4A2 EULER EXTENSIONS 2021

For the final part of the exercise, it is up to you to extend and improve your Euler program. Suggestions of methods to do this are included in this note. It is not expected that you will do more than three or four of the extensions and there is no need to try them in the order that they are presented below. For convenience the extensions have been roughly classified as A, B and C. Where class A improves the accuracy of the algorithm in time, Class B improves the accuracy in space and class C improves the speed of the calculation. It is suggested that you try one of each class. You may also create an extension of your own as an alternative to one of the suggestions. You should then apply the final version of your code to the test cases which will be provided and include the results in your final report.

Some of the suggestions are means to speed up the calculation. Some do this by reducing the number of time steps to convergence and some reduce the CPU time per time step. A method that increases the time step length but requires more CPU time to reach convergence is of no advantage. To check this, you can time each run by running it with the command:

time ./Euler instead of just **./Euler**

The second of the three times printed out at the end of the run is the CPU time, i.e. a measure of the computer resources used. (The first time printed out is the elapsed time and the third is the input/output time).

Once your program is "debugged" you can make it run much faster by using an optimising compiler. This is not very good at finding errors in the coding but does speed up the calculation considerably. Edit the file **Makefile** and select the optimising compiler option **-O2**, clean all **.o** files by typing

make clean

and recompile all **.f90** files, by typing

make

The executable program is called **Euler** as before.

Before starting on any extensions you should explore the limitations of your basic LAX scheme by varying the CFL number and the smoothing factor. Note how these affect the stability, the number of steps to convergence and the accuracy of the solution. This will give you a basis against which to compare any improvements.

Remember to keep a copy of latest version of your program before starting on any of these modifications.

CONSTANT STAGNATION ENTHALPY FLOW (CLASS C)

This is a simplification that is possible if we discard the need for time accuracy and if the flow is adiabatic and the inlet stagnation enthalpy is uniform. The steady flow solution must then have uniform stagnation enthalpy (from the steady flow energy equation with no heat flow and no work). In this case we can replace the solution of the energy equation by simply specifying that h_0 = cp * tstagin and then calculating p from ro and ts. This can most conveniently be done in subroutine **"set_others"** . Remember that roe is now no longer being calculated and so cannot be used to calculate any of the other variables.

Check how this modification affects the solution, the maximum stable time step, the number of time steps for convergence and the run time.

SPATIALLY VARIABLE TIME STEPS (CLASS C)

If we discard time accuracy, we can also use different time steps for different elements. This is because the steady state equations reduce to

$$\Sigma \text{ FLUXES} = 0,$$

for every element and for each equation. When we reach the steady state we are solving

$$\Sigma \text{ (FLUXES } \Delta t/\Delta \text{vol)} = 0.$$

So the true steady state equations are satisfied no matter what the length of the time step Δt.
Hence we can take different time steps for different elements and still reach the same steady state solution. If the average time step is increased we should reach the steady state more quickly.

The stable time step for each element is determined by the length of its smallest side and by the sum of its **local** speed of sound and **local** velocity.

$$\Delta t = dmin/(c + v)$$

Hence we define a new array step(i,j) and calculate values of this for each cell. This can initially be done in subroutine **"set_timestep"** using the guessed velocities and Mach numbers. dmin must now be calculated separately for every element and the speed of sound and velocity must also be calculated as the average values for the element based on the values at its 4 corners.

The local stable timestep will change as the calculation converges and the local Mach number and velocity change. Hence, a further improvement is to recalculate step(i,j), using the latest velocities and Mach numbers, say every 5-10 steps of the main loop. This can be done by including your modified **set_timestep** in the main loop and calling it only every 5-10 steps. There is a small overhead in doing this but the advantages should outweigh the costs.

The array step needs to be dimensioned in the main program header so that all parts of the program can access it.

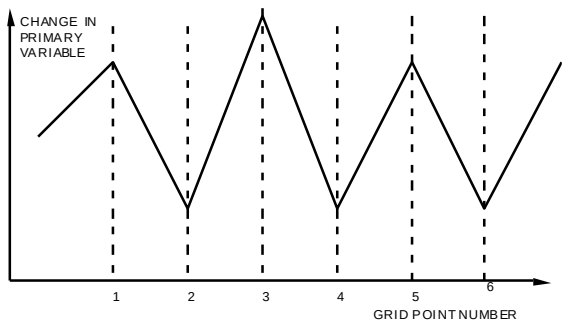
Check the maximum stable CFL value, the run time and the rate of convergence with this addition. You might find that the maximum possible value of CFL is reduced. This is because the most unstable element was probably being updated by much less than its stable time step in the original method whilst it will now be updated by its own maximum stable time step.

Variable time steps give little improvement when the grid size is reasonably uniform (as in our problems) but are very advantageous for viscous calculations when the grid size must vary enormously between the boundary layers and the mainstream flow.

RESIDUAL AVERAGING (CLASS C)

This is a technique that can be used to allow increased lengths of time step at the expense of some extra calculation.

When the calculation becomes unstable it is found that alternate grid points have high and low values of the change in variable and the changes at any one grid point oscillate between high and low values on alternate time steps.



If the time step is too large the amplitude of this zig-zag pattern grows from one step to the next until the calculation fails. This suggests that we might be able to damp down the instability by smoothing the changes in the primary variables before we add them to the old values. i.e.

$\Delta(\text{PROP})_{\text{used at i}} = (1.\text{-SF})*\Delta(\text{PROP})_i + 0.5*\text{SF}*(\Delta(\text{PROP})_{i-1} + \Delta\text{PROP}_{i+1})$

Provided that we reach a steady state solution where

$$\Delta(\text{PROP})_i = \Sigma\ (\text{FLUXES}\ \Delta t/\Delta\text{vol}) = 0,$$

then our solution will still be correct because if the values of $\Delta(\text{PROP})_i$ become zero for every cell then the above smoothing of the "residuals" will not change them or the solution.

Note that this is quite different from using smoothing to add artificial viscosity. We are smoothing the *changes in the variables* and *not the variables themselves* and this should permit longer time steps but not change the solution.

The technique is very easily applied in our code by simply calling the smoothing subroutine with the values of the changes calculated per cell rather than the primary variables as the argument, i.e. within subroutine **sum_fluxes,** having calculated the changes for the elements and saved then as store(i,j), we add a line

```
call smooth(store, 0.5, ni-1,nj-1)
```

Note that the number of elements is ni-1 and nj-1 and so these need to be sent as arguments to smooth and the limits of the do loops in smooth changed accordingly. The smoothing factor also has to be added to the argument list so that we can use different values for the variables and for the residuals. For the residuals. The smoothing factor needs to be fairly large, of order 0.5 as shown above, in order to get significant benefit but the optimum value must be found by trial and error.

An alternative to modifying the arguments to smooth is to take a copy of smooth and use it to create your own routine res_smooth. smooth can then be used for variables, res_smooth for residuals.

In practice this "residual averaging is usually applied "implicitly" which means that we base the new smoothed value at the central point on values which have been already smoothed at the adjacent points. This involves a line matrix inversion and so is considerably more complex. A simple approach to implicit residual averaging can be achieved by calling smooth several times with reduced values of the smoothing factor on each call, e.g. instead of 1 call with sf = 0.5 use 5 calls each with sf = 0.1.

Explore how the maximum stable time step and the rate of convergence vary with the amount of residual smoothing and the smoothing factor. Because the extra smoothing uses significant extra work, especially if it is called several times, it is necessary to check the run times to decide on the overall benefit. The benefits of this method seem to vary with the numerical scheme. For the simple LAX scheme it gives little if any improvement but for the Runge-Kutta scheme very worthwhile increases in time step length are possible.

INCLUSION OF A SECOND ORDER TERM IN THE TIME DERIVATIVE (CLASS A)

The simplest method of doing this is to apply the *Crocco method* to make the program second order accurate in time. It should then permit you to use much lower values of artificial viscosity.

The changes of the primary variables in the last timestep are already stored as delro, delrovx, delrovy and delroe and these become delprop in subroutine **"sum_fluxes"** . They are copied into the variable previous. We first calculate the residual for the current timestep in the normal way and store it as store(i,j). We then add on a change proportional to that over the last timestep

```
delprop(i,j) = delprop(i,j)+ facsec*(delprop(i,j)-previous(i,j)),
```

where facsec is a constant that may be read in from data set flow and declared in the header of the main program**.**

When updating the variable delprop(i,j) in subroutine **"sum_fluxes"**, if facsec is zero we have our original scheme.

If facsec = 0.5, we are including the $0.5\ \partial^2 / \partial t^2$ term in the Taylor series for the rate of change in time and this makes the method second order accurate in time. This is the *Adams-Bashforth method*.

If facsec > 0.5 we are adding more of the second derivative than is needed for time accuracy but may get more stability and/or need less smoothing.

Study the effect of this modification on the maximum stable CFL value and on the minimum stable smoothing factor, vary the second order factor facsec.

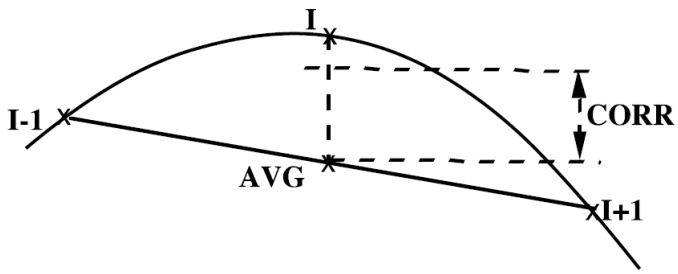
N.B. (a) If the residuals are now second-order accurate, then forming prop_inc in the normal way will make the solution second order accurate.

(b) We are assuming on entry to this routine that delprop contains the first-order accurate residual from the previous timestep. Before leaving **sum_fluxes**, therefore, delprop needs to be restored to this value (which has been stored in store).

DEFERRED CORRECTION FACTORS. (CLASS B)

These can greatly improve the accuracy of the solution by cancelling out most of the artificial viscosity. We need only change subroutine **"smooth"** and the calls to it . If we added on a correction factor to cancel out the smoothing immediately after calculating the correction we would effectively reduce the smoothing factor and would soon lose the stability it gives. However, if the correction factor is added on very gradually so that on every time step we only take a small part of the new correction and a large part of the old one, we can maintain stability whilst greatly reducing the effective smoothing.

What we are doing is shown schematically below



In subroutine **"smooth"** , having calculated avg as before, define a new variable corrnew using

corrnew = fcorr*(prop(i,j) – avg)

where fcorr is the proportion of the artificial viscosity that we want to cancel, typically about 0.9. Set fcorr to your chosen value either within subroutine **"smooth"** or by reading it from data set **flow**. In the latter case it needs to be declared in the header to the main program and passed as an argument to subroutine **"smooth"**.

Update the stored correction, corr_prop(i,j) , by only a small proportion of the new value on every time step, e.g.

corr_prop(i,j) = .99* corr_prop(i,j) + 0.01* corrnew

so that when the flow becomes steady, corr_prop(i,j) = corrnew .

Then smooth the variable prop(i,j) not towards avg but towards

avg + corr_prop(i,j)

and set the result to store(i,j) as before.

The array corr_prop needs to be included in the argument list of subroutine **"smooth"** and also in the dimension statement of that subroutine. In the main program corr_prop becomes corr_ro, corr_rovx, corr_rovy, and corr_roe,. These 4 arrays should already be declared in the header for the main program.

You should now be able to run with a high value of **"smooth_fac"** and so get rapid convergence, but with the errors due to artificial viscosity greatly reduced. Check this by whatever measures of accuracy you can think of. Vary the factor fcorr to see how large it can be. Also consider the effect on convergence.

HIGHER ORDER SMOOTHING. (CLASS B)

The smoothing that we have used is called second order smoothing because the error it introduces is proportional to the second derivatives of the variables. It is possible to reduce the effects of smoothing by using a smoothing function that introduces only higher order errors. Fourth order smoothing is widely used because it involves a symmetrical curve fit. This is stable on its own in the bulk of the flow but it is usually necessary to retain a small amount of second order smoothing to control shock waves.

To use fourth order smoothing, assume that the grid points are evenly spaced and fit a cubic through the values at i-2, i-1, i+1, i+2 (omitting the value at i). Evaluate this cubic at the central i grid point and smooth the calculated value at grid point i towards this value instead of towards the average of the two surrounding points as in the second order smoothing.

To combine this with some second order smoothing we can use:

$$prop_{smoothed} = (1- sf2 - sf4)*prop_{unsmoothed} + sf2*prop_{avg} + sf4*prop_{cubic}$$

Where $prop_{avg}$ is the value we get from second order smoothing and $prop_{cubic}$ is the value obtained from the cubic curve fit. sf2 is the smoothing factor for the 2nd order smoothing and sf4 is that for the 4th order smoothing. Typically sf2 is about 1/4 of sf4.

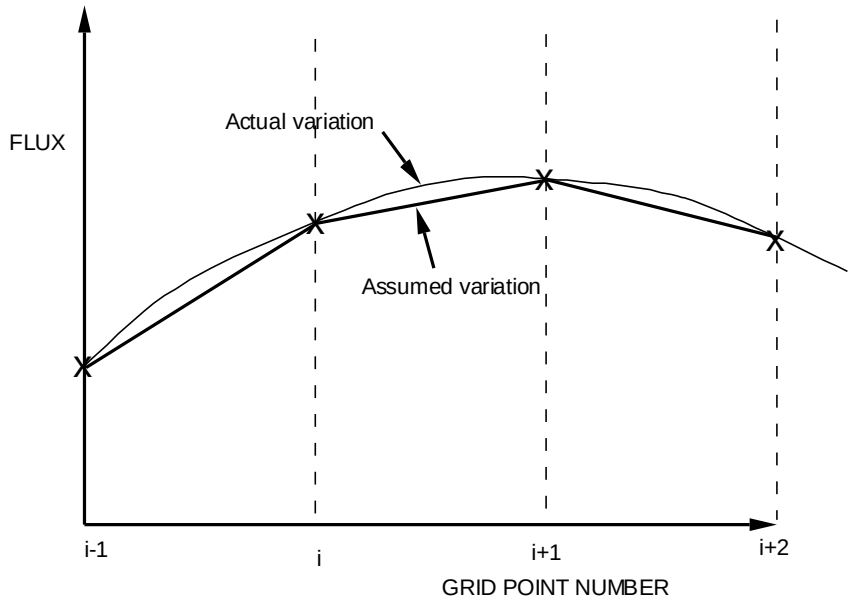
A further refinement is to vary the smoothing factors with the second derivatives of the flow variables so that high smoothing is only applied when the second derivatives are large. This is called adaptive smoothing. This is easily applied in subroutine **smooth** because we can treat $(prop - prop_{avg})$ as being proportional to the local second derivatives. When using this it is necessary to non-dimensionalise the value of $(prop - prop_{avg})$ by suitable reference values. The latter must be a typical change in the variable , not its absolute value. Suitable reference values are already defined as ref_ro, ref_rovx, etc. If these are transferred to subroutine **smooth** and then called prop_ref then we can define a local smoothing factor as

```
sfac_loc = smooth_fac*abs( prop(i,j) – avg) / prop_ref
```

and use this to smooth the variable prop(i,j) in the usual way.

HIGHER ORDER DIFFERENCING. (CLASS B)

Once we are able to run with much lower values of artificial viscosity (smoothing), then the dominant source of numerical error becomes the assumption that the flow properties vary linearly between grid points. This can be improved upon by using a higher order curve fit to estimate the variation between grid points and integrating it to obtain the average flux through the face.



We need to estimate the area between the curve and the lines on the above sketch and use that as the flux rather than simply averaging the values at adjacent points. This can be done by fitting a cubic through the fluxes at points i-1, i, i+1 and i+2. We can assume uniform grid spacing to simplify the algebra. The cubic can then be integrated between limits i and i+1 to find the average flux crossing the face between points i and i+1. Special treatment is needed for the end points and here it is simplest to fit parabolas through the first and last 3 points and to integrate the parabola over the faces of the first and last cells.

To implement this it is best to first store the terms (i.e rovx, rovy, p, h0) that must be integrated over the cell faces as temporary 1-dimensional arrays. Then call a subroutine which fits the cubic/parabolas to these arrays, integrates them over the cell face and returns the integrated value to the main program. These integrated values are then multipled by the projected areas of the cell faces, as usual, in **set_fluxes**.

Compare the accuracy of this approach with that of the original for the same values of smoothing, CFL, etc.

RUNGE-KUTTA SCHEME. (CLASS A)

This is one of the most popular methods currently in use, there are many possible variants of it. We will try a 4 step method in which the main time step is broken down into 4 sub-steps. The changes in the primary variables calculated in each sub-step are added onto the values of the primary variables at the start of the overall step (**NOT** onto the values from the last sub-step) and the new values are used to find the changes in the next sub-step. The time steps used for the successive sub-steps are 1/4, 1/3, 1/2 and 1.0 of the main time step. At the end of the 4th sub-step the values are stored as the new values at the start of the next main time step. The actual equations are given in the course notes.

There is a wide range of possibilities for applying the smoothing, applying the boundary conditions, etc. One approach that works is to follow the steps:

1. do nstep =1,nsteps

2. ro_start = ro, etc**.**

3. Start a loop:

```
do  nrkut = 1,4
    frkut =
    call apply_bconds
    call set_others
    call set_fluxes
    call sum_fluxes(……..., frkut)
    ro = ro_start + ro_inc,  etc
```

4. Call **smooth** as usual (Try)

```
end do
```

5. CONTINUE AS IN ORIGINAL PROGRAM

6. end do

Subroutine **sum_fluxes** needs to be modified to include the factor frkut to modify the timestep.

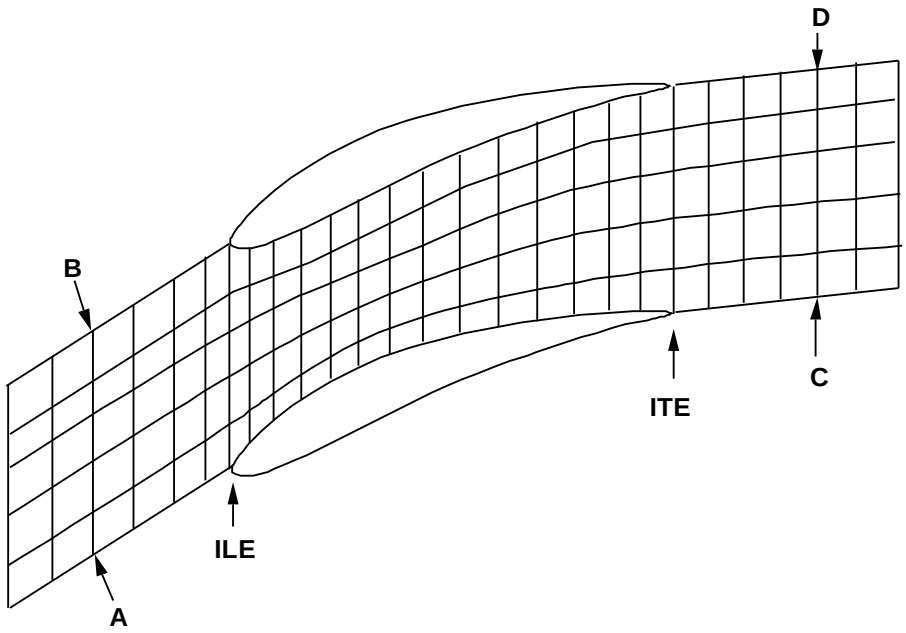
If you succeed in getting the *Runge-Kutta scheme* working you should find that it allows much larger time steps but takes significantly more time per step. Is the extra complexity worth while in terms of CPU time saving? You should also find that it works with low values of artificial viscosity and so gives very low numerical errors, but this may delay convergence. An additional refinement would be to include *deferred corrections* into the scheme. Your scheme should now be as good as many practical codes.

If you are really interested you can code the above as a variable step scheme and compare 2, 3, 4, 5, --step *Runge-Kutta methods*. The factor frkut is always set by

frkut = 1/(1 + nrkut_max – nrkut) .

PERIODIC CASCADE FLOW (DO ONLY IF INTERESTED)

Those of you doing a turbomachinery based project may like to extend your scheme to calculate cascades of blades instead of a single duct. To do this we must apply a new boundary condition that, upstream and downstream of the blades, the values of all variables are the same at points one blade pitch apart in the y direction. This is called the "periodicity condition". To apply it we must use a grid so that the " i = constant " grid lines are lines of constant x, i.e. they are perpendicular to the x axis and parallel to the y axis. This is called an H grid as illustrated below



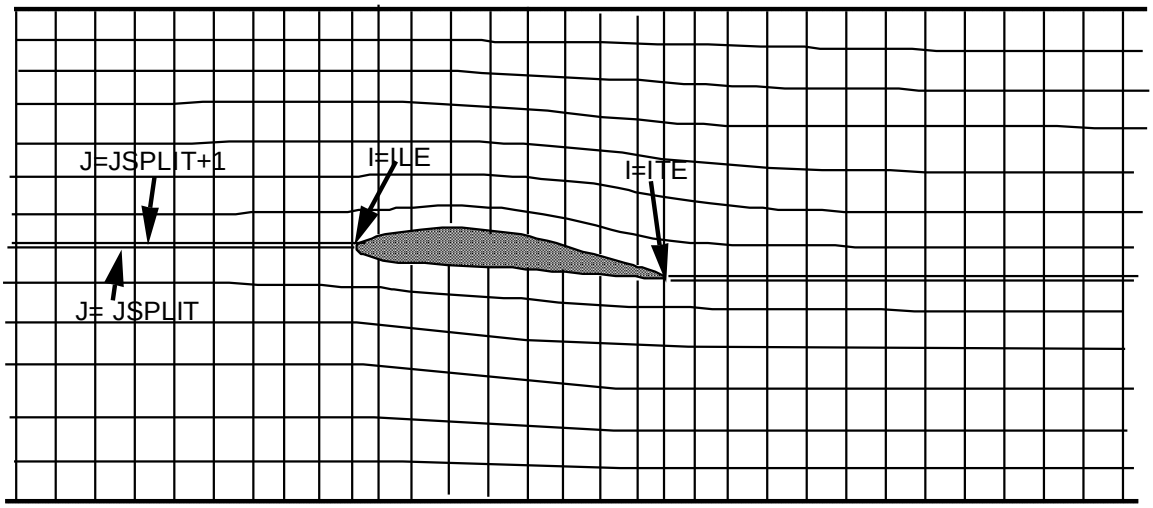
The "j = constant" grid lines are quasi-streamlines, the j= 1 and j= nj lines coincide with the blade surfaces within the blade passage but need only be aligned approximately with the flow upstream and downstream of the blades. However, upstream and downstream of the blades the j=1 and j = nj lines must be exactly one blade pitch apart in the y direction.

To apply the periodicity condition we firstly set the mass fluxes through the faces j=1 and j=nj to zero (in **set_fluxes**) only from i = ile to i = ite-1, then we simply average values of all primary variables on the periodic boundaries. i.e. from i=1 up to and including i = ile we set prop at point A = prop at point B = average of the calculated values at A and B. Similarly downstream of the blades from i= ite to i=ni we set prop P at point C = prop at point D = average of the calculated values at C and D. This must be done for all the primary variables and after every time step. It is most conveniently done is subroutine **"smooth"** after the primary variables have been smoothed.

When calculating a cascade it is necessary to have the "i = constant" grid lines very closely spaced around the leading edge where the velocity gradients are likely to be extreme. It is also necessary to use a fine grid in the j direction to avoid numerical errors, this can be done by making the "i lines" non-uniformly spaced in the j direction. If the trailing edge is sharp it is not necessary to explicitly apply the kutta condition because the numerical viscosity of the solution acts like physical viscosity and makes the flow leave the trailing edge smoothly.

AEROFOIL IN A WIND TUNNEL. (DO ONLY IF INTERESTED)

This is a little more tricky than the cascade extension. We divide the channel into two parts from j =1 to j = jsplit and from j = jsplit + 1 to j = nj as shown in the sketch below. The lines j = jsplit and j = split + 1 are coincident off the aerofoil, but they separate to form the upper and lower surfaces of the aerofoil between i = ile and i = ite . The cells between j = jsplit and j = jsplit+1 are dummy cells that will not be used in the calculation.



We must generate the grid first from j=1 to j = jsplit and then from j = jsplit+1 to j = nj , ensuring that the gap between the lines jsplit and jsplit +1 is the aerofoil thickness when on the aerofoil and is zero off the aerofoil. The exact position of the jsplit and jsplit+1 lines is not important upstream and downstream of the aerofoil. It is advantageous to make the spacing of the lines in the j direction closer as we approach the aerofoil in order to resolve the steep gradients near its surface, the lines can be widely spaced near the tunnel walls. It is also very desirable to make the spacing of the "i = constant" lines small near the leading edge.

We evaluate the areas of the cells and lengths of the cell faces in subroutine **generate_grid**, exactly as before but we must discount the zero length of the j faces of the dummy cells between j=jsplit and j= jsplit+1 when evaluating the minimum length , DMIN, on which to base the timestep.

It is most convenient to use spatially varied time steps and so evaluate step(i,j) for every cell, then set step(i,jsplit) = zero for the dummy cells so that the changes calculated for those cells are zero. At the same time set the areas of the dummy cells to a non-zero value, say = 1.0, to avoid the possibility of trying to divide by zero.

Subroutine **set_fluxes** needs changing to set the fluxes on the aerofoil surfaces to zero between ile and ite-1, we evaluate the fluxes for the dummy cells, even though they will not be used, because it is simpler and more economical to do so than to have a lot of "if" statements within the do loops.

Within subroutine **sum_fluxes** treat the j = jsplit and j = jsplit+1 lines the same as j = 1 and j = nj, i.e. send extra changes to them. Since we have set step(i,j)) = 0 for the dummy cells the change calculated for them will be zero. Similarly within subroutine **smooth** use the one sided averages for the smoothing at j=jsplit and j=jsplit+1 exactly as at j=1 and j=nj. So far we are effectively we are treating the flow from j = 1 to jsplit and from j = jsplit+1 to nj as two independent passages.

At the end of subroutine **smooth** we must set the values of the primary variables at (i,jsplit) and (i,jsplit+1) both equal to the average of the calculated values for points off the aerofoil. Do this for all the points between (and including) i = 1 and i = ile and between (and including) i = ite and i = ni. Do not average the points on the aerofoil. This ensures that the points j = jsplit and j = jsplit+1, which are effectively the same point off the aerofoil, have exactly the same values of all flow properties.

This should be all that is needed to calculate the aerofoil. The plotting program **"eulplt"** should still work but it will try to draw contours within the aerofoil because it does not know that this is not part of the flow. It can easily be adapted to avoid this if required.

UNSTEADY PROBLEMS

If you look back at the extension INCLUSION OF A SECOND ORDER TERM IN THE TIME DERIVATIVE, you will see that taking facsec = 0.5 makes the scheme second order accurate in time and space. You may wish to try an unsteady problem. This will involve making the boundary conditions in some way unsteady (perhaps varying sinusoidally about a mean value). You will loose the concept of convergence to a steady state; if the problem is unsteady, then the solution will be unsteady and convergence then means that the solution is settling down to something that is periodic with the periodicity of the boundary conditions. This is a good vehicle for exploring “non-reflectin”g boundary conditions which you can look up as a topic of active current research.

.

A variation of this might be to move the boundaries with steady inlet/exit conditions, for example tackle the “starting” problem of a supersonic wind tunnel.