A Comparison of Numerical Schemes on Triangular and Quadrilateral Meshes

by

Dana Rae Lindquist

B.S. Cornell University (1986)

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

Master of Science
in
Aeronautics and Astronautics
at the
Massachusetts Institute of Technology

May 1988

Signature of Author _____
Department of Aeronautics and Astronautics
May 6, 1988

Certified by _____
Professor Michael B. Giles
Thesis Supervisor, Department of Aeronautics and Astronautics

Accepted by _____
Professor Harold Y. Wachman
Chairman, Department Graduate Committee

A Comparison of Numerical Schemes on Triangular and Quadrilateral Meshes

by

Dana Rae Lindquist

Submitted to the Department of Aeronautics and Astronautics

on May 6, 1988

in partial fulfillment of the requirements for the degree of

Master of Science in Aeronautics and Astronautics

Finite volume node based Jameson and Ni schemes are compared on both triangular and quadrilateral meshes. The triangular Ni scheme is developed as an extension to the quadrilateral Ni scheme which is described as well as the triangular and quadrilateral Jameson schemes. Two different conservative freestream numerical smoothing methods are used, one of which is high-accuracy and the other low-accuracy. An extensive numerical accuracy study is performed to investigate the accuracy of these schemes and the effect of the different numerical smoothing methods. When the high-accuracy numerical smoothing is used, the accuracy study shows that all the schemes are indeed second order accurate. The low-accuracy smoothing reduces the accuracy of the schemes as well as producing less smooth solutions on irregular meshes. Both triangular and quadrilateral schemes can be used for spatial adaptation, but triangular meshes are more easily fit around complex geometries and do not require modifications in the flow solver.

Thesis Supervisor: Michael B. Giles,

Assistant Professor of Aeronautics and Astronautics

Acknowledgments

I owe a great deal of thanks to my parents Lois and Norman for the support they have always offered and the love that never wavered. They have always believed I could do anything I wanted to do. Their belief in me has always been a motivating force behind my achievements. I would also like to thank my sister Cristina. These people are more than my family, they are my friends.

While my parents had a large influence in the earlier part of my life, later on I met my best friend, Chris. Thanks to him I was able to retain some reality in my life while I worked on my academics. Thank you Chris for believing in me and making me laugh.

To my advisor Michael Giles, many thanks to you also, you were never too busy to answer my questions or help look for bugs in my code. I hope we can continue to work well together in the years to come.

Finally I'd like to thank the members of the CFD Lab. To the students and staff who have been here for a while and are always there to donate their wisdom and experience which only comes with time. To the students who are just starting out for reminding me how much I have learned since I started here.

This work was supported by Rolls-Royce PLC.

Contents

List of Figures

Nomenclature

a	speed of sound
A	area
E	total energy
F, G	flux vectors in x and y directions
H	total enthalpy
M	Mach number
p	pressure
s	entropy
t	time
U	state vector at nodes
U_p	primitive state vector at node
u	velocity in the x direction
v	velocity in the y direction
w	velocity magnitude
α	flow angle
γ	specific heat ratio
ρ	density
ϕ	characteristic variable

Chapter 1
Introduction

Over the last decade more and more interest has been taken in using Computational Fluid Dynamics (CFD) as a design and research tool. A productive tool must be affordable as well as accurate. Since CFD methods have become an important tool in aerodynamic design, it is important that we understand the numerical methods being used by knowing their strong points and limitations. The goal of this work is to compare the use of finite volume methods which solve the Euler equations on quadrilateral and triangular meshes to better understand them.

1.1 Background

The use of numerical solution procedures with quadrilateral meshes has been extensively studied as well as procedures such as multi-grid and spatial adaptation which reduce the computer time required to compute a solution. Multi-grid is a technique which is limited to steady state calculations in which iterations in the solution procedure are successively computed on several meshes of different node densities, and flow changes are interpolated from coarser meshes to finer meshes. In this manner flow changes are greatly accelerated and so the computational time required is on the same order as for the coarsest mesh and the accuracy is that of the finest mesh. For quadrilaterals, the finer meshes are typically found by dividing a coarse cell into four finer cells. The first multi-grid method for the Euler equations on a quadrilateral mesh was developed by Ni [15] in 1981. In 1983 another method was developed by Jameson [9]. Methods which spatially adapt automatically during the solution procedure have also been developed. These methods place small cells where the physical characteristic length is small, such as in shocks, and large cells where the physical characteristic length

is large. Dannenhoffer and Baron [2] discuss the details by which a cell can be chosen for adaptation. By using adaptation, the computational time is reduced, since to achieve a desired refinement in one portion of a mesh, a globally refined mesh is not required. The method of Dannenhoffer and Baron combines both multi-grid and adaptation to obtain the benefits of both methods.

To use CFD methods to compute the flow around complex geometries, several methods have been tried. The difficult task is placing a mesh around the geometry. One approach is to divide the flow field into several coarse blocks and a finer mesh is created in each block. This method requires a large amount of interaction between the user and the mesh generation program, but is widely used today. Another approach uses meshes which surround different parts of a geometry and overlap to form a completely meshed region. The complexity of this approach is in the need to interpolate between each mesh. A third approach involves using unstructured triangles to completely mesh the region. One particular method to create this mesh is based on Delaunay [4] triangulation which allows the mesh generation process to be automatic. Baker [1] describes the generation of a tetrahedral mesh about an entire aircraft using this method.

Interest in the third kind of mesh has prompted research in the use of triangular meshes to solve the Euler and Navier-Stokes equations. In 1985 Löhner, Morgan, Peraire and Zienkiewicz [12] presented a finite element procedure for solving the Navier-Stokes equations on a triangular mesh. A flow solver was developed by Jameson and Mavriplis [14] which solves the Euler equations for a mesh composed of triangles and a similar method was developed by Jameson, Baker and Weatherill [10] for use with tetrahedra. As with quadrilateral meshes, multi-grid and spatial adaptation methods have been developed for use with triangular meshes. A triangular multi-grid method was developed by Mavriplis and Jameson [14] which uses completely unrelated grids at each level and interpolates the solution for transfer from one mesh to another. Several spatial adaptation procedures have been developed to refine the mesh where the physical characteristic length is small. Unstructured triangular meshes may be refined without creating interfaces between regions or coarse and fine cells. Substantial work has been done by Holmes, Lamson and Connell [8] using the Delaunay triangulation method to

add new points. Another method of adaptation has been developed by Peraire, Vahdeti, Morgan and Zienkiewicz [16] which completely remeshes the region putting finer cells where the physical characteristic length is small.

Controversy exists in the CFD community as to the relative benefits of quadrilateral and triangular meshes. In particular a recent paper by Roe [19] proved that the local truncation error is only first order and he conjectured that this implies that the solution is only first order accurate. This contrasts with the demonstrated second order accuracy of finite volume quadrilateral schemes. Another paper by Giles [5] argues that on the contrary these triangular schemes can be globally second order accurate. By being more widely used for a longer period of time, quadrilateral schemes are better understood and accepted, therefore the important part of the controversy is focused on how good triangular schemes are.

The goal of this study is to address this question of how well a given computational method can perform on a triangular mesh as compared to the more commonly used quadrilateral meshes. In particular, two schemes will be examined: the node-based quadrilateral cell Jameson scheme which has been modified for triangular meshes by Mavriplis and Jameson [14], and the quadrilateral cell Ni scheme [15] which has been modified here for use on triangular meshes. Care has been taken to keep the triangular and quadrilateral versions of a scheme similar to provide a fair basis for comparison.

1.2 Overview

First a description of the numerical schemes which solve the Euler equations examined is given in Chapter 2. Next the numerical smoothing required to make these schemes stable and suppress unwanted spurious modes as well as capture shocks is described in Chapter 3. Chapter 4 presents the meshes and the pointer system which is used to describe them. These chapters complete the description of the theory required to write a computer program to implement the schemes described in Chapter 2. Flows computed with these schemes are described in Chapter 5. In Chapter 6 mathematical

and numerical methods of determining the accuracy of these schemes are described and numerical results are given. Finally a short discussion on spatial adaptation is given in Chapter 7 and some conclusions of this study are given in Chapter 8.

Chapter 2
Numerical Schemes

In this chapter the governing equations for an inviscid gas, known as the Euler equations, are presented along with two methods for numerically solving these equations. Both methods are node based, finite volume numerical schemes. The first method is a Lax-Wendroff scheme which was originally developed for quadrilateral cell meshes by Ni [15] and further expanded upon by Giles [7]. This scheme will be referred to as a "Ni Scheme" in the future. A triangular cell mesh extension of the Ni scheme which was developed by the author will also be discussed. The second method is a four step scheme which was developed by Jameson [11] for quadrilateral cell meshes and extended for triangular cell meshes by Mavriplis and Jameson [14]. This scheme will be referred to as a "Jameson Scheme".

2.1 Governing Equations

The flows examined here are steady. To reach a steady state, unsteady equations will be integrated in time from some initial condition until there is no change in the state of the flow field.

For flows with sufficiently large Reynolds numbers the effect of viscosity is confined to a thin region near solid walls where a boundary layer exists. The governing equations used in this study neglect the viscous terms and the heat transfer terms from the full Navier-Stokes equations and are referred to as the Euler equations. These equations are first-order hyperbolic partial differential equations which can be written as

$$\frac{\partial U}{\partial t} = -\left(\frac{\partial F}{\partial x} + \frac{\partial G}{\partial y}\right) \tag{2.1}$$

where U is a state vector of dependent variables and F and G are flux vectors in the x and y directions respectively. F and G are functions of the state vector U.

$$U = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho E \end{bmatrix}, \quad F = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ \rho uH \end{bmatrix}, \quad G = \begin{bmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ \rho vH \end{bmatrix} \tag{2.2}$$

The pressure p and enthalpy H can be written in terms of elements of the state vector U and the specific heat ratio γ, which is assumed to be constant.

$$p = (\gamma - 1)\rho\left(E - \frac{1}{2}(u^2 + v^2)\right) \tag{2.3}$$

$$H = E + \frac{p}{\rho} \tag{2.4}$$

The differential Equation (2.1) is written in conservation law form since the coefficients of the derivative terms are constant. Equations which can be written in conservation law form are well suited for finite volume numerical methods. The control volume used for these methods can be composed of quadrilaterals or triangles.

The flow variables are non-dimensionalized using the upstream stagnation density and stagnation speed of sound. This non-dimensionalization does not change the governing equations and gives the following stagnation quantities.

$$H = \frac{1}{\gamma - 1}, \quad \rho_o = 1, \quad p_o = \frac{1}{\gamma} \tag{2.5}$$

2.2 Quadrilateral Ni Scheme

Given values for the state vector U at time t for all nodes in the domain, it is desired to know what the state vector will be at time $t + \Delta t$. A second order Taylor series expansion for $U_{t+\Delta t}$ at a node is taken about the solution at time t where the subscripts represent the time where the function is evaluated.

$$\begin{aligned} U_{t+\Delta t} &= U_t + \Delta t\left(\frac{\partial U}{\partial t}\right)_t + \frac{1}{2}\Delta t^2\left(\frac{\partial^2 U}{\partial t^2}\right)_t \\ &= U_t - \Delta t\left(\frac{\partial F}{\partial x} + \frac{\partial G}{\partial y}\right)_t - \frac{1}{2}\Delta t\left(\frac{\partial}{\partial x}\Delta F + \frac{\partial}{\partial y}\Delta G\right)_t \\ &= U_t - \frac{\Delta t}{A}\left(\oint (F dy - G dx) + \frac{1}{2}\oint (\Delta F dy - \Delta G dx)\right)_t \end{aligned} \tag{2.6}$$

where

$$\begin{aligned} \Delta F &= \Delta t\left(\frac{\partial F}{\partial t}\right)_t \\ \Delta G &= \Delta t\left(\frac{\partial G}{\partial t}\right)_t \end{aligned} \tag{2.7}$$

Equation (2.6) is developed by substituting in the differential equation form of the governing Equation (2.1) and using Green's Theorem. The integral in Equation (2.6) is taken about a control volume around the node. This control volume is formed by connecting the centroids of the four cells surrounding the node to the midpoints of their shared faces. The control volume about node 1 is denoted by the dashed lines in Figure 2.1.

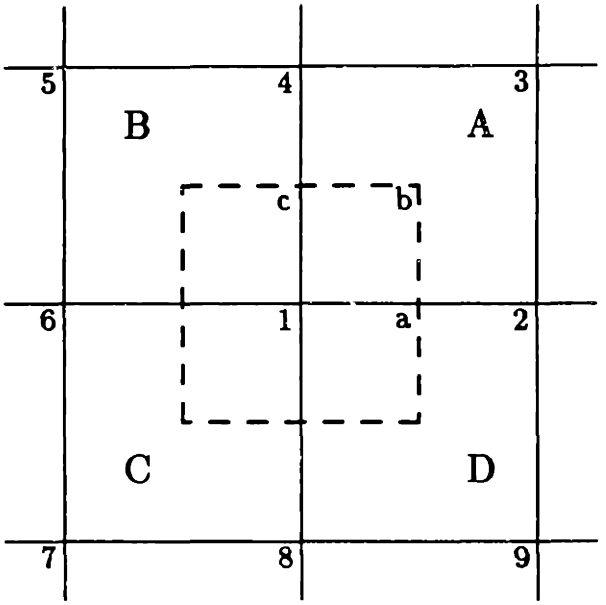


Figure 2.1: Quadrilateral cells surrounding node 1 with control volume for the Ni scheme

Equation (2.6) can be rearranged to define a change in the state vector U in time Δt.

$$\begin{aligned} \delta U &= U_{t+\Delta t} - U_t \\ &= \frac{\Delta t}{A}\left(\oint (F dy - G dx) + \frac{1}{2}\oint (\Delta F dy - \Delta G dx)\right)_t \end{aligned} \tag{2.8}$$

The first term in Equation (2.8) is the flux through the control volume and will be found by calculating the flux into each cell and distributing one quarter of this flux to each node which belongs to the cell. ΔF and ΔG will also be calculated for each cell. The second term in Equation (2.8) will be found by integrating around the control volume using the values of ΔF and ΔG for the cell which the boundary of the control volume passes through.

The change in the state vector at node 1 includes contributions from cells A, B, C and D given by

$$\delta U_1 = \delta U_{1A} + \delta U_{1B} + \delta U_{1C} + \delta U_{1D} \tag{2.9}$$

All four contributions are calculated in a similar manner.

δU_{1A}, the contribution of cell A to node 1, as shown in Figure 2.1 is given by

$$\begin{aligned}
\delta U_{1A} &= (U_{t+\Delta t} - U_t)_{1A} \\
&= \left(\frac{\Delta t}{A}\right)_1 \left[-\frac{1}{4} \oint_{cellA} (F dy - G dx) - \frac{1}{2} \int_{abc} (\Delta F dy - \Delta G dx) \right]_t \\
&= \left(\frac{\Delta t}{A}\right)_1 \left[\frac{1}{4} \left(\frac{A}{\Delta t}\right)_A \Delta U_A - \frac{1}{4} \Big(\Delta F(y_4 - y_2) - \Delta G(x_4 - x_2) \Big) \right]_t
\end{aligned} \tag{2.10}$$

where ΔU_A is found by simple trapezoidal integration around cell A.

$$\begin{aligned}
\Delta U_A &= \frac{1}{2} \left(\frac{\Delta t}{A}\right)_A \Big[\; -(F_2 + F_1)(y_2 - y_1) + (G_2 + G_1)(x_2 - x_1) \\
&\qquad -(F_3 + F_2)(y_3 - y_2) + (G_3 + G_2)(x_3 - x_2) \\
&\qquad -(F_4 + F_3)(y_4 - y_3) + (G_4 + G_3)(x_4 - x_3) \\
&\qquad -(F_1 + F_4)(y_1 - y_4) + (G_1 + G_4)(x_1 - x_4) \Big] \\
&= \frac{1}{2} \left(\frac{\Delta t}{A}\right)_A \Big[\quad (F_3 - F_1)(y_2 - y_4) - (G_3 - G_1)(x_2 - x_4) \\
&\qquad +(F_4 - F_2)(y_3 - y_1) - (G_4 - G_2)(x_3 - x_1) \Big]
\end{aligned} \tag{2.11}$$

The terms $(\frac{\Delta t}{A})_A$ and $(\frac{\Delta t}{A})_1$ are the timestep divided by the area for cell A and node 1 respectively. The calculation of these terms will be described in Section 2.8.

Using the chain rule ΔF and ΔG evaluated at time t in Equation (2.7) can be rewritten as

$$\Delta F = \Delta t \left(\frac{\partial F}{\partial t}\right)_t = \left(\frac{\partial F}{\partial U}\right)_t \Delta U_t$$

$$\Delta G = \Delta t\left(\frac{\partial G}{\partial t}\right)_t = \left(\frac{\partial G}{\partial U}\right)_t \Delta U_t \tag{2.12}$$

where ΔU was defined in Equation (2.11). $(\frac{\partial F}{\partial U})_t$ and $(\frac{\partial G}{\partial U})_t$ are Jacobians of the flux vectors F and G evaluated at time t. For the Euler equations U, F and G are given in Equation (2.2), and for these state and flux vectors ΔF and ΔG are

$$\Delta F_A = \left[\begin{array}{c} (\Delta\rho u) \\ u(\Delta\rho u) + u(\rho\Delta u) + (\Delta p) \\ v(\Delta\rho u) + u(\rho\Delta v) \\ u\Big((\Delta\rho E) + (\Delta p)\Big) + H(\rho\Delta u) \end{array}\right]_A \tag{2.13}$$

$$\Delta G_A = \left[\begin{array}{c} (\Delta\rho v) \\ u(\Delta\rho v) + v(\rho\Delta u) \\ v(\Delta\rho u) + v(\rho\Delta v) + (\Delta p) \\ v\Big((\Delta\rho E) + (\Delta p)\Big) + H(\rho\Delta v) \end{array}\right]_A \tag{2.14}$$

where

$$U_A = \left[\begin{array}{c} \rho \\ \rho u \\ \rho v \\ \rho E \end{array}\right]_A, \quad \Delta U_A = \left[\begin{array}{c} (\Delta\rho) \\ (\Delta\rho u) \\ (\Delta\rho v) \\ (\Delta\rho E) \end{array}\right]_A \tag{2.15}$$

$$\begin{aligned} (\rho\Delta u) &= (\Delta\rho u) - u(\Delta\rho) \\ (\rho\Delta v) &= (\Delta\rho v) - v(\Delta\rho) \\ (\Delta p) &= (\gamma-1)\Big((\Delta\rho E) - u(\Delta\rho u) - v(\Delta\rho v) + \frac{1}{2}(u^2+v^2)(\Delta\rho)\Big) \end{aligned} \tag{2.16}$$

U_A is the value of the state vector at the cell which is the average of the state vector at the four nodes which belong the the cell. ΔU_A is found from Equation (2.11).

2.3 Triangular Ni Scheme

The triangular Ni scheme was developed here as another triangular scheme which can be used for comparison. The main difference between the quadrilateral and triangular Ni schemes is that now the control volume is formed by connecting the centroids of the triangular cells which surround the node to the midpoints of their shared faces. For discussion, let n be the number of cells surrounding a node. The control volume about node 1 where $n = 6$ is denoted by the dashed lines in Figure 2.2.

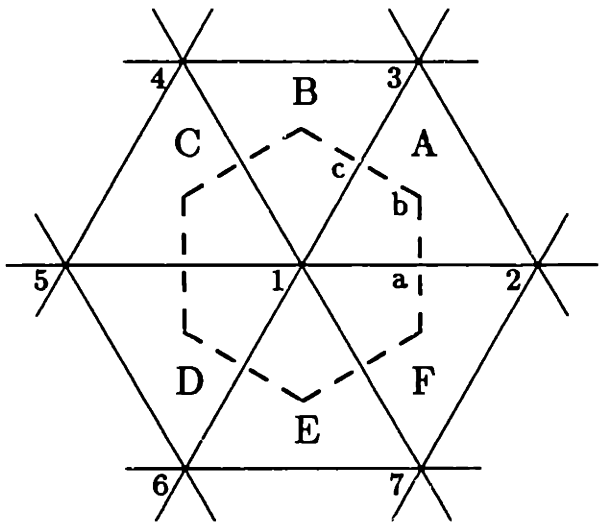


Figure 2.2: Triangular cells surrounding node 1 with control volume for the Ni scheme

The change in the state vector at node 1 includes contributions from all n cells surrounding node 1 and is given by

$$\delta U_1 = \delta U_{1A} + \delta U_{1B} + \delta U_{1C} + \cdots \text{ all } n \text{ cells} \tag{2.17}$$

All n contributions are calculated in a similar manner.

Equation (2.10) extended for the triangular scheme now becomes

$$\begin{aligned}
\delta U_{1A} &= (U_{t+\Delta t} - U_t)_{1A} \\
&= \left(\frac{\Delta t}{A}\right)_1 \left[-\frac{1}{3}\oint_{cellA}(F dy - G dx) - \frac{1}{2}\int_{abc}(\Delta F dy - \Delta G dx) \right]_t \\
&= \left(\frac{\Delta t}{A}\right)_1 \left[\frac{1}{3}\left(\frac{A}{\Delta t}\right)_A \Delta U_A - \frac{1}{4}(\Delta F(y_3 - y_2) - \Delta G(x_3 - x_2)) \right]_t \qquad (2.18)
\end{aligned}$$

where ΔU_A is again found by simple trapezoidal integration around cell A.

$$\begin{aligned}
\Delta U_A &= \frac{1}{2}\left(\frac{\Delta t}{A}\right)_A \Big[-(F_2+F_1)(y_2-y_1) + (G_2+G_1)(x_2-x_1) \\
&\qquad -(F_3+F_2)(y_3-y_2) + (G_3+G_2)(x_3-x_2) \\
&\qquad -(F_1+F_3)(y_1-y_3) + (G_1+G_3)(x_1-x_3) \Big] \\
&= \frac{1}{2}\left(\frac{\Delta t}{A}\right)_A \Big[-F_1(y_2-y_3) + G_1(x_2-x_3) \\
&\qquad -F_2(y_3-y_1) + G_2(x_3-x_1) \\
&\qquad -F_3(y_1-y_2) + G_3(x_1-x_2) \Big] \qquad (2.19)
\end{aligned}$$

The terms $(\frac{\Delta t}{A})_A$ and $(\frac{\Delta t}{A})_1$ are once again the timestep divided by the area for cell A and node 1 respectively. The calculation of these terms will be described in Section 2.8.

For the Euler equations ΔF and ΔG are found in the same manner as for the quadrilaterals from Equation (2.16), except U_A is the average of the state vector at the three nodes which belong the the cell and ΔU_A is found from Equation (2.19).

2.4 Quadrilateral Jameson Scheme

Like the Ni scheme, the value of the state vector, U, is known at time t for all nodes in the domain, and it is desired to know the value of this state vector at time $t + \Delta t$.

A multi-stage time stepping method is used for the Jameson scheme where a first order approximation is used at each step. In particular a four stage time stepping method is used in this thesis.

$$
\begin{aligned}
U_0 &= U_t \\
U_1 &= U_0 + \alpha_1 \Delta t \left(\frac{\partial U}{\partial t} \right)_0 \\
&= U_0 - \alpha_1 \Delta t \left(\frac{\partial F}{\partial x} + \frac{\partial G}{\partial y} \right)_0 \\
&= U_0 - \alpha_1 \frac{\Delta t}{A} \oint (F dy - G dx)_0 \\
&= U_0 - \alpha_1 \frac{\Delta t}{A} (\text{flux})_0 \\
U_2 &= U_0 - \alpha_2 \frac{\Delta t}{A} (\text{flux})_1 \\
U_3 &= U_0 - \alpha_3 \frac{\Delta t}{A} (\text{flux})_2 \\
U_{t+\Delta t} &= U_0 - \alpha_4 \frac{\Delta t}{A} (\text{flux})_3
\end{aligned}
\tag{2.20}
$$

$$
\alpha_1 = \frac{1}{4}, \quad \alpha_2 = \frac{1}{3}, \quad \alpha_3 = \frac{1}{2}, \quad \alpha_4 = 1 \tag{2.21}
$$

where subscripts indicate which state vector is used to calculate the flux. The term $(\frac{\Delta t}{A})$ is the timestep divided by the area evaluated at the node to keep the scheme conservative. The calculation of this term will be described in Section 2.8.

Equation (2.20) is developed in a similar fashion as Equation (2.7) by substituting in the differential form of the governing Equation (2.1) and using Green's theorem. The first order terms in the Jameson scheme are the same as the first order terms in the Ni scheme, but are described differently by considering a different control volume. In fact, in the limit of small timestep the Ni scheme reduces to the Jameson scheme. The integral in Equation (2.20) is taken about a control volume around the node which is formed by all of the four cells around the node, therefore the control volumes in the

Jameson scheme are overlapping. This control volume is shown about node 1 by the dashed lines in Figure 2.3.

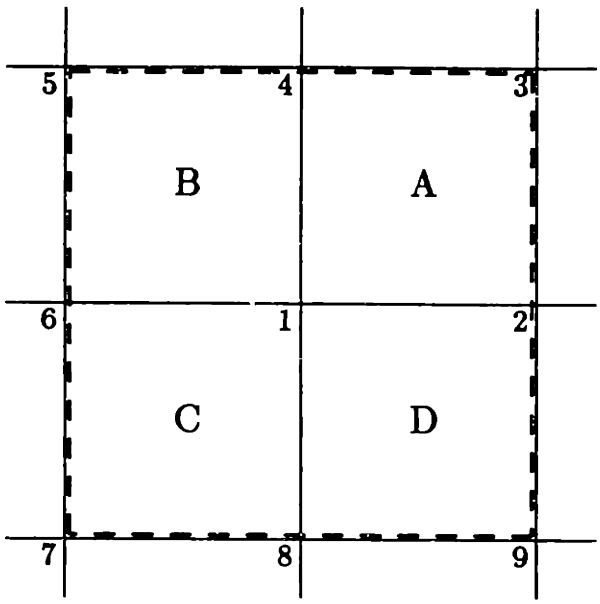


Figure 2.3: Quadrilateral cells surrounding node 1 with control volume for the Jameson scheme

The integral in Equation (2.20) is calculated by finding the contribution of each cell to the flux into the control volume for each node. The contribution from cell A to the flux at nodes 1, 2, 3 and 4 is the same.

$$\text{flux}_{1A} = \text{flux}_{2A} = \text{flux}_{3A} = \text{flux}_{4A} = \text{flux}_A \tag{2.22}$$

where

$$\begin{aligned} \text{flux}_A = \frac{1}{2}\Big[& -(F_2+F_1)(y_2-y_1) + (G_2+G_1)(x_2-x_1) \\ & -(F_3+F_2)(y_3-y_2) + (G_3+G_2)(x_3-x_2) \\ & -(F_4+F_3)(y_4-y_3) + (G_4+G_3)(x_4-x_3) \\ & -(F_1+F_4)(y_1-y_4) + (G_1+G_4)(x_1-x_4) \Big] \end{aligned}$$

$$= \frac{1}{2}\Big[\quad (F_3 - F_1)(y_2 - y_4) - (G_3 - G_1)(x_2 - x_4)$$
$$+(F_4 - F_2)(y_3 - y_1) - (G_4 - G_2)(x_3 - x_1) \Big] \tag{2.23}$$

The flux at node 1 is then the sum of the contributions from the four surrounding cells.

$$\text{flux}_1 = \text{flux}_{1A} + \text{flux}_{1B} + \text{flux}_{1C} + \text{flux}_{1D} \tag{2.24}$$

where flux_{1B}, flux_{1C} and flux_{1D} are calculated in a similar fashion as flux_{1A}.

2.5 Triangular Jameson Scheme

The triangular Jameson scheme is similar to the quadrilateral Jameson scheme except now the cells which surround a node are triangular. For discussion let n be the number of cells surrounding a node. The control volume about node 1 where $n = 6$ is denoted by the dashed lines in Figure 2.4.

The integral in Equation (2.20) is once again calculated by finding the contribution of each cell to the flux into the control volume for each node. The contribution from cell A to the flux at nodes 1, 2 and 3 is the same.

$$\text{flux}_{1A} = \text{flux}_{2A} = \text{flux}_{3A} = \text{flux}_A \tag{2.25}$$

where

$$\begin{aligned} \text{flux}_A \quad = \frac{1}{2}\Big[& -(F_2 + F_1)(y_2 - y_1) + (G_2 + G_1)(x_2 - x_1) \\ & -(F_3 + F_2)(y_3 - y_2) + (G_3 + G_2)(x_3 - x_2) \\ & -(F_1 + F_3)(y_1 - y_3) + (G_1 + G_3)(x_1 - x_3) \Big] \end{aligned}$$

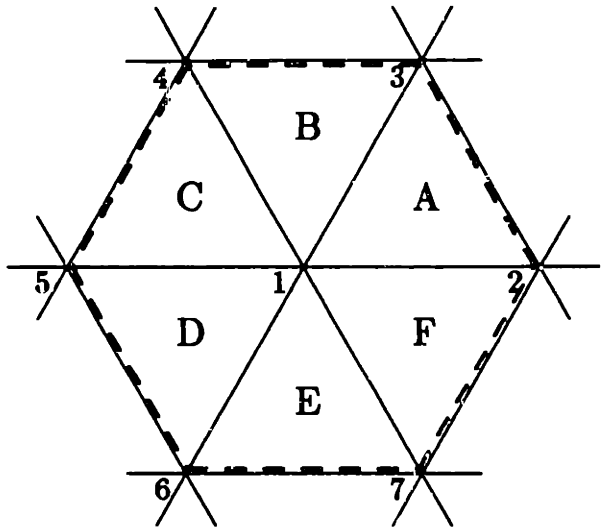


Figure 2.4: Triangular cells surrounding node 1 with control volume for the Jameson scheme

$$= \frac{1}{2}\Big[\; -F_1(y_2-y_3) \; + \; G_1(x_2-x_3)$$
$$-F_2(y_3-y_1) \; + \; G_2(x_3-x_1)$$
$$-F_3(y_1-y_2) \; + \; G_3(x_1-x_2) \Big] \tag{2.26}$$

The flux at node 1 is then the sum of the contributions from the n surrounding cells.

$$\text{flux}_1 \; = \; \text{flux}_{1A} \; + \; \text{flux}_{1B} \; + \; \cdots \; \text{all } n \text{ cells} \tag{2.27}$$

where flux_{1B}, flux_{1C}, ... are calculated in a similar fashion as flux_{1A}.

2.6 Farfield Boundary Conditions

The farfield boundary conditions are applied at the inlet and outlet boundary nodes, and are used for both the Ni and Jameson schemes. The boundary conditions on hyperbolic equations must correctly close the system of equations. Linear characteristic

theory determines the direction of wave motion in and out of the domain, and thus where boundary conditions must be imposed. The following analysis is described by Giles in [7] and a general formulation is presented by Dannenhoffer in [3]. To simplify the process, primitive state vector variables U_p are used where

$$U_p = \begin{bmatrix} \rho \\ u \\ v \\ p \end{bmatrix} \tag{2.28}$$

The first step in the process is to linearize the governing equations where the spatial directions x and y are along the grid lines.

$$\frac{\partial U_p}{\partial t} + \mathbf{A}\frac{\partial U_p}{\partial x} + \mathbf{B}\frac{\partial U_p}{\partial y} = 0 \tag{2.29}$$

$\mathbf{A}$ and $\mathbf{B}$ are constant matrices evaluated at some reference state. The wave propagation normal to the boundary (in the x direction) is dominant, therefore variations parallel to the boundary (in the y direction) will be neglected. Equation (2.29) becomes

$$\frac{\partial U_p}{\partial t} + \mathbf{A}\frac{\partial U_p}{\partial x} = 0 \tag{2.30}$$

The reference state for evaluation of the matrix $\mathbf{A}$ will be the flow on the boundary at the old timestep. To reduce computational effort, the average value of the state vector on the boundary will be used to evaluate $\mathbf{A}$. This state will be denoted by the subscript $(\,)_{old-ave}$. The constant matrix $\mathbf{A}$ is then

$$\mathbf{A} = \begin{bmatrix} u & \rho & 0 & 0 \\ 0 & u & 0 & \frac{1}{\rho} \\ 0 & 0 & u & 0 \\ 0 & \rho a^2 & 0 & u \end{bmatrix}_{old-ave} \tag{2.31}$$

The matrix $\mathbf{A}$ can be diagonalized by a similarity transformation,

$$\mathbf{T}^{-1}\mathbf{A}\mathbf{T} = \begin{bmatrix} u & 0 & 0 & 0 \\ 0 & u & 0 & 0 \\ 0 & 0 & u+a & 0 \\ 0 & 0 & 0 & u-a \end{bmatrix}_{old-ave} = \mathbf{\Lambda} \tag{2.32}$$

where the matrix $\mathbf{T}$ is the matrix of right eigenvectors of $\mathbf{A}$ and the matrix $\mathbf{T}^{-1}$ is the matrix of left eigenvectors of $\mathbf{A}$. Matrix $\mathbf{\Lambda}$ is a diagonal matrix whose elements are the eigenvalues of matrix $\mathbf{A}$.

$$\mathbf{T} = \begin{bmatrix} \frac{-1}{a^2} & 0 & \frac{1}{2a^2} & \frac{1}{2a^2} \\ 0 & 0 & \frac{1}{2\rho a} & \frac{-1}{2\rho a} \\ 0 & \frac{1}{\rho a} & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \end{bmatrix}_{old-ave}, \qquad \mathbf{T}^{-1} = \begin{bmatrix} -a^2 & 0 & 0 & 1 \\ 0 & 0 & \rho a & 0 \\ 0 & \rho a & 0 & 1 \\ 0 & -\rho a & 0 & 1 \end{bmatrix}_{old-ave} \tag{2.33}$$

Multiplication of Equation (2.30) by $\mathbf{T}^{-1}$ produces the equation

$$\frac{\partial \phi}{\partial t} + \mathbf{\Lambda}\frac{\partial \phi}{\partial x} = 0 \tag{2.34}$$

where $\phi = \mathbf{T}^{-1}U_p$. Variation from the values at the old timestep will be considered, therefore $\delta\phi = \mathbf{T}^{-1}\delta U_p$. The four equations in the system of equations (2.34) are now independent. The elements of ϕ are the linearized characteristic variables and the corresponding elements of $\mathbf{\Lambda}$ indicate the direction of the flow of information. For subsonic flow where $0 < u < a$ the first three characteristics give waves propagating downstream since the corresponding elements of $\mathbf{\Lambda}$ are positive and the fourth propagating upstream since the fourth element of $\mathbf{\Lambda}$ is negative. For supersonic flow where $u > a$ all waves propagate downstream since all elements of $\mathbf{\Lambda}$ are positive.

2.6.1 Subsonic Inlet Boundary

For subsonic flow three pieces of information must be specified at the inlet boundary; here they will be the total enthalpy, H, the entropy, s, and the flow angle, α.

$$H = \left(\frac{\gamma}{\gamma-1}\right)\frac{p}{\rho} + \frac{1}{2}(u^2+v^2) \tag{2.35}$$

$$s = \log(\gamma p) - \gamma \log(\rho) \tag{2.36}$$

$$\tan\alpha = \frac{v}{u} \tag{2.37}$$

A fourth piece of information comes from the interior of the flow field and will be taken from the values the numerical solver predicts, therefore denoted by the subscript $(\,)_{pred}$. Let the subscript $(\,)_{spec}$ stand for the value which is specified by the inlet flow conditions. The subscripts $(\,)_{old}$ and $(\,)_{new}$ will stand for the values at the old and new time steps. The amount needed to bring the old values of H, s and $\tan\alpha$ on the boundary to the specified values can be written in terms of a first order Taylor series in ϕ. The constant coefficient of the series can be changed by using the chain rule to contain elements of U_p.

$$\begin{aligned}
(H)_{spec} &= (H)_{new} \\
&= (H)_{old} + \left(\frac{\partial H}{\partial \phi}\right)_{old-ave} \delta\phi \\
&= (H)_{old} + \left(\frac{\partial H}{\partial U_p}\right)_{old-ave} \mathbf{T}\,\delta\phi \\
&= (H)_{old} + \left[\begin{array}{cccc} \frac{1}{(\gamma-1)\rho} & \frac{v}{\rho a} & \frac{a+u}{2\rho a} & \frac{a-u}{2\rho a} \end{array}\right]_{old-ave} \delta\phi
\end{aligned} \tag{2.38}$$

$$(s)_{spec} = (s)_{old} + \left[\begin{array}{cccc} \frac{1}{\rho} & \rho & 0 & 0 \end{array}\right]_{old-ave} \delta\phi \tag{2.39}$$

$$(\tan\alpha)_{spec} = (\tan\alpha)_{old} + \left[\begin{array}{cccc} 0 & \frac{1}{\rho u a} & \frac{-v}{2\rho u^2 a} & \frac{v}{2\rho u^2 a} \end{array}\right]_{old-ave} \delta\phi \tag{2.40}$$

The change in the fourth characteristic $\delta(\phi_4)$ is equal to the change that the flow field predicts, $\delta(\phi_4)_{pred}$. Since $\delta\phi = \mathbf{T}^{-1}\delta U_p$, the predicted change in the characteristic variable $\delta(\phi_4)_{pred}$ is found to be

$$\delta(\phi_4)_{pred} = (-\rho a)_{old}\,\delta u_{pred} + \delta p_{pred} \tag{2.41}$$

There are now four equations for the change in the characteristic variable ϕ which can be written in matrix form.

$$\begin{bmatrix} (H)_{spec} - (H)_{old} \\ s_{spec} - s_{old} \\ (\tan\alpha)_{spec} - (\tan\alpha)_{old} \\ \delta(\phi_4)_{pred} \end{bmatrix} = \begin{bmatrix} \frac{1}{(\gamma-1)\rho} & \frac{u}{\rho a} & \frac{a+u}{2\rho a} & \frac{a-u}{2\rho a} \\ \frac{1}{\rho} & \rho & 0 & 0 \\ 0 & \frac{1}{\rho u a} & \frac{-v}{2\rho u^2 a} & \frac{v}{2\rho u^2 a} \\ 0 & 0 & 0 & 1 \end{bmatrix}_{old-ave} \begin{bmatrix} \delta(\phi_1) \\ \delta(\phi_2) \\ \delta(\phi_3) \\ \delta(\phi_4) \end{bmatrix} \tag{2.42}$$

Using the relation $\delta U_p = \mathbf{T}\delta\phi$, $\delta\phi$ in Equation (2.42) can now be changed back into primitive state vector values U_p.

$$\delta U_p = \begin{bmatrix} \delta(\rho) \\ \delta(u) \\ \delta(v) \\ \delta(p) \end{bmatrix}_{new} =$$

$$\frac{1}{ua+u^2+v^2} \begin{bmatrix} \frac{\rho u}{a} & \frac{-p}{a}\left(\frac{\gamma u}{\gamma-1}+\frac{u^2+v^2}{a}\right) & \frac{-\rho v}{a} & \frac{u^2+v^2}{a^2} \\ u & \frac{-up}{(\gamma-1)\rho} & -v & \frac{-u}{\rho} \\ v & \frac{-vp}{(\gamma-1)\rho} & a+u & \frac{-v}{\rho} \\ \rho a u & \frac{-aup}{\gamma-1} & -\rho a v & u^2+v^2 \end{bmatrix}_{old-ave} \begin{bmatrix} (H)_{spec} - (H)_{old} \\ s_{spec} - s_{old} \\ \tan\alpha_{spec} - \tan\alpha_{old} \\ \delta\phi_{4_{pred}} \end{bmatrix} \tag{2.43}$$

To transform the change in the primitive state vector variables δU_p back into the change in the state vector δU the following transformation is performed

$$\delta U_{new} = \begin{bmatrix} \delta(\rho) \\ \delta(\rho u) \\ \delta(\rho v) \\ \delta(\rho E) \end{bmatrix}_{new} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ u & \rho & 0 & 0 \\ v & \rho & 0 & 0 \\ \frac{1}{2}(u^2+v^2) & \rho u & \rho v & \frac{1}{\gamma-1} \end{bmatrix}_{old} \begin{bmatrix} \delta(\rho) \\ \delta(u) \\ \delta(v) \\ \delta(p) \end{bmatrix}_{new} \tag{2.44}$$

So finally the new value of the state vector at the inlet nodes is

$$U_{new} = U_{old} + \delta U_{new} \tag{2.45}$$

2.6.2 Subsonic Outlet Boundary

For subsonic flow the outlet boundary is similar to the inlet boundary except now only one piece of information must be specified; here static pressure p will be used. Once again the amount needed to bring the old value of p to the specified value on the boundary can be written in terms of a first order Taylor series in ϕ.

$$\begin{aligned} p_{spec} &= p_{new} \\ &= p_{old} + \left(\frac{\partial p}{\partial \phi}\right)_{old-ave} \delta\phi \\ &= p_{old} + \left(\frac{\partial p}{\partial U_p}\right)_{old-ave} \left(\frac{\partial U_p}{\partial \phi}\right)_{old-ave} \delta\phi \\ &= p_{old} + \left[0 \;\; 0 \;\; \frac{1}{2} \;\; \frac{1}{2}\right]_{old-ave} \delta\phi \end{aligned} \tag{2.46}$$

The change in the first, second and third characteristics is equal to the change the flow field predicts. Again, since $\delta\phi = \mathbf{T}^{-1}\delta U_p$, the predicted change in the characteristic variables $\delta(\phi_1)_{pred}$, $\delta(\phi_2)_{pred}$ and $\delta(\phi_3)_{pred}$ are found to be

$$\delta(\phi_1)_{pred} = (-a^2)_{old}\, \delta\rho_{pred} + \delta p_{pred} \tag{2.47}$$

$$\delta(\phi_2)_{pred} = (\rho a)_{old}\, \delta v_{pred} \tag{2.48}$$

$$\delta(\phi_3)_{pred} = (\rho a)_{old}\, \delta u_{pred} + \delta p_{pred} \tag{2.49}$$

As in the subsonic inlet condition, there are now four equations for the change in the characteristic variable ϕ.

$$\begin{bmatrix} \delta(\phi_1)_{pred} \\ \delta(\phi_2)_{pred} \\ \delta(\phi_3)_{pred} \\ p_{spec} - p_{old} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \end{bmatrix} \begin{bmatrix} \delta(\phi_1) \\ \delta(\phi_2) \\ \delta(\phi_3) \\ \delta(\phi_4) \end{bmatrix} \tag{2.50}$$

Using the relation $\delta U_p = \mathbf{T}\delta\phi$, $\delta\phi$ in Equation (2.50) can now be changed back into primitive state vector values U_p.

$$\begin{aligned} \delta U_p &= \begin{bmatrix} \delta(\rho) \\ \delta(u) \\ \delta(v) \\ \delta(p) \end{bmatrix}_{new} \\ &= \begin{bmatrix} \frac{-1}{a^2} & 0 & 0 & \frac{1}{a^2} \\ 0 & 0 & \frac{1}{\rho a} & \frac{-1}{\rho a} \\ 0 & \frac{1}{\rho a} & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}_{old-ave} \begin{bmatrix} \delta\phi_{1pred} \\ \delta\phi_{2pred} \\ \delta\phi_{3pred} \\ p_{spec} - p_{old} \end{bmatrix} \end{aligned} \tag{2.51}$$

The primitive state vector variables, δU_p, are transformed to the state vector, δU, by Equation (2.44) and the update is performed as in Equation (2.45).

2.6.3 Supersonic Inlet Boundary

Since for supersonic flow all waves flow downstream, to implement the inlet boundary condition it is simply necessary to prescribe the flow conditions. The inlet Mach number M_{spec} is specified as well as the flow angle α_{spec}, and the flow conditions are found from these variables.

$$\rho = \left(1 + \frac{(\gamma-1)}{2} M_{spec}^2\right)^{\frac{-1}{\gamma-1}}$$

$$a = \left(1 + \frac{(\gamma-1)}{2} M_{spec}^2\right)^{-\frac{1}{2}}$$

$$w = \sqrt{u^2 + v^2} = M_{spec}a$$

$$p = \frac{\rho a^2}{\gamma}$$

The state vector is fixed at

$$U = \begin{bmatrix} \rho \\ \rho w \cos \alpha_{spec} \\ \rho w \sin \alpha_{spec} \\ \frac{p}{\gamma - 1} + \frac{1}{2} w^2 \rho \end{bmatrix}_{new} \tag{2.52}$$

2.6.4 Supersonic Outlet Boundary

For the supersonic outlet all waves flow out of the boundary, so the change in the state vector predicted by the scheme is used here.

$$\delta U_{new} = \delta U_{pred} \tag{2.53}$$

2.7 Wall Boundary

Two conditions are applied at the solid wall boundaries; first that there is no flux through the wall faces and second that the flow is tangent to the wall at the wall nodes. It would seem that the first condition would be sufficient to satisfy the second condition without explicitly enforcing tangency, but this is not the case since the scheme is node based. The flux on the face is computed by averaging the flux at the nodes, therefore it is possible to have no flux through the walls while the nodal flux oscillates about zero at the wall nodes. By imposing tangency at the nodes, this oscillatory state cannot occur.

2.7.1 Ni Scheme

Since for the Ni scheme only half as many cells surround a boundary node as an interior node, the control volume is half the size of the control volumes in the interior of the flow field. Boundary cells are shown in Figure 2.5 with the boundaries of the control volume denoted by dashed lines. When computing ΔU for the wall boundary cells from Equation (2.11) for quadrilateral cells or Equation (2.19) for triangular cells, the flux through the wall face consists only of the pressure term. The contribution to ΔU from cells A, B (and C in the triangular case) to node 1 in Figure 2.5 is then found in the same manner as for the interior nodes. The second order contribution to node 1 is also the same as for the interior nodes, except that now the integral must also be taken along the wall boundary where ΔF and ΔG consist only of the pressure term. For cell A the second order contribution from ΔF and ΔG along the boundary become

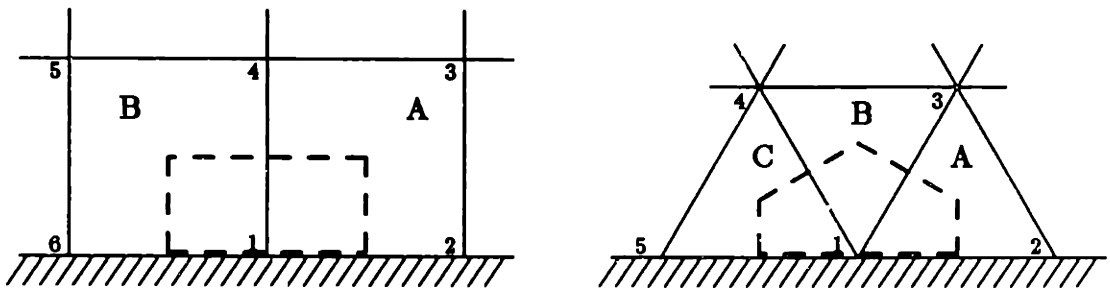


Figure 2.5: Boundary cells for Ni scheme with control volume

$$\Delta F_A = \begin{bmatrix} 0 \\ (\Delta p) \\ 0 \\ 0 \end{bmatrix}_A, \quad \Delta G_A = \begin{bmatrix} 0 \\ 0 \\ (\Delta p) \\ 0 \end{bmatrix}_A \tag{2.54}$$

where as before

$$(\Delta p) = (\gamma-1)\Big((\Delta \rho E) - u(\Delta \rho u) - v(\Delta \rho v) + \frac{1}{2}(u^2+v^2)(\Delta \rho)\Big) \tag{2.55}$$

2.7.2 Jameson Scheme

The control volume for the Jameson scheme boundary nodes is also half the size of the control volumes in the interior of the flow field. The boundary control volume is shown in Figure 2.6. The no flux condition is easier to implement for the Jameson scheme since there are only first order terms. When computing (flux) for the wall boundary cells from Equation (2.23) for quadrilateral cells or Equation (2.26) for triangular cells the flux through the wall face consists only of the pressure terms.

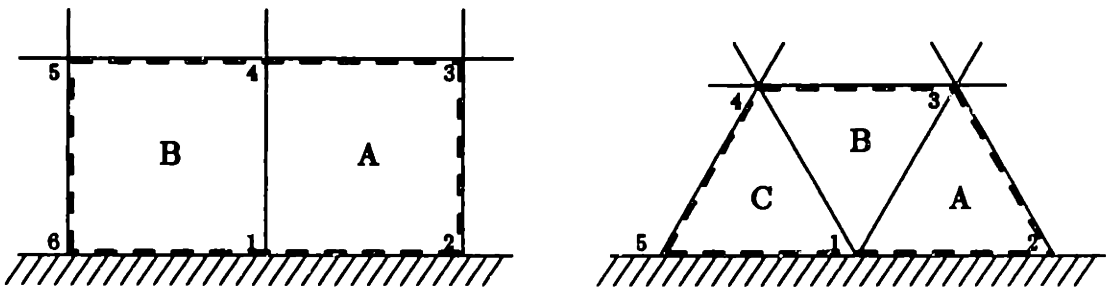


Figure 2.6: Boundary cells for Jameson scheme with control volume

2.7.3 Tangency Condition

The change in the state vector for both schemes is such that the new state vector will satisfy flow tangency on the boundary. At each wall node a flow angle, α, is prescribed.

For both schemes the second and third components of the state vector are changed from the value predicted by the solver, denoted by the subscript $(\,)_{before}$, so the new values become

$$\begin{aligned} (\rho u)_{new} &= (\rho u)_{before} + (\rho w)_n \sin\alpha \\ (\rho v)_{new} &= (\rho v)_{before} - (\rho w)_n \cos\alpha \end{aligned} \tag{2.56}$$

where

$$(\rho w)_n = -(\rho u)_{before} \sin\alpha \;+\; (\rho v)_{before} \cos\alpha \tag{2.57}$$

and for the Ni scheme,

$$(\rho u)_{before} = (\rho u)_{old} + \delta(\rho u)_{old} \tag{2.58}$$

$$(\rho v)_{before} = (\rho v)_{old} + \delta(\rho v)_{old} \tag{2.59}$$

2.8 Timestep

In the description of the basic schemes the timestep divided by the area, $(\frac{\Delta t}{A})$, was referred to for a cell or a node. The term $(\frac{\Delta t}{A})$ is computed on a cell basis and the value at the node is found by summing $(\frac{\Delta t}{A})$ for the cells surrounding the node. A factor of $\frac{1}{4}$ is multiplied by the nodal $(\frac{\Delta t}{A})$ for the quadrilateral Ni scheme and $\frac{1}{3}$ for the triangular Ni scheme since the control volume at a node only includes this fraction of the cells surrounding that node.

For the quadrilateral schemes the maximum stable $(\frac{\Delta t}{A})$ on a cell A is defined by Usab in [22] from a Von Neumann stability analysis for the linearized 2-D Euler equations.

$$\left(\frac{\Delta t}{A}\right)_{max} = CFL \cdot min\left\{ \frac{1}{|u\Delta y^l - v\Delta x^l| + a\Delta l}, \; \frac{1}{|u\Delta y^m - v\Delta x^m| + a\Delta m} \right\} \tag{2.60}$$

The flow variables u, v and a are the average of the nodal values for the cell. For cell A with nodes 1, 2, 3 and 4 running counterclockwise around the cell as shown in Figure 2.1,

$$\begin{aligned} \Delta x^l &= \frac{1}{2}(x_2 + x_3 - x_1 - x_4), \quad & \Delta x^m &= \frac{1}{2}(x_1 + x_2 - x_4 - x_5) \\ \Delta y^l &= \frac{1}{2}(y_2 + y_3 - y_1 - y_4), \quad & \Delta y^m &= \frac{1}{2}(y_1 + y_2 - y_4 - y_5) \\ \Delta l &= \sqrt{(\Delta x^l)^2 + (\Delta y^l)^2}, \quad & \Delta m &= \sqrt{(\Delta x^m)^2 + (\Delta y^m)^2} \end{aligned} \tag{2.61}$$

For the triangular schemes the $(\frac{\Delta t}{A})$ is found by Giles in [6] from energy methods.

$$\left(\frac{\Delta t}{A}\right)_{max} = \frac{CFL \cdot 2}{|u\Delta x^l - v\Delta y^l| + |u\Delta x^m - v\Delta y^m| + |u\Delta x^n - v\Delta y^n| + a(\Delta l + \Delta m + \Delta n)} \quad (2.62)$$

The flow variables u, v and a are again the average of the nodal values for the cell. For cell A with nodes 1, 2 and 3 running counterclockwise around the cell as shown in Figure 2.3

$$\begin{aligned}
\Delta x^l &= \frac{1}{2}(x_2 - x_1), \qquad \Delta y^l = \frac{1}{2}(y_2 - y_1) \\
\Delta x^m &= \frac{1}{2}(x_3 - x_2), \qquad \Delta y^m = \frac{1}{2}(y_3 - y_2) \\
\Delta x^n &= \frac{1}{2}(x_1 - x_3), \qquad \Delta y^n = \frac{1}{2}(y_1 - y_3)
\end{aligned}$$

$$\begin{aligned}
\Delta l &= \sqrt{(\Delta x^l)^2 + (\Delta y^l)^2} \\
\Delta m &= \sqrt{(\Delta x^m)^2 + (\Delta y^m)^2} \\
\Delta n &= \sqrt{(\Delta x^n)^2 + (\Delta y^n)^2}
\end{aligned} \quad (2.63)$$

CFL stands for the Courant, Friedrichs, and Lewy number which gives the timestep limit for stability. In [22] Usab presents a Von Neumann stability analysis for the linearized 2-D Euler equations on the quadrilateral Ni scheme. The stability restriction from this analysis is $CFL \leq \frac{1}{\sqrt{2}}$. Usab then says that this restriction is too strict and that in practice $CFL \leq 1$ is used. This observation was confirmed in this work for both the quadrilateral and triangular Ni schemes. It is possible that the non-linearity in the Euler equations or numerical smoothing cause the increase in CFL limit. Using energy methods, Giles shows in [6] that for the four step method used here for the triangular Jameson scheme the stability limit gives $CFL \leq 2\sqrt{2}$. The method finds the CFL limit for which the energy associated with a solution is non-increasing. It has been shown that for the four step quadrilateral Jameson scheme the stability limit also gives $CFL \leq 2\sqrt{2}$. In practice it was found that this limit is not strict enough and $CFL \leq 2$ was used.

Chapter 3
Numerical Smoothing

Numerical smoothing is a dissipative operator which is added to numerical schemes to damp out oscillations in the solution and provide stability. The Jameson schemes do not have a dissipative term and are unstable without the addition of a dissipative smoothing operator. Numerical smoothing is also required for the Jameson schemes to eliminate steady state, spatially-oscillatory modes which are allowed by the scheme. Three modes are allowed for the quadrilateral Jameson scheme and are shown as modes a, b and c in Figure 3.1. Three modes are also allowed for the triangular Jameson scheme. One is shown in Figure 3.2 and the other two are similar modes shifted by one node. For the Ni schemes there is a dissipative term in the numerical operator and the scheme is stable for a smooth flow field. For the quadrilateral Ni scheme only one oscillatory mode is allowed which is shown as mode a in Figure 3.1. The triangular Ni scheme does not allow any of the oscillatory modes, but in the limit of very small timestep the Ni scheme reduces to the Jameson scheme which, as mentioned, allows three modes. To provide stability and to eliminate oscillatory modes a fourth difference *freestream* smoothing operator is used. For both schemes numerical smoothing is required to capture discontinuities such as shocks. The smoothing required to capture shocks will be referred to as *shock* smoothing.

First, the different second difference operators used here will be described. These operators are used to formulate the smoothing operators. Next the different methods of freestream and shock smoothing will be described.

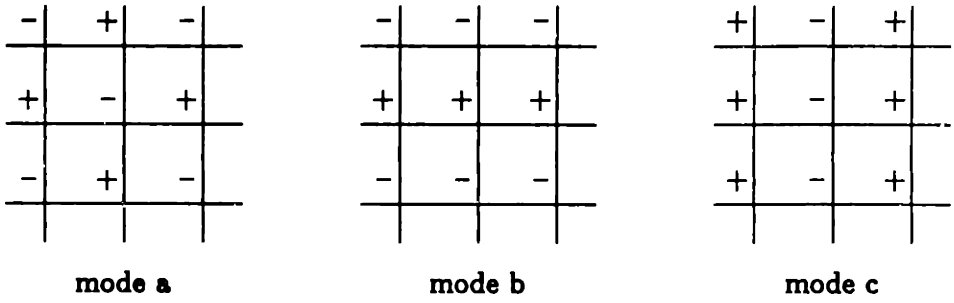


Figure 3.1: Possible modes for quadrilateral cells

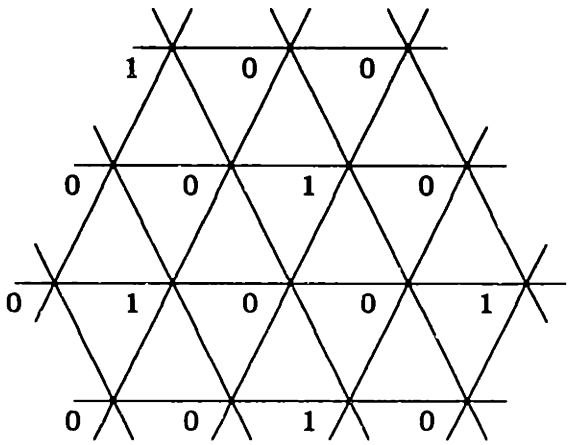


Figure 3.2: Possible mode for triangular cells

3.1 Second Difference Operators

To compute the fourth difference smoothing operator for freestream smoothing, a second difference of a second difference is is computed. For both quadrilaterals and triangles two second difference operators are examined. The first is a relatively fast operator which gives a non-zero second difference for a linear function on a non-uniform grid. The second operator is slower but results in a zero second difference for a linear function. By examining the effect of the second difference operator on a linear function the accuracy of the operator is tested, since for second order or higher accuracy the contribution must be zero.

Typical triangular and quadrilateral cells are shown in Figure 3.3 with the corresponding nodes.

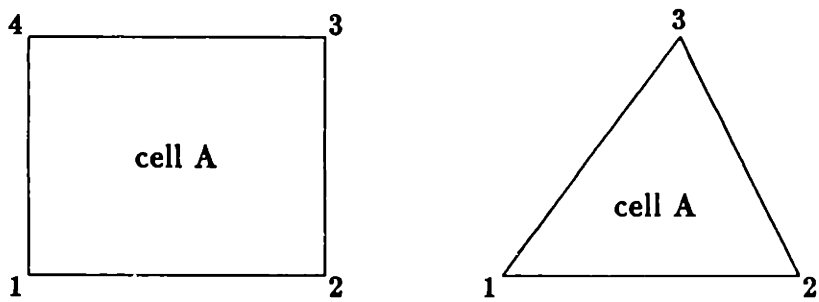


Figure 3.3: Typical triangular and quadrilateral cells

3.1.1 Low-Accuracy Second Difference

This operator is not dependent on the location of the nodes surrounding the node for which the second difference is computed, but merely on the function values at these nodes. For a triangular mesh the contribution from cell A to the second difference at node 1 is

$$(D^2S)_{1A} = (S_3 + S_2 - 2S_1) \tag{3.1}$$

where S is the variable for which the second difference is computed. For a quadrilateral mesh the contribution from cell A to the second difference at node 1 is

$$(D^2S)_{1A} = (S_4 + S_3 + S_2 - 3S_1) \tag{3.2}$$

This second difference is conservative for both triangular and quadrilateral meshes since the total contribution of each cell is zero.

3.1.2 High-Accuracy Second Difference

This operator consists of finding the first derivative for each cell and then combining the derivatives on the cells surrounding a node to form a second difference. Unlike the low-accuracy second difference operator, this operator is dependent on the grid geometry. The operator for a triangular cell mesh will be examined first since the operator for a quadrilateral cell mesh is essentially a triangular operator.

Referring to Figure 2.3 the first derivative is found with respect to x and y for cell A.

$$\begin{aligned}
(S_x)_A &= \frac{1}{A_A}\iint_{cellA}\frac{\partial S}{\partial x}dx\,dy \\
&= \frac{1}{A_A}\int_{1-2-3} S\,dy \\
&= \frac{1}{A_A}\left(\frac{1}{2}(S_2+S_1)(y_2-y_1) + \frac{1}{2}(S_3+S_2)(y_3-y_2) + \frac{1}{2}(S_1+S_3)(y_1-y_3)\right) \\
&= \frac{1}{2A_A}\Big(S_1(y_2-y_3) + S_2(y_3-y_1) + S_3(y_1-y_2)\Big) \qquad (3.3)
\end{aligned}$$

$$\begin{aligned}
(S_y)_A &= \frac{1}{A_A}\iint_{cellA}\frac{\partial S}{\partial y}dx\,dy \\
&= \frac{-1}{A_A}\int_{1-2-3} S\,dx
\end{aligned}$$

$$= \frac{-1}{A_A}\left(\frac{1}{2}(S_2+S_1)(x_2-x_1) + \frac{1}{2}(S_3+S_2)(x_3-x_2) + \frac{1}{2}(S_1+S_3)(x_1-x_3)\right)$$

$$= \frac{-1}{2A_A}\left(S_1(x_2-x_3) + S_2(x_3-x_1) + S_3(x_1-x_2)\right) \tag{3.4}$$

A similar process is performed to create a second difference. The integration is taken around all the triangles which surround the node for which the second difference is computed, using the derivative values calculated at the cells. To get a second difference instead of a second derivative, there is no division by the area of the integrated region. The contribution to the second difference at node 1 from cell A is

$$(D^2S)_{1A} = \int_{2-3}(S_x)_A dy - (S_y)_A dx$$

$$= \frac{1}{2}\left[(S_x)_A(y_3 - y_2) - (S_y)_A(x_3 - x_2)\right] \tag{3.5}$$

This second difference operator takes about twice as long to compute as the low-accuracy second difference operator. It is also conservative since again the total contribution of each cell is zero.

To formulate a second difference operator for a quadrilateral mesh a similar process is employed. It turns out that if the first derivative is found by integrating around the complete quadrilateral, the oscillatory modes are not damped out, and the primary purpose of the operator is not fulfilled. To prevent this problem, the quadrilateral is broken into triangles and the triangular operator is applied. The division of cell A is shown in Figure 3.4. For cell A the first derivatives for the triangle corresponding to node 1 is

$$(S_x)_{A1} = \frac{1}{A_{A1}}\iint_{cellA}\frac{\partial S}{\partial x}dx\,dy$$

$$= \frac{1}{A_{A1}}\int_{1-2-4} S\,dy$$

$$= \frac{1}{A_{A1}}\left(\frac{1}{2}(S_2+S_1)(y_2-y_1) + \frac{1}{2}(S_4+S_2)(y_4-y_2) + \frac{1}{2}(S_1+S_4)(y_1-y_4)\right)$$

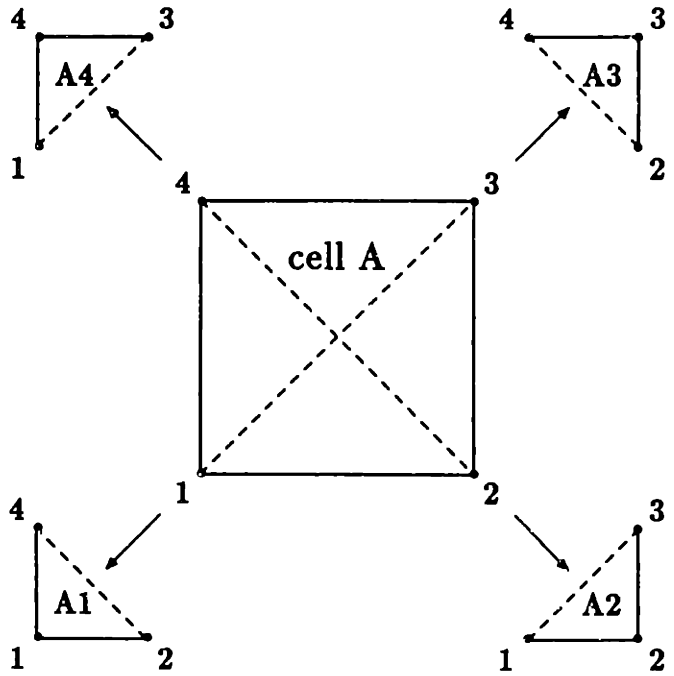


Figure 3.4: Division of quadrilateral cell for non-conservative, high-accuracy second difference

$$= \frac{1}{2A_{A1}}\Big(S_1(y_2-y_4)+S_2(y_4-y_1)+S_4(y_1-y_2)\Big) \tag{3.6}$$

$$\begin{aligned}
(S_y)_{A1} &= \frac{1}{A_{A1}}\int\int_{cellA}\frac{\partial S}{\partial y}dx\,dy \\
&= \frac{-1}{A_{A1}}\int_{1-2-4} S\,dx \\
&= \frac{-1}{A_{A1}}\Big(\frac{1}{2}(S_2+S_1)(x_2-x_1)+\frac{1}{2}(S_4+S_2)(x_4-x_2)+\frac{1}{2}(S_1+S_4)(x_1-x_4)\Big) \\
&= \frac{-1}{2A_{A1}}\Big(S_1(x_2-x_4)+S_2(x_4-x_1)+S_4(x_1-x_2)\Big) \qquad (3.7)
\end{aligned}$$

where A_{A1} refers to the area of the triangle corresponding to node 1 for cell A.

Next the integral is taken around the outside edge of the triangle to give the contribution of cell A1 to the second difference at node 1.

$$
\begin{aligned}
(D^2S)_{1A} &= \int_{2-4} (S_x)_{A1}dy - (S_y)_{A1}dx \\
&= \frac{1}{2}\Big[(S_x)_{A1}(y_4 - y_2) - (S_y)_{A1}(x_4 - x_2)\Big] \qquad (3.8)
\end{aligned}
$$

Similar contributions to corresponding nodes are found for the other three triangles into which cell A is divided. As with the triangular version of this operator it takes about twice as long to compute this operator as the low-accuracy second difference operator. For the quadrilateral mesh this second difference is not conservative since the total contribution from each cell is not zero since the first derivative contribution is not constant in the cell.

3.2 Freestream Smoothing

To damp out oscillations and provide stability a fourth difference operator is added throughout the flow field. It is desirable to have a conservative operator so all terms in the flow field cancel and therefore no mass, momentum or energy production occurs due to the smoothing.

The first method of creating a fourth difference is to use the low-accuracy second difference twice by operating first on the state vector and then operating on this second difference. This fourth difference is conservative since the contribution of each cell to the numerical smoothing is zero, but is only second order accurate on a uniform mesh since the second difference operator used is only second order accurate on a uniform mesh. The second method is to compute a second difference of the state vector using the high-accuracy method and operate on this second difference with the low-accuracy second difference. This operator is second order accurate since the first operator is second order accurate and conservative since the second operator is conservative and again the contribution of each cell to the numerical smoothing is zero. The second

method is more expensive than the first, but the effect per iteration is an increase of only 5-10% which is a small increase for the gain in accuracy.

The fourth difference is computed as a contribution from each cell to the nodes which make up that cell. The freestream smoothing is multiplied by a coefficient $\epsilon^{(1)}$, between 0.0 and 0.05, to control the amount of smoothing added to the scheme. To make the numerical smoothing term consistent with the numerical scheme, the fourth difference added to the change in the state vector at each node must be multiplied by the ratio of $(\frac{\Delta t}{A})$ for the node to $(\frac{\Delta t}{A})$ for the cell as described by Roberts in [18]. For the quadrilateral Ni scheme in Equation (2.9) and for the triangular Ni scheme in Equation (2.17) the change to the state vector due to cell A at node 1 becomes

$$\delta U_{1A} = \delta U_{1A} + \epsilon^{(1)} \left(\frac{\Delta t}{A}\right)_1 \left(\frac{A}{\Delta t}\right)_A (D^2(D^2U))_{1A} \tag{3.9}$$

The value of the flux contribution for the quadrilateral Jameson scheme in Equation (2.24) and the triangular Jameson scheme in Equation (2.27) is changed such that the contribution from cell A to node 1 becomes

$$\text{flux}_{1A} = \text{flux}_{1A} + \epsilon^{(1)} \left(\frac{A}{\Delta t}\right)_A (D^2(D^2U))_{1A} \tag{3.10}$$

The flux terms are multiplied by $(\frac{\Delta t}{A})$ for each node when the change in the state vector is computed in Equation (2.20), so the smoothing term is consistent with the numerical scheme. The numerical smoothing term for the Jameson scheme is only computed after the first two of the four steps.

3.3 Shock Smoothing

In regions with strong discontinuities the fourth difference smoothing is not enough to cause any of the schemes mentioned here to be stable. When strong discontinuities are detected the freestream smoothing is turned off since it is destabilizing, and shock smoothing is turned on. For the Jameson schemes the shock smoothing consists of a

low-accuracy second difference which is turned on in regions of high pressure gradient. Another method is used for the Ni schemes which adds a bulk viscosity term in regions of high velocity flux.

3.3.1 Second Difference Smoothing

To determine when to turn on the second difference smoothing a pressure switch is used. This switch is found for each node and consists of the second difference of pressure computed using the low-accuracy operator as shown in Equation (3.1) for triangular meshes and Equation (3.2) for quadrilateral meshes and divided by the pressure at the node. Near shocks this switch will be of order 1 but in the freestream it will be of order Δx^2. For node 1 this switch is

$$(s_p)_1 = \frac{D^2 p}{p_1} \tag{3.11}$$

Once the switch is found it is used when finding the second difference of the state vector at the nodes which is computed using an operator similar to the low-accuracy operator and multiplied by a coefficient, $\epsilon^{(2)}$, between 0.0 and 0.1.

For the triangular scheme the value of the flux contribution in Equation (2.27) is changed such that the contribution from cell A to node 1 becomes

$$\begin{aligned} \text{flux}_{1A} \;=\; \text{flux}_{1A} \;+\; \epsilon^{(2)} \frac{1}{2} \Bigg[& \Big((s_p)_1 + (s_p)_2\Big)(U_2 - U_1) \\ & + \Big((s_p)_1 + (s_p)_3\Big)(U_3 - U_1) \Bigg] \end{aligned} \tag{3.12}$$

Similarly for the quadrilateral scheme the flux contribution in Equation (2.24) is changed such the contribution from cell A to node 1 becomes

$$\begin{aligned} \text{flux}_{1A} &= \text{flux}_{1A} + \epsilon^{(2)} \frac{1}{2} \Big[\Big((s_p)_1 + (s_p)_2 \Big)(U_2 - U_1) \\ &\quad + \Big((s_p)_1 + (s_p)_3 \Big)(U_3 - U_1) \\ &\quad + \Big((s_p)_1 + (s_p)_4 \Big)(U_4 - U_1) \Big] \end{aligned} \tag{3.13}$$

3.3.2 Bulk Viscosity Smoothing

To capture shocks, viscosity is added near shocks since viscosity would capture a shock if it were accounted for in the governing equations. This method is similar to the method described by Richtmyer and Morton in [17]. To ensure that the shock width remains nearly the same, regardless of shock strength, terms quadratic in the strain rate are added to the differential equation. The volumetric dilatation, or the velocity divergence, is a measure of the strain rate. In shocks the volumetric dilatation is highly negative since shocks represent regions of extreme compression. The shock smoothing is turned on when the volumetric dilatation is negative and is limited such that it is never less than -0.1. The viscosity term which is added to the flux vectors is proportional to the volumetric dilatation squared and, to prevent excessive smoothing at stagnation points, the velocity squared. The change in the flux vectors, ΔF and ΔG now have the viscosity term added to their second and third elements respectively.

$$\begin{aligned} (\Delta F_2)_A &= (\Delta F_2)_A + \tfrac{1}{2} max(-0.1, min(0., \text{div}\vec{w}_A))\, \text{div}\vec{w}_A (u^2+v^2) \\ (\Delta G_3)_A &= (\Delta G_3)_A + \tfrac{1}{2} max(-0.1, min(0., \text{div}\vec{w}_A))\, \text{div}\vec{w}_A (u^2+v^2) \end{aligned} \tag{3.14}$$

For triangles the volumetric dilatation for cell A can be found by

$$\begin{aligned} \text{div}\vec{w}_A &= \text{scaled volumetric dilatation of triangular cell A} \\ &= \sqrt{\frac{2}{A}} \iint_{cell A} \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right) dx dy \end{aligned}$$

$$= \sqrt{\frac{2}{A}} \oint_{1-2-3} (udy - vdx)$$

$$= \frac{1}{\sqrt{2A}} \Big(u_1(y_2 - y_3) + u_2(y_3 - y_1) + u_2(y_1 - y_2) - (v_1(x_2 - x_3) - v_2(x_3 - x_1) - v_2(x_1 - x_2)) \Big) \qquad (3.15)$$

and similarly for quadrilaterals

$$\text{div}\vec{w}_A = \text{scaled volumetric dilatation of quadrilateral cell A}$$

$$= \sqrt{\frac{2}{A}} \iint_{cellA} \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right) dxdy$$

$$= \sqrt{\frac{2}{A}} \oint_{1-2-3-4} (udy - vdx)$$

$$= \frac{1}{\sqrt{2A}} \Big((u_1 - u_3)(y_2 - y_4) + (u_2 - u_4)(y_3 - y_1) - (v_1 - v_3)(x_2 - x_4) + (v_2 - v_4)(x_3 - x_1) \Big) \qquad (3.16)$$

Chapter 4

Mesh Generation and Pointer System

In this chapter the mesh generation and the system for storing the information in the mesh are described. An elliptic mesh generator was used to generate a quadrilateral mesh and, since the development of a triangular mesh generator was not a major focus of this work, the quadrilaterals in this mesh were divided into triangles to generate a triangular mesh.

For meshes consisting of a regular arrangement of quadrilateral cells, the the nodes of the mesh have traditionally been described in two dimensions by an (i,j) indexing. An alternative to this method of describing a mesh is to assign each node a number and describe the connection between the nodes by one dimension of an array of mesh elements. These elements can consist of cells, faces, edges or any other element which is important in the numerical scheme. This system of describing a mesh will be referred to as a pointer system. When a pointer system is used the flow solver is separated from the mesh generator. Due to the inherent irregularity of most triangular meshes, it is usually not possible to use the first method of describing a mesh and a pointer system must be used. All the meshes used for this study are described using a pointer system.

4.1 Elliptic Mesh Generator

Elliptic partial differential equations are used to generate a smooth mesh. The equations are defined in a computational plane with coordinates ξ and η whose nodes have a one to one mapping to the physical plane. The equations used are

$$\xi_{xx} + \xi_{yy} = P(\xi,\eta) \tag{4.1}$$

$$\eta_{xx} + \eta_{yy} = Q(\xi, \eta) \tag{4.2}$$

where P and Q are forcing terms defined by Steger and Sorenson [21]. By switching the independent and dependent variables these equations become

$$\alpha x_{\xi\xi} - 2\beta x_{\xi\eta} + \gamma x_{\eta\eta} = -J^2(P x_\xi + Q x_\eta) \tag{4.3}$$

$$\alpha y_{\xi\xi} - 2\beta y_{\xi\eta} + \gamma y_{\eta\eta} = -J^2(P y_\xi + Q y_\eta) \tag{4.4}$$

where

$$\begin{aligned} J &= x_\xi y_\eta - x_\eta y_\xi \\ \alpha &= x_\eta^2 + y_\eta^2 \\ \beta &= x_\xi x_\eta + y_\eta y_\xi \\ \gamma &= x_\xi^2 + y_\xi^2 \end{aligned}$$

An H-mesh in a duct or blade cascade is created where the line $\xi = 0$ corresponds to the inlet boundary and the line $\xi = 1$ corresponds to the outlet boundary. The lines $\eta = 0$ and $\eta = 1$ correspond to the upper and lower surface of the domain.

4.2 Complete Pointer System

To assure that the constraints of the pointer system would not restrict the code development process, a very complete pointer system was put together. This system stores three to four times as much information as is required to program the numerical schemes described here. The minimum requirements are described in the following section.

The pointer system used here consists of arrays of nodes, cells, faces, edge faces and edge nodes. These arrays are interconnected in that elements in one array will point

to elements in another array. Nodes are the only elements which do not point to other elements. The interconnection of the arrays is shown in Figure 4.1 where the arrows indicate pointing from one array to another. In essence the complete pointer system overdefines the connection between the arrays, but this allowed more freedom in the code development process.

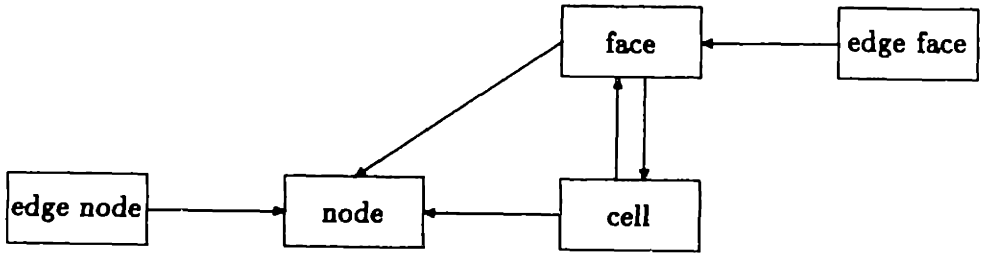


Figure 4.1: Interconnection of pointer arrays

4.2.1 Node Arrays

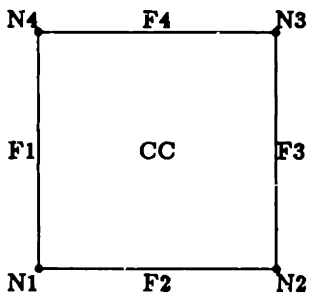
Each node is assigned a number. Arrays containing all the nodes hold information about the x and y coordinate as well as the four elements of the state vector U for each node. These variables were stored in

$$\begin{aligned} x(\mathrm{N}) \quad & \mathrm{N} = 1, \mathrm{N}_{max} \\ y(\mathrm{N}) \quad & \mathrm{N} = 1, \mathrm{N}_{max} \\ U(i,\mathrm{N}) \quad & \mathrm{N} = 1, \mathrm{N}_{max} \\ & i = 1,4 \end{aligned}$$

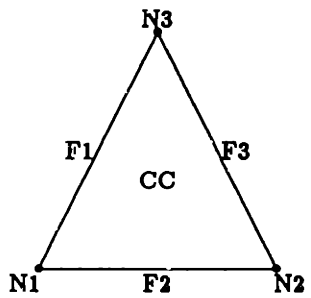
Other information is temporarily stored at each node, such as the flux vectors F and G and the change in the state vector δU.

4.2.2 Cell Arrays

The cell is the basic computational unit for all the schemes described here. The value of a flux, for example, is calculated for the cell and distributed to the nodes which make up the cell. To completely describe the cell and its surrounding elements, each cell points to the nodes and faces which make up the cell. The cell array for cell CC consists of



$$
\begin{aligned}
\text{cell}(1,\text{CC}) &= \text{F1} \\
\text{cell}(2,\text{CC}) &= \text{F2} \\
\text{cell}(3,\text{CC}) &= \text{F3} \\
\text{cell}(4,\text{CC}) &= \text{F4} \\
\text{cell}(5,\text{CC}) &= \text{N1} \\
\text{cell}(6,\text{CC}) &= \text{N2} \\
\text{cell}(7,\text{CC}) &= \text{N3} \\
\text{cell}(8,\text{CC}) &= \text{N4}
\end{aligned}
$$

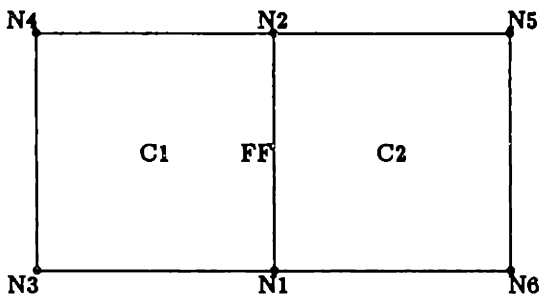


$$
\begin{aligned}
\text{cell}(1,\text{CC}) &= \text{F1} \\
\text{cell}(2,\text{CC}) &= \text{F2} \\
\text{cell}(3,\text{CC}) &= \text{F3} \\
\text{cell}(4,\text{CC}) &= \text{N1} \\
\text{cell}(5,\text{CC}) &= \text{N2} \\
\text{cell}(6,\text{CC}) &= \text{N3}
\end{aligned}
$$

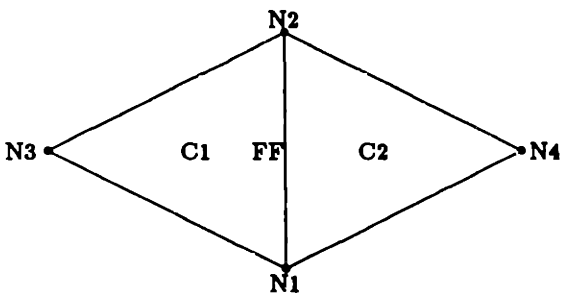
As with the nodal arrays, other information is temporarily stored at each cell, such as the area divided by the timestep ($\frac{A}{\Delta t}$) and the change in the state vector at the cell ΔU. This information can be accessed by merely knowing the cell number.

4.2.3 Face Arrays

At one time, using the face as the basic computational element was considered. This method is effective for the Jameson schemes mentioned here, but for consistency between the Ni and Jameson schemes was replaced by the cell based method. Currently the face elements are used to indirectly describe edge faces and plot the computational mesh. Each face points to the cells on either side of it and the nodes which belong to these cells. When a node or face lies outside the computational domain its value is set to zero. The face array for face FF consists of



$$
\begin{aligned}
face(1,FF) &= C1 \\
face(2,FF) &= C2 \\
face(3,FF) &= N1 \\
face(4,FF) &= N2 \\
face(5,FF) &= N3 \\
face(6,FF) &= N4 \\
face(7,FF) &= N5 \\
face(8,FF) &= N6
\end{aligned}
$$



$$
\begin{aligned}
face(1,FF) &= C1 \\
face(2,FF) &= C2 \\
face(3,FF) &= N1 \\
face(4,FF) &= N2 \\
face(5,FF) &= N3 \\
face(6,FF) &= N4
\end{aligned}
$$

4.2.4 Edge Face Arrays

To implement the boundary conditions it is necessary to know which faces or cells lie on the boundary and their orientation with respect to the boundary. The elements

of the edge face array point to the elements of the face array. Each face which is on the boundary is oriented such that C1 and N3 (and N4 for the quadrilaterals) lie inside the domain.

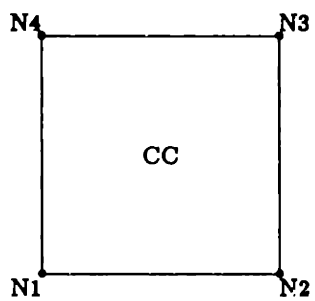
$$\text{edge(EF)} \;=\; \text{FF}$$

4.2.5 Edge Node Arrays

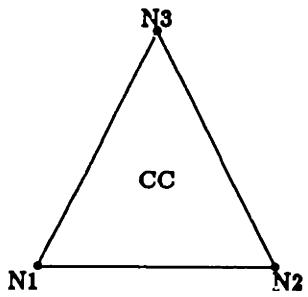
For convenience an array of edge nodes was also created. These nodes correspond to the solid wall boundary nodes. Information such as the flow angle α at a boundary node was stored in an array indexed by the edge nodes.

4.3 Required Pointer System

The pointer system described in the previous section contains more information than is required to implement the numerical schemes described here. The nodal arrays are the independent variables of the system and should remain as they were described. The rest of the pointer system can be replaced by a single array of cells. To specify which cells lie on the boundaries the edge cells can be placed at the beginning of the array with the first two nodes of these cells lying on the boundary. The cell array should be arranged such that the entry for cell CC would consist of



$$\begin{aligned} \text{cell}(1,\text{CC}) &= \text{N1} \\ \text{cell}(2,\text{CC}) &= \text{N2} \\ \text{cell}(3,\text{CC}) &= \text{N3} \\ \text{cell}(4,\text{CC}) &= \text{N4} \end{aligned}$$



$$\begin{aligned} cell(1,CC) &= N1 \\ cell(2,CC) &= N2 \\ cell(3,CC) &= N3 \end{aligned}$$

4.4 Vectorization

To eliminate data dependencies which restrict vectorization and to create long vector lengths, a coloring system is used. This system assigns a color to each element in the array such that all elements of the same color do not contain data dependencies. The arrays are then sorted such that all elements of the same color occupy consecutive locations in the array. For example, data dependencies may occur in an operation on an array of cells when two different cells contain a node in common. For quadrilateral cells it is possible for simple cases to color the cell array with four colors such that the cells of one color do not have common nodes.

All arrays in the pointer system used here were colored so the code would run on an Alliant FX/3 computer which used vector/concurrent arithmetic.

Chapter 5
Computational Examples

To demonstrate the similarities and differences between the numerical methods described here, some examples will be shown. Little reference will be made to computational times since in practice multi-grid or adaptation will be used to accelerate the computational time. The convergence rate is also related to the numerical smoothing used for freestream and shock capturing.

A discussion on the accuracy of the numerical schemes discussed here is given in Chapter 6. In general the accuracy of a numerical method is reduced by using the low-accuracy smoothing as opposed to the high-accuracy smoothing.

The regular and irregular quadrilateral meshes used here are similar to the meshes shown in Figures 6.1 and 6.2. The irregular triangular meshes are similar to the mesh shown in Figure 6.3.

5.1 Supersonic Circular Arc Bump

The first problem is supersonic flow in a channel with a 4% circular arc bump on the lower surface. The inlet Mach number is 1.4. A 64×16 cell mesh is used. This problem illustrates the effect of the shock capturing methods used as well as interactions between shocks and solid wall boundaries, and shocks with other shocks.

The quadrilateral and triangular Jameson and Ni schemes with the high-accuracy numerical smoothing required essentially the same amount of computational time to converge from a uniform freestream initial condition. Slightly less time was required when the low-accuracy numerical smoothing was used since it is less expensive to compute

than the high-accuracy smoothing. Essentially twice as many iterations were required for the Ni schemes than the Jameson schemes since the CFL number for the Ni schemes is half as large as for the Jameson schemes.

Several Mach number contours with increment 0.05 are shown in Figures 5.1 to 5.7 for different numerical schemes and numerical smoothing techniques. For the most part the solutions look the same.

In Figure 5.1 the effect of the low-accuracy smoothing on an irregular mesh can be seen. Comparing the solution in Figure 5.1 to the solutions in Figure 5.2 with the low-accuracy smoothing on a regular mesh and Figure 5.3 with the high-accuracy smoothing on an irregular mesh, the contours are not as smooth. This effect exists because the low-accuracy smoothing includes no information about the location of the neighboring mesh points. For an irregular mesh this becomes important. In Figure 5.1 another effect of the low-accuracy smoothing can be seen. The contours on the lower surface do not intersect the solid wall smoothly. A slight turning of the contour lines can be seen. This effect exists because the inviscid solution has $\frac{\partial u}{\partial n} \neq 0$ on curved walls, whereas the low-accuracy smoothing has a one-sided bias which implicitly assumes that $\frac{\partial u}{\partial n} = 0$.

All the solutions pick up the normal shock on the upper surface of the duct in the reflection from the leading edge shock. This reflection interacts with the trailing edge shock behind the bump, and reflects of the lower surface of the duct to combine with the trailing edge shock. In general the quadrilateral schemes pick up this interaction between the shocks better than the triangular schemes. This is partly due to the resolution since as the mesh is refined, the shock is picked up by the triangular schemes as well as the quadrilateral schemes. The triangular schemes have a harder time keeping the leading edge shock and its first reflection straight. More work must be done with the shock capturing smoothing to correct this.

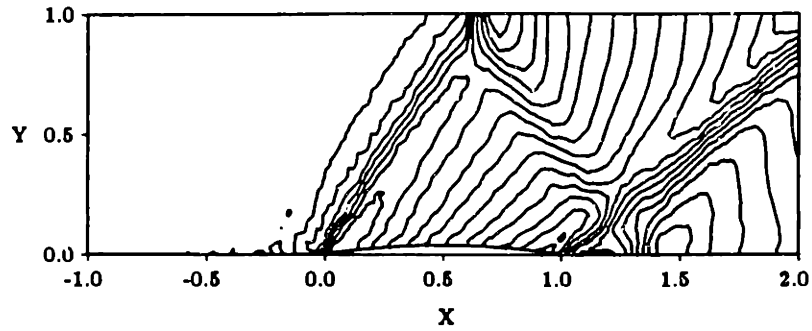


Figure 5.1: Supersonic case: quadrilateral Jameson scheme with low-accuracy smoothing on an irregular mesh

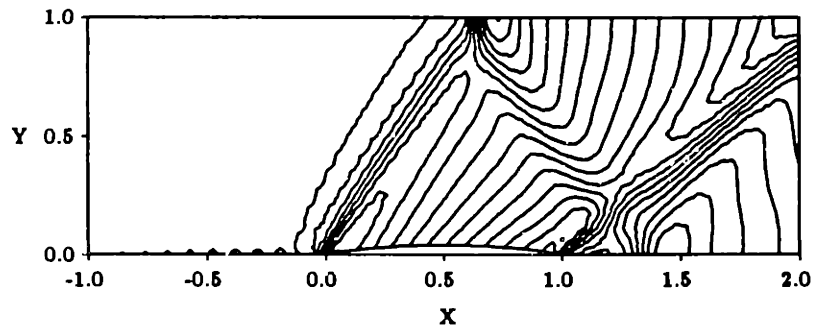


Figure 5.2: Supersonic case: quadrilateral Jameson scheme with low-accuracy smoothing on a regular mesh

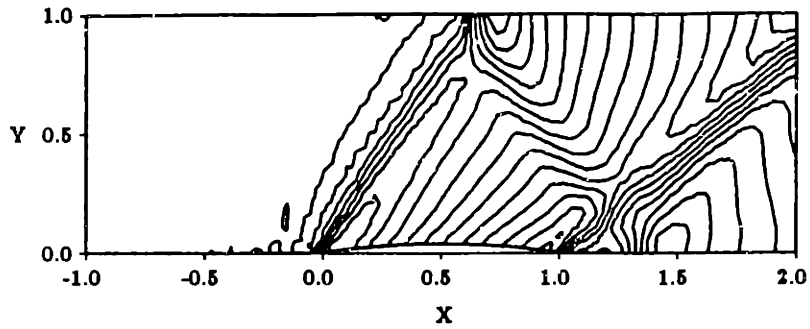


Figure 5.3: Supersonic case: quadrilateral Jameson scheme with high-accuracy smoothing on an irregular mesh

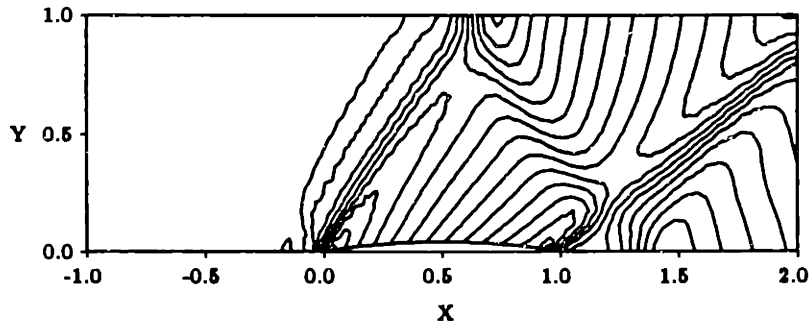


Figure 5.4: Supersonic case: quadrilateral Ni scheme with high-accuracy smoothing on a regular mesh

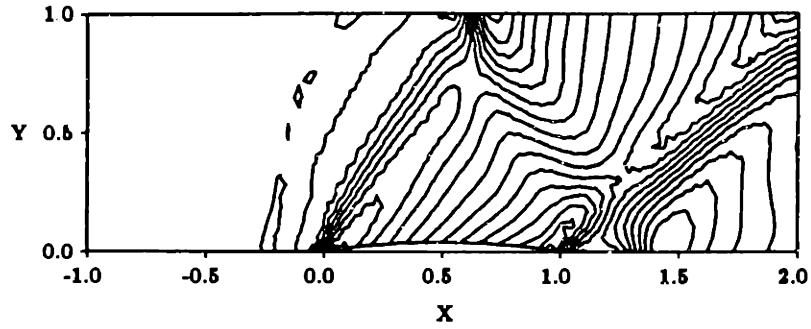


Figure 5.5: Supersonic case: triangular Jameson scheme with high-accuracy smoothing on an irregular mesh

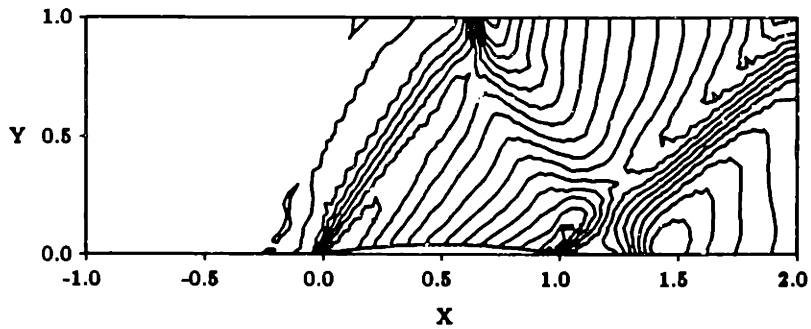


Figure 5.6: Supersonic case: triangular Jameson scheme with low-accuracy smoothing on an irregular mesh

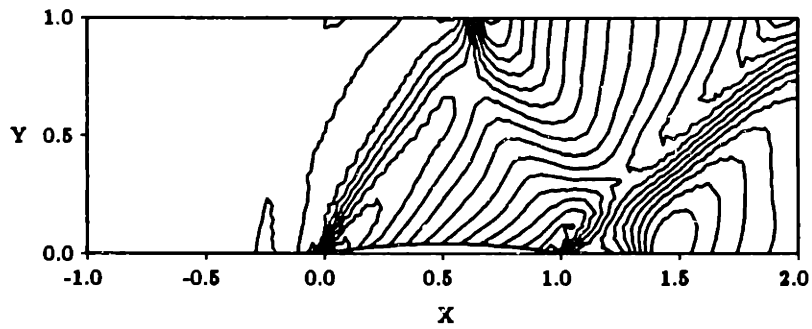


Figure 5.7: Supersonic case: triangular Ni scheme with high-accuracy smoothing on an irregular mesh

5.2 Transonic Circular Arc Bump

The second problem is transonic flow in a channel with a 10% circular arc bump on the lower surface. The inlet Mach number is 0.675. A 64×16 cell mesh is used. The inlet is subsonic, but the flow is accelerated over the top of the bump and a shock forms. This problem again illustrates the shock capturing methods as well as subsonic interactions with solid wall boundaries.

Several Mach number contours with increment 0.1 are shown in Figures 5.8 to 5.15 for different numerical schemes and numerical smoothing techniques. Again, for the most part the solutions look the same. Some plots show tighter shocks, but this is due to different shock capturing methods more than the numerical schemes themselves.

As with the supersonic test case, the solution found using the quadrilateral Jameson scheme with low-accuracy smoothing on an irregular mesh, as shown in Figure 5.8, is significantly less smooth than the same case on a regular mesh, as shown in Figure 5.9. The high-accuracy smoothing used with the quadrilateral Jameson scheme produces much smoother results with both regular and irregular meshes as shown in Figures 5.11 and 5.10. In fact the irregular mesh has a very smooth solution with the high-accuracy smoothing.

In Figure 5.13 an effect of the low-accuracy smoothing on the interaction with solid wall boundaries can be seen. The smoothing tends to keep the Mach contours from intersecting the solid wall smoothly. This can be seen by a turning of the contour lines near the wall. This effect can also be seen in for the supersonic case in Figure 5.1. Once again the lack of information about the location of the mesh points in the low-accuracy smoothing and the imposition of an unnatural boundary condition for the numerical smoothing causes a problem.

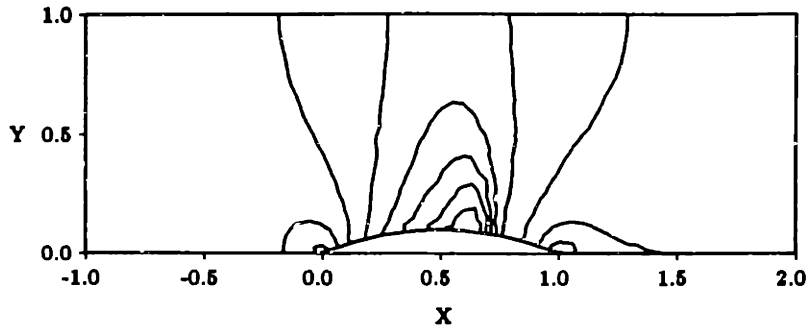


Figure 5.8: Transonic case: quadrilateral Jameson scheme with low-accuracy smoothing on an irregular mesh

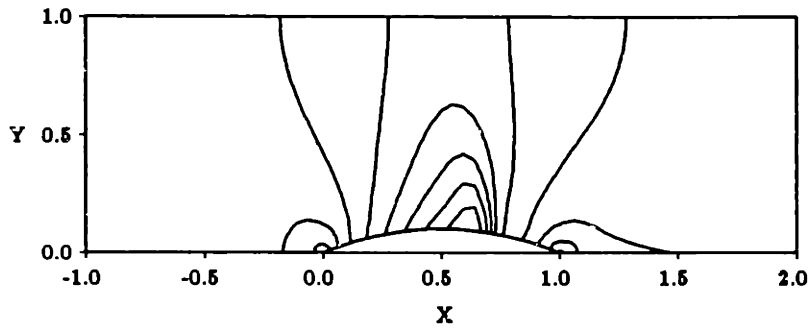


Figure 5.9: Transonic case: quadrilateral Jameson scheme with low-accuracy smoothing on a regular mesh

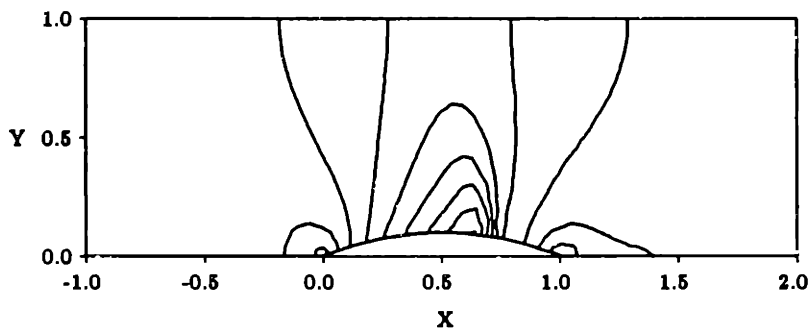


Figure 5.10: Transonic case: quadrilateral Jameson scheme with high-accuracy smoothing on an irregular mesh

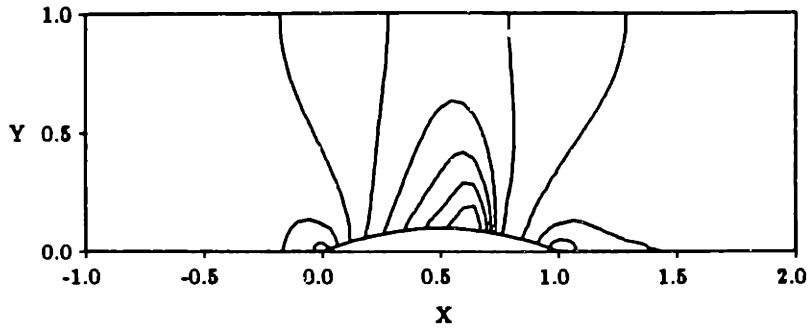


Figure 5.11: Transonic case: quadrilateral Jameson scheme with high-accuracy smoothing on a regular mesh

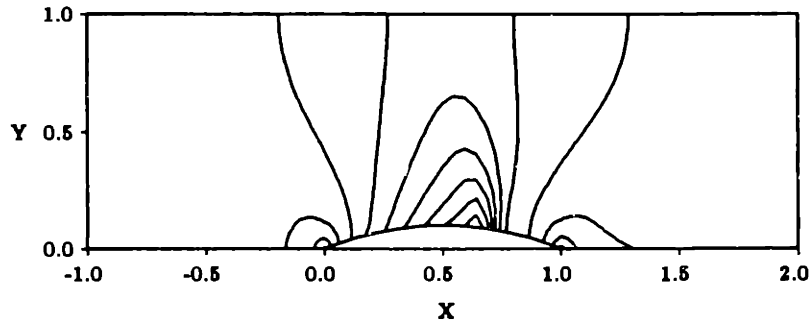


Figure 5.12: Transonic case: quadrilateral Ni scheme with high-accuracy smoothing on a regular mesh

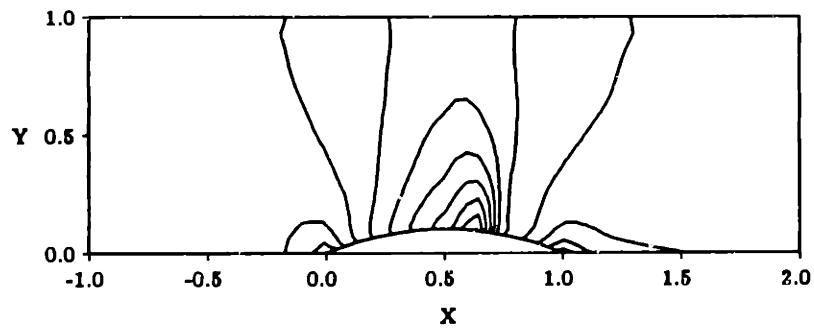


Figure 5.13: Transonic case: triangular Jameson scheme with low-accuracy smoothing on an irregular mesh

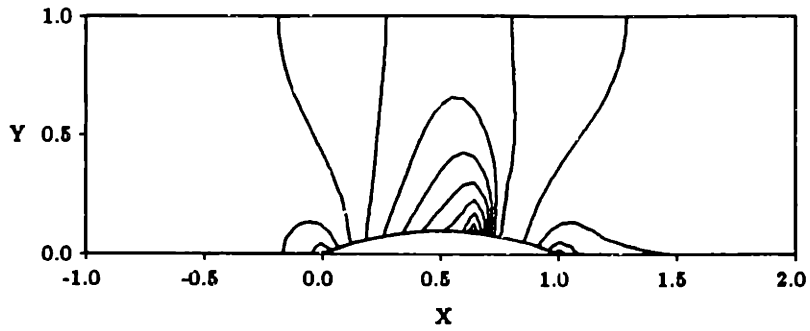


Figure 5.14: Transonic case: triangular Jameson scheme with high-accuracy smoothing on an irregular mesh

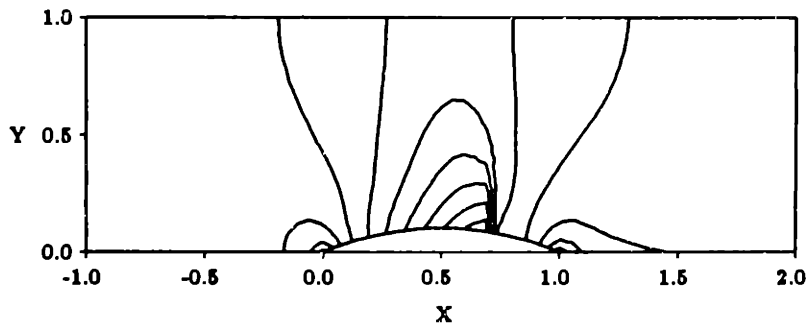


Figure 5.15: Transonic case: triangular Ni scheme with high-accuracy smoothing on an irregular mesh

Chapter 6
Accuracy Study

It is shown mathematically by Giles [5] that both the Ni scheme and the Jameson scheme are second order accurate for either quadrilateral or triangular meshes. The nature of the mathematical proof as well as a numerical accuracy study which confirms this result are examined in this chapter.

6.1 Mathematical Study

Giles shows in [5] that the global solution error is second order for steady state node based finite volume schemes on irregular meshes. The analysis states that the local truncation error is first order, but that this does not imply that the global error is also first order. It is assumed that the numerical finite volume scheme is conservative and h is some typical cell length. With this in mind, it is shown that the truncation error has a spectral content which can be split into two parts, a low-frequency component with an amplitude which is $O(h^2)$ and a high-frequency component with an amplitude which is $O(h)$. It is then shown that the hyperbolic differential operator

$$\mathcal{L}U = \frac{\partial F(U)}{\partial x} + \frac{\partial G(U)}{\partial y} \tag{6.1}$$

has a transfer function which is $O(1)$ at low frequencies and $O(1/h)$ at high frequencies. From this it can be concluded that the global error is $O(h^2)$ for both low and high frequencies, therefore the schemes are second order accurate. An important assumption in the analysis is that the numerical smoothing does not produce a truncation error which is worse than first order on irregular meshes.

6.2 Numerical Study

A numerical study was performed to determine the accuracy of the schemes described in Chapter 2. The purpose of this study was to confirm that these schemes are second order accurate.

For inviscid flow, there should be no total pressure loss for smooth, subsonic flow. Thus, for such flows any total pressure loss is purely numerical in nature. With this in mind, the total pressure loss is used to define the error in subsonic flow through a duct with a $\sin^2 x$ bump on the lower surface and a 0.50 inlet Mach number. The height of the lower surface is given by

$$y = 0.10 \sin^2(\pi x) \quad 0 \leq x \leq 1 \tag{6.2}$$

To describe the numerical error on a mesh a global error is defined as the L_2-norm of this local error.

$$\eta = \frac{(p_o)_{upstream} - (p_o)_{local}}{(p_o)_{upstream}} \tag{6.3}$$

$$error = \sqrt{\frac{\sum_{i=1}^{N} \eta_i^2}{N}} \tag{6.4}$$

For meshes composed of quadrilateral cells it is easy to define an appropriate mesh for an accuracy study. To create a *regular quadrilateral mesh* a mesh is laid out in a rectangular duct with a $\sin^2 x$ bump on the lower surface which is 3 units long and 1 unit high where cells away from the bump are square. h, a typical cell length, is defined as the length of the cell faces. Four different meshes are used for the study with 8, 16, 32 and 64 faces per unit length. The second mesh in this series is shown in Figure 6.1. To create an *irregular quadrilateral mesh* the nodes of these meshes are perturbed by an amount determined by a sine function whose period has no relationship to the mesh. Four different meshes are again used for the study with 8, 16, 32 and 64 faces per unit length. The second mesh is shown in Figure 6.2.

Two different mesh types are used for triangular cells. The first type is an *irregular triangular mesh* and is identical to the meshes described for the quadrilateral cells where the cells are split along the shorter diagonal to create triangles. h is defined in the same manner as for the quadrilaterals since this defines a reference length for the cells. Again four different meshes are used with 8, 16, 32 and 64 faces per unit length. An example of the second mesh is shown in Figure 6.3. A second type of mesh was created which contains only triangles which are very nearly equilateral. This type of mesh is what could be referred to as a *regular triangular mesh*.. The duct for this mesh is slightly wider than the previously described duct to facilitate the use of equilateral triangles. The dimensions of the domain are now $3 \times \frac{5\sqrt{3}}{8}$ units instead of 3×1 units. Three different meshes are used with 8, 16, and 32 faces per unit length. The second mesh is shown in Figure 6.4.

Mach contours are shown in Figure 6.5 for the quadrilateral mesh shown in Figure 6.1 which are computed using the quadrilateral Ni scheme. Figure 6.6 shows % total pressure loss contours for this flow. These are representative examples of solutions for this flow field regardless of the numerical scheme or mesh used.

The error is computed for each scheme on the previously defined sets of meshes. The order of accuracy for the scheme is found by plotting $\log(h)$ verses $\log(error)$ and finding the slope of the resulting line. Since the data will not exactly lie in a line, a last squares approximation was used to fit a line through the data points and compute the slope.

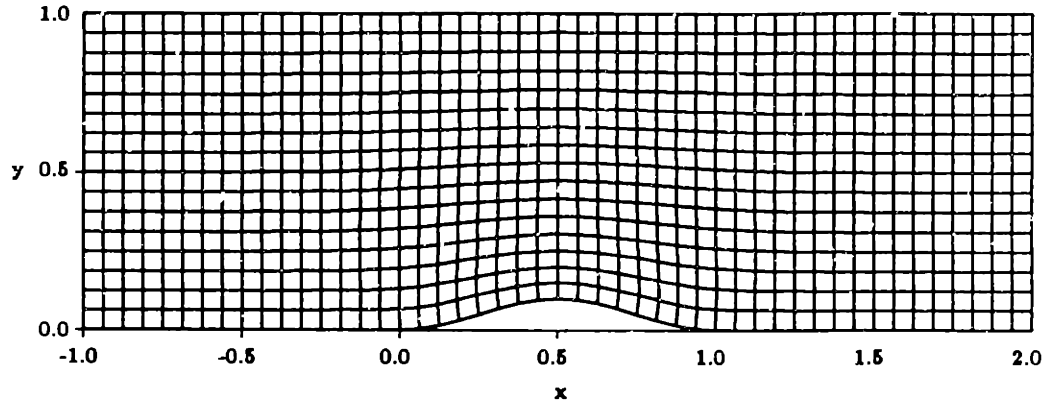


Figure 6.1: Quadrilateral cell mesh for $\sin^2 x$ duct

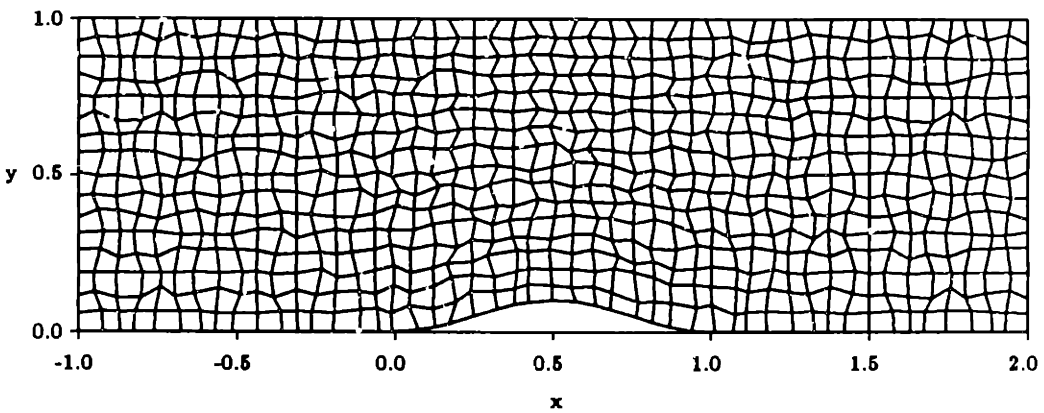


Figure 6.2: Irregular quadrilateral cell mesh for $\sin^2 x$ duct

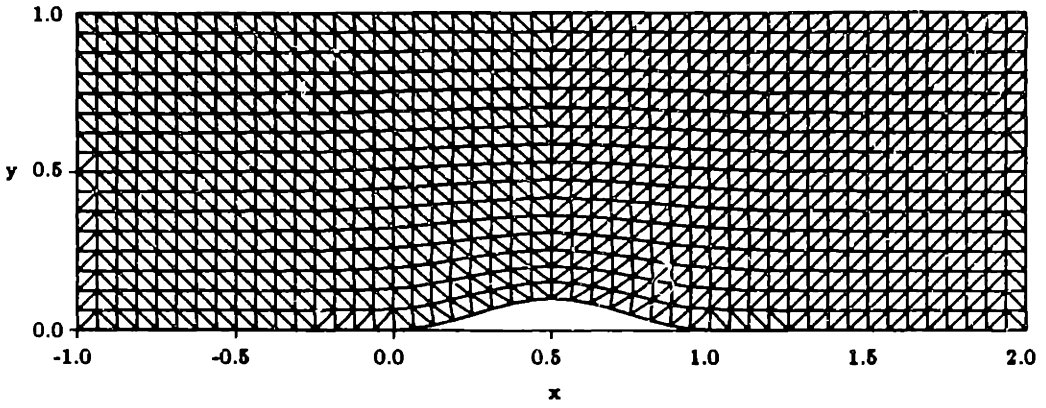


Figure 6.3: Split quadrilateral triangle cell mesh for $\sin^2 x$ duct

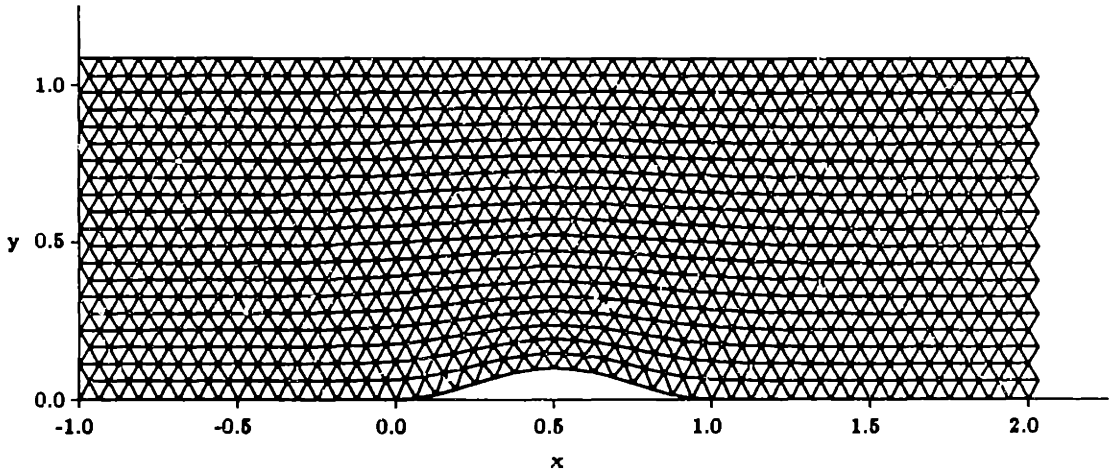


Figure 6.4: Equilateral triangle cell mesh for $\sin^2 x$ duct

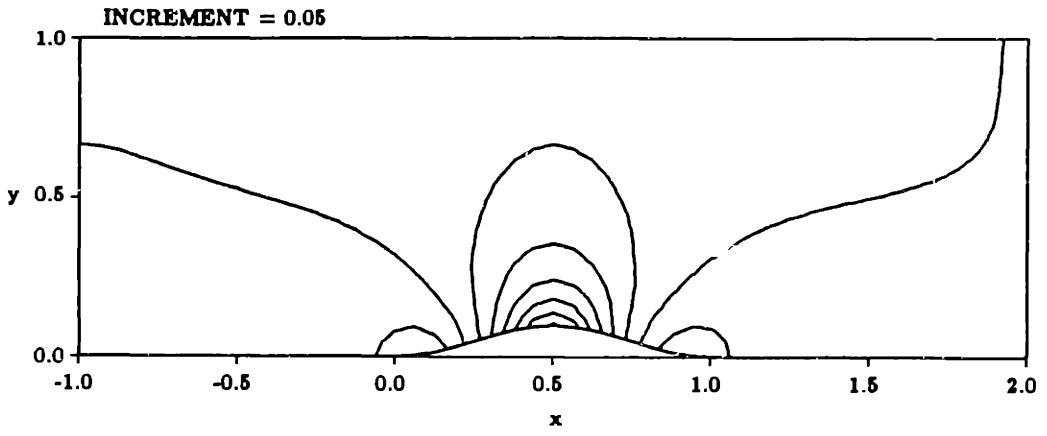


Figure 6.5: Mach contours for $\sin^2 x$ duct

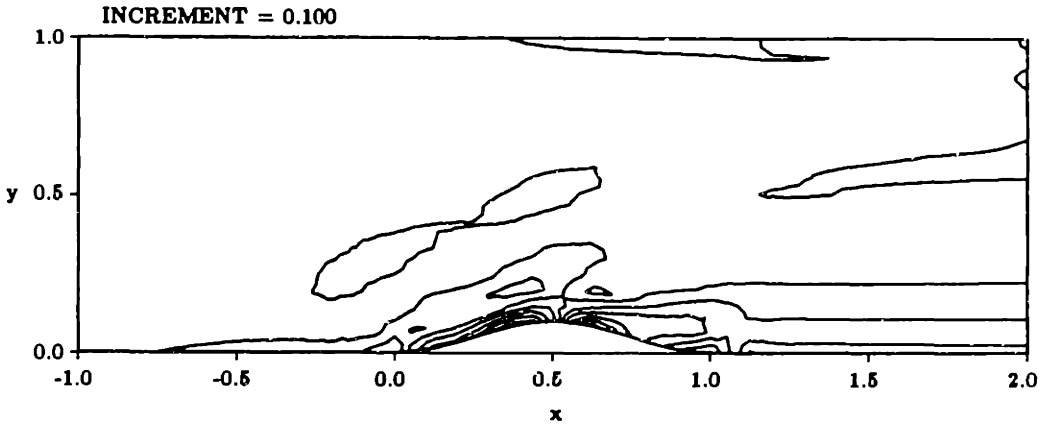


Figure 6.6: % total pressure loss contours for $\sin^2 x$ duct

6.3 Results of Numerical Study

For each of the numerical schemes described in Chapter 2 a numerical accuracy study was preformed as described in the proceeding section. For the Jameson schemes two different numerical smoothing methods are used as described in Chapter 3. The schemes described here are all shown mathematically to be second order accurate. The order of accuracy of the numerical smoothing has an effect on the accuracy of the numerical scheme. This effect will be described in this section.

In the plots of the numerical accuracy data shown in Figures 6.7 to 6.14 it can be seen that not all points lie on a straight line. For all points the smoothing coefficient is the same, and the effect of this constant coefficient varies as h varies. In general, the points lie on a line and a least squares fit to the points is a good approximation to find the slope of this line. When fitting a line through the data there is some error due to a limited number of points which are used. Some sets of data consist of three points and others consist of four points, but the error is obviously small and depends on the particular case. It is also possible for the numerical accuracy to be more than 2, which is the numerical accuracy of the basic scheme, since the truncation errors are not composed of terms exactly proportional to h^2, but will also contain higher order terms. This will be seen in the results presented here.

6.3.1 Effect of Numerical Smoothing on Accuracy

Two numerical fourth difference freestream smoothing methods were described in Chapter 3 for both the triangular and quadrilateral schemes. The first was a low-accuracy method and the second was a high-accuracy method, both of which are conservative. In the accuracy studies only freestream smoothing which is applied throughout the flow field was used. This is sufficient for stability since the problem used for the accuracy study was designed to have sufficiently smooth flow. For the Ni schemes only the high-accuracy smoothing was used. However, to illustrate the effect of smoothing on accuracy, both freestream methods were used for the Jameson schemes.

With the high-accuracy smoothing, the data for both the triangular and quadrilateral schemes show that these schemes are second order accurate on both regular and irregular meshes. This effect can be seen for the triangular Ni scheme by referring to Figures 6.7 and 6.8, and for the quadrilateral Ni scheme by referring to Figures 6.9 and 6.10. For the triangular Jameson scheme this can be seen by referring to Figures 6.11 and 6.12, and for the quadrilateral Jameson scheme by referring to Figures 6.13 and 6.14. Second order accuracy is also achieved for smoothing coefficient values larger than the values used here.

For the low-accuracy smoothing the story is different than when using the high-accuracy smoothing. The Jameson schemes require some smoothing to be stable. With a coefficient which is large enough to barely provide stability, the Jameson schemes may retain their second order accuracy. It is important to note that the flow field here is very smooth, and the value of the smoothing coefficient which barely provides stability here is smaller than it would be in most flow fields. With a smoothing coefficient a factor of five times larger the accuracy drops. In the first case the error due to the numerical scheme which is second order accurate dominates over the error due to the numerical smoothing. The order of accuracy which is calculated is effectively the order of accuracy of the numerical scheme without the smoothing. As the mesh resolution increases, eventually the smoothing error will dominate and the order of accuracy will deteriorate. Formally, order of accuracy is concerned with the limit $h \rightarrow 0$, and in this limit it will be less than second order. The beginning of this effect can be seen in Figure 6.22. As the smoothing coefficient is increased the accuracy is reduced and the error due to the numerical smoothing begins to play a role in the accuracy. The accuracy decreases for the regular as well as irregular meshes with the low-accuracy smoothing due to effects on the boundary. Dissipative errors occur on the boundary because the solid wall boundary condition which is imposed is not appropriate for the low-accuracy smoothing. The error can be estimated by noting that the smoothing attempts to enforce $\frac{\partial u}{\partial n} = 0$, and so it will create a numerical boundary layer at least one cell wide with an error that is of order $h(\frac{\partial u}{\partial n})_{\text{inviscid}} = O(h)$. The numerical boundary layer has $O(1/h)$ nodes compared to the total number of nodes which is $O(1/h^2)$, so a lower bound on the order of magnitude of the root mean square error is

$$error = O\left(\sqrt{\frac{1}{h}h^2/\frac{1}{h^2}}\right) = O(h^{3/2}) \tag{6.5}$$

This explains why the numerical error reduces to approximately $O(h^{3/2})$. The effect of increasing the smoothing coefficient can be seen for the triangular Jameson scheme on a regular mesh by referring to Figure 6.15, and Figure 6.16 for high and low smoothing coefficients respectively. The lowest smoothing coefficient which produces stability for an irregular triangular mesh would not give second order accuracy as shown in Figure 6.18. For the quadrilateral Jameson scheme this effect can be seen for both regular and irregular meshes by referring to Figures 6.19 and 6.20, and Figure 6.21 and 6.22 for high and low smoothing coefficients respectively.

The numerical smoothing formulation used is very important. The accuracy of the scheme will only be as good as the least accurate component, be that the basic numerical scheme or the numerical smoothing. If the numerical smoothing is only first order accurate, then the accuracy of the numerical scheme will be contaminated by the smoothing.

6.3.2 Conclusions

In Figure 6.23 the results of the numerical accuracy study are summarized. For the high-accuracy numerical smoothing, both the Ni and Jameson schemes for both quadrilateral and triangular meshes are second order accurate. This applies for both regular and irregular meshes. On relatively coarse meshes, when the smoothing coefficient for the low-accuracy smoothing is at the minimum required to keep a scheme stable the Jameson schemes for both quadrilateral and triangular meshes are approximately second order accurate. When this coefficient is above this minimum or when much finer meshes are used the accuracy of the basic numerical scheme is contaminated and the accuracy drops below second order.

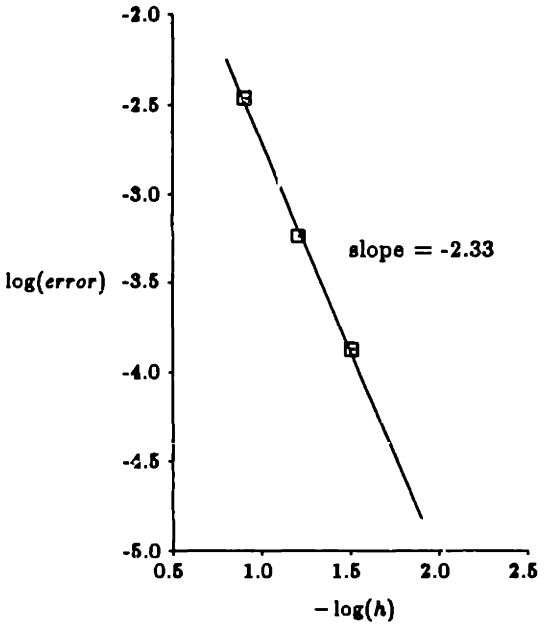


Figure 6.7: Order of accuracy for triangular Ni scheme with high-accuracy smoothing on a regular mesh

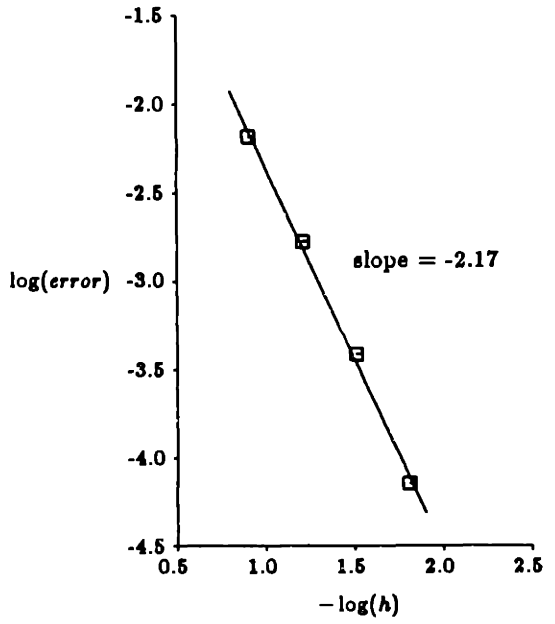


Figure 6.8: Order of accuracy for triangular Ni scheme with high-accuracy smoothing on an irregular mesh

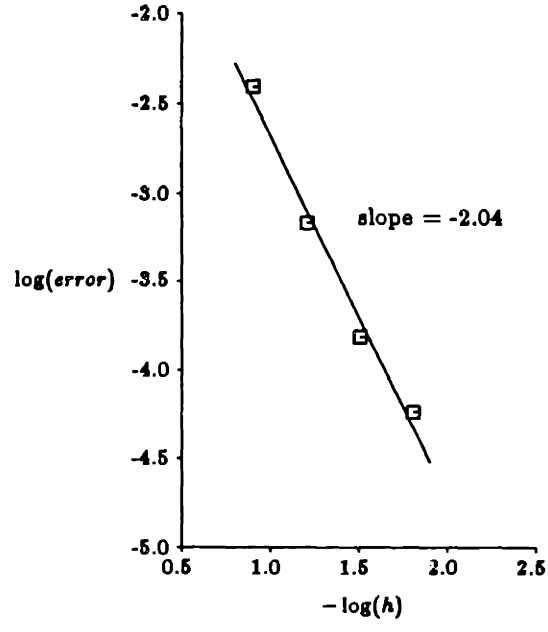


Figure 6.9: Order of accuracy for quadrilateral Ni scheme with high-accuracy smoothing on a regular mesh

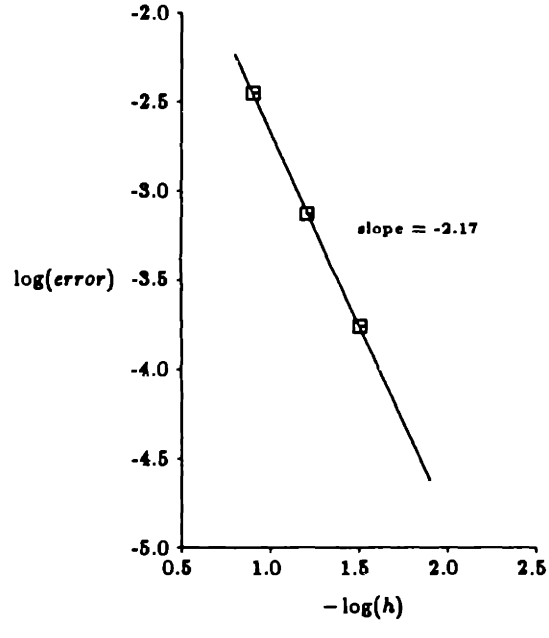


Figure 6.10: Order of accuracy for quadrilateral Ni scheme with high-accuracy smoothing on an irregular mesh

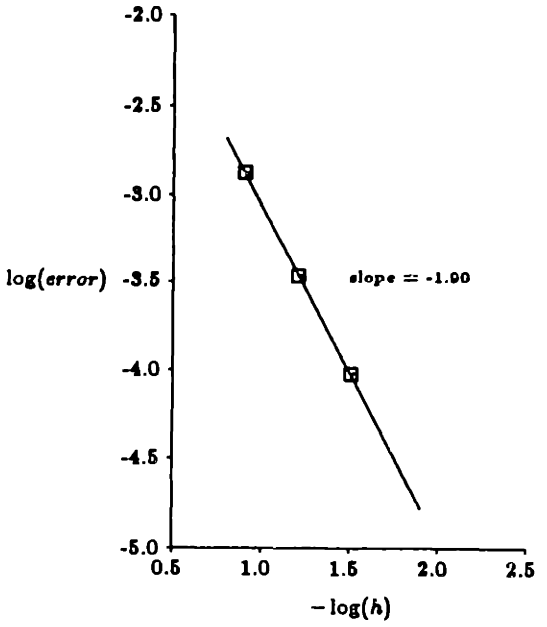


Figure 6.11: Order of accuracy for triangular Jameson scheme with high-accuracy smoothing on a regular mesh

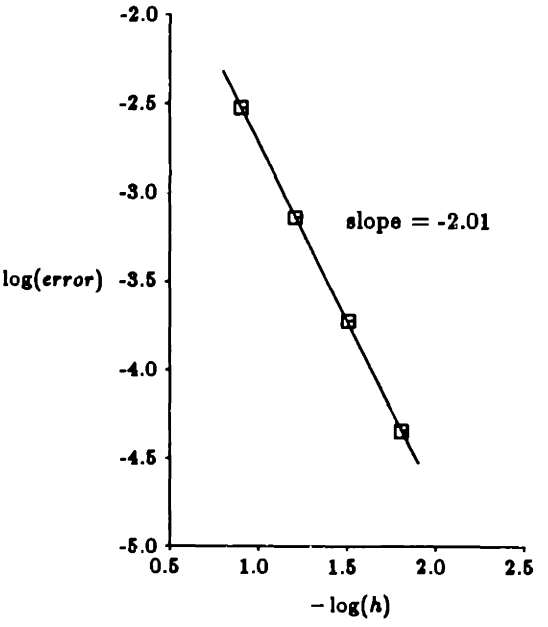


Figure 6.12: Order of accuracy for triangular Jameson scheme with high-accuracy smoothing on an irregular mesh

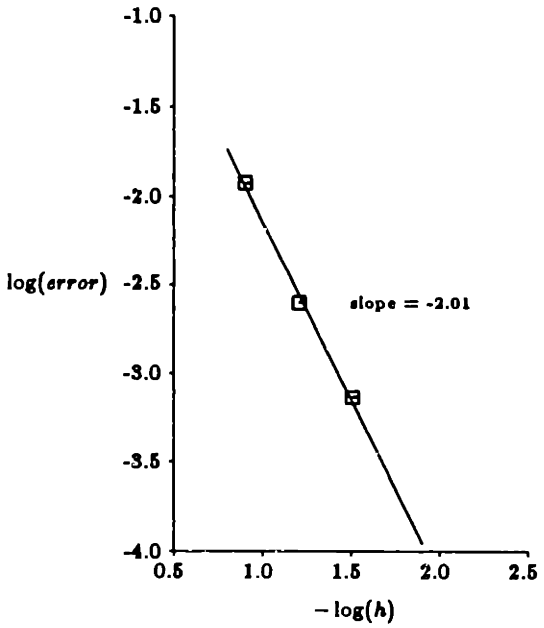


Figure 6.13: Order of accuracy for quadrilateral Jameson scheme with high-accuracy smoothing on a regular mesh

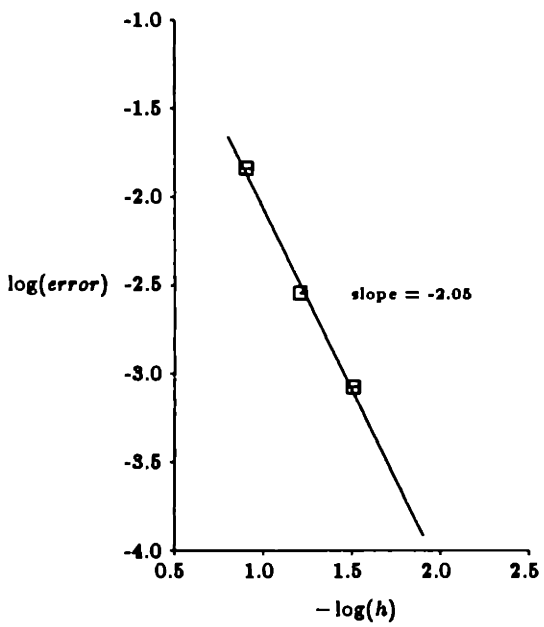


Figure 6.14: Order of accuracy for quadrilateral Jameson scheme with high-accuracy smoothing on an irregular mesh

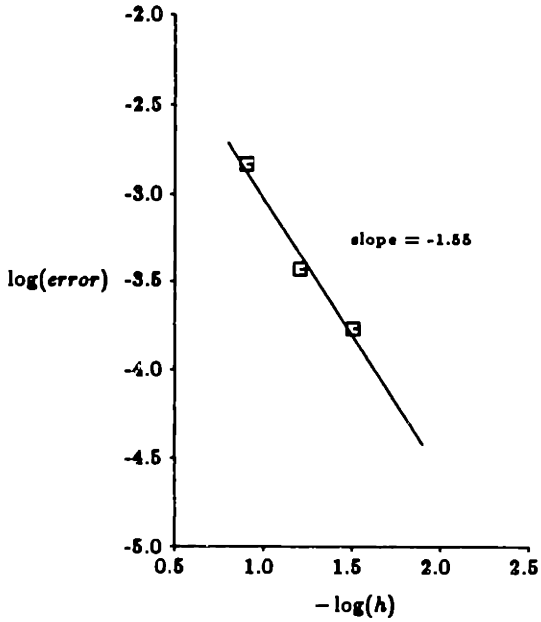


Figure 6.15: Order of accuracy for triangular Jameson scheme with low-accuracy smoothing on a regular mesh, smoothing coefficient = 0.0005

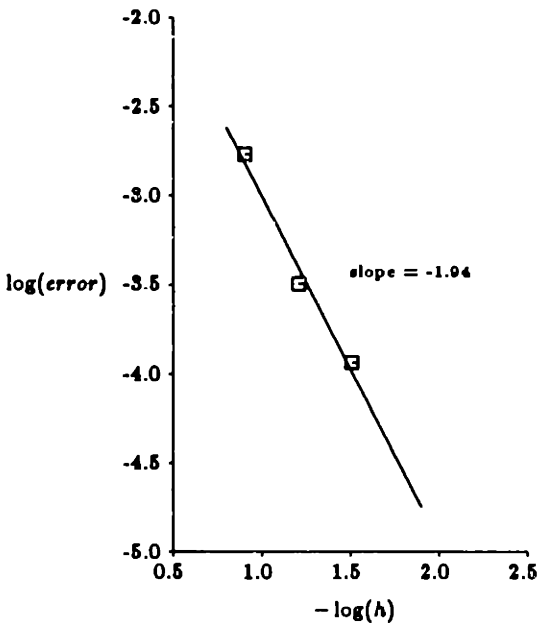


Figure 6.16: Order of accuracy for triangular Jameson scheme with low-accuracy smoothing on a regular mesh, smoothing coefficient = 0.0001

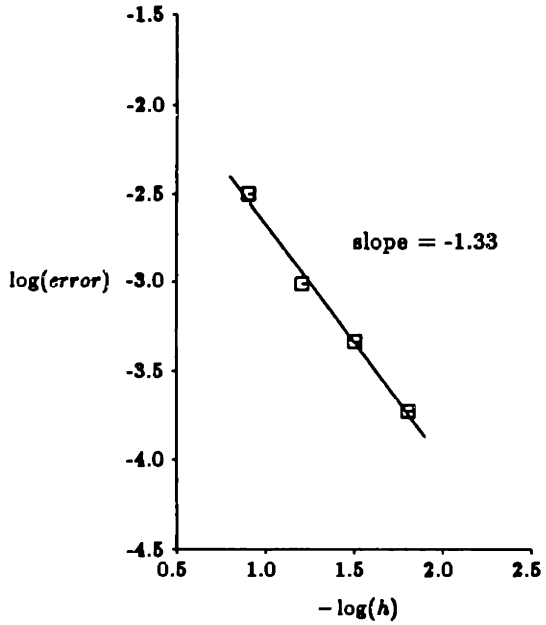


Figure 6.17: Order of accuracy for triangular Jameson scheme with low-accuracy smoothing on an irregular mesh, smoothing coefficient = 0.0005

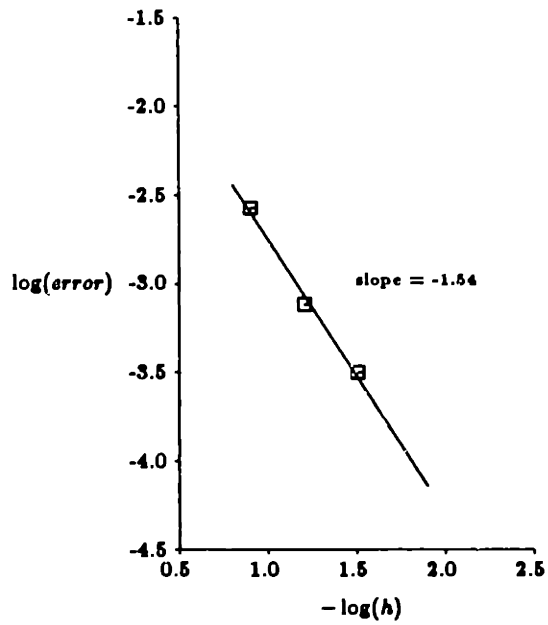


Figure 6.18: Order of accuracy for triangular Jameson scheme with low-accuracy smoothing on an irregular mesh, smoothing coefficient = 0.0001

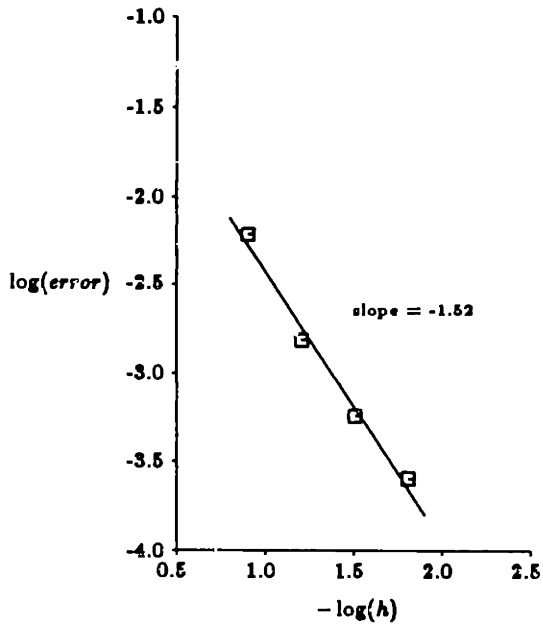


Figure 6.19: Order of accuracy for quadrilateral Jameson scheme with low-accuracy smoothing on a regular mesh, smoothing coefficient = 0.005

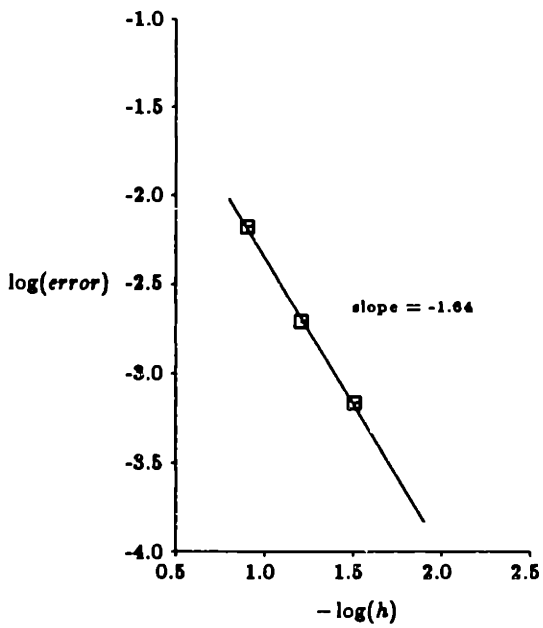


Figure 6.20: Order of accuracy for quadrilateral Jameson scheme with low-accuracy smoothing on an irregular mesh, smoothing coefficient = 0.005

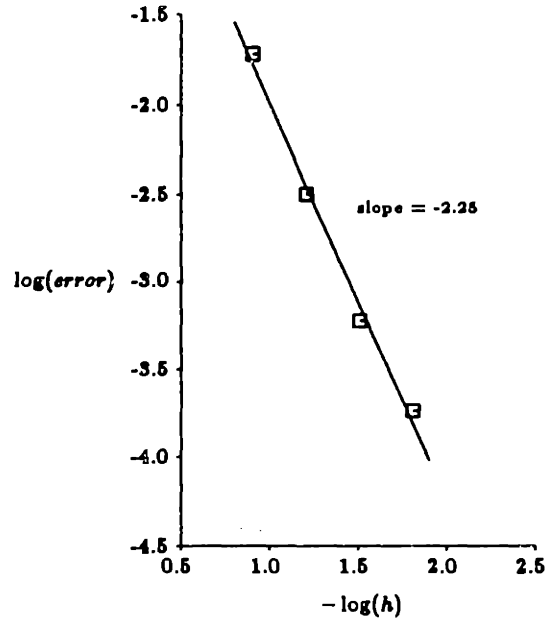


Figure 6.21: Order of accuracy for quadrilateral Jameson scheme with low-accuracy smoothing on a regular mesh, smoothing coefficient = 0.001

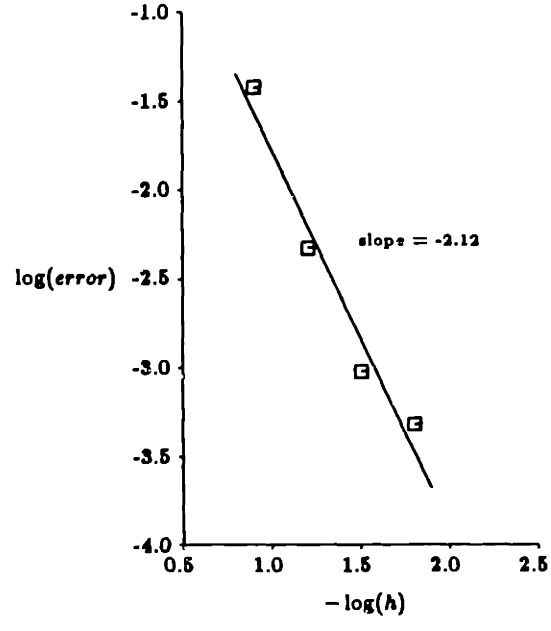


Figure 6.22: Order of accuracy for quadrilateral Jameson scheme with low-accuracy smoothing on an irregular mesh, smoothing coefficient = 0.001

	Numerical Accuracy	Smoothing Coef.
Triangular Ni scheme on regular mesh with high-accuracy smoothing	2.33	0.0025
Triangular Ni scheme on irregular mesh with high-accuracy smoothing	2.17	0.0025
Triangular Jameson scheme on regular mesh with low-accuracy smoothing	1.55 1.94	0.0005 0.0001
Triangular Jameson scheme on irreguiar mesh with low-accuracy smoothing	1.33 1.54	0.0005 0.0001
Triangular Jameson scheme on regular mesh with high-accuracy smoothing	1.90	0.0025
Triangular Jameson scheme on irregular mesh with high-accuracy smoothing	2.01	0.0025
Quadrilateral Ni scheme on regular mesh with high-accuracy smoothing	2.04	0.005
Quadrilateral Ni scheme on irregular mesh with high-accuracy smoothing	2.17	0.005
Quadrilateral Jameson scheme on regular mesh with low-accuracy smoothing	1.52 2.25	0.005 0.001
Quadrilateral Jameson scheme on irregular mesh with low-accuracy smoothing	1.64 2.12	0.005 0.001
Quadrilateral Jameson scheme on regular mesh with high-accuracy smoothing	2.01	0.005
Quadrilateral Jameson scheme on irregular mesh with high-accuracy smoothing	2.05	0.005

Figure 6.23: Numerical order of accuracy of numerical schemes

Chapter 7
Adaptation

In order to reduce the computational time required to find the solution to a problem, spatial adaptation can be used. Spatial adaptation is a method which places small cells where the physical characteristic length is small, such as in shocks, and large cells where the physical characteristic length is large. Since the solution is not known before hand, one method for creating an adapted mesh is to start with a course mesh and to divide cells when the physical characteristic length becomes larger than some prescribed value.

7.1 Adaptation Criteria

The features which will be resolved by the adaptation procedure depend on the parameter used to determine when to adapt. Dannenhoffer [3] discusses the merit of different parameters, and concludes that the first difference of density provides a parameter which will resolve both shocks and slip lines in isoenergetic flow, and is inexpensive to calculate. If the division will take place on a cell basis, the first difference of density is found for all the cells, and if the division will take place on a face basis, it is found for all the faces. All cells or faces whose first difference of density is greater than some reference value are divided. The mean and standard deviation of the first difference of density are found over all the cells or faces, and the reference value will be the average plus a factor between 0 and 1 of the standard deviation. It was found that a factor of 0.4 produces good results.

7.2 Quadrilateral Adaptation

For quadrilateral cells the division process typically is made on a cell basis. Several cells are chosen for adaptation and are divided into four smaller cells by adding a node in the center of each chosen course cell and in the center of each of its faces. This process produces an interface between a region of course cells and a region of fine cells. This interface is a result of the regularity in a quadrilateral mesh. An example of such an interface is shown in Figure 7.1. Methods have been developed for dealing with the problem of interfaces [3], but will not be discussed in detail here. Part of the problem is the need to store data on the adaptation history and interface location and to deal with interfaces with special methods in the numerical solver. Results produced by Shapiro [20] for a quadrilateral mesh are shown in Figure 7.2. The computations by Shapiro were done using a cell vertex finite element scheme which is essentially the same as the quadrilateral Jameson scheme discussed here. The three meshes shown have 192, 609 and 1647 cells.

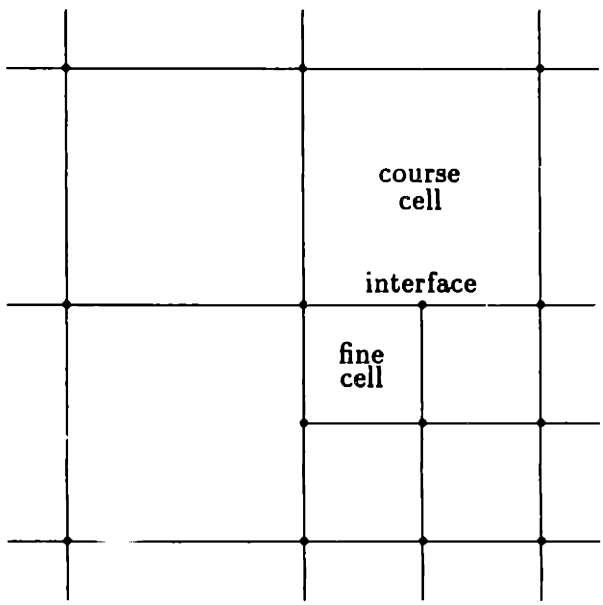


Figure 7.1: Manner of quadrilateral cell division with interfaces

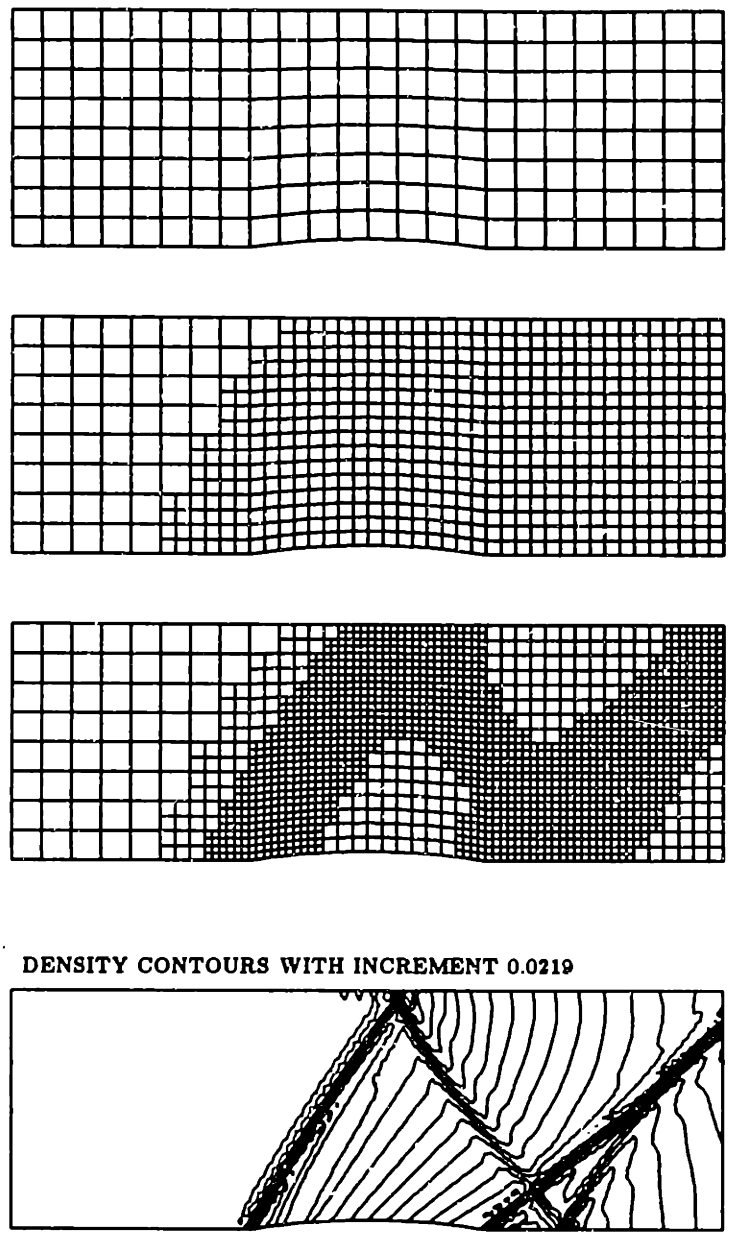


Figure 7.2: Levels of adaptation with a quadrilateral mesh

7.3 Triangular Adaptation

For triangular cells the irregularity in the mesh allows a cell to be divided into smaller cells without creating interfaces. This means that once a cell is divided, no added information must be stored and no special cases are required in the solver. The order of accuracy of the numerical scheme as well as consistency are also preserved. To illustrate this, a simple method for dividing triangular cells was developed.

The division process used here for triangular cells is face based. It is possible for a cell to have one, two or three faces selected for division. To create a smooth mesh, when two sides of a triangle are chosen for division the third is also divided. The manner of cell division is shown in Figure 7.3. Other processes of cell division have been devised which are based on cell division or Delaunay triangulation. These methods would probably create a better adapted mesh, but were not investigated here since the sole objective was to demonstrate the relative ease with which triangular adaptation can be implemented. A simple triangular mesh was used as a base mesh and the meshes after each level of adaptation are shown in Figure 7.4. The triangular Jameson scheme was used to compute the solution. The three meshes shown have 384, 1001 and 2228 cells.

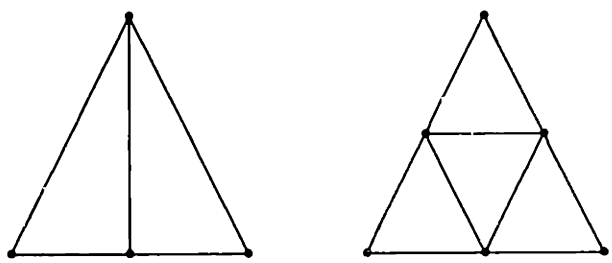


Figure 7.3: Manner of triangular cell division

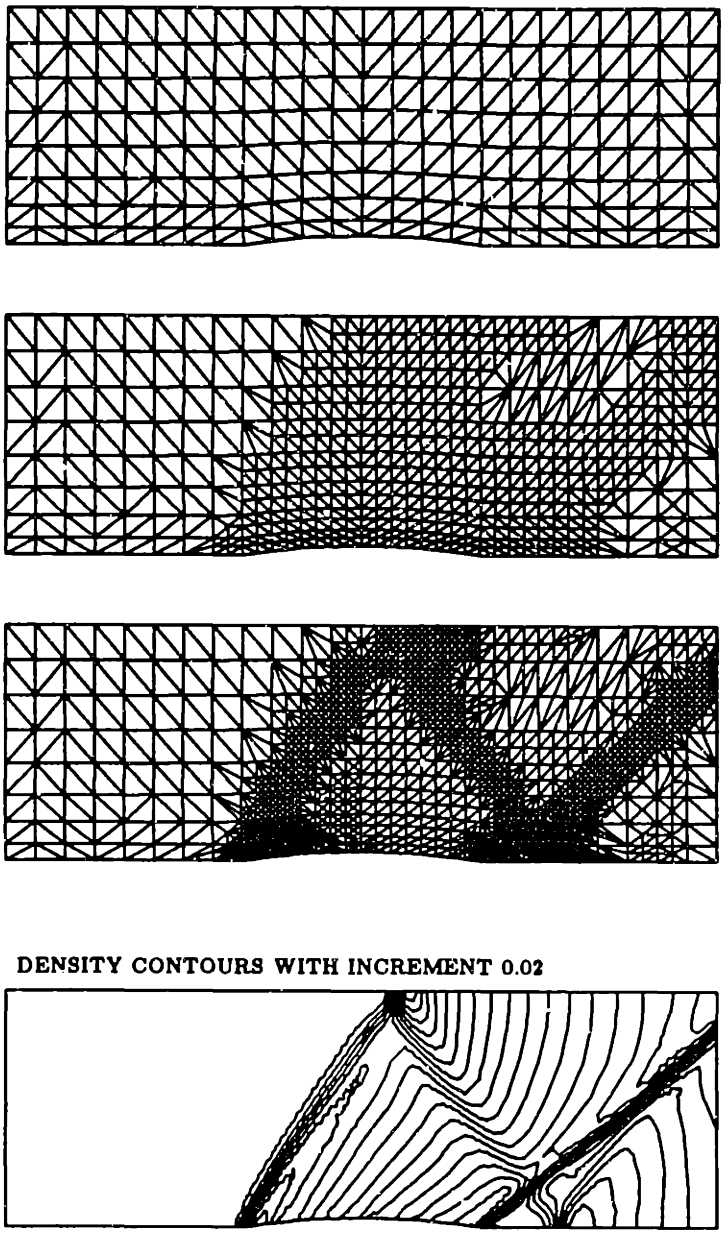


Figure 7.4: Levels of adaptation with a triangular mesh

Chapter 8
Conclusions

This work sheds some light on the controversy in the CFD community over the advantages of quadrilateral and triangular cell schemes. Two different node based finite volume schemes for the Euler equations were studied: a Jameson scheme and a Ni scheme.

A triangular Jameson scheme flow solver was programmed as described by Mavriplis [13] and a similar quadrilateral Jameson scheme was also programmed. A quadrilateral Ni scheme flow solver as described by Ni [15] and Giles [7] was also programmed and a triangular extension of this scheme was developed. Both schemes on triangular and quadrilateral meshes produce similar results. In fact, the quality of the results is more dependent on the numerical smoothing used than the flow solver.

Two different freestream smoothing techniques are examined and implemented with the numerical schemes. Numerical smoothing is required for stability of the Jameson schemes and to reduce undesirable modes in all the schemes. The freestream smoothing consists of a fourth difference of the state vector variables which is formed by taking the second difference of a second difference. Two different methods for finding a second difference are examined, one of which has low-accuracy and the other which has high-accuracy but requires more time to compute. The result is that two conservative fourth differences for numerical smoothing can be created one of which has low-accuracy and another which has high-accuracy, but is more expensive. The low-accuracy smoothing contains no information about the locations of the nodes in the mesh, and as a result produces less smooth solutions on irregular meshes and poorer solutions at solid walls for triangular meshes than the high-accuracy smoothing. The problems with the solid wall boundaries are caused by imposing boundary conditions on the flow field which

cause the low-accuracy numerical smoothing to be convective.

A mathematical study shows that these node based numerical schemes for both quadrilateral and triangular meshes are second order accurate. This is verified here by a numerical accuracy study. The effect of numerical smoothing on the accuracy of the scheme can be large. A numerical smoothing method which is first order accurate can reduce this second order accuracy to first order, since the scheme is only as accurate as the lowest order accurate component. This effect is demonstrated here. When using the low-accuracy smoothing, the accuracy drops below second order due to the lower accuracy of the smoothing. The high-accuracy smoothing allows the schemes to retain their second order accuracy. Therefore both the quadrilateral and triangular schemes are second order accurate if the proper numerical smoothing is used.

Spatial adaptation reduces the computational time required to compute a solution by placing small cells where the physical characteristic length of the flow is small and larger cells where the physical characteristic length of the flow is large. One of the advantages of triangles is the ability to divided cells such that no special interfaces exist. Quadrilateral spatial adaptation, however, requires interfaces between regions of fine and coarse cells. While with triangular adaptation both second order accuracy and conservation can be retained in the adapted region, the quadrilateral adaptation can retain only one of these qualities at the interfaces. In all, spatial adaptation is better suited for triangular meshes, but methods have been developed for quadrilateral meshes which are quite satisfactory.

Triangular meshes by nature of their irregularity allow for mesh generation about complex geometries. The mesh generation process may be quite complicated, but can be made to handle arbitrary several types of geometries and is extendable to three dimensions. The ability to be quite general can be advantageous when compared with the complexities involved in block meshes or overlapping meshes which require user input when generated which is harder to visualize in three dimensions.

A method to define an irregular mesh which is referred to as a pointer system must be used for triangular meshes. Quadrilateral meshes require pointer systems only

when adaptation or other special mesh operations are performed. A pointer system describes the interconnection between elements of a mesh and can easily deal with any irregularities or changes in the mesh structure. In general, pointer systems are beginning to be used for both triangular and quadrilateral meshes, and don't require much more information than the traditional (i, j) method of defining a mesh and allow for longer vector lengths and more concurrency for a parallel processing computer.

This study has focussed on the basic development, performance and accuracy of quadrilateral and triangular mesh schemes. The triangular schemes perform as well as the quadrilateral schemes in both quality of results and accuracy provided an appropriate numerical smoothing technique is used. This result justifies the continued research in the use of triangular mesh schemes. More work must be done in the field of triangular mesh generation and adaptation to bring these computational methods for triangular meshes to a similar level of understanding as on quadrilateral meshes. In three dimensions the use of tetrahedra as opposed to hexahedra significantly increases the ability to produce meshes around complex three dimensional geometries and for this reason more than any other justifies the need to continue research in this area.

Bibliography

[1] T. J. Baker. *Three Dimensional Mesh Generation by Triangulation of Arbitrary Point Sets.* AIAA-87-1124, June 1987.

[2] J. F. Dannenhoffer and J. R. Baron. *Robust Grid Adaptation for Complex Transonic Flows.* AIAA-86-0495, January 1986.

[3] J. F. Dannenhoffer III. *Grid Adaptation for Complex Two-Dimensional Transonic Flows.* ScD thesis, Massachusetts Institute of Technology, August 1987.

[4] Delaunay. "Sur la Sphere vide." *Bull. Acad. Science USSR VII: Class Scil, Mat. Nat.*, 793–800, 1934.

[5] M. B. Giles. *Accuracy of Node-Based Solutions on Irregular Meshes.* 11^{th} International Conference on Numerical Methods in Fluid Dynamics, June 1988.

[6] M. B. Giles. *Energy Stability Analysis of Multi-Step Methods on Unstructured Meshes.* Technical Report CFDL-TR-87-1, M.I.T., March 1987.

[7] M. B. Giles. *UNSFLO: A Numerical Method for Unsteady Inviscid Flow in Turbomachinery.* Technical Report CFDL-TR-86-6, M.I.T., December 1986.

[8] D. G. Holmes, S. A. Lamson, and S. D. Connell. *Quasi-3D Solutions for Transonic, Inviscid Flows by Adaptive Triangulation.* ASME Gas Turbine Meeting, June 1988.

[9] A. Jameson. "Solution of the Euler Equations by a Multigrid Method." *Applied Math. and Computation*, 13:327–356, June 1983.

[10] A. Jameson, T. J. Baker, and N. P. Weatherill. *Calculation of Inviscid Transonic Flow over a Complete Aircraft.* AIAA-86-0103, January 1986.

[11] A. Jameson, W. Schmidt, and E. Turkel. *Numerical Solutions of the Euler Equations by a Finite Volume Method Using Runge-Kutta Time Stepping Schemes.* AIAA-81-1259, June 1981.

[12] R. Löhner, K. Morgan, J. Peraire, and O. C. Zienkiewicz. *Finite Element Methods for High Speed Flows.* AIAA-85-1531, July 1985.

[13] D. Mavriplis. *Solution of the Two-Dimensional Euler Equations on Unstructured Triangular Meshes.* PhD thesis, Princeton University, June 1987.

[14] D. Mavriplis and A. Jameson. *Multigrid Solution of the Two-Dimensional Euler Equations on Unstructured Triangular Meshes.* AIAA-87-0353, January 1987.

[15] R.-H. Ni. *A Multiple-Grid Scheme for Solving the Euler Equations.* AIAA 181-025R, June 1981.

[16] J. Peraire, M. Vahdeti, K. Morgan, and O. C. Zienkiewicz. "Adaptive Remeshing for Compressible Flow Computations." *Journal of Computational Physics*, 72:449–466, October 1987.

[17] Robert D. Richtmyer and K. W. Morton. *Difference Methods for Initial-Value Problems*, pp. 311–313. John Wiley & Sons, second edition, 1967.

[18] T. W. Roberts. *Euler Equation Computations for the Flow Over a Hovering Helicopter Rotor.* PhD thesis, Massachusetts Institute of Technology, November 1986.

[19] P. Roe. *Error Estimates for Cell-Vertex Solutions of the Compressible Euler Equations.* Technical Report 87-6, ICASE, 1987.

[20] R. Shapiro. *An Adaptive Finite Element Solution Algorithm for the Euler Equations.* PhD thesis, Massachusetts Institute of Technology, May 1988.

[21] J. Steger and R. Sorenson. "Automatic Mesh-Point Clustering Near a Boundary in Grid Generation with Elliptic Partial Differential Equations." *Journal of Computational Physics*, 33:405–410, 1979.

[22] W. J. Usab III. *Embedded Mesh Solutions of the Euler Equation Using a Multiple-Grid Method.* PhD thesis, Massachusetts Institute of Technology, December 1983.

Appendix A
Computer Code

This appendix contains the sources for the flow solvers for the quadrilateral Ni scheme, the quadrilateral Jameson scheme, the triangular Ni scheme and the triangular Jameson scheme. Also included is the code for the mesh generator and the plotting program. The subroutines from the graphics package GRAFIC are not included.

A.1 Triangular Schemes

A.1.1 Common Files

This is file TRI.INC which includes many declarations and common block statements and is included in all subroutines for the triangular schemes.

```
parameter Maxnodes = 5000
parameter Maxfaces = 15141      ! = 3*Maxnodes + 2*sqrt(Maxnodes)
parameter Maxcells = 10000      ! = 2*Maxnodes
parameter Maxedges = 424        ! = 6*sqrt(Maxnodes)

integer Nmax                    !number of nodes
integer Fmax                    !number of faces (including edges)
integer Cmax                    !number of cells (including edge cells)
integer Imax, Omax              !number of inlet and outlet nodes
integer Wmax                    !number of wall faces
integer Emax                    !number of edge faces
integer Pmax                    !number of periodic nodes
integer ENmax                   !number of edge nodes
integer tnode, bnode, inode     !location of diff. types of edge nodes

real pitch                      !blade pitch
real pout                       !outlet pressure for blades
real Sinl                       !tan of inlet flow angle
real x(Maxnodes)                !x values of nodes
real y(Maxnodes)                !y values of nodes
integer eface(Maxedges)         !array of edge faces
integer enode(Maxedges)         !array of edge nodes
real senode(Maxedges)           !sin and cos at edge nodes
real cenode(Maxedges)
integer face(Maxfaces,6)        !array of faces
integer cell(Maxcells,6)        !array of cells
integer innode(Maxedges)        !inlet nodes
```

```
      integer outnode(Maxedges)        !outlet nodes
      integer pnode(Maxedges,2)        !periodic nodes
      integer NCcolor(60)              !number of cells colored each color
      integer Ccolormax                !max number of colors used
      integer NFcolor(60)              !number of faces colored each color
      integer Fcolormax                !max number of colors used
      integer NEcolor(10)              !number of edges colored each color
      integer E1colormax, E2colormax   !max number of colors used

      common /tri/ Imax, Omax, Wmax, Pmax
      common /tri/ Nmax, Fmax, Cmax, Emax
      common /tri/ ENmax, tnode, bnode, inode
      common /tri/ pitch, pout, Sinl
      common /tri/ x, y, eface, enode, senode, cenode, face, cell
      common /tri/ innode, outnode, pnode
      common /color/ NCcolor, Ccolormax, NFcolor, Fcolormax
      common /color/ NEcolor, E1colormax, E2colormax

      real Minl                        !inlet Mach number
      integer Maxiter                  !max number of iterations
      real CFL                         !CFL number
      real epsicoef, eps2              !smoothing coef
      real sigE, sigV                  !smoothing coef
      real vol(Maxcells)               !for vorticity smoothing
      real durms                       !rms difference in state vector
      real U(4,0:Maxnodes)             !state vector
      real dU(4,Maxnodes)              !change in state vector at nodes
      real Uc(4,Maxcells)              !state vector at cells
      real dUc(4,0:Maxcells)           !change in state vector at cells
      real F(4,Maxnodes)               !flux vectors at nodes
      real G(4,Maxnodes)
      real areaN(0:Maxnodes)           !control area around each node
      real areaC(0:Maxcells)           !area of each cell
      real deltN(0:Maxnodes)           !time step at each node (Jameson)
                                       ! or cell (Ni)
      real deltC(0:Maxcells)           !time step at each cell
      real dis(4,Maxnodes)             !dissipation at each node
      real flux(4,0:Maxnodes)          !flux at each node

      common /flo/ Minl, durms, Maxiter, CFL
      common /flo/  epsicoef, eps2, sigE, sigV, vol
      common /flo/ U, dU, Uc, dUc
      common /flo/ F, G
      common /flo/ deltC, deltN, areaC, areaN, dis, flux

      parameter gam = 1.4
      parameter gam1 = 0.4
```

This file contains the subroutines for input and output and is linked with all the triangular schemes.

```
c*****************************************************************
c*                                                               *
c*      write or read grid data from file                        *
c*                                                               *
c*****************************************************************

      subroutine gridio(process)
      implicit none

      include 'TRI.INC'
      integer process                        !read or write?
      integer i, NN, FF, CC, EF, EN, col     !pointers
      integer IN, ON, WN, PN

      if (process.eq.1) then
```

```
c** read in data from file
          open(unit=30, status='unknown', form='unformatted')

          read(30) Nmax, (x(NN),NN=1,Nmax), (y(NN),NN=1,Nmax)
          read(30) Fmax, ((face(FF,i),i=1,6),FF=1,Fmax)
          read(30) Wmax, Emax, (eface(EF),EF=1,Emax)
          read(30) tnode, bnode, inode, ENmax, (enode(EN),EN=1,ENmax)
          read(30) (senode(EN),EN=1,ENmax), (cenode(EN),EN=1,ENmax)
          read(30) Imax, (innode(IN),IN=1,Imax)
          read(30) Omax, (outnode(ON),ON=1,Omax)
          read(30) Pmax, ((pnode(PN,i),i=1,2),PN=1,Pmax)
          read(30) Cmax, ((cell(CC,i),i=1,6),CC=1,Cmax)
          read(30) Ccolormax, (NCcolor(col),col=1,Ccolormax)
          read(30) Fcolormax, (NFcolor(col),col=1,Fcolormax)
          read(30) E1colormax, E2colormax
          read(30) (NEcolor(col),col=1,E2colormax)
          read(30) pitch

          close(unit=30)

       else if (process.eq.0) then

c*  write data to file
          open(unit=30, status='unknown', form='unformatted')

          write(30) Nmax, (x(NN),NN=1,Nmax), (y(NN),NN=1,Nmax)
          write(30) Fmax, ((face(FF,i),i=1,6),FF=1,Fmax)
          write(30) Wmax, Emax, (eface(EF),EF=1,Emax)
          write(30) tnode, bnode, inode, ENmax, (enode(EN),EN=1,ENmax)
          write(30) (senode(EN),EN=1,ENmax), (cenode(EN),EN=1,ENmax)
          write(30) Imax, (innode(IN),IN=1,Imax)
          write(30) Omax, (outnode(ON),ON=1,Omax)
          write(30) Pmax, ((pnode(PN,i),i=1,2),PN=1,Pmax)
          write(30) Cmax, ((cell(CC,i),i=1,6),CC=1,Cmax)
          write(30) Ccolormax, (NCcolor(col),col=1,Ccolormax)
          write(30) Fcolormax, (NFcolor(col),col=1,Fcolormax)
          write(30) E1colormax, E2colormax
          write(30) (NEcolor(col),col=1,E2colormax)
          write(30) pitch

          close(unit=30)

       endif

       return
       end



c*****************************************************************
c*                                                               *
c*      write or read flow data from file                        *
c*                                                               *
c*****************************************************************

       subroutine flowio(process)
       implicit none

       include 'TRI.INC'
       integer process                !read or write?
       integer i, NN                  !pointers


       if (process.eq.1) then

c** read in data from file
          open(unit=25, status='unknown', form='unformatted')

          read(25) Minl
          read(25) Nmax, ((U(i,NN),i=1,4),NN=1,Nmax)
```

```
            close(unit=25)

         else if (process.eq.0) then

c*  write data to file
            open(unit=25, status='unknown', form='unformatted')

            write(25) Minl
            write(25) Nmax, ((U(i,NN),i=1,4),NN=1,Nmax)

            close(unit=25)

         endif

         return
         end



c*********************************************************************
c*                                                                   *
c*      read input data from file                                    *
c*                                                                   *
c*********************************************************************

         subroutine input
         implicit none

         include 'TRI.INC'
         real pi                          !the one and only

         pi = 3.14159

         open(unit=20, status='old')

         read(20,*) Maxiter, CFL
         read(20,*) Minl, Sinl, pout
         read(20,*) sigE, sigV, epsicoef, eps2
         close (unit=20)

         pout = pout/gam
         Sinl = tan(pi*Sinl/180.0)

         return
         end
```

A.1.2 Mesh Generator

This is file GRID.INC which includes many declarations and common block statements for the mesh generator.

```
         parameter Maxdim = 200

         real xx(0:Maxdim,0:Maxdim)       !x coordinate of grid points
         real yy(0:Maxdim,0:Maxdim)       !y coordinate of grid points
         integer Nx                       !number cells on x axis
         integer Ny                       !  "      "   "  y  "
         integer ILE, ITE                 !leading and trailing edge of blade
```

```
      common /grid/ xx, yy, Nx, Ny
      common /grid/ ILE, ITE


      parameter IBX=251

      real GSINL, GSOUT, CHINL, CHOUT
      integer II, JJ
      real XB(IBX), XPB(IBX), YB(IBX), YPB(IBX), SB(IBX)
      integer IIB, IBLE
      real SBLE, SBLOLD, SS, SP
      real XMIN, XMAX, YMIN, XLE, YLE, XPOS(Maxdim), YPOS(Maxdim)
      integer NINL, NOUT, NBLD

      common /c01/ GSINL, GSOUT, CHINL, CHOUT, II, JJ,
     &             XB, XPB, YB, YPB, SB,
     &             IIB, IBLE, SBLE, SBLOLD, SS, SP,
     &             XMIN, XMAX, YMIN, XLE, YLE, XPOS, YPOS,
     &             NINL, NOUT, NBLD

c********************** FACE ARRAY **********************
c
c            J1              face(L,1) = K1
c           /|\              face(L,2) = K2
c          / | \             face(L,3) = J1
c         /  |  \            face(L,4) = J2
c        /   |   \           face(L,5) = J3
c  J4<  K2   |  K1   >J3     face(L,6) = J4
c       \     L     /
c        \    |    /
c         \   |   /
c          \  |  /
c           \|/
c            J2
c
c
c********************** CELL ARRAY **********************
c
c              J2            cell(K,1) = L1
c             /\             cell(K,2) = L2
c            /  \            cell(K,3) = L3
c        L3 /    \ L2        cell(K,4) = J1
c          /      \          cell(K,5) = J2
c         /   K    \         cell(K,6) = J3
c     J3 /__________\ J1
c            L1
c
c
c********************** EDGE ARRAY **********************
c
c              J3
c              /\
c             /  \
c            /    \
c           /  K1  \
c          /        \
c       J1/__________\J2     eface(EF) F
c            F(edge)
c
c*******************************************************


      program gridgen
      implicit none

      include 'TRI.INC'
      include 'GRID.INC'
      integer gtype                    !type of grid
      integer EN, EF, FF, NN, CC, col  !pointers
      integer ON, IN, PN
      integer i, j                     !pointers
```

```
      integer JJJ

c**  determine geometry
      type*, 'Type of grid '
      type*, '   1) Ni bump   '
      type*, '   2) blade'
      type 100
 100  format($,' selection = ')
      accept 110, gtype
 110  format(I)

      write(6,*) ' '
      write(6,*) 'Generating grid ...'
      write(6,*) ' '

      if (gtype.eq.1) then
c**  create rectangular mesh
         call rectangle
         pitch = 0.

      else if (gtype.eq.2) then
c** Read in, normalize and spline blade data
         call readin

c** Initialize grid
         call grinit

c** Fix up grid
         call ellip(Mexdim,Maxdim,II,JJ,JJJ,XX,YY,YPOS,XPOS)
         call improv

c** change pointers
         do i = 1, ii
            do j = 1, jj
               xx(i-1,j-1) = xx(i,j)
               yy(i-1,j-1) = yy(i,j)
            enddo
         enddo
         Nx = ii-1
         Ny = jj-1
      endif

c**  change rectangular mesh to triangular mesh
      call pointers
      call bpointers

c**  rearrange node numbers on edges
      call edgenumber

c**  color cells and faces
      call cellcolor
      call facecolor
      call edgecolor

c**  write out data to file
      call gridio(0)

      stop
      end




      subroutine rectangle
      implicit none

      include 'GRID.INC'

      real*4 yymax, xxmax, xxmin      !coordinates of grid boundaries
```

```
      real*4 delta_x                  !grid spacing in x direction
      real*4 delta_y                  !grid spacing in y direction
      integer n, i, j, kk, m          !counters
      real*4 num
      real*4 delta_xbot
      real*4 omega                    !relaxation constant for
                                      !interior point SLOR
      real*4 pi                       !the one and only
      real*4 tau                      !height of bump
      real*4 rr                       !radius of bump
      real*4 yc                       !rr-tau
      real*4 ang, delta_ang           !angle of bump,angle between nodes
      real*4 angplus
      real*4 delx, dely               !help define initial conditions
      real*4 alpha(Maxdim), beta(Maxdim) !coef. in modefied equation
      real*4 gamma(Maxdim)
      real*4 AA(Maxdim), BB(Maxdim)   !values for system of equations
      real*4 CC(Maxdim), DD(Maxdim)
      real*4 EPS                      !allowable error
      integer Niter                   !number of iterations so far
      real*4 error                    !largest error for an iteration
      real*4 x_xi(Maxdim), x_eta(Maxdim) !derivatives on boundary
      real*4 x_xi_xi(Maxdim), x_eta_eta(Maxdim)
      real*4 x_eta_xi(Maxdim)
      real*4 y_xi(Maxdim), y_eta(Maxdim)
      real*4 y_xi_xi(Maxdim), y_eta_eta(Maxdim)
      real*4 y_eta_xi(Maxdim)
      real*4 theta, AR                !angle and aspect ratio of edge cells
      real*4 R1, R2                   !part of source term
      real*4 Q1(Maxdim), P1(Maxdim)   !part of source term, function of xi
      real*4 a1(Maxdim), b1(Maxdim)   !alpha, beta, gamma on lower boundary
      real*4 c1(Maxdim)
      real*4 omega_P, omega_Q         !relaxation consts. for source terms
      real*4 Jacobi(Maxdim)           !Jacobian on the lower boundary
      real*4 Jacobi2(Maxdim)          !Jacobian squared in region
      real*4 xxi, yxi, xeta, yeta     !derivatives in region
      real*4 a, b                     !exponents in source terms

      pi = 3.14159

c** number of nodes on x and y axes
      xxmax = 2.
      xxmin = -1.
      yymax = 1.

      call Irequest('        Nx',Nx)
      call Irequest('        Ny',Ny)
      call Rrequest('       tau',tau)
c     call Rrequest('     omega',omega)
c     call Rrequest('   omega_P',omega_P)
c     call Rrequest('   omega_Q',omega_Q)
c     call Rrequest('         a',a)
c     call Rrequest('         b',b)
c     call Rrequest('        AR',AR)
c     call Rrequest('       EPS',EPS)

      omega = 1.
      a = 0.8
      b = 0.8
      AR = 0.5
      EPS = 0.0005

      if (tau.eq.0.) then
         omega_P = 0.
         omega_Q = 0.
      else if (tau.lt.0.) then
         AR = 1.
         omega_P = 0.02
         omega_Q = 0.02
      else
         omega_P = 0.02
         omega_Q = 0.02
```

```
      endif

      theta = .5*pi
      delta_x = (xxmax-xxmin)/real(Nx)

c** set initial and boundary conditions for x and y
      if (tau.gt.0.) then
         yc = 0.5*(-tau**2 + 0.25)/tau
         rr = sqrt(0.25 + yc**2)
         ang = asin(.5/rr)
         if (mod(Nx,3).ne.0) then
            num = real(Nx - mod(Nx,3))
            delta_xbot = (xxmax-xxmin)/num
            angplus = 2.*ang/(num/3. + real(mod(Nx,3)))
         else
            angplus = 2.*ang*delta_x
            delta_xbot = delta_x
         endif
         delta_ang = 0.
         do i = 0, Nx
            yy(i,Ny) = yymax
            xx(i,Ny) = xxmin + i*delta_x
            if (xx(i,Ny).le.0.) then
               xx(i,0) = xxmin + i*delta_xbot
               yy(i,0) = 0.
            else if (xx(i,Ny).gt.0. .and. xx(i,Ny).lt.1.) then
               delta_ang = delta_ang + angplus
               xx(i,0) = 0.5 - rr*sin(ang-delta_ang)
               yy(i,0) = rr*cos(ang-delta_ang) - yc
            else if (xx(i,Ny).ge.1.) then
               xx(i,0) = xxmin + (i-mod(Nx,3))*delta_xbot
               yy(i,0) = 0.
            endif
            delx = (xx(i,Ny) - xx(i,0))/real(Ny)
            dely = (yy(i,Ny) - yy(i,0))/real(Ny)
            do j = 1, Ny-1
               xx(i,j) = delx*j + xx(i,0)
               yy(i,j) = dely*j + yy(i,0)
            enddo
         enddo
      else if (tau.lt.0.) then
         delta_xbot = delta_x
         do i = 0, Nx
            yy(i,Ny) = yymax
            xx(i,Ny) = xxmin + i*delta_x
            if (xx(i,Ny).le.0.) then
               xx(i,0) = xxmin + i*delta_xbot
               yy(i,0) = 0.
            else if (xx(i,Ny).gt.0. .and. xx(i,Ny).lt.1.) then
               xx(i,0) = xxmin + i*delta_xbot
               yy(i,0) = -tau*(sin(pi*xx(i,0)))**2
            else if (xx(i,Ny).ge.1.) then
               xx(i,0) = xxmin + i*delta_xbot
               yy(i,0) = 0.
            endif
            delx = (xx(i,Ny) - xx(i,0))/real(Ny)
            dely = (yy(i,Ny) - yy(i,0))/real(Ny)
            do j = 1, Ny-1
               xx(i,j) = delx*j + xx(i,0)
               yy(i,j) = dely*j + yy(i,0)
            enddo
         enddo
      else if (tau.eq.0) then
         delta_y = yymax/real(Ny)
         do i = 0, Nx
            do j = 0, Ny
               xx(i,j) = i*delta_x + xxmin
               yy(i,j) = j*delta_y
            enddo
         enddo
      endif
```

```
c* Solve for source terms
        do n = 1, Nx-1
           P1(n) = 0.
           Q1(n) = 0.
           x_xi(n) = .5*(xx(n+1,0) - xx(n-1,0))
           y_xi(n) = .5*(yy(n+1,0) - yy(n-1,0))

           x_xi_xi(n) = (xx(n+1,0) - 2.*xx(n,0) + xx(n-1,0))
           y_xi_xi(n) = (yy(n+1,0) - 2.*yy(n,0) + yy(n-1,0))

           if (xx(n,0).eq.0. .or. xx(n,0).eq.1.) then
              x_eta(n) = max(5.*tau,1.)*AR*(-x_xi(n)*cos(theta)
     &                    - y_xi(n)*sin(theta))
              y_eta(n) = max(5.*tau,1.)*AR*(-y_xi(n)*cos(theta)
     &                    + x_xi(n)*sin(theta))
           else
              x_eta(n) = AR*(-x_xi(n)*cos(theta) - y_xi(n)*sin(theta))
              y_eta(n) = AR*(-y_xi(n)*cos(theta) + x_xi(n)*sin(theta))
           endif

           x_eta_xi(n) = 0.5*(x_eta(n+1)-x_eta(n-1))
           y_eta_xi(n) = 0.5*(y_eta(n+1)-y_eta(n-1))

           Jacobi(n) = x_xi(n)*y_ ta(n) - x_eta(n)*y_xi(n)

           a1(n) = x_eta(n)**2 + y_eta(n)**2
           b1(n) = x_xi(n)*x_eta(n) + y_xi(n)*y_eta(n)
           c1(n) = x_xi(n)**2 + y_xi(n)**2
        enddo

c* SOR by lines
        Niter = 0
        error = 9999.
        do while (error.gt.EPS)
           error = 0.
           Niter = Niter + 1

c** solve for source terms
           if (tau.ne.0.) then
              do n = 1, Nx-1
                 x_eta_eta(n) = 0.5*(-7.*xx(n,0) + 8.*xx(n,1) - xx(n,2))
     &                         - 3.*x_eta(n)
                 y_eta_eta(n) = 0.5*(-7.*yy(n,0) + 8.*yy(n,1) - yy(n,2))
     &                         - 3.*y_eta(n)
                 R1 = (-a1(n)*x_xi_xi(n) + 2.*b1(n)*x_eta_xi(n)
     &                - c1(n)*x_eta_eta(n))/Jacobi(n)**2
                 R2 = (-a1(n)*y_xi_xi(n) + 2.*b1(n)*y_eta_xi(n)
     &                - c1(n)*y_eta_eta(n))/Jacobi(n)**2
                 P1(n) = P1(n) + omega_P*((y_eta(n)*R1 - x_eta(n)*R2)
     &                 /Jacobi(n) - P1(n))
                 Q1(n) = Q1(n) + omega_Q*((-y_xi(n)*R1 + x_xi(n)*R2)
     &                 /Jacobi(n) - Q1(n))
              enddo
           endif

c* solve for each line
           do kk = 1, Ny-1

c* evaluate alpha, beta, and gamma
              do i = 1, Nx-1
                 alpha(i) = .25*(xx(i,kk+1)-xx(i,kk-1))**2
     &                    + .25*(yy(i,kk+1)-yy(i,kk-1))**2
                 beta(i) = .25*(xx(i+1,kk)-xx(i-1,kk))*(xx(i,kk+1)
     &                   -xx(i,kk-1)) + .25*(yy(i+1,kk)-yy(i-1,kk))
     &                   *(yy(i,kk+1)-yy(i,kk-1))
                 gamma(i) = .25*(xx(i+1,kk)-xx(i-1,kk))**2
     &                    + .25*(yy(i+1,kk)-yy(i-1,kk))**2
                 Jacobi2(i) = (.25*(xx(i+1,kk)-xx(i-1,kk))*
     &                         (yy(i,kk+1)-yy(i,kk-1))
     &                       - .25*(yy(i+1,kk)-yy(i-1,kk))*
     &                         (xx(i,kk+1)-xx(i,kk-1)))**2
              enddo
```

```
c* Solve for x
c* set up matrix for tridiagonal system of equations
               do i = 1, Nx-1
                  AA(i) = omega*alpha(i)
                  DD(i) = -2.*(gamma(i) + alpha(i))
                  BB(i) = omega*alpha(i)
               enddo

c* set up vector of constants for tridiagonal system of equations
               do i = 1, Nx-1
                  xxi = xx(i+1,kk) - xx(i-1,kk)
                  xeta = xx(i,kk+1) - xx(i,kk-1)

                  CC(i) = -omega*gamma(i)*(xx(i,kk+1) + xx(i,kk-1))
     &                    + omega*0.5*beta(i)*(xx(i+1,kk+1)
     &                    - xx(i+1,kk-1) - xx(i-1,kk+1) + xx(i-1,kk-1))
     &                    + 2.*(omega-1.)*(alpha(i) + gamma(i))*xx(i,kk)
     &                    - Jacobi2(i)*(P1(i)*xxi*exp(-a*kk) +
     &                    Q1(i)*xeta*exp(-b*kk))
               enddo
               CC(1)    = CC(1)    - BB(1)*xx(0,kk)
               CC(Nx-1) = CC(Nx-1) - AA(Nx-1)*xx(Nx,kk)

c* solve tridiagonal system of equations
               call tridiag(1, Nx-1, BB, DD, AA, CC)
               do m = 1, Nx-1
                  if (abs(CC(m)-xx(m,kk)).gt.error) then
                     error = abs(CC(m) - xx(m,kk))
                  endif
                  xx(m,kk) = CC(m)
               enddo

c* Solve for y
c* set up matrix for tridiagonal system of equations
               do i = 1, Nx-1
                  AA(i) = omega*alpha(i)
                  DD(i) = -2.*(gamma(i) + alpha(i))
                  BB(i) = omega*alpha(i)
               enddo

c* set up vector of constants for tridiagonal system of equations
               do i = 1, Nx-1
                  yxi = yy(i+1,kk) - yy(i-1,kk)
                  yeta = yy(i,kk+1) - yy(i,kk-1)

                  CC(i) = -omega*gamma(i)*(yy(i,kk+1) + yy(i,kk-1))
     &                    + 0.5*omega*beta(i)*(yy(i+1,kk+1)
     &                    -yy(i+1,kk-1) - yy(i-1,kk+1) + yy(i-1,kk-1))
     &                    + 2.*(omega-1.)*(alpha(i)+gamma(i))*yy(i,kk)
     &                    - Jacobi2(i)*(P1(i)*yxi*exp(-a*kk) +
     &                    Q1(i)*yeta*exp(-b*kk))
               enddo
               CC(1)    = CC(1)    - BB(1)*yy(0,kk)
               CC(Nx-1) = CC(Nx-1) - AA(Nx-1)*yy(Nx,kk)

c* solve tridiagonal system of equations
               call tridiag(1, Nx-1, BB, DD, AA, CC)
               do m = 1, Nx-1
                  if (abs(CC(m)-yy(m,kk)).gt.error) then
                     error = abs(CC(m) - yy(m,kk))
                  endif
                  yy(m,kk) = CC(m)
               enddo

            enddo

c* set x=xxmax and x=xxmin boundary conditions to next interior point
            do kk = 1, Ny-1
               yy(0,kk) = yy(1,kk)
               yy(Nx,kk) = yy(Nx-1,kk)
            enddo
```

```
         type*, 'iteration number = ',Niter,'  error = ',error
      enddo

      return
      end



c*************************************************************************
c*                                                                       *
c*     Subroutine to solve a tridiagonal system of equations.            *
c*     Taken from "Computational Fluid Mechanics and Heat Transfer"      *
c*     by Anderson, Tannehill and Pletcher.                              *
c*                                                                       *
c*************************************************************************

      SUBROUTINE TRIDIAG(IL,IU,BB,DD,AA,RR)
      IMPLICIT NONE

      include 'GRID.INC'

      INTEGER IL                        !SUBSCRIPT OF FIRST EQUATION
      INTEGER IU                        !SUBSCRIPT OF LAST EQUATION
      REAL*4 BB(Maxdim)                 !COEFFICIENT BEHIND DIAGONAL
      REAL*4 DD(Maxdim)                 !COEFFICIENT ON DIAGONAL
      REAL*4 AA(Maxdim)                 !COEFFICIENT AHEAD OF DIAGONAL
      REAL*4 RR(Maxdim)                 !ELEMENT OF CONSTANT VECTOR
      INTEGER LP
      INTEGER I, J                      !POINTERS
      REAL*4 R

C**   ESTABLISH UPPER TRIANGULAR MATRIX

      LP = IL + 1
      DO I = LP, IU
         R = BB(I)/DD(I-1)
         DD(I) = DD(I)-R*AA(I-1)
         RR(I) = RR(I)-R*RR(I-1)
      ENDDO

C**   BACK SUBSTITUTION

      RR(IU) = RR(IU)/DD(IU)
      DO I = LP, IU
         J = IU - I + IL
         RR(J) = (RR(J)-AA(J)*RR(J+1))/DD(J)
      ENDDO

C**   SOLUTION STORED IN RR

      RETURN
      END



c*************************************************************************
c*                                                                       *
c*     this subroutine requests the user to input the value of a         *
c*     real variable                                                     *
c*                                                                       *
c*************************************************************************

      subroutine Rrequest (name,var)
      character*10 name
      real*4 var

      write(6,1) name
 1    format($,' ',A,' = ')
      accept 11, var
```

```
 11     format(F)

        return
        end


c*************************************************************************
c*                                                                       *
c*      this subroutine requests the user to input the value of a        *
c*      integer variable                                                 *
c*                                                                       *
c*************************************************************************

        subroutine Irequest (name,var)
        character*10 name
        integer var

        write(6,1) name
 1      format(1x,' ',A,' = ')
        accept 11, var
 11     format(I)

        return
        end




      SUBROUTINE READIN

C---- Read in, normalize and spline blade data

      INCLUDE 'TRI.INC'
      INCLUDE 'GRID.INC'

      CHARACTER*32 NAMEXT
      CHARACTER*80 NAME

C---- Read in blade data
      OPEN(UNIT=3,STATUS='OLD')
 1000 FORMAT(A32)
      READ(3,1000) NAME
      READ(3,*) GSINL, GSOUT, CHINL, CHOUT, PITCH

      WRITE(6,1001) NAME
 1001 FORMAT(/,' Blade name: ',A60)

      READ(3,*) XB(1), YB(1)
      XMIN = XB(1)
      XMAX = XB(1)
      YMIN = YB(1)
      DO 1 IB = 2, 12345
        READ(3,*,END=11) XB(IB),YB(IB)
        XMAX = AMAX1(XMAX,XB(IB))
        IF(XMIN.GT.XB(IB)) THEN
         XMIN = XB(IB)
         YMIN = YB(IB)
        ENDIF
 1    CONTINUE
 11   IIB = IB - 1
      CLOSE(UNIT=3)

      IF(IIB.GT.IBX) STOP 'Array overflow:  IBX too small'

C---- Normalize blade and calculate surface arc length array
      PITCH = PITCH/(XMAX-XMIN)
      DO 2 IB = 1, IIB
        XB(IB) = (XB(IB)-XMIN) / (XMAX-XMIN)
        YB(IB) = (YB(IB)-YMIN) / (XMAX-XMIN)
 2    CONTINUE
```

```
C---- close t.e. if open
      IF( XB(1).NE.XB(IIB) .OR. YB(1).NE.YB(IIB) ) THEN
       ASOUT = ATAN( (YB(2)  -YB(1))     / (XB(2)  -XB(1))     )
       APOUT = ATAN( (YB(IIB)-YB(IIB-1)) / (XB(IIB)-XB(IIB-1)) )
       DSTE = SQRT( (XB(1)-XB(IIB))**2 + (YB(1)-YB(IIB))**2 )
       AOUT = 0.5*(APOUT+ASOUT)
       XOUT = 0.5*(XB(1)+XB(IIB)) + 3.0*DSTE*COS(AOUT)
       YOUT = 0.5*(YB(1)+YB(IIB)) + 3.0*DSTE*SIN(AOUT)
       XS1 =  (XB(1)-XOUT)*COS(AOUT) + (YB(1)-YOUT)*SIN(AOUT)
       YS1 = -(XB(1)-XOUT)*SIN(AOUT) + (YB(1)-YOUT)*COS(AOUT)
       YPS1 = TAN(ASOUT-AOUT) * XS1
       XP1 =  (XB(IIB)-XOUT)*COS(AOUT) + (YB(IIB)-YOUT)*SIN(AOUT)
       YP1 = -(XB(IIB)-XOUT)*SIN(AOUT) + (YB(IIB)-YOUT)*COS(AOUT)
       YPP1 = TAN(APOUT-AOUT) * XP1

       WRITE(6,1002)
 1002  FORMAT(/,' Input flap deflection angle (degrees): ',$)
       READ(5,*) AFLAP
       YPFLAP = TAN(3.14159*AFLAP/180.0)

       WRITE(NAMEXT,1003) AFLAP
 1003  FORMAT(' (flap deflection angle =',F4.1,')')
       LENSTART = INDEX(NAME,'       ')
       NAME(LENSTART:LENSTART+31) = NAMEXT

       IIB = IIB+20
       IF(IIB.GT.IBX) STOP 'Array overflow:  IBX too small'

       DO 3 IB = IIB-20, 1, -1
         XB(IB+10) = XB(IB)
         YB(IB+10) = YB(IB)
 3     CONTINUE

       DO 4 IB = 1, 10
         ETA = 0.1*(IB-1)
         XXS = XS1*ETA
         YYS = YS1*ETA*ETA*(3.0-2.0*ETA) + YPS1*ETA*ETA*(ETA-1.0)
     &       - YPFLAP*XS1*0.5*(ETA-1.0)**2
         XB(IB) = XOUT + XXS*COS(AOUT) - YYS*SIN(AOUT)
         YB(IB) = YOUT + XXS*SIN(AOUT) + YYS*COS(AOUT)
 4     CONTINUE

       DO 5 IB = IIB-9, IIB
         ETA = 0.1*(IIB-IB)
         XXP = XP1*ETA
         YYP = YP1*ETA*ETA*(3.0-2.0*ETA) + YPP1*ETA*ETA*(ETA-1.0)
     &       - YPFLAP*XP1*0.5*(ETA-1.0)**2
         XB(IB) = XOUT + XXP*COS(AOUT) - YYP*SIN(AOUT)
         YB(IB) = YOUT + XXP*SIN(AOUT) + YYP*COS(AOUT)
 5     CONTINUE

      ENDIF

C---- Spline blade surface(s) and find leading edge position
      SB(1) = 0.
      DO 6 IB = 2, IIB
        ALF = FLOAT( MIN(IB-1,IIB-IB) ) / FLOAT(IIB/2)
        SB(IB) = SB(IB-1) +
     &       SQRT( (XB(IB)-XB(IB-1))**2 + (ALF*(YB(IB)-YB(IB-1)))**2 )
 6    CONTINUE

      CALL SPLINE(XB,XPB,SB,IIB)
      CALL SPLINE(YB,YPB,SB,IIB)

      DO 7 IB=2, IIB
        DP1 = XPB(IB-1) + GSINL*YPB(IB-1)
        DP2 = XPB(IB)   + GSINL*YPB(IB)
        IF(DP1.LT.0.0 .AND. DP2.GE.0.0) GO TO 71
 7    CONTINUE

      STOP 'Leading edge not found'
 71   DSB  = SB(IB) - SB(IB-1)
```

```
      SBLE = SB(IB-1) + DSB*DP1/(DP1-DP2)
      XLE  = SEVAL(SBLE,XB,XPB,SB,IIB)
      YLE  = SEVAL(SBLE,YB,YPB,SB,IIB)

      RETURN
      END



      SUBROUTINE GRINIT

C---- Fix grid points on boundary of domain, and initialize interior

      INCLUDE 'TRI.INC'
      INCLUDE 'GRID.INC'

C---- Input and check grid size
      WRITE(6,1000)
 1000 FORMAT(/,' Input II, JJ: ',$)
      READ(5,*) II,JJ
      IF(II.GT.Maxdim) STOP 'Array overflow:  Maxdim too small'
      IF(JJ.GT.Maxdim) STOP 'Array overflow:  Maxdim too small'

C---- Set various parameters
      SLEN = CHINL + 0.5*SB(IIB) + CHOUT
      NINL = INT( FLOAT(II)*CHINL/SLEN )
      NOUT = INT( FLOAT(II)*CHOUT/SLEN )
      NBLD = II - NOUT - NINL + 2
      ILE = NINL
      ITE = II - NOUT + 1

C---- Set inlet stagnation streamline
      DO 1 K=1, NINL
        XX(K,1) = XLE + CHINL * FLOAT(K-NINL) / FLOAT(NINL-1)
        YY(K,1) = YLE + (XX(K,1)-XLE) * GSINL
        XX(K,JJ) = XX(K,1)
        YY(K,JJ) = YY(K,1) + PITCH
 1    CONTINUE

C---- Set outlet stagnation streamline
      XTE = XB(1)
      YTE = YB(1)

      DO 2 K= 1, NOUT
        I = II-NOUT+K
        XX(I,1)  = XTE + CHOUT * FLOAT(K-1) / FLOAT(NOUT-1)
        YY(I,1)  = YTE + (XX(I,1)-XTE) * GSOUT
        XX(I,JJ) = XX(I,1)
        YY(I,JJ) = YY(I,1) + PITCH
 2    CONTINUE

C---- Set points on blade suction surface
      DO 3 K=1, NBLD
        I = NINL + K - 1
        S = SBLE - SBLE*FLOAT(K-1)/FLOAT(NBLD-1)
        XX(I,1) = SEVAL(S,XB,XPB,SB,IIB)
        YY(I,1) = SEVAL(S,YB,YPB,SB,IIB)
 3    CONTINUE

C---- Set points on blade pressure surface
      DO 4 K=1, NBLD
        I = NINL + K - 1
        S = SBLE + (SB(IIB)-SBLE)*FLOAT(K-1)/FLOAT(NBLD-1)
        XX(I,JJ) = SEVAL(S,XB,XPB,SB,IIB)
        YY(I,JJ) = SEVAL(S,YB,YPB,SB,IIB) + PITCH
 4    CONTINUE

C---- set up metrics
      DO 5 I=1, II
        XPOS(I) = FLOAT(I-1)/FLOAT(II-1)
```

```
 5    CONTINUE

      DO 6 J = 1, JJ
        RJ = FLOAT(J-1)/FLOAT(JJ-1)
        YPOS(J) = RJ - 1.8 * ( (RJ-0.5) * ((RJ-0.5)**2-0.25) )
 6    CONTINUE

C---- Initialize interior grid
      DO 7 I = 1, II
        DO 71 J =2, JJ-1
          XX(I,J) = XX(I,1) + YPOS(J)*(XX(I,JJ)-XX(I,1))
          YY(I,J) = YY(I,1) + YPOS(J)*(YY(I,JJ)-YY(I,1))
 71     CONTINUE
 7    CONTINUE

      RETURN
      END




      SUBROUTINE IMPROV

C---- Improves grid after elliptic grid generation


      INCLUDE 'TRI.INC'
      INCLUDE 'GRID.INC'

      DIMENSION SUM(Maxdim), XT1(Maxdim), YT1(Maxdim)
      DIMENSION XT2(Maxdim), YT2(Maxdim)

      DO 1 I = 1, II
        IM = I-1
        IP = I+1
        IF(I.EQ.1) IM=1
        IF(I.EQ.II) IP=II

        SUM(1) = 0.
        DO 11 J = 1, JJ-1
          JP = J+1
          XS = XX(IP,J)+XX(IP,JP) - XX(IM,J)-XX(IM,JP)
          YS = YY(IP,J)+YY(IP,JP) - YY(IM,J)-YY(IM,JP)
          SS = SQRT(XS*XS + YS*YS)
          XS = XS/SS
          YS = YS/SS
          SUM(JP) = SUM(J) + ABS( (XX(I,J)-XX(I,JP))*YS
     &                          - (YY(I,J)-YY(I,JP))*XS )
 11     CONTINUE

        J = 1
        DO 12 JO = 2, JJ-1
          SUMJ = FLOAT(JO-1)/FLOAT(JJ-1) * SUM(JJ)
 121       IF(SUMJ.GT.SUM(J+1)) THEN
           J = J+1
           GOTO 121
          ENDIF
          ALPHA = (SUMJ-SUM(J)) / (SUM(J+1)-SUM(J))
          XT2(JO) = XX(I,J) + ALPHA*(XX(I,J+1)-XX(I,J))
          YT2(JO) = YY(I,J) + ALPHA*(YY(I,J+1)-YY(I,J))
 12     CONTINUE

        DO 13 J = 2, JJ-1
          IF(I.NE.1) THEN
           XX(IM,J) = XT1(J)
           YY(IM,J) = YT1(J)
          ENDIF
          XT1(J) = XT2(J)
          YT1(J) = YT2(J)
          IF(I.EQ.II) THEN
```

```
          XX(I,J) = XT1(J)
          YY(I,J) = YT1(J)
        ENDIF
 13    CONTINUE

 1    CONTINUE
      RETURN
      END



C*************************************************C
C                                                 C
C   ISES - an Integrated Steamtube Euler Solver   C
C                                                 C
C   Written by M. Giles and M.Drela               C
C                                                 C
C   Copyright M.I.T. (1985)                       C
C                                                 C
C*************************************************C
C
      SUBROUTINE ELLIP(IMAX,JMAX,II,JJ,JJJ,X,Y,YPOS,XPOS)
      DIMENSION X(0:IMAX,0:JMAX), Y(0:IMAX,0:JMAX)
      DIMENSION YPOS(JMAX), XPOS(IMAX)
      CHARACTER*1 ANS
C
      DIMENSION C(400),D(2,400)
      IF(II.GT.400) STOP 'ELLIP dimensions must be increased'
C
      ITMAX = 50
C
      DSET1 = 1.0E-1
      DSET2 = 5.0E-3
      DSET3 = 2.0E-4
C
      RLX1 = 1.30           !          DMAX > DSET1
      RLX2 = 1.50           !  DSET1 > DMAX > DSET2
      RLX3 = 1.60           !  DSET2 > DMAX > DSET3
CCC   STOP                  !  DSET3 > DMAX
C
      RLX = RLX1
C
      DO 1 ITER = 1, ITMAX
C
        DMAX = 0.
        DO 5 JO=2, JJ-1
          JM = JO-1
          JP = JO+1
C
          IF(JO.EQ.JJJ) THEN
           DO 2 IO=2, II-1
             X(IO,JO) = X(IO,JM)
 2         CONTINUE
           GO TO 5
          ELSE IF(JO.EQ.JJJ+1) THEN
           DO 3 IO=2, II-1
             X(IO,JO) = X(IO,JP)
 3         CONTINUE
           GO TO 5
          ENDIF
C
          DO 6 IO=2, II-1
            IM = IO-1
            IP = IO+1
C
            XMM = X(IM,JM)
            XOM = X(IO,JM)
            XPM = X(IP,JM)
            XMO = X(IM,JO)
            XOO = X(IO,JO)
```

```
            XPO = X(IP,JO)
            XMP = X(IM,JP)
            XOP = X(IO,JP)
            XPP = X(IP,JP)
            YMM = Y(IM,JM)
            YOM = Y(IO,JM)
            YPM = Y(IP,JM)
            YMO = Y(IM,JO)
            YOO = Y(IO,JO)
            YPO = Y(IP,JO)
            YMP = Y(IM,JP)
            YOP = Y(IO,JP)
            YPP = Y(IP,JP)
C
            DXIM = XPOS(IO)-XPOS(IM)
            DXIP = XPOS(IP)-XPOS(IO)
            DXIAV = 0.5*(DXIM+DXIP)
C
            DETM = YPOS(JO)-YPOS(JM)
            DETP = YPOS(JP)-YPOS(JO)
            DETAV = 0.5*(DETM+DETP)
C
            DXDET = ( XOP - XOM ) / DETAV
            DYDET = ( YOP - YOM ) / DETAV
            DXDXI = ( XPO - XMO ) / DXIAV
            DYDXI = ( YPO - YMO ) / DXIAV
C
            ALF = DXDET**2 + DYDET**2
            BET = DXDET*DXDXI + DYDET*DYDXI
            GAM = DXDXI**2 + DYDXI**2
C
            CXIM = 1.0 / (DXIM*DXIAV)
            CXIP = 1.0 / (DXIP*DXIAV)
            CETM = 1.0 / (DETM*DETAV)
            CETP = 1.0 / (DETP*DETAV)
C
            B =      -ALF*CXIM
            A = ALF*(CXIM+CXIP) + GAM*(CETM+CETP)
            C(IO) =  -ALF*CXIP
            IF(IO.EQ.2) B = 0
C
            D(1,IO) = ALF*((XMO-XOO)*CXIM + (XPO-XOO)*CXIP)
     &              - 2.0*BET*(XPP-XMP-XPM+XMM) / (4.0*DXIAV*DETAV)
     &              + GAM*((XOM-XOO)*CETM + (XOP-XOO)*CETP)
C
            D(2,IO) = ALF*((YMO-YOO)*CXIM + (YPO-YOO)*CXIP)
     &              - 2.0*BET*(YPP-YMP-YPM+YMM) / (4.0*DXIAV*DETAV)
     &              + GAM*((YOM-YOO)*CETM + (YOP-YOO)*CETP)
C
            AINV = 1.0/(A - B*C(IM))
            C(IO) = C(IO) * AINV
            D(1,IO) = ( D(1,IO) - B*D(1,IM) ) * AINV
            D(2,IO) = ( D(2,IO) - B*D(2,IM) ) * AINV
C
    6     CONTINUE
C
          D(1,II) = 0.
          D(2,II) = 0.
C
          IFIN = II-1
          DO 8 IBACK=2, IFIN
            IO = II-IBACK+1
            IP = IO+1
            D( 1,IO) = D( 1,IO) - C(IO)*D(1,IP)
            D( 2,IO) = D( 2,IO) - C(IO)*D(2,IP)
            X(IO,JO) = X(IO,JO) + RLX*D(1,IO)
            Y(IO,JO) = Y(IO,JO) + RLX*D(2,IO)
            AD1 = ABS(D(1,IO))
            AD2 = ABS(D(2,IO))
            DMAX = AMAX1(DMAX,AD1,AD2)
    8     CONTINUE
C
```

```
    5    CONTINUE
C
       WRITE(6,*) 'Dmax = ', DMAX, RLX
C
       RLX = RLX1
       IF(DMAX.LT.DSET1) RLX = RLX2
       IF(DMAX.LT.DSET2) RLX = RLX3
       IF(DMAX.LT.DSET3) RETURN
C
    1  CONTINUE
C
       RETURN
       END ! ELLIP




       SUBROUTINE SPLINE(X,XP,S,II)
C
       DIMENSION X(1),XP(1),S(1)
       DIMENSION A(480),B(480),C(480)
C
       IF(II.GT.480) STOP 'SPLINE: Array overflow'
       DO 1 I = 1, II
C
C------ Beginning points
        IF(I.EQ.1 .OR. S(I).EQ.S(I-1)) THEN
         DSMI = 0.
         DXM  = 0.
         DSPI = 1.0 / (S(I+1)-S(I))
         DXP  = X(I+1) - X(I)
C
C------ End points
        ELSE IF(I.EQ.II .OR. S(I).EQ.S(I+1)) THEN
         DSMI = 1.0 / (S(I) - S(I-1))
         DXM  = X(I) - X(I-1)
         DSPI = 0.
         DXP  = 0.
C
C------ Interior points
        ELSE
         DSMI = 1.0 / (S(I) - S(I-1))
         DXM  = X(I) - X(I-1)
         DSPI = 1.0 / (S(I+1) - S(I))
         DXP  = X(I+1) - X(I)
C
        ENDIF
C
        B(I)  = DSMI
        A(I)  = 2.0 * (DSMI + DSPI)
        C(I)  = DSPI
        XP(I) = 3.0 * (DXP*DSPI**2 + DXM*DSMI**2)
C
  1   CONTINUE
C
       CALL TRISOL(A,B,C,XP,II)
C
       RETURN
       END ! SPLINE




       SUBROUTINE TRISOL(A,B,C,D,KK)
C
       DIMENSION A(1),B(1),C(1),D(1)
C
       DO 1 K = 2, KK
         KM = K - 1
```

```
        C(KM) = C(KM) / A(KM)
        D(KM) = D(KM) / A(KM)
        A(K) = A(K) - B(K) * C(KM)
        D(K) = D(K) - B(K) * D(KM)
  1   CONTINUE
C
      D(KK) = D(KK) / A(KK)
C
      DO 2 K = KK-1, 1, -1
        D(K) = D(K) - C(K) * D(K+1)
  2   CONTINUE
C
      RETURN
      END ! TRISOL




      FUNCTION SEVAL(SS,X,XP,S,N)
      REAL X(1), XP(1), S(1)
C
      ILOW = 1
      I = N
C
   10 IF(I-ILOW .LE. 1) GO TO 11
C
      IMID = (I+ILOW)/2
      IF(SS .LT. S(IMID)) THEN
       I = IMID
      ELSE
       ILOW = IMID
      ENDIF
      GO TO 10
C
   11 DS = S(I) - S(I-1)
      T = (SS-S(I-1)) / DS
      CX1 = DS*XP(I-1) - X(I) + X(I-1)
      CX2 = DS*XP(I)   - X(I) + X(I-1)
      SEVAL = T*X(I) + (1.0-T)*X(I-1) + (T-T*T)*((1.0-T)*CX1 - T*CX2)
      RETURN
      END ! SEVAL




      FUNCTION DEVAL(SS,X,XP,S,N)
      REAL X(1), XP(1), S(1)
C
      ILOW = 1
      I = N
C
   10 IF(I-ILOW .LE. 1) GO TO 11
C
      IMID = (I+ILOW)/2
      IF(SS .LT. S(IMID)) THEN
       I = IMID
      ELSE
       ILOW = IMID
      ENDIF
      GO TO 10
C
   11 DS = S(I) - S(I-1)
      T = (SS-S(I-1)) / DS
      CX1 = DS*XP(I-1) - X(I) + X(I-1)
      CX2 = DS*XP(I)   - X(I) + X(I-1)
      DEVAL = (X(I)-X(I-1) + (1.-4.*T+3.*T*T)*CX1 + T*(3.*T-2.)*CX2)/DS
      RETURN
      END ! DEVAL
```

```
      subroutine pointers
      implicit none

      include 'GRID.INC'
      include 'TRI.INC'
      integer i, j                    !pointers
      real*4 D1, D2                   !diagonals of rectangle
      integer NN, FF, CC              !pointers

      NN = 0
      CC = 0
      FF = 0

c** assign values to node arrays
      do j = 0, Ny
         do i = 0, Nx
            NN = NN + 1
            x(NN) = xx(i,j)
            y(NN) = yy(i,j)
         enddo
      enddo
      Nmax = NN

c**  assign horizontal and vertical faces and cells
      do j = 0, Ny-1
         do i = 1, Nx
            FF = FF + 1
            face(FF,1) = 2*Nx*j + i
            face(FF,2) = Nx*(2*j-1) + i
            face(FF,3) = (Nx+1)*j + i
            face(FF,4) = (Nx+1)*j + i + 1
            CC = CC + 1
            cell(CC,1) = FF
            cell(CC,6) = face(FF,3)
            cell(CC,4) = face(FF,4)
         enddo
         do i = 1, Nx
            FF = FF + 1
            face(FF,3) = (Nx+1)*(j+1) + i
            face(FF,4) = (Nx+1)*j + i
            CC = CC + 1
            cell(CC,1) = FF + Nx + 1
            cell(CC,4) = (Nx+1)*(j+1) + i
            cell(CC,6) = (Nx+1)*(j+1) + i + 1
         enddo
         FF = FF + 1
         face(FF,3) = (Nx+1)*(j+2)
         face(FF,4) = (Nx+1)*(j+1)
      enddo

      Cmax = CC

c**  assign top boundary faces
      do i = 1, Nx
         FF = FF + 1
         face(FF,1) = 0
         face(FF,2) = Nx*(2*Ny-1) + i
         face(FF,3) = (Nx+1)*Ny + i
         face(FF,4) = (Nx+1)*Ny + 1 + i
         face(FF,5) = 0
      enddo

c**  set bottom edge face array
      do i = 1, Nx
         face(i,2) = 0
         face(i,6) = 0
      enddo
```

```
c**  set inlet edge face array
        do j = 1, Ny
           face((2*Nx+1)*j-Nx,2) = 0
           face((2*Nx+1)*j-Nx,6) = 0
        enddo

c**  set outlet edge face array
        do j = 1, Ny
           face((2*Nx+1)*j,1) = 0
           face((2*Nx+1)*j,5) = 0
        enddo

c** split rectangular cells into two triangular cells in grid
        do j = 1, Ny
           do i = 1, Nx
              FF = FF + 1
              D1 = (xx(i,j)-xx(i-1,j-1))**2 + (yy(i,j)-yy(i-1,j-1))**2
              D2 = (xx(i-1,j)-xx(i,j-1))**2 + (yy(i-1,j)-yy(i,j-1))**2
              if (D1 .gt. D2) then
c             if (mod(i+j,2).eq.0) then
                 face((2*Nx+1)*(j-1)+Nx+i,1) = 2*(j-1)*Nx + i
                 face((2*Nx+1)*j-Nx+i,2) = (2*j-1)*Nx + i
                 face((2*Nx+1)*(j-1)+Nx+i,5) = (Nx+1)*(j-1) + 1 + i
                 face((2*Nx+1)*j-Nx+i,6) = (Nx+1)*j + i
                 face((2*Nx+1)*(j-1)+i,5) = (Nx+1)*j + i
                 face((2*Nx+1)*j+i,6) = (Nx+1)*(j-1) + 1 + i
                 face(FF,3) = (Nx+1)*j + i
                 face(FF,4) = (Nx+1)*(j-1) + 1 + i
                 face(FF,5) = (Nx+1)*j + 1 + i
                 face(FF,6) = (Nx+1)*(j-1) + i
                 cell(2*Nx*(j-1)+i,2) = FF
                 cell(2*Nx*(j-1)+i,3) = (2*Nx+1)*(j-1) + Nx + i
                 cell(2*Nx*(j-1)+i,5) = (Nx+1)*j + i
                 cell(2*Nx*(j-1)+Nx+i,2) = FF
                 cell(2*Nx*(j-1)+Nx+i,3) = (2*Nx+1)*j - Nx + i
                 cell(2*Nx*(j-1)+Nx+i,5) = (Nx+1)*(j-1) + 1 + i
              else
                 face((2*Nx+1)*(j-i)+Nx+i,1) = (2*j-1)*Nx + i
                 face((2*Nx+1)*j-Nx+i,2) = 2*(j-1)*Nx + i
                 face((2*Nx+1)*(j-1)+Nx+i,5) = (Nx+1)*j + 1 + i
                 face((2*Nx+1)*j-Nx+i,6) = (Nx+1)*(j-1) + i
                 face((2*Nx+1)*(j-1)+i,5) = (Nx+1)*j + 1 + i
                 face((2*Nx+1)*j+i,6) = (Nx+1)*(j-1) + i
                 face(FF,3) = (Nx+1)*(j-1) + i
                 face(FF,4) = (Nx+1)*j + 1 + i
                 face(FF,5) = (Nx+1)*j + i
                 face(FF,6) = (Nx+1)*(j-1) + 1 + i
                 cell(2*Nx*(j-1)+i,2) = (2*Nx+1)*j - Nx + i
                 cell(2*Nx*(j-1)+i,3) = FF
                 cell(2*Nx*(j-1)+i,5) = (Nx+1)*j + 1 + i
                 cell(2*Nx*(j-1)+Nx+i,2) = (2*Nx+1)*(j-1) + Nx + i
                 cell(2*Nx*(j-1)+Nx+i,3) = FF
                 cell(2*Nx*(j-1)+Nx+i,5) = (Nx+1)*(j-1) + i
              endif
              face(FF,1) = (2*j-1)*Nx + i
              face(FF,2) = 2*(j-1)*Nx + i
           enddo
        enddo

        Fmax = FF

        return
        end




        subroutine bpointers
        implicit none
```

```
      include 'GRID.INC'
      include 'TRI.INC'
      integer i, j, n                          !pointers
      integer FF, EN, EF, IN, ON, WN, PN       !pointers
      integer CC
      integer F1, F2                           !periodic faces
      real dxM, dyM, dxP, dyP, dsM, dsP, dx, dy!change in x & y near node
      integer node                             !current node processed

      EN = 0
      EF = 0
      IN = 0
      ON = 0
      WN = 0
      PN = 0

c**   wall boundaries
      if (pitch.eq.0.) then
         do i = 1, Nx
            EF = EF + 1
            eface(EF) = Ny*(2*Nx+1) + i
            EN = EN + 1
            node = (Nx+1)*Ny + i
            enode(EN) = node
            dxP = x(node+1) - x(node)
            dyP = y(node+1) - y(node)
            if (i .eq. 1) then
               dxM = -dxP
               dyM = -dyP
            else
               dxM = x(node-1) - x(node)
               dyM = y(node-1) - y(node)
            endif
            if ((dxM*dxP + dyM*dyP) .gt. 0.) then
               dsM = sqrt(dxM**2 + dyM**2)
               dsP = sqrt(dxP**2 + dyP**2)
               dx = dxM/dsM + dxP/dsP
               dy = dyM/dsM + dyP/dsP
            else
               dx = dxP - dxM
               dy = dyP - dyM
            endif
            senode(EN) = dy/sqrt(dx**2 + dy**2)
            cenode(EN) = dx/sqrt(dx**2 + dy**2)
         enddo

         EN = EN + 1
         node = (Nx+1)*(Ny+1)
         enode(EN) = node
         dxM = x(node-1) - x(node)
         dyM = y(node-1) - y(node)
         dxP = -dxM
         dyP = -dyM
         dx = dxP - dxM
         dy = dyP - dyM
         senode(EN) = dy/sqrt(dx**2 + dy**2)
         cenode(EN) = dx/sqrt(dx**2 + dy**2)

         tnode = EN

         do i = 1, Nx
            EF = EF + 1
            eface(EF) = i
            EN = EN + 1
            node = i
            enode(EN) = node
            dxP = x(node+1) - x(node)
            dyP = y(node+1) - y(node)
            if (i .eq. 1) then
               dxM = -dxP
               dyM = -dyP
```

```
               else
                  dxM = x(node-1) - x(node)
                  dyM = y(node-1) - y(node)
               endif
               if ((dxM*dxP + dyM*dyP) .gt. 0.) then
                  dsM = sqrt(dxM**2 + dyM**2)
                  dsP = sqrt(dxP**2 + dyP**2)
                  dx = dxM/dsM + dxP/dsP
                  dy = dyM/dsM + dyP/dsP
               else
                  dx = dxP - dxM
                  dy = dyP - dyM
               endif
               senode(EN) = dy/sqrt(dx**2 + dy**2)
               cenode(EN) = dx/sqrt(dx**2 + dy**2)
            enddo

            EN = EN + 1
            node = Nx + 1
            enode(EN) = node
            dxM = x(node-1) - x(node)
            dyM = y(node-1) - y(node)
            dxP = -dxM
            dyP = -dyM
            dx = dxP - dxM
            dy = dyP - dyM
            senode(EN) = dy/sqrt(dx**2 + dy**2)
            cenode(EN) = dx/sqrt(dx**2 + dy**2)

            bnode = EN
            Wmax = 2*Nx
         else if (pitch.ne.0.) then
c**   periodic nodes
            do i = 1, ILE
               PN = PN + 1
               pnode(PN,1) = i
               pnode(PN,2) = (Nx+1)*Ny + i
            enddo

            do i = ITE, Nx+1
               PN = PN + 1
               pnode(PN,1) = i
               pnode(PN,2) = (Nx+1)*Ny + i
            enddo
            Pmax = ILE + Nx + 2 - ITE

            do i = 1, ILE-1
               F1 = i
               F2 = Ny*(2*Nx+1) + i
               face(F1,2) = face(F2,2)
               face(F1,6) = face(F2,6)
               face(F2,1) = face(F1,1)
               face(F2,5) = face(F1,5)
            enddo

            do i = ITE, Nx
               F1 = i
               F2 = Ny*(2*Nx+1) + i
               face(F1,2) = face(F2,2)
               face(F1,6) = face(F2,6)
               face(F2,1) = face(F1,1)
               face(F2,5) = face(F1,5)
            enddo

c**   the rest of the edge nodes and faces

c**   lower edge of airfoil
            do i = ILE, ITE-1
               EN = EN + 1
               node = (Nx+1)*Ny + i
               enode(EN) = node
               dxP = x(node+1) - x(node)
```

```
               dyP = y(node+1) - y(node)
               if (i .eq. ILE) then
                  dxM = x(ILE+1) - x(node)
                  dyM = y(ILE+1) + pitch - y(node)
               else
                  dxM = x(node-1) - x(node)
                  dyM = y(node-1) - y(node)
               endif
               if ((dxM*dxP + dyM*dyP) .gt. 0.) then
                  dsM = sqrt(dxM**2 + dyM**2)
                  dsP = sqrt(dxP**2 + dyP**2)
                  dx = dxM/dsM + dxP/dsP
                  dy = dyM/dsM + dyP/dsP
               else
                  dx = dxP - dxM
                  dy = dyP - dyM
               endif
               senode(EN) = dy/sqrt(dx**2 + dy**2)
               cenode(EN) = dx/sqrt(dx**2 + dy**2)
               FF = Ny*(2*Nx+1) + 1
               do n = 1, 6
                  face(FF-ILE+1,n) = face(FF,n)
               enddo
               EF = EF + 1
               eface(EF) = FF - ILE + 1
            enddo

            do FF = Ny*(2*Nx+1)+Nx+1, Fmax
               CC = face(FF,1)
               cell(CC,1) = FF-Nx+ITE-ILE
               do n = 1, 6
                  face(FF-Nx+ITE-ILE,n) = face(FF,n)
               enddo
            enddo

            Fmax = Fmax - Nx + ITE - ILE

            EN = EN + 1
            node = (Nx+1)*Ny + ITE
            enode(EN) = node
            dxM = x(node-1) - x(node)
            dyM = y(node-1) - y(node)
            dxP = x(ITE-1) - x(node)
            dyP = y(ITE-1) + pitch - y(node)
            if ((dxM*dxP + dyM*dyP) .gt. 0.) then
               dsM = sqrt(dxM**2 + dyM**2)
               dsP = sqrt(dxP**2 + dyP**2)
               dx = dxM/dsM + dxP/dsP
               dy = dyM/dsM + dyP/dsP
            else
               dx = dxP - dxM
               dy = dyP - dyM
            endif
            senode(EN) = dy/sqrt(dx**2 + dy**2)
            cenode(EN) = dx/sqrt(dx**2 + dy**2)

            tnode = EN

c**   upper edge of airfoil
            do i = ILE, ITE-1
               EN = EN + 1
               EF = EF + 1
               eface(EF) = i
               node =  i
               enode(EN) = node
               dxP = x(node+1) - x(node)
               dyP = y(node+1) - y(node)
               if (i .eq. ILE) then
                  dxM = x((Nx+1)*Ny+ILE+1) - x(node)
                  dyM = y((Nx+1)*Ny+ILE+1) - pitch - y(node)
               else
                  dxM = x(node-1) - x(node)
```

```
                  dyM = y(node-1) - y(node)
               endif
               if ((dxM*dxP + dyM*dyP) .gt. 0.) then
                  dsM = sqrt(dxM**2 + dyM**2)
                  dsP = sqrt(dxP**2 + dyP**2)
                  dx = dxM/dsM + dxP/dsP
                  dy = dyM/dsM + dyP/dsP
               else
                  dx = dxP - dxM
                  dy = dyP - dyM
               endif
               senode(EN) = dy/sqrt(dx**2 + dy**2)
               cenode(EN) = dx/sqrt(dx**2 + dy**2)
            enddo

            EN = EN + 1
            node = ITE
            enode(EN) = node
            dxM = x(node-1) - x(node)
            dyM = y(node-1) - y(node)
            dxP = x((Nx+1)*Ny+ITE-1) - x(node)
            dyP = y((Nx+1)*Ny+ITE-1) - pitch - y(node)
            if ((dxM*dxP + dyM*dyP) .gt. 0.) then
               dsM = sqrt(dxM**2 + dyM**2)
               dsP = sqrt(dxP**2 + dyP**2)
               dx = dxM/dsM + dxP/dsP
               dy = dyM/dsM + dyP/dsP
            else
               dx = dxP - dxM
               dy = dyP - dyM
            endif
            senode(EN) = dy/sqrt(dx**2 + dy**2)
            cenode(EN) = dx/sqrt(dx**2 + dy**2)

            bnode = EN
            Wmax = 2*(ITE-ILE)

         endif

c**   inlet and outlet boundaries
         do j = 1, Ny
            EN = EN + 1
            enode(EN) = (j-1)*(Nx+1) + 1
            IN = IN + 1
            innode(IN) = (j-1)*(Nx+1) + 1
            EF = EF + 1
            eface(EF) = (2*Nx+1)*j - Nx
         enddo

         EN = EN + 1
         enode(EN) = Ny*(Nx+1) + 1
         IN = IN + 1
         innode(IN) = Ny*(Nx+1) + 1

         inode = bnode + Ny + 1
         Imax = IN

         do j = 1, Ny
            EN = EN + 1
            enode(EN) = (Nx+1)*j
            ON = ON + 1
            outnode(ON) = (Nx+1)*j
            EF = EF + 1
            eface(EF) = (2*Nx+1)*j
         enddo

         EN = EN + 1
         enode(EN) = (Nx+1)*(Ny+1)
         ON = ON + 1
         outnode(ON) = (Nx+1)*(Ny+1)

         ENmax = EN
```

```
      Emax = EF
      Cmax = CN

      return
      end



c*****************************************************************
c*                                                               *
c*      renumber edges so nodes are consecutive along edges      *
c*      and N3 is the interior node                              *
c*                                                               *
c*****************************************************************

      subroutine edgenumber
      implicit none

      include 'TRI.INC'
      integer EF, FF, i                !pointers
      integer save                     !save value in face

      do EF = 1, Emax
         FF = eface(EF)
         if (face(FF,1).eq.0) then
            do i = 1, 5, 2
               save = face(FF,i)
               face(FF,i) = face(FF,i+1)
               face(FF,i+1) = save
            enddo
         endif
      enddo

      return
      end



      subroutine cellcolor
      implicit none

      include 'TRI.INC'

      integer CC, NN, FF               !pointers
      integer Ncells                   !number of cells colored
      integer CC2(0:Maxcells)          !new number of cell
      integer Ncolor                   !current color
      integer cell2(Maxcells,6)              !new cell number
      logical hascolor(Maxcells)       !has node been colored current color?

c**   Initialize everything
      Ncells = 0
      do CC = 1, Cmax
         CC2(CC) = 0
      enddo

c**   Loop over colors
      do Ncolor = 1, 50

c**   Initialize node color array
         NCcolor(Ncolor) = 0
         do NN = 1, Nmax
            hascolor(NN) = .FALSE.
         enddo

c**   Loop over cells
         do CC = 1, Cmax

c**   Check if cell is already colored with old color
```

```
            if(CC2(CC).NE.0) GOTO 22
c**   Check if cell nodes are already colored with new color
            if( hascolor(cell(CC,4)) .OR.
     &          hascolor(cell(CC,5)) .OR.
     &          hascolor(cell(CC,6)) ) GOTO 22

c**   Set color markers
            Ncells = Ncells + 1
            CC2(CC) = Ncells
            NCcolor(Ncolor) = NCcolor(Ncolor) + 1
            hascolor(cell(CC,4)) = .TRUE.
            hascolor(cell(CC,5)) = .TRUE.
            hascolor(cell(CC,6)) = .TRUE.

 22       enddo

          if(Ncells.EQ.Cmax) GOTO 23

       enddo

       STOP 'COLOR; more than 50 colors required for cells'

 23    Ccolormax= Ncolor

c**   Redo pointers
       do CC = 1, Cmax
          cell2(CC2(CC),1) = cell(CC,1)
          cell2(CC2(CC),2) = cell(CC,2)
          cell2(CC2(CC),3) = cell(CC,3)
          cell2(CC2(CC),4) = cell(CC,4)
          cell2(CC2(CC),5) = cell(CC,5)
          cell2(CC2(CC),6) = cell(CC,6)
       enddo

       do CC = 1, Cmax
          cell(CC,1) = cell2(CC,1)
          cell(CC,2) = cell2(CC,2)
          cell(CC,3) = cell2(CC,3)
          cell(CC,4) = cell2(CC,4)
          cell(CC,5) = cell2(CC,5)
          cell(CC,6) = cell2(CC,6)
       enddo

       CC2(0) = 0

       do FF = 1, Fmax
          face(FF,1) = CC2(face(FF,1))
          face(FF,2) = CC2(face(FF,2))
       enddo

       return
       end




       subroutine facecolor
       implicit none

       include 'TRI.INC'
       integer FF, NN, CC, EF          !pointers
       integer Nfaces                  !number of faces colored
       integer FF2(Maxfaces)           !new number of face
       integer Ncolor                  !current color
       integer face2(Maxfaces,6)       !new face number
       logical hascolor(0:Maxfaces)    !has node been colored current color?

c**   Initialize everything
       Nfaces = 0
```

```
      do FF = 1, Fmax
         FF2(FF) = 0
      enddo

c**   Loop over colors
      do Ncolor = 1, 50

c**   Initialize node color array
         NFcolor(Ncolor) = 0
         do NN = 1, Nmax
            hascolor(NN) = .FALSE.
         enddo

c**   Loop over faces
         do FF = 1, Fmax

            hascolor(0) = .FALSE.

c**   Check if face is already colored with old color
            if(FF2(FF).NE.0) GOTO 22

c**   Check if face nodes are already colored with new color
            if( hascolor(face(FF,3)) .OR.
     &          hascolor(face(FF,4)) .OR.
     &          hascolor(face(FF,5)) .OR.
     &          hascolor(face(FF,6)) ) GOTO 22

c**   Set color markers
            Nfaces = Nfaces + 1
            FF2(FF) = Nfaces
            NFcolor(Ncolor) = NFcolor(Ncolor) + 1
            hascolor(face(FF,3)) = .TRUE.
            hascolor(face(FF,4)) = .TRUE.
            hascolor(face(FF,5)) = .TRUE.
            hascolor(face(FF,6)) = .TRUE.

 22      enddo

         if(Nfaces.EQ.Fmax) GOTO 23

      enddo

      STOP 'COLOR; more than 50 colors required for faces'

 23   Fcolormax= Ncolor

c**   Redo pointers
      do FF = 1, Fmax
         face2(FF2(FF),1) = face(FF,1)
         face2(FF2(FF),2) = face(FF,2)
         face2(FF2(FF),3) = face(FF,3)
         face2(FF2(FF),4) = face(FF,4)
         face2(FF2(FF),5) = face(FF,5)
         face2(FF2(FF),6) = face(FF,6)
      enddo

      do FF = 1, Fmax
         face(FF,1) = face2(FF,1)
         face(FF,2) = face2(FF,2)
         face(FF,3) = face2(FF,3)
         face(FF,4) = face2(FF,4)
         face(FF,5) = face2(FF,5)
         face(FF,6) = face2(FF,6)
      enddo

      do CC = 1, Cmax
         cell(CC,1) = FF2(cell(CC,1))
         cell(CC,2) = FF2(cell(CC,2))
         cell(CC,3) = FF2(cell(CC,3))
      enddo

      do EF = 1, Emax
```

```
         eface(EF) = FF2(eface(EF))
      enddo

      return
      end



      subroutine edgecolor
      implicit none

      include 'TRI.INC'
      integer FF, NN, EF                !pointers
      integer Nedges                    !number of faces colored
      integer EF2(Maxedges)             !new number of face
      integer Ncolor                    !current color
      integer eface2(Maxedges)          !new face number
      logical hascolor(Maxnodes)        !has node been colored current color?

c**   Initialize everything
      Nedges = 0
      do EF = 1, Emax
         EF2(EF) = 0
      enddo

c**   Loop over colors
      do Ncolor = 1, 5

c**   Initialize node color array
         NEcolor(Ncolor) = 0
         do NN = 1, Nmax
            hascolor(NN) = .FALSE.
         enddo

c**   Loop over faces
         do EF = 1, Wmax
            FF = eface(EF)

c**   Check if face is already colored with old color
            if(EF2(EF).NE.0) GOTO 22

c**   Check if face nodes are already colored with new color
            if( hascolor(face(FF,3)) .OR.
     &          hascolor(face(FF,4)) .OR.
     &          hascolor(face(FF,5)) ) GOTO 22

c**   Set color markers
            Nedges = Nedges + 1
            EF2(EF) = Nedges
            NEcolor(Ncolor) = NEcolor(Ncolor) + 1
            hascolor(face(FF,3)) = .TRUE.
            hascolor(face(FF,4)) = .TRUE.
            hascolor(face(FF,5)) = .TRUE.

 22         enddo

            if(Nedges.EQ.Wmax) GOTO 23

         enddo

         STOP 'COLOR; more than 5 colors required for wall edges'

 23      Elcolormax = Ncolor

c**   Loop over colors
      do Ncolor = Elcolormax+1, 10

c**   Initialize node color array
         NEcolor(Ncolor) = 0
         do NN = 1, Nmax
            hascolor(NN) = .FALSE.
```

```
          enddo

c**   Loop over faces
          do EF = Wmax+1, Emax
             FF = eface(EF)

c**   Check if face is already colored with old color
             if(EF2(EF).NE.0) GOTO 32

c**   Check if face nodes are already colored with new color
             if( hascolor(face(FF,3)) .OR.
     &           hascolor(face(FF,4)) .OR.
     &           hascolor(face(FF,5)) ) GOTO 32

c**   Set color markers
             Nedges = Nedges + 1
             EF2(EF) = Nedges
             NEcolor(Ncolor) = NEcolor(Ncolor) + 1
             hascolor(face(FF,3)) = .TRUE.
             hascolor(face(FF,4)) = .TRUE.
             hascolor(face(FF,5)) = .TRUE.

 32       enddo

          if(Nedges.EQ.Emax) GOTO 33

       enddo

       STOP 'COLOR; more than 10 colors required for all edges'

 33    E2colormax= Ncolor

c**   Redo pointers
       do EF = 1, Emax
          eface2(EF2(EF)) = eface(EF)
       enddo

       do EF = 1, Emax
          eface(EF) = eface2(EF)
       enddo

       return
       end
```

A.1.3 Ni Scheme

```
c*****************************************************************
c*                                                               *
c*      main program for triangular Ni scheme                    *
c*                                                               *
c*****************************************************************

       program triangle
       implicit none

       include 'TRI.INC'
       integer Niter            !number of iterations
       integer NN, CC, i        !pointer
       integer N1, N2, N3       !nodes at corners of cell
       real maxchange           !max change in state vector
       integer maxnode, maxeqn !where max change occurs
```

```
      real onethird           !one divided by three

c*  read in data from file
      call gridio(1)
      call flowio(1)
      call input

c*  calculate control area around each cell
      call calcarea

      Niter = 0
      durms = 999.
      onethird = 1./3.

c*  start history file from the top
      open(unit=35,status='unknown',form='formatted')
      write(35,2) Minl
      close(unit=35)
 2    format(' inlet Mach number = ',f5.3)

c*  loop until converged
      do while ((Niter.lt.Maxiter) .and. (durms.gt.2.e-7))
         Niter = Niter + 1

c*  set cell values of state vector
         do CC = 1, Cmax
            N1 = cell(CC,4)
            N2 = cell(CC,5)
            N3 = cell(CC,6)
            Uc(1,CC) = (U(1,N1) + U(1,N2) + U(1,N3))*onethird
            Uc(2,CC) = (U(2,N1) + U(2,N2) + U(2,N3))*onethird
            Uc(3,CC) = (U(3,N1) + U(3,N2) + U(3,N3))*onethird
            Uc(4,CC) = (U(4,N1) + U(4,N2) + U(4,N3))*onethird
         enddo

c*  calculate time step for each cell
         call timestep

c*  calculate flux at each node
         call nodeflux

c*  calculate change in state vector and fluxes at each cell
         call delcell

c*  calculate change in state vector at each node
         call delstate

c*  add smoothing term
         call smooth

c**  account for periodic nodes
         call bperiodic

c*  set inlet and outlet boundary conditions
         call binlet
         call boutlet

c*  change momentum change to make flow tangent at walls
         call tangent

c*  update state vector value
         do NN = 1, Nmax
            U(1,NN) = U(1,NN) + dU(1,NN)
            U(2,NN) = U(2,NN) + dU(2,NN)
            U(3,NN) = U(3,NN) + dU(3,NN)
            U(4,NN) = U(4,NN) + dU(4,NN)
         enddo

c*  find root mean square difference in state vector
         if (mod(Niter,10).eq.0 .or. Niter.lt.10) then
            durms = 0.0
            maxchange = 0.
```

```
            do NN = 1, Nmax
               do i = 1, 4
                  durms = durms + dU(i,NN)**2
                  if (abs(dU(i,NN)).gt.abs(maxchange)) then
                     maxchange = dU(i,NN)
                     maxnode = NN
                     maxeqn = i
                  endif
               enddo
            enddo

            durms = sqrt(durms/(4.*Nmax))

c*  print diagnostics to screen
            call flowio(0)
            open(unit=50, status='unknown', form='unformatted')
            write(50) Cmax, (vol(CC),CC=1,Cmax)
            close(unit=50)

 10         open(unit=35, status='old',access='append',err=10)
            write(35,1) Niter, durms, maxchange, x(maxnode),
     &                  y(maxnode), maxeqn
            close(unit=35)
            write(6,1) Niter, durms, maxchange, x(maxnode),
     &                  y(maxnode), maxeqn
         endif
      enddo
 1    format('Niter=',i4,'  rms=',f9.7,'  max=',f9.7,'  x=',
     &                  f6.3,'  y=',f6.3,'  eqn=',i1)


c*  write out data to file
      call flowio(0)

      stop
      end




c*********************************************************************
c*                                                                   *
c*      calculate areas of cells and distribute to nodal area        *
c*      which is made up of the sum of the cell areas                *
c*      surrounding the node                                         *
c*                                                                   *
c*********************************************************************

      subroutine calcarea
      implicit none

      include 'TRI.INC'
      integer CC                        !pointer
      integer N1, N2, N3                !nodes at corner of cell
      real dx12, dx31, dy12, dy31       !length of cell edges

c* calculate area of cells
      do CC = 1, Cmax
         N1 = cell(CC,4)
         N2 = cell(CC,5)
         N3 = cell(CC,6)
         dx12 = x(N1) - x(N2)
         dx31 = x(N3) - x(N1)
         dy12 = y(N1) - y(N2)
         dy31 = y(N3) - y(N1)
         areaC(CC) = abs(dx31*dy12 - dy31*dx12)
      enddo

      return
      end
```

```
c*********************************************************************
c*                                                                   *
c*      calculate time step for nodes                                *
c*                                                                   *
c*********************************************************************

        subroutine timestep
        implicit none

        include 'TRI.INC'
        real onethird                   !one divided by three
        integer NN, CC, PN, EN          !pointer
        integer N1, N2, N3              !nodes related to face
        integer P1, P2                  !periodic nodes
        real delx, dely                 !x and y length of side
        real delside                    !length of side
        real uu, vv, aa, rr, vv2        !values at node
        integer CC1, CC2, col           !pointers

        onethird = 1./3.

        do NN = 1, Nmax
           deltN(NN) = 0.
        enddo

c**  find time step for each cell
c**  delt is realy delt/area
        CC2 = 0

        do col = 1, Ccolormax
           CC1 = CC2 + 1
           CC2 = CC1 - 1 + NCcolor(col)

CVD$    NODEPCHK
           do CC = CC1, CC2
              N1 = cell(CC,4)
              N2 = cell(CC,5)
              N3 = cell(CC,6)

              rr = Uc(1,CC)
              uu = Uc(2,CC)/rr
              vv = Uc(3,CC)/rr
              vv2 = uu**2 + vv**2
              aa = sqrt(gam*gam1*(Uc(4,CC)/rr - 0.5*vv2))

              delx = x(N2) - x(N1)
              dely = y(N2) - y(N1)
              delside = sqrt(delx**2 + dely**2)

              deltC(CC) = abs(uu*dely-vv*delx) + aa*delside

              delx = x(N3) - x(N2)
              dely = y(N3) - y(N2)
              delside = sqrt(delx**2 + dely**2)

              deltC(CC) = deltC(CC) + abs(uu*dely-vv*delx) + aa*delside

              delx = x(N1) - x(N3)
              dely = y(N1) - y(N3)
              delside = sqrt(delx**2 + dely**2)

              deltC(CC) = deltC(CC) + abs(uu*dely-vv*delx) + aa*delside

              deltC(CC) = CFL*2./deltC(CC)

              deltN(N1) = deltN(N1) + onethird/deltC(CC)
              deltN(N2) = deltN(N2) + onethird/deltC(CC)
```

```
            deltN(N3) = deltN(N3) + onethird/deltC(CC)

         enddo
      enddo

CVD$  NODEPCHK
      do PN = 1, Pmax
         P1 = pnode(PN,1)
         P2 = pnode(PN,2)
         deltN(P1) = deltN(P1) + deltN(P2)
         deltN(P2) = deltN(P1)
      enddo

      do NN = 1, Nmax
         deltN(NN) = 1./deltN(NN)
      enddo

      return
      end




c*********************************************************************
c*                                                                   *
c*      calculate flux vector values at nodes                        *
c*                                                                   *
c*********************************************************************

      subroutine nodeflux
      implicit none

      include 'TRI.INC'
      integer NN                        !pointers
      real WW                           !kinetic energy

      do NN = 1, Nmax
c*  calculate f and g at nodes
         WW = 0.5*(U(2,NN)**2 + U(3,NN)**2)/U(1,NN)
         F(1,NN) = U(2,NN)
         F(2,NN) = U(2,NN)**2/U(1,NN) + gam1*(U(4,NN) - WW)
         F(3,NN) = U(2,NN)*U(3,NN)/U(1,NN)
         F(4,NN) = (U(2,NN)/U(1,NN))*(gam*U(4,NN) - gam1*WW)

         G(1,NN) = U(3,NN)
         G(2,NN) = U(2,NN)*U(3,NN)/U(1,NN)
         G(3,NN) = U(3,NN)**2/U(1,NN) + gam1*(U(4,NN) - WW)
         G(4,NN) = (U(3,NN)/U(1,NN))*(gam*U(4,NN) - gam1*WW)
      enddo

      return
      end




c*********************************************************************
c*                                                                   *
c*      this subroutine finds the change in the fluxes               *
c*      F and G and the change in the state U at each cell           *
c*                                                                   *
c*********************************************************************

      subroutine delcell
      implicit none

      include 'TRI.INC'
      integer CC                        !pointers
      integer N1, N2, N3                !nodes at corners of cells
```

```
      real u1, u2, u3, v1, v2, v3      !values at nodes
      real coef
      real dx23, dx31, dx12            !change in x on cell edges
      real dy23, dy31, dy12            !chagne in y on cell edges

c*  find change in state vector for cell
      do CC = 1, Cmax
         N1 = cell(CC,4)
         N2 = cell(CC,5)
         N3 = cell(CC,6)

         u1 = U(2,N1)/U(1,N1)
         u2 = U(2,N2)/U(1,N2)
         u3 = U(2,N3)/U(1,N3)
         v1 = U(3,N1)/U(1,N1)
         v2 = U(3,N2)/U(1,N2)
         v3 = U(3,N3)/U(1,N3)

         dy23 = y(N2) - y(N3)
         dy31 = y(N3) - y(N1)
         dy12 = y(N1) - y(N2)
         dx23 = x(N2) - x(N3)
         dx31 = x(N3) - x(N1)
         dx12 = x(N1) - x(N2)

         coef = 0.5*deltC(CC)

         dUc(1,CC) = coef*(-F(1,N1)*dy23 + G(1,N1)*dx23
     &                     -F(1,N2)*dy31 + G(1,N2)*dx31
     &                     -F(1,N3)*dy12 + G(1,N3)*dx12)

         dUc(2,CC) = coef*(-F(2,N1)*dy23 + G(2,N1)*dx23
     &                     -F(2,N2)*dy31 + G(2,N2)*dx31
     &                     -F(2,N3)*dy12 + G(2,N3)*dx12)

         dUc(3,CC) = coef*(-F(3,N1)*dy23 + G(3,N1)*dx23
     &                     -F(3,N2)*dy31 + G(3,N2)*dx31
     &                     -F(3,N3)*dy12 + G(3,N3)*dx12)

         dUc(4,CC) = coef*(-F(4,N1)*dy23 + G(4,N1)*dx23
     &                     -F(4,N2)*dy31 + G(4,N2)*dx31
     &                     -F(4,N3)*dy12 + G(4,N3)*dx12)

c**  shock smoothing
         vol(CC) = u1*dy23 - v1*dx23 +
     &             u2*dy31 - v2*dx31 +
     &             u3*dy12 - v3*dx12
      enddo

c*  implement wall boundary conditions
      call bwall

      return
      end




c*********************************************************************
c*                                                                   *
c*      this subroutine applies the wall boundary condition          *
c*                                                                   *
c*********************************************************************

      subroutine bwall
      implicit none

      include 'TRI.INC'
      integer EF, FF, CC       !pointer
      integer N1, N3           !nodes on edge
      real u1, u3, v1, v3      !values at nodes
```

```
        real coef
        real dy13, dx13          !change in x and y
        real pres1, pres3        !pressure
        integer EF1, EF2, col

c*  upper and lower boundary
        EF2 = 0

        do col = 1, Elcolormax
         EF1 = EF2 + 1
         EF2 = EF1 - 1 + NEcolor(col)

CVD$    NODEPCHK
         do EF = EF1, EF2
           FF = eface(EF)
           CC = face(FF,1)
           N1 = face(FF,4)
           N3 = face(FF,3)

           u1 = U(2,N1)/U(1,N1)
           u3 = U(2,N3)/U(1,N3)
           v1 = U(3,N1)/U(1,N1)
           v3 = U(3,N3)/U(1,N3)

           coef = 0.5*deltC(CC)

           dy13 = y(N1) - y(N3)
           dx13 = x(N1) - x(N3)

           pres1 = gam1*(U(4,N1) - 0.5*(U(2,N1)**2 + U(3,N1)**2)/
     &          U(1,N1))
           pres3 = gam1*(U(4,N3) - 0.5*(U(2,N3)**2 + U(3,N3)**2)/
     &          U(1,N3))

           dUc(1,CC) = dUc(1,CC) + coef*(
     &                 (F(1,N3) + F(1,N1))*dy13 -
     &                 (G(1,N3) + G(1,N1))*dx13)

           dUc(2,CC) = dUc(2,CC) + coef*(
     &                 (F(2,N3) + F(2,N1) - pres1 - pres3)*dy13 -
     &                 (G(2,N3) + G(2,N1))*dx13)

           dUc(3,CC) = dUc(3,CC) + coef*(
     &                 (F(3,N3) + F(3,N1))*dy13 -
     &                 (G(3,N3) + G(3,N1) - pres1 - pres3)*dx13)

           dUc(4,CC) = dUc(4,CC) + coef*(
     &                 (F(4,N3) + F(4,N1))*dy13 -
     &                 (G(4,N3) + G(4,N1))*dx13)

c** shock smoothing
           vol(CC) = vol(CC) - (u1 + u3)*dy13 + (v1 + v3)*dx13

         enddo
        enddo

        return
        end




c*****************************************************************
c*                                                               *
c*      this subroutine calculates the change at each node       *
c*                                                               *
c*****************************************************************

        subroutine delstate
        implicit none
```

```
      include 'TRI.INC'
      integer NN, CC, FF, EF          !pointers
      integer N1, N2, N3              !nodes at corners of cell
      real rr, uu, vv, vv2            !values at cell
      real rdu, rdv                   !part of dFc and dGc
      real HH, dp
      real dFc(4), dGc(4)
      real dy21, dy32, dy13           !change in y on cell edges
      real dx21, dx32, dx13           !change in x on cell edges
      real COH                        !coefficient
      integer CC1, CC2, col           !pointers
      integer EF1, EF2
      real coef                       !coefficient
      real onethird                   !one divided by three

      do NN = 1, Nmax
         dU(1,NN) = 0.
         dU(2,NN) = 0.
         dU(3,NN) = 0.
         dU(4,NN) = 0.
      enddo

      onethird = 1./3.

      CC2 = 0

      do col = 1, Ccolormax
       CC1 = CC2 + 1
       CC2 = CC1 - 1 + NCcolor(col)

CVD$  NODEPCHK
       do CC = CC1, CC2
         N1 = cell(CC,4)
         N2 = cell(CC,5)
         N3 = cell(CC,6)

         rr = Uc(1,CC)
         uu = Uc(2,CC)/rr
         vv = Uc(3,CC)/rr
         vv2 = 0.5*(uu*uu + vv*vv)

         dy21 = y(N2) - y(N1)
         dy32 = y(N3) - y(N2)
         dy13 = y(N1) - y(N3)
         dx21 = x(N2) - x(N1)
         dx32 = x(N3) - x(N2)
         dx13 = x(N1) - x(N3)

c*  find second order change in flux vector at cells
         HH = gam*Uc(4,CC)/Uc(1,CC) - gam1*vv2
         rdu = dUc(2,CC) - uu*dUc(1,CC)
         rdv = dUc(3,CC) - vv*dUc(1,CC)
         dp = gam1*(dUc(4,CC) - uu*dUc(2,CC) - vv*dUc(3,CC)
     &             + vv2*dUc(1,CC))

         dFc(1) = dUc(2,CC)
         dFc(2) = uu*(dUc(2,CC) + rdu) + dp
         dFc(3) = vv*dUc(2,CC) + uu*rdv
         dFc(4) = uu*(dUc(4,CC) + dp) + HH*rdu

         dGc(1) = dUc(3,CC)
         dGc(2) = uu*dUc(3,CC) + vv*rdu
         dGc(3) = vv*(dUc(3,CC) + rdv) + dp
         dGc(4) = vv*(dUc(4,CC) + dp) + HH*rdv

c**  shock smoothing
c        COH = 0.25*min(0.,vol(CC))*vol(CC)*vv2/areaC(CC)
         vol(CC) = vol(CC)/sqrt(2.*areaC(CC))
         COH = epsicoef*0.5*max(-0.1,min(0.,vol(CC)))*vol(CC)*vv2
c        vol(CC) = -vol(CC)/sqrt(2.*areaC(CC))
c        COH = epsicoef*0.5*max(-0.1,min(0.,vol(CC)))*vol(CC)*vv2
         dFc(2) = dFc(2) + COH
```

```
      dGc(3) = dGc(3) + CON

      coef = onethird/deltC(CC)

      dU(1,N1) = dU(1,N1) + coef*dUc(1,CC)
     &             - 0.25*(dFc(1)*dy32 - dGc(1)*dx32)
      dU(1,N2) = dU(1,N2) + coef*dUc(1,CC)
     &             - 0.25*(dFc(1)*dy13 - dGc(1)*dx13)
      dU(1,N3) = dU(1,N3) + coef*dUc(1,CC)
     &             - 0.25*(dFc(1)*dy21 - dGc(1)*dx21)

      dU(2,N1) = dU(2,N1) + coef*dUc(2,CC)
     &             - 0.25*(dFc(2)*dy32 - dGc(2)*dx32)
      dU(2,N2) = dU(2,N2) + coef*dUc(2,CC)
     &             - 0.25*(dFc(2)*dy13 - dGc(2)*dx13)
      dU(2,N3) = dU(2,N3) + coef*dUc(2,CC)
     &             - 0.25*(dFc(2)*dy21 - dGc(2)*dx21)

      dU(3,N1) = dU(3,N1) + coef*dUc(3,CC)
     &             - 0.25*(dFc(3)*dy32 - dGc(3)*dx32)
      dU(3,N2) = dU(3,N2) + coef*dUc(3,CC)
     &             - 0.25*(dFc(3)*dy13 - dGc(3)*dx13)
      dU(3,N3) = dU(3,N3) + coef*dUc(3,CC)
     &             - 0.25*(dFc(3)*dy21 - dGc(3)*dx21)

      dU(4,N1) = dU(4,N1) + coef*dUc(4,CC)
     &             - 0.25*(dFc(4)*dy32 - dGc(4)*dx32)
      dU(4,N2) = dU(4,N2) + coef*dUc(4,CC)
     &             - 0.25*(dFc(4)*dy13 - dGc(4)*dx13)
      dU(4,N3) = dU(4,N3) + coef*dUc(4,CC)
     &             - 0.25*(dFc(4)*dy21 - dGc(4)*dx21)

    enddo
   enddo

   EF2 = 0
   do col = 1, E2colormax
    EF1 = EF2 + 1
    EF2 = EF1 - 1 + Necolor(col)
    do EF = EF1, EF2
      FF = eface(EF)
      CC = face(FF,1)
      N1 = face(FF,3)
      N2 = face(FF,4)

      dy21 = y(N2) - y(N1)
      dx21 = x(N2) - x(N1)

      uu = Uc(2,CC)/Uc(1,CC)
      vv = Uc(3,CC)/Uc(1,CC)
      ww2 = 0.5*(uu*uu + vv*vv)

      dp = gam1*(dUc(4,CC) - uu*dUc(2,CC) - vv*dUc(3,CC)
     &          + ww2*dUc(1,CC))

      dFc(2) = dp
      dGc(3) = dp

      dU(2,N1) = dU(2,N1) - 0.25*dFc(2)*dy21
      dU(2,N2) = dU(2,N2) - 0.25*dFc(2)*dy21

      dU(3,N1) = dU(3,N1) + 0.25*dGc(3)*dx21
      dU(3,N2) = dU(3,N2) + 0.25*dGc(3)*dx21

    enddo
   enddo

   do NN = 1, Nmax
      dU(1,NN) = dU(1,NN)*deltN(NN)
      dU(2,NN) = dU(2,NN)*deltN(NN)
      dU(3,NN) = dU(3,NN)*deltN(NN)
      dU(4,NN) = dU(4,NN)*deltN(NN)
```

```
      enddo

      return
      end



c*********************************************************************
c*                                                                   *
c*      subroutine to calculate forth difference smoothing           *
c*                                                                   *
c*********************************************************************

      subroutine smooth
      implicit none

      include 'TRI.INC'
      integer N1, N2, N3                !nodes at corners of cell
      integer NN, CC1, CC2, col, CC     !pointers
      integer EF, EF1, EF2
      integer FF, FF1, FF2, PN
      integer P1, P2                    !periodic nodes
      real dx31, dx12, dx23             !change in x
      real dy31, dy12, dy23             !change in y
      real dxC1, dxC2, dxC3, dxC4       !change in x in cell
      real dyC1, dyC2, dyC3, dyC4       !change in y in cell
      real delN(4,Maxnodes)             !gradient at node
      real coef, coef1, coef2           !coefficient

      do NN = 1, Nmax
         delN(1,NN) = 0.
         delN(2,NN) = 0.
         delN(3,NN) = 0.
         delN(4,NN) = 0.
      enddo

      CC2 = 0

      do col = 1, Ccolormax
       CC1 = CC2 + 1
       CC2 = CC1 - 1 + NCcolor(col)

CVD$    NODEPCHK
        do CC = CC1, CC2
          N1 = cell(CC,4)
          N2 = cell(CC,5)
          N3 = cell(CC,6)

          dx31 = x(N3) - x(N1)
          dx12 = x(N1) - x(N2)
          dx23 = x(N2) - x(N3)
          dy31 = y(N3) - y(N1)
          dy12 = y(N1) - y(N2)
          dy23 = y(N2) - y(N3)

          coef = ((max(0.,min(1.,(1.+10.*vol(CC))))-1.)*epsicoef +1.)
     &                *(0.5/areaC(CC))

          dxC1 = (U(1,N1)*dy23 + U(1,N2)*dy31 + U(1,N3)*dy12)
     &                *coef
          dxC2 = (U(2,N1)*dy23 + U(2,N2)*dy31 + U(2,N3)*dy12)
     &                *coef
          dxC3 = (U(3,N1)*dy23 + U(3,N2)*dy31 + U(3,N3)*dy12)
     &                *coef
          dxC4 = (U(4,N1)*dy23 + U(4,N2)*dy31 + U(4,N3)*dy12)
     &                *coef

          dyC1 = (U(1,N1)*dx23 + U(1,N2)*dx31 + U(1,N3)*dx12)
     &                *coef
          dyC2 = (U(2,N1)*dx23 + U(2,N2)*dx31 + U(2,N3)*dx12)
```

```
     &                    *coef
           dyC3 = (U(3,N1)*dx23 + U(3,N2)*dx31 + U(3,N3)*dx12)
     &                    *coef
           dyC4 = (U(4,N1)*dx23 + U(4,N2)*dx31 + U(4,N3)*dx12)
     &                    *coef

           delN(1,N1) = delN(1,N1) + (dxC1*dy23 + dyC1*dx23)
           delN(1,N2) = delN(1,N2) + (dxC1*dy31 + dyC1*dx31)
           delN(1,N3) = delN(1,N3) + (dxC1*dy12 + dyC1*dx12)

           delN(2,N1) = delN(2,N1) + (dxC2*dy23 + dyC2*dx23)
           delN(2,N2) = delN(2,N2) + (dxC2*dy31 + dyC2*dx31)
           delN(2,N3) = delN(2,N3) + (dxC2*dy12 + dyC2*dx12)

           delN(3,N1) = delN(3,N1) + (dxC3*dy23 + dyC3*dx23)
           delN(3,N2) = delN(3,N2) + (dxC3*dy31 + dyC3*dx31)
           delN(3,N3) = delN(3,N3) + (dxC3*dy12 + dyC3*dx12)

           delN(4,N1) = delN(4,N1) + (dxC4*dy23 + dyC4*dx23)
           delN(4,N2) = delN(4,N2) + (dxC4*dy31 + dyC4*dx31)
           delN(4,N3) = delN(4,N3) + (dxC4*dy12 + dyC4*dx12)
         enddo
        enddo

c*  upper and lower boundary
        EF2 = 0

        do col = 1, Elcolormax
         EF1 = EF2 + 1
         EF2 = EF1 - 1 + NEcolor(col)

CVD$    NODEPCHK
         do EF = EF1, EF2
           FF = eface(EF)
           CC = face(FF,1)
           N1 = face(FF,3)
           N2 = face(FF,4)
           N3 = face(FF,5)

           dx31 = x(N3) - x(N1)
           dx12 = x(N1) - x(N2)
           dx23 = x(N2) - x(N3)
           dy31 = y(N3) - y(N1)
           dy12 = y(N1) - y(N2)
           dy23 = y(N2) - y(N3)

           coef = ((max(0.,min(1.,(1.+10.*vol(CC))))-1.)*epsicoef +1.)
     &                *(0.5/areaC(CC))

           dxC1 = (U(1,N1)*dy23 + U(1,N2)*dy31 + U(1,N3)*dy12)
     &                    *coef
           dxC2 = (U(2,N1)*dy23 + U(2,N2)*dy31 + U(2,N3)*dy12)
     &                    *ccef
           dxC3 = (U(3,N1)*dy23 + U(3,N2)*dy31 + U(3,N3)*dy12)
     &                    *coef
           dxC4 = (U(4,N1)*dy23 + U(4,N2)*dy31 + U(4,N3)*dy12)
     &                    *coef

           dyC1 = (U(1,N1)*dx23 + U(1,N2)*dx31 + U(1,N3)*dx12)
     &                    *coef
           dyC2 = (U(2,N1)*dx23 + U(2,N2)*dx31 + U(2,N3)*dx12)
     &                    *coef
           dyC3 = (U(3,N1)*dx23 + U(3,N2)*dx31 + U(3,N3)*dx12)
     &                    *coef
           dyC4 = (U(4,N1)*dx23 + U(4,N2)*dx31 + U(4,N3)*dx12)
     &                    *coef

           delN(1,N1) = delN(1,N1) + (dxC1*dy12 + dyC1*dx12)
           delN(1,N2) = delN(1,N2) + (dxC1*dy12 + dyC1*dx12)

           delN(2,N1) = delN(2,N1) + (dxC2*dy12 + dyC2*dx12)
           delN(2,N2) = delN(2,N2) + (dxC2*dy12 + dyC2*dx12)
```

```
            delN(3,N1) = delN(3,N1) + (dxC3*dy12 + dyC3*dx12)
            delN(3,N2) = delN(3,N2) + (dxC3*dy12 + dyC3*dx12)

            delN(4,N1) = delN(4,N1) + (dxC4*dy12 + dyC4*dx12)
            delN(4,N2) = delN(4,N2) + (dxC4*dy12 + dyC4*dx12)

          enddo
        enddo

CVD$    NODEPCHK
        do PN = 1, Pmax
           P1 = pnode(PN,1)
           P2 = pnode(PN,2)
           delN(1,P1) = delN(1,P1) + delN(1,P2)
           delN(2,P1) = delN(2,P1) + delN(2,P2)
           delN(3,P1) = delN(3,P1) + delN(3,P2)
           delN(4,P1) = delN(4,P1) + delN(4,P2)
           delN(1,P2) = delN(1,P1)
           delN(2,P2) = delN(2,P1)
           delN(3,P2) = delN(3,P1)
           delN(4,P2) = delN(4,P1)
        enddo

        FF2 = 0

        do col = 1, Fcolormax
         FF1 = FF2 + 1
         FF2 = FF1 - 1 + NFcolor(col)

CVD$    NODEPCHK
         do FF = FF1, FF2
           N1 = face(FF,3)
           N2 = face(FF,4)

           coef1 = 0.5*eps2*(1./deltN(N1) + 1./deltN(N2))*deltN(N1)
           coef2 = 0.5*eps2*(1./deltN(N1) + 1./del N(N2))*deltN(N2)

           dU(1,N1) = dU(1,N1) + coef1*(delN(1,N2) - delN(1,N1))
           dU(1,N2) = dU(1,N2) + coef2*(delN(1,N1) - delN(1,N2))

           dU(2,N1) = dU(2,N1) + coef1*(delN(2,N2) - delN(2,N1))
           dU(2,N2) = dU(2,N2) + coef2*(delN(2,N1) - delN(2,N2))

           dU(3,N1) = dU(3,N1) + coef1*(delN(3,N2) - delN(3,N1))
           dU(3,N2) = dU(3,N2) + coef2*(delN(3,N1) - delN(3,N2))

           dU(4,N1) = dU(4,N1) + coef1*(delN(4,N2) - delN(4,N1))
           dU(4,N2) = dU(4,N2) + coef2*(delN(4,N1) - delN(4,N2))
         enddo
        enddo

        return
        end



c*****************************************************************
c*                                                               *
c*      this subroutine accounts for periodic nodes              *
c*                                                               *
c*****************************************************************

        subroutine bperiodic
        implicit none

        include 'TRI.INC'
        integer PN              !pointer
        integer P1, P2          !periodic nodes
```

```
CVD$    NODEPCHK
        do PN = 1, Pmax
           P1 = pnode(PN,1)
           P2 = pnode(PN,2)
           dU(1,P1) = dU(1,P1) + dU(1,P2)
           dU(2,P1) = dU(2,P1) + dU(2,P2)
           dU(3,P1) = dU(3,P1) + dU(3,P2)
           dU(4,P1) = dU(4,P1) + dU(4,P2)
           dU(1,P2) = dU(1,P1)
           dU(2,P2) = dU(2,P1)
           dU(3,P2) = dU(3,P1)
           dU(4,P2) = dU(4,P1)
        enddo

        return
        end



c*********************************************************************
c*                                                                   *
c*      adjust inlet state vector for boundary condition             *
c*                                                                   *
c*********************************************************************

        subroutine binlet
        implicit none

        include 'TRI.INC'
        real rrinl, uuinl, vvinl, ppinl !average values at inlet
        real vv2inl, aainl
        real r0inf, a0inf               !stag. density and speed of sound
        real H0pres                     !stagnation enthalpy and pressure
        real spres                      !entropy
        real coef
        real aaa1, aaa2, aaa3, aaa4     !coef. of inverse of matrix
        real bbb1, bbb2, bbb3, bbb4
        real ccc1, ccc2, ccc3, ccc4
        real ddd1, ddd2, ddd3, ddd4
        real vv2, uu, vv, pp, rr, aa    !values at next interior node
        real dH0, ds, dtan, dw4
        real drr, duu, dvv, dpp
        integer IN, NN                  !boundary face and its nodes

c**  average values at inlet
        rrinl = U(1,innode(Imax/2))
        uuinl = U(2,innode(Imax/2))/rrinl
        vvinl = U(3,innode(Imax/2))/rrinl
        vv2inl = uuinl**2 + vvinl**2
        ppinl = gam1*(U(4,innode(Imax/2))
     &              - 0.5*rrinl*vv2inl)
        aainl = sqrt(gam*ppinl/rrinl)

c*  subsonic inlet nodes
        if ((sqrt(vv2inl)/aainl).lt.1. .or. pitch.ne.0.) then
c**  prescribed values
           H0pres = 1./gam1
           spres = 0.

c**  coefficient of matrix
           coef = 1./(uuinl*(aainl+uuinl) + vvinl**2)
           aaa1 = (rrinl*uuinl/aainl)*coef
           aaa2 = -(ppinl/aainl)*(uuinl*gam/gam1 + vv2inl/aainl)*coef
           aaa3 = -(rrinl*vvinl/aainl)*coef
           aaa4 = (vv2inl/aainl**2)*coef
           bbb1 = uuinl*coef
           bbb2 = -(uuinl*ppinl/(gam1*rrinl))*coef
           bbb3 = -vvinl*coef
           bbb4 = -(uuinl/rrinl)*coef
           ccc1 = vvinl*coef
```

```
            ccc2 = -(vvinl*ppinl/(gam1*rrinl))*coef
            ccc3 = (aainl + uuinl)*coef
            ccc4 = -(vvinl/rrinl)*coef
            ddd1 = rrinl*aainl*uuinl*coef
            ddd2 = -(ppinl*aainl*uuinl/gam1)*coef
            ddd3 = -rrinl*aainl*vvinl*coef
            ddd4 = ww2inl*coef

CVD$     NODEPCHK
            do IN = 1, Imax
               NN = innode(IN)
               rr = U(1,NN)
               uu = U(2,NN)/rr
               vv = U(3,NN)/rr
               ww2 = uu**2 + vv**2
               pp = gam1*(U(4,NN) - 0.5*rr*ww2)
               aa = sqrt(gam*pp/rr)
               dH0  = H0pres - ((gam/gam1)*pp/rr + 0.5*ww2)
               ds   = spres - (log(gam*pp) - gam*log(rr))
               dtan = (Sinl - vv/uu)*uuinl**2
               dw4  =   dU(1,NN)*(aa*uu + gam1*ww2)
     &                      - dU(2,NN)*(aa + gam1*uu)
     &                      - dU(3,NN)*gam1*vv
     &                      + dU(4,NN)*gam1
               drr = aaa1*dH0 + aaa2*ds + aaa3*dtan + aaa4*dw4
               duu = bbb1*dH0 + bbb2*ds + bbb3*dtan + bbb4*dw4
               dvv = ccc1*dH0 + ccc2*ds + ccc3*dtan + ccc4*dw4
               dpp = ddd1*dH0 + ddd2*ds + ddd3*dtan + ddd4*dw4

               dU(1,NN) = drr
               dU(2,NN) = rr*duu + uu*drr
               dU(3,NN) = rr*dvv + vv*drr
               dU(4,NN) = dpp/gam1 + 0.5*ww2*drr + rr*(uu*duu + vv*dvv)
            enddo

c*  supersonic inlet nodes
         else if ((sqrt(ww2irl)/aainl).ge.1.) then
c*  state vector components far from body
            r0inf = 1.
            a0inf = 1.

CVD$     NODEPCHK
            do IN = 1, Imax
               NN = innode(IN)
               rr = r0inf*(1.0+0.5*gam1*Minl**2)**(-1./gam1)
               aa = a0inf*(1.0+0.5*gam1*Minl**2)**(-0.5)
               uu = Minl*aa
               pp = rr*aa**2/gam

               dU(1,NN) = rr - U(1,NN)
               dU(2,NN) = rr*uu - U(2,NN)
               dU(3,NN) = - U(3,NN)
               dU(4,NN) = pp/gam1 + .5*(uu**2)*rr - U(4,NN)
            enddo
         endif

         return
         end



c*****************************************************************
c*                                                               *
c*      adjust outlet state vector for boundary condition        *
c*                                                               *
c*****************************************************************

         subroutine boutlet
         implicit none

         include 'TRI.INC'
```

```
      real rrout, uuout, vvout, ppout !average values at outet
      real aaout, ww2out
      real aaa1, aaa4                 !coef. of inverse of matrix
      real bbb3, bbb4
      real ccc2
      real ddd4
      real ww2, uu, vv, pp, rr, aa    !values at next interior node
      real dw1, dw2, dw3, dp
      real drr, duu, dvv, dpp
      integer ON, NN                  !boundary face and its nodes

      rrout = U(1,outnode(Omax/2))
      uuout = U(2,outnode(Omax/2))/rrout
      vvout = U(3,outnode(Omax/2))/rrout
      ww2out = uuout**2 + vvout**2
      ppout = gam1*(U(4,outnode(Omax/2))
     &             - 0.5*rrout*ww2out)
      aaout = sqrt(gam*ppout/rrout)

      aaa1 = -1./aaout**2
      aaa4 = 1./aaout**2
      bbb3 = .1/(rrout*aaout)
      bbb4 = -1./(rrout*aaout)
      ccc2 = .1/(rrout*aaout)
      ddd4 = 1.

c*  set boundary values of state vector for nodes
CVD$    NODEPCHK
      do ON = 1, Omax
         NN = outnode(ON)
         rr = U(1,NN)
         uu = U(2,NN)/rr
         vv = U(3,NN)/rr
         ww2 = uu**2 + vv**2
         pp = gam1*(U(4,NN) - 0.5*rr*ww2)
         aa = sqrt(gam*pp/rr)
         if (sqrt(ww2) .lt. aa) then
            dw1 =   dU(1,NN)*(0.5*ww2*gam1 - aa**2)
     &            - dU(2,NN)*gam1*uu
     &            - dU(3,NN)*gam1*vv
     &            + dU(4,NN)*gam1
            dw2 = - dU(1,NN)*aa*vv
     &            + dU(3,NN)*aa
            dw3 =   dU(1,NN)*(0.5*ww2*gam1 - aa*uu)
     &            - dU(2,NN)*(gam1*uu - aa)
     &            - dU(3,NN)*gam1*vv
     &            + dU(4,NN)*gam1
            dp = pout - pp
            drr = aaa1*dw1 + aaa4*dp
            duu = bbb3*dw3 + bbb4*dp
            dvv = ccc2*dw2
            dpp = ddd4*dp
            dU(1,NN) = drr
            dU(2,NN) = rr*duu + uu*drr
            dU(3,NN) = rr*dvv + vv*drr
            dU(4,NN) = dpp/gam1 + 0.5*ww2*drr + rr*(uu*duu + vv*dvv)

         endif
      enddo

      return
      end



c*****************************************************************
c*                                                               *
c*      this subroutine changes the momentum change to make      *
c*      flow tangent to wall                                     *
c*                                                               *
```

```
c*********************************************************************

        subroutine tangent
        implicit none

        include 'TRI.INC'
        integer EN                          !pointer
        integer NN                          !node on wall
        real drwn                           !change in momentum normal to wall

CVD$    NODEPCHK
        do EN = 1, bnode
           NN = enode(EN)
           drwn = -(U(2,NN) + dU(2,NN))*senode(EN) +
     &             (U(3,NN) + dU(3,NN))*cenode(EN)
           dU(2,NN) = dU(2,NN) + drwn*senode(EN)
           dU(3,NN) = dU(3,NN) - drwn*cenode(EN)
        enddo

        return
        end
```

A.1.4 Jameson Scheme

```
c*********************************************************************
c*                                                                   *
c*      main program for triangular Jameson scheme                   *
c*                                                                   *
c*********************************************************************

        program triangle
        implicit none

        include 'TRI.INC'
        integer Niter                       !number of iterations
        real maxchange                      !max change in state vector
        integer CC                          !pointer
        integer maxnode, maxeqn             !where max change occurs

c*  read in data from file
        call gridio(1)
        call flowio(1)
        call input

        Niter = 0
        drms = 999.

c*  start history file from the top
        open(unit=35,status='unknown',form='formatted')
        write(35,2) Minl
        close(unit=35)
 2      format(' inlet Mach number = ',f5.3)

c*  loop until converged
        do while ((Niter.lt.Maxiter) .and. (drms.gt.2.e-7))
           Niter = Niter + 1

           call update(maxchange, maxnode, maxeqn)

           if (mod(Niter,10).eq.0 .or. Niter.lt.10) then
              call flowio(0)
              open(unit=50, status='unknown', form='unformatted')
              write(50) Cmax, (vol(CC),CC=1,Cmax)
              close(unit=50)

 10           open(unit=35, status='old',access='append',err=10)
              write(35,1) Niter, drms, maxchange, x(maxnode),
```

```
     &                 y(maxnode), maxeqn
            close(unit=35)
            write(6,1) Niter, durms, maxchange, x(maxnode),
     &                 y(maxnode), maxeqn
         endif

 1       format('Niter=',i4,'  rms=',f9.7,'  max=',f9.7,'  x=',f6.3,
     &          '  y=',f6.3,'  eqn=',i1)

        enddo

c*  write out data to file
        call flowio(0)

        stop
        end




c*****************************************************************
c*                                                               *
c*      update state vectors at next time step                   *
c*                                                               *
c*****************************************************************

        subroutine update(maxchange, maxnode, maxeqn)

        include 'TRI.INC'
        integer NN, i                   !pointer
        real UO(4,Maxnodes)             !starting values of state vector
        real alpha1, alpha2, alpha3     !coefficients
        real alpha4
        real maxchange          !max change in state vector
        integer maxnode, maxeqn         !where max change occurs

        alpha1 = 0.25
        alpha2 = 1./3.
        alpha3 = 0.5
        alpha4 = 1.

c*  find timestep at each node
        call timestep

c*  advance to next time step in four steps

c*  step one
        call calcflux
        call dissipation

        do NN = 1, Nmax
           UO(1,NN) = U(1,NN)
           UO(2,NN) = U(2,NN)
           UO(3,NN) = U(3,NN)
           UO(4,NN) = U(4,NN)
           dU(1,NN) = U(1,NN)
           dU(2,NN) = U(2,NN)
           dU(3,NN) = U(3,NN)
           dU(4,NN) = U(4,NN)
           U(1,NN) = UO(1,NN) - (alpha1*deltN(NN))
     &                          *(flux(1,NN) - dis(1,NN))
           U(2,NN) = UO(2,NN) - (alpha1*deltN(NN))
     &                          *(flux(2,NN) - dis(2,NN))
           U(3,NN) = UO(3,NN) - (alpha1*deltN(NN))
     &                          *(flux(3,NN) - dis(3,NN))
           U(4,NN) = UO(4,NN) - (alpha1*deltN(NN))
     &                          *(flux(4,NN) - dis(4,NN))
           dU(1,NN) = U(1,NN) - dU(1,NN)
           dU(2,NN) = U(2,NN) - dU(2,NN)
```

```
            dU(3,NN) = U(3,NN) - dU(3,NN)
            dU(4,NN) = U(4,NN) - dU(4,NN)
         enddo

         call binlet
         call boutlet
         call tangent

c*  step two
         call calcflux
         call dissipation

         do NN = 1, Nmax
            dU(1,NN) = U(1,NN)
            dU(2,NN) = U(2,NN)
            dU(3,NN) = U(3,NN)
            dU(4,NN) = U(4,NN)
            U(1,NN) = UO(1,NN) - (alpha2*deltN(NN))
     &                           *(flux(1,NN) - dis(1,NN))
            U(2,NN) = UO(2,NN) - (alpha2*deltN(NN))
     &                           *(flux(2,NN) - dis(2,NN))
            U(3,NN) = UO(3,NN) - (alpha2*deltN(NN))
     &                           *(flux(3,NN) - dis(3,NN))
            U(4,NN) = UO(4,NN) - (alpha2*deltN(NN))
     &                           *(flux(4,NN) - dis(4,NN))
            dU(1,NN) = U(1,NN) - dU(1,NN)
            dU(2,NN) = U(2,NN) - dU(2,NN)
            dU(3,NN) = U(3,NN) - dU(3,NN)
            dU(4,NN) = U(4,NN) - dU(4,NN)
         enddo

         call binlet
         call boutlet
         call tangent

c*  step three
         call calcflux

         do NN = 1, Nmax
            dU(1,NN) = U(1,NN)
            dU(2,NN) = U(2,NN)
            dU(3,NN) = U(3,NN)
            dU(4,NN) = U(4,NN)
            U(1,NN) = UO(1,NN) - (alpha3*deltN(NN))
     &                           *(flux(1,NN) - dis(1,NN))
            U(2,NN) = UO(2,NN) - (alpha3*deltN(NN))
     &                           *(flux(2,NN) - dis(2,NN))
            U(3,NN) = UO(3,NN) - (alpha3*deltN(NN))
     &                           *(flux(3,NN) - dis(3,NN))
            U(4,NN) = UO(4,NN) - (alpha3*deltN(NN))
     &                           *(flux(4,NN) - dis(4,NN))
            dU(1,NN) = U(1,NN) - dU(1,NN)
            dU(2,NN) = U(2,NN) - dU(2,NN)
            dU(3,NN) = U(3,NN) - dU(3,NN)
            dU(4,NN) = U(4,NN) - dU(4,NN)
         enddo

         call binlet
         call boutlet
         call tangent

c*  step four
         call calcflux

         do NN = 1, Nmax
            dU(1,NN) = U(1,NN)
            dU(2,NN) = U(2,NN)
            dU(3,NN) = U(3,NN)
            dU(4,NN) = U(4,NN)
            U(1,NN) = UO(1,NN) - (alpha4*deltN(NN))
     &                           *(flux(1,NN) - dis(1,NN))
            U(2,NN) = UO(2,NN) - (alpha4*deltN(NN))
```

```
     &                          *(flux(2,NN) - dis(2,NN))
           U(3,NN) = UO(3,NN) - (alpha4*deltN(NN))
     &                          *(flux(3,NN) - dis(3,NN))
           U(4,NN) = UO(4,NN) - (alpha4*deltN(NN))
     &                          *(flux(4,NN) - dis(4,NN))
           dU(1,NN) = U(1,NN) - dU(1,NN)
           dU(2,NN) = U(2,NN) - dU(2,NN)
           dU(3,NN) = U(3,NN) - dU(3,NN)
           dU(4,NN) = U(4,NN) - dU(4,NN)
        enddo

        call binlet
        call boutlet
        call tangent

c*  find root mean square difference in state vector
        durms = 0.0
        maxchange = 0.

        do i = 1, 4
           do NN = 1, Nmax
              durms = durms + (U(i,NN) - UO(i,NN))**2
              if (abs(U(i,NN)-UO(i,NN)).gt.abs(maxchange)) then
                 maxchange = U(i,NN) - UO(i,NN)
                 maxnode = NN
                 maxeqn = i
              endif
           enddo
        enddo

        durms = sqrt(durms/(4.*Nmax))

        return
        end




c*****************************************************************
c*                                                               *
c*      calculate time step for nodes                            *
c*                                                               *
c*****************************************************************

        subroutine timestep
        implicit none

        include 'TRI.INC'
        integer EF, FF, NN              !pointer
        integer CC, CC1, CC2
        integer N1, N2, N3, N4          !nodes related to face
        integer PN, P1, P2              !periodic nodes
        real delx, dely         !x and y length of side
        real delside                    !length of side
        real uu, vv, aa, rr, ww2        !values at node
        integer FF1, FF2, col           !color pointers
        integer EF1, EF2
        real onethird

c*  set values for false node 0
        U(1,0) = 1.
        U(2,0) = 0.
        U(3,0) = 0.
        U(4,0) = 0.

c*  zero out delt
        do NN = 1, Nmax
           deltN(NN) = 0.
        enddo

c*  find time step for each node
```

```
        FF2 = 0

        do col = 1, Fcolormax
         FF1 = FF2 + 1
         FF2 = FF1 - 1 + NFcolor(col)
         do FF = FF1, FF2
           N1 = face(FF,3)
           N2 = face(FF,4)
           N3 = face(FF,5)
           N4 = face(FF,6)

           delx = x(N2) - x(N1)
           dely = y(N2) - y(N1)
           delside = sqrt(delx**2 + dely**2)

           rr = U(1,N3)
           uu = U(2,N3)/rr
           vv = U(3,N3)/rr
           ww2 = uu**2 + vv**2
           aa = sqrt(gam*gam1*(U(4,N3)/rr - 0.5*ww2))

           deltN(N3) = deltN(N3) + abs(uu*dely-vv*delx) + aa*delside

           rr = U(1,N4)
           uu = U(2,N4)/rr
           vv = U(3,N4)/rr
           ww2 = uu**2 + vv**2
           aa = sqrt(gam*gam1*(U(4,N4)/rr - 0.5*ww2))

           deltN(N4) = deltN(N4) + abs(uu*dely-vv*delx) + aa*delside

         enddo
        enddo

c*  add to timestep for boundary nodes
        EF2 = 0

        do col = 1, E2colormax
         EF1 = EF2 + 1
         EF2 = EF1 - 1 + NEcolor(col)

         do EF = EF1, EF2
           FF = eface(EF)
           N1 = face(FF,3)
           N2 = face(FF,4)

           delx = x(N2) - x(N1)
           dely = y(N2) - y(N1)
           delside = sqrt(delx**2 + dely**2)

           rr = U(1,N2)
           uu = U(2,N2)/rr
           vv = U(3,N2)/rr
           ww2 = uu**2 + vv**2
           aa = sqrt(gam*gam1*(U(4,N2)/rr - 0.5*ww2))

           deltN(N2) = deltN(N2) + abs(uu*dely-vv*delx) + aa*delside

           rr = U(1,N1)
           uu = U(2,N1)/rr
           vv = U(3,N1)/rr
           ww2 = uu**2 + vv**2
           aa = sqrt(gam*gam1*(U(4,N1)/rr - 0.5*ww2))

           deltN(N1) = deltN(N1) + abs(uu*dely-vv*delx) + aa*delside

         enddo
        enddo

c**  find time step at periodic nodes
        do PN = 1, Pmax
           P1 = pnode(PN,1)
```

```
            P2 = pnode(PN,2)
            deltN(P1) = deltN(P1) + deltN(P2)
            deltN(P2) = deltN(P1)
         enddo

c**   delt is actually delt/areaN
         do NN = 1, Nmax
            deltN(NN) = CFL*2./deltN(NN)
         enddo

         return
         end




c*****************************************************************
c*                                                               *
c*      calculate flux vector values at nodes                    *
c*                                                               *
c*****************************************************************

         subroutine calcflux
         implicit none

         include 'TRI.INC'
         integer CC, CC1, CC2
         real dx23, dx31, dx12          !change in x on cell edges
         real dy23, dy31, dy12          !chagne in y on cell edges
         integer EF, NN, FF, i          !pointers
         real delx, dely          !change in x and y on face
         real WW                  !kinetic energy
         integer N1, N2, N3, N4         !nodes around edge
         real dflux                     !fluxes throuth face
         integer FF1, FF2, col          !color pointers

         do NN = 1, Nmax
c*    set flux to zero
            flux(1,NN) = 0.
            flux(2,NN) = 0.
            flux(3,NN) = 0.
            flux(4,NN) = 0.

c*    calculate f and g at nodes
            WW = 0.5*(U(2,NN)**2 + U(3,NN)**2)/U(1,NN)
            F(1,NN) = U(2,NN)
            F(2,NN) = U(2,NN)**2/U(1,NN) + gam1*(U(4,NN) - WW)
            F(3,NN) = U(2,NN)*U(3,NN)/U(1,NN)
            F(4,NN) = (U(2,NN)/U(1,NN))*(gam*U(4,NN) - gam1*WW)

            G(1,NN) = U(3,NN)
            G(2,NN) = U(2,NN)*U(3,NN)/U(1,NN)
            G(3,NN) = U(3,NN)**2/U(1,NN) + gam1*(U(4,NN) - WW)
            G(4,NN) = (U(3,NN)/U(1,NN))*(gam*U(4,NN) - gam1*WW)
         enddo

         do CC = 1, Cmax
            N1 = cell(CC,4)
            N2 = cell(CC,5)
            N3 = cell(CC,6)

            dy23 = y(N2) - y(N3)
            dy31 = y(N3) - y(N1)
            dy12 = y(N1) - y(N2)
            dx23 = x(N2) - x(N3)
            dx31 = x(N3) - x(N1)
            dx12 = x(N1) - x(N2)

            dUc(1,CC) = 0.5*(-F(1,N1)*dy23 + G(1,N1)*dx23
     &                       -F(1,N2)*dy31 + G(1,N2)*dx31
```

```
     &                        -F(1,N3)*dy12 + G(1,N3)*dx12)

           dUc(2,CC) = 0.5*(-F(2,N1)*dy23 + G(2,N1)*dx23
     &                        -F(2,N2)*dy31 + G(2,N2)*dx31
     &                        -F(2,N3)*dy12 + G(2,N3)*dx12)

           dUc(3,CC) = 0.5*(-F(3,N1)*dy23 + G(3,N1)*dx23
     &                        -F(3,N2)*dy31 + G(3,N2)*dx31
     &                        -F(3,N3)*dy12 + G(3,N3)*dx12)

           dUc(4,CC) = 0.5*(-F(4,N1)*dy23 + G(4,N1)*dx23
     &                        -F(4,N2)*dy31 + G(4,N2)*dx31
     &                        -F(4,N3)*dy12 + G(4,N3)*dx12)
         enddo

c*  implement wall boundary conditions
        call bwall

        CC2 = 0

        do col = 1, Ccolormax
         CC1 = CC2 + 1
         CC2 = CC1 - 1 + NCcolor(col)

CVD$     NODEPCHK
         do CC = CC1, CC2
           N1 = cell(CC,4)
           N2 = cell(CC,5)
           N3 = cell(CC,6)

           flux(1,N1) = flux(1,N1) - dUc(1,CC)
           flux(1,N2) = flux(1,N2) - dUc(1,CC)
           flux(1,N3) = flux(1,N3) - dUc(1,CC)

           flux(2,N1) = flux(2,N1) - dUc(2,CC)
           flux(2,N2) = flux(2,N2) - dUc(2,CC)
           flux(2,N3) = flux(2,N3) - dUc(2,CC)

           flux(3,N1) = flux(3,N1) - dUc(3,CC)
           flux(3,N2) = flux(3,N2) - dUc(3,CC)
           flux(3,N3) = flux(3,N3) - dUc(3,CC)

           flux(4,N1) = flux(4,N1) - dUc(4,CC)
           flux(4,N2) = flux(4,N2) - dUc(4,CC)
           flux(4,N3) = flux(4,N3) - dUc(4,CC)

         enddo
        enddo

        return
        end




c*********************************************************************
c*                                                                   *
c*      this subroutine applies the wall boundary condition          *
c*                                                                   *
c*********************************************************************

        subroutine bwall
        implicit none

        include 'TRI.INC'
        integer EF, FF, CC        !pointer
        integer N1, N3            !nodes on edge
        real coef
        real dy13, dx13           !change in x and y
        real p1, p3               !pressure
        integer EF1, EF2, col
```

```
c*  upper and lower boundary
        EF2 = 0

        do col = 1, Eicolormax
         EF1 = EF2 + 1
         EF2 = EF1 - 1 + NEcolor(col)

CVD$    NODEPCHK
         do EF = EF1, EF2
           FF = eface(EF)
           CC = face(FF,1)
           N1 = face(FF,4)
           N3 = face(FF,3)

           dy13 = y(N1) - y(N3)
           dx13 = x(N1) - x(N3)

           p1 = gam1*(U(4,N1) - 0.5*(U(2,N1)**2 + U(3,N1)**2)/
     &          U(1,N1))
           p3 = gam1*(U(4,N3) - 0.5*(U(2,N3)**2 + U(3,N3)**2)/
     &          U(1,N3))

           dUc(1,CC) = dUc(1,CC) + 0.5*(
     &                 (F(1,N3) + F(1,N1))*dy13 -
     &                 (G(1,N3) + G(1,N1))*dx13)

           dUc(2,CC) = dUc(2,CC) + 0.5*(
     &                 (F(2,N3) + F(2,N1) - p1 - p3)*dy13 -
     &                 (G(2,N3) + G(2,N1))*dx13)

           dUc(3,CC) = dUc(3,CC) + 0.5*(
     &                 (F(3,N3) + F(3,N1))*dy13 -
     &                 (G(3,N3) + G(3,N1) - p1 - p3)*dx13)

           dUc(4,CC) = dUc(4,CC) + 0.5*(
     &                 (F(4,N3) + F(4,N1))*dy13 -
     &                 (G(4,N3) + G(4,N1))*dx13)

         enddo
        enddo

        return
        end




c*****************************************************************
c*                                                               *
c*      calculate the dissipation at each of the nodes for       *
c*      the current values of the state vector                   *
c*                                                               *
c*****************************************************************

        subroutine dissipation
        implicit none

        include 'TRI.INC'
        integer N1, N2, N3              !nodes at end of face
        integer PN, P1, P2              !periodic nodes
        real change                     !change in dissipation
        integer NN, FF, i               !pointers
        real dx31, dx12, dx23           !change in x
        real dy31, dy12, dy23           !change in y
        real dxC1, dxC2, dxC3, dxC4     !change in x in cell
        real dyC1, dyC2, dyC3, dyC4     !change in y in cell
        real del2(4,Maxnodes)           !second order changes
        real pres                       !pressure at nodes
        real delp                       !change in pressure
        real eps1(Maxnodes)             !dissipation coeficients
```

```
      real coef, coef2                !combination of eps1
      integer FF1, FF2, col           !color pointers
      integer CC1, CC2, CC            !pointers
      integer EF, EF1, EF2

      do NN = 1, Nmax
         dis(1,NN)  = 0.
         dis(2,NN)  = 0.
         dis(3,NN)  = 0.
         dis(4,NN)  = 0.

         del2(1,NN) = 0.
         del2(2,NN) = 0.
         del2(3,NN) = 0.
         del2(4,NN) = 0.

         eps1(NN) = 0.
      enddo

c*  undivided Laplacian of pressure for eps1 coefficient
      FF2 = 0

      do col = 1, Fcolormax
       FF1 = FF2 + 1
       FF2 = FF1 - 1 + NFcolor(col)

CVD$    NODEPCHK
       do FF = FF1, FF2
         N1 = face(FF,3)
         N2 = face(FF,4)

         delp = gam1*(U(4,N1) - U(4,N2)
     &           + .5*(U(2,N2)**2 + U(3,N2)**2)/U(1,N2)
     &           - .5*(U(2,N1)**2 + U(3,N1)**2)/U(1,N1))
         eps1(N1) = eps1(N1) - eps1coef*delp
         eps1(N2) = eps1(N2) + eps1coef*delp

       enddo
      enddo

c**  find eps1 and del2 at periodic nodes
      do PN = 1, Pmax
         P1 = pnode(PN,1)
         P2 = pnode(PN,2)
         eps1(P1) = eps1(P1) + eps1(P2)
         eps1(P2) = eps1(P1)
      enddo

      do NN = 1, Nmax
         pres = gam1*(U(4,NN) - 0.5*(U(2,NN)**2 +
     &                 U(3,NN)**2)/U(1,NN))
         eps1(NN) = abs(eps1(NN)/pres)
      enddo

c** low-accuracy smoothing
      if (sigE .eq. 0.) then

      FF2 = 0

      do col = 1, Fcolormax
       FF1 = FF2 + 1
       FF2 = FF1 - 1 + NFcolor(col)

CVD$    NODEPCHK
       do FF = FF1, FF2
         N1 = face(FF,3)
         N2 = face(FF,4)

         del2(1,N1) = del2(1,N1) - U(1,N2) + U(1,N1)
         del2(1,N2) = del2(1,N2) - U(1,N1) + U(1,N2)

         del2(2,N1) = del2(2,N1) - U(2,N2) + U(2,N1)
```

```
            del2(2,N2) = del2(2,N2) - U(2,N1) + U(2,N2)

            del2(3,N1) = del2(3,N1) - U(3,N2) + U(3,N1)
            del2(3,N2) = del2(3,N2) - U(3,N1) + U(3,N2)

            del2(4,N1) = del2(4,N1) - U(4,N2) + U(4,N1)
            del2(4,N2) = del2(4,N2) - U(4,N1) + U(4,N2)
          enddo
         enddo

c**  high-accuracy smoothing
        else if (sigE .eq. 1.) then

        CC2 = 0

        do col = 1, Ccolormax
         CC1 = CC2 + 1
         CC2 = CC1 - 1 + NCcolor(col)

CVD$    NODEPCHK
         do CC = CC1, CC2
           N1 = cell(CC,4)
           N2 = cell(CC,5)
           N3 = cell(CC,6)

           dx31 = x(N3) - x(N1)
           dx12 = x(N1) - x(N2)
           dx23 = x(N2) - x(N3)
           dy31 = y(N3) - y(N1)
           dy12 = y(N1) - y(N2)
           dy23 = y(N2) - y(N3)

           coef = 0.5/abs(-dx12*dy31 + dy12*dx31)

           dxC1 = (U(1,N1)*dy23 + U(1,N2)*dy31 + U(1,N3)*dy12)
     &                 *coef
           dxC2 = (U(2,N1)*dy23 + U(2,N2)*dy31 + U(2,N3)*dy12)
     &                 *coef
           dxC3 = (U(3,N1)*dy23 + U(3,N2)*dy31 + U(3,N3)*dy12)
     &                 *coef
           dxC4 = (U(4,N1)*dy23 + U(4,N2)*dy31 + U(4,N3)*dy12)
     &                 *coef

           dyC1 = (U(1,N1)*dx23 + U(1,N2)*dx31 + U(1,N3)*dx12)
     &                 *coef
           dyC2 = (U(2,N1)*dx23 + U(2,N2)*dx31 + U(2,N3)*dx12)
     &                 *coef
           dyC3 = (U(3,N1)*dx23 + U(3,N2)*dx31 + U(3,N3)*dx12)
     &                 *coef
           dyC4 = (U(4,N1)*dx23 + U(4,N2)*dx31 + U(4,N3)*dx12)
     &                 *coef

           del2(1,N1) = del2(1,N1) + (dxC1*dy23 + dyC1*dx23)
           del2(1,N2) = del2(1,N2) + (dxC1*dy31 + dyC1*dx31)
           del2(1,N3) = del2(1,N3) + (dxC1*dy12 + dyC1*dx12)

           del2(2,N1) = del2(2,N1) + (dxC2*dy23 + dyC2*dx23)
           del2(2,N2) = del2(2,N2) + (dxC2*dy31 + dyC2*dx31)
           del2(2,N3) = del2(2,N3) + (dxC2*dy12 + dyC2*dx12)

           del2(3,N1) = del2(3,N1) + (dxC3*dy23 + dyC3*dx23)
           del2(3,N2) = del2(3,N2) + (dxC3*dy31 + dyC3*dx31)
           del2(3,N3) = del2(3,N3) + (dxC3*dy12 + dyC3*dx12)

           del2(4,N1) = del2(4,N1) + (dxC4*dy23 + dyC4*dx23)
           del2(4,N2) = del2(4,N2) + (dxC4*dy31 + dyC4*dx31)
           del2(4,N3) = del2(4,N3) + (dxC4*dy12 + dyC4*dx12)
         enddo
        enddo

c*  upper and lower boundary
       EF2 = 0
```

```
      do col = 1, Eicolormax
       EF1 = EF2 + 1
       EF2 = EF1 - 1 + NEcolor(col)

CVD$  NODEPCHK
       do EF = EF1, EF2
         FF = eface(EF)
         CC = face(FF,1)
         N1 = face(FF,3)
         N2 = face(FF,4)
         N3 = face(FF,5)

         dx31 = x(N3) - x(N1)
         dx12 = x(N1) - x(N2)
         dx23 = x(N2) - x(N3)
         dy31 = y(N3) - y(N1)
         dy12 = y(N1) - y(N2)
         dy23 = y(N2) - y(N3)

         coef = 0.5/abs(-dx12*dy31 + dy12*dx31)

         dxC1 = (U(1,N1)*dy23 + U(1,N2)*dy31 + U(1,N3)*dy12)
     &                *coef
         dxC2 = (U(2,N1)*dy23 + U(2,N2)*dy31 + U(2,N3)*dy12)
     &                *coef
         dxC3 = (U(3,N1)*dy23 + U(3,N2)*dy31 + U(3,N3)*dy12)
     &                *coef
         dxC4 = (U(4,N1)*dy23 + U(4,N2)*dy31 + U(4,N3)*dy12)
     &                *coef

         dyC1 = (U(1,N1)*dx23 + U(1,N2)*dx31 + U(1,N3)*dx12)
     &                *coef
         dyC2 = (U(2,N1)*dx23 + U(2,N2)*dx31 + U(2,N3)*dx12)
     &                *coef
         dyC3 = (U(3,N1)*dx23 + U(3,N2)*dx31 + U(3,N3)*dx12)
     &                *coef
         dyC4 = (U(4,N1)*dx23 + U(4,N2)*dx31 + U(4,N3)*dx12)
     &                *coef

         del2(1,N1) = del2(1,N1) + (dxC1*dy12 + dyC1*dx12)
         del2(1,N2) = del2(1,N2) + (dxC1*dy12 + dyC1*dx12)

         del2(2,N1) = del2(2,N1) + (dxC2*dy12 + dyC2*dx12)
         del2(2,N2) = del2(2,N2) + (dxC2*dy12 + dyC2*dx12)

         del2(3,N1) = del2(3,N1) + (dxC3*dy12 + dyC3*dx12)
         del2(3,N2) = del2(3,N2) + (dxC3*dy12 + dyC3*dx12)

         del2(4,N1) = del2(4,N1) + (dxC4*dy12 + dyC4*dx12)
         del2(4,N2) = del2(4,N2) + (dxC4*dy12 + dyC4*dx12)

       enddo
      enddo

      endif

CVD$  NODEPCHK
      do PN = 1, Pmax
         P1 = pnode(PN,1)
         P2 = pnode(PN,2)
         del2(1,P1) = del2(1,P1) + del2(1,P2)
         del2(2,P1) = del2(2,P1) + del2(2,P2)
         del2(3,P1) = del2(3,P1) + del2(3,P2)
         del2(4,P1) = del2(4,P1) + del2(4,P2)
         del2(1,P2) = del2(1,P1)
         del2(2,P2) = del2(2,P1)
         del2(3,P2) = del2(3,P1)
         del2(4,P2) = del2(4,P1)
      enddo

      FF2 = 0
```

```
        do col = 1, Fcolormax
         FF1 = FF2 + 1
         FF2 = FF1 - 1 + NFcolor(col)

CVD$    NCDEPCHK
         do FF = FF1, FF2
           N1 = face(FF,3)
           N2 = face(FF,4)

           coef = 0.5*(eps1(N1) + eps1(N2))
           coef2 = 0.5*max(0.,eps2-coef)*(1./deltN(N1) + 1./deltN(N2))

           change = coef2*(del2(1,N2) - del2(1,N1)) +
     &              coef*(U(1,N2) - U(1,N1))
           dis(1,N1) = dis(1,N1) + change
           dis(1,N2) = dis(1,N2) - change

           change = coef2*(del2(2,N2) - del2(2,N1)) +
     &              coef*(U(2,N2) - U(2,N1))
           dis(2,N1) = dis(2,N1) + change
           dis(2,N2) = dis(2,N2) - change

           change = coef2*(del2(3,N2) - del2(3,N1)) +
     &              coef*(U(3,N2) - U(3,N1))
           dis(3,N1) = dis(3,N1) + change
           dis(3,N2) = dis(3,N2) - change

           change = coef2*(del2(4,N2) - del2(4,N1)) +
     &              coef*(U(4,N2) - U(4,N1))
           dis(4,N1) = dis(4,N1) + change
           dis(4,N2) = dis(4,N2) - change
         enddo
        enddo

c**  find eps1 and del2 at periodic nodes
        do PN = 1, Pmax
           P1 = pnode(PN,1)
           P2 = pnode(PN,2)
           dis(1,P1) = dis(1,P1) + dis(1,P2)
           dis(2,P1) = dis(2,P1) + dis(2,P2)
           dis(3,P1) = dis(3,P1) + dis(3,P2)
           dis(4,P1) = dis(4,P1) + dis(4,P2)
           dis(1,P2) = dis(1,P1)
           dis(2,P2) = dis(2,P1)
           dis(3,P2) = dis(3,P1)
           dis(4,P2) = dis(4,P1)
        enddo

        return
        end




c*********************************************************************
c*                                                                   *
c*      adjust inlet state vector for boundary condition             *
c*                                                                   *
c*********************************************************************

        subroutine binlet
        implicit none

        include 'TRI.INC'
        real rrinl, uuinl, vvinl, ppinl !average values at inlet
        real vv2inl, aainl
        real r0inf, a0inf               !stag. density and speed of sound
        real H0pres                     !stagnation enthalpy and pressure
        real spres                      !entropy
        real coef
```

```
      real aaa1, aaa2, aaa3, aaa4     !coef. of inverse of matrix
      real bbb1, bbb2, bbb3, bbb4
      real ccc1, ccc2, ccc3, ccc4
      real ddd1, ddd2, ddd3, ddd4
      real ww2, uu, vv, pp, rr, aa    !values at next interior node
      real dH0, ds, dtan, dw4
      real drr, duu, dvv, dpp
      integer IN, NN                  !boundary face and its nodes

c**   average values at inlet
      rrinl = U(1,innode(Imax/2)) - dU(1,innode(Imax/2))
      uuinl = (U(2,innode(Imax/2)) - dU(2,innode(Imax/2)))/rrinl
      vvinl = (U(3,innode(Imax/2)) - dU(3,innode(Imax/2)))/rrinl
      ww2inl = uuinl**2 + vvinl**2
      ppinl = gam1*((U(4,innode(Imax/2)) - dU(4,innode(Imax/2)))
     &              - 0.5*rrinl*ww2inl)
      aainl = sqrt(gam*ppinl/rrinl)

c*    subsonic inlet nodes
      if ((sqrt(ww2inl)/aainl).lt.1. .or. pitch.ne.0.) then
c**   prescribed values
        H0pres = 1./gam1
        spres = 0.

c**   coefficient of matrix
        coef = 1./(uuinl*(aainl+uuinl) + vvinl**2)
        aaa1 = (rrinl*uuinl/aainl)*coef
        aaa2 = -(ppinl/aainl)*(uuinl*gam/gam1 + ww2inl/aainl)*coef
        aaa3 = -(rrinl*vvinl/aainl)*coef
        aaa4 = (ww2inl/aainl**2)*coef
        bbb1 = uuinl*coef
        bbb2 = -(uuinl*ppinl/(gam1*rrinl))*coef
        bbb3 = -vvinl*coef
        bbb4 = -(uuinl/rrinl)*coef
        ccc1 = vvinl*coef
        ccc2 = -(vvinl*ppinl/(gam1*rrinl))*coef
        ccc3 = (aainl + uuinl)*coef
        ccc4 = -(vvinl/rrinl)*coef
        ddd1 = rrinl*aainl*uuinl*coef
        ddd2 = -(ppinl*aainl*uuinl/gam1)*coef
        ddd3 = -rrinl*aainl*vvinl*coef
        ddd4 = ww2inl*coef

CVD$    NODEPCHK
        do IN = 1, Imax
          NN = innode(IN)
          rr = U(1,NN) - dU(1,NN)
          uu = (U(2,NN) - dU(2,NN))/rr
          vv = (U(3,NN) - dU(3,NN))/rr
          ww2 = uu**2 + vv**2
          pp = gam1*((U(4,NN) - dU(4,NN)) - 0.5*rr*ww2)
          aa = sqrt(gam*pp/rr)
          dH0  = H0pres - ((gam/gam1)*pp/rr + 0.5*ww2)
          ds   = spres - (log(gam*pp) - gam*log(rr))
          dtan = (Sinl - vv/uu)*uuinl**2
          dw4  =   dU(1,NN)*(aa*uu + gam1*ww2)
     &                - dU(2,NN)*(aa + gam1*uu)
     &                - dU(3,NN)*gam1*vv
     &                + dU(4,NN)*gam1
          drr = aaa1*dH0 + aaa2*ds + aaa3*dtan + aaa4*dw4
          duu = bbb1*dH0 + bbb2*ds + bbb3*dtan + bbb4*dw4
          dvv = ccc1*dH0 + ccc2*ds + ccc3*dtan + ccc4*dw4
          dpp = ddd1*dH0 + ddd2*ds + ddd3*dtan + ddd4*dw4

          U(1,NN) = U(1,NN) - dU(1,NN) + drr
          U(2,NN) = U(2,NN) - dU(2,NN) + rr*duu + uu*drr
          U(3,NN) = U(3,NN) - dU(3,NN) + rr*dvv + vv*drr
          U(4,NN) = U(4,NN) - dU(4,NN) + dpp/gam1 + 0.5*ww2*drr
     &              + rr*(uu*duu + vv*dvv)
        enddo

c*    supersonic inlet nodes
```

```
        else if ((sqrt(ww2inl)/aainl).ge.1.) then
c*  state vector components far from body
           rOinf = 1.
           aOinf = 1.

CVD$    NODEPCHK
           do IN = 1, Imax
              NN = innode(IN)
              rr = rOinf*(1.0+0.5*gam1*Minl**2)**(-1./gam1)
              aa = aOinf*(1.0+0.5*gam1*Minl**2)**(-0.5)
              uu = Minl*aa
              pp = rr*aa**2/gam

              U(1,NN) = rr
              U(2,NN) = rr*uu
              U(3,NN) = 0.
              U(4,NN) = pp/gam1 + .5*(uu**2)*rr
           enddo
        endif

        return
        end



c*****************************************************************
c*                                                               *
c*      adjust outlet state vector for boundary condition        *
c*                                                               *
c*****************************************************************

        subroutine boutlet
        implicit none

        include 'TRI.INC'
        real rrout, uuout, vvout, ppout !average values at outet
        real aaout, ww2out
        real aaa1, aaa4                 !coef. of inverse of matrix
        real bbb3, bbb4
        real ccc2
        real ddd4
        real ww2, uu, vv, pp, rr, aa    !values at next interior node
        real dw1, dw2, dw3, dp
        real drr, duu, dvv, dpp
        integer ON, NN                  !boundary face and its nodes

        rrout = U(1,outnode(Omax/2)) - dU(1,outnode(Omax/2))
        uuout = (U(2,outnode(Omax/2)) - dU(2,outnode(Omax/2)))/rrout
        vvout = (U(3,outnode(Omax/2)) - dU(3,outnode(Omax/2)))/rrout
        ww2out = uuout**2 + vvout**2
        ppout = gam1*((U(4,outnode(Omax/2)) - dU(4,outnode(Omax/2)))
     &               - 0.5*rrout*ww2out)
        aaout = sqrt(gam*ppout/rrout)

        aaa1 = -1./aaout**2
        aaa4 = 1./aaout**2
        bbb3 = .1/(rrout*aaout)
        bbb4 = -1./(rrout*aaout)
        ccc2 = .1/(rrout*aaout)
        ddd4 = 1.

c*  set boundary values of state vector for nodes
CVD$    NODEPCHK
        do ON = 1, Omax
           NN = outnode(ON)
           rr = U(1,NN) - dU(1,NN)
           uu = (U(2,NN) - dU(2,NN))/rr
           vv = (U(3,NN) - dU(3,NN))/rr
           ww2 = uu**2 + vv**2
           pp = gam1*((U(4,NN) - dU(4,NN)) - 0.5*rr*ww2)
           aa = sqrt(gam*pp/rr)
```

```
         if (sqrt(ww2) .lt. aa) then
            dw1 =   dU(1,NN)*(0.5*ww2*gam1 - aa**2)
     &            - dU(2,NN)*gam1*uu
     &            - dU(3,NN)*gam1*vv
     &            + dU(4,NN)*gam1
            dw2 = - dU(1,NN)*aa*vv
     &            + dU(3,NN)*aa
            dw3 =   dU(1,NN)*(0.5*ww2*gam1 - aa*uu)
     &            - dU(2,NN)*(gam1*uu - aa)
     &            - dU(3,NN)*gam1*vv
     &            + dU(4,NN)*gam1
            dp = pout - pp
            drr = aaa1*dw1 + aaa4*dp
            duu = bbb3*dw3 + bbb4*dp
            dvv = ccc2*dw2
            dpp = ddd4*dp
            U(1,NN) = U(1,NN) - dU(1,NN) + drr
            U(2,NN) = U(2,NN) - dU(2,NN) + rr*duu + uu*drr
            U(3,NN) = U(3,NN) - dU(3,NN) + rr*dvv + vv*drr
            U(4,NN) = U(4,NN) - dU(4,NN) + dpp/gam1 + 0.5*ww2*drr
     &                + rr*(uu*duu + vv*dvv)
         endif
      enddo

      return
      end
```

```
c*****************************************************************
c*                                                               *
c*      this subroutine changes the momentum change to make      *
c*      flow tangent to wall                                     *
c*                                                               *
c*****************************************************************

      subroutine tangent
      implicit none

      include 'TRI.INC'
      integer EN                        !pointer
      integer NN                        !node on wall
      real drwn                         !change in momentum normal to wall

      do EN = 1, bnode
         NN = enode(EN)
         drwn = -U(2,NN)*senode(EN) + U(3,NN)*cenode(EN)
         U(2,NN) = U(2,NN) + drwn*senode(EN)
         U(3,NN) = U(3,NN) - drwn*cenode(EN)
      enddo

      return
      end
```

A.1.5 Plotting Package

```
      program plotgen
      implicit none

      include 'TRI.INC'

      integer NN, EN, CC                !pointers
      integer N1, N2, N3
```

```
      integer ptype                    !type of plot
      integer ctype                    !type of contour
      integer stype                    !type of surface distributions
      character*80 TITLE, ITITLE       !title for plots
      integer NTITL                    !number of leters in title
      external g2pltg, g2pltc          !plotting subroutines
      integer indgr
      integer a4, a5, a6, a7, a8, a9, a10!dummy variables
      real uu, vv, pp, rr, mm, mm2     !from state vectors
      real pt, ptinf                   !total pressure
      real zz, z(Maxnodes)             !contour values
      real zmax, zmin                  !max and min of z array
      integer NCONT                    !contour level info
      real CBASE, CSTEP
      real Cinc                        !contour increment
      character*6 NUM                  !same as CSTEP but character
      integer NLINE, IOPT(2)           !indicators for line plots
      integer points, npts(2)          !number of points on line
      real xline(Maxedges)             !points on line to plot
      real yline(Maxedges)

c*  read data from file
      call gridio(1)
      call flowio(1)

c*  initialize GRAFIC
      write(ITITLE,1) Minl
 1    format('INLET MACH NUMBER = ',F5.3)
      call grinit(5,6,ITITLE)

      do while (1)

c*  prompt user for type of plot
         type*, 'Type of plot '
         type*, '   0) STOP   '
         type*, '   1) grid   '
         type*, '   2) contour'
         type*, '   3) surface distribution'
         type*, '   4) data'
         type 11
 11      format($,' selection = ')
         accept 111, ptype
 111     format(I)

         if (ptype.eq.0) then
            stop
         else if (ptype.eq.1) then

            TITLE = '   X       Y    COMPUTATIONAL GRID'
            indgr = 23
            call gr_control(g2pltg, indgr, TITLE, x, y, Nmax, a4,
     &                 a5, a6, a7, a8, a9, a10)

         else if (ptype.eq.2) then
c*  choose type of contour plot
            do while (1)

            Cinc = 0.

 2          type*, 'Type of contour'
            type*, '   0) TOP LEVEL'
            type*, '   1) density'
            type*, '   2) Mach number'
            type*, '   3) normal velocity'
            type*, '   4) pressure'
            type*, '   5) total pressure loss'
            type*, '   6) speed of sound'
            type*, '   7) entropy'
            type*, '   8) stagnation enthalpy'
            type*, '   9) vol'
            type*, '  10) CONTOUR INCREMENT'
            type 22
```

```
 22          format($,' selection = ')
             accept 222, ctype
 222         format(I)

c*  exit from contour loop
             if (ctype.eq.0) goto 999

             if (ctype.eq.10) then
                type 21
 21             format($,' CONTOUR INCREMENT = ')
                accept 221, Cinc
 221            format(F)
                type*, ' '
                goto 2
             endif

c*  set up nodal contour values
             if (ctype.eq.5) then
                ptinf = 1./gam
             endif

             zmin = 1.e20
             zmax = -1.e20

             do NN = 1, Nmax
                rr = U(1,NN)
                uu = U(2,NN)/rr
                vv = U(3,NN)/rr

                if (ctype.eq.1) then
                   zz = rr
                else if (ctype.eq.2) then
                   pp = gam1*(U(4,NN) - 0.5*rr*(uu**2 + vv**2))
                   mm = sqrt(rr*(uu**2 + vv**2)/(gam*pp))
                   zz = mm
                else if (ctype.eq.3) then
                   zz = sqrt(uu**2 + vv**2)
                else if (ctype.eq.4) then
                   pp = gam1*(U(4,NN) - 0.5*rr*(uu**2 + vv**2))
                   zz = pp
                else if (ctype.eq.5) then
                   pp = gam1*(U(4,NN) - 0.5*rr*(uu**2 + vv**2))
                   mm2 = rr*(uu**2 + vv**2)/(gam*pp)
                   pt = pp*(1.+.5*gam1*mm2)**(gam/gam1)
                   zz = (1. - pt/ptinf)*100.
                else if (ctype.eq.6) then
                   pp = gam1*(U(4,NN) - 0.5*rr*(uu**2 + vv**2))
                   zz = sqrt(gam*pp/rr)
                else if (ctype.eq.7) then
                   pp = gam1*(U(4,NN) - 0.5*rr*(uu**2 + vv**2))
                   zz = (log(gam*pp) - gam*log(rr))*100.
                else if (ctype.eq.8) then
                   pp = gam1*(U(4,NN) - 0.5*rr*(uu**2 + vv**2))
                   zz = ((gam/gam1)*(pp/rr) + 0.5*(uu**2 + vv**2))*100.
                endif

                zmax = max(zmax,zz)
                zmin = min(zmin,zz)
                z(NN) = zz
             enddo

             if (ctype.eq.9) then
                open(unit=50, status='unknown', form='unformatted')
                read(50) Cmax, (vol(CC),CC=1,Cmax)
                close(unit=50)
                zmin = 1.e20
                zmax = -1.e20
                do NN = 1, Nmax
                   z(NN) = 0.
                enddo
                do CC = 1, Cmax
                   N1 = cell(CC,4)
```

```
                  N2 = cell(CC,5)
                  N3 = cell(CC,6)
                  zz = vol(CC)
                  zmax = max(zmax,zz)
                  zmin = min(zmin,zz)
                  z(N1) = z(N1) + zz/3.
                  z(N2) = z(N2) + zz/3.
                  z(N3) = z(N3) + zz/3.
               enddo
            endif

c*  set title and extra variables needed for GRAFIC
            if (ctype.eq.1) then
               TITLE = '   X       Y    DENSITY '
               NTITL = 24
               NCONT = 20
            else if (ctype.eq.2) then
               TITLE = '   X       Y    MACH NUMBER '
               NTITL = 28
               NCONT = 20
            else if (ctype.eq.3) then
               TITLE = '   X       Y    NORMAL VELOCITY '
               NTITL = 32
               NCONT = 20
            else if (ctype.eq.4) then
               TITLE = '   X       Y    PRESSURE '
               NTITL = 25
               NCONT = 20
            else if (ctype.eq.5) then
               TITLE = '   X       Y    % TOTAL PRESSURE LOSS '
               NTITL = 38
               NCONT = 20
            else if (ctype.eq.6) then
               TITLE = '   X       Y    SPEED OF SOUND '
               NTITL = 31
               NCONT = 20
            else if (ctype.eq.7) then
               TITLE = '   X       Y    % ENTROPY '
               NTITL = 24
               NCONT = 20
            else if (ctype.eq.8) then
               TITLE = '   X       Y    % STAGNATION ENTHALPY '
               NTITL = 38
               NCONT = 20
            else if (ctype.eq.9) then
               TITLE = '   X       Y    VOL '
               NTITL = 20
               NCONT = 20
            endif

c*  find contour levels
            if (Cinc.ne.0.) then
               NCONT = int((zmax-zmin)/Cinc + 2.)
               CSTEP = Cinc
               CBASE = (real(int((zmin/Cinc)-1.)))*Cinc
            else
               call GR_SCALE(zmin, zmax, NCONT-1,  CBASE, CSTEP)
            endif


            z(Nmax+1) = NCONT
            z(Nmax+2) = CBASE + 0.01*CSTEP
            z(Nmax+3) = CSTEP

c*  finish up title
            TITLE(NTITL+1:NTITL+24) = 'CONTOURS WITH INCREMENT '
            write(NUM,10) CSTEP
 10         format(F6.4)
            TITLE(NTITL+25:80) = NUM

c*  plot the contour lines
            indgr = 23
```

```
            call gr_control(g2pltc, indgr, TITLE, z, Nmax, x, y,
     &                  a5, a6, a7, a8, a9, a10)

            enddo

c*  plot surface distribution
          else if (ptype.eq.3) then
            do while (1)

            type*, 'Type of surface distribution'
            type*, '   0) TOP LEVEL'
            type*, '   1) density'
            type*, '   2) Mach number'
            type*, '   3) normal velocity'
            type*, '   4) surface flow angle'
            type*, '   5) analytical surface flow angle'
            type*, '   6) pressure'
            type*, '   7) entropy'
            type*, '   8) stagnation enthalpy'
            type*, '   9) total pressure loss'
            type 33
 33         format($,' selection = ')
            accept 333, stype
 333        format(I)

c*  exit from contour loop
            if (stype.eq.0) goto 999

c*  set up array with points on line for upper surface
            if (stype.eq.9) then
               ptinf = 1./gam
            endif

            points = 0
            do EN = 1, tnode
               points = points + 1
               NN = enode(EN)
               rr = U(1,NN)
               uu = U(2,NN)/rr
               vv = U(3,NN)/rr

               if (stype.eq.1) then
                  zz = rr
               else if (stype.eq.2) then
                  pp = gam1*(U(4,NN) - 0.5*rr*(uu**2 + vv**2))
                  mn = sqrt(rr*(uu**2 + vv**2)/(gam*pp))
                  zz = mn
               else if (stype.eq.3) then
                  zz = sqrt(uu**2 + vv**2)
               else if (stype.eq.4) then
                  zz = atan(vv/uu)
               else if (stype.eq.5) then
                  zz = atan(senode(EN)/cenode(EN))
               else if (stype.eq.6) then
                  pp = gam1*(U(4,NN) - 0.5*rr*(uu**2 + vv**2))
                  zz = pp
               else if (stype.eq.7) then
                  pp = gam1*(U(4,NN) - 0.5*rr*(uu**2 + vv**2))
                  zz = (log(gam*pp) - gam*log(rr))*100.
               else if (stype.eq.8) then
                  pp = gam1*(U(4,NN) - 0.5*rr*(uu**2 + vv**2))
                  zz = ((gam/gam1)*(pp/rr) + 0.5*(uu**2 + vv**2))*100.
               else if (stype.eq.9) then
                  pp = gam1*(U(4,NN) - 0.5*rr*(uu**2 + vv**2))
                  mn2 = rr*(uu**2 + vv**2)/(gam*pp)
                  pt = pp*(1.+.5*gam1*mn2)**(gam/gam1)
                  zz = (1. - pt/ptinf)*100.
               endif

               xline(points) = x(NN)
               yline(points) = zz
            enddo
```

```
            npts(1) = points

c*  set up array with points on line for lower surface
            points= 0
            do EN = tnode+1, bnode
               points = points + 1
               NN = enode(EN)
               rr = U(1,NN)
               uu = U(2,NN)/rr
               vv = U(3,NN)/rr

               if (stype.eq.1) then
                  zz = rr
               else if (stype.eq.2) then
                  pp = gami*(U(4,NN) - 0.5*rr*(uu**2 + vv**2))
                  mm = sqrt(rr*(uu**2 + vv**2)/(gam*pp))
                  zz = mm
               else if (stype.eq.3) then
                  zz = sqrt(uu**2 + vv**2)
               else if (stype.eq.4) then
                  zz = atan(vv/uu)
               else if (stype.eq.5) then
                  zz = atan(senode(EN)/cenode(EN))
               else if (stype.eq.6) then
                  pp = gami*(U(4,NN) - 0.5*rr*(uu**2 + vv**2))
                  zz = pp
               else if (stype.eq.7) then
                  pp = gami*(U(4,NN) - 0.5*rr*(uu**2 + vv**2))
                  zz = (log(gam*pp) - gam*log(rr))*100.
               else if (stype.eq.8) then
                  pp = gami*(U(4,NN) - 0.5*rr*(uu**2 + vv**2))
                  zz = ((gam/gami)*(pp/rr) + 0.5*(uu**2 + vv**2))*100.
               else if (stype.eq.9) then
                  pp = gami*(U(4,NN) - 0.5*rr*(uu**2 + vv**2))
                  mm2 = rr*(uu**2 + vv**2)/(gam*pp)
                  pt = pp*(1.+.5*gami*mm2)**(gam/gami)
                  zz = (1. - pt/ptinf)*100.
               endif

               xline(npts(1)+points) = x(NN)
               yline(npts(1)+points) = zz
            enddo

            npts(2) = points

c** set plot indicators
            NLINE = 2
            IOPT(1) = 14
            IOPT(2) = 14

c**  set title for plot
            if (stype.eq.1) then
               TITLE = '   X    DENSITY DENSITY'
            else if (stype.eq.2) then
               TITLE = '   X    MACH NO.MACH NUMBER'
            else if (stype.eq.3) then
               TITLE = '   X    VELOCITYNORMAL VELOCITY'
            else if (stype.eq.4) then
               TITLE = '   X     ANGLE  SURFACE FLOW ANGLE'
            else if (stype.eq.5) then
               TITLE = '   X     ANGLE  ANALYTICAL SURFACE FLOW ANGLE'
            else if (stype.eq.6) then
               TITLE = '   X    PRESSUREPRESSURE'
            else if (stype.eq.7) then
               TITLE = '   X    ENTROPY"% ENTROPY'
            else if (stype.eq.8) then
               TITLE =
     &           '   X    "STAGNATION ENTHALPY"% STAGNATION ENTHALPY'
            else if (stype.eq.9) then
               TITLE =
     &           '   X    "TOTAL PRESSURE LOSS"% TOTAL PRESSURE LOSS'
```

```
            endif

c*  plot the line
            indgr = 21
            call gr_line(IOPT,NLINE,TITLE,INDGR,xline,yline,npts)

            enddo

         else if (ptype.eq.4) then
c*  calculate mean of total pressure loss
            ptinf = 1./gam
            zz = 0.
            zmax = 0.
            do NN = 1, Nmax
               rr = U(1,NN)
               uu = U(2,NN)/rr
               vv = U(3,NN)/rr
               pp = gam1*(U(4,NN) - 0.5*rr*(uu**2 + vv**2))
               mm2 = rr*(uu**2 + vv**2)/(gam*pp)
               pt = pp*(1.+.5*gam1*mm2)**(gam/gam1)
               zz = zz + (1. - pt/ptinf)**2
               zmax = max(zmax,abs(1.-pt/ptinf))
            enddo
            zz = sqrt(zz/real(Nmax))

            type*, ' '
            type*, 'inlet Mach number     = ', Minl
            type*, 'rms total pres. loss  = ', zz, ' (',log10(zz),')'
            type*, 'max total pres. loss  = ', zmax, ' (',log10(zmax),')'
            type*, 'cells                 = ', Cmax
            type*, 'faces                 = ', Fmax
            type*, 'nodes                 = ', Nmax
            type*, 'inlet nodes           = ', Imax
            type*, 'outlet nodes          = ', Omax
            type*, 'periodic nodes        = ', Pmax
            type*, 'edge nodes            = ', Emax
            type*, 'face colors           = ', Fcolormax
            type*, 'cell colors           = ', Ccolormax
            type*, 'edge colors           = ', E2colormax
            type*, ' '
 999      endif
      enddo

      end



      subroutine g2pltg(ifun, indgr, TITLE, alimits, info_string,
     &                  a1, a2, a3, a4, a5, a6, a7, a8, a9, a10)
      implicit none

      include 'TRI.INC'

      integer FF, EF, NN
      integer indgr
      character*80 TITLE
      integer ifun
      real alimits(4)
      integer a1, a2, a3, a4, a5, a6, a7, a8, a9, a10
      character*80 info_string
      real x1, x2, y1, y2

      if (ifun.eq.0) then
         return
      elseif (ifun.eq.1) then
         call gr_get_limits(x, y, Nmax, alimits)
         if (pitch.ne.0.) then
            alimits(4) = alimits(4) + pitch
         endif
      elseif (ifun.eq.2) then
         info_string = ' '
      elseif (ifun.eq.3) then
```

```
            do FF = 1, Fmax
               NN = face(FF,3)
               x1 = x(NN)
               y1 = y(NN)
               NN = face(FF,4)
               x2 = x(NN)
               y2 = y(NN)
               call gr_move(x1, y1, 0)
               call gr_draw(x2, y2, 0)
               if (pitch.ne.0.) then
                  y1 = y1 + pitch
                  y2 = y2 + pitch
                  call gr_move(x1, y1, 0)
                  call gr_draw(x2, y2, 0)
               endif
            enddo
         endif

         return
         end




         subroutine g2pltc(ifun, indgr, TITLE, alimits, info_string,
     &                     z, a2, a3, a4, a5, a6, a7, a8, a9, a10)
         implicit none

         include 'TRI.INC'

         integer FF, EF, NN, CC, NC       !pointers
         integer node                     !pointer
         integer indgr
         character*80 TITLE
         integer ifun
         real alimits(4)
         real z(Maxnodes)
         integer a2, a3, a4, a5, a6, a7, a8, a9, a10
         character*80 info_string
         real y1, y2, y3                  !location of cell nodes
         integer ncont                    !determine contour levels
         real cbase, cstep
         real zn12, zn32, zn31            !contour crossing on sides
         integer N1, N2, N3               !nodes at end of edge face
         real xn(3), yn(3), zn(3)         !x, y, and contour value at nodes
         real xpoint, ypoint              !x, y of pointer
         real atriangle                   !area of triangle
         real value                       !value of contour at pointer
         real cn(2,3)                     !x, y at cell nodes
         integer IIN                      !is pointer in cell?
         real aa1, aa2, aa3, aa           !areas
         real zcont, cont(80)             !contour levels
         real zmax, zmin                  !max & min contour values in cell

         if (ifun.eq.0) then
            return
         elseif (ifun.eq.1) then
            call gr_get_limits(x, y, Nmax, alimits)
            if (pitch.ne.0.) then
               alimits(4) = alimits(4) + pitch
            endif

         elseif (ifun.eq.2) then
c*  find contour value at point
            xpoint = alimits(1)
            ypoint = alimits(2)

c*  look for cell (x,y) is in
```

```
            do CC = 1, Cmax
               N1 = cell(CC,4)
               N2 = cell(CC,5)
               N3 = cell(CC,6)
               cn(1,1) = x(N1)
               cn(1,2) = x(N2)
               cn(1,3) = x(N3)
               cn(2,1) = y(N1)
               cn(2,2) = y(N2)
               cn(2,3) = y(N3)
               call gr_inside(IIN,cn,3,xpoint,ypoint)
               if (IIN.eq.1) then

c*  linear interpolation for value
                  aa2 = atriangle(x(N1),y(N1),x(N3),y(N3),xpoint,ypoint)
                  aa3 = atriangle(x(N2),y(N2),x(N1),y(N1),xpoint,ypoint)
                  aa1 = atriangle(x(N3),y(N3),x(N2),y(N2),xpoint,ypoint)
                  aa  = atriangle(x(N3),y(N3),x(N2),y(N2),x(N1),y(N1))
                  if (aa.eq.0) then
                     aa1 = 1.
                     aa2 = 1.
                     aa3 = 1.
                     aa  = 3.
                  endif
                  value = (aa1*z(N1) + aa2*z(N2) + aa3*z(N3))/aa
                  goto 30
               endif
               cn(2,1) = cn(2,1) + pitch
               cn(2,2) = cn(2,2) + pitch
               cn(2,3) = cn(2,3) + pitch
               call gr_inside(IIN,cn,3,xpoint,ypoint)
               if (IIN.eq.1) then

c*  linear interpolation for value
                  y1 = y(N1) + pitch
                  y2 = y(N2) + pitch
                  y3 = y(N3) + pitch
                  aa2 = atriangle(x(N1),y1,x(N3),y3,xpoint,ypoint)
                  aa3 = atriangle(x(N2),y2,x(N1),y1,xpoint,ypoint)
                  aa1 = atriangle(x(N3),y3,x(N2),y2,xpoint,ypoint)
                  aa  = atriangle(x(N3),y3,x(N2),y2,x(N1),y1)
                  if (aa.eq.0) then
                     aa1 = 1.
                     aa2 = 1.
                     aa3 = 1.
                     aa  = 3.
                  endif
                  value = (aa1*z(N1) + aa2*z(N2) + aa3*z(N3))/aa
                  goto 30
               endif
            enddo
 30         write (INFO_STRING,31) value
 31         format(' function value = ',f15.6)

         elseif (ifun.eq.3) then

c*  plot boundary
            do EF = 1, Emax
               FF = eface(EF)
               N1 = face(FF,3)
               N2 = face(FF,4)
               call gr_move(x(N1), y(N1), 0)
               call gr_draw(x(N2), y(N2), 0)
            enddo

            if (pitch.ne.0.) then
               do EF = 1, Emax
                  FF = eface(EF)
                  N1 = face(FF,3)
                  N2 = face(FF,4)
                  y1 = y(N1) + pitch
                  y2 = y(N2) + pitch
```

```
               call gr_move(x(N1), y1, 0)
               call gr_draw(x(N2), y2, 0)
            enddo
         endif

c*  set up contour levels
         ncont = z(Nmax+1)
         cbase = z(Nmax+2)
         cstep = z(Nmax+3)
         do NC = 1, ncont
            cont(NC) = cbase + (NC-1)*cstep
         enddo

c*  loop through cells
         do CC = 1, Cmax

            do NN = 1, 3
               node = cell(CC,NN+3)
               xn(NN) = x(node)
               yn(NN) = y(node)
               zn(NN) = z(node)
            enddo
            zmax = max(zn(1), zn(2), zn(3))
            zmin = min(zn(1), zn(2), zn(3))

c*  find contour crossing on triangle for each contour level
            do NC = 1, ncont
               zcont = cont(NC)
               if (zcont.lt.zmin .or. zcont.gt.zmax) then
c*                no need to compute
               else
c*                find contour crossing in triangle
                  call gr_cross(zn(1),zn(2),zcont,zn12)
                  call gr_cross(zn(3),zn(2),zcont,zn32)
                  call gr_cross(zn(3),zn(1),zcont,zn31)

c*  plot contour levels in triangle
                  call gr_ctriangle(xn(1), yn(1),
     &                              xn(2), yn(2),
     &                              xn(3), yn(3),
     &                              zn31, zn32, zn12)
                  if (pitch.ne.0.) then
                     y1 = yn(1) + pitch
                     y2 = yn(2) + pitch
                     y3 = yn(3) + pitch
                     call gr_ctriangle(xn(1), y1,
     &                                 xn(2), y2,
     &                                 xn(3), y3,
     &                                 zn31, zn32, zn12)
                  endif
               endif
            enddo
         enddo

      endif

      return
      end



c*********************************************************************
c*                                                                   *
c*      function to find the area of a triangle                      *
c*                                                                   *
c*********************************************************************

      function atriangle(x1,y1,x2,y2,x3,y3)

      real x1,y1,x2,y2,x3,y3
      real*8 distance,a,b,c,s
```

```
distance(xa,ya,xb,yb) = dsqrt(dble(xa-xb)**2 + dble(ya-yb)**2)

a = distance(x1,y1,x2,y2)
b = distance(x3,y3,x2,y2)
c = distance(x1,y1,x3,y3)

s = .5*(a + b + c)
a = dsqrt(s*(s-a)*(s-b)*(s-c))
atriangle = a
return
end
```

A.2 Quadrilateral Schemes

A.2.1 Common Files

This is file QUAD.INC which includes many declarations and common block statements and is included in all subroutines for the quadrilateral schemes.

```
parameter Maxnodes = 5000
parameter Maxfaces = 10000
parameter Maxcells = 5000
parameter Maxedges = 400

integer Nmax                    !number of nodes
integer Fmax                    !number of faces (including edges)
integer Cmax                    !number of cells (including edge cells)
integer Imax, Omax              !number of inlet and outlet nodes
integer Wmax                    !number of wall faces
integer Emax                    !number of edges
integer tedge, bedge, iedge     !where different types of edges are
integer Pmax                    !number of periodic nodes
integer ENmax                   !number of edge nodes
integer tnode, bnode, inode     !location of diff. types of edge nodes

real pitch                      !blade pitch
real pout                       !outlet pressure for blades
real Sinl                       !tan of inlet flow angle
real x(Maxnodes)                !x values of nodes
real y(Maxnodes)                !y values of nodes
integer eface(Maxedges)         !array of edge faces
integer enode(Maxedges)         !array of edge nodes
real senode(Maxedges)           !sin and cos at edge nodes
real cenode(Maxedges)
integer face(Maxfaces,8)        !array of faces
integer cell(Maxcells,8)        !array of cells
integer innode(Maxedges)        !inlet nodes
integer outnode(Maxedges)       !outlet nodes
integer pnode(Maxedges,2)       !periodic nodes
integer NCcolor(50)             !number of cells colored each color
integer Ccolormax               !max number of colors used
integer NFcolor(50)             !number of faces colored each color
integer Fcolormax               !max number of colors used
integer NEcolor(10)             !number of edges colored each color
integer E1colormax, E2colormax  !max number of colors used

common /quad/ Imax, Omax, Wmax, Pmax
common /quad/ Nmax, Fmax, Cmax, Emax, tedge, bedge, iedge
common /quad/ ENmax, tnode, bnode, inode
```

```
      common /quad/ pitch, pout, Sinl
      common /quad/ x, y, eface, enode, senode, cenode, face, cell
      common /quad/ innode, outnode, pnode
      common /color/ NCcolor, Ccolormax, NFcolor, Fcolormax
      common /color/ NEcolor, E1colormax, E2colormax

      real Minl                        !inlet Mach number
      integer Maxiter                  !max number of iterations
      real CFL                         !CFL number
      real epsicoef, eps2              !smoothing coef
      real sigE, sigV                  !smoothing coef
      real vol(Maxcells)               !for vorticity smoothing
      real durms                       !rms difference in state vector
      real U(4,0:Maxnodes)             !state vector
      real dU(4,Maxnodes)              !change in state vector at nodes
      real Uc(4,Maxcells)              !state vector at cells
      real dUc(4,0:Maxcells)           !change in state vector at cells
      real F(4,Maxnodes)               !flux vectors at nodes
      real G(4,Maxnodes)
      real areaN(0:Maxnodes)           !control area around each node
      real areaC(0:Maxcells)           !area of each cell
      real deltC(0:Maxcells)           !time step at each cell
      real deltN(0:Maxnodes)           !time step at each node
      real dis(4,Maxnodes)             !dissipation at each node
      real flux(4,0:Maxnodes)          !flux at each node

      common /flo/ Minl, durms, Maxiter, CFL
      common /flo/ epsicoef, eps2, sigE, sigV, vol
      common /flo/ U, dU, Uc, dUc
      common /flo/ F, G
      common /flo/ deltC, deltN, areaC, areaN, dis, flux

      parameter gam = 1.4
      parameter gam1 = 0.4
```

This file contains the subroutines for input and output and is linked with all the quadrilateral schemes.

```
c*****************************************************************
c*                                                               *
c*      write or read grid data from file                        *
c*                                                               *
c*****************************************************************

      subroutine gridio(process)
      implicit none

      include 'QUAD.INC'
      integer process                        !read or write?
      integer i, NN, FF, CC, EN, EF, col     !pointers
      integer IN, ON, WN, PN


      if (process.eq.1) then

c** read in data from file
         open(unit=30, status='unknown', form='unformatted')

         read(30) Nmax, (x(NN),NN=1,Nmax), (y(NN),NN=1,Nmax)
         read(30) Fmax, ((face(FF,i),i=1,8),FF=1,Fmax)
         read(30) tedge, bedge, iedge
         read(30) Wmax, Emax, (eface(EF),EF=1,Emax)
         read(30) tnode, bnode, inode, ENmax, (enode(EN),EN=1,ENmax)
         read(30) (senode(EN),EN=1,ENmax), (cenode(EN),EN=1,ENmax)
         read(30) Imax, (innode(IN),IN=1,Imax)
```

```
            read(30) Omax, (outnode(ON),ON=1,Omax)
            read(30) Pmax, ((pnode(PN,i),i=1,2),PN=1,Pmax)
            read(30) Cmax, ((cell(CC,i),i=1,8),CC=1,Cmax)
            read(30) Ccolormax, (NCcolor(col),col=1,Ccolormax)
            read(30) Fcolormax, (NFcolor(col),col=1,Fcolormax)
            read(30) E1colormax, E2colormax
            read(30) (NEcolor(col),col=1,E2colormax)
            read(30) pitch

            close(unit=30)

         else if (process.eq.0) then

c*  write data to file
            open(unit=30, status='unknown', form='unformatted')

            write(30) Nmax, (x(NN),NN=1,Nmax), (y(NN),NN=1,Nmax)
            write(30) Fmax, ((face(FF,i),i=1,8),FF=1,Fmax)
            write(30) tedge, bedge, iedge
            write(30) Wmax, Emax, (eface(EF),EF=1,Emax)
            write(30) tnode, bnode, inode, ENmax, (enode(EN),EN=1,ENmax)
            write(30) (senode(EN),EN=1,ENmax), (cenode(EN),EN=1,ENmax)
            write(30) Imax, (innode(IN),IN=1,Imax)
            write(30) Omax, (outnode(ON),ON=1,Omax)
            write(30) Pmax, ((pnode(PN,i),i=1,2),PN=1,Pmax)
            write(30) Cmax, ((cell(CC,i),i=1,8),CC=1,Cmax)
            write(30) Ccolormax, (NCcolor(col),col=1,Ccolormax)
            write(30) Fcolormax, (NFcolor(col),col=1,Fcolormax)
            write(30) E1colormax, E2colormax
            write(30) (NEcolor(col),col=1,E2colormax)
            write(30) pitch

            close(unit=30)

         endif

         return
         end



c*****************************************************************
c*                                                               *
c*      write or read flow data from file                        *
c*                                                               *
c*****************************************************************

         subroutine flowio(process)
         implicit none

         include 'QUAD.INC'
         integer process                !read or write?
         integer i, NN                  !pointers


         if (procers.eq.1) then

c** read in data from file
            open(unit=25, status='unknown', form='unformatted')

            read(25) Minl
            read(25) Nmax, ((U(i,NN),i=1,4),NN=1,Nmax)

            close(unit=25)

         else if (process.eq.0) then

c*  write data to file
            open(unit=25, status='unknown', form='unformatted')

            write(25) Minl
```

```
      write(25) Nmax, ((U(i,NN),i=1,4),NN=1,Nmax)

      close(unit=25)

    endif

    return
    end


c*****************************************************************
c*                                                               *
c*      read input data from file                                *
c*                                                               *
c*****************************************************************

    subroutine input
    implicit none

    include 'QUAD.INC'
    real pi                          !the one and only

    pi = 3.14159

    open(unit=20, status='old')

    read(20,*) Maxiter, CFL
    read(20,*) Minl, Sinl, pout
    read(20,*) sigE, sigV, epsicoef, eps2

    close (unit=20)

    pout = pout/gam
    Sinl = tan(pi*Sinl/180.0)

    return
    end
```

A.2.2 Mesh Generator

This is file GRID.INC which includes many declarations and common block statements for the mesh generator.

```
    parameter Maxdim = 200

    real xx(0:Maxdim,0:Maxdim)       !x coordinate of grid points
    real yy(0:Maxdim,0:Maxdim)       !y coordinate of grid points
    integer Nx                       !number cells on x axis
    integer Ny                       !  "      "   "  y  "
    integer ILE, ITE                 !leading and trailing edge of blade

    common /grid/ xx, yy, Nx, Ny
    common /grid/ ILE, ITE


    parameter IBX=251

    real GSINL, GSOUT, CHINL, CHOUT
    integer II, JJ
    real XB(IBX), XPB(IBX), YB(IBX), YPB(IBX), SB(IBX)
```

```
      integer IIB, IBLE
      real SBLE, SBLOLD, SS, SP
      real XMIN, XMAX, YMIN, XLE, YLE, XPOS(Maxdim), YPOS(Maxdim)
      integer NINL, NOUT, NBLD

      common /c01/ GSINL, GSOUT, CHINL, CHOUT, II, JJ,
     &             XB, XPB, YB, YPB, SB,
     &             IIB, IBLE, SBLE, SBLOLD, SS, SP,
     &             XMIN, XMAX, YMIN, XLE, YLE, XPOS, YPOS,
     &             NINL, NOUT, NBLD

c********************  FACE ARRAY  ********************
c
c  N6_________N1_____ N3      face(F,1) = C1
c    |        |       |       face(F,2) = C2
c    |        |       |       face(F,3) = N1
c    |        |       |       face(F,4) = N2
c    |        F       |       face(F,5) = N3
c    |  C2    |   C1  |       face(F,6) = N4
c    |        |       |       face(F,7) = N5
c    |        |       |       face(F,8) = N6
c  N5--------N2--------N4
c
c
c********************  CELL ARRAY  ********************
c
c      N3____F3____N2          cell(C,1) = F1
c       |           |          cell(C,2) = F2
c       |           |          cell(C,3) = F3
c     F4|    C      |F2        cell(C,4) = F4
c       |           |          cell(C,5) = N1
c       |           |          cell(C,6) = N2
c     N4|___________|N1        cell(C,7) = N3
c           F1                 cell(C,8) = N4
c
c
c********************  EDGE ARRAY  ********************
c
c      N4__________N2
c       |          |
c       |          |
c       |    C1    |F(edge)
c       |          |
c       |          |
c     N3|__________|N1
c                              eface(EF) = F
c
c************************************************************


      program gridgen
      implicit none

      include 'QUAD.INC'
      include 'GRID.INC'
      integer gtype                     !type of grid
      integer EN, EF, FF, NN, CC, col !pointers
      integer ON, IN, PN
      integer i, j                      !pointers
      integer JJJ

c**  determine geometry
      type*, 'Type of grid '
      type*, '   1) Ni bump   '
      type*, '   2) blade'
      type 100
 100  format($,' selection = ')
      accept 110, gtype
 110  format(I)

      write(6,*) ' '
```

```
      write(6,*) 'Generating grid ...'
      write(6,*) ' '

      if (gtype.eq.1) then
c**  create rectangular mesh
         call rectangle
         pitch = 0.

      else if (gtype.eq.2) then
c** Read in, normalize and spline blade data
         call readin

c** Initialize grid
         call grinit

c** Fix up grid
         call ellip(Maxdim,Maxdim,II,JJ,JJJ,XX,YY,YPOS,XPOS)
         call improv

c** change pointers
         do i = 1, ii
            do j = 1, jj
               xx(i-1,j-1) = xx(i,j)
               yy(i-1,j-1) = yy(i,j)
            enddo
         enddo
         Nx = ii-1
         Ny = jj-1
      endif

c**  change rectangular mesh to triangular mesh
      call pointers
      call bpointers

c**  rearrange node numbers on edges
      call edgenumber

c**  color cells and faces
      call cellcolor
      call facecolor
      call edgecolor

c**  write out data to file
      call gridio(0)

      stop
      end



      subroutine rectangle
      implicit none

      include 'GRID.INC'

      real*4 yymax, xxmax, xxmin      !coordinates of grid boundaries
      real*4 delta_x                  !grid spacing in x direction
      real*4 delta_y                  !grid spacing in y direction
      integer n, i, j, kk, m          !counters
      real*4 num
      real*4 delta_xbot
      real*4 omega                    !relaxation constant for
                                      !interior point SLOR
      real*4 pi                       !the one and only
      real*4 tau                      !height of bump
      real*4 rr                       !radius of bump
      real*4 yc                       !rr-tau
      real*4 ang, delta_ang           !angle of bump,angle between nodes
      real*4 angplus
      real*4 delx, dely               !help define initial conditions
      real*4 alpha(Maxdim), beta(Maxdim) !coef. in modefied equation
```

```
      real*4 gamma(Maxdim)
      real*4 AA(Maxdim), BB(Maxdim)   !values for system of equations
      real*4 CC(Maxdim), DD(Maxdim)
      real*4 EPS                      !allowable error
      integer Niter                   !number of iterations so far
      real*4 error                    !largest error for an iteration
      real*4 x_xi(Maxdim), x_eta(Maxdim) !derivatives on boundary
      real*4 x_xi_xi(Maxdim), x_eta_eta(Maxdim)
      real*4 x_eta_xi(Maxdim)
      real*4 y_xi(Maxdim), y_eta(Maxdim)
      real*4 y_xi_xi(Maxdim), y_eta_eta(Maxdim)
      real*4 y_eta_xi(Maxdim)
      real*4 theta, AR                !angle and aspect ratio of edge cells
      real*4 R1, R2                   !part of source term
      real*4 Q1(Maxdim), P1(Maxdim)   !part of source term, function of xi
      real*4 a1(Maxdim), b1(Maxdim)   !alpha, beta, gamma on lower boundary
      real*4 c1(Maxdim)
      real*4 omega_P, omega_Q         !relexation consts. for source terms
      real*4 Jacobi(Maxdim)           !Jacobian on the lower boundary
      real*4 Jacobi2(Maxdim)          !Jacobian squared in region
      real*4 xxi, yxi, xeta, yeta     !derivatives in region
      real*4 a, b                     !exponents in source terms

      pi = 3.14159

c** number of nodes on x and y axes
      xxmax = 2.
      xxmin = -1.
      yymax = 1.

      call Irequest('        Nx',Nx)
      call Irequest('        Ny',Ny)
      call Rrequest('       tau',tau)
c     call Rrequest('     omega',omega)
c     call Rrequest('   omega_P',omega_P)
c     call Rrequest('   omega_Q',omega_Q)
c     call Rrequest('         a',a)
c     call Rrequest('         b',b)
c     call Rrequest('        AR',AR)
c     call Rrequest('       EPS',EPS)

      omega = 1.
      a = 0.8
      b = 0.8
      AR = 0.5
      EPS = 0.0005

      if (tau.eq.0.) then
         omega_P = 0.
         omega_Q = 0.
      else if (tau.lt.0.) then
         AR = 1.
         omega_P = 0.02
         omega_Q = 0.02
      else
         omega_P = 0.02
         omega_Q = 0.02
      endif

      theta = .5*pi
      delta_x = (xxmax-xxmin)/real(Nx)

c** set initial and boundary conditions for x and y
      if (tau.gt.0.) then
         yc = 0.5*(-tau**2 + 0.25)/tau
         rr = sqrt(0.25 + yc**2)
         ang = asin(.5/rr)
         if (mod(Nx,3).ne.0) then
            num = real(Nx - mod(Nx,3))
            delta_xbot = (xxmax-xxmin)/num
            angplus = 2.*ang/(num/3. + real(mod(Nx,3)))
         else
```

```
            angplus = 2.*ang*delta_x
            delta_xbot = delta_x
         endif
         delta_ang = 0.
         do i = 0, Nx
            yy(i,Ny) = yymax
            xx(i,Ny) = xxmin + i*delta_x
            if (xx(i,Ny).le.0.) then
               xx(i,0) = xxmin + i*delta_xbot
               yy(i,0) = 0.
            else if (xx(i,Ny).gt.0. .and. xx(i,Ny).lt.1.) then
               delta_ang = delta_ang + angplus
               xx(i,0) = 0.5 - rr*sin(ang-delta_ang)
               yy(i,0) = rr*cos(ang-delta_ang) - yc
            else if (xx(i,Ny).ge.1.) then
               xx(i,0) = xxmin + (i-mod(Nx,3))*delta_xbot
               yy(i,0) = 0.
            endif
            delx = (xx(i,Ny) - xx(i,0))/real(Ny)
            dely = (yy(i,Ny) - yy(i,0))/real(Ny)
            do j = 1, Ny-1
               xx(i,j) = delx*j + xx(i,0)
               yy(i,j) = dely*j + yy(i,0)
            enddo
         enddo
      else if (tau.lt.0.) then
         delta_xbot = delta_x
         do i = 0, Nx
            yy(i,Ny) = yymax
            xx(i,Ny) = xxmin + i*delta_x
            if (xx(i,Ny).le.0.) then
               xx(i,0) = xxmin + i*delta_xbot
               yy(i,0) = 0.
            else if (xx(i,Ny).gt.0. .and. xx(i,Ny).lt.1.) then
               xx(i,0) = xxmin + i*delta_xbot
               yy(i,0) = -tau*(sin(pi*xx(i,0)))**2
            else if (xx(i,Ny).ge.1.) then
               xx(i,0) = xxmin + i*delta_xbot
               yy(i,0) = 0.
            endif
            delx = (xx(i,Ny) - xx(i,0))/real(Ny)
            dely = (yy(i,Ny) - yy(i,0))/real(Ny)
            do j = 1, Ny-1
               xx(i,j) = delx*j + xx(i,0)
               yy(i,j) = dely*j + yy(i,0)
            enddo
         enddo
      else if (tau.eq.0) then
         delta_y = yymax/real(Ny)
         do i = 0, Nx
            do j = 0, Ny
               xx(i,j) = i*delta_x + xxmin
               yy(i,j) = j*delta_y
            enddo
         enddo
      endif

c* Solve for source terms
      do n = 1, Nx-1
         P1(n) = 0.
         Q1(n) = 0.
         x_xi(n) = .5*(xx(n+1,0) - xx(n-1,0))
         y_xi(n) = .5*(yy(n+1,0) - yy(n-1,0))

         x_xi_xi(n) = (xx(n+1,0) - 2.*xx(n,0) + xx(n-1,0))
         y_xi_xi(n) = (yy(n+1,0) - 2.*yy(n,0) + yy(n-1,0))

         if (xx(n,0).eq.0. .or. xx(n,0).eq.1.) then
            x_eta(n) = max(5.*tau,1.)*AR*(-x_xi(n)*cos(theta)
     &                    - y_xi(n)*sin(theta))
            y_eta(n) = max(5.*tau,1.)*AR*(-y_xi(n)*cos(theta)
     &                    + x_xi(n)*sin(theta))
```

```
            else
               x_eta(n) = AR*(-x_xi(n)*cos(theta) - y_xi(n)*sin(theta))
               y_eta(n) = AR*(-y_xi(n)*cos(theta) + x_xi(n)*sin(theta))
            endif

            x_eta_xi(n) = 0.5*(x_eta(n+1)-x_eta(n-1))
            y_eta_xi(n) = 0.5*(y_eta(n+1)-y_eta(n-1))

            Jacobi(n) = x_xi(n)*y_eta(n) - x_eta(n)*y_xi(n)

            ai(n) = x_eta(n)**2 + y_eta(n)**2
            bi(n) = x_xi(n)*x_eta(n) + y_xi(n)*y_eta(n)
            ci(n) = x_xi(n)**2 + y_xi(n)**2
         enddo

c* SOR by lines
         Niter = 0
         error = 9999.
         do while (error.gt.EPS)
            error = 0.
            Niter = Niter + 1

c** solve for source terms
            if (tau.ne.0.) then
               do n = 1, Nx-1
                  x_eta_eta(n) = 0.5*(-7.*xx(n,0) + 8.*xx(n,1) - xx(n,2))
     &                           - 3.*x_eta(n)
                  y_eta_eta(n) = 0.5*(-7.*yy(n,0) + 8.*yy(n,1) - yy(n,2))
     &                           - 3.*y_eta(n)
                  R1 = (-ai(n)*x_xi_xi(n) + 2.*bi(n)*x_eta_xi(n)
     &                 - ci(n)*x_eta_eta(n))/Jacobi(n)**2
                  R2 = (-ai(n)*y_xi_xi(n) + 2.*bi(n)*y_eta_xi(n)
     &                 - ci(n)*y_eta_eta(n))/Jacobi(n)**2
                  P1(n) = P1(n) + omega_P*((y_eta(n)*R1 - x_eta(n)*R2)
     &                          /Jacobi(n) - P1(n))
                  Q1(n) = Q1(n) + omega_Q*((-y_xi(n)*R1 + x_xi(n)*R2)
     &                          /Jacobi(n) - Q1(n))
               enddo
            endif

c* solve for each line
            do kk = 1, Ny-1

c* evaluate alpha, beta, and gamma
               do i = 1, Nx-1
                  alpha(i) = .25*(xx(i,kk+1)-xx(i,kk-1))**2
     &                     + .25*(yy(i,kk+1)-yy(i,kk-1))**2
                  beta(i) = .25*(xx(i+1,kk)-xx(i-1,kk))*(xx(i,kk+1)
     &                 -xx(i,kk-1)) + .25*(yy(i+1,kk)-yy(i-1,kk))
     &                 *(yy(i,kk+1)-yy(i,kk-1))
                  gamma(i) = .25*(xx(i+1,kk)-xx(i-1,kk))**2
     &                     + .25*(yy(i+1,kk)-yy(i-1,kk))**2
                  Jacobi2(i) = (.25*(xx(i+1,kk)-xx(i-1,kk))*
     &                              (yy(i,kk+1)-yy(i,kk-1))
     &                         - .25*(yy(i+1,kk)-yy(i-1,kk))*
     &                              (xx(i,kk+1)-xx(i,kk-1)))**2
               enddo

c* Solve for x
c* set up matrix for tridiagonal system of equations
               do i = 1, Nx-1
                  AA(i) = omega*alpha(i)
                  DD(i) = -2.*(gamma(i) + alpha(i))
                  BB(i) = omega*alpha(i)
               enddo

c* set up vector of constants for tridiagonal system of equations
               do i = 1, Nx-1
                  xxi = xx(i+1,kk) - xx(i-1,kk)
                  xeta = xx(i,kk+1) - xx(i,kk-1)

                  CC(i) = -omega*gamma(i)*(xx(i,kk+1) + xx(i,kk-1))
```

```
     &                   + omega*0.5*beta(i)*(xx(i+1,kk+1)
     &                   - xx(i+1,kk-1) - xx(i-1,kk+1) + xx(i-1,kk-1))
     &                   + 2.*(omega-1.)*(alpha(i) + gamma(i))*xx(i,kk)
     &                   - Jacobi2(i)*(P1(i)*xxi*exp(-a*kk) +
     &                   Q1(i)*xeta*exp(-b*kk))
               enddo
               CC(1)    = CC(1)    - BB(1)*xx(0,kk)
               CC(Nx-1) = CC(Nx-1) - AA(Nx-1)*xx(Nx,kk)

c* solve tridiagonal system of equations
               call tridiag(1, Nx-1, BB, DD, AA, CC)
               do m = 1, Nx-1
                  if (abs(CC(m)-xx(m,kk)).gt.error) then
                     error = abs(CC(m) - xx(m,kk))
                  endif
                  xx(m,kk) = CC(m)
               enddo

c* Solve for y
c* set up matrix for tridiagonal system of equations
               do i = 1, Nx-1
                  AA(i) = omega*alpha(i)
                  DD(i) = -2.*(gamma(i) + alpha(i))
                  BB(i) = omega*alpha(i)
               enddo

c* set up vector of constants for tridiagonal system of equations
               do i = 1, Nx-1
                  yxi = yy(i+1,kk) - yy(i-1,kk)
                  yeta = yy(i,kk+1) - yy(i,kk-1)

                  CC(i) = -omega*gamma(i)*(yy(i,kk+1) + yy(i,kk-1))
     &                    + 0.5*omega*beta(i)*(yy(i+1,kk+1)
     &                    -yy(i+1,kk-1) - yy(i-1,kk+1) + yy(i-1,kk-1))
     &                    + 2.*(omega-1.)*(alpha(i)+gamma(i))*yy(i,kk)
     &                    - Jacobi2(i)*(P1(i)*yxi*exp(-a*kk) +
     &                    Q1(i)*yeta*exp(-b*kk))
               enddo
               CC(1)    = CC(1)    - BB(1)*yy(0,kk)
               CC(Nx-1) = CC(Nx-1) - AA(Nx-1)*yy(Nx,kk)

c* solve tridiagonal system of equations
               call tridiag(1, Nx-1, BB, DD, AA, CC)
               do m = 1, Nx-1
                  if (abs(CC(m)-yy(m,kk)).gt.error) then
                     error = abs(CC(m) - yy(m,kk))
                  endif
                  yy(m,kk) = CC(m)
               enddo

            enddo

c* set x=xxmax and x=xxmin boundary conditions to next interior point
            do kk = 1, Ny-1
               yy(0,kk) = yy(1,kk)
               yy(Nx,kk) = yy(Nx-1,kk)
            enddo

            type*, 'iteration number = ',Niter,'  error = ',error
         enddo

c**  write out data to data file
C        write(10,*) Nx+1, Ny+1
C        do i = 0, Nx
C           do j = 0, Ny
C              write(10,*) xx(i,j), yy(i,j)
C           enddo
C        enddo
c        close(unit=10)

c**  plot the grid
C        call plot
```

```
        return
        end



c*************************************************************************
c*                                                                       *
c*      Subroutine to solve a tridiagonal system of equations.           *
c*      Taken from "Computational Fluid Mechanics and Heat Transfer"     *
c*      by Anderson, Tannehill and Pletcher.                             *
c*                                                                       *
c*************************************************************************

        SUBROUTINE TRIDIAG(IL,IU,BB,DD,AA,RR)
        IMPLICIT NONE

        include 'GRID.INC'

        INTEGER IL                      !SUBSCRIPT OF FIRST EQUATION
        INTEGER IU                      !SUBSCRIPT OF LAST EQUATION
        REAL*4 BB(Maxdim)               !COEFFICIENT BEHIND DIAGONAL
        REAL*4 DD(Maxdim)               !COEFFICIENT ON DIAGONAL
        REAL*4 AA(Maxdim)               !COEFFICIENT AHEAD OF DIAGONAL
        REAL*4 RR(Maxdim)               !ELEMENT OF CONSTANT VECTOR
        INTEGER LP
        INTEGER I, J                    !POINTERS
        REAL*4 R

C**     ESTABLISH UPPER TRIANGULAR MATRIX

        LP = IL + 1
        DO I = LP, IU
           R = BB(I)/DD(I-1)
           DD(I) = DD(I)-R*AA(I-1)
           RR(I) = RR(I)-R*RR(I-1)
        ENDDO

C**     BACK SUBSTITUTION

        RR(IU) = RR(IU)/DD(IU)
        DO I = LP, IU
           J = IU - I + IL
           RR(J) = (RR(J)-AA(J)*RR(J+1))/DD(J)
        ENDDO

C**     SOLUTION STORED IN RR

        RETURN
        END



c*************************************************************************
c*                                                                       *
c*      this subroutine requests the user to input the value of a        *
c*      real variable                                                    *
c*                                                                       *
c*************************************************************************

        subroutine Rrequest (name,var)
        character*10 name
        real*4 var

        write(6,1) name
 1      format($,' ',A,' = ')
        accept 11, var
 11     format(F)

        return
```

```
        end

c*************************************************************************
c*                                                                       *
c*      this subroutine requests the user to input the value of a        *
c*      integer variable                                                 *
c*                                                                       *
c*************************************************************************

        subroutine Irequest (name,var)
        character*10 name
        integer var

        write(6,1) name
 1      format($,' ',A,' = ')
        accept 11, var
 11     format(I)

        return
        end



      SUBROUTINE READIN

C---- Read in, normalize and spline blade data

      INCLUDE 'QUAD.INC'
      INCLUDE 'GRID.INC'

      CHARACTER*32 NAMEXT
      CHARACTER*80 NAME

C---- Read in blade data
      OPEN(UNIT=3,STATUS='OLD')
 1000 FORMAT(A32)
      READ(3,1000) NAME
      READ(3,*) GSINL, GSOUT, CHINL, CHOUT, PITCH

      WRITE(6,1001) NAME
 1001 FORMAT(/,' Blade name: ',A60)

      READ(3,*) XB(1), YB(1)
      XMIN = XB(1)
      XMAX = XB(1)
      YMIN = YB(1)
      DO 1 IB = 2, 12345
        READ(3,*,END=11) XB(IB),YB(IB)
        XMAX = AMAX1(XMAX,XB(IB))
        IF(XMIN.GT.XB(IB)) THEN
         XMIN = XB(IB)
         YMIN = YB(IB)
        ENDIF
 1    CONTINUE
 11   IIB = IB - 1
      CLOSE(UNIT=3)

      IF(IIB.GT.IBX) STOP 'Array overflow:  IBX too small'

C---- Normalize blade and calculate surface arc length array
      PITCH = PITCH/(XMAX-XMIN)
      DO 2 IB = 1, IIB
        XB(IB) = (XB(IB)-XMIN) / (XMAX-XMIN)
        YB(IB) = (YB(IB)-YMIN) / (XMAX-XMIN)
 2    CONTINUE

C---- close t.e. if open
      IF( XB(1).NE.XB(IIB) .OR. YB(1).NE.YB(IIB) ) THEN
       ABOUT = ATAN( (YB(2)  -YB(1))     / (XB(2)  -XB(1))     )
```

```
      APOUT = ATAN( (YB(IIB)-YB(IIB-1)) / (XB(IIB)-XB(IIB-1)) )
      DSTE = SQRT( (XB(1)-XB(IIB))**2 + (YB(1)-YB(IIB))**2 )
      AOUT = 0.5*(APOUT+ASOUT)
      XOUT = 0.5*(XB(1)+XB(IIB)) + 3.0*DSTE*COS(AOUT)
      YOUT = 0.5*(YB(1)+YB(IIB)) + 3.0*DSTE*SIN(AOUT)
      XS1 =  (XB(1)-XOUT)*COS(AOUT) + (YB(1)-YOUT)*SIN(AOUT)
      YS1 = -(XB(1)-XOUT)*SIN(AOUT) + (YB(1)-YOUT)*COS(AOUT)
      YPS1 = TAN(ASOUT-AOUT) * XS1
      XP1 =  (XB(IIB)-XOUT)*COS(AOUT) + (YB(IIB)-YOUT)*SIN(AOUT)
      YP1 = -(XB(IIB)-XOUT)*SIN(AOUT) + (YB(IIB)-YOUT)*COS(AOUT)
      YPP1 = TAN(APOUT-AOUT) * XP1

      WRITE(6,1002)
 1002 FORMAT(/,' Input flap deflection angle (degrees): ',$)
      READ(5,*) AFLAP
      YPFLAP = TAN(3.14159*AFLAP/180.0)

      WRITE(NAMEXT,1003) AFLAP
 1003 FORMAT(' (flap deflection angle =',F4.1,')')
      LENSTART = INDEX(NAME,'        ')
      NAME(LENSTART:LENSTART+31) = NAMEXT

      IIB = IIB+20
      IF(IIB.GT.IBX) STOP 'Array overflow:  IBX too small'

      DO 3 IB = IIB-20, 1, -1
        XB(IB+10) = XB(IB)
        YB(IB+10) = YB(IB)
 3    CONTINUE

      DO 4 IB = 1, 10
        ETA = 0.1*(IB-1)
        XXS = XS1*ETA
        YYS = YS1*ETA*ETA*(3.0-2.0*ETA) + YPS1*ETA*ETA*(ETA-1.0)
     &      - YPFLAP*XS1*0.5*(ETA-1.0)**2
        XB(IB) = XOUT + XXS*COS(AOUT) - YYS*SIN(AOUT)
        YB(IB) = YOUT + XXS*SIN(AOUT) + YYS*COS(AOUT)
 4    CONTINUE

      DO 5 IB = IIB-9, IIB
        ETA = 0.1*(IIB-IB)
        XXP = XP1*ETA
        YYP = YP1*ETA*ETA*(3.0-2.0*ETA) + YPP1*ETA*ETA*(ETA-1.0)
     &      - YPFLAP*XP1*0.5*(ETA-1.0)**2
        XB(IB) = XOUT + XXP*COS(AOUT) - YYP*SIN(AOUT)
        YB(IB) = YOUT + XXP*SIN(AOUT) + YYP*COS(AOUT)
 5    CONTINUE

      ENDIF

C---- Spline blade surface(s) and find leading edge position
      SB(1) = 0.
      DO 6 IB = 2, IIB
        ALF = FLOAT( MIN(IB-1,IIB-IB) ) / FLOAT(IIB/2)
        SB(IB) = SB(IB-1) +
     &       SQRT( (XB(IB)-XB(IB-1))**2 + (ALF*(YB(IB)-YB(IB-1)))**2 )
 6    CONTINUE

      CALL SPLINE(XB,XPB,SB,IIB)
      CALL SPLINE(YB,YPB,SB,IIB)

      DO 7 IB=2, IIB
        DP1 = XPB(IB-1) + GSINL*YPB(IB-1)
        DP2 = XPB(IB)   + GSINL*YPB(IB)
        IF(DP1.LT.0.0 .AND. DP2.GE.0.0) GO TO 71
 7    CONTINUE

      STOP 'Leading edge not found'
 71   DSB  = SB(IB) - SB(IB-1)
      SBLE = SB(IB-1) + DSB*DP1/(DP1-DP2)
      XLE  = SEVAL(SBLE,XB,XPB,SB,IIB)
      YLE  = SEVAL(SBLE,YB,YPB,SB,IIB)
```

```
      RETURN
      END



      SUBROUTINE GRINIT

C---- Fix grid points on boundary of domain, and initialize interior

      INCLUDE 'QUAD.INC'
      INCLUDE 'GRID.INC'

C---- Input and check grid size
      WRITE(6,1000)
 1000 FORMAT(/,' Input II, JJ: ',$)
      READ(5,*) II,JJ
      IF(II.GT.Maxdim) STOP 'Array overflow:  Maxdim too small'
      IF(JJ.GT.Maxdim) STOP 'Array overflow:  Maxdim too small'

C---- Set various parameters
      SLEN = CHINL + 0.5*SB(IIB) + CHOUT
      NINL = INT( FLOAT(II)*CHINL/SLEN )
      NOUT = INT( FLOAT(II)*CHOUT/SLEN )
      NBLD = II - NOUT - NINL + 2
      ILE = NINL
      ITE = II - NOUT + 1

C---- Set inlet stagnation streamline
      DO 1 K=1, NINL
        XX(K,1) = XLE + CHINL * FLOAT(K-NINL) / FLOAT(NINL-1)
        YY(K,1) = YLE + (XX(K,1)-XLE) * GSINL
        XX(K,JJ) = XX(K,1)
        YY(K,JJ) = YY(K,1) + PITCH
 1    CONTINUE

C---- Set outlet stagnation streamline
      XTE = XB(1)
      YTE = YB(1)

      DO 2 K= 1, NOUT
        I = II-NOUT+K
        XX(I,1)  = XTE + CHOUT * FLOAT(K-1) / FLOAT(NOUT-1)
        YY(I,1)  = YTE + (XX(I,1)-XTE) * GSOUT
        XX(I,JJ) = XX(I,1)
        YY(I,JJ) = YY(I,1) + PITCH
 2    CONTINUE

C---- Set points on blade suction surface
      DO 3 K=1, NBLD
        I = NINL + K - 1
        S = SBLE - SBLE*FLOAT(K-1)/FLOAT(NBLD-1)
        XX(I,1) = SEVAL(S,XB,XPB,SB,IIB)
        YY(I,1) = SEVAL(S,YB,YPB,SB,IIB)
 3    CONTINUE

C---- Set points on blade pressure surface
      DO 4 K=1, NBLD
        I = NINL + K - 1
        S = SBLE + (SB(IIB)-SBLE)*FLOAT(K-1)/FLOAT(NBLD-1)
        XX(I,JJ) = SEVAL(S,XB,XPB,SB,IIB)
        YY(I,JJ) = SEVAL(S,YB,YPB,SB,IIB) + PITCH
 4    CONTINUE

C---- set up metrics
      DO 5 I=1, II
        XPOS(I) = FLOAT(I-1)/FLOAT(II-1)
 5    CONTINUE

      DO 6 J = 1, JJ
```

```
        RJ = FLOAT(J-1)/FLOAT(JJ-1)
        YPOS(J) = RJ - 1.8 * ( (RJ-0.5) * ((RJ-0.5)**2-0.25) )
 6    CONTINUE

C---- Initialize interior grid
      DO 7 I = 1, II
        DO 71 J =2, JJ-1
          XX(I,J) = XX(I,1) + YPOS(J)*(XX(I,JJ)-XX(I,1))
          YY(I,J) = YY(I,1) + YPOS(J)*(YY(I,JJ)-YY(I,1))
 71     CONTINUE
 7    CONTINUE

      RETURN
      END




      SUBROUTINE IMPROV

C---- Improves grid after elliptic grid generation


      INCLUDE 'QUAD.INC'
      INCLUDE 'GRID.INC'

      DIMENSION SUM(Maxdim), XT1(Maxdim), YT1(Maxdim)
      DIMENSION XT2(Maxdim), YT2(Maxdim)

      DO 1 I = 1, II
        IM = I-1
        IP = I+1
        IF(I.EQ 1) IM=1
        IF(I.EQ.II) IP=II

        SUM(1) = 0.
        DO 11 J = 1, JJ-1
          JP = J+1
          XS = XX(IP,J)+XX(IP,JP) - XX(IM,J)-XX(IM,JP)
          YS = YY(IP,J)+YY(IP,JP) - YY(IM,J)-YY(IM,JP)
          SS = SQRT(XS*XS + YS*YS)
          XS = XS/SS
          YS = YS/SS
          SUM(JP) = SUM(J) + ABS( (XX(I,J)-XX(I,JP))*YS
     &                          - (YY(I,J)-YY(I,JP))*XS )
 11     CONTINUE

        J = 1
        DO 12 JO = 2, JJ-1
          SUMJ = FLOAT(JO-1)/FLOAT(JJ-1) * SUM(JJ)
 121       IF(SUMJ.GT.SUM(J+1)) THEN
            J = J+1
            GOTO 121
          ENDIF
          ALPHA = (SUMJ-SUM(J)) / (SUM(J+1)-SUM(J))
          XT2(JO) = XX(I,J) + ALPHA*(XX(I,J+1)-XX(I,J))
          YT2(JO) = YY(I,J) + ALPHA*(YY(I,J+1)-YY(I,J))
 12     CONTINUE

        DO 13 J = 2, JJ-1
          IF(I.NE.1) THEN
            XX(IM,J) = XT1(J)
            YY(IM,J) = YT1(J)
          ENDIF
          XT1(J) = XT2(J)
          YT1(J) = YT2(J)
          IF(I.EQ.II) THEN
            XX(I,J) = XT1(J)
            YY(I,J) = YT1(J)
          ENDIF
```

```
 13     CONTINUE

 1    CONTINUE
      RETURN
      END



C*************************************************C
C                                                 C
C   ISES - an Integrated Steamtube Euler Solver   C
C                                                 C
C   Written by M. Giles and M.Drela               C
C                                                 C
C   Copyright M.I.T. (1985)                       C
C                                                 C
C*************************************************C
C
      SUBROUTINE ELLIP(IMAX,JMAX,II,JJ,JJJ,X,Y,YPOS,XPOS)
      DIMENSION X(0:IMAX,0:JMAX), Y(0:IMAX,0:JMAX)
      DIMENSION YPOS(JMAX), XPOS(IMAX)
      CHARACTER*1 ANS
C
      DIMENSION C(400),D(2,400)
      IF(II.GT.400) STOP 'ELLIP dimensions must be increased'
C
      ITMAX = 50
C
      DSET1 = 1.0E-1
      DSET2 = 5.0E-3
      DSET3 = 2.0E-4
C
      RLX1 = 1.30           !          DMAX > DSET1
      RLX2 = 1.50           !  DSET1 > DMAX > DSET2
      RLX3 = 1.60           !  DSET2 > DMAX > DSET3
CCC   STOP                  !  DSET3 > DMAX
C
      RLX = RLX1
C
      DO 1 ITER = 1, ITMAX
C
        DMAX = 0.
        DO 5 JO=2, JJ-1
          JM = JO-1
          JP = JO+1
C
          IF(JO.EQ.JJJ) THEN
           DO 2 IO=2, II-1
             X(IO,JO) = X(IO,JM)
  2        CONTINUE
           GO TO 5
          ELSE IF(JO.EQ.JJJ+1) THEN
           DO 3 IO=2, II-1
             X(IO,JO) = X(IO,JP)
  3        CONTINUE
           GO TO 5
          ENDIF
C
          DO 6 IO=2, II-1
            IM = IO-1
            IP = IO+1
C
            XMM = X(IM,JM)
            XOM = X(IO,JM)
            XPM = X(IP,JM)
            XMO = X(IM,JO)
            XOO = X(IO,JO)
            XPO = X(IP,JO)
            XMP = X(IM,JP)
            XOP = X(IO,JP)
```

```
            XPP = X(IP,JP)
            YMM = Y(IM,JM)
            YOM = Y(IO,JM)
            YPM = Y(IP,JM)
            YMO = Y(IM,JO)
            YOO = Y(IO,JO)
            YPO = Y(IP,JO)
            YMP = Y(IM,JP)
            YOP = Y(IO,JP)
            YPP = Y(IP,JP)
C
            DXIM = XPOS(IO)-XPOS(IM)
            DXIP = XPOS(IP)-XPOS(IO)
            DXIAV = 0.5*(DXIM+DXIP)
C
            DETM = YPOS(JO)-YPOS(JM)
            DETP = YPOS(JP)-YPOS(JO)
            DETAV = 0.5*(DETM+DETP)
C
            DXDET = ( XOP - XOM ) / DETAV
            DYDET = ( YOP - YOM ) / DETAV
            DXDXI = ( XPO - XMO ) / DXIAV
            DYDXI = ( YPO - YMO ) / DXIAV
C
            ALF = DXDET**2 + DYDET**2
            BET = DXDET*DXDXI + DYDET*DYDXI
            GAM = DXDXI**2 + DYDXI**2
C
            CXIM = 1.0 / (DXIM*DXIAV)
            CXIP = 1.0 / (DXIP*DXIAV)
            CETM = 1.0 / (DETM*DETAV)
            CETP = 1.0 / (DETP*DETAV)
C
            B =      -ALF*CXIM
            A = ALF*(CXIM+CXIP) + GAM*(CETM+CETP)
            C(IO) =  -ALF*CXIP
            IF(IO.EQ.2) B = 0
C
            D(1,IO) = ALF*((XMO-XOO)*CXIM + (XPO-XOO)*CXIP)
     &              - 2.0*BET*(XPP-XMP-XPM+XMM) / (4.0*DXIAV*DETAV)
     &              + GAM*((XOM-XOO)*CETM + (XOP-XOO)*CETP)
C
            D(2,IO) = ALF*((YMO-YOO)*CXIM + (YPO-YOO)*CXIP)
     &              - 2.0*BET*(YPP-YMP-YPM+YMM) / (4.0*DXIAV*DETAV)
     &              + GAM*((YOM-YOO)*CETM + (YOP-YOO)*CETP)
C
            AINV = 1.0/(A - B*C(IM))
            C(IO) = C(IO) * AINV
            D(1,IO) = ( D(1,IO) - B*D(1,IM) ) * AINV
            D(2,IO) = ( D(2,IO) - B*D(2,IM) ) * AINV
C
    6     CONTINUE
C
          D(1,II) = 0.
          D(2,II) = 0.
C
          IFIN = II-1
          DO 8 IBACK=2, IFIN
            IO = II-IBACK+1
            IP = IO+1
            D( 1,IO) = D( 1,IO) - C(IO)*D(1,IP)
            D( 2,IO) = D( 2,IO) - C(IO)*D(2,IP)
            X(IO,JO) = X(IO,JO) + RLX*D(1,IO)
            Y(IO,JO) = Y(IO,JO) + RLX*D(2,IO)
            AD1 = ABS(D(1,IO))
            AD2 = ABS(D(2,IO))
            DMAX = AMAX1(DMAX,AD1,AD2)
    8     CONTINUE
C
    5   CONTINUE
C
        WRITE(6,*) 'Dmax = ', DMAX, RLX
```

```
C
        RLX = RLX1
        IF(DMAX.LT.DSET1) RLX = RLX2
        IF(DMAX.LT.DSET2) RLX = RLX3
        IF(DMAX.LT.DSET3) RETURN
C
    1   CONTINUE
C
        RETURN
        END ! ELLIP




        SUBROUTINE SPLINE(X,XP,S,II)
C
        DIMENSION X(1),XP(1),S(1)
        DIMENSION A(480),B(480),C(480)
C
        IF(II.GT.480) STOP 'SPLINE: Array overflow'
        DO 1 I = 1, II
C
C------ Beginning points
        IF(I.EQ.1 .OR. S(I).EQ.S(I-1)) THEN
         DSMI = 0.
         DXM  = 0.
         DSPI = 1.0 / (S(I+1)-S(I))
         DXP  = X(I+1) - X(I)
C
C------ End points
        ELSE IF(I.EQ.II .OR. S(I).EQ.S(I+1)) THEN
         DSMI = 1.0 / (S(I) - S(I-1))
         DXM  = X(I) - X(I-1)
         DSPI = 0.
         DXP  = 0.
C
C------ Interior points
        ELSE
         DSMI = 1.0 / (S(I) - S(I-1))
         DXM  = X(I) - X(I-1)
         DSPI = 1.0 / (S(I+1) - S(I))
         DXP  = X(I+1) - X(I)
C
        ENDIF
C
        B(I)  = DSMI
        A(I)  = 2.0 * (DSMI + DSPI)
        C(I)  = DSPI
        XP(I) = 3.0 * (DXP*DSPI**2 + DXM*DSMI**2)
C
    1   CONTINUE
C
        CALL TRISOL(A,B,C,XP,II)
C
        RETURN
        END ! SPLINE




        SUBROUTINE TRISOL(A,B,C,D,KK)
C
        DIMENSION A(1),B(1),C(1),D(1)
C
        DO 1 K = 2, KK
          KM = K - 1
          C(KM) = C(KM) / A(KM)
          D(KM) = D(KM) / A(KM)
          A(K) = A(K) - B(K) * C(KM)
```

```
        D(K) = D(K) - B(K) * D(KM)
  1   CONTINUE
C
      D(KK) = D(KK) / A(KK)
C
      DO 2 K = KK-1, 1, -1
        D(K) = D(K) - C(K) * D(K+1)
  2   CONTINUE
C
      RETURN
      END ! TRISOL




      FUNCTION SEVAL(SS,X,XP,S,N)
      REAL X(1), XP(1), S(1)
C
      ILOW = 1
      I = N
C
   10 IF(I-ILOW .LE. 1) GO TO 11
C
      IMID = (I+ILOW)/2
      IF(SS .LT. S(IMID)) THEN
       I = IMID
      ELSE
       ILOW = IMID
      ENDIF
      GO TO 10
C
   11 DS = S(I) - S(I-1)
      T = (SS-S(I-1)) / DS
      CX1 = DS*XP(I-1) - X(I) + X(I-1)
      CX2 = DS*XP(I)   - X(I) + X(I-1)
      SEVAL = T*X(I) + (1.0-T)*X(I-1) + (T-T*T)*((1.0-T)*CX1 - T*CX2)
      RETURN
      END ! SEVAL




      FUNCTION DEVAL(SS,X,XP,S,N)
      REAL X(1), XP(1), S(1)
C
      ILOW = 1
      I = N
C
   10 IF(I-ILOW .LE. 1) GO TO 11
C
      IMID = (I+ILOW)/2
      IF(SS .LT. S(IMID)) THEN
       I = IMID
      ELSE
       ILOW = IMID
      ENDIF
      GO TO 10
C
   11 DS = S(I) - S(I-1)
      T = (SS-S(I-1)) / DS
      CX1 = DS*XP(I-1) - X(I) + X(I-1)
      CX2 = DS*XP(I)   - X(I) + X(I-1)
      DEVAL = (X(I)-X(I-1) + (1.-4.*T+3.*T*T)*CX1 + T*(3.*T-2.)*CX2)/DS
      RETURN
      END ! DEVAL
```

```
      subroutine pointers
      implicit none

      include 'GRID.INC'
      include 'QUAD.INC'
      integer i, j                        !pointers
      integer NN, FF, CC                  !pointers

      NN = 0
      CC = 0
      FF = 0

c** assign values to node arrays
      do j = 0, My
         do i = 0, Nx
            NN = NN + 1
            x(NN) = xx(i,j)
            y(NN) = yy(i,j)
         enddo
      enddo
      Nmax = NN
      if (Nmax .gt. Maxnodes) then
         write(6,*) 'NN = ', NN
         write(6,*) 'more than Maxnodes required'
      endif

c**  assign horizontal and vertical faces and cells
      do j = 0, My-1
         do i = 1, Nx
            FF = FF + 1
            face(FF,1) = Nx*j + i
            face(FF,2) = Nx*(j-1) + i
            face(FF,3) = (Nx+1)*j + i
            face(FF,4) = (Nx+1)*j + 1 + i
            face(FF,5) = (Nx+1)*(j+1) + i
            face(FF,6) = (Nx+1)*(j+1) + 1 + i
            face(FF,7) = (Nx+1)*(j-1) + 1 + i
            face(FF,8) = (Nx+1)*(j-1) + i
            CC = CC + 1
            cell(CC,1) = FF
            cell(CC,2) = FF + Nx + 1
            cell(CC,3) = FF + 2*Nx + 1
            cell(CC,4) = FF + Nx
            cell(CC,5) = (Nx+1)*j + 1 + i
            cell(CC,6) = (Nx+1)*(j+1) + 1 + i
            cell(CC,7) = (Nx+1)*(j+1) + i
            cell(CC,8) = (Nx+1)*j + i
         enddo
         do i = 1, Nx
            FF = FF + 1
            face(FF,1) = Nx*j + i
            face(FF,2) = Nx*j - 1 + i
            face(FF,3) = (Nx+1)*(j+1) + i
            face(FF,4) = (Nx+1)*j + i
            face(FF,5) = (Nx+1)*(j+1) + 1 + i
            face(FF,6) = (Nx+1)*j + 1 + i
            face(FF,7) = (Nx+1)*j - 1 + i
            face(FF,8) = (Nx+1)*(j+1) - 1 + i
         enddo
         FF = FF + 1
         face(FF,1) = 0
         face(FF,2) = Nx*(j+1)
         face(FF,3) = (Nx+1)*(j+2)
         face(FF,4) = (Nx+1)*(j+1)
         face(FF,5) = 0
         face(FF,6) = 0
         face(FF,7) = (Nx+1)*j + Nx
         face(FF,8) = (Nx+1)*(j+1) + Nx
      enddo
```

```
      Cmax = CC
      if (Cmax .gt. Maxcells) then
         write(6,*) 'CC = ', CC
         write(6,*) 'more than Maxcells required'
      endif

c**   assign top boundary faces
      do i = 1, Nx
         FF = FF + 1
         face(FF,1) = 0
         face(FF,2) = Nx*(Ny-1) + i
         face(FF,3) = (Nx+1)*Ny + i
         face(FF,4) = (Nx+1)*Ny + 1 + i
         face(FF,5) = 0
         face(FF,6) = 0
         face(FF,7) = (Nx+1)*(Ny-1) + 1 + i
         face(FF,8) = (Nx+1)*(Ny-1) + i
      enddo

      Fmax = FF
      if (Fmax .gt. Maxfaces) then
         write(6,*) 'FF = ', FF
         write(6,*) 'more than Maxfaces required'
      endif

c**   set bottom edge face array
      do i = 1, Nx
         face(i,2) = 0
         face(i,7) = 0
         face(i,8) = 0
      enddo

c**   set inlet edge face array
      do j = 1, Ny
         face((2*Nx+1)*j-Nx,2) = 0
         face((2*Nx+1)*j-Nx,7) = 0
         face((2*Nx+1)*j-Nx,8) = 0
      enddo

c**   set outlet edge face array
      do j = 1, Ny
         face((2*Nx+1)*j,1) = 0
         face((2*Nx+1)*j,5) = 0
         face((2*Nx+1)*j,6) = 0
      enddo

      return
      end



      subroutine bpointers
      implicit none

      include 'GRID.INC'
      include 'QUAD.INC'
      integer i, j                        !pointers
      integer n                           !pointer
      integer EN, EF, IN, ON, WN, PN      !pointers
      integer CC, FF                      !pointers
      integer F1, F                       !periodic faces
      real dxM, dyM, dxP, dyP             !change in x & y near node
      real dsM, dsP, dx, dy
      integer node                        !current node processed

      EN = 0
      EF = 0
      IN = 0
      ON = 0
      WN = 0
```

```
        PN = 0

c**  wall boundaries
        if (pitch.eq.0.) then
           do i = 1, Nx
              EF = EF + 1
              eface(EF) = Ny*(2*Nx+1) + i
              EN = EN + 1
              node = (Nx+1)*Ny + i
              enode(EN) = node
              dxP = x(node+1) - x(node)
              dyP = y(node+1) - y(node)
              if (i .eq. 1) then
                 dxM = -dxP
                 dyM = -dyP
              else
                 dxM = x(node-1) - x(node)
                 dyM = y(node-1) - y(node)
              endif
              if ((dxM*dxP + dyM*dyP) .gt. 0.) then
                 dsM = sqrt(dxM**2 + dyM**2)
                 dsP = sqrt(dxP**2 + dyP**2)
                 dx = dxM/dsM + dxP/dsP
                 dy = dyM/dsM + dyP/dsP
              else
                 dx = dxP - dxM
                 dy = dyP - dyM
              endif
              senode(EN) = dy/sqrt(dx**2 + dy**2)
              cenode(EN) = dx/sqrt(dx**2 + dy**2)
           enddo

           EN = EN + 1
           node = (Nx+1)*(Ny+1)
           enode(EN) = node
           dxM = x(node-1) - x(node)
           dyM = y(node-1) - y(node)
           dxP = -dxM
           dyP = -dyM
           dx = dxP - dxM
           dy = dyP - dyM
           senode(EN) = dy/sqrt(dx**2 + dy**2)
           cenode(EN) = dx/sqrt(dx**2 + dy**2)

           tnode = EN

           do i = 1, Nx
              EF = EF + 1
              eface(EF) = i
              EN = EN + 1
              node = i
              enode(EN) = node
              dxP = x(node+1) - x(node)
              dyP = y(node+1) - y(node)
              if (i .eq. 1) then
                 dxM = -dxP
                 dyM = -dyP
              else
                 dxM = x(node-1) - x(node)
                 dyM = y(node-1) - y(node)
              endif
              if ((dxM*dxP + dyM*dyP) .gt. 0.) then
                 dsM = sqrt(dxM**2 + dyM**2)
                 dsP = sqrt(dxP**2 + dyP**2)
                 dx = dxM/dsM + dxP/dsP
                 dy = dyM/dsM + dyP/dsP
              else
                 dx = dxP - dxM
                 dy = dyP - dyM
              endif
              senode(EN) = dy/sqrt(dx**2 + dy**2)
              cenode(EN) = dx/sqrt(dx**2 + dy**2)
```

```
            enddo

            EN = EN + 1
            node = Nx + 1
            enode(EN) = node
            dxM = x(node-1) - x(node)
            dyM = y(node-1) - y(node)
            dxP = -dxM
            dyP = -dyM
            dx = dxP - dxM
            dy = dyP - dyM
            senode(EN) = dy/sqrt(dx**2 + dy**2)
            cenode(EN) = dx/sqrt(dx**2 + dy**2)

            bnode = EN
            Wmax = 2*Nx
         else
c**   periodic nodes
            do i = 1, ILE
               PN = PN + 1
               pnode(PN,1) = i
               pnode(PN,2) = (Nx+1)*Ny + i
            enddo

            do i = ITE, Nx+1
               PN = PN + 1
               pnode(PN,1) = i
               pnode(PN,2) = (Nx+1)*Ny + i
            enddo
            Pmax = ILE + Nx + 2 - ITE

            do i = 1, ILE-1
               F1 = i
               F2 = Ny*(2*Nx+1) + i
               face(F1,2) = face(F2,2)
               face(F1,7) = face(F2,7)
               face(F1,8) = face(F2,8)
               face(F2,1) = face(F1,1)
               face(F2,5) = face(F1,5)
               face(F2,6) = face(F1,6)
            enddo

            do i = ITE, Nx
               F1 = i
               F2 = Ny*(2*Nx+1) + i
               face(F1,2) = face(F2,2)
               face(F1,7) = face(F2,7)
               face(F1,8) = face(F2,8)
               face(F2,1) = face(F1,1)
               face(F2,5) = face(F1,5)
               face(F2,6) = face(F1,6)
            enddo

c**   the rest of the edge nodes and faces
            do i = ILE, ITE-1
               EN = EN + 1
               node = (Nx+1)*Ny + i
               enode(EN) = node
               dxP = x(node+1) - x(node)
               dyP = y(node+1) - y(node)
               if (i .eq. ILE) then
                  dxM = x(ILE+1) - x(node)
                  dyM = y(ILE+1) + pitch - y(node)
               else
                  dxM = x(node-1) - x(node)
                  dyM = y(node-1) - y(node)
               endif
               if ((dxM*dxP + dyM*dyP) .gt. 0.) then
                  dsM = sqrt(dxM**2 + dyM**2)
                  dsP = sqrt(dxP**2 + dyP**2)
                  dx = dxM/dsM + dxP/dsP
                  dy = dyM/dsM + dyP/dsP
```

```
            else
               dx = dxP - dxM
               dy = dyP - dyM
            endif
            senode(EN) = dy/sqrt(dx**2 + dy**2)
            cenode(EN) = dx/sqrt(dx**2 + dy**2)
            FF = Ny*(2*Nx+1) + 1
            do n = 1, 8
               face(FF-ILE+1,n) = face(FF,n)
            enddo
            EF = EF + 1
            eface(EF) = FF - ILE + 1
         enddo

         do FF = Ny*(2*Nx+1)+Nx+1, Fmax
            CC = face(FF,1)
            cell(CC,1) = FF-Nx+ITE-ILE
            do n = 1, 8
               face(FF-Nx+ITE-ILE,n) = face(FF,n)
            enddo
         enddo

         Fmax = Fmax - Nx + ITE - ILE

         EN = EN + 1
         node = (Nx+1)*Ny + ITE
         enode(EN) = node
         dxM = x(node-1) - x(node)
         dyM = y(node-1) - y(node)
         dxP = x(ITE-1) - x(node)
         dyP = y(ITE-1) + pitch - y(node)
         if ((dxM*dxP + dyM*dyP) .gt. 0.) then
            dsM = sqrt(dxM**2 + dyM**2)
            dsP = sqrt(dxP**2 + dyP**2)
            dx = dxM/dsM + dxP/dsP
            dy = dyM/dsM + dyP/dsP
         else
            dx = dxP - dxM
            dy = dyP - dyM
         endif
         senode(EN) = dy/sqrt(dx**2 + dy**2)
         cenode(EN) = dx/sqrt(dx**2 + dy**2)

         tnode = EN

c**   upper edge of airfoil
         do i = ILE, ITE-1
            EF = EF + 1
            eface(EF) = i
            EN = EN + 1
            node =  i
            enode(EN) = node
            dxP = x(node+1) - x(node)
            dyP = y(node+1) - y(node)
            if (i .eq. ILE) then
               dxM = x((Nx+1)*Ny+ILE+1) - x(node)
               dyM = y((Nx+1)*Ny+ILE+1) - pitch - y(node)
            else
               dxM = x(node-1) - x(node)
               dyM = y(node-1) - y(node)
            endif
            if ((dxM*dxP + dyM*dyP) .gt. 0.) then
               dsM = sqrt(dxM**2 + dyM**2)
               dsP = sqrt(dxP**2 + dyP**2)
               dx = dxM/dsM + dxP/dsP
               dy = dyM/dsM + dyP/dsP
            else
               dx = dxP - dxM
               dy = dyP - dyM
            endif
            senode(EN) = dy/sqrt(dx**2 + dy**2)
            cenode(EN) = dx/sqrt(dx**2 + dy**2)
```

```
            enddo

            EN = EN + 1
            node = ITE
            enode(EN) = node
            dxM = x(node-1) - x(node)
            dyM = y(node-1) - y(node)
            dxP = x((Nx+1)*Ny+ITE-1) - x(node)
            dyP = y((Nx+1)*Ny+ITE-1) - pitch - y(node)
            if ((dxM*dxP + dyM*dyP) .gt. C.) then
               dsM = sqrt(dxM**2 + dyM**2)
               dsP = sqrt(dxP**2 + dyP**2)
               dx = dxM/dsM + dxP/dsP
               dy = dyM/dsM + dyP/dsP
            else
               dx = dxP - dxM
               dy = dyP - dyM
            endif
            senode(EN) = dy/sqrt(dx**2 + dy**2)
            cenode(EN) = dx/sqrt(dx**2 + dy**2)

            bnode = EN
            Wmax = 2*(ITE-ILE)

         endif

c**   inlet and outlet boundaries
         do j = 1, Ny
            EN = EN + 1
            enode(EN) = (j-1)*(Nx+1) + 1
            IN = IN + 1
            innode(IN) = (j-1)*(Nx+1) + 1
            EF = EF + 1
            eface(EF) = (2*Nx+1)*j - Nx
         enddo

         EN = EN + 1
         enode(EN) = Ny*(Nx+1) + 1
         IN = IN + 1
         innode(IN) = Ny*(Nx+1) + 1

         inode = bnode + Ny + 1
         Imax = IN

         do j = 1, Ny
            EN = EN + 1
            enode(EN) = (Nx+1)*j
            ON = ON + 1
            outnode(ON) = (Nx+1)*j
            EF = EF + 1
            eface(EF) = (2*Nx+1)*j
         enddo

         EN = EN + 1
         enode(EN) = (Nx+1)*(Ny+1)
         ON = ON + 1
         outnode(ON) = (Nx+1)*(Ny+1)

         ENmax = EN
         Emax = EF
         Omax = ON
         if ((Emax .gt. Maxedges) .or.
     &       (ENmax .gt. Maxedges) .or.
     &       (Pmax .gt. Maxedges)) then
            write(6,*) 'EF = ', EF
            write(6,*) 'EN = ', EN
            write(6,*) 'PN = ', PN
            write(6,*) 'more than Maxedges required'
         endif

         return
         end
```

```
c*********************************************************************
c*                                                                   *
c*      renumber edges so nodes are consecutive along edges          *
c*      and N3 is the interior node                                  *
c*                                                                   *
c*********************************************************************

        subroutine edgenumber
        implicit none

        include 'QUAD.INC'
        integer EF, FF                  !pointers
        integer save                    !save value in face

        do EF = 1, Emax
           FF = eface(EF)
           if (face(FF,1).eq.0) then
              save = face(FF,1)
              face(FF,1) = face(FF,2)
              face(FF,2) = save

              save = face(FF,3)
              face(FF,3) = face(FF,4)
              face(FF,4) = save

              save = face(FF,5)
              face(FF,5) = face(FF,7)
              face(FF,7) = save

              save = face(FF,6)
              face(FF,6) = face(FF,8)
              face(FF,8) = save
           endif
        enddo

        return
        end




        subroutine cellcolor
        implicit none

        include 'QUAD.INC'
        integer CC, NN, FF              !pointers
        integer Ncells                  !number of cells colored
        integer CC2(0:Maxcells)         !new number of cell
        integer Ncolor                  !current color
        integer cell2(Maxcells,8)       !new cell number
        logical hascolor(Maxcells)      !has node been colored current color?

c**  Initialize everything
        Ncells = 0
        do CC = 1, Cmax
           CC2(CC) = 0
        enddo

c**  Loop over colors
        do Ncolor = 1, 50

c**  Initialize node color array
           NCcolor(Ncolor) = 0
           do NN = 1, Nmax
              hascolor(NN) = .FALSE.
           enddo
```

```
c**  Loop over cells
          do CC = 1, Cmax

c**  Check if cell is already colored with old color
               if(CC2(CC).NE.0) GOTO 22
c**  Check if cell nodes are already colored with new color
               if( hascolor(cell(CC,5)) .OR.
     &             hascolor(cell(CC,6)) .OR.
     &             hascolor(cell(CC,7)) .OR.
     &             hascolor(cell(CC,8)) ) GOTO 22

c**  Set color markers
               Ncells = Ncells + 1
               CC2(CC) = Ncells
               NCcolor(Ncolor) = NCcolor(Ncolor) + 1
               hascolor(cell(CC,5)) = .TRUE.
               hascolor(cell(CC,6)) = .TRUE.
               hascolor(cell(CC,7)) = .TRUE.
               hascolor(cell(CC,8)) = .TRUE.

 22         enddo

            if(Ncells.EQ.Cmax) GOTO 23

          enddo

          STOP 'COLOR; more than 50 colors required for cells'

 23       Ccolormax= Ncolor

c**  Redo pointers
          do CC = 1, Cmax
             cell2(CC2(CC),1) = cell(CC,1)
             cell2(CC2(CC),2) = cell(CC,2)
             cell2(CC2(CC),3) = cell(CC,3)
             cell2(CC2(CC),4) = cell(CC,4)
             cell2(CC2(CC),5) = cell(CC,5)
             cell2(CC2(CC),6) = cell(CC,6)
             cell2(CC2(CC),7) = cell(CC,7)
             cell2(CC2(CC),8) = cell(CC,8)
          enddo

          do CC = 1, Cmax
             cell(CC,1) = cell2(CC,1)
             cell(CC,2) = cell2(CC,2)
             cell(CC,3) = cell2(CC,3)
             cell(CC,4) = cell2(CC,4)
             cell(CC,5) = cell2(CC,5)
             cell(CC,6) = cell2(CC,6)
             cell(CC,7) = cell2(CC,7)
             cell(CC,8) = cell2(CC,8)
          enddo

          CC2(0) = 0

          do FF = 1, Fmax
             face(FF,1) = CC2(face(FF,1))
             face(FF,2) = CC2(face(FF,2))
          enddo

          return
          end




          subroutine facecolor
          implicit none
```

```
      include 'QUAD.INC'
      integer FF, NN, CC, EF          !pointers
      integer Nfaces                  !number of faces colored
      integer FF2(Maxfaces)           !new number of face
      integer Ncolor                  !current color
      integer face2(Maxfaces,8)       !new face number
      logical hascolor(0:Maxfaces)    !has node been colored current color?

c**   Initialize everything
      Nfaces = 0
      do FF = 1, Fmax
         FF2(FF) = 0
      enddo

c**   Loop over colors
      do Ncolor = 1, 50

c**   Initialize node color array
         NFcolor(Ncolor) = 0
         do NN = 1, Nmax
            hascolor(NN) = .FALSE.
         enddo

c**   Loop over faces
         do FF = 1, Fmax

            hascolor(0) = .FALSE.

c**   Check if face is already colored with old color
            if(FF2(FF).NE.0) GOTO 22

c**   Check if face nodes are already colored with new color
            if( hascolor(face(FF,5)) .OR.
     &          hascolor(face(FF,6)) .OR.
     &          hascolor(face(FF,7)) .OR.
     &          hascolor(face(FF,8)) ) GOTO 22

c**   Set color markers
            Nfaces = Nfaces + 1
            FF2(FF) = Nfaces
            NFcolor(Ncolor) = NFcolor(Ncolor) + 1
            hascolor(face(FF,5)) = .TRUE.
            hascolor(face(FF,6)) = .TRUE.
            hascolor(face(FF,7)) = .TRUE.
            hascolor(face(FF,8)) = .TRUE.

 22       enddo

          if(Nfaces.EQ.Fmax) GOTO 23

       enddo

       STOP 'COLOR; more than 50 colors required for faces'

 23    Fcolormax= Ncolor

c**   Redo pointers
       do FF = 1, Fmax
          face2(FF2(FF),1) = face(FF,1)
          face2(FF2(FF),2) = face(FF,2)
          face2(FF2(FF),3) = face(FF,3)
          face2(FF2(FF),4) = face(FF,4)
          face2(FF2(FF),5) = face(FF,5)
          face2(FF2(FF),6) = face(FF,6)
          face2(FF2(FF),7) = face(FF,7)
          face2(FF2(FF),8) = face(FF,8)
       enddo

       do FF = 1, Fmax
          face(FF,1) = face2(FF,1)
          face(FF,2) = face2(FF,2)
          face(FF,3) = face2(FF,3)
```

```
            face(FF,4) = face2(FF,4)
            face(FF,5) = face2(FF,5)
            face(FF,6) = face2(FF,6)
            face(FF,7) = face2(FF,7)
            face(FF,8) = face2(FF,8)
         enddo

         do CC = 1, Cmax
            cell(CC,1) = FF2(cell(CC,1))
            cell(CC,2) = FF2(cell(CC,2))
            cell(CC,3) = FF2(cell(CC,3))
            cell(CC,4) = FF2(cell(CC,4))
         enddo

         do EF = 1, Emax
            eface(EF) = FF2(eface(EF))
         enddo

         return
         end



         subroutine edgecolor
         implicit none

         include 'QUAD.INC'
         integer FF, NN, EF              !pointers
         integer Nedges                  !number of faces colored
         integer EF2(Maxedges)           !new number of face
         integer Ncolor                  !current color
         integer eface2(Maxedges)        !new face number
         logical hascolor(Maxnodes)      !has node been colored current color?

c**   Initialize everything
         Nedges = 0
         do EF = 1, Emax
            EF2(EF) = 0
         enddo

c**   Loop over colors
         do Ncolor = 1, 5

c**   Initialize node color array
            NEcolor(Ncolor) = 0
            do NN = 1, Nmax
               hascolor(NN) = .FALSE.
            enddo

c**   Loop over faces
            do EF = 1, Wmax
               FF = eface(EF)

c**   Check if face is already colored with old color
               if(EF2(EF).NE.0) GOTO 22

c**   Check if face nodes are already colored with new color
               if( hascolor(face(FF,3)) .OR.
     &             hascolor(face(FF,4)) .OR.
     &             hascolor(face(FF,5)) .OR.
     &             hascolor(face(FF,6)) ) GOTO 22

c**   Set color markers
               Nedges = Nedges + 1
               EF2(EF) = Nedges
               NEcolor(Ncolor) = NEcolor(Ncolor) + 1
               hascolor(face(FF,3)) = .TRUE.
               hascolor(face(FF,4)) = .TRUE.
               hascolor(face(FF,5)) = .TRUE.
               hascolor(face(FF,6)) = .TRUE.
```

```
 22        enddo

           if(Nedges.EQ.Wmax) GOTO 23

         enddo

         STOP 'COLOR; more than 5 colors required for wall edges'

 23      E1colormax= Ncolor

c**   Loop over colors
        do Ncolor = E1colormax+1, 10

c**   Initialize node color array
           NEcolor(Ncolor) = 0
           do NN = 1, Nmax
              hascolor(NN) = .FALSE.
           enddo

c**   Loop over faces
           do EF = Wmax+1, Emax
              FF = eface(EF)

c**   Check if face is already colored with old color
              if(EF2(EF).NE.0) GOTO 32

c**   Check if face nodes are already colored with new color
              if( hascolor(face(FF,3)) .OR.
     &            hascolor(face(FF,4)) .OR.
     &            hascolor(face(FF,5)) .OR.
     &            hascolor(face(FF,6)) ) GOTO 32

c**   Set color markers
              Nedges = Nedges + 1
              EF2(EF) = Nedges
              NEcolor(Ncolor) = NEcolor(Ncolor) + 1
              hascolor(face(FF,3)) = .TRUE.
              hascolor(face(FF,4)) = .TRUE.
              hascolor(face(FF,5)) = .TRUE.
              hascolor(face(FF,6)) = .TRUE.

 32        enddo

           if(Nedges.EQ.Emax) GOTO 33

         enddo

         STOP 'COLOR; more than 10 colors required for all edges'

 33      E2colormax= Ncolor

c**   Redo pointers
        do EF = 1, Emax
           eface2(EF2(EF)) = eface(EF)
        enddo

        do EF = 1, Emax
           eface(EF) = eface2(EF)
        enddo

        return
        end
```

```
c*****************************************************************
```

```
c*                                                                     *
c*      program to create an irregular quadrilateral mesh              *
c*      from a regular quadrilateral mesh                              *
c*                                                                     *
c**********************************************************************

        program griddis
        implicit none

        include 'QUAD.INC'
        integer NN, EN                  !pointers
        real CC, AA
        integer N1, N2
        real change
        real yc, RR2
        real tau
        real pi

        pi = 3.14159

        write(6,1)
 1      format($,' tau = ')
        accept 11, tau
 11     format(F)

c       tau = 0.1

        call gridio(1)

        N1 = face(1,3)
        N2 = face(1,4)
        CC = 0.3*(x(N1) - x(N2))
        AA = (0.3*30.)/CC
        do NN = 1, Nmax
           if ((x(NN).ne.-1.) .and. (x(NN).ne.2.)) then
              change = CC*cos(AA*y(NN)+12.)*cos(AA*x(NN)+43.)
              x(NN) = x(NN) + change
           endif
           if ((y(NN).ne.0.) .and. (y(NN).ne.1.)) then
              change = CC*cos(AA*y(NN)+57.)*cos(AA*x(NN)+33.)
              y(NN) = y(NN) + change
           endif
        enddo

        do EN = tnode, bnode
           NN = enode(EN)
           if (x(NN).gt.0. .and. x(NN).lt.1.) then
              if (tau.lt.0) then
                 y(NN) = tau*(sin(pi*x(NN)))**2
              else if (tau.gt.0) then
                 yc = (tau**2 - 0.25)/(2.*tau)
                 RR2 = 0.25 + yc**2
                 y(NN) = yc + sqrt(RR2 - (x(NN)-0.5)**2)
              endif
           endif
        enddo

        call gridio(0)

        stop
        end
```

A.2.3 Ni Scheme

```
c**********************************************************************
```

```
c*                                                                     *
c*      main program for quadrilateral Ni scheme                       *
c*                                                                     *
c**********************************************************************

        program quadrilateral
        implicit none

        include 'QUAD.INC'
        integer Niter           !number of iterations
        integer NN, CC, i       !pointer
        integer N1, N2, N3, N4  !nodes at corners of cell
        real maxchange          !max change in state vector
        integer maxnode, maxeqn !where max change occurs

c*  read in data from file
        call gridio(1)
        call flowio(1)
        call input

c*  calculate control area around each cell
        call calcarea

        Niter = 0
        durms = 999.

c*  start history file from the top
        open(unit=35,status='unknown',form='formatted')
        write(35,2) Minl
        close(unit=35)
 2      format(' inlet Mach number = ',f5.3)

c*  loop until converged
        do while ((Niter.lt.Maxiter) .and. (durms.gt.2.e-7))
           Niter = Niter + 1

c*  set cell values of state vector
           do CC = 1, Cmax
              N1 = cell(CC,5)
              N2 = cell(CC,6)
              N3 = cell(CC,7)
              N4 = cell(CC,8)
              Uc(1,CC) = (U(1,N1) + U(1,N2) + U(1,N3) + U(1,N4))*0.25
              Uc(2,CC) = (U(2,N1) + U(2,N2) + U(2,N3) + U(2,N4))*0.25
              Uc(3,CC) = (U(3,N1) + U(3,N2) + U(3,N3) + U(3,N4))*0.25
              Uc(4,CC) = (U(4,N1) + U(4,N2) + U(4,N3) + U(4,N4))*0.25
           enddo

c*  calculate time step for each cell
           call timestep

c*  calculate flux at each node
           call nodeflux

c*  calculate change in state vector and fluxes at each cell
           call delcell

c*  calculate change in state vector at each node
           call delstate

c*  add smoothing term
           if (eps2 .ne. 0.) call smooth

c**  account for periodic nodes
           call bperiodic

c*  set inlet and outlet boundary conditions
           call binlet
           call boutlet

c*  change momentum change to make flow tangent at walls
           call tangent
```

```
c*  update state vector
          do NN = 1, Nmax
             U(1,NN) = U(1,NN) + dU(1,NN)
             U(2,NN) = U(2,NN) + dU(2,NN)
             U(3,NN) = U(3,NN) + dU(3,NN)
             U(4,NN) = U(4,NN) + dU(4,NN)
          enddo

c*  find root mean square difference in state vector
          if (mod(Niter,10).eq.0 .or. Niter.lt.10) then
             durms = 0.0
             maxchange = 0.

             do i = 1, 4
                do NN = 1, Nmax
                   durms = durms + dU(i,NN)**2
                   if (abs(dU(i,NN)).gt.abs(maxchange)) then
                      maxchange = dU(i,NN)
                      maxnode = NN
                      maxeqn = i
                   endif
                enddo
             enddo

             durms = sqrt(durms/(4.*Nmax))

c*  print diagnostics to screen and file
             call flowio(0)
             open(unit=50, status='unknown', form='unformatted')
             write(50) Cmax, (vol(CC),CC=1,Cmax)
             close(unit=50)

 10          open(unit=35, status='old',access='append',err=10)
             write(35,1) Niter, durms, maxchange, x(maxnode),
     &                   y(maxnode), maxeqn
             close(unit=35)
             write(6,1) Niter, durms, maxchange, x(maxnode),
     &                   y(maxnode), maxeqn

          endif

       enddo

 1     format('Niter=',i4,'  rms=',f9.7,'  max=',f9.7,'  x=',f6.3,
     &                  '  y=',f6.3,'  eqn=',i1)

c*  write out data to file
       call flowio(0)

       stop
       end




c*********************************************************************
c*                                                                   *
c*      calculate areas of cells                                     *
c*                                                                   *
c*********************************************************************

       subroutine calcarea
       implicit none

       include 'QUAD.INC'
       integer EF, FF, NN, CC, PN      !pointer
       integer N1, N2, N3, N4          !nodes at corner of cell
       integer C1                      !cell on boundary
       integer P1, P2                  !periodic nodes
       real AA                         !area of cell
```

```
      integer CC1, CC2, col          !coloring pointers
      integer EF1, EF2

c* calculate area of cells and distribute to nodes
      do CC = 1, Cmax
         N1 = cell(CC,5)
         N2 = cell(CC,6)
         N3 = cell(CC,7)
         N4 = cell(CC,8)
         AA = -0.5*((x(N2) - x(N4))*(y(N1) - y(N3))
     &             - (x(N1) - x(N3))*(y(N2) - y(N4)))
         areaC(CC) = AA
      enddo

      return
      end



c*********************************************************************
c*                                                                   *
c*      calculate time step for nodes and cells                      *
c*                                                                   *
c*********************************************************************

      subroutine timestep
      implicit none

      include 'QUAD.INC'
      integer CC, NN, PN             !pointers
      integer P1, P2                 !periodic nodes
      integer N1, N2, N3, N4         !nodes around cell
      real dxl, dyl, dl
      real dxm, dym, dm
      real rr, uu, vv, pp, aa        !values for cell
      real time1, time2

      do NN = 1, Nmax
         deltN(NN) = 0.
      enddo

c*  find time step for each cell
      do CC = 1, Cmax
         N1 = cell(CC,5)
         N2 = cell(CC,6)
         N3 = cell(CC,7)
         N4 = cell(CC,8)
         dxl = .5*(x(N3) + x(N2) - x(N4) - x(N1))
         dyl = .5*(y(N3) + y(N2) - y(N4) - y(N1))
         dl  = sqrt(dxl**2 + dyl**2)
         dxm = .5*(x(N4) + x(N3) - x(N1) - x(N2))
         dym = .5*(y(N4) + y(N3) - y(N1) - y(N2))
         dm  = sqrt(dxm**2 + dym**2)
         rr = Uc(1,CC)
         uu = Uc(2,CC)/rr
         vv = Uc(3,CC)/rr
         pp = gam1*(Uc(4,CC) - 0.5*rr*(uu**2 + vv**2))
         aa = sqrt(gam*pp/rr)

         time1 = 1./(abs(uu*dyl-vv*dxl) + aa*dl)
         time2 = 1./(abs(uu*dym-vv*dxm) + aa*dm)

         deltC(CC) = CFL*min(time1,time2)

         deltN(N1) = deltN(N1) + 0.25/deltC(CC)
         deltN(N2) = deltN(N2) + 0.25/deltC(CC)
         deltN(N3) = deltN(N3) + 0.25/deltC(CC)
         deltN(N4) = deltN(N4) + 0.25/deltC(CC)
      enddo
```

```
c**  periodic nodes
       do PN = 1, Pmax
          P1 = pnode(PN,1)
          P2 = pnode(PN,2)
          deltN(P1) = deltN(P1) + deltN(P2)
          deltN(P2) = deltN(P1)
       enddo

       do NN = 1, Nmax
          deltN(NN) = 1./deltN(NN)
       enddo

       return
       end




c*********************************************************************
c*                                                                   *
c*      calculate flux vector values at nodes                        *
c*                                                                   *
c*********************************************************************

       subroutine nodeflux
       implicit none

       include 'QUAD.INC'
       integer NN                        !pointers
       real WW                           !kinetic energy

       do NN = 1, Nmax
c*  calculate f and g at nodes
          WW = 0.5*(U(2,NN)**2 + U(3,NN)**2)/U(1,NN)
          F(1,NN) = U(2,NN)
          F(2,NN) = U(2,NN)**2/U(1,NN) + gam1*(U(4,NN) - WW)
          F(3,NN) = U(2,NN)*U(3,NN)/U(1,NN)
          F(4,NN) = (U(2,NN)/U(1,NN))*(gam*U(4,NN) - gam1*WW)

          G(1,NN) = U(3,NN)
          G(2,NN) = U(2,NN)*U(3,NN)/U(1,NN)
          G(3,NN) = U(3,NN)**2/U(1,NN) + gam1*(U(4,NN) - WW)
          G(4,NN) = (U(3,NN)/U(1,NN))*(gam*U(4,NN) - gam1*WW)
       enddo

       return
       end



c*********************************************************************
c*                                                                   *
c*      this subroutine finds the change in the fluxes               *
c*      F and G and the change in the state U at each cell           *
c*                                                                   *
c*********************************************************************

       subroutine delcell
       implicit none

       include 'QUAD.INC'
       integer CC                        !pointers
       integer N1, N2, N3, N4            !nodes at corners of cells
       real coef
       real dy31, dy24, dx31, dx24       !change in x and y
       real dF24, dF13, dG24, dG13       !change in F and G
       real u1, u2, u3, u4               !velocity at nodes
       real v1, v2, v3, v4
```

```
c*  find change in state vector for cell
       do CC = 1, Cmax
          N1 = cell(CC,5)
          N2 = cell(CC,6)
          N3 = cell(CC,7)
          N4 = cell(CC,8)

          u1 = U(2,N1)/U(1,N1)
          u2 = U(2,N2)/U(1,N2)
          u3 = U(2,N3)/U(1,N3)
          u4 = U(2,N4)/U(1,N4)
          v1 = U(3,N1)/U(1,N1)
          v2 = U(3,N2)/U(1,N2)
          v3 = U(3,N3)/U(1,N3)
          v4 = U(3,N4)/U(1,N4)

          dy31 = y(N3) - y(N1)
          dy24 = y(N2) - y(N4)
          dx31 = x(N3) - x(N1)
          dx24 = x(N2) - x(N4)

          coef = 0.5*delC(CC)

          dF24 = F(1,N2) - F(1,N4)
          dF13 = F(1,N1) - F(1,N3)
          dG24 = G(1,N2) - G(1,N4)
          dG13 = G(1,N1) - G(1,N3)
          dUc(1,CC) = coef*(-dF24*dy31 + dG24*dx31 -
     &                      dF13*dy24 + dG13*dx24)

          dF24 = F(2,N2) - F(2,N4)
          dF13 = F(2,N1) - F(2,N3)
          dG24 = G(2,N2) - G(2,N4)
          dG13 = G(2,N1) - G(2,N3)
          dUc(2,CC) = coef*(-dF24*dy31 + dG24*dx31 -
     &                      dF13*dy24 + dG13*dx24)

          dF24 = F(3,N2) - F(3,N4)
          dF13 = F(3,N1) - F(3,N3)
          dG24 = G(3,N2) - G(3,N4)
          dG13 = G(3,N1) - G(3,N3)
          dUc(3,CC) = coef*(-dF24*dy31 + dG24*dx31 -
     &                      dF13*dy24 + dG13*dx24)

          dF24 = F(4,N2) - F(4,N4)
          dF13 = F(4,N1) - F(4,N3)
          dG24 = G(4,N2) - G(4,N4)
          dG13 = G(4,N1) - G(4,N3)
          dUc(4,CC) = coef*(-dF24*dy31 + dG24*dx31 -
     &                      dF13*dy24 + dG13*dx24)

c**  shock smoothing
          vol(CC) = (u1-u3)*dy24 - (v1-v3)*dx24 +
     &              (u2-u4)*dy31 - (v2-v4)*dx31
       enddo

c*  implement wall boundary conditions
       call bwall

       return
       end
```

```
c*********************************************************************
c*                                                                   *
c*      this subroutine applies the wall boundary condition          *
c*                                                                   *
c*********************************************************************
```

```
      subroutine bwall
      implicit none

      include 'QUAD.INC'
      integer EF, FF, CC        !pointer
      integer N1, N2            !nodes on edge
      real coef
      real dy12, dx12           !change in x and y
      real pres1, pres2         !pressure
      real u1, u2, v1, v2       !velocity
      integer EF1, EF2, col     !coloring pointers

c*  upper and lower boundary
      EF2 = 0

      do col = 1, E1colormax
       EF1 = EF2 + 1
       EF2 = EF1 - 1 + NEcolor(col)
       do EF = EF1, EF2

         FF = eface(EF)
         CC = face(FF,1)
         N1 = face(FF,3)
         N2 = face(FF,4)

         coef = 0.5*deltC(CC)

         dy12 = y(N1) - y(N2)
         dx12 = x(N1) - x(N2)

         pres1 = gam1*(U(4,N1) - 0.5*(U(2,N1)**2 + U(3,N1)**2)/
     &         U(1,N1))
         pres2 = gam1*(U(4,N2) - 0.5*(U(2,N2)**2 + U(3,N2)**2)/
     &         U(1,N2))
         u1 = U(2,N1)/U(1,N1)
         u2 = U(2,N2)/U(1,N2)
         v1 = U(3,N1)/U(1,N1)
         v2 = U(3,N2)/U(1,N2)

         dUc(1,CC) = dUc(1,CC) + coef*(
     &                -(F(1,N2) + F(1,N1))*dy12 +
     &                (G(1,N2) + G(1,N1))*dx12)

         dUc(2,CC) = dUc(2,CC) + coef*(
     &                -(F(2,N2) + F(2,N1) - pres1 - pres2)*dy12 +
     &                (G(2,N2) + G(2,N1))*dx12)

         dUc(3,CC) = dUc(3,CC) + coef*(
     &                -(F(3,N2) + F(3,N1))*dy12 +
     &                (G(3,N2) + G(3,N1) - pres1 - pres2)*dx12)

         dUc(4,CC) = dUc(4,CC) + coef*(
     &                -(F(4,N2) + F(4,N1))*dy12 +
     &                (G(4,N2) + G(4,N1))*dx12)

c**  shock smoothing
         vol(CC) = vol(CC) + (u1+u2)*dy12 - (v1+v2)*dx12

       enddo
      enddo

      return
      end




c*****************************************************************
c*                                                               *
c*      this subroutine calculates the change at each node       *
c*                                                               *
```

```
c*********************************************************************

        subroutine delstate
        implicit none

        include 'QUAD.INC'
        integer NN, CC, EF, FF          !pointers
        integer N1, N2, N3, N4          !nodes at corners of cell
        real rr, uu, vv, pp, aa, vv2    !values at cell
        real rdv, rdu, dp, HH           !changes
        real CON
        real dy42, dy13                 !change in y on cell edges
        real dx42, dx13                 !change in x on cell edges
        real dy21, dx21                 !  "          "       "
        real coef                       !coefficient
        integer CC1, CC2, col           !color pointers
        integer EF1, EF2                !color pointers
        real dFc(4), dGc(4)

        do NN = 1, Nmax
           dU(1,NN) = 0.
           dU(2,NN) = 0.
           dU(3,NN) = 0.
           dU(4,NN) = 0.
        enddo

        CC2 = 0

        do col = 1, Ccolormax
         CC1 = CC2 + 1
         CC2 = CC1 - 1 + NCcolor(col)

CVD$    NODEPCHK
         do CC = CC1, CC2
           N1 = cell(CC,5)
           N2 = cell(CC,6)
           N3 = cell(CC,7)
           N4 = cell(CC,8)

           rr = Uc(1,CC)
           uu = Uc(2,CC)/rr
           vv = Uc(3,CC)/rr
           vv2 = 0.5*(uu*uu + vv*vv)

           dy42 = y(N4) - y(N2)
           dy13 = y(N1) - y(N3)
           dx42 = x(N4) - x(N2)
           dx13 = x(N1) - x(N3)

c*  find second order change in flux vector at cells
           HH = gam*Uc(4,CC)/Uc(1,CC) - gam1*vv2
           rdu = dUc(2,CC) - uu*dUc(1,CC)
           rdv = dUc(3,CC) - vv*dUc(1,CC)
           dp = gam1*(dUc(4,CC) - uu*dUc(2,CC) - vv*dUc(3,CC)
     &             + vv2*dUc(1,CC))

           dFc(1) = dUc(2,CC)
           dFc(2) = uu*(dUc(2,CC) + rdu) + dp
           dFc(3) = vv*dUc(2,CC) + uu*rdv
           dFc(4) = uu*(dUc(4,CC) + dp) + HH*rdu

           dGc(1) = dUc(3,CC)
           dGc(2) = uu*dUc(3,CC) + vv*rdu
           dGc(3) = vv*(dUc(3,CC) + rdv) + dp
           dGc(4) = vv*(dUc(4,CC) + dp) + HH*rdv

c**  shock smoothing
           vol(CC) = vol(CC)/sqrt(2.*areaC(CC))
           CON = -epsicoef*min(0.5,abs(vol(CC)))*vol(CC)*vv2
           dFc(2) = dFc(2) + CON
           dGc(3) = dGc(3) + CON
```

```
      coef = 0.25/deltC(CC)

      dU(1,N1) = dU(1,N1) + coef*dUc(1,CC) - 0.25*
     &                    (dFc(1)*dy42 - dGc(1)*dx42)
      dU(1,N2) = dU(1,N2) + coef*dUc(1,CC)  - 0.25*
     &                    (dFc(1)*dy13 - dGc(1)*dx13)
      dU(1,N3) = dU(1,N3) + coef*dUc(1,CC) + 0.25*
     &                    (dFc(1)*dy42 - dGc(1)*dx42)
      dU(1,N4) = dU(1,N4) + coef*dUc(1,CC) + 0.25*
     &                    (dFc(1)*dy13 - dGc(1)*dx13)

      dU(2,N1) = dU(2,N1) + coef*dUc(2,CC) - 0.25*
     &                    (dFc(2)*dy42 - dGc(2)*dx42)
      dU(2,N2) = dU(2,N2) + coef*dUc(2,CC) - 0.25*
     &                    (dFc(2)*dy13 - dGc(2)*dx13)
      dU(2,N3) = dU(2,N3) + coef*dUc(2,CC) + 0.25*
     &                    (dFc(2)*dy42 - dGc(2)*dx42)
      dU(2,N4) = dU(2,N4) + coef*dUc(2,CC) + 0.25*
     &                    (dFc(2)*dy13 - dGc(2)*dx13)

      dU(3,N1) = dU(3,N1) + coef*dUc(3,CC) - 0.25*
     &                    (dFc(3)*dy42 - dGc(3)*dx42)
      dU(3,N2) = dU(3,N2) + coef*dUc(3,CC) - 0.25*
     &                    (dFc(3)*dy13 - dGc(3)*dx13)
      dU(3,N3) = dU(3,N3) + coef*dUc(3,CC) + 0.25*
     &                    (dFc(3)*dy42 - dGc(3)*dx42)
      dU(3,N4) = dU(3,N4) + coef*dUc(3,CC) + 0.25*
     &                    (dFc(3)*dy13 - dGc(3)*dx13)

      dU(4,N1) = dU(4,N1) + coef*dUc(4,CC) - 0.25*
     &                    (dFc(4)*dy42 - dGc(4)*dx42)
      dU(4,N2) = dU(4,N2) + coef*dUc(4,CC) - 0.25*
     &                    (dFc(4)*dy13 - dGc(4)*dx13)
      dU(4,N3) = dU(4,N3) + coef*dUc(4,CC) + 0.25*
     &                    (dFc(4)*dy42 - dGc(4)*dx42)
      dU(4,N4) = dU(4,N4) + coef*dUc(4,CC) + 0.25*
     &                    (dFc(4)*dy13 - dGc(4)*dx13)
    enddo
   enddo

   EF2 = 0

   do col = 1, E2colormax
    EF1 = EF2 + 1
    EF2 = EF1 - 1 + NEcolor(col)
    do EF = EF1, EF2

      FF = eface(EF)
      CC = face(FF,1)
      N1 = face(FF,3)
      N2 = face(FF,4)

      dy21 = y(N2) - y(N1)
      dx21 = x(N2) - x(N1)

      uu = Uc(2,CC)/Uc(1,CC)
      vv = Uc(3,CC)/Uc(1,CC)
      ww2 = 0.5*(uu**2 + vv**2)

      dp = gam1*(dUc(4,CC) - uu*dUc(2,CC) - vv*dUc(3,CC)
     &          + ww2*dUc(1,CC))

      dFc(2) = dp
      dGc(3) = dp

      dU(2,N1) = dU(2,N1) - 0.25*dFc(2)*dy21
      dU(2,N2) = dU(2,N2) - 0.25*dFc(2)*dy21

      dU(3,N1) = dU(3,N1) + 0.25*dGc(3)*dx21
      dU(3,N2) = dU(3,N2) + 0.25*dGc(3)*dx21

    enddo
```

```
      enddo

      do NN = 1, Nmax
         dU(1,NN) = dU(1,NN)*deltN(NN)
         dU(2,NN) = dU(2,NN)*deltN(NN)
         dU(3,NN) = dU(3,NN)*deltN(NN)
         dU(4,NN) = dU(4,NN)*deltN(NN)
      enddo

      return
      end



c*********************************************************************
c*                                                                   *
c*      subroutine to calculate forth difference smoothing           *
c*                                                                   *
c*********************************************************************

      subroutine smooth
      implicit none

      include 'QUAD.INC'
      integer N1, N2, N3, N4          !nodes at corners of cell
      integer NN, CC1, CC2, col, CC   !pointers
      integer FF, EF, EF1, EF2        !edge nodes
      integer P1, P2, PN              !periodic nodes
      real dx31, dx24                 !change in x
      real dx12, dx23, dx34, dx41
      real dy31, dy24                 !change in y
      real dy12, dy23, dy34, dy41
      real dxC1, dxC2, dxC3, dxC4     !change in x in cell
      real dyC1, dyC2, dyC3, dyC4     !change in y in cell
      real del12, del13, del14
      real del23, del24, del34
      real coef12, coef13, coef14
      real coef23, coef24, coef34
      real del2(4,Maxnodes)           !gradient at node
      real coef                       !coefficient

      do NN = 1, Nmax
         del2(1,NN) = 0.
         del2(2,NN) = 0.
         del2(3,NN) = 0.
         del2(4,NN) = 0.
      enddo

      CC2 = 0

      do col = 1, Ccolormax
       CC1 = CC2 + 1
       CC2 = CC1 - 1 + NCcolor(col)

CVD$  NODEPCHK
       do CC = CC1, CC2
         N1 = cell(CC,5)
         N2 = cell(CC,6)
         N3 = cell(CC,7)
         N4 = cell(CC,8)

         dy24 = y(N2) - y(N4)
         dy31 = y(N3) - y(N1)
         dy12 = y(N1) - y(N2)
         dy23 = y(N2) - y(N3)
         dy34 = y(N3) - y(N4)
         dy41 = y(N4) - y(N1)

         dx24 = x(N2) - x(N4)
         dx31 = x(N3) - x(N1)
         dx12 = x(N1) - x(N2)
```

```
      dx23 = x(N2) - x(N3)
      dx34 = x(N3) - x(N4)
      dx41 = x(N4) - x(N1)

      coef = ((max(0.,min(1.,(1.+10.*vol(CC))))-1.)*epsicoef +1.)
     &                *(0.5/abs(-dx12*dy41 + dy12*dx41))

      dxC1 = U(1,N1)*dy24 + U(1,N2)*dy41 + U(1,N4)*dy12
      dxC2 = U(2,N1)*dy24 + U(2,N2)*dy41 + U(2,N4)*dy12
      dxC3 = U(3,N1)*dy24 + U(3,N2)*dy41 + U(3,N4)*dy12
      dxC4 = U(4,N1)*dy24 + U(4,N2)*dy41 + U(4,N4)*dy12

      dyC1 = U(1,N1)*dx24 + U(1,N2)*dx41 + U(1,N4)*dx12
      dyC2 = U(2,N1)*dx24 + U(2,N2)*dx41 + U(2,N4)*dx12
      dyC3 = U(3,N1)*dx24 + U(3,N2)*dx41 + U(3,N4)*dx12
      dyC4 = U(4,N1)*dx24 + U(4,N2)*dx41 + U(4,N4)*dx12

      del2(1,N1) = del2(1,N1) + coef*(dxC1*dy24 + dyC1*dx24)
      del2(2,N1) = del2(2,N1) + coef*(dxC2*dy24 + dyC2*dx24)
      del2(3,N1) = del2(3,N1) + coef*(dxC3*dy24 + dyC3*dx24)
      del2(4,N1) = del2(4,N1) + coef*(dxC4*dy24 + dyC4*dx24)

      coef = ((max(0.,min(1.,(1.+10.*vol(CC))))-1.)*epsicoef +1.)
     &                *(0.5/abs(-dx12*dy23 + dy12*dx23))

      dxC1 = U(1,N1)*dy23 + U(1,N2)*dy31 + U(1,N3)*dy12
      dxC2 = U(2,N1)*dy23 + U(2,N2)*dy31 + U(2,N3)*dy12
      dxC3 = U(3,N1)*dy23 + U(3,N2)*dy31 + U(3,N3)*dy12
      dxC4 = U(4,N1)*dy23 + U(4,N2)*dy31 + U(4,N3)*dy12

      dyC1 = U(1,N1)*dx23 + U(1,N2)*dx31 + U(1,N3)*dx12
      dyC2 = U(2,N1)*dx23 + U(2,N2)*dx31 + U(2,N3)*dx12
      dyC3 = U(3,N1)*dx23 + U(3,N2)*dx31 + U(3,N3)*dx12
      dyC4 = U(4,N1)*dx23 + U(4,N2)*dx31 + U(4,N3)*dx12

      del2(1,N2) = del2(1,N2) + coef*(dxC1*dy31 + dyC1*dx31)
      del2(2,N2) = del2(2,N2) + coef*(dxC2*dy31 + dyC2*dx31)
      del2(3,N2) = del2(3,N2) + coef*(dxC3*dy31 + dyC3*dx31)
      del2(4,N2) = del2(4,N2) + coef*(dxC4*dy31 + dyC4*dx31)

      coef = ((max(0.,min(1.,(1.+10.*vol(CC))))-1.)*epsicoef +1.)
     &                *(0.5/abs(-dx23*dy34 + dy23*dx34))

      dxC1 = U(1,N4)*dy23 + U(1,N2)*dy34 - U(1,N3)*dy24
      dxC2 = U(2,N4)*dy23 + U(2,N2)*dy34 - U(2,N3)*dy24
      dxC3 = U(3,N4)*dy23 + U(3,N2)*dy34 - U(3,N3)*dy24
      dxC4 = U(4,N4)*dy23 + U(4,N2)*dy34 - U(4,N3)*dy24

      dyC1 = U(1,N4)*dx23 + U(1,N2)*dx34 - U(1,N3)*dx24
      dyC2 = U(2,N4)*dx23 + U(2,N2)*dx34 - U(2,N3)*dx24
      dyC3 = U(3,N4)*dx23 + U(3,N2)*dx34 - U(3,N3)*dx24
      dyC4 = U(4,N4)*dx23 + U(4,N2)*dx34 - U(4,N3)*dx24

      del2(1,N3) = del2(1,N3) - coef*(dxC1*dy24 + dyC1*dx24)
      del2(2,N3) = del2(2,N3) - coef*(dxC2*dy24 + dyC2*dx24)
      del2(3,N3) = del2(3,N3) - coef*(dxC3*dy24 + dyC3*dx24)
      del2(4,N3) = del2(4,N3) - coef*(dxC4*dy24 + dyC4*dx24)

      coef = ((max(0.,min(1.,(1.+10.*vol(CC))))-1.)*epsicoef +1.)
     &                *(0.5/abs(-dx34*dy41 + dy34*dx41))

      dxC1 = U(1,N1)*dy34 + U(1,N3)*dy41 - U(1,N4)*dy31
      dxC2 = U(2,N1)*dy34 + U(2,N3)*dy41 - U(2,N4)*dy31
      dxC3 = U(3,N1)*dy34 + U(3,N3)*dy41 - U(3,N4)*dy31
      dxC4 = U(4,N1)*dy34 + U(4,N3)*dy41 - U(4,N4)*dy31

      dyC1 = U(1,N1)*dx34 + U(1,N3)*dx41 - U(1,N4)*dx31
      dyC2 = U(2,N1)*dx34 + U(2,N3)*dx41 - U(2,N4)*dx31
      dyC3 = U(3,N1)*dx34 + U(3,N3)*dx41 - U(3,N4)*dx31
      dyC4 = U(4,N1)*dx34 + U(4,N3)*dx41 - U(4,N4)*dx31

      del2(1,N4) = del2(1,N4) - coef*(dxC1*dy31 + dyC1*dx31)
```

```
            del2(2,N4) = del2(2,N4) - coef*(dxC2*dy31 + dyC2*dx31)
            del2(3,N4) = del2(3,N4) - coef*(dxC3*dy31 + dyC3*dx31)
            del2(4,N4) = del2(4,N4) - coef*(dxC4*dy31 + dyC4*dx31)

          enddo
        enddo

        EF2 = 0

        do col = 1, Elcolormax
          EF1 = EF2 + 1
          EF2 = EF1 - 1 + NEcolor(col)

CVD$    NODEPCHK
          do EF = EF1, EF2
            FF = eface(EF)
            CC = face(FF,1)
            N1 = face(FF,3)
            N2 = face(FF,4)
            N3 = face(FF,6)
            N4 = face(FF,5)

            dy24 = y(N2) - y(N4)
            dy31 = y(N3) - y(N1)
            dy12 = y(N1) - y(N2)
            dy23 = y(N2) - y(N3)
            dy41 = y(N4) - y(N1)

            dx24 = x(N2) - x(N4)
            dx31 = x(N3) - x(N1)
            dx12 = x(N1) - x(N2)
            dx23 = x(N2) - x(N3)
            dx41 = x(N4) - x(N1)

            coef = ((max(0.,min(1.,(1.+10.*vol(CC))))-1.)*epsicoef +1.)
     &                  *(0.5/abs(-dx12*dy31 + dy12*dx31))

            dxC1 = U(1,N1)*dy23 + U(1,N2)*dy31 + U(1,N3)*dy12
            dxC2 = U(2,N1)*dy23 + U(2,N2)*dy31 + U(2,N3)*dy12
            dxC3 = U(3,N1)*dy23 + U(3,N2)*dy31 + U(3,N3)*dy12
            dxC4 = U(4,N1)*dy23 + U(4,N2)*dy31 + U(4,N3)*dy12

            dyC1 = U(1,N1)*dx23 + U(1,N2)*dx31 + U(1,N3)*dx12
            dyC2 = U(2,N1)*dx23 + U(2,N2)*dx31 + U(2,N3)*dx12
            dyC3 = U(3,N1)*dx23 + U(3,N2)*dx31 + U(3,N3)*dx12
            dyC4 = U(4,N1)*dx23 + U(4,N2)*dx31 + U(4,N3)*dx12

            del2(1,N1) = del2(1,N1) + coef*(dxC1*dy12 + dyC1*dx12)
            del2(2,N1) = del2(2,N1) + coef*(dxC2*dy12 + dyC2*dx12)
            del2(3,N1) = del2(3,N1) + coef*(dxC3*dy12 + dyC3*dx12)
            del2(4,N1) = del2(4,N1) + coef*(dxC4*dy12 + dyC4*dx12)

            coef = ((max(0.,min(1.,(1.+10.*vol(CC))))-1.)*epsicoef +1.)
     &                  *(0.5/abs(-dx12*dy24 + dy12*dx24))

            dxC1 = U(1,N1)*dy24 + U(1,N2)*dy41 + U(1,N4)*dy12
            dxC2 = U(2,N1)*dy24 + U(2,N2)*dy41 + U(2,N4)*dy12
            dxC3 = U(3,N1)*dy24 + U(3,N2)*dy41 + U(3,N4)*dy12
            dxC4 = U(4,N1)*dy24 + U(4,N2)*dy41 + U(4,N4)*dy12

            dyC1 = U(1,N1)*dx24 + U(1,N2)*dx41 + U(1,N4)*dx12
            dyC2 = U(2,N1)*dx24 + U(2,N2)*dx41 + U(2,N4)*dx12
            dyC3 = U(3,N1)*dx24 + U(3,N2)*dx41 + U(3,N4)*dx12
            dyC4 = U(4,N1)*dx24 + U(4,N2)*dx41 + U(4,N4)*dx12

            del2(1,N2) = del2(1,N2) + coef*(dxC1*dy12 + dyC1*dx12)
            del2(2,N2) = del2(2,N2) + coef*(dxC2*dy12 + dyC2*dx12)
            del2(3,N2) = del2(3,N2) + coef*(dxC3*dy12 + dyC3*dx12)
            del2(4,N2) = del2(4,N2) + coef*(dxC4*dy12 + dyC4*dx12)

          enddo
        enddo
```

```
CVD$    NODEPCHK
        do PN = 1, Pmax
           P1 = pnode(PN,1)
           P2 = pnode(PN,2)
           del2(1,P1) = del2(1,P1) + del2(1,P2)
           del2(2,P1) = del2(2,P1) + del2(2,P2)
           del2(3,P1) = del2(3,P1) + del2(3,P2)
           del2(4,P1) = del2(4,P1) + del2(4,P2)
           del2(1,P2) = del2(1,P1)
           del2(2,P2) = del2(2,P1)
           del2(3,P2) = del2(3,P1)
           del2(4,P2) = del2(4,P1)
        enddo

        CC2 = 0

        do col = 1, Ccolormax
         CC1 = CC2 + 1
         CC2 = CC1 - 1 + NCcolor(col)

CVD$    NODEPCHK
         do CC = CC1, CC2
           N1 = cell(CC,5)
           N2 = cell(CC,6)
           N3 = cell(CC,7)
           N4 = cell(CC,8)

           coef12 = eps2/deltC(CC)
           coef13 = eps2/deltC(CC)
           coef14 = eps2/deltC(CC)
           coef23 = eps2/deltC(CC)
           coef24 = eps2/deltC(CC)
           coef34 = eps2/deltC(CC)

           del12 = coef12*(del2(1,N1) - del2(1,N2))
           del13 = coef13*(del2(1,N1) - del2(1,N3))
           del14 = coef14*(del2(1,N1) - del2(1,N4))
           del23 = coef23*(del2(1,N2) - del2(1,N3))
           del24 = coef24*(del2(1,N2) - del2(1,N4))
           del34 = coef34*(del2(1,N3) - del2(1,N4))

           dU(1,N1) = dU(1,N1) + (-del12 - del13 - del14)*deltN(N1)
           dU(1,N2) = dU(1,N2) + ( del12 - del23 - del24)*deltN(N2)
           dU(1,N3) = dU(1,N3) + ( del13 + del23 - del34)*deltN(N3)
           dU(1,N4) = dU(1,N4) + ( del14 + del24 + del34)*deltN(N4)

           del12 = coef12*(del2(2,N1) - del2(2,N2))
           del13 = coef13*(del2(2,N1) - del2(2,N3))
           del14 = coef14*(del2(2,N1) - del2(2,N4))
           del23 = coef23*(del2(2,N2) - del2(2,N3))
           del24 = coef24*(del2(2,N2) - del2(2,N4))
           del34 = coef34*(del2(2,N3) - del2(2,N4))

           dU(2,N1) = dU(2,N1) + (-del12 - del13 - del14)*deltN(N1)
           dU(2,N2) = dU(2,N2) + ( del12 - del23 - del24)*deltN(N2)
           dU(2,N3) = dU(2,N3) + ( del13 + del23 - del34)*deltN(N3)
           dU(2,N4) = dU(2,N4) + ( del14 + del24 + del34)*deltN(N4)

           del12 = coef12*(del2(3,N1) - del2(3,N2))
           del13 = coef13*(del2(3,N1) - del2(3,N3))
           del14 = coef14*(del2(3,N1) - del2(3,N4))
           del23 = coef23*(del2(3,N2) - del2(3,N3))
           del24 = coef24*(del2(3,N2) - del2(3,N4))
           del34 = coef34*(del2(3,N3) - del2(3,N4))

           dU(3,N1) = dU(3,N1) + (-del12 - del13 - del14)*deltN(N1)
           dU(3,N2) = dU(3,N2) + ( del12 - del23 - del24)*deltN(N2)
           dU(3,N3) = dU(3,N3) + ( del13 + del23 - del34)*deltN(N3)
           dU(3,N4) = dU(3,N4) + ( del14 + del24 + del34)*deltN(N4)

           del12 = coef12*(del2(4,N1) - del2(4,N2))
```

```
            del13 = coef13*(del2(4,N1) - del2(4,N3))
            del14 = coef14*(del2(4,N1) - del2(4,N4))
            del23 = coef23*(del2(4,N2) - del2(4,N3))
            del24 = coef24*(del2(4,N2) - del2(4,N4))
            del34 = coef34*(del2(4,N3) - del2(4,N4))

            dU(4,N1) = dU(4,N1) + (-del12 - del13 - del14)*deltN(N1)
            dU(4,N2) = dU(4,N2) + ( del12 - del23 - del24)*deltN(N2)
            dU(4,N3) = dU(4,N3) + ( del13 + del23 - del34)*deltN(N3)
            dU(4,N4) = dU(4,N4) + ( del14 + del24 + del34)*deltN(N4)

         enddo
        enddo

        return
        end




c*********************************************************************
c*                                                                   *
c*      this subroutine accounts for periodic nodes                  *
c*                                                                   *
c*********************************************************************

        subroutine bperiodic
        implicit none

        include 'QUAD.INC'
        integer PN              !pointer
        integer P1, P2          !periodic nodes

CVD$    NODEPCHK
        do PN = 1, Pmax
           P1 = pnode(PN,1)
           P2 = pnode(PN,2)
           dU(1,P1) = dU(1,P1) + dU(1,P2)
           dU(2,P1) = dU(2,P1) + dU(2,P2)
           dU(3,P1) = dU(3,P1) + dU(3,P2)
           dU(4,P1) = dU(4,P1) + dU(4,P2)
           dU(1,P2) = dU(1,P1)
           dU(2,P2) = dU(2,P1)
           dU(3,P2) = dU(3,P1)
           dU(4,P2) = dU(4,P1)
        enddo

        return
        end




c*********************************************************************
c*                                                                   *
c*      adjust inlet state vector for boundary condition             *
c*                                                                   *
c*********************************************************************

        subroutine binlet
        implicit none

        include 'QUAD.INC'
        real rrinl, uuinl, vvinl, ppinl !average values at inlet
        real ww2inl, aainl
        real rOinf, aOinf               !stag. density and speed of sound
        real HOpres                     !stagnation enthalpy and pressure
        real spres                      !entropy
        real coef
        real aaa1, aaa2, aaa3, aaa4     !coef. of inverse of matrix
```

```
      real bbb1, bbb2, bbb3, bbb4
      real ccc1, ccc2, ccc3, ccc4
      real ddd1, ddd2, ddd3, ddd4
      real ww2, uu, vv, pp, rr, aa    !values at next interior node
      real dHO, ds, dtan, dw4
      real drr, duu, dvv, dpp
      integer IN, NN                  !boundary face and its nodes

c**   average values at inlet
      rrinl = U(1,innode(Imax/2))
      uuinl = U(2,innode(Imax/2))/rrinl
      vvinl = U(3,innode(Imax/2))/rrinl
      ww2inl = uuinl**2 + vvinl**2
      ppinl = gam1*(U(4,innode(Imax/2))
     &              - 0.5*rrinl*ww2inl)
      aainl = sqrt(gam*ppinl/rrinl)

c*    subsonic inlet nodes
      if ((sqrt(ww2inl)/aainl).lt.1. .or. pitch.ne.0.) then
c**   prescribed values
        HOpres = 1./gam1
        spres = 0.

c**   coefficient of matrix
        coef = 1./(uuinl*(aainl+uuinl) + vvinl**2)
        aaa1 = (rrinl*uuinl/aainl)*coef
        aaa2 = -(ppinl/aainl)*(uuinl*gam/gam1 + ww2inl/aainl)*coef
        aaa3 = -(rrinl*vvinl/aainl)*coef
        aaa4 = (ww2inl/aainl**2)*coef
        bbb1 = uuinl*coef
        bbb2 = -(uuinl*ppinl/(gam1*rrinl))*coef
        bbb3 = -vvinl*coef
        bbb4 = -(uuinl/rrinl)*coef
        ccc1 = vvinl*coef
        ccc2 = -(vvinl*ppinl/(gam1*rrinl))*coef
        ccc3 = (aainl + uuinl)*coef
        ccc4 = -(vvinl/rrinl)*coef
        ddd1 = rrinl*aainl*uuinl*coef
        ddd2 = -(ppinl*aainl*uuinl/gam1)*coef
        ddd3 = -rrinl*aainl*vvinl*coef
        ddd4 = ww2inl*coef

CVD$    NODEPCHK
        do IN = 1, Imax
          NN = innode(IN)
          rr = U(1,NN)
          uu = U(2,NN)/rr
          vv = U(3,NN)/rr
          ww2 = uu**2 + vv**2
          pp = gam1*(U(4,NN) - 0.5*rr*ww2)
          aa = sqrt(gam*pp/rr)
          dHO  = HOpres - ((gam/gam1)*pp/rr + 0.5*ww2)
          ds   = spres - (log(gam*pp) - gam*log(rr))
          dtan = (Sinl - vv/uu)*uuinl**2
          dw4  =   dU(1,NN)*(aa*uu + gam1*ww2)
     &                - dU(2,NN)*(aa + gam1*uu)
     &                - dU(3,NN)*gam1*vv
     &                + dU(4,NN)*gam1
          drr = aaa1*dHO + aaa2*ds + aaa3*dtan + aaa4*dw4
          duu = bbb1*dHO + bbb2*ds + bbb3*dtan + bbb4*dw4
          dvv = ccc1*dHO + ccc2*ds + ccc3*dtan + ccc4*dw4
          dpp = ddd1*dHO + ddd2*ds + ddd3*dtan + ddd4*dw4

          dU(1,NN) = drr
          dU(2,NN) = rr*duu + uu*drr
          dU(3,NN) = rr*dvv + vv*drr
          dU(4,NN) = dpp/gam1 + 0.5*ww2*drr + rr*(uu*duu + vv*dvv)
        enddo

c*    supersonic inlet nodes
      else if ((sqrt(ww2inl)/aainl).ge.1.) then
c*    state vector components far from body
```

```
            r0inf = 1.
            a0inf = 1.

CVD$     NODEPCHK
            do IN = 1, Imax
               NN = innode(IN)
               rr = r0inf*(1.0+0.5*gam1*Minl**2)**(-1./gam1)
               aa = a0inf*(1.0+0.5*gam1*Minl**2)**(-0.5)
               uu = Minl*aa
               pp = rr*aa**2/gam

               dU(1,NN) = rr - U(1,NN)
               dU(2,NN) = rr*uu - U(2,NN)
               dU(3,NN) = - U(3,NN)
               dU(4,NN) = pp/gam1 + .5*(uu**2)*rr - U(4,NN)
            enddo
         endif

         return
         end



c*************************************************************
c*                                                           *
c*      adjust   &lat state vector for boundary condition    *
c*                                                           *
c*************************************************************

         subroutine boutlet
         implicit none

         include 'QUAD.INC'
         real rrout, uuout, vvout, ppout !average values at outet
         real aaout, ww2out
         real aaa1, aaa4                 !coef. of inverse of matrix
         real bbb3, bbb4
         real ccc2
         real ddd4
         real ww2, uu, vv, pp, rr, aa    !values at next interior node
         real dw1, dw2, dw3, dp
         real drr, duu, dvv, dpp
         integer ON, NN                  !boundary face and its nodes

         rrout = U(1,outnode(Onax/2))
         uuout = U(2,outnode(Onax/2))/rrout
         vvout = U(3,outnode(Onax/2))/rrout
         ww2out = uuout**2 + vvout**2
         ppout = gam1*(U(4,outnode(Onax/2))
     &                - 0.5*rrout*ww2out)
         aaout = sqrt(gam*ppout/rrout)

         aaa1 = -1./aacut**2
         aaa4 = 1./aaout**2
         bbb3 = .1/(rrout*aaout)
         bbb4 = -1./(rrout*aaout)
         ccc2 = .1/(rrout*aaout)
         ddd4 = 1.

c*  set boundary values of state vector for nodes
CVD$     NODEPCHK
         do ON = 1, Onax
            NN = outnode(ON)
            rr = U(1,NN)
            uu = U(2,NN)/rr
            vv = U(3,NN)/rr
            ww2 = uu**2 + vv**2
            pp = gam1*(U(4,NN) - 0.5*rr*ww2)
            aa = sqrt(gam*pp/rr)
            if (sqrt(ww2) .lt. aa) then
               dw1 =   dU(1,NN)*(0.5*ww2*gam1 - aa**2)
```

```
     &              - dU(2,NN)*gam1*uu
     &              - dU(3,NN)*gam1*vv
     &              + dU(4,NN)*gam1
            dw2 = - dU(1,NN)*aa*vv
     &            + dU(3,NN)*aa
            dw3 =   dU(1,NN)*(0.5*ww2*gam1 - aa*uu)
     &            - dU(2,NN)*(gam1*uu - aa)
     &            - dU(3,NN)*gam1*vv
     &            + dU(4,NN)*gam1
            dp = pout - pp
            drr = aaa1*dw1 + aaa4*dp
            duu = bbb3*dw3 + bbb4*dp
            dvv = ccc2*dw2
            dpp = ddd4*dp
            dU(1,NN) = drr
            dU(2,NN) = rr*duu + uu*drr
            dU(3,NN) = rr*dvv + vv*drr
            dU(4,NN) = dpp/gam1 + 0.5*ww2*drr + rr*(uu*duu + vv*dvv)

         endif
      enddo

      return
      end



c*********************************************************************
c*                                                                   *
c*      this subroutine changes the momentum change to make          *
c*      flow tangent to wall                                         *
c*                                                                   *
c*********************************************************************

      subroutine tangent
      implicit none

      include 'QUAD.INC'
      integer EN                        !pointer
      integer NN                        !node on wall
      real drwn                         !change in momentum normal to wall

CVD$  NODEPCHK
      do EN = 1, bnode
         NN = enode(EN)
         drwn = -(U(2,NN) + dU(2,NN))*senode(EN) +
     &           (U(3,NN) + dU(3,NN))*cenode(EN)
         dU(2,NN) = dU(2,NN) + drwn*senode(EN)
         dU(3,NN) = dU(3,NN) - drwn*cenode(EN)
      enddo

      return
      end
```

A.2.4 Jameson Scheme

```
c*********************************************************************
c*                                                                   *
c*      main program for quadrilateral Jameson scheme                *
c*                                                                   *
c*********************************************************************

      program quadrilateral
      implicit none
```

```
      include 'QUAD.INC'
      integer Niter                    !number of iterations
      real maxchange                   !max change in state vector
      integer maxnode, maxeqn          !where max change occurs
      integer CC                       !pointer

c*  read in data from file
      call gridio(1)
      call flowio(1)
      call input

      durms = 999.
      Niter = 0

c*  start history file from the top
      open(unit=35,status='unknown',form='formatted')
      write(35,2) Minl
      close(unit=35)
 2    format('inlet Mach number = ',f5.3)

c*  loop until converged
      do while ((Niter.lt.Maxiter) .and. (durms.gt.2.e-7))
         Niter = Niter + 1

c*  calculate value at next time step
         call update(maxchange, maxnode, maxeqn)

         if (mod(Niter,10).eq.0 .or. Niter.lt.10) then
            call flowio(0)
            open(unit=50, status='unknown', form='unformatted')
            write(50) Cmax, (vol(CC),CC=1,Cmax)
            close(unit=50)

 10         open(unit=35, status='old',access='append',err=10)
            write(35,1) Niter, durms, maxchange, x(maxnode),
     &                  y(maxnode), maxeqn
            close(unit=35)
            write(6,1) Niter, durms, maxchange, x(maxnode),
     &                 y(maxnode), maxeqn
         endif

 1       format('Niter=',i4,'  rms=',f9.7,'  max=',f9.7,'  x=',f6.3,
     &          '  y=',f6.3,'  eqn=',i1)

      enddo

c*  write out data to file
      call flowio(0)

      stop
      end




c*********************************************************************
c*                                                                   *
c*      update state vectors at next time step                       *
c*                                                                   *
c*********************************************************************

      subroutine update(maxchange, maxnode, maxeqn)

      include 'QUAD.INC'
      integer NN, i                    !pointer
      real U0(4,Maxnodes)              !starting values of state vector
      real alpha1, alpha2              !coefficients
      real alpha3, alpha4
      real maxchange                   !max change in state vector
      integer maxnode, maxeqn          !where max change occurs
```

```
c*  set coefficient values
        alpha1 = 0.25
        alpha2 = 1./3.
        alpha3 = 0.5
        alpha4 = 1.

c*  find timestep at each node
        call timestep

c*  advance to next time step in four steps

c*  step one
        call calcflux
        call dissipation
        do NN = 1, Nmax
           UO(1,NN) = U(1,NN)
           UO(2,NN) = U(2,NN)
           UO(3,NN) = U(3,NN)
           UO(4,NN) = U(4,NN)
           dU(1,NN) = U(1,NN)
           dU(2,NN) = U(2,NN)
           dU(3,NN) = U(3,NN)
           dU(4,NN) = U(4,NN)
           U(1,NN) = UO(1,NN) - alpha1*deltN(NN)
     &                          *(flux(1,NN)-dis(1,NN))
           U(2,NN) = UO(2,NN) - alpha1*deltN(NN)
     &                          *(flux(2,NN)-dis(2,NN))
           U(3,NN) = UO(3,NN) - alpha1*deltN(NN)
     &                          *(flux(3,NN)-dis(3,NN))
           U(4,NN) = UO(4,NN) - alpha1*deltN(NN)
     &                          *(flux(4,NN)-dis(4,NN))
           dU(1,NN) = U(1,NN) - dU(1,NN)
           dU(2,NN) = U(2,NN) - dU(2,NN)
           dU(3,NN) = U(3,NN) - dU(3,NN)
           dU(4,NN) = U(4,NN) - dU(4,NN)
        enddo
        call binlet
        call boutlet
        call tangent

c*  step two
        call calcflux
        call dissipation
        do NN = 1, Nmax
           dU(1,NN) = U(1,NN)
           dU(2,NN) = U(2,NN)
           dU(3,NN) = U(3,NN)
           dU(4,NN) = U(4,NN)
           U(1,NN) = UO(1,NN) - alpha2*deltN(NN)
     &                          *(flux(1,NN)-dis(1,NN))
           U(2,NN) = UO(2,NN) - alpha2*deltN(NN)
     &                          *(flux(2,NN)-dis(2,NN))
           U(3,NN) = UO(3,NN) - alpha2*deltN(NN)
     &                          *(flux(3,NN)-dis(3,NN))
           U(4,NN) = UO(4,NN) - alpha2*deltN(NN)
     &                          *(flux(4,NN)-dis(4,NN))
           dU(1,NN) = U(1,NN) - dU(1,NN)
           dU(2,NN) = U(2,NN) - dU(2,NN)
           dU(3,NN) = U(3,NN) - dU(3,NN)
           dU(4,NN) = U(4,NN) - dU(4,NN)
        enddo
        call binlet
        call boutlet
        call tangent

c*  step three
        call calcflux
        do NN = 1, Nmax
           dU(1,NN) = U(1,NN)
           dU(2,NN) = U(2,NN)
           dU(3,NN) = U(3,NN)
```

```
            dU(4,NN) = U(4,NN)
            U(1,NN) = UO(1,NN) - alpha3*deltN(NN)
     &                           *(flux(1,NN)-dis(1,NN))
            U(2,NN) = UO(2,NN) - alpha3*deltN(NN)
     &                           *(flux(2,NN)-dis(2,NN))
            U(3,NN) = UO(3,NN) - alpha3*deltN(NN)
     &                           *(flux(3,NN)-dis(3,NN))
            U(4,NN) = UO(4,NN) - alpha3*deltN(NN)
     &                           *(flux(4,NN)-dis(4,NN))
            dU(1,NN) = U(1,NN) - dU(1,NN)
            dU(2,NN) = U(2,NN) - dU(2,NN)
            dU(3,NN) = U(3,NN) - dU(3,NN)
            dU(4,NN) = U(4,NN) - dU(4,NN)
         enddo
         call binlet
         call boutlet
         call tangent

c*  step four
         call calcflux
         do NN = 1, Nmax
            dU(1,NN) = U(1,NN)
            dU(2,NN) = U(2,NN)
            dU(3,NN) = U(3,NN)
            dU(4,NN) = U(4,NN)
            U(1,NN) = UO(1,NN) - alpha4*deltN(NN)
     &                           *(flux(1,NN)-dis(1,NN))
            U(2,NN) = UO(2,NN) - alpha4*deltN(NN)
     &                           *(flux(2,NN)-dis(2,NN))
            U(3,NN) = UO(3,NN) - alpha4*deltN(NN)
     &                           *(flux(3,NN)-dis(3,NN))
            U(4,NN) = UO(4,NN) - alpha4*deltN(NN)
     &                           *(flux(4,NN)-dis(4,NN))
            dU(1,NN) = U(1,NN) - U(1,NN)
            dU(2,NN) = U(2,NN) - U(2,NN)
            dU(3,NN) = U(3,NN) - U(3,NN)
            dU(4,NN) = U(4,NN) - U(4,NN)
         enddo
         call binlet
         call boutlet
         call tangent

c*  find root mean square difference in state vector
         durms = 0.0
         maxchange = 0.

         do i = 1, 4
            do NN = 1, Nmax
               durms = durms + (U(i,NN) - UO(i,NN))**2
               if (abs(U(i,NN)-UO(i,NN)).gt.abs(maxchange)) then
                  maxchange = U(i,NN) - UO(i,NN)
                  maxnode = NN
                  maxeqn = i
               endif
            enddo
         enddo

         durms = sqrt(durms/(4.*Nmax))

         return
         end




c*****************************************************************
c*                                                               *
c*      calculate time step for nodes                            *
c*                                                               *
c*****************************************************************
```

```
      subroutine timestep
      implicit none

      include 'QUAD.INC'
      integer CC, NN                    !pointer
      integer N1, N2, N3, N4            !nodes related to face
      integer PN, P1, P2                !periodic nodes
      real dxl, dyl, dl
      real dxm, dym, dm
      real uu, vv, aa, rr, vv2, pp      !values at node
      real time1, time2

c*    zero out delt
      do NN = 1, Nmax
         deltN(NN) = 0.
      enddo

c*    find time step for each cell
      do CC = 1, Cmax
         N1 = cell(CC,5)
         N2 = cell(CC,6)
         N3 = cell(CC,7)
         N4 = cell(CC,8)
         dxl = .5*(x(N3) + x(N2) - x(N4) - x(N1))
         dyl = .5*(y(N3) + y(N2) - y(N4) - y(N1))
         dl  = sqrt(dxl**2 + dyl**2)
         dxm = .5*(x(N4) + x(N3) - x(N1) - x(N2))
         dym = .5*(y(N4) + y(N3) - y(N1) - y(N2))
         dm  = sqrt(dxm**2 + dym**2)
         rr = 0.25*(U(1,N1) + U(1,N2) + U(1,N3) + U(1,N4))
         uu = 0.25*(U(2,N1) + U(2,N2) + U(2,N3) + U(2,N4))/rr
         vv = 0.25*(U(3,N1) + U(3,N2) + U(3,N3) + U(3,N4))/rr
         pp = gam1*(0.25*(U(4,N1) + U(4,N2) + U(4,N3) + U(4,N4))
     &           - 0.5*rr*(uu**2 + vv**2))
         aa = sqrt(gam*pp/rr)

         time1 = CFL/(abs(uu*dyl-vv*dxl) + aa*dl)
         time2 = CFL/(abs(uu*dym-vv*dxm) + aa*dm)

         deltC(CC) = min(time1,time2)

         deltN(N1) = deltN(N1) + 1./deltC(CC)
         deltN(N2) = deltN(N2) + 1./deltC(CC)
         deltN(N3) = deltN(N3) + 1./deltC(CC)
         deltN(N4) = deltN(N4) + 1./deltC(CC)
      enddo

c**   find time step at periodic nodes
CVD$     NODEPCHK
      do PN = 1, Pmax
         P1 = pnode(PN,1)
         P2 = pnode(PN,2)
         deltN(P1) = deltN(P1) + deltN(P2)
         deltN(P2) = deltN(P1)
      enddo

c**   delt is actually delt/areaN
      do NN = 1, Nmax
         deltN(NN) = 1./deltN(NN)
      enddo

      return
      end
```

```
c*****************************************************************
```

```
c*                                                                     *
c*      calculate flux vector values at nodes                          *
c*                                                                     *
c**********************************************************************

        subroutine calcflux
        implicit none

        include 'QUAD.INC'
        real delflux
        integer CC, NN                  !pointers
        integer N1, N2, N3, N4          !nodes around edge
        real WW
        real dy31, dy24, dx31, dx24     !change in x and y
        real dF24, dF13, dG24, dG13     !change in F and G

        do NN = 1, Nmax
c*  set flux to zero
           flux(1,NN) = 0.
           flux(2,NN) = 0.
           flux(3,NN) = 0.
           flux(4,NN) = 0.

c*  calculate f and g at all nodes
           WW = 0.5*(U(2,NN)**2 + U(3,NN)**2)/U(1,NN)
           F(1,NN) = U(2,NN)
           F(2,NN) = U(2,NN)**2/U(1,NN) + gam1*(U(4,NN) - WW)
           F(3,NN) = U(2,NN)*U(3,NN)/U(1,NN)
           F(4,NN) = (U(2,NN)/U(1,NN))*(gam*U(4,NN) - gam1*WW)

           G(1,NN) = U(3,NN)
           G(2,NN) = U(2,NN)*U(3,NN)/U(1,NN)
           G(3,NN) = U(3,NN)**2/U(1,NN) + gam1*(U(4,NN) - WW)
           G(4,NN) = (U(3,NN)/U(1,NN))*(gam*U(4,NN) - gam1*WW)
        enddo

c*  find change in state vector for cell
        do CC = 1, Cmax
           N1 = cell(CC,5)
           N2 = cell(CC,6)
           N3 = cell(CC,7)
           N4 = cell(CC,8)

           dy31 = y(N3) - y(N1)
           dy24 = y(N2) - y(N4)
           dx31 = x(N3) - x(N1)
           dx24 = x(N2) - x(N4)

           dF24 = F(1,N2) - F(1,N4)
           dF13 = F(1,N1) - F(1,N3)
           dG24 = G(1,N2) - G(1,N4)
           dG13 = G(1,N1) - G(1,N3)

           delflux = 0.5*(-dF24*dy31 + dG24*dx31 -
     &                    dF13*dy24 + dG13*dx24)

           flux(1,N1) = flux(1,N1) - delflux
           flux(1,N2) = flux(1,N2) - delflux
           flux(1,N3) = flux(1,N3) - delflux
           flux(1,N4) = flux(1,N4) - delflux

           dF24 = F(2,N2) - F(2,N4)
           dF13 = F(2,N1) - F(2,N3)
           dG24 = G(2,N2) - G(2,N4)
           dG13 = G(2,N1) - G(2,N3)

           delflux = 0.5*(-dF24*dy31 + dG24*dx31 -
     &                    dF13*dy24 + dG13*dx24)

           flux(2,N1) = flux(2,N1) - delflux
           flux(2,N2) = flux(2,N2) - delflux
           flux(2,N3) = flux(2,N3) - delflux
```

```
          flux(2,N4) = flux(2,N4) - delflux

          dF24 = F(3,N2) - F(3,N4)
          dF13 = F(3,N1) - F(3,N3)
          dG24 = G(3,N2) - G(3,N4)
          dG13 = G(3,N1) - G(3,N3)

          delflux = 0.5*(-dF24*dy31 + dG24*dx31 -
     &                    dF13*dy24 + dG13*dx24)

          flux(3,N1) = flux(3,N1) - delflux
          flux(3,N2) = flux(3,N2) - delflux
          flux(3,N3) = flux(3,N3) - delflux
          flux(3,N4) = flux(3,N4) - delflux

          dF24 = F(4,N2) - F(4,N4)
          dF13 = F(4,N1) - F(4,N3)
          dG24 = G(4,N2) - G(4,N4)
          dG13 = G(4,N1) - G(4,N3)

          delflux = 0.5*(-dF24*dy31 + dG24*dx31 -
     &                    dF13*dy24 + dG13*dx24)

          flux(4,N1) = flux(4,N1) - delflux
          flux(4,N2) = flux(4,N2) - delflux
          flux(4,N3) = flux(4,N3) - delflux
          flux(4,N4) = flux(4,N4) - delflux

        enddo

c*  implement wall boundary conditions
        call bwall

        return
        end



c*********************************************************************
c*                                                                   *
c*      this subroutine applies the wall boundary condition          *
c*                                                                   *
c*********************************************************************

        subroutine bwall
        implicit none

        include 'QUAD.INC'
        real delflux
        integer PN, EF, FF, CC  !pointer
        integer N1, N2          !nodes on edge
        integer P1, P2
        real coef
        real dy12, dx12         !change in x and y
        real pres1, pres2       !pressure
        integer EF1, EF2, col   !coloring pointers

c*  upper and lower boundary
        EF2 = 0

        do col = 1, Elcolormax
         EF1 = EF2 + 1
         EF2 = EF1 - 1 + NEcolor(col)
         do EF = EF1, EF2

           FF = eface(EF)
           CC = face(FF,1)
           N1 = face(FF,3)
           N2 = face(FF,4)
```

```
            dy12 = y(N1) - y(N2)
            dx12 = x(N1) - x(N2)

            pres1 = gam1*(U(4,N1) - 0.5*(U(2,N1)**2 + U(3,N1)**2)/
     &           U(1,N1))
            pres2 = gam1*(U(4,N2) - 0.5*(U(2,N2)**2 + U(3,N2)**2)/
     &           U(1,N2))

            delflux = 0.5*(-(F(1,N2) + F(1,N1))*dy12 +
     &                     (G(1,N2) + G(1,N1))*dx12)

            flux(1,N1) = flux(1,N1) - delflux
            flux(1,N2) = flux(1,N2) - delflux

            delflux = 0.5*(-(F(2,N2) + F(2,N1) - pres1 - pres2)*dy12 +
     &                     (G(2,N2) + G(2,N1))*dx12)

            flux(2,N1) = flux(2,N1) - delflux
            flux(2,N2) = flux(2,N2) - delflux

            delflux = 0.5*(-(F(3,N2) + F(3,N1))*dy12 +
     &                     (G(3,N2) + G(3,N1) - pres1 - pres2)*dx12)

            flux(3,N1) = flux(3,N1) - delflux
            flux(3,N2) = flux(3,N2) - delflux

            delflux = 0.5*(-(F(4,N2) + F(4,N1))*dy12 +
     &                     (G(4,N2) + G(4,N1))*dx12)

            flux(4,N1) = flux(4,N1) - delflux
            flux(4,N2) = flux(4,N2) - delflux

          enddo
        enddo

c*  account for periodic nodes
CVD$    NODEPCHK
        do PN = 1, Pmax
           P1 = pnode(PN,1)
           P2 = pnode(PN,2)
           flux(1,P1) = flux(1,P1) + flux(1,P2)
           flux(2,P1) = flux(2,P1) + flux(2,P2)
           flux(3,P1) = flux(3,P1) + flux(3,P2)
           flux(4,P1) = flux(4,P1) + flux(4,P2)
           flux(1,P2) = flux(1,P1)
           flux(2,P2) = flux(2,P1)
           flux(3,P2) = flux(3,P1)
           flux(4,P2) = flux(4,P1)
        enddo

        return
        end




c*********************************************************************
c*                                                                   *
c*      calculate the dissipation at each of the nodes for           *
c*      the current values of the state vector                       *
c*                                                                   *
c*********************************************************************

        subroutine dissipation
        implicit none

        include 'QUAD.INC'
        integer N1, N2, N3, N4          !nodes at end of face
        real pres1, pres2, pres3        !pressure at nodes
        real pres4, pres
        integer EF, FF, CC, NN          !pointers
```

```
      integer EF1, EF2
      integer PN, P1, P2              !periodic nodes
      real del2(4,Maxnodes)           !second order changes
      real eps1(Maxnodes)             !dissipation coeficients
      real del12, del13, del14
      real del23, del24, del34
      real coef12, coef13, coef14
      real coef23, coef24, coef34
      real coef1122, coef1133, coef1144
      real coef2233, coef2244, coef3344
      real coef
      real dx31, dx24                 !change in x
      real dx12, dx23, dx34, dx41
      real dy31, dy24                 !change in y
      real dy12, dy23, dy34, dy41
      real dxC1, dxC2, dxC3, dxC4     !change in x in cell
      real dyC1, dyC2, dyC3, dyC4     !change in y in cell
      integer CC1, CC2, col           !color pointers

      do NN = 1, Nmax
         del2(1,NN) = 0.
         del2(2,NN) = 0.
         del2(3,NN) = 0.
         del2(4,NN) = 0.

         dis(1,NN) = 0.
         dis(2,NN) = 0.
         dis(3,NN) = 0.
         dis(4,NN) = 0.

         eps1(NN) = 0.
      enddo

c** Mavriplis smoothing
      if (sigE .eq. 0.) then

      CC2 = 0

      do col = 1, Ccolormax
       CC1 = CC2 + 1
       CC2 = CC1 - 1 + NCcolor(col)

CVD$    NODEPCHK
        do CC = CC1, CC2
          N1 = cell(CC,5)
          N2 = cell(CC,6)
          N3 = cell(CC,7)
          N4 = cell(CC,8)

          del2(1,N1) = del2(1,N1) +
     &                 (-3.*U(1,N1) + U(1,N2) +
     &                        U(1,N3) + U(1,N4))
          del2(1,N2) = del2(1,N2) +
     &                 (U(1,N1) - 3.*U(1,N2) +
     &                    U(1,N3) + U(1,N4))
          del2(1,N3) = del2(1,N3) +
     &                 (U(1,N1) + U(1,N2) -
     &                    3.*U(1,N3) + U(1,N4))
          del2(1,N4) = del2(1,N4) +
     &                 (U(1,N1) + U(1,N2) +
     &                       U(1,N3) - 3.*U(1,N4))

          del2(2,N1) = del2(2,N1) +
     &                 (-3.*U(2,N1) + U(2,N2) +
     &                        U(2,N3) + U(2,N4))
          del2(2,N2) = del2(2,N2) +
     &                 (U(2,N1) - 3.*U(2,N2) +
     &                       U(2,N3) + U(2,N4))
          del2(2,N3) = del2(2,N3) +
     &                 (U(2,N1) + U(2,N2) -
     &                    3.*U(2,N3) + U(2,N4))
          del2(2,N4) = del2(2,N4) +
```

```
     &                (U(2,N1) + U(2,N2) +
     &                     U(2,N3) - 3.*U(2,N4))

          del2(3,N1) = del2(3,N1) +
     &                (-3.*U(3,N1) + U(3,N2) +
     &                        U(3,N3) + U(3,N4))
          del2(3,N2) = del2(3,N2) +
     &                (U(3,N1) - 3.*U(3,N2) +
     &                     U(3,N3) + U(3,N4))
          del2(3,N3) = del2(3,N3) +
     &                (U(3,N1) + U(3,N2) -
     &                   3.*U(3,N3) + U(3,N4))
          del2(3,N4) = del2(3,N4) +
     &                (U(3,N1) + U(3,N2) +
     &                     U(3,N3) - 3.*U(3,N4))

          del2(4,N1) = del2(4,N1) +
     &                (-3.*U(4,N1) + U(4,N2) +
     &                        U(4,N3) + U(4,N4))
          del2(4,N2) = del2(4,N2) +
     &                (U(4,N1) - 3.*U(4,N2) +
     &                     U(4,N3) + U(4,N4))
          del2(4,N3) = del2(4,N3) +
     &                (U(4,N1) + U(4,N2) -
     &                   3.*U(4,N3) + U(4,N4))
          del2(4,N4) = del2(4,N4) +
     &                (U(4,N1) + U(4,N2) +
     &                     U(4,N3) - 3.*U(4,N4))

         enddo
        enddo

CVD$    NODEPCHK
        do PN = 1, Pmax
           P1 = pnode(PN,1)
           P2 = pnode(PN,2)
           del2(1,P1) = del2(1,P1) + del2(1,P2)
           del2(2,P1) = del2(2,P1) + del2(2,P2)
           del2(3,P1) = del2(3,P1) + del2(3,P2)
           del2(4,P1) = del2(4,P1) + del2(4,P2)
           del2(1,P2) = del2(1,P1)
           del2(2,P2) = del2(2,P1)
           del2(3,P2) = del2(3,P1)
           del2(4,P2) = del2(4,P1)
        enddo

c** Ni scheme method of smoothing
        else if (sigE .eq. 1.) then

        CC2 = 0

        do col = 1, Ccolormax
         CC1 = CC2 + 1
         CC2 = CC1 - 1 + NCcolor(col)

CVD$    NODEPCHK
         do CC = CC1, CC2
           N1 = cell(CC,5)
           N2 = cell(CC,6)
           N3 = cell(CC,7)
           N4 = cell(CC,8)

           dy24 = y(N2) - y(N4)
           dy31 = y(N3) - y(N1)
           dy12 = y(N1) - y(N2)
           dy23 = y(N2) - y(N3)
           dy34 = y(N3) - y(N4)
           dy41 = y(N4) - y(N1)

           dx24 = x(N2) - x(N4)
           dx31 = x(N3) - x(N1)
           dx12 = x(N1) - x(N2)
```

```
dx23 = x(N2) - x(N3)
dx34 = x(N3) - x(N4)
dx41 = x(N4) - x(N1)

coef = 1./abs(-dx12*dy41 + dy12*dx41)

dxC1 = U(1,N1)*dy24 + U(1,N2)*dy41 + U(1,N4)*dy12
dxC2 = U(2,N1)*dy24 + U(2,N2)*dy41 + U(2,N4)*dy12
dxC3 = U(3,N1)*dy24 + U(3,N2)*dy41 + U(3,N4)*dy12
dxC4 = U(4,N1)*dy24 + U(4,N2)*dy41 + U(4,N4)*dy12

dyC1 = U(1,N1)*dx24 + U(1,N2)*dx41 + U(1,N4)*dx12
dyC2 = U(2,N1)*dx24 + U(2,N2)*dx41 + U(2,N4)*dx12
dyC3 = U(3,N1)*dx24 + U(3,N2)*dx41 + U(3,N4)*dx12
dyC4 = U(4,N1)*dx24 + U(4,N2)*dx41 + U(4,N4)*dx12

del2(1,N1) = del2(1,N1) - 0.5*coef*(dxC1*dy24 + dyC1*dx24)
del2(2,N1) = del2(2,N1) - 0.5*coef*(dxC2*dy24 + dyC2*dx24)
del2(3,N1) = del2(3,N1) - 0.5*coef*(dxC3*dy24 + dyC3*dx24)
del2(4,N1) = del2(4,N1) - 0.5*coef*(dxC4*dy24 + dyC4*dx24)

coef = 1./abs(-dx12*dy23 + dy12*dx23)

dxC1 = U(1,N1)*dy23 + U(1,N2)*dy31 + U(1,N3)*dy12
dxC2 = U(2,N1)*dy23 + U(2,N2)*dy31 + U(2,N3)*dy12
dxC3 = U(3,N1)*dy23 + U(3,N2)*dy31 + U(3,N3)*dy12
dxC4 = U(4,N1)*dy23 + U(4,N2)*dy31 + U(4,N3)*dy12

dyC1 = U(1,N1)*dx23 + U(1,N2)*dx31 + U(1,N3)*dx12
dyC2 = U(2,N1)*dx23 + U(2,N2)*dx31 + U(2,N3)*dx12
dyC3 = U(3,N1)*dx23 + U(3,N2)*dx31 + U(3,N3)*dx12
dyC4 = U(4,N1)*dx23 + U(4,N2)*dx31 + U(4,N3)*dx12

del2(1,N2) = del2(1,N2) - 0.5*coef*(dxC1*dy31 + dyC1*dx31)
del2(2,N2) = del2(2,N2) - 0.5*coef*(dxC2*dy31 + dyC2*dx31)
del2(3,N2) = del2(3,N2) - 0.5*coef*(dxC3*dy31 + dyC3*dx31)
del2(4,N2) = del2(4,N2) - 0.5*coef*(dxC4*dy31 + dyC4*dx31)

coef = 1./abs(-dx23*dy34 + dy23*dx34)

dxC1 = U(1,N4)*dy23 + U(1,N2)*dy34 - U(1,N3)*dy24
dxC2 = U(2,N4)*dy23 + U(2,N2)*dy34 - U(2,N3)*dy24
dxC3 = U(3,N4)*dy23 + U(3,N2)*dy34 - U(3,N3)*dy24
dxC4 = U(4,N4)*dy23 + U(4,N2)*dy34 - U(4,N3)*dy24

dyC1 = U(1,N4)*dx23 + U(1,N2)*dx34 - U(1,N3)*dx24
dyC2 = U(2,N4)*dx23 + U(2,N2)*dx34 - U(2,N3)*dx24
dyC3 = U(3,N4)*dx23 + U(3,N2)*dx34 - U(3,N3)*dx24
dyC4 = U(4,N4)*dx23 + U(4,N2)*dx34 - U(4,N3)*dx24

del2(1,N3) = del2(1,N3) + 0.5*coef*(dxC1*dy24 + dyC1*dx24)
del2(2,N3) = del2(2,N3) + 0.5*coef*(dxC2*dy24 + dyC2*dx24)
del2(3,N3) = del2(3,N3) + 0.5*coef*(dxC3*dy24 + dyC3*dx24)
del2(4,N3) = del2(4,N3) + 0.5*coef*(dxC4*dy24 + dyC4*dx24)

coef = 1./abs(-dx34*dy41 + dy34*dx41)

dxC1 = U(1,N1)*dy34 + U(1,N3)*dy41 - U(1,N4)*dy31
dxC2 = U(2,N1)*dy34 + U(2,N3)*dy41 - U(2,N4)*dy31
dxC3 = U(3,N1)*dy34 + U(3,N3)*dy41 - U(3,N4)*dy31
dxC4 = U(4,N1)*dy34 + U(4,N3)*dy41 - U(4,N4)*dy31

dyC1 = U(1,N1)*dx34 + U(1,N3)*dx41 - U(1,N4)*dx31
dyC2 = U(2,N1)*dx34 + U(2,N3)*dx41 - U(2,N4)*dx31
dyC3 = U(3,N1)*dx34 + U(3,N3)*dx41 - U(3,N4)*dx31
dyC4 = U(4,N1)*dx34 + U(4,N3)*dx41 - U(4,N4)*dx31

del2(1,N4) = del2(1,N4) + 0.5*coef*(dxC1*dy31 + dyC1*dx31)
del2(2,N4) = del2(2,N4) + 0.5*coef*(dxC2*dy31 + dyC2*dx31)
del2(3,N4) = del2(3,N4) + 0.5*coef*(dxC3*dy31 + dyC3*dx31)
del2(4,N4) = del2(4,N4) + 0.5*coef*(dxC4*dy31 + dyC4*dx31)
```

```
 enddo
enddo

EF2 = 0

do col = 1, Elcolormax
 EF1 = EF2 + 1
 EF2 = EF1 - 1 + NEcolor(col)

 do EF = EF1, EF2
   FF = eface(EF)
   CC = face(FF,1)
   N1 = face(FF,3)
   N2 = face(FF,4)
   N3 = face(FF,6)
   N4 = face(FF,5)

   dy24 = y(N2) - y(N4)
   dy31 = y(N3) - y(N1)
   dy12 = y(N1) - y(N2)
   dy23 = y(N2) - y(N3)
   dy41 = y(N4) - y(N1)

   dx24 = x(N2) - x(N4)
   dx31 = x(N3) - x(N1)
   dx12 = x(N1) - x(N2)
   dx23 = x(N2) - x(N3)
   dx41 = x(N4) - x(N1)

   coef = 1./abs(-dx12*dy41 + dy12*dx41)

   dxC1 = U(1,N1)*dy24 + U(1,N2)*dy41 + U(1,N4)*dy12
   dxC2 = U(2,N1)*dy24 + U(2,N2)*dy41 + U(2,N4)*dy12
   dxC3 = U(3,N1)*dy24 + U(3,N2)*dy41 + U(3,N4)*dy12
   dxC4 = U(4,N1)*dy24 + U(4,N2)*dy41 + U(4,N4)*dy12

   dyC1 = U(1,N1)*dx24 + U(1,N2)*dx41 + U(1,N4)*dx12
   dyC2 = U(2,N1)*dx24 + U(2,N2)*dx41 + U(2,N4)*dx12
   dyC3 = U(3,N1)*dx24 + U(3,N2)*dx41 + U(3,N4)*dx12
   dyC4 = U(4,N1)*dx24 + U(4,N2)*dx41 + U(4,N4)*dx12

   del2(1,N1) = del2(1,N1) + 0.5*coef*(dxC1*dy12 + dyC1*dx12)
   del2(2,N1) = del2(2,N1) + 0.5*coef*(dxC2*dy12 + dyC2*dx12)
   del2(3,N1) = del2(3,N1) + 0.5*coef*(dxC3*dy12 + dyC3*dx12)
   del2(4,N1) = del2(4,N1) + 0.5*coef*(dxC4*dy12 + dyC4*dx12)

   coef = 1./abs(-dx12*dy23 + dy12*dx23)

   dxC1 = U(1,N1)*dy23 + U(1,N2)*dy31 + U(1,N3)*dy12
   dxC2 = U(2,N1)*dy23 + U(2,N2)*dy31 + U(2,N3)*dy12
   dxC3 = U(3,N1)*dy23 + U(3,N2)*dy31 + U(3,N3)*dy12
   dxC4 = U(4,N1)*dy23 + U(4,N2)*dy31 + U(4,N3)*dy12

   dyC1 = U(1,N1)*dx23 + U(1,N2)*dx31 + U(1,N3)*dx12
   dyC2 = U(2,N1)*dx23 + U(2,N2)*dx31 + U(2,N3)*dx12
   dyC3 = U(3,N1)*dx23 + U(3,N2)*dx31 + U(3,N3)*dx12
   dyC4 = U(4,N1)*dx23 + U(4,N2)*dx31 + U(4,N3)*dx12

   del2(1,N2) = del2(1,N2) + 0.5*coef*(dxC1*dy12 + dyC1*dx12)
   del2(2,N2) = del2(2,N2) + 0.5*coef*(dxC2*dy12 + dyC2*dx12)
   del2(3,N2) = del2(3,N2) + 0.5*coef*(dxC3*dy12 + dyC3*dx12)
   del2(4,N2) = del2(4,N2) + 0.5*coef*(dxC4*dy12 + dyC4*dx12)

 enddo
enddo

CVD$  NODEPCHK
      do PN = 1, Pmax
         P1 = pnode(PN,1)
         P2 = pnode(PN,2)
         del2(1,P1) = del2(1,P1) + del2(1,P2)
         del2(2,P1) = del2(2,P1) + del2(2,P2)
```

```
            del2(3,P1) = del2(3,P1) + del2(3,P2)
            del2(4,P1) = del2(4,P1) + del2(4,P2)
            del2(1,P2) = del2(1,P1)
            del2(2,P2) = del2(2,P1)
            del2(3,P2) = del2(3,P1)
            del2(4,P2) = del2(4,P1)
          enddo

          endif

c*  undivided Laplacian of pressure for eps1 coefficient
          CC2 = 0

          do col = 1, Ccolormax
           CC1 = CC2 + 1
           CC2 = CC1 - 1 + NCcolor(col)

cCVD$     NODEPCHK
           do CC = CC1, CC2
            N1 = cell(CC,5)
            N2 = cell(CC,6)
            N3 = cell(CC,7)
            N4 = cell(CC,8)

            pres1 = gam1*(U(4,N1) - 0.5*(U(2,N1)**2 +
     &                  U(3,N1)**2)/U(1,N1))
            pres2 = gam1*(U(4,N2) - 0.5*(U(2,N2)**2 +
     &                  U(3,N2)**2)/U(1,N2))
            pres3 = gam1*(U(4,N3) - 0.5*(U(2,N3)**2 +
     &                  U(3,N3)**2)/U(1,N3))
            pres4 = gam1*(U(4,N4) - 0.5*(U(2,N4)**2 +
     &                  U(3,N4)**2)/U(1,N4))

            eps1(N1) = eps1(N1) - 3.*pres1 + pres2 + pres3 + pres4
            eps1(N2) = eps1(N2) + pres1 - 3.*pres2 + pres3 + pres4
            eps1(N3) = eps1(N3) + pres1 + pres2 - 3.*pres3 + pres4
            eps1(N4) = eps1(N4) + pres1 + pres2 + pres3 - 3.*pres4
           enddo
          enddo

CVD$      NODEPCHK
          do PN = 1, Pmax
            P1 = pnode(PN,1)
            P2 = pnode(PN,2)
            eps1(P1) = eps1(P1) + eps1(P2)
            eps1(P2) = eps1(P1)
          enddo

          do NN = 1, Nmax
            pres = gam1*(U(4,NN) - 0.5*(U(2,NN)**2 +
     &                  U(3,NN)**2)/U(1,NN))
            eps1(NN) = abs(eps1(NN)/pres)
          enddo

          CC2 = 0

          do col = 1, Ccolormax
           CC1 = CC2 + 1
           CC2 = CC1 - 1 + NCcolor(col)

CVD$      NODEPCHK
           do CC = CC1, CC2
            N1 = cell(CC,5)
            N2 = cell(CC,6)
            N3 = cell(CC,7)
            N4 = cell(CC,8)

            coef12 = 0.5*eps1coef*(eps1(N1) + eps1(N2))
            coef13 = 0.5*eps1coef*(eps1(N1) + eps1(N3))
            coef14 = 0.5*eps1coef*(eps1(N1) + eps1(N4))
            coef23 = 0.5*eps1coef*(eps1(N2) + eps1(N3))
            coef24 = 0.5*eps1coef*(eps1(N2) + eps1(N4))
```

```
      coef34 = 0.5*epsicoef*(epsi(N3) + epsi(N4))

      coef1122 = max(0.,eps2-coef12)/deltC(CC)
      coef1133 = max(0.,eps2-coef13)/deltC(CC)
      coef1144 = max(0.,eps2-coef14)/deltC(CC)
      coef2233 = max(0.,eps2-coef23)/deltC(CC)
      coef2244 = max(0.,eps2-coef24)/deltC(CC)
      coef3344 = max(0.,eps2-coef34)/deltC(CC)

      del12 = coef12*(U(1,N1) - U(1,N2)) -
     &        coef1122*(del2(1,N1) - del2(1,N2))
      del13 = coef13*(U(1,N1) - U(1,N3)) -
     &        coef1133*(del2(1,N1) - del2(1,N3))
      del14 = coef14*(U(1,N1) - U(1,N4)) -
     &        coef1144*(del2(1,N1) - del2(1,N4))
      del23 = coef23*(U(1,N2) - U(1,N3)) -
     &        coef2233*(del2(1,N2) - del2(1,N3))
      del24 = coef24*(U(1,N2) - U(1,N4)) -
     &        coef2244*(del2(1,N2) - del2(1,N4))
      del34 = coef34*(U(1,N3) - U(1,N4)) -
     &        coef3344*(del2(1,N3) - del2(1,N4))

      dis(1,N1) = dis(1,N1) - del12 - del13 - del14
      dis(1,N2) = dis(1,N2) + del12 - del23 - del24
      dis(1,N3) = dis(1,N3) + del13 + del23 - del34
      dis(1,N4) = dis(1,N4) + del14 + del24 + del34

      del12 = coef12*(U(2,N1) - U(2,N2)) -
     &        coef1122*(del2(2,N1) - del2(2,N2))
      del13 = coef13*(U(2,N1) - U(2,N3)) -
     &        coef1133*(del2(2,N1) - del2(2,N3))
      del14 = coef14*(U(2,N1) - U(2,N4)) -
     &        coef1144*(del2(2,N1) - del2(2,N4))
      del23 = coef23*(U(2,N2) - U(2,N3)) -
     &        coef2233*(del2(2,N2) - del2(2,N3))
      del24 = coef24*(U(2,N2) - U(2,N4)) -
     &        coef2244*(del2(2,N2) - del2(2,N4))
      del34 = coef34*(U(2,N3) - U(2,N4)) -
     &        coef3344*(del2(2,N3) - del2(2,N4))

      dis(2,N1) = dis(2,N1) - del12 - del13 - del14
      dis(2,N2) = dis(2,N2) + del12 - del23 - del24
      dis(2,N3) = dis(2,N3) + del13 + del23 - del34
      dis(2,N4) = dis(2,N4) + del14 + del24 + del34

      del12 = coef12*(U(3,N1) - U(3,N2)) -
     &        coef1122*(del2(3,N1) - del2(3,N2))
      del13 = coef13*(U(3,N1) - U(3,N3)) -
     &        coef1133*(del2(3,N1) - del2(3,N3))
      del14 = coef14*(U(3,N1) - U(3,N4)) -
     &        coef1144*(del2(3,N1) - del2(3,N4))
      del23 = coef23*(U(3,N2) - U(3,N3)) -
     &        coef2233*(del2(3,N2) - del2(3,N3))
      del24 = coef24*(U(3,N2) - U(3,N4)) -
     &        coef2244*(del2(3,N2) - del2(3,N4))
      del34 = coef34*(U(3,N3) - U(3,N4)) -
     &        coef3344*(del2(3,N3) - del2(3,N4))

      dis(3,N1) = dis(3,N1) - del12 - del13 - del14
      dis(3,N2) = dis(3,N2) + del12 - del23 - del24
      dis(3,N3) = dis(3,N3) + del13 + del23 - del34
      dis(3,N4) = dis(3,N4) + del14 + del24 + del34

      del12 = coef12*(U(4,N1) - U(4,N2)) -
     &        coef1122*(del2(4,N1) - del2(4,N2))
      del13 = coef13*(U(4,N1) - U(4,N3)) -
     &        coef1133*(del2(4,N1) - del2(4,N3))
      del14 = coef14*(U(4,N1) - U(4,N4)) -
     &        coef1144*(del2(4,N1) - del2(4,N4))
      del23 = coef23*(U(4,N2) - U(4,N3)) -
     &        coef2233*(del2(4,N2) - del2(4,N3))
      del24 = coef24*(U(4,N2) - U(4,N4)) -
```

```
     &             coef2244*(del2(4,N2) - del2(4,N4))
           del34 = coef34*(U(4,N3) - U(4,N4)) -
     &             coef3344*(del2(4,N3) - del2(4,N4))

           dis(4,N1) = dis(4,N1) - del12 - del13 - del14
           dis(4,N2) = dis(4,N2) + del12 - del23 - del24
           dis(4,N3) = dis(4,N3) + del13 + del23 - del34
           dis(4,N4) = dis(4,N4) + del14 + del24 + del34

         enddo
        enddo

CVD$     NODEPCHK
        do PN = 1, Pmax
           P1 = pnode(PN,1)
           P2 = pnode(PN,2)
           dis(1,P1) = dis(1,P1) + dis(1,P2)
           dis(2,P1) = dis(2,P1) + dis(2,P2)
           dis(3,P1) = dis(3,P1) + dis(3,P2)
           dis(4,P1) = dis(4,P1) + dis(4,P2)
           dis(1,P2) = dis(1,P1)
           dis(2,P2) = dis(2,P1)
           dis(3,P2) = dis(3,P1)
           dis(4,P2) = dis(4,P1)
        enddo

        return
        end




c*********************************************************************
c*                                                                   *
c*      adjust inlet state vector for boundary condition             *
c*                                                                   *
c*********************************************************************

        subroutine binlet
        implicit none

        include 'QUAD.INC'
        real rrinl, uuinl, vvinl, ppinl !average values at inlet
        real ww2inl, aainl
        real r0inf, a0inf               !stag. density and speed of sound
        real H0pres                     !stagnation enthalpy and pressure
        real spres                      !entropy
        real coef
        real aaa1, aaa2, aaa3, aaa4     !coef. of inverse of matrix
        real bbb1, bbb2, bbb3, bbb4
        real ccc1, ccc2, ccc3, ccc4
        real ddd1, ddd2, ddd3, ddd4
        real ww2, uu, vv, pp, rr, aa    !values at next interior node
        real dH0, ds, dtan, dw4
        real drr, duu, dvv, dpp
        integer IN, NN                  !boundary face and its nodes

c**  average values at inlet
        rrinl = U(1,innode(Imax/2)) - dU(1,innode(Imax/2))
        uuinl = (U(2,innode(Imax/2)) - dU(2,innode(Imax/2)))/rrinl
        vvinl = (U(3,innode(Imax/2)) - dU(3,innode(Imax/2)))/rrinl
        ww2inl = uuinl**2 + vvinl**2
        ppinl = gam1*((U(4,innode(Imax/2)) - dU(4,innode(Imax/2)))
     &              - 0.5*rrinl*ww2inl)
        aainl = sqrt(gam*ppinl/rrinl)

c*  subsonic inlet nodes
        if ((sqrt(ww2inl)/aainl).lt.1. .or. pitch.ne.0.) then
c**  prescribed values
           H0pres = 1./gam1
```

```
          spres = 0.

c**  coefficient of matrix
          coef = 1./(uuinl*(aainl+uuinl) + vvinl**2)
          aaa1 = (rrinl*uuinl/aainl)*coef
          aaa2 = -(ppinl/aainl)*(uuinl*gam/gam1 + ww2inl/aainl)*coef
          aaa3 = -(rrinl*vvinl/aainl)*coef
          aaa4 = (ww2inl/aainl**2)*coef
          bbb1 = uuinl*coef
          bbb2 = -(uuinl*ppinl/(gam1*rrinl))*coef
          bbb3 = -vvinl*coef
          bbb4 = -(uuinl/rrinl)*coef
          ccc1 = vvinl*coef
          ccc2 = -(vvinl*ppinl/(gam1*rrinl))*coef
          ccc3 = (aainl + uuinl)*coef
          ccc4 = -(vvinl/rrinl)*coef
          ddd1 = rrinl*aainl*uuinl*coef
          ddd2 = -(ppinl*aainl*uuinl/gam1)*coef
          ddd3 = -rrinl*aainl*vvinl*coef
          ddd4 = ww2inl*coef

CVD$     NODEPCHK
          do IN = 1, Imax
             NN = innode(IN)
             rr = U(1,NN) - dU(1,NN)
             uu = (U(2,NN) - dU(2,NN))/rr
             vv = (U(3,NN) - dU(3,NN))/rr
             ww2 = uu**2 + vv**2
             pp = gam1*((U(4,NN) - dU(4,NN)) - 0.5*rr*ww2)
             aa = sqrt(gam*pp/rr)
             dH0  = H0pres - ((gam/gam1)*pp/rr + 0.5*ww2)
             ds   = spres - (log(gam*pp) - gam*log(rr))
             dtan = (Sinl - vv/uu)*uuinl**2
             dw4  =   dU(1,NN)*(aa*uu + gam1*ww2)
     &                   - dU(2,NN)*(aa + gam1*uu)
     &                   - dU(3,NN)*gam1*vv
     &                   + dU(4,NN)*gam1
             drr = aaa1*dH0 + aaa2*ds + aaa3*dtan + aaa4*dw4
             duu = bbb1*dH0 + bbb2*ds + bbb3*dtan + bbb4*dw4
             dvv = ccc1*dH0 + ccc2*ds + ccc3*dtan + ccc4*dw4
             dpp = ddd1*dH0 + ddd2*ds + ddd3*dtan + ddd4*dw4

             U(1,NN) = U(1,NN) - dU(1,NN) + drr
             U(2,NN) = U(2,NN) - dU(2,NN) + rr*duu + uu*drr
             U(3,NN) = U(3,NN) - dU(3,NN) + rr*dvv + vv*drr
             U(4,NN) = U(4,NN) - dU(4,NN) + dpp/gam1 + 0.5*ww2*drr
     &                   + rr*(uu*duu + vv*dvv)
          enddo

c*   supersonic inlet nodes
        else if ((sqrt(ww2inl)/aainl).ge.1.) then
c*   state vector components far from body
          rOinf = 1.
          aOinf = 1.

CVD$     NODEPCHK
          do IN = 1, Imax
             NN = innode(IN)
             rr = rOinf*(1.0+0.5*gam1*Minl**2)**(-1./gam1)
             aa = aOinf*(1.0+0.5*gam1*Minl**2)**(-0.5)
             uu = Minl*aa
             pp = rr*aa**2/gam

             U(1,NN) = rr
             U(2,NN) = rr*uu
             U(3,NN) = 0.
             U(4,NN) = pp/gam1 + .5*(uu**2)*rr
          enddo
        endif

        return
        end
```

```
c*****************************************************************
c*                                                               *
c*      adjust outlet state vector for boundary condition        *
c*                                                               *
c*****************************************************************

        subroutine boutlet
        implicit none

        include 'QUAD.INC'
        real rrout, uuout, vvout, ppout !average values at outet
        real aaout, ww2out
        real aaa1, aaa4                 !coef. of inverse of matrix
        real bbb3, bbb4
        real ccc2
        real ddd4
        real ww2, uu, vv, pp, rr, aa    !values at next interior node
        real dw1, dw2, dw3, dp
        real drr, duu, dvv, dpp
        integer ON, NN                  !boundary face and its nodes

        rrout = U(1,outnode(Omax/2)) - dU(1,outnode(Omax/2))
        uuout = (U(2,outnode(Omax/2)) - dU(2,outnode(Omax/2)))/rrout
        vvout = (U(3,outnode(Omax/2)) - dU(3,outnode(Omax/2)))/rrout
        ww2out = uuout**2 + vvout**2
        ppout = gam1*((U(4,outnode(Omax/2)) - dU(4,outnode(Omax/2)))
     &              - 0.5*rrout*ww2out)
        aaout = sqrt(gam*ppout/rrout)

        aaa1 = -1./aaout**2
        aaa4 = 1./aaout**2
        bbb3 = .1/(rrout*aaout)
        bbb4 = -1./(rrout*aaout)
        ccc2 = .1/(rrout*aaout)
        ddd4 = 1.

c*  set boundary values of state vector for nodes
CVD$    NODEPCHK
        do ON = 1, Omax
           NN = outnode(ON)
           rr = U(1,NN) - dU(1,NN)
           uu = (U(2,NN) - dU(2,NN))/rr
           vv = (U(3,NN) - dU(3,NN))/rr
           ww2 = uu**2 + vv**2
           pp = gam1*((U(4,NN) - dU(4,NN)) - 0.5*rr*ww2)
           aa = sqrt(gam*pp/rr)
           if (sqrt(ww2) .lt. aa) then
              dw1 =   dU(1,NN)*(0.5*ww2*gam1 - aa**2)
     &              - dU(2,NN)*gam1*uu
     &              - dU(3,NN)*gam1*vv
     &              + dU(4,NN)*gam1
              dw2 = - dU(1,NN)*aa*vv
     &              + dU(3,NN)*aa
              dw3 =   dU(1,NN)*(0.5*ww2*gam1 - aa*uu)
     &              - dU(2,NN)*(gam1*uu - aa)
     &              - dU(3,NN)*gam1*vv
     &              + dU(4,NN)*gam1
              dp = pout - pp
              drr = aaa1*dw1 + aaa4*dp
              duu = bbb3*dw3 + bbb4*dp
              dvv = ccc2*dw2
              dpp = ddd4*dp
              U(1,NN) = U(1,NN) - dU(1,NN) + drr
              U(2,NN) = U(2,NN) - dU(2,NN) + rr*duu + uu*drr
              U(3,NN) = U(3,NN) - dU(3,NN) + rr*dvv + vv*drr
              U(4,NN) = U(4,NN) - dU(4,NN) + dpp/gam1 + 0.5*ww2*drr
     &                  + rr*(uu*duu + vv*dvv)
           endif
```

```
      enddo

      return
      end



c*********************************************************************
c*                                                                   *
c*     this subroutine changes the momentum change to make           *
c*     flow tangent to wall                                          *
c*                                                                   *
c*********************************************************************

      subroutine tangent
      implicit none

      include 'QUAD.INC'
      integer EN                        !pointer
      integer NN                        !node on wall
      real drwn                         !change in momentum normal to wall

CVD$  NODEPCHK
      do EN = 1, bnode
         NN = enode(EN)
         drwn = -U(2,NN)*senode(EN) + U(3,NN)*cenode(EN)
         U(2,NN) = U(2,NN) + drwn*senode(EN)
         U(3,NN) = U(3,NN) - drwn*cenode(EN)
      enddo

      return
      end
```

A.2.5 Plotting Package

```
      program plotgen
      implicit none

      include 'QUAD.INC'

      integer i
      integer NN, EN, CC
      integer ptype                     !type of plot
      integer ctype                     !type of contour
      integer stype                     !type of surface distributions
      integer N1, N2, N3, N4
      character*80 TITLE, ITITLE
      integer NTITL                     !number of leters in title
      external g2pltg, g2pltc           !plotting subroutines
      integer indgr
      integer a4, a5, a6, a7, a8, a9, a10
      real uu, vv, pp, rr, mm, mm2      !from state vectors
      real pt, ptinf
      real zz, z(Maxnodes)              !contour values
      real zmax, zmin                   !max and min of z array
      real boundary                     !boundary on contour levels
      integer NCONT                     !contour level info
      real CBASE, CSTEP
      real Cinc                         !contour increment
      character*6 NUM                   !same as CSTEP but character
      integer NLINE, IOPT(2)            !indicators for line plots
      real tau, radius2                 !define bump geometry
      integer points, npts(2)           !number of points on line
      real xline(Maxedges)              !points on line to plot
```

```
      real yline(Maxedges)

c*  read data from file
      call gridio(1)
      call flowio(1)

c*  initialize GRAFIC
      write(ITITLE,1) Minl
 1    format('INLET MACH NUMBER = ',F5.3)
      call grinit(5,6,ITITLE)

      do while (1)

c*  prompt user for type of plot
         type*, 'Type of plot'
         type*, '   0) STOP'
         type*, '   1) grid'
         type*, '   2) contour'
         type*, '   3) surface distribution'
         type*, '   4) data'
         type 11
 11      format($,' selection = ')
         accept 111, ptype
 111     format(I)

         if (ptype.eq.0) then
            stop
         else if (ptype.eq.1) then

            TITLE = '   X       Y    COMPUTATIONAL GRID'
            indgr = 23
            call gr_control(g2pltg, indgr, TITLE, x, y, Nmax, a4,
     &                  a5, a6, a7, a8, a9, a10)

         else if (ptype.eq.2) then
c*  choose type of contour plot
            do while (1)

            Cinc = 0.

 2          type*, 'Type of contour'
            type*, '   0) TOP LEVEL'
            type*, '   1) density'
            type*, '   2) Mach number'
            type*, '   3) normal velocity'
            type*, '   4) pressure'
            type*, '   5) total pressure loss'
            type*, '   6) entropy'
            type*, '   7) linear (not related to solution)'
            type*, '   8) vol'
            type*, '   9) stagnation enthalpy'
            type*, '  10) CONTOUR INCREMENT'
            type 22
 22         format($,' selection = ')
            accept 222, ctype
 222        format(I)

c*  exit from contour loop
            if (ctype.eq.0) goto 999

            if (ctype.eq.10) then
               type 21
 21            format($,' CONTOUR INCREMENT = ')
               accept 221, Cinc
 221           format(F)
               type*, ' '
               goto 2
            endif

c*  set up nodal contour values
            if (ctype.eq.5) then
               ptinf = 1./gam
```

```
      endif

      zmin = 1.e20
      zmax = -1.e20

      do NN = 1, Nmax
         rr = U(1,NN)
         uu = U(2,NN)/rr
         vv = U(3,NN)/rr

         if (ctype.eq.1) then
            zz = rr
         else if (ctype.eq.2) then
            pp = gami*(U(4,NN) - 0.5*rr*(uu**2 + vv**2))
            mm = sqrt(rr*(uu**2 + vv**2)/(gam*pp))
            zz = mm
         else if (ctype.eq.3) then
            zz = sqrt(uu**2 + vv**2)
         else if (ctype.eq.4) then
            pp = gami*(U(4,NN) - 0.5*rr*(uu**2 + vv**2))
            zz = pp
         else if (ctype.eq.5) then
            pp = gami*(U(4,NN) - 0.5*rr*(uu**2 + vv**2))
            mm2 = rr*(uu**2 + vv**2)/(gam*pp)
            pt = pp*(1.+.5*gami*mm2)**(gam/gami)
            zz = (1. - pt/ptinf)*100.
         else if (ctype.eq.6) then
            pp = gami*(U(4,NN) - 0.5*rr*(uu**2 + vv**2))
            zz = (log(gam*pp) - gam*log(rr))*100.
         else if (ctype.eq.7) then
            zz = (1.+x(NN))*10.
         else if (ctype.eq.9) then
            pp = gami*(U(4,NN) - 0.5*rr*(uu**2 + vv**2))
            zz = ((gam/gami)*(pp/rr) + 0.5*(uu**2 + vv**2))*100.
         endif

         zmax = max(zmax,zz)
         zmin = min(zmin,zz)
         z(NN) = zz
      enddo

      if (ctype.eq.8) then
         open(unit=50, status='unknown', form='unformatted')
         read(50) Cmax, (vol(CC),CC=1,Cmax)
         close(unit=50)
         zmin = 1.e20
         zmax = -1.e20
         do NN = 1, Nmax
            z(NN) = 0.
         enddo
         do CC = 1, Cmax
            N1 = cell(CC,5)
            N2 = cell(CC,6)
            N3 = cell(CC,7)
            N4 = cell(CC,8)
            zz = vol(CC)
            zmax = max(zmax,zz)
            zmin = min(zmin,zz)
            z(N1) = z(N1) + 0.25*zz
            z(N2) = z(N2) + 0.25*zz
            z(N3) = z(N3) + 0.25*zz
            z(N4) = z(N4) + 0.25*zz
         enddo
      endif

c*    set title and extra variables needed for GRAFIC
      if (ctype.eq.1) then
         TITLE = '   X       Y    DENSITY '
         NTITL = 24
         NCONT = 20
      else if (ctype.eq.2) then
         TITLE = '   X       Y    MACH NUMBER '
```

```
                  NTITL = 28
                  NCONT = 20
               else if (ctype.eq.3) then
                  TITLE = '   X       Y    NORMAL VELOCITY '
                  NTITL = 32
                  NCONT = 20
               else if (ctype.eq.4) then
                  TITLE = '   X       Y    PRESSURE '
                  NTITL = 25
                  NCONT = 20
               else if (ctype.eq.5) then
                  TITLE = '   X       Y    % TOTAL PRESSURE LOSS '
                  NTITL = 38
                  NCONT = 20
               else if (ctype.eq.6) then
                  TITLE = '   X       Y    % ENTROPY '
                  NTITL = 26
                  NCONT = 20
               else if (ctype.eq.7) then
                  TITLE = '   X       Y    LINEAR '
                  NTITL = 23
                  NCONT = 20
               else if (ctype.eq.8) then
                  TITLE = '   X       Y    VOL '
                  NTITL = 20
                  NCONT = 20
               else if (ctype.eq.9) then
                  TITLE = '   X       Y    % STAGNATION ENTHALPY '
                  NTITL = 38
                  NCONT = 20
               endif

c*  find contour levels
               if (Cinc.ne.0.) then
                  NCONT = int((zmax-zmin)/Cinc + 2.)
                  CSTEP = Cinc
                  CBASE = (real(int((zmin/Cinc)-1.)))*Cinc
               elseif (ctype.eq.2) then
                  boundary = 0.
                  do while (1)
                     boundary = boundary + 0.05
                     if (boundary.gt.zmin) then
                        zmin = boundary - 0.05
                        goto 200
                     endif
                  enddo
 200              boundary = 0.
                  do while (1)
                     boundary = boundary + 0.05
                     if (boundary.gt.zmax) then
                        zmax = boundary
                        goto 210
                     endif
                  enddo

 210              CBASE = zmin
                  NCONT = int((zmax-zmin)/0.05)
                  CSTEP = 0.05
               else
                  call GR_SCALE(zmin, zmax, NCONT-1,  CBASE, CSTEP)
               endif

               z(Nmax+1) = NCONT
               z(Nmax+2) = CBASE + 0.01*CSTEP
               z(Nmax+3) = CSTEP

c*  finish up title
               TITLE(NTITL+1:NTITL+24) = 'CONTOURS WITH INCREMENT '
               write(NUM,10) CSTEP
 10            format(F6.3)
               TITLE(NTITL+25:80) = NUM
```

```
c*  plot the contour lines
            indgr = 23
            call gr_control(g2pltc, indgr, TITLE, z, Nmax, x, y,
     &                  a5, a6, a7, a8, a9, a10)

            enddo

c*  plot surface distribution
          else if (ptype.eq.3) then
            do while (1)

            type*, 'Type of surface distribution'
            type*, '   0) TOP LEVEL'
            type*, '   1) density'
            type*, '   2) Mach number'
            type*, '   3) normal velocity'
            type*, '   4) surface flow angle'
            type*, '   5) analytical surface flow angle'
            type*, '   6) pressure'
            type*, '   7) entropy'
            type*, '   8) stagnation enthalpy'
            type*, '   9) total pressure loss'
            type 33
 33         format($,' selection = ')
            accept 333, stype
 333        format(I)

c*  exit from contour loop
            if (stype.eq.0) goto 999

c*  set up array with points on line for upper surface
            if (stype.eq.9) then
               ptinf = 1./gam
            endif

            points = 0
            do EN = 1, tnode
               points = points + 1
               NN = enode(EN)
               rr = U(1,NN)
               uu = U(2,NN)/rr
               vv = U(3,NN)/rr

               if (stype.eq.1) then
                  zz = rr
               else if (stype.eq.2) then
                  pp = gami*(U(4,NN) - 0.5*rr*(uu**2 + vv**2))
                  mm = sqrt(rr*(uu**2 + vv**2)/(gam*pp))
                  zz = mm
               else if (stype.eq.3) then
                  zz = sqrt(uu**2 + vv**2)
               else if (stype.eq.4) then
                  zz = atan(vv/uu)
               else if (stype.eq.5) then
                  zz = atan(senode(EN)/cenode(EN))
               else if (stype.eq.6) then
                  pp = gami*(U(4,NN) - 0.5*rr*(uu**2 + vv**2))
                  zz = pp
               else if (stype.eq.7) then
                  pp = gami*(U(4,NN) - 0.5*rr*(uu**2 + vv**2))
                  zz = (log(gam*pp) - gam*log(rr))*100.
               else if (stype.eq.8) then
                  pp = gami*(U(4,NN) - 0.5*rr*(uu**2 + vv**2))
                  zz = ((gam/gami)*(pp/rr) + 0.5*(uu**2 + vv**2))*100.
               else if (stype.eq.9) then
                  pp = gami*(U(4,NN) - 0.5*rr*(uu**2 + vv**2))
                  mm2 = rr*(uu**2 + vv**2)/(gam*pp)
                  pt = pp*(1.+.5*gami*mm2)**(gam/gami)
                  zz = (1. - pt/ptinf)*100.
               endif

               xline(points) = x(NN)
```

```
                  yline(points) = zz
               enddo

               npts(1) = points

c*  set up array with points on line for lower surface
               points= 0
               do EN = tnode+1, bnode
                  points = points + 1
                  NN = enode(EN)
                  rr = U(1,NN)
                  uu = U(2,NN)/rr
                  vv = U(3,NN)/rr

                  if (stype.eq.1) then
                     zz = rr
                  else if (stype.eq.2) then
                     pp = gam1*(U(4,NN) - 0.5*rr*(uu**2 + vv**2))
                     mm = sqrt(rr*(uu**2 + vv**2)/(gam*pp))
                     zz = mm
                  else if (stype.eq.3) then
                     zz = sqrt(uu**2 + vv**2)
                  else if (stype.eq.4) then
                     zz = atan(vv/uu)
                  else if (stype.eq.5) then
                     zz = atan(senode(EN)/cenode(EN))
                  else if (stype.eq.6) then
                     pp = gam1*(U(4,NN) - 0.5*rr*(uu**2 + vv**2))
                     zz = pp
                  else if (stype.eq.7) then
                     pp = gam1*(U(4,NN) - 0.5*rr*(uu**2 + vv**2))
                     zz = (log(gam*pp) - gam*log(rr))*100.
                  else if (stype.eq.8) then
                     pp = gam1*(U(4,NN) - 0.5*rr*(uu**2 + vv**2))
                     zz = ((gam/gam1)*(pp/rr) + 0.5*(uu**2 + vv**2))*100.
                  else if (stype.eq.9) then
                     pp = gam1*(U(4,NN) - 0.5*rr*(uu**2 + vv**2))
                     mm2 = rr*(uu**2 + vv**2)/(gam*pp)
                     pt = pp*(1.+.5*gam1*mm2)**(gam/gam1)
                     zz = (1. - pt/ptinf)*100.
                  endif

                  xline(npts(1)+points) = x(NN)
                  yline(npts(1)+points) = zz
               enddo

               npts(2) = points

c** set plot indicators
               NLINE = 2
               IOPT(1) = 14
               IOPT(2) = 14

c**  set title for plot
               if (stype.eq.1) then
                  TITLE = '    X    DENSITY DENSITY'
               else if (stype.eq.2) then
                  TITLE = '    X    MACH NO.MACH NUMBER'
               else if (stype.eq.3) then
                  TITLE = '    X    VELOCITYNORMAL VELOCITY'
               else if (stype.eq.4) then
                  TITLE = '    X     ANGLE  SURFACE FLOW ANGLE'
               else if (stype.eq.5) then
                  TITLE = '    X     ANGLE  ANALYTICAL SURFACE FLOW ANGLE'
               else if (stype.eq.6) then
                  TITLE = '    X    PRESSUREPRESSURE'
               else if (stype.eq.7) then
                  TITLE = '    X    ENTROPY~% ENTROPY'
               else if (stype.eq.8) then
                  TITLE =
     &             '    X    ~STAGNATION ENTHALPY~% STAGNATION ENTHALPY'
               else if (stype.eq.9) then
```

```
               TITLE =
     &          '   X    "TOTAL PRESSURE LOSS"% TOTAL PRESSURE LOSS'
            endif

c*  plot the line
            indgr = 21
            call gr_line(IOPT,NLINE,TITLE,INDGR,xline,yline,npts)

            enddo
         else if (ptype.eq.4) then
c*  calculate mean of total pressure loss
            ptinf = 1./gam
            zz = 0.
            zmax = 0.
            do NN = 1, Nmax
               rr = U(1,NN)
               uu = U(2,NN)/rr
               vv = U(3,NN)/rr
               pp = gam1*(U(4,NN) - 0.5*rr*(uu**2 + vv**2))
               mm2 = rr*(uu**2 + vv**2)/(gam*pp)
               pt = pp*(1.+.5*gam1*mm2)**(gam/gam1)
               zz = zz + (1. - pt/ptinf)**2
               zmax = max(zmax,abs(1.-pt/ptinf))
            enddo
            zz = sqrt(zz/real(Nmax))

            type*, ' '
            type*, 'inlet Mach number     = ', Minl
            type*, 'rms total pres. loss  = ', zz, ' (',log10(zz),')'
            type*, 'max total pres. loss  = ', zmax, ' (',log10(zmax),')'
            type*, 'cells                 = ', Cmax
            type*, 'faces                 = ', Fmax
            type*, 'nodes                 = ', Nmax
            type*, 'inlet nodes           = ', Imax
            type*, 'outlet nodes          = ', Omax
            type*, 'periodic nodes        = ', Pmax
            type*, 'edge nodes            = ', Emax
            type*, 'face colors           = ', Fcolormax
            type*, 'cell colors           = ', Ccolormax
            type*, 'edge colors           = ', E2colormax
            type*, ' '
 999     endif
      enddo

      end



      subroutine g2pltg(ifun, indgr, TITLE, alimits, info_string,
     &                  a1, a2, a3, a4, a5, a6, a7, a8, a9, a10)
      implicit none

      include 'QUAD.INC'

      integer FF, EF, NN
      integer indgr
      character*80 TITLE
      integer ifun
      real alimits(4)
      integer a1, a2, a3, a4, a5, a6, a7, a8, a9, a10
      character*80 info_string
      real x1, x2, y1, y2

      if (ifun.eq.0) then
         return
      elseif (ifun.eq.1) then
         call gr_get_limits(x, y, Nmax, alimits)
         if (pitch.ne.0.) then
            alimits(4) = alimits(4) + pitch
         endif
      elseif (ifun.eq.2) then
         info_string = ' '
```

```
      elseif (ifun.eq.3) then
         do FF = 1, Fmax
            NN = face(FF,3)
            x1 = x(NN)
            y1 = y(NN)
            NN = face(FF,4)
            x2 = x(NN)
            y2 = y(NN)
            call gr_move(x1, y1, 0)
            call gr_draw(x2, y2, 0)
            if (pitch.ne.0.) then
               y1 = y1 + pitch
               y2 = y2 + pitch
               call gr_move(x1, y1, 0)
               call gr_draw(x2, y2, 0)
            endif
         enddo
      endif

      return
      end




      subroutine g2pltc(ifun, indgr, TITLE, alimits, info_string,
     &                  z, a2, a3, a4, a5, a6, a7, a8, a9, a10)
      implicit none

      include 'QUAD.INC'

      integer FF, EF, NN, CC, NC       !pointers
      integer node                     !pointer
      integer indgr
      character*80 TITLE
      integer ifun
      real alimits(4)
      real z(Maxnodes)
      integer a2, a3, a4, a5, a6, a7, a8, a9, a10
      character*80 info_string
      real x1, x2, y1, y2, y3, y4
      integer ncont                    !determine contour levels
      real cbase, cstep
      integer KK                       !pointer
      integer N1, N2, N3, N4           !nodes at end of edge face
      real xn(4), yn(4), zn(4)         !x, y, and contour value at nodes
      real xpoint, ypoint              !x, y of pointer
      real value                       !value of contour at pointer
      real cn(2,4)                     !x, y at cell nodes
      integer IIN                      !is pointer in cell?
      real zcont, cont(30)             !contour levels
      real zmax, zmin                  !max & min contour values in cell

      if (ifun.eq.0) then
         return
      elseif (ifun.eq.1) then
         call gr_get_limits(x, y, Nmax, alimits)
         if (pitch.ne.0.) then
            alimits(4) = alimits(4) + pitch
         endif

      elseif (ifun.eq.2) then
c*  find contour value at point
         xpoint = alimits(1)
         ypoint = alimits(2)

c*  look for cell (x,y) is in
         do KK = 1, Cmax
```

```
                N1 = cell(KK,6)
                N2 = cell(KK,6)
                N3 = cell(KK,7)
                N4 = cell(KK,8)
                cn(1,1) = x(N1)
                cn(1,2) = x(N2)
                cn(1,3) = x(N3)
                cn(1,4) = x(N4)
                cn(2,1) = y(N1)
                cn(2,2) = y(N2)
                cn(2,3) = y(N3)
                cn(2,4) = y(N4)
                call gr_inside(IIN,cn,4,xpoint,ypoint)
                if (IIN.eq.1) then
c*  linear interpolation for value
                   zn(1) = z(N1)
                   zn(2) = z(N2)
                   zn(3) = z(N3)
                   zn(4) = z(N4)
                   call gr_contour_value(cn,zn,xpoint,ypoint,value)
                   goto 30
                endif
                cn(2,1) = cn(2,1) + pitch
                cn(2,2) = cn(2,2) + pitch
                cn(2,3) = cn(2,3) + pitch
                cn(2,4) = cn(2,4) + pitch
                call gr_inside(IIN,cn,4,xpoint,ypoint)
                if (IIN.eq.1) then
c*  linear interpolation for value
                   zn(1) = z(N1)
                   zn(2) = z(N2)
                   zn(3) = z(N3)
                   zn(4) = z(N4)
                   call gr_contour_value(cn,zn,xpoint,ypoint,value)
                   goto 30
                endif
             enddo
 30          write (INFO_STRING,31) value
 31          format(' function value = ',f15.6)

          elseif (ifun.eq.3) then

c*  plot boundary
             do EF = 1, Emax
                FF = eface(EF)
                N1 = face(FF,3)
                N2 = face(FF,4)
                call gr_move(x(N1), y(N1), 0)
                call gr_draw(x(N2), y(N2), 0)
             enddo

             if (pitch.ne.0.) then
                do EF = 1, Emax
                   FF = eface(EF)
                   N1 = face(FF,3)
                   N2 = face(FF,4)
                   y1 = y(N1) + pitch
                   y2 = y(N2) + pitch
                   call gr_move(x(N1), y1, 0)
                   call gr_draw(x(N2), y2, 0)
                enddo
             endif

c*  set up contour levels
             ncont = z(Nmax+1)
             cbase = z(Nmax+2)
             cstep = z(Nmax+3)
             do NC = 1, ncont
                cont(NC) = cbase + (NC-1)*cstep
             enddo

c*  loop through cells
```

```
            do CC = 1, Cmax

               do NN = 1, 4
                  node = cell(CC,NN+4)
                  xn(NN) = x(node)
                  yn(NN) = y(node)
                  zn(NN) = z(node)
               enddo

               zmax = max(zn(1), zn(2), zn(3), zn(4))
               zmin = min(zn(1), zn(2), zn(3), zn(4))

c*  find contour crossing on triangle for each contour level
               do NC = 1, ncont
                  zcont = cont(NC)
                  if (zcont.lt.zmin .or. zcont.gt.zmax) then
c*                  no need to compute
                  else
                     call gr_cbox(xn(1), yn(1), zn(1),
     &                            xn(2), yn(2), zn(2),
     &                            xn(3), yn(3), zn(3),
     &                            xn(4), yn(4), zn(4),zcont)
                     if (pitch.ne.0.) then
                        y1 = yn(1) + pitch
                        y2 = yn(2) + pitch
                        y3 = yn(3) + pitch
                        y4 = yn(4) + pitch
                        call gr_cbox(xn(1), y1, zn(1),
     &                               xn(2), y2, zn(2),
     &                               xn(3), y3, zn(3),
     &                               xn(4), y4, zn(4),zcont)
                     endif
                  endif
               enddo
            enddo
         endif

         return
         end
```