Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

H.T.M. van der Maarel, P.W. Hemker, C.T.H. Everaars

EULER: An adaptive Euler code

Department of Numerical Mathematics Report NM-R9015 August

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Research (N.W.O.).

EULER:
An Adaptive Euler Code

H.T.M. van der Maarel
P.W. Hemker
C.T.H. Everaars

Centre for Mathematics and Computer Science
P.O. box 4079, 1009 AB Amsterdam, The Netherlands

Abstract

In this report the implementation of an algorithm to perform first order accurate Euler flow computations on a self-adaptive grid is described. The data structure used allows for multigrid convergence acceleration on locally refined grids, consisting of nested quadrilaterals. The discretization and the multigrid procedure are based on Osher's scheme and finite volumes.

1987 CR Categories: E.1, G.1.8
1980 Mathematic Subject Classification: 65N50, 76N10, 76-04
Keywords & Prhases: adaptive multigrid, Euler equations
Note: This research was performed as part of a BRITE/EURAM project under Contract no. AERO-0003-C.

1 Introduction

In this report we describe the FORTRAN implementation of an algorithm to perform 2D Euler flow computations with automatic mesh adaptation. The implementation uses the data structure as described in [1]. This structure allows multigrid convergence acceleration and the use of locally refined nested grids.

The data handling routines and the routines for Euler flow computation are separate modules in the code. The former, which set up and handle the data structure, can be used to implement different computational schemes and are independent of the computational part of the code. This module is called 'BASIS'. The module performing the adaptive multigrid Euler flow computation is called 'EULER'.

The Euler solver used is based on the solver developed by Hemker and Spekreijse [2]. It uses a first-order upwind discretization of the steady Euler equations, in a finite volume context. The numerical fluxes are computed using the P-variant of Osher's numerical flux function [2]. The system of discrete equations is then solved by a collective point Gauss-Seidel relaxation, with a multigrid convergence acceleration technique.

First we give a short review of the features of the data structure that are essential for the present application and a brief survey of the basic computational method. Next we describe how these two are combined into the adaptive Euler code. Finally a computational example is shown.

Report NM-R9015
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

2 Review of the data structure

2.1 Grids on different levels of refinement

A strict definition of the elements of the data structure is given in [1]. It is assumed that the physical domain of definition of the problem, $\Omega \in I\!R^2$, can be approximated by a regular partitioning of *quadrilaterals* and that this set—possibly after some coordinate transformation—forms a regular set of quadrangles. Each such quadrilateral or quadrangle is called a *cell*. The cells and their edges make up the grid which covers the complete, or only a part of the computational domain.

We consider different levels of refinement. The coarsest grid is denoted by Ω^0. Each cell of the grid Ω^l on level $l > 0$ is a member of a division of a cell of Ω^{l-1}, into a set of 2×2 smaller cells. The coarse cell of Ω^{l-1} is not removed when the four smaller cells are generated, but it is coexistent with the cells of Ω^l. Except for the cells of the coarsest grid, Ω^0, each cell is one of the four descendants of a cell on a coarser (lower) level. The cell of the coarser grid is called the *parent* and its four descendants are called its *kids*. In this way all the cells in the data structure are related to each other in a *quad tree* structure.

2.2 Coordinates

The grid Ω^0 of the coarsest level, completely covers the computational domain Ω. Since each cell of Ω^0 can have either *none* or only *one* neighbouring cell at each side, each cell is determined by a set of coordinates $(i, j) \in I\!N^2$. Then we have: a cell $\Omega^l_{i,j} \in \Omega^l$ is the eastern neighbour of $\Omega^l_{i-1,j}$ and the northern neighbour of $\Omega^l_{i,j-1}$. A similar numbering is used on levels $l > 0$. The coordinates (m, n) of the kids $\Omega^{l+1}_{m,n}$ of a cell $\Omega^l_{i,j}$ on level l are defined such, that the kids are $\Omega^{l+1}_{2i,2j}$, $\Omega^{l+1}_{2i+1,2j}$, $\Omega^{l+1}_{2i,2j+1}$ and $\Omega^{l+1}_{2i+1,2j+1}$.

In an analogous way the coordinates (i, j) and the level l determine a point of intersection $P^l_{i,j}$ of grid lines: a grid point. Grid point $P^l_{i,j}$ is defined as the south-western corner point of $\Omega^l_{i,j}$. Furthermore, this integer coordinate system is chosen such that for Ω^0 the smallest coordinate in either direction is zero.

2.3 Patches

The grid, which forms the geometric system of the structure, is composed of so-called *patches*. To each corner point a patch is associated. A patch consists of the corner point and possibly a horizontal wall, a vertical wall and a cell. On the other hand, each corner point, each wall and each cell belongs to some patch. As mentioned before, cells are related in a quad tree structure. Due to this and the definition of a patch, patches are also related in a quad tree structure. Just like cells, every patch has a parent patch (except for the very root), and may have neighbour and kid patches.

Remark 1 *The notions 'patch' and 'cell' should not be confused; they are not the same.*

The patches make up the cells. The cell, its southern and western edge, together with its south-western point are all part of the same patch. The eastern edge and south-eastern point are part of the eastern neighbour patch, the northern edge and the north-western corner point are part of the northern neighbour patch. Finally, the north-eastern point is part of the eastern neighbour of the northern neighbour patch.

If the cell of a patch exists, then the patch is called *complete*. Then also both walls of the patch exist. Of course, if a cell exists, then all walls surrounding the cell exist and hence a

complete patch always has a northern and an eastern neighbour. In its turn this neighbour may be either complete or incomplete.

If a patch is not complete, then it exists because it is needed for one or more of the following reasons:

- the southern neighbour is complete and the cell of this southern neighbour patch needs the horizontal wall of the patch as its northern edge, and the point as an end point of its northern and western edges;
- the western neighbour is complete and the cell of this western neighbour patch needs the vertical wall of the patch as its eastern edge, and the point as an end point of its eastern and southern edges;
- the western neighbour of the southern neighbour is a complete patch and the cell of this patch needs the point as an end point of its northern and eastern edges.

It follows that there is always a set of incomplete patches along the eastern and northern boundary of a domain covered by the grid.

2.4 Boundaries

The edges of a cell and hence the walls of a patch may be a part of the boundary $\partial\Omega$ of the computational domain Ω. The grid Ω^l, on level $l > 0$ does not necessarily cover the complete domain Ω. The part of the boundary of the subdomain covered by Ω^l, which does not coincide with $\partial\Omega$, is called the *green boundary*. The walls of a patch which resides on a level $l > 0$, may be part of such a green boundary.

2.5 Data contents

To each patch a number of data is associated, which together make up the data contents of the data structure. These are data concerning the walls, the point, or the cell of the patch.

Data concerning the walls are for example the lengths of the walls or their directions with respect to some reference direction. Data concerning the point of the patch are for example, its physical coordinates, or the solution components and right-hand sides in a vertex centered method. Data concerning the cell are for example its area or, in a cell centered method, the solution components, etc.

2.6 Storage

Here we describe how in the FORTRAN implementation the data in the data structure are stored. Upon creation, each patch is given an (arbitrary) identification number. All data concerning a patch are referenced through that number. All data are stored in only three large arrays: an integer array, a Boolean array and a real array.

- The integer array is called 'PNTR' (PoiNTeR). It contains the integer data for each patch. These data are mainly the patch numbers of the related patches: its parent, kids and direct neighbours. In the same array three more integer data are stored for each patch: its topological location. This location is given by the two integer coordinates (i, j) and the level l on which the patch resides.
- In the Boolean array, which is called 'PPTY' (ProPerTY), for each patch a number of *properties* are stored. These properties indicate whether the cell and the walls of a patch exist. They also indicate, whether the walls and the point of the patch are part of the

boundary, or maybe a green boundary, and whether the cell borders the boundary or a green boundary. Furthermore, there are two Booleans which indicate whether the cell of a patch needs to be refined or maybe removed from the system, at the next earliest occasion. Finally, there is one Boolean which indicates whether data locations in the arrays are assigned to a patch, or whether they are still unused and may be used by a newly created patch.

- The real data array called 'DATA', contains the numerical data which are used and/or changed during the actual computation. For each patch real data concerning (among other quantities) the geometry, the solution and the right-hand sides of the discrete equations are stored.

These arrays are passed to a subroutine through the parameter list. They are declared as 2-dimensional arrays. The first index in the array determines *which data* are referenced and the second index determines *which patch* is referenced. If, for example, we want to know the patch number of the north-eastern kid of a patch with number 'p', we can find this by reading the data stored in 'PNTR(NE, p)'. The second index determines to which patch the data belong—which in this case is patch 'p'—and the first index determines what datum of the patch 'p' is meant—which in this case is the NE-kid (assumed the proper value has been assigned to 'NE'). For the other arrays this procedure is exactly the same. In the implementation *named constants* are used for the first indices (such as 'NE' in our example). These named constants and their values are fixed and can be found in [1].

2.7 The data structure used

In EULER, the computational module of the adaptive multigrid Euler code, subroutines are provided which act on the data in the data structure, in order to solve the discretized Euler equations on a locally refined grid. The module BASIS provides subroutines to set up and maintain the data structure. How to use BASIS, is described in [1].

The subroutines in EULER assume that a data structure has been set up and that one or more *full* levels (i.e. levels with grids covering all of Ω) have been generated. This can all be done by using routines from BASIS. In EULER only three subroutines from BASIS are called directly.

Two of these are the subroutines which generate or remove the four kids of a cell. They are called 'MkOfsp' (MaKeOFfSPring), the subroutine which generates the four kids of a cell on the next higher level, and 'RmOfsp' (ReMoveOFfSPring), the subroutine removing the four kids of a cell. The former is used each time when new refinements are made.

The third subroutine from BASIS that is called in EULER is the *scanning* routine 'Scan'. This subroutine visits all or part of the patches of the tree. On each patch visited an action can be taken. When this subroutine is called, the caller can specify:

- which (sub-)tree must be scanned;
- the order of scanning;
- on which levels action must be taken;
- which action must be taken.

The subroutine 'Scan' is used in almost every task performed by the EULER subroutines. The action is defined by the user of the data structure, by means of a subroutine which is passed as an actual parameter to 'Scan'. Such a subroutine performs a subtask for each patch visited. The tree is searched from root to top. In [1] 'Scan' is given in pseudo-code.

In the case that the actual subroutine given as an argument to 'Scan' and the subroutine that calls 'Scan', need to share more data than can be passed through the parameter list of the subroutines, these data are placed in a named common block. Such a common block is declared only in the actual subroutine and the calling subroutine.

3 The Euler solver

3.1 Discretization method

Here we describe the method by Hemker and Spekreijse [2], on which the adaptive Euler solver is based. Both, the discretization and the multigrid solution process are given.

As usual for finite volume methods, the Euler equations are discretized in integral form

$$\oint_{\partial\Omega^*} (f(q)\cos\phi + g(q)\sin\phi)\, ds = 0. \tag{1}$$

The state vector q is given by $q = (\rho,\ \rho u,\ \rho v,\ \rho E)^T$ and the functions $f(q)$ and $g(q)$ are given by $f(q) = (\rho u,\ \rho u^2 + p,\ \rho uv,\ \rho uH)^T$ and $g(q) = (\rho v,\ \rho uv,\ \rho v^2 + p,\ \rho vH)^T$, where u and v are the velocity component in x- and y-direction respectively, ρ the density, p the pressure, E the specific total energy and H the specific total enthalpy, given by $H = E + p/\rho$. For a perfect gas we have $E = p/\rho(\gamma - 1) + \frac{1}{2}(u^2 + v^2)$, where γ is the (constant) ratio of specific heats. The domain Ω^* is an arbitrary subdomain of the computational domain Ω, $\partial\Omega^*$ the boundary of Ω^*, and $\cos\phi$ and $\sin\phi$ the x- and y-component of the outward unit normal on $\partial\Omega^*$.

The discretization is obtained by subdividing Ω, into disjunct, non-overlapping subdomains $\Omega_{i,j}$ (the finite volumes) and by requiring that

$$\oint_{\partial\Omega_{i,j}} (f(q)\cos\phi + g(q)\sin\phi)\, ds = 0, \quad \forall i, j. \tag{2}$$

Using the rotational invariance of the Euler equations

$$f(q)\cos\phi + g(q)\sin\phi = T^{-1}(\phi)\, f(T(\phi)q), \tag{3}$$

where $T(\phi)$ is the rotation matrix

$$T(\phi) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & \sin\phi & 0 \\ 0 & -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{4}$$

equation (2) can be rewritten as

$$\oint_{\partial\Omega_{i,j}} T^{-1}(\phi) f(T(\phi)q)\, ds = 0. \tag{5}$$

As finite volumes $\Omega_{i,j}$, arbitrarily shaped quadrilaterals are considered, the subdivision being such that $\Omega_{i\pm1,j}$ and $\Omega_{i,j\pm1}$ are the neighbouring volumes of $\Omega_{i,j}$.

In this discretization a proper evaluation of the flux vector along $\partial\Omega_{i,j}$ is crucial for the success of the multigrid technique. Following the Godunov approach, along each cell face the flux vector is assumed to be constant, and to be determined by a uniformly constant left and right state, q^l and q^r, only. Hence (5) becomes

$$
\begin{aligned}
&\mathcal{F}_{i,j}(q_{i,j}) \equiv \\
&\quad T^{-1}(\phi_{i+1/2,j})F(T(\phi_{i+1/2,j})q^l_{i+1/2,j},\, T(\phi_{i+1/2,j})q^r_{i+1/2,j})l_{i+1/2,j} - \\
&\quad T^{-1}(\phi_{i-1/2,j})F(T(\phi_{i-1/2,j})q^l_{i-1/2,j},\, T(\phi_{i-1/2,j})q^r_{i-1/2,j})l_{i-1/2,j} + \\
&\quad T^{-1}(\phi_{i,j+1/2})F(T(\phi_{i,j+1/2})q^l_{i,j+1/2},\, T(\phi_{i,j+1/2})q^r_{i,j+1/2})l_{i,j+1/2} - \\
&\quad T^{-1}(\phi_{i,j-1/2})F(T(\phi_{i,j-1/2})q^l_{i,j-1/2},\, T(\phi_{i,j-1/2})q^r_{i,j-1/2})l_{i,j-1/2} = 0,
\end{aligned} \tag{6}
$$

where for example $F(T(\phi_{i+1/2,j})q^l_{i+1/2,j},\, T(\phi_{i+1/2,j})q^r_{i+1/2,j})$ represents the transport of mass, momentum and energy (per unit length and time) across $\partial\Omega_{i+1/2,j}$, the cell face between $\Omega_{i,j}$ and $\Omega_{i+1,j}$, where $l_{i+1/2,j}$ denotes the length of $\partial\Omega_{i+1/2,j}$, and $\phi_{i+1/2,j}$ the angle between the normal on $\partial\Omega_{i+1/2.j}$, pointing from $\Omega_{i,j}$ to $\Omega_{i+1,j}$, and the positive x-direction.

In the Godunov approach the flux evaluation is identical to the solution of a one-dimensional Riemann problem. Instead, an approximate Riemann solver is applied, which is the P-variant [2] of Osher's scheme. Further, the fluxes are conceived as functions of u, v, c (local speed of sound) and $z \equiv \ln(p\rho^{-\gamma})$ (entropy), leading to simple algebraic relations for the Riemann invariants.

The evaluation of a flux across a boundary wall is consistent with the flux evaluation across an internal wall. E.g. for a boundary wall $\partial\Omega_{i-1/2,j}$ on the left side of the computational domain we, have the flux function $F(\tilde{q}_B(\tilde{q}^r_{i-1/2,j}),\, \tilde{q}^r_{i-1/2,j})$, where $\tilde{q}^r_{i-1/2,j} = T(\phi_{i-1/2,j})q^r_{i-1/2,j}$. The relation $\tilde{q}_B(\tilde{q}^r_{i-1/2,j})$ (the boundary state) is determined by the type of boundary condition to be applied: different for sub-/supersonic in-/outflow or solid wall boundary conditions.

Osher's approximate Riemann solver and the boundary condition treatment are continuously differentiable. This property is exploited in the solution method.

The choice of the left and right states, such as $q^l_{i+1/2,j}$ and $q^r_{i+1/2,j}$, determines the accuracy of the discretization. First-order accuracy is obtained by simply taking

$$
\begin{aligned}
q^l_{i+1/2,j} &= q_{i,j}, \\
q^r_{i+1/2,j} &= q_{i+1,j},
\end{aligned} \tag{7}
$$

where $q_{i,j}$ and $q_{i+1,j}$ are the cell-centered states in $\Omega_{i,j}$ and $\Omega_{i+1,j}$, respectively. In spite of its suitability for multigrid, two severe drawbacks of the first-order accurate discretization are

- its need for relatively fine grids in smooth flow regions, and
- its strong smearing of discontinuities that are not aligned with the grid.

As a remedy against these drawbacks (besides the use of a higher-order discretization) the present Euler solver on adaptively refined grids has been developed.

3.2 Solution method

The solution method for the first-order discretized Euler equations uses a multigrid technique. As the smoothing technique for the discrete equations collective symmetric point Gauss-Seidel relaxation is applied. *Point* refers to the property that during the update of the state vector $q_{i,j}$ all other state vectors are kept fixed. *Collective* refers to the property that the update of $q_{i,j}$ is done for all of its four components simultaneously. Further, *symmetric* means that after a relaxation sweep, i.e. after an update of all state vectors $q_{i,j}$ in a given order, a new sweep is made with the reverse ordering. At each volume visited during a relaxation sweep the system of four nonlinear equations (6) is approximately solved by Newton iteration, the differential

operator applied being $(\partial/\partial u,\ \partial/\partial v,\ \partial/\partial c,\ \partial/\partial z)^T$. The most efficient relaxation is obtained by selecting a large tolerance for the Newton iteration, so that in all but exceptional cases only a single Newton step is needed. All derivatives that are needed for the relaxation method are clearly listed in [3]. The relaxation method mentioned is simple and robust and for the first-order upwind discretized equations considered, it has good smoothing properties.

For the multigrid acceleration the nonlinear approach (FAS) is applied, preceded by nested iteration (FMG). To apply multigrid, a nested set of grids is constructed, such that a finite volume on a coarse grid is the union of 2×2 volumes on the next finer grid. In the original method only grids completely covering Ω are considered, whereas in the present case a fine grid may cover only a subdomain of the domain covered by the next coarser grid.

Let $\Omega^0, \ldots, \Omega^{l-1}, \Omega^l, \Omega^{l+1}, \ldots, \Omega^L$ be a sequence of such nested grids, with Ω^0 the coarsest and Ω^L the finest grid. Then, nested iteration is applied to obtain a good initial solution on Ω^L, whereas nonlinear multigrid is applied to compute q^L, the solution on Ω^L. The first iterate for the nonlinear multigrid cycling is the solution obtained by nested iteration.

The nested iteration starts with a user-defined initial estimate of q^0, the solution on the coarsest grid Ω^0. To obtain an initial solution on a finer grid Ω^{l+1}, first the solution on the coarser grid Ω^l is improved by a single nonlinear multigrid cycle. Hereafter, this solution is interpolated to the finer grid Ω^{l+1}. These steps are repeated until the highest level (finest grid) is reached. The interpolation of a solution on Ω^l used to obtain the initial solution on a grid Ω^{l+1}, is bilinear.

Let $N^l q^l = 0$ denote the nonlinear system of first-order discretized equations on Ω^l. Then a single nonlinear multigrid cycle is recurrently defined by the following steps:

1. Improve on Ω^l the latest obtained solution q^l by application of p pre-relaxation sweeps.
2. Compute on the next coarser grid Ω^{l-1} the right-hand side $r^{l-1} = N^{l-1}q^{l-1} - I_l^{l-1} N^l q^l$, where q^{l-1} may be an earlier obtained solution on level $l-1$ or a restriction of the fine grid solution, and where I_l^{l-1} is a restriction operator for right-hand sides.
3. Approximate the solution of $N^{l-1}q^{l-1} = r^{l-1}$ by the application of σ nonlinear multigrid cycles.
4. Correct the current solution by $q^l := q^l + \tilde{I}_{l-1}^l (q^{l-1} - q_0^{l-1})$, where $\tilde{I}_{l-1}^l$ is a prolongation operator for solutions, and q_0^{l-1} a previously obtained solution on level $l-1$.
5. Improve again q^l by application of q post-relaxations.

Steps 2, 3 and 4 form the so-called coarse grid correction. These three steps are skipped on the coarsest grid. In general the efficiency of a coarse grid correction depends on the coarseness of the coarsest grid. (Generally the efficiency increases with the number of grids used.) The restriction operator I_l^{l-1} and the prolongation operator $\tilde{I}_{l-1}^l$ are defined by

$$r_{i,j}^{l-1} = (I_l^{l-1} r^l)_{i,j} \equiv r_{2i,2j}^l + r_{2i+1,2j}^l + r_{2i,2j+1}^l + r_{2i+1,2j+1}^l, \tag{8}$$

$$\begin{aligned} (\tilde{I}_{l-1}^l q^{l-1})_{2i,2j} &= (\tilde{I}_{l-1}^l q^{l-1})_{2i+1,2j} = \\ (\tilde{I}_{l-1}^l q^{l-1})_{2i,2j+1} &= (\tilde{I}_{l-1}^l q^{l-1})_{2i+1,2j+1} \equiv q_{i,j}^{l-1}. \end{aligned} \tag{9}$$

Defining the grid transfer operators in this way, it can be verified that

$$N^{l-1}(q^{l-1}) = I_l^{l-1} N^l (\tilde{I}_{l-1}^l q^{l-1}), \tag{10}$$

i.e. a coarse grid discretization of the Euler equations is a Galerkin approximation of the discretization on the next finer grid. This implies that the coarse grid correction efficiently reduces the short-wave-length components in the defect.

When we use $\sigma = 1$ we have V-cycles with p pre- and q post-relaxations per level. When we use $\sigma = 2$ we have W-cycles with p and q pre- and post-relaxations per level.

4 The data structure and Euler solver

4.1 The sequence of grids

At some stage in the adaptive multigrid computation a sequence of grids $\Omega^0, \ldots, \Omega^L$ has been generated. A grid Ω^l, $l > 0$, does not necessarily cover all of the domain Ω. Therefore, the grid Ω^l consists of a part Ω^l_f of refined cells (with kids residing on level $l+1$) and a part Ω^l_c which is not refined (does not have kids).

Remark 2 *The grid on a level l, $0 \leq l \leq L$ consists of a refined and a non-refined part*

$$\Omega^l = \Omega^l_f \cup \Omega^l_c, \tag{11}$$

where Ω^l_c, $0 \leq l < L$ may be empty and where Ω^L_f is empty.

Remark 3 *The grids Ω^l_f and Ω^{l+1}, $0 \leq l < L$, cover the same part of the computational domain Ω.*

Definition 4 (Composite grid) *For a sequence of grids $\Omega^0, \ldots, \Omega^L$ the* composite grid *is*

$$\Omega_c = \cup_{l=0,\ldots,L} \Omega^l_c. \tag{12}$$

Remark 5 *In the adaptive computations we are interested in the solution on the composite grid Ω_c.*

4.2 Addressing data

Here we describe the use of the data structure for the Euler solver. All subprograms that 'read from' or 'write into' the data structure, have the three data structure arrays (PNTR, PPTY, DATA) in their parameter list. Their dimensions are declared either as named constants (FstPtr, LstPtr, FstPpt, LstPpt, see [1]), or they are declared as integer variables in the common block 'DatGlb' (MNOP, MNOD, see [1]).

To solve the discretized Euler equations by the multigrid technique, for each patch we need storage for the following real data: twelve numbers for respectively the right-hand sides of the equations, r^l, the current solution q^l and a previous solution q^l_0 of the FAS algorithm, and six numbers for the geometric data of the grid. The geometric data stored are the components of the unit normal vectors and the lengths of the horizontal (H-) and vertical (V-) wall.

The real or double precision array 'DATA' is used to keep these real data. For each patch space is reserved and referenced as:

Array element	**referring to**
DATA(PRhs1, patch)	right-hand side first equation (conservation of mass);
DATA(PRhs2, patch)	right-hand side second equation (conservation of x-momentum);
DATA(PRhs3, patch)	right-hand side third equation (conservation of y-momentum);
DATA(PRhs4, patch)	right-hand side fourth equation (conservation of energy);
DATA(PQ1, patch)	first component of solution (usually u);
DATA(PQ2, patch)	second component of solution (usually v);
DATA(PQ3, patch)	third component of solution (usually c);
DATA(PQ4, patch)	fourth component of solution (usually z);
DATA(PQold1, patch)	first component old solution;
DATA(PQold2, patch)	second component old solution;
DATA(PQold3, patch)	third component old solution;
DATA(PQold4, patch)	fourth component old solution;

DATA(PSiH, patch)	y-component unit normal on H-wall;
DATA(PCoH, patch)	x-component unit normal on H-wall;
DATA(PDsH, patch)	length H-wall;
DATA(PSiV, patch)	y-component unit normal on V-wall;
DATA(PCoV, patch)	x-component unit normal on V-wall;
DATA(PDsV, patch)	length V-wall.

Remark 6 *Whereas the ordering of the real data for a patch in the array has no intrinsic meaning, the actual implementation is made by named integer constants* [1].

4.3 Description of the code

4.3.1 Initialization of technical data

Before any computation can be done, an initialization needs to be made. The technical subroutines of the EULER module use a number of constants which only depend on the ratio of specific heats γ. These constants are stored in a named common block 'gammas'. Their values are initialized by making a call to the subroutine 'GamIni' (GAMmaINItialization), which is declared as

```
subroutine        GamIni(gamma)
double precision gamma
```

Description of variables:

Input

gamma constant defining the ratio of specific heats γ, which for air usually is $\gamma = 1.4$.

The common block 'gammas' must be declared in the main program to contain twelve double precision reals.

4.3.2 Construction of geometric data

The geometric data used in the finite volume discretization, consist of the lengths of the cell faces and the components of the unit normal on the cell faces. The user of the module EULER can construct the geometric data by making a call to the subroutine 'MkGeo' (MaKeGEOmetricdata). This subroutine is declared as

```
subroutine MkGeo(lev, PNTR, PPTY, DATA)
integer    lev
```

Description of variables:

Input

lev (LEVel) geometric data of the grid on level *lev* are constructed.

The construction of the geometric data requires the physical coordinates of the grid points of the grid covering the physical domain. The mapping of the topological coordinates of $P^l_{i,j}$ to the physical coordinates (x, y) is problem-dependent. Therefore, the user of the code has to specify a subroutine, named 'GetXY', which delivers the physical coordinates for given integer coordinates. This subroutine should be declared as

[1] PRhs1 = 1, PRhs2 = 2, PRhs3 = 3, PRhs4 = 4, PQ1 = 5, PQ2 = 6, PQ3 = 7, PQ4 = 8, PQold1 = 9, PQold2 = 10, PQold3 = 11, PQold4 = 12, PSiH = 13, PCoH = 14, PDsH = 15, PSiV = 16, PCoV = 17, PDsV = 18.

```
subroutine       GetXY(i, j, lev, x, y)
integer          i, j, lev
double precision x,y
```

Description of variables:

Input

i i-coordinate of the grid point $P_{i,j}^l$;

j j-coordinate of the grid point $P_{i,j}^l$;

lev (LEVel) level l on which the grid point $P_{i,j}^l$ resides;

Output

x physical x-coordinate, associated with $P_{i,j}^l$;

y physical y-coordinate, associated with $P_{i,j}^l$.

Notice that the quantities $\xi = i2^{-l}$ and $\eta = j2^{-l}$ determine the topological coordinates of a point in the computational domain, independent of the level l.

4.3.3 The FMG and FAS-algorithm

The nested iteration algorithm FMG and nonlinear multigrid algorithm FAS are described in pseudo-code in [1]. In the EULER module these algorithms are implemented by the subroutines 'FMG' and 'FAS'. The user of the code can make a call to the subroutine 'FMG' after the geometric data have been provided for all levels available in the data structure and the solution has been initialized on the coarsest grid (level zero). Initialization of the solution is done by calling the subroutine 'SolIni' (SOLutionINItialization), declared as

```
subroutine       SolIni(lev, q1, q2, q3, q4, PNTR, PPTY, DATA)
integer          lev
double precision q1, q2, q3, q4
```

Description of variables:

Input

lev (LEVel) the solution is initialized on level *lev*;

q1–q4 the solution is initialized with the constants *q1, q2, q3, q4*.

This subroutine uses a named common block called 'IniC', which is invisible to the user. Since the physical quantities are expressed in physical, Cartesian coordinates, in this way a uniform flow is the initial solution. Next, the FMG-algorithm can be performed, which computes solutions on the grids from level zero to the finest level, all starting from the initial solution on level zero. The solutions can be further improved by the FAS-algorithm. The subroutines 'FMG' and 'FAS' are declared as

```
 subroutine FMG(TopLev, nfas, npmg, nqmg, ncycl, ff,
+               PNTR, PPTY, DATA
 integer    TopLev, nfas(0:TopLev), npmg(0:TopLev),
+           nqmg(0:TopLev), ncycl(0:TopLev), ff
```

```
 subroutine FAS(BtmLev, TopLev, npmg, nqmg, ncycl, ff,
+               PNTR, PPTY, DATA)
 integer    BtmLev, TopLev, npmg(0:TopLev), nqmg(0:TopLev),
+           ncycl(0:TopLev), ff
```

Description of variables:

Input

BtmLev (BoTtoMLEVel) level of the coarsest grid in the sequence of grids used in the multigrid algorithm (usually equal zero);

TopLev (TOPLEVel) level of the finest grid in the sequence of grids used in the multigrid algorithm;

nfas (NumberofFAS) the number of multigrid cycles to be performed on each level in the FMG algorithm;

npmg number of pre-relaxations on each level;

nqmg number of post-relaxations on each level;

ncycl (NumberofCYCLes) type of multigrid cycle to be performed
ncycl(i) = 1 : V-cycle;
ncycl(i) = 2 : W-cycle;

ff extra parameter to indicate the multigrid cycle to be used
ff = 0: V- or W-cycle;
ff = 1: V-F- or W-F-cycle.

4.3.4 Relaxation

As smoother for the multigrid process point Gauss-Seidel relaxation is used. For this purpose the cells on a level are visited in some given ordering and in each cell $\Omega^l_{i,j}$ visited, the solution is updated by the (approximate) solution of eqn. (6). The performance of the point Gauss-Seidel relaxation can be strongly dependent of the ordering. Generally, we want the ordering to be such that each cell is visited after two of its neighbours already have been visited (except maybe for a boundary cell). The subroutine 'Scan' visits the cells on a level in such an order. This allows 'Scan' to be used for making a relaxation sweep over the grid on a level. The actual task to be performed when a cell is visited, is 'solve eqn. (6)'. The subroutine doing this is passed as an actual parameter to the routine 'Scan', which itself is called by the relaxation routine.

The direction of the sweep over the grid is determined by the ordering used by 'Scan' to scan the tree. The visiting order of 'Scan' is determined by an integer array of length four, passed to 'Scan' through its parameter list. This array contains a permutation of the wind directions NE, SE, SW, NW. (See [1] for their definitions.) The effective sweep directions are given in table 1, where we assume that this array is called 'Order'.

The relaxation algorithm needs some further strategy parameters, set by the user. These parameters determine whether the relaxation sweep must be symmetric, and when the Newton iteration solving eqn. (6), must be stopped. The Newton iterations may be stopped either because the residual of eqn. (6) becomes smaller than some criterion, or when a maximum number of iterations is exceeded. In order to supply the relaxation with these parameters and the relaxation order, the relaxation subroutine uses a named common block. This common

	Order(1 : 4)	**sweep direction**
	(SW, SE, NW, NE)	from SW to NE corner
or	(SW, NW, SE, NE)	
	(NE, SE, NW, SW)	from NE to SW corner
or	(NE, NW, SE, SW)	
	(SE, NE, SW, NW)	from SE to NW corner
or	(SE, SW, NE, NW)	
	(NW, NE, SW, SE)	from NW to SE corner
or	(NW, SW, NE, SE)	

Table 1: The use of the ordering array to be passed to 'Scan'.

block is initialized with default values by a call to the subroutine 'RelIni' (RELaxationINItialization). The common block is called 'strat' (STRATegy) and must be declared in the main program as

```
integer          MaxNwt, RelOrd(4)
double precision RelTol
logical          symm
common /strat/   RelTol, MaxNwt, RelOrd, symm
```

Description of variables:

RelTol (RELaxationTOLerance) the Newton iteration is stopped when the sum of absolute values of the components of the residual vector becomes smaller than *RelTol*;

MaxNwt (MAXimumNeWTon) maximum number of Newton iterations to be performed;

RelOrd (RELaxationORDering) the relaxation ordering is defined by the array *RelOrd*, passed to 'Scan' as an actual value for the ordering array 'Order' (see table 1);

symm (SYMMetric) a symmetric relaxation sweep is made when *symm*= .true., and a single relaxation sweep when *symm*=.false.

Default values for the parameters that control the relaxation are:

Variable	**default**
RelTol	0.1
MaxNwt	13
RelOrd	(SW, NW, SE, NE)
symm	.true.

The user may change the values of these parameters any time.

4.3.5 Boundaries

In the relaxation, where the system of equations (6) is solved for each $\Omega^l_{i,j}$, fluxes across the cell faces are computed. A cell face can be any of the three possible walls: an internal wall, a boundary wall or a green wall.

The flux across an internal wall is computed from the state vectors in the adjacent cells. The flux across a boundary wall also depends on the type of boundary and on the boundary condition. These are problem-dependent and must be specified by the user of the EULER module. The user must specify a subroutine 'GetBC' (GETBoundaryCondition). This subroutine delivers the boundary condition and the type of boundary. It is defined as

```
subroutine        GetBC(i, j, lev, hor, BdyTp, QB)
integer           i, j, lev
double precision  QB(4)
logical           hor
```

Description of variables:

Input

i i-coordinate of the point $P^l_{i,j}$, the left end point of the horizontal or vertical wall (see *hor*);

j j-coordinate of the point $P^l_{i,j}$, the left end point of the horizontal or vertical wall (see *hor*);

lev (LEVel) the level l of $P^l_{i,j}$, the left end point of the horizontal or vertical wall (see *hor*);

hor (HORizontal) the boundary condition is requested for the horizontal or vertical wall
hor = .true. : horizontal wall
hor = .false. : vertical wall

Output

BdyTp (BounDarYTyPe) type of boundary parameter (see table 2);

QB boundary condition state vector (as far as it needs to be specified).

Upon leaving the subroutine, the type of boundary parameter 'BdyTp' and the array 'QB', should contain one of the combinations given in table 2. Here p denotes the pressure, u and v the velocity components (in physical x- and y-direction respectively), c the speed of sound and z the entropy function (see section 3.1).

BdyTp	QB(1)	QB(2)	QB(3)	QB(4)	type of boundary
0	–	–	–	–	supersonic outflow
1	p	–	–	–	subsonic outflow
2	u	v	z	–	subsonic inflow
3	u	v	c	–	subsonic inflow
4	u	v	c	z	supersonic inflow
5	–	–	–	–	solid wall

Table 2: Boundary conditions and parameters to be set by 'GetBC'.

4.3.6 Prolongation and restriction (right-hand side)

In the FMG-algorithm the fine gird solution on Ω^l is found from bilinear interpolation of the solution on the coarse grid Ω^{l-1}. The interpolation is made by a call to 'Scan' to visit all cells on the coarse level. For each cell the solutions in the kid cells are constructed by bilinear interpolation of the solution in the four coarse grid cells nearest to the kid cell.

In the FAS-algorithm the correction of the solution, $q_{i,j}^{l-1} - (q_0)_{i,j}^{l-1}$, in the coarse grid cell $\Omega_{i,j}^{l-1}$ is prolongated to the finer grid by a piecewise constant interpolation. Similar to the interpolation of the solution in the FMG-algorithm, the *prolongation* of the correction is made by a 'Scan' over the coarse grid. In each coarse grid cell the correction is computed and added to the solution in its kid cells.

The *restriction* of the fine grid defect is computed as soon as the coarse grid right-hand side is needed. In a cell $\Omega_{i,j}^l \in \Omega_f^l$ the right-hand side is

$$r_{i,j}^l = (N^l q^l)_{i,j} - (I_{l+1}^l (N^{l+1} q^{l+1} - r^{l+1}))_{i,j},$$

and in a cell $\Omega_{i,j}^l \in \Omega_c^l$ the right-hand side is

$$r_{i,j}^l = s_{i,j}^l.$$

In steady Euler flow computations we usually have for the source term

$$s_{i,j}^l = 0.$$

The coarse grid right-hand side in $\Omega_{i,j}^l \in \Omega_f^l$ is a summation of fluxes across coarse grid and fine grid walls, and of the fine grid right-hand sides. The right-hand side in $\Omega_{i,j}^l \in \Omega_c^l$ however, requires the evaluation of the source term $s_{i,j}^l$.

The right-hand sides are constructed in a 'Scan' over the patches that make up the coarse grid. In each patch visited, actions are undertaken to calculate fluxes and to send them to the appropriate memory locations reserved in 'DATA' for the right-hand sides of the cell and/or its neighbours. Then also the right-hand sides of the kids are added, or when there are no kids, the source term is evaluated. This is all done in the subroutine 'MkRhsP' (MaKeRightHandSidePatch) which is passed as an actual argument to 'Scan'. The subroutine calling 'Scan' with actual argument 'MkRhsP' communicate through a named common block called 'RhsC'. This common block is invisible to the user. The sending around of the fluxes is such that the right-hand sides are constructed, using a minimal number of flux evaluations.

Each time when in the FAS-cycle the problem on a level is approximately solved, the right-hand sides for that level are computed. This allows the possible computation with a changing source term (i.e. depending on the solution) on Ω_c. The user should then adapt the subroutine 'MkRhsP' to his own needs.

Usually for Euler flow computations the source term will be zero. In order to allow possible applications of the code to compute solutions of the Euler equations with a source term (i.e. a source term that is independent of the solution), the user must provide a subroutine which delivers the source terms. The source term $s_{i.j}^l$ to be delivered for a cell $\Omega_{i,j}^l$, should be an approximation of the source, integrated over the volume (area) $\Omega_{i,j}^l$. The subroutine has to be provided, even when the source term is zero. The subroutine is called 'GetSrc' (GETSouRCe) and is declared as

```
subroutine        GetSrc(i, j, lev, s)
integer           i, j, lev
double precision  s(4)
```

Description of variables:

Input

i i-coordinate of the cell $\Omega_{i,j}^l$;

j j-coordinate of the cell $\Omega_{i,j}^l$;

lev (LEVel) level l on which the cell $\Omega_{i,j}^l$ resides;

Output

s the source term vector in cell $\Omega_{i,j}^l$ (integrated over the volume).

4.3.7 Residual

From time to time a user may want to check the convergence behaviour of the computation. For this purpose, the code is supplied with a subroutine which computes the residual of an iterate for the discrete equations. Since we are only interested in the solution on the composite grid Ω_c, the residual is computed only for $\Omega_{i,j}^l \in \Omega_c$. The residual is given by

$$R_{i,j}^l = s_{i,j}^l - (N^l q^l)_{i,j}.$$

The subroutine that computes the residual, first delivers the residual for each cell separately to the memory locations reserved for the right-hand sides of the equations of the cells of Ω_c. The components of $(N^l q^l)_{i,j}$ are sums of fluxes across the cell faces of $\Omega_{i,j}^l$. The field of residuals is constructed by making a 'Scan' through the data structure, for each patch computing fluxes when necessary and adding or subtracting them from the appropriate right-hand sides. In this way a minimal number of flux evaluations is needed to construct the residual field.

After the residual has been computed, another 'Scan' through the data structure is made, to construct the L_1- and L_∞-norm of the residual field on Ω_c. Before residuals are summed (L_1-norm) or compared (L_∞-norm), the residual is weighted by the volume (area) of the cell. Since we have for the area $A_{i,j}^l$ of a cell $\Omega_{i,j}^l$

$$A_{2i,2j}^l \approx A_{2i+1,2j}^l \approx A_{2i,2j+1}^l \approx A_{2i+1,2j+1}^l \approx \frac{1}{4} A_{i,j}^{l-1},$$

the weighting is done by a multiplication of the residual with a factor 4^{-l}, where l is the level on which the cell resides. (Notice that the computed residual is dependent on the non-uniformity of the mesh. Nevertheless, a proper measure for the convergence of the computational process is obtained.) In this way we obtain the L_1-norm and L_∞-norm for the four components of the residual. In addition to these norms the four L_1-norms are averaged to deliver a 'mean residual' and the maximum of the four L_∞-norms is delivered as a 'maximum residual'.

The residual computation is performed by the subroutine called 'Res' (RESidual). This subroutine may be called by the user, whenever the right-hand side data are no longer needed (e.g. after each FAS-cycle). The subroutine 'Res' is defined as

```
subroutine        Res(MaxLev, RMean, RMax, PNTR, PPTY, DATA)
integer           MaxLev
double precision RMean(0:4), RMax(0:4)
```

Description of variables:

Input

MaxLev (MAXimumLEVel) a value greater then or equal to the highest level in the sequence of grids available at the moment 'Res' is called;

Output

RMean(0) (ResidualMEAN) the mean residual, i.e. the mean value of the L_1-norms of the residual over Ω_c;

RMean(i) L_1-norm of the i-th component of the residual over Ω_c;

RMax(0) (ResidualMAXimum) the maximum residual, i.e. the maximum of the L_∞-norms of the residual over Ω_c;

RMax(i) L_∞-norm of the i-th component of the residual over Ω_c.

This subroutine uses a named common block called 'ResC', which is invisible to the user.

4.3.8 Local refinements

The code has been developed to use self-adaptive, locally refined grids. This requires a criterion to decide whether a cell should be refined further. This criterion is to be specified by the user.

The construction of refinements is divided into two parts:

1. first the cells that must be refined are flagged by setting the property 'Pregnant' to .true. (see [1]);
2. next the tree is searched for flagged cells and refinements are generated.

The first task is done by the subroutine to be provided by the user. This subroutine will have to use the arrays of the data structure and probably the subroutine 'Scan' (see [1]). Based on the refinement criterion, it should mark a cell 'Pregnant', when it needs refinement. For the second task a subroutine called 'Refn' (REFiNe) is enclosed in the EULER module. This subroutine 'Scans' the tree of the data structure and at each patch visited it is checked whether the cell of the patch must be refined and/or the geometrical data of kid patches must be generated. When the cell must be refined, the solution components for the new kids are found from bilinear interpolation of the solution on the coarser level. After a call to 'Refn' the data structure has been adapted to the new situation, an initial solution in the new cells has been computed and the geometrical data for the grid have been generated. The subroutine 'Refn' is defined as:

```
subroutine Refn(BtmLev, MaxLev, PNTR, PPTY, DATA)
integer    BasLev, MaxLev
```

Description of variables:

Input

BtmLev (BoTtoMLEVel) lowest level to be scanned when searching for flagged cells;

MaxLev (MAXimumLEVel) a value greater then or equal to the highest level in the sequence of grids at the moment 'Refn' is called.

The removal of previously generated refinements is also allowed. Similarly to the generation of refinements, refined cells are removed in two steps:

1. first cells to be removed are flagged by setting the property 'Sentenced' to .true. (see [1]);
2. next the tree is 'Scanned' and for each cell visited, its kids are removed if they are all marked 'Sentenced'.

The first task should be done by a user-defined subroutine which uses the data structure arrays (see [1]). The second can be done by calling the subroutine 'UnRefn' (UNREFiNe), provided in the EULER module. This subroutine is declared as:

```
subroutine UnRefn(BtmLev, MaxLev, PNTR, PPTY, DATA)
integer    BasLev, MaxLev
```

Description of variables:

Input

BtmLev (BoTtoMLEVel) lowest level to be scanned when searching for flagged cells;

MaxLev (MAXimumLEVel) a value greater then or equal to the highest level in the sequence of grids at the moment 'Refn' is called.

After calling this subroutine the kids of a cell have been removed if all four kids were flagged as 'Sentenced', and the data structure has been adapted accordingly. The space left by the removed patches becomes reusable space and will used by new patches.

4.3.9 Summary

So far we described the subroutines which can be called by a user to perform self-adaptive multigrid computations for Euler flow problems. This description includes the subroutines available in the module, as well as the subroutines that are problem-dependent and should be provided by the user.

First we summarize the subroutines which are to be called by the user, and which are already provided in the EULER module. It is assumed that a structure has been set up already by the subroutines available in the data structure module BASIS (see [1]).

GamIni initialization of named common block 'gammas' (see section 4.3.1);

RelIni initialization of named common block 'strat' with default values (see section 4.3.4);

MkGeo construction of geometrical data of the grid (see section 4.3.2);

SolIni initialization of the solution (see section 4.3.3);

FMG the nested iteration algorithm (see section 4.3.3);

FAS the nonlinear multigrid algorithm (see section 4.3.3);

Res computation residual field and L_1- and L_∞-norms of residual field (see section 4.3.7);

Refn refining of cells flagged 'Pregnant' (see section 4.3.8);

UnRefn removal of cells flagged 'Sentenced', if all kids of the same parent are flagged (see section 4.3.8).

In case of more complex problems the user may want to write his own subroutines 'MkGeo' and 'SolIni', to introduce more complex geometries or more sophisticated initial approximations.

Next follows a summary of the subroutines to be provided by the user. We distinguish subroutines that use the data structure and those which are independent of the data structure. The subroutines that use the data structure are:

- a subroutine to mark cells 'Pregnant' (see section 4.3.8); in the example in section 5 this subroutine is called 'RFlags' (RefinementFLAGS);
- a subroutine to mark cells 'Sentenced' (see section 4.3.8).

The subroutines which *must always* be specified by the user and which do not use the data structure are:

GetXY subroutine to map the coordinates in topological space to coordinates in physical space (see section 4.3.2);

GetBC subroutine to provide the boundary conditions for the problem (see section 4.3.5);

GetSrc subroutine to provide possible source terms for the problem (see section 4.3.6).

The named common blocks that must be declared in the main program, are

gammas common block containing constants for the technical subroutines of the code (see section 4.3.1);

strat common block containing strategy parameters for the relaxation (see section 4.3.4).

Finally, we summarize the named common blocks that are used by some subroutines for the communication between the subroutine calling 'Scan' and the subroutine called by 'Scan'. These common blocks are invisible to the user. We give their names, so the user will not accidently use the names for possible user-defined common blocks.

IniC common block used in the initialization of the solution;

RhsC common block used in the computation of the right-hand side;

ResC common block used in the computation of the residual.

An example of the user-defined subroutines, summarized above, is provided in the example in the next section.

5 Example

In this last section we show an application of the code developed. The problem we consider is a shock reflection problem. We take $\Omega = [0,4] \times [0,1]$. The exact solution consists of constant fields in three areas as shown in Figure 1.
Boundary conditions for the problem are on the western edge: supersonic flow in x-direction,

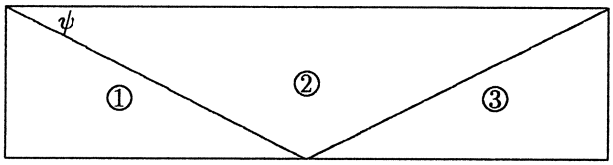

Figure 1: Example: Shock reflection. In each area $1, 2, 3$ the exact solution is constant.

with $M = 2.9$, $u = 1$ and $\rho = 1$; northern edge: subsonic inflow such that $\psi = 29°$; eastern edge: supersonic outflow; southern edge: boundary wall.

The coarsest grid is a 6×2 grid. This coarsest grid is twice uniformly refined. The resulting 24×8 grid is locally refined. A refinement cycle is made after each four FAS V-cycles. Then in this case the residual is about 10^{-4}. A cell is refined when a first undivided difference of the density with the neighbour cells is larger than 0.06. The composite grid and the grid on the levels two to five are shown in figures 2-6 respectively. Iso-level plots for the Mach number and density, computed on the composite grid of figure 2, are shown in figures 7 and 8.

The exact solution is computed by the call to subroutine 'Exact'. This solution is used to determine the correct boundary conditions in area two of the computational domain (see figure 1), such that $\psi = 29°$.

Refinement flags are set by subroutine 'RFlags'. Before 'RFlags' is called the Mach number M and density ρ are computed from the solution components u, v, c and z. This is done by subroutine 'StatSF', which delivers M and ρ to the memory locations of the first and second old solution, respectively. Therefore, the subroutine 'RFlags' is called with 'PQold2' as actual parameter, indicating that the values stored in these memory locations are used to set the refinement flags. After a call to 'StatSF' the locations indicated by 'PQold2' contain the density.

A subroutine which sets flags for the cells that may be removed, should be constructed similarly to the flag setting routine 'RFlags'.

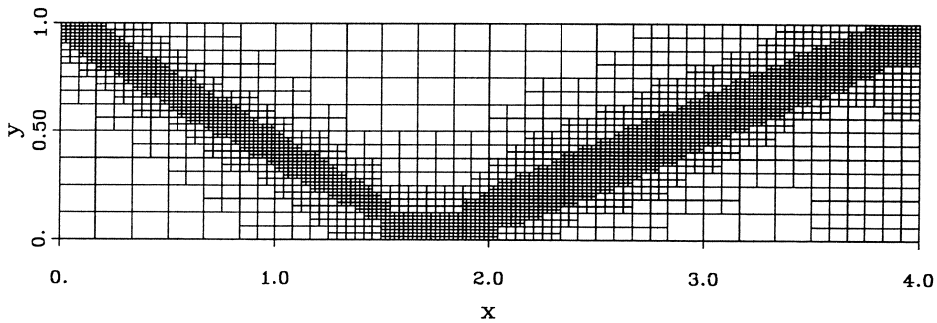

Figure 2: Self-sadaptive, locally refined grid Ω_c of example problem.

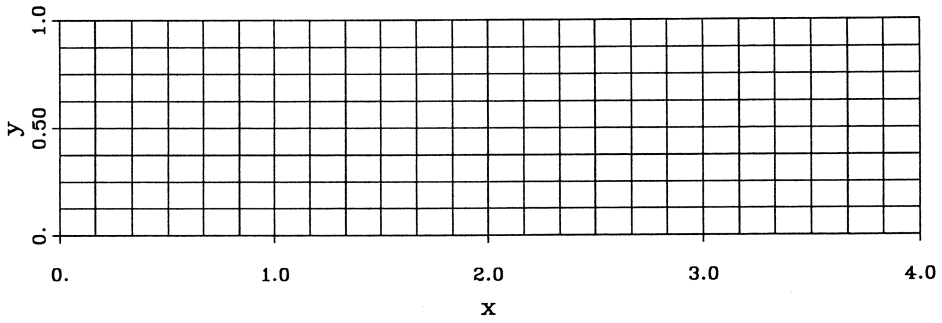

Figure 3: Grid Ω^2, the highest full level of example problem.

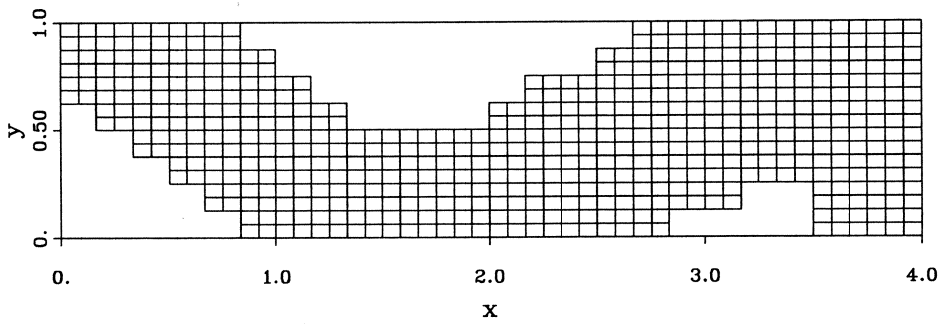

Figure 4: Grid Ω^3 of example problem.

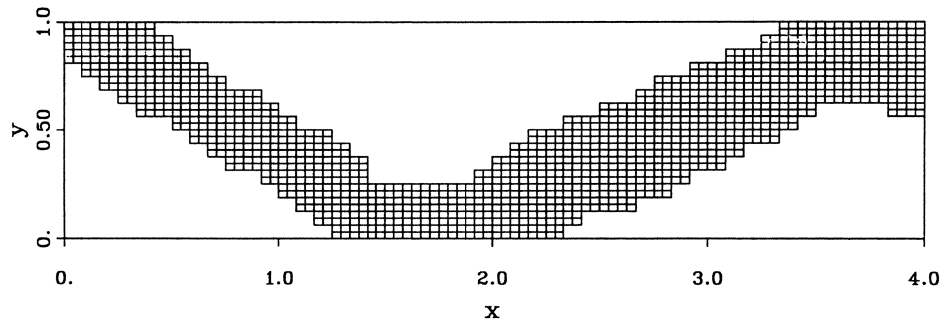

Figure 5: Grid Ω^4 of example problem.

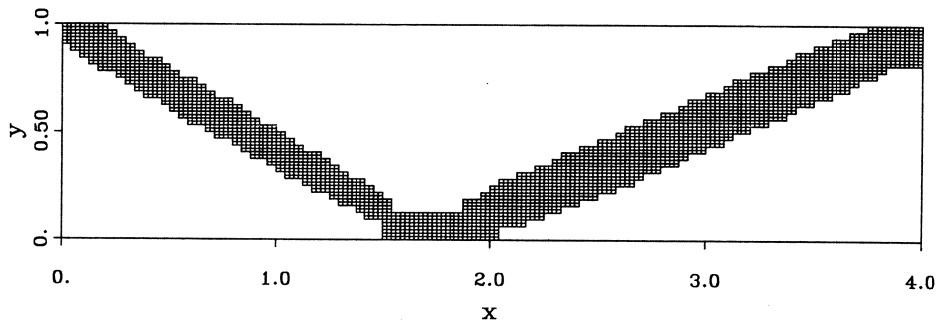

Figure 6: Grid Ω^5 of example problem.

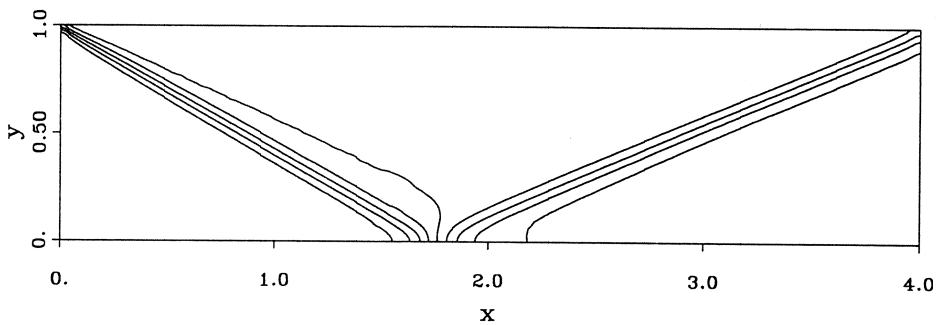

Figure 7: Mach number distribution of example problem; levels 1.9-2.9, step 0.1.

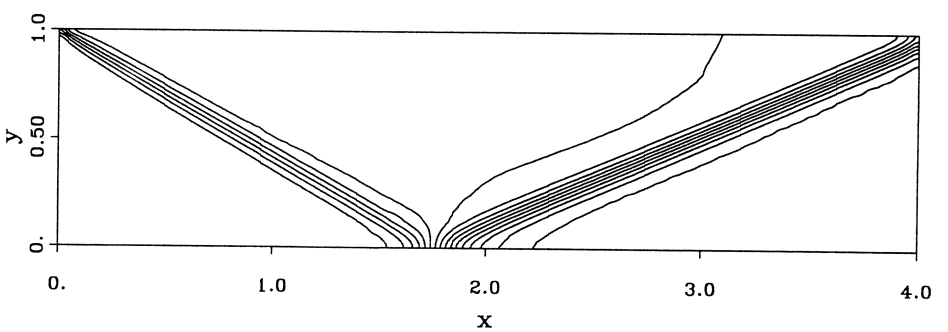

Figure 8: Density distribution of example problem; levels 1.0-2.6, step 0.1.

Here we give the listing of the main program, the user-defined subroutines for this problem and auxiliary subroutines such as the subroutine setting the refinement flags.

```
      program shrefl

c     Program to compute SHOCK REFLECTION problem

c     The size of the space claimed for the data structure is given in
c     the global variables:
c     MNOP  : Maximum Number Of Patches
c     MNOD  : Maximum Number Of real Data for a patch
c
c     The user declares the actual values by:

      integer MNOPac, MNODac
      parameter(MNOPac = 8000, MNODac = 18)

c     (where the constant values may be adapted by the user).
c     These parameter values are used in the actual declarations
c     of PNTR, PPTY and DATA. Further these parameters are communicated
c     to the data structure program through the call of the routine
c     'DatIni', where the data structure is initialized.

c     A copy of the include file 'basis.i',
c     except for the arrays PNTR, PPTY and DATA

      integer          FstPtr, LstPtr,
     +                 LV, XX, YY, PT,
     +                 NE, SE, SW, NW, NN, EE, SS, WW,
     +                 FstPpt, LstPpt,
     +                 Compl, WallH, WallV,
     +                 BdyPnt, BdyWaH, BdyWaV, BdyCel,
     +                 GrnPnt, GrnWaH, GrnWaV, GrnCel,
     +                 Prgnt, Sntncd, Dead,
     +                 Nil, RtPtr
      parameter        (FstPtr =-3, LstPtr = 8,
     +                 LV     =-3, XX     =-2, YY     =-1,  PT     = 0,
     +                 NE     = 1, SE     = 2, SW     = 3,  NW     = 4,
     +                 NN     = 5, EE     = 6, SS     = 7,  WW     = 8,
     +                 FstPpt = 1, LstPpt =14,
     +                 Compl  = 1, WallH  = 2, WallV  = 3,
     +                 BdyPnt = 4, BdyWaH = 5, BdyWaV = 6,  BdyCel = 7,
     +                 GrnPnt = 8, GrnWaH = 9, GrnWaV = 10, GrnCel = 11,
     +                 Prgnt  =12, Sntncd =13, Dead   = 14,
     +                 Nil    = 0, RtPtr  = 1)

      integer          MNOP, MNOD,
     +                 RtLv, LstSpa, NOP, SizeX0, SizeY0, NrmOrd(4)
      common /DatGlb/  MNOP, MNOD,
     +                 RtLv, LstSpa, NOP, SizeX0, SizeY0, NrmOrd

c     The arrays for the data structure

      integer          PNTR(FstPtr:LstPtr, 0:MNOPac)
      logical          PPTY(FstPpt:LstPpt, 0:MNOPac)
      double precision DATA(1:MNODac, 0:MNOPac)
```

```
c     Copy of the include file 'euler.i'

      integer   PRhs1, PRhs2, PRhs3, PRhs4,
     +          PQ1, PQ2, PQ3, PQ4,
     +          PQold1, PQold2, PQold3, PQold4,
     +          PSiH, PCoH, PDsH,
     +          PSiV, PCoV, PDsV
      parameter(PRhs1  = 1,  PRhs2  = 2,  PRhs3  = 3,  PRhs4  = 4,
     +          PQ1    = 5,  PQ2    = 6,  PQ3    = 7,  PQ4    = 8,
     +          PQold1 = 9,  PQold2 = 10, PQold3 = 11, PQold4 = 12,
     +          PSiH   = 13, PCoH   = 14, PDsH   = 15,
     +          PSiV   = 16, PCoV   = 17, PDsV   = 18)

c     Constants related to the ratio of specific heats gamma

      double precision gamma, ogam, otgam, gamm1, hgamm1, ogamm1, togm1,
     +                 gogm1, oggm1, gamp1, togp1, gmogp
      common /gammas/  gamma, ogam, otgam, gamm1, hgamm1, ogamm1, togm1,
     +                 gogm1, oggm1, gamp1, togp1, gmogp

c     Strategy parameters for the relaxation

      integer          MaxNwt, RelOrd(4)
      double precision RelTol
      logical          symm
      common /strat/   RelTol, MaxNwt, RelOrd, symm

c     The highest and lowest level

      integer          MNOL, LNOL
      parameter       (MNOL = 20, LNOL = -12)

c     Grid dimensions on level 0

      integer          nx0, ny0
      common /mesh/    nx0, ny0

c     Exact solution

      double precision u1, v1, c1, z1, u2, v2, c2, z2, u3, v3, c3, z3,
     +                 psi, delta, sigma, xr, yr
      common /exct/    u1, v1, c1, z1, u2, v2, c2, z2, u3, v3, c3, z3,
     +                 psi, delta, sigma, xr, yr

c     Remaining declarations

      integer          nfas(0:MNOL), npmg(0:MNOL), nqmg(0:MNOL),
     +                 ncycl(0:MNOL), ff,
     +                 level, nlevel, ifas, nnfas, iref, nref,
     +                 i, k, BasLev

      double precision uini, vini, cini, zini,
     +                 M1, R1, M2, R2, M3, R3, pi, RMx(0:4), RMn(0:4),
     +                 RMax(0:100), RMean(0:100), ScMx, ScMn, RefCr
```

```
      external        MkOfsp

c-----------------------------------------------------------------------

c     Main program statements

c-----------------------------------------------------------------------

      pi = 4.0d0*atan(1.0d0)

c     Grid dimensions on level 0

      nx0 = 6
      ny0 = 2

c     Number of fully refined levels and number of
c     levels initially available

      nlevel = 3

c     Highest full level available

      BasLev = nlevel - 1

c     Number of FAS-cycles in FMG-algorithm and
c     between each refinement cycle

      nnfas = 4

c     Number of refinement cycles

      nref = 3

c     Refinement criterion

      RefCr = 0.60d-1

c     FMG and FAS parameters

      do 10 i = 0, MNOL
        nfas(i)  = 1
        npmg(i)  = 1
        nqmg(i)  = 1
        ncycl(i) = 1
10    continue
      ff = 0

c     Initialization of the data structure

      call DatIni(MNOPac, MNODac, PNTR, PPTY, DATA)

c     Initialization of common block gammas

      call GamIni(1.4d0)
```

```
c     Initialization of relaxation parameters

      call RelIni

c     Construction of data structure for rectangular topology,
c     up to and including level 0

      call MkRec(nx0, ny0, PNTR, PPTY, DATA)

c     Construction of data structure up to and including 'BasLev'

      do 20 level = 0, BasLev - 1
        call Scan(RtPtr, NrmOrd, level, level, MkOfsp, PNTR, PPTY, DATA)
   20 continue

c     Problem definition

c     Incoming flow

      u1  = 1.0d0
      v1  = 0.0d0
      M1  = 2.9d0
      R1  = 1.0d0

c     Incoming shock angle

      psi = 29.0d0*pi/180.0d0

c     Computation exact solution

      write (6, '(a)') ' Exact solution:'
      call Exact(psi, u1, v1, M1, R1, u2, v2, M2, R2,
     +           u3, v3, M3, R3, delta, sigma)

      xr = 1.0d0/tan(psi)
      yr = 0.0d0

      write (6, '(a, 4f16.8)') ' u1,v1,M1,R1:', u1, v1, M1, R1
      write (6, '(a,  f16.8)') ' psi        :', psi*180.0d0/pi
      write (6, '(a, 4f16.8)') ' u2,v2,M2,R2:', u2, v2, M2, R2
      write (6, '(a, 4f16.8)') ' u3,v3,M3,R3:', u3, v3, M3, R3
      write (6, '(a,  f16.8)') ' delta      :', delta*180.0d0/pi
      write (6, '(a,  f16.8)') ' refl.angle :', (sigma-delta)*180.0d0/pi
      write (6, '(a, 2f16.8)') ' refl.point :', xr, yr

      c1 = sqrt(u1*u1+v1*v1)/M1
      z1 = log(c1*c1/(gamma*(R1**(gamm1))))
      c2 = sqrt(u2*u2+v2*v2)/M2
      z2 = log(c2*c2/(gamma*(R2**(gamm1))))
      c3 = sqrt(u3*u3+v3*v3)/M3
      z3 = log(c3*c3/(gamma*(R3**(gamm1))))

c     Initial solution
```

```
      uini  = u1
      vini  = v1
      cini  = c1
      zini  = z1

c     Construction geometric data

      do 30 level = 0, BasLev
        call MkGeo(level, PNTR, PPTY, DATA)
   30 continue

c     Initialization solution on level 0

      call SolIni(0, uini, vini, cini, zini, PNTR, PPTY, DATA)

c     FMG-algorithm

      call FMG(BasLev, nfas, npmg, nqmg, ncycl, ff,
     +         PNTR, PPTY, DATA)

c     Computation residual after FMG-algorithm

      call Res(BasLev, RMx, RMn, PNTR, PPTY, DATA)

      RMax(0)  = RMx(0)
      RMean(0) = RMn(0)

c     FAS cycles

      do 40 ifas = 1, nnfas
        call FAS(0, BasLev, npmg, nqmg, ncycl, ff,
     +           PNTR, PPTY, DATA)
        call Res(BasLev, RMx, RMn, PNTR, PPTY, DATA)
        RMax(ifas)  = RMx(0)
        RMean(ifas) = RMn(0)
   40 continue

c     Values for scaling residuals

      ScMx= RMean(0)
      ScMn= RMean(0)

c     Possibly scaling of residual
c     do 50 k = 0, nnfas
c       RMean(k)= RMean(k)/ScMn
c       RMax(k) = RMax(k)/ScMx
   50 continue

c     Printing of convergence history

      write (6,'(a)') ' convergence history:'
      write (6,'(a)') '       RMean      RMax'
      do 60 k = 0, nnfas
        write (6, '(i4, 2e11.4)') k, RMean(k), RMax(k)
   60 continue
```

```
c     Refinement cycles

      do 200 iref = 1, nref

        ifas = iref*nnfas

c       Storing Mach number and density in old solution memory locations

        call StatSF(BasLev, nlevel-1, PNTR, PPTY, DATA)

c       Setting of refinement flags

        call RFlags(BasLev, nlevel-1, RefCr, PQold2, PNTR, PPTY, DATA)

c       Construction of refinements

        call Refn (BasLev, nlevel-1, PNTR, PPTY, DATA)
        nlevel = nlevel + 1

c       FAS-cycles on adaptively refined grid

        do 100 k = 1, nnfas
          call FAS(0, nlevel-1, npmg, nqmg, ncycl, ff,
     +             PNTR, PPTY, DATA)
          call Res(nlevel-1, RMx, RMn, PNTR, PPTY, DATA)
          RMax(ifas+k)  = RMx(0)
          RMean(ifas+k) = RMn(0)
c         RMean(ifas+k) = RMean(ifas+k)/ScMn
c         RMax(ifas+k)  = RMax(ifas+k)/ScMx
          write (6, '(i4, 2e11.4)') ifas+k, RMean(ifas+k),
     +           RMax(ifas+k)
  100   continue

  200 continue

c     Print solution

      do 210 k = 1, NOP
        call ShowS(k, PNTR, PPTY, DATA)
  210 continue

      end

c-----------------------------------------------------------------------


c     User subroutines

c-----------------------------------------------------------------------
      subroutine       GetXY(i, j, lev, x, y)
      integer          i, j, lev
      double precision x, y

c     This subroutine delivers the physical coordinates
```

```
c     '(x,y)' of the grid point with topological coordinates
c     '(i,j)' and level 'lev'.
c     This version is meant for the shock reflection problem.

      integer          nx0, ny0
      common /mesh/    nx0, ny0
      double precision xmin, xmax, ymin, ymax, dx, dy, dx0, dy0

c     Physical area

      xmin = 0.0
      xmax = 4.0
      ymin = 0.0
      ymax = 1.0

c     Cell width on level 0

      dx0 = (xmax - xmin)/float(nx0)
      dy0 = (ymax - ymin)/float(ny0)

c     Cell width

      dx = dx0/float(2**lev)
      dy = dy0/float(2**lev)

c     Physical coordinates

      x = float(i)*dx
      y = float(j)*dy

      end

c-----------------------------------------------------------------------
      subroutine       GetBC(i, j, lev, hor, BdyTp, QB)
      integer          i, j, lev, BdyTp
      double precision QB(4)
      logical          hor

c     This subroutine delivers the boundary condition for a
c     boundary wall (horizontal or vertical dependent on 'hor')
c     of a boundary wall with left end point '(i,j)' on level 'lev'.
c     The type of boundary is determined by 'BdyTp'.
c     This version is meant for the shock reflection problem.

      double precision u1, v1, c1, z1, u2, v2, c2, z2, u3, v3, c3, z3,
     +                 psi, delta, sigma, xr, yr
      common /exct/    u1, v1, c1, z1, u2, v2, c2, z2, u3, v3, c3, z3,
     +                 psi, delta, sigma, xr, yr

      if (hor) then

c     Horizontal boundary wall

        if (j .eq. 0) then
```

```
c         Solid wall boundary

          BdyTp = 5
          QB(1) = 0.0
          QB(2) = 0.0
          QB(3) = 0.0
          QB(4) = 0.0

        else

c         Subsonic inflow boundary

          BdyTp = 2
          QB(1) = u2
          QB(2) = v2
          QB(3) = z2
          QB(4) = 0.0

        end if

      else

c       Vertical boundary wall

        if (i .eq. 0) then

c         Supersonic inflow boundary

          BdyTp = 4
          QB(1) = u1
          QB(2) = v1
          QB(3) = c1
          QB(4) = z1

        else

c         Supersonic outflow boundary

          BdyTp = 0
          QB(1) = 0.0
          QB(2) = 0.0
          QB(3) = 0.0
          QB(4) = 0.0

        end if

      end if

      end

c-----------------------------------------------------------------------
      subroutine GetSrc(i, j, lev, s)
      integer    i, j, lev

c     This subroutine delivers the source term for the
```

```
c     cell with topological coordinates '(i,j)' on level 'lev'.
c     This subroutine is for the general case: Euler flow
c     with source terms equal 0

      integer          k
      double precision s(4)

      do 10 k = 1, 4
        s(k) = 0.0
10    continue

      end

c-----------------------------------------------------------------------


c     Auxiliary subroutines

c-----------------------------------------------------------------------
      subroutine ShowS(patch, PNTR, PPTY, DATA)
      include    'basis.i'
      include    'euler.i'
      integer     patch

c     This subroutine prints the solution in the cell of
c     patch 'patch' on standard output.

      if (PPTY(Compl, patch)) then
        write(6, '(a, 3i5, 4e11.4)') ' i, j, lev, sol: ',
     +             PNTR(XX, patch), PNTR(YY, patch), PNTR(LV, patch),
     +             DATA(PQ1, patch), DATA(PQ2, patch),
     +             DATA(PQ3, patch), DATA(PQ4, patch)
      end if

      end


c-----------------------------------------------------------------------
      subroutine       RFlags(FrmLev, ToLev, RefCr, comp
     +                 PNTR, PPTY, DATA)
      include         'basis.i'
      include         'euler.i'
      integer          FrmLev, ToLev, comp
      double precision RefCr

c     This subroutine Scans the patches on level 'ToLev' to find
c     which should be refind. Refinement depends on the criterion
c     'RefCr'

      integer          cmp
      double precision cr
      common /RFlagC/  cr, cmp
      external         RFlagP

      cmp = comp
```

```
      cr  = RefCr

      call Scan(RtPtr, NrmOrd, FrmLev, ToLev, RFlagP, PNTR, PPTY, DATA)

      end


c-----------------------------------------------------------------------
      subroutine RFlagP(patch, PNTR, PPTY, DATA)
      include   'basis.i'
      include   'euler.i'
      integer    patch

c     This subroutine determines an error estimate of the
c     solution at patch 'patch' by the difference
c     in one of the solution components.
c     If the difference is larger than criterion 'cr' then
c     the patch is marked pregnant.
c     It is assumed that the component to be used for refinement
c     detection is stored in the memory locations identified by 'cmp'.

      logical          hor, ver
      parameter       (hor = .true., ver = .false.)
      integer          cmp
      double precision cr, cmp
      common /RFlagC/  cr, cmp
      integer          NNb, ENb
      double precision D(PRhs1:PRhs4)

      if (PPTY(Compl, patch) .and. PNTR(NE, patch).eq.Nil) then

        NNb = PNTR(NN, patch)
        ENb = PNTR(EE, patch)

c       Determine difference in component and
c       set refinement flag when necessary.

        call Diff(NNb, hor, D, PNTR, PPTY, DATA)
        PPTY(Prgnt, patch) = (abs(D(cmp)) .gt. cr)
        if (.not.PPTY(Prgnt, patch)) then
          call Diff(ENb, ver, D, PNTR, PPTY, DATA)
          PPTY(Prgnt, patch) = (abs(D(cmp)) .gt. cr)
          if (.not.PPTY(Prgnt, patch)) then
            call Diff(patch, hor, D, PNTR, PPTY, DATA)
            PPTY(Prgnt, patch) = (abs(D(cmp)) .gt. cr)
            if (.not.PPTY(Prgnt, patch)) then
              call Diff(patch, ver, D, PNTR, PPTY, DATA)
              PPTY(Prgnt, patch) = (abs(D(cmp)) .gt. cr)
            end if
          end if
        end if

      end if

      end
```

References

[1] P.W. Hemker, H.T.M. van der Maarel and C.T.H. Everaars, *BASIS: A Data Structure for adaptive multigrid computations*, CWI Amsterdam, Report NM-R9014, 1990.

[2] P.W. Hemker and S.P. Spekreijse, *Multiple Grid and Osher's scheme for the Efficient Solution of the Steady Euler Equations*, Appl. Numer. Math. 2, 1986.

[3] S.P. Spekreijse, *Multigrid Solution of the Steady Euler Equations*, CWI-tract 46, Centre for Mathematics and Computer Science, Amsterdam, 1988.

Contents