but initially it is more important to get it working than to make it efficient.

The initial guess of the flow is quite arbitrary and sometimes the solution is started from a guess that the velocities are everywhere zero and the pressure is uniform. However, the better the initial guess the faster your program will converge and so it is worth going to some trouble to make a reasonable guess. We will assume that the flow at any *i* station does not vary in the *j* direction. i.e. that it is one-dimensional.

We first estimate the exit flow conditions by assuming isentropic flow from the inlet stagnation conditions to the exit static pressure. Assuming that the flow is aligned perpendicular to the "*i*" = constant grid lines enables the width of the passage to be evaluated at every "*i*" line. The value of this width at exit, together with the exit density and velocity can be used to estimate the mass flow rate passing through the duct per unit depth in the direction perpendicular to the x, y plane. We will always calculate the mass flow in future by assuming unit depth.

Knowing the mass flow rate, the passage width at every "*i*" station, and assuming isentropic flow enables the flow conditions at every "*i*" station to be estimated. However, obtaining this exactly requires iteration and we will only bother to do a single loop of the iteration. It is also necessary to check that the flow is not highly supersonic as this would cause the iteration to diverge.

First guess that the density is everywhere the same as the exit density. Use this to calculate the velocity at every "i" grid line from continuity. Then use this velocity to calculate a new density at every "i" grid line from the steady flow energy equation plus an assumption of isentropic flow. Use this density and continuity to estimate a new velocity. Check that the Mach number is not greater than a limiting value mlim and set the velocity to this limit if it exceeds it.

Take the velocity as calculated above to be the component along the "*j* = constant" lines with no component perpendicular to these lines. Hence find the x and y velocity components (which must be named vx(i,j) and vy(i,j)) by resolving into the coordinate directions.

Having obtained a guess of the density and velocity at every grid point we can obtain any other flow property required by assuming isentropic flow from the inlet stagnation pressure and temperature. At this stage we only need to set the primary flow variables ro, rovx, rovy and roe. Note that the internal energy includes the kinetic energy term, $e = c_v T + .5v^2$.

After you have generated your guess of the flow you can call subroutine "**output**" to generate a plotting file and then use **"eulplt"** (or matlab or paraview) to check your guess. You cannot plot the pressure at this stage as you have not yet set it.

Back up your program before starting on the next subroutine.

subroutine **set_timestep**

This is a very simple operation and would not be worth putting in a subroutine except that we may want to change to using spatially variable timesteps at a later stage. Initially, however, just evaluate the uniform timestep that will be applied to all elements based on the size of the smallest element, dmin.

We need to estimate the speed of sound and the flow velocity for the smallest (i.e. the least stable) element and a pessimistic estimate is to take both of these equal to the stagnation speed of sound.

The timestep that you calculate must be called deltat which is the name assumed by other subroutines.

Back up your program before starting on the next subroutine.

subroutine **set_others**

This is another very simple subroutine which calculates the secondary variables given the primary ones. The primary variables are those that you set in **flow_guess**. The secondary variables that must be set at every grid point are:

The velocity components vx(i,j) and vy(i,j).

The static pressure p(i,j).

The stagnation enthalpy ho(i,j).

This is the first subroutine to be included in the main time stepping loop. It will be called thousands of times and so efficient coding is important.

Back up your program before starting on the next subroutine.

subroutine **apply_bconds**

This subroutine applies the boundary conditions at the inlet and exit of the calculated domain. The other boundary conditions of no flow through the solid surfaces is most conveniently applied in **set_fluxes** .

We will initially assume that the flow is subsonic at both inlet and outlet. This means that we must specify P_O, T_O (and hence ρ_o) and the inlet flow direction at the inlet boundary, and the static pressure at the downstream boundary.

At the downstream boundary simply set p(i,j) = pdown for all values of j .

At the inlet boundary we use the value of density calculated from the solution, together with an assumption of isentropic flow from the specified inlet stagnation conditions and flow angle, to calculate the velocity components, hence rovx and rovy and the pressure and internal energy. At low Mach numbers small changes in density cause a large change in velocity This can be seen from Bernoulli which gives:

$$\delta v = -c^2 \frac{\delta \rho}{\rho v}$$

where c is the speed of sound. This in turn produces a large change in mass flux, which can cause an over-correction of the density in the next time step. (You can try a point stability analysis on this).

Hence the procedure can become unstable. To prevent this we relax the change in inlet density by taking only part of the calculated change on every step. i.e.

```
roinlet(j) = (1-rfin)*roinlet(j) + rfin*ro(1,j)
```

where rfin needs to be reduced at low Mach numbers. Unless you want to try extremely low velocity flows a value of rfin = 0.25 should be suitable. Note that this procedure is not "time accurate" and if we wish to maintain time accuracy we must consider the waves reflected from the boundary.

It is also possible for the inlet density to rise above the specified inlet stagnation value (i.e. above poin/(rgas*toin)) during the transient part of the calculation, for example a shock wave moving upstream can increase the pressure to quite arbitrary values behind the shock. This would cause failure of our boundary condition because it involves taking the square root of (toin-t) to find the velocity. To prevent this it is safer to check that roinlet is not greater than the stagnation value and to set it to just less than this value if it is, e.g.

```
if( roinlet(j)>0.999*rostagin )  roinlet(j) = 0.999*rostagin
```

This prevents failure during the transient calculation but will not affect the steady flow for which the inlet static density must be less than the specified stagnation value.

Back up your program before starting on the next subroutine.

subroutine **set_fluxes**

This subroutine calculates the fluxes for all the primary variables. Once calculated, the fluxes are stored and not used in this subroutine. We only need to calculate and store the flux through two faces of each cell, because every face is shared by 2 cells. The fluxes for the cell (icell,jcell) are calculated through its i = icell and j= jcell faces, whilst the fluxes through its i = icell+1 and j=jcell+1 faces are calculated and stored for the adjacent cells.

The equations for the fluxes are given in the notes on "The Finite Volume Method". We are using cell corner storage and so the average value of a variable on a face is always obtained by averaging the values stored at the ends of the face.

Be careful with the names. The flux variables must be named as follows.

fluxi_mass(i,j) , fluxj_mass(i,j)
fluxi_xmom(i,j) , fluxj_xmom(i,j)
fluxi_ymom(i,j) , fluxj_ymom(i,j)
fluxi_enth(i,j) , fluxj_enth(i,j)

Whilst setting the mass fluxes it is convenient to apply the boundary condition that there is no mass flow through the solid walls at j=1 and j=nj . It is also useful to integrate the mass flow across every " *i* = constant " line to find the mass flow rate crossing it. This must be named flow(i) and is used in checking the convergence.

Check your coding of the flux terms in this subroutine *very carefully*. Any errors in them can waste hours of your time later on when you try to run the program.

This subroutine is within the main time stepping loop, it is called thousands of times and uses about 25% of the total CPU time. There is scope for minimising the number of arithmetic operations within it.

Back up your program before starting on the next subroutine.

subroutine **sum_fluxes**

This subroutine is somewhat different from the previous ones in that it is called with arguments. This is because we wish to perform exactly the same operations on the fluxes for all 4 equations. Rather than writing the same code 4 times with different flux terms, it is much more convenient to pass over the fluxes as arguments of the subroutine and only write the code to sum them once. The fluxes and primary variables are called by different names within the subroutine, e.g.

call sum_fluxes (fluxi_mass,fluxj_mass,delro,ro_inc)

subroutine sum_fluxes(iflux,jflux ,delprop, prop_inc)

The fluxes become "iflux" and "jflux" and the primary variable cell residual becomes "delprop", the change in the variable that we will calculate will be called "prop_inc". These new variables must be dimensioned within the subroutine.

The subroutine first sums the fluxes for each cell and calculates the change in the primary variable in the *cell*. It stores this in the array variable "delprop(i,j)".

The change in the primary variable is then distributed to the corners of the cell. Each interior corner is shared by 4 cells and to ensure that each corner node receives a full update we give it 1/4 of the change from each of its adjacent cells. The boundary nodes are only shared by 2 cells and so do not get their fair share op the update unless we add extra changes to them. Hence, interior nodes get 1/4 of the changes calculated for the cells and boundary nodes get 1/2 of the changes for the adjacent cells. This applies to both the inlet and exit boundaries and to the solid boundaries. The 4 corner nodes of the whole grid must be given the whole of the change from the single cell of which they form a corner. By ensuring that every node gets the same total proportion of the changes we are maintaining time accuracy.

“delprop” is used for checking convergence and in future developments to the algorithm. If it has been modified from its original use during an extension, restore it before ending the routine and passing it back to the main program via the subroutine arguments (this comment is not relevant to the initial basic scheme)

This subroutine is called 4 times within each main loop. It uses about 1/4 of the total CPU time so again try to minimise the number or arithmetic operations.

Back up your program before starting on the next subroutine.

subroutine **smooth**

This subroutine applies the smoothing or artificial viscosity needed to stabilise the solution. There are many ways of doing this but again we will initially choose the simplest. Hence, after every time step, we simply smooth the values of the primary variables towards the average of the values at its neighbouring points. On a uniformly spaced rectangular grid this is equivalent to adding a term proportional to: **smooth_fac * ∇^2 (prop)** to the equation for the primary variable prop. In practice (to save CPU time) it is not usual to allow for any non-uniformity in grid spacing and so our simple approach is quite conventional.

To implement this we simply use

$$prop_{smooth} = prop_{calculated} + smooth\_fac*(prop_{avg} - prop_{calculated})$$

or $$prop_{smoothed} = (1-smooth\_fac)*prop_{calculated} + smooth\_fac*prop_{avg}$$

The artificial change in prop due to the smoothing on every time step is proportional to "smooth_fac", whilst the physically correct change produced by the fluxes is proportional to the length of the time step which is determined by "cfl" . Hence, in order to make the ratio of the change produced by smoothing to the correct change independent of the CFL value (and so to maintain the same effect of the artificial viscosity on the solution for all CFL numbers) the input value of "smooth_fac" was multiplied by " cfl " within subroutine "**read_data**" .

In practice we must find out by trial and error how large a value of " cfl " and how small a value of " smooth_fac " we can use.

Once again the subroutine is called with arguments in order to save writing out the same coding 4 times. In the subroutine the primary variable being smoothed becomes "prop". In order to avoid using some already smoothed values in the estimate of $prop_{avg}$ it is safer to firstly store the smoothed values in a new array called store and then reset store to prop before leaving the subroutine. This may not be essential and uses a bit more CPU time but it avoids any possibility of asymmetrical smoothing affecting the solution.

For the interior points it is easy to express $prop_{avg}$ as being the average of the values at the 4 surrounding points minus the central value.

$$prop_{avg} = 0.25*(prop(i+1,j) + prop(i-1,j) + prop(i,j-1) + prop(i,j+1))$$

For the boundary points we must use one-sided averaging.

e.g. at j=1 we take $prop_{avg}$ as

$$prop_{avg} = [prop(i-1,1)+prop(I+1,1)+2.*prop(i,2)-prop(i,3)]/3$$

and similarly at the other boundaries.

This subroutine is called 4 times within the main loop and so uses a significant proportion (about 15%) of the CPU time.

Back up your program before starting on the next subroutine.

subroutine **check_conv**

You do not need to make any changes to this subroutine

This subroutine checks the convergence of the whole calculation and writes out an output summary to unit 6 (which is usually the screen) and the convergence history file to unit 3. The subroutine is within the main loop but, to save time, it is suggested that you call it every 5 steps to save CPU time. This is done in the main loop by using the expression:

```
if(mod(nstep,5)==0)
```

which is only true when nstep is exactly divisible by 5 .

The subroutine checks the change in the primary variables at all grid points to find the maximum absolute change for each variable and also the average change over all grid points. The changes are calculated relative to the values stored as ro_old(i,j), rovx_old(i,j), etc, which are set at the end of the subroutine. Hence the changes are calculated over the last 5 time steps, not over one time step. This should be reflected in the value of conlim used for the convergence check. All changes are non-dimensionalised by a reference value of the variable, which is usually the isentropic change between inlet stagnation and outlet static conditions.

For the short output summary to unit 6 the value of the maximum absolute change of rovx and the average change of rovx for all grid points are printed out together with the i and j locations of the maximum change.

The convergence of the whole calculation is checked by comparing the maximum and the average values of the non-dimensional change in rovx with the convergence limit conlim, which is the input value conlim_in multiplied by the CFL number used. For convergence the maximum change must be less than conlim and the average change must be less than 0.5* conlim.

This subroutine is already completed, and you do not have to make any changes to it.

Back up your program which is now completed.