Multigrid for Steady Gas Dynamics Problems *

P.W. Hemker B. Koren W.M. Lioen
M. Nool
CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
H.T.M. van der Maarel
MARIN, P.O. Box 28, 6700 AA Wageningen, The Netherlands

Abstract

This paper consists of two parts. In the first part we give a review of a good multigrid method for solving the steady Euler equations of gas dynamics on a locally refined mesh. The method is self-adaptive and makes use of unstructured grids that can be considered as parts of a nested sequence of structured grids. It is briefly described and applied to some steady Euler-flow problems. The method appears to be much more accurate and efficient than the corresponding multigrid method that applies global refinements only.

In the second part of the paper, vectorisation of the code is treated. To enable this vectorisation, index arrays are introduced and added to the quad-tree type data-structure that is applied in the scalar case. Speed-up factors are given for the same test cases as considered in the first part of the paper. The results are most satisfactory.

Note: In essence, this paper was already published in [12].

1 A solution-adaptive multigrid method for the steady Euler equations

In this first section we describe a self-adaptive multigrid method, developed at CWI, for the solution of steady gas dynamics problems. The method can be applied for the Euler as well as for the Navier-Stokes equations. However, because of space limitations in this paper we restrict ourselves to the treatment of the Euler equations. The extension to the Navier-Stokes equations is found e.g. in [18]. For a survey of different multigrid approaches to these equations we refer to [11].

Our Euler solver essentially uses a sequence of nested refinements, and the discretisations at the various levels of refinement yield a set of nonlinear algebraic equations for

*The research reported in this paper was partly performed in the framework of the BRITE-EURAM Aeronautics R&D Programme of the European Communities (Contract No. AER2-CT92-0040). The vectorisation was supported by Cray Research, Inc., under grant CRG 91.01. Use of supercomputer facilities was sponsored by the Stichting Nationale Computerfaciliteiten.

each discretisation level. The method to solve these nonlinear systems is the nonlinear multigrid scheme (FAS), which may be embedded in the full multigrid algorithm (FMG) as well as in an iterative defect-correction process (ItDeC). First these three basic algorithms are described as they are combined with the use of locally refined grids. Next the strategy to introduce local refinements is described. Following that, aspects of the computer coding are discussed for the local grid-refinement method. As an illustration, numerical results are presented for shock reflection on a flat surface and for transonic flow around an airfoil. These test cases are standard; they are used to validate the method and to get an idea of the gain in efficiency by the local grid-refinement method, in comparison to the corresponding uniform-grid method. We give CPU execution times for the scalar implementation of the local grid-refinement code and we compare these to those obtained with an implementation for uniform grids only. The latter implementation uses the same multigrid and defect-correction algorithms as the adaptive code (see Section 1.4.1). Significant speed-up factors are observed.

In Section 2 of this paper we describe the technique used for this vectorisation on a Cray Y-MP.

1.1 Multigrid and iterative defect correction

1.1.1 The first-order finite volume discretisation

To discretise the problem, the domain Ω is subdivided into disjoint quadrilateral cells $\Omega_{i,j}$, such that $\Omega_{i,j}$ and $\Omega_{i,j\pm1}$ or $\Omega_{i\pm1,j}$ are neighbouring cells. Further we denote the neighbours of $\Omega_{i,j}$ by Ω_{ijk}, (k=N,S,E,W) and a common cell edge by $\Gamma_{ijk} = \overline{\Omega}_{ij} \cap \overline{\Omega}_{ijk}$. The restriction to this kind of regular geometry is not necessary for the discretisation method but it leads to simple data structures when the method is implemented.

By integration of the fluid dynamics equations over $\Omega_{i,j}$ we obtain

$$\frac{\partial}{\partial t}\int\int_{\Omega_{i,j}} q \; dx\, dy + \int_{\delta\Omega_{i,j}} (f\, n_x + g\, n_y)\, ds = 0, \tag{1}$$

or

$$V_{ij}\, \frac{\partial}{\partial t}\, q_{ij} + \sum_k \int_{\Gamma_{ijk}} (f\, n_x + g\, n_y)\; ds = 0, \tag{2}$$

where V_{ij} is the volume of cell $\Omega_{i,j}$ and q_{ij} is the mean value of the state variable q over $\Omega_{i,j}$. Further we introduce the notation

$$\int_{\Gamma_{ijk}} (f\, n_x + g\, n_y)\; ds = f_{ijk}\, s_{ijk}, \tag{3}$$

where s_{ijk} is the length of Γ_{ijk} and f_{ijk} is the mean flux outward $\Omega_{i,j}$ over the side Γ_{ijk}. The space discretisation of the equations is done according to the Godunov principle: the state $q(t,x,y)$ is approximated by $q_{ij}(t)$ for all $\Omega_{i,j}$ and the mean fluxes f_{ijk} are approximated from the states in the adjacent cells. For this purpose, a computed flux $f_{ijk}(q^k_{ij}, q^k_{ijk})$ is introduced to replace f_{ijk}. Here, q^k_{ij} and q^k_{ijk} are approximations of q at both sides of Γ_{ijk}. Thus we obtain the semi-discretisation of (2):

$$V_{ij}\, \frac{\partial q_{ij}}{\partial t} = -\sum_k s_{ijk}\; f_{ijk}(q^k_{ij}, q^k_{ijk})\,, \tag{4}$$

and for the steady equations we obtain the discrete system of equations

$$N_h(q_h) = 0, \tag{5}$$

which is short for

$$(N_h(q_h))_{ij} := \sum_k s_{ijk}\, f_{ijk}(q_{ij}^k, q_{ijk}^k) = 0, \qquad \forall\, i,j\,.$$

In order to solve (5), we first generalise the problem slightly to

$$N_h(q_h) = r_h\,. \tag{6}$$

We use iteration with the full approximation scheme (FAS). For this we need a sequence of discretisations

$$N_{h_i}(q_{h_i})\,, \qquad \text{with} \quad h_0 > h_1 > \cdots > h_l = h\ .$$

For the meshwidth h_{i-1} we take $h_{i-1} = 2\,h_i$. For an irregular mesh we delete each second line of mesh points to obtain the coarser grid.

One FAS cycle for the solution of (6) consists of the following steps: start with an approximate solution q_h; improve q_h by application of ν_1 nonlinear (pre-) relaxation iterations to $N_h(q_h) = r_h$; compute the residual $N_h(q_h)$; find an approximation of q_h on the next coarser grid, say q_{2h}. For this we may either use a restriction $q_{2h} = R_{2h,h}q_h$, or use another previously obtained approximation q_{2h}. Compute

$$r_{2h} = N_{2h}(q_{2h}) + \overline{R}_{2h,h}\,(r_h \; - \; N_h(q_h)),$$

and approximate the solution of

$$N_{2h}(q_{2h}) = r_{2h} \tag{7}$$

by application of σ FAS cycles. The result is $\tilde{q}_{2h}$; correct the current solution by

$$q_h \; := \; q_h + P_{h,2h}\,(\tilde{q}_{2h} - q_{2h})\,;$$

improve q_h by application of ν_2 nonlinear (post-) relaxation iterations to $N_h(q_h) = r_h$.

In this process the steps between the pre- and the post-relaxation are the *coarse grid correction.* These steps are skipped on the coarsest grid h_0. For the solution of the nonlinear system (5), FAS iteration is simply applied with $r_h = 0$ on the finest grid. During the FAS iteration, on the coarser grids, non-zero right-hand sides appear in (7).

For the grid transfer operators $P_{h,2h}$ and $\overline{R}_{2h,h}$ we make a choice that is consistent with the concept of our finite volume discretisation. This discretisation is essentially a weighted residual method, where the solution is approximated by a piecewise constant function (on cells $\Omega_{i,j}$) and where the residual is weighted by characteristic functions on all $\Omega_{i,j}$. From this point of view, it is natural to use a piecewise constant interpolation for $P_{h,2h}$ and to use addition over subcells for $\overline{R}_{2h,h}$. Notice that $\overline{R}_{2h,h} = P_{h,2h}^T$. With these choices it is clear that

$$N_{2h}(q_{2h}) = \overline{R}_{2h,h}\, N_h\,(\,P_{h,2h}q_{2h}\,)\,, \tag{8}$$

i.e. the coarse grid finite volume discretisation is a formal *Galerkin approximation* of the fine grid finite volume discretisation. Using (8) on all different levels, we obtain a nested sequence of discretisations.

The effect of the Galerkin approximation $N_{2h} = \overline{R}_{2h,h}\, N_h\, P_{h,2h}$ on the approximate solution $\tilde{q}_h$ obtained after a coarse grid correction is the following property of its residual

$$\overline{R}_{2h,h}\; [r_h - N_h(\tilde{q})] = \mathcal{O}(\|q_h - \tilde{q}\|^2)\ .$$

Because $\overline{R}_{2h,h}$ is an addition over 4 neighbouring cells, this means that the restriction of the residual mainly contains high frequency components. A small restriction of the residual implies that possible large residuals over neighbouring cells cancel: the residual is rapidly varying. Local relaxation methods, such as point Gauss-Seidel, are now able to eliminate such residuals efficiently. In this way, FAS is a quite efficient method to solve the first order discrete equations. The second order accurate discrete equations cannot be solved this way because they lack sufficient numerical diffusion.

For the second-order discretisation FAS is embedded in an iterative defect-correction (ItDeC) process [2]. The implementation of the multigrid scheme for the self-adaptive discretisation is based on the same principles. These principles are described in detail in [14, 15, 28, 30] and applied in [18, 16, 17]. Iterative defect correction is applied in [18, 16, 9, 29].

In the next section we proceed by giving a brief description of these methods, including their slight modifications for our solution-adaptive application as studied in [22].

1.1.2 A locally nested sequence of discretisations

In order to use multigrid for discretisations on locally refined grids, we have to specify grid-transfer operators. Starting from a certain level, l_b, now the discrete problems do not extend over all the domain Ω, but —for finer levels— over increasingly smaller subdomains. The discrete problem on such a fine grid (level $l+1$) is completed by boundary conditions that require the solution to join with the solution on the next coarser grid (level l). We denote the restriction operators for the solution and the residual by R^l_{l+1} and $\overline{R}^l_{l+1}$ respectively, and the prolongation operator by P^{l+1}_l. These grid transfer operators are defined such that a sequence of locally nested discretisations $\{N^l\}_{l=0,\dots,L}$ is obtained. This means that a Galerkin approximation, $R^l_{l+1}N^{l+1}P^{l+1}_l$, of the fine-grid discretisation N^{l+1} equals the coarse-grid discrete operator N^l, restricted to the refined cells. This implies that, for each cell $\Omega^l_{i,j}$ for which a refinement exists on level $l+1$, we have

$$\left\{N^l(q^l)\right\}^l_{i,j} = \left\{\overline{R}^l_{l+1}N^{l+1}(P^{l+1}_l q^l)\right\}^l_{i,j}. \tag{9}$$

The restriction operator for source terms, $\overline{R}^l_{l+1}$, is given by addition of the sources:

$$\left\{\overline{R}^l_{l+1}r^{l+1}\right\}^l_{i,j} = \sum_{\Omega^{l+1}_{n,m}\subset\Omega^l_{i,j}} r^{l+1}_{n,m}. \tag{10}$$

The restriction operator for the solution, R^l_{l+1}, is defined by taking the mean value:

$$\left\{R^l_{l+1}q^{l+1}\right\}^l_{i,j} = \frac{1}{4}\sum_{\Omega^{l+1}_{n,m}\subset\Omega^l_{i,j}} q^{l+1}_{n,m}.$$

If the grid is sufficiently smooth, this restriction is second-order accurate. The operator for the prolongation of a solution correction, P^{l+1}_l, corresponds with piecewise constant interpolation:

$$\left\{P^{l+1}_l q^l\right\}^{l+1}_{n,m} = q^l_{i,j}, \quad \forall\Omega^{l+1}_{n,m}\subset\Omega^l_{i,j}. \tag{11}$$

As shown in [16, 18, 30], because of (9), these restrictions and prolongation show a good multigrid performance, even with simple point Gauss-Seidel relaxation.

The prolongation (11) and restriction (10) satisfy the multigrid rule (see [7, 10, 32])

$$m_p + m_r > m,$$

where m_p is the order of accuracy of the prolongation, m_r is the order of accuracy of the restriction and m is the order of the differential equation.
(For P_l^{l+1}, $m_p = 1$, for $\overline{R}_{l+1}^l$, $m_r = 1$ and $m = 1$ for the Euler equations.)

For the restrictions and prolongation described above, the first-order accurate discrete equations form a locally nested sequence, except for the equations at the inter-grid boundaries. A first-order accurate interpolation near the inter-grid boundary, which introduces first-order accurate virtual states at the other side of this boundary (and hence shows first-order weak consistency, see [23, Chapter 2]), yields a locally nested sequence.

1.1.3 FAS and FMG

On level l, we denote the discrete operator, N_h, with first-order accuracy by N_{I}^l, and that with second-order accuracy by N_{II}^l. The set of equations to be solved is then first

$$N_{\mathrm{I}}^l(q^l) = r^l, \tag{12a}$$

where the right-hand side r^l is given by

$$r_{i,j}^l = \begin{cases} \int_{\Omega_{i,j}^l} r_h \, dx, \\ \quad \text{when } \Omega_{i,j}^l \text{ is on the finest grid;} \\ \left\{N_{\mathrm{I}}^l(R_{l+1}^l q^l)\right\}_{i,j}^l \\ -\left\{\overline{R}_{l+1}^l \left(N_{\mathrm{I}}^{l+1}(q^{l+1}) - r^{l+1}\right)\right\}_{i,j}^l, \\ \quad \text{when the cell } \Omega_{i,j}^l \text{ is subdivided.} \end{cases} \tag{12b}$$

A complete FAS-cycle for the locally refined grid, where all q^l, $l = 0, \dots, L$ are improved, is a recursive algorithm defined by the following steps:

1. improve the solution q^l by applying ν_1 pre-relaxations to (12a) at level l, resulting in the approximate solution $(q^l)_0$;
2. compute the right-hand side r^{l-1}, determined by (12b) at level l, using $q^l = (q^l)_0$;
3. improve the solution q^{l-1} by applying σ FAS-cycles to the equations (12a) at level $l-1$;
4. compute the coarse grid correction $d^{l-1} = q^{l-1} - R_l^{l-1}(q^l)_0$;
5. improve the solution q^l by adding the prolongation of the coarse-grid correction:
$$q^l = (q^l)_0 + P_{l-1}^l d^{l-1};$$
6. improve the solution q^l by applying ν_2 post-relaxation sweeps to the system (12a) at level l.

The steps (2)-(5) together are called the coarse-grid correction steps. These steps are skipped at the level $l = 0$.

Upon convergence of the nonlinear multigrid iteration, the solution not only satisfies (12) for all l, but also

$$q_{i,j}^l = \{R_{l+1}^l q^{l+1}\}_{i,j}^l. \tag{13}$$

As a relaxation procedure in FAS a *collective* symmetric point Gauss-Seidel relaxation is used on each level of refinement. This means that for each cell $\Omega^l_{i,j}$ visited during this relaxation process, the complete state $q^l_{i,j}$ is updated. This implies the solution of a local 4×4 nonlinear system, $\{N^l_{\rm I}(q^l)\}^l_{i,j} = r^l_{i,j}$, which is done by Newton iteration. The stopping criterion for the Newton iteration is chosen so that, in all but exceptional cases, only one or two iteration steps are made. On each level, after a first (nonlinear) relaxation sweep in the usual lexicographical order, another sweep is done in the reverse direction. This smoother is shown to be very efficient in both subsonic and supersonic Euler-flow computations [14].

The initial solution on the finest level is obtained by the application of nested iteration, also called full multigrid (FMG) [3, 4, 7]. This means that the system is first solved on the coarsest grid, $l = 0$, simply by sufficient relaxation. On the finer levels, $l = 1, 2, \cdots, L$, the FMG-scheme is recursively defined as follows:

1. initialise the solution on level l by $q^l := \tilde{P}^l_{l-1} q^{l-1}$;
2. improve the solution on level l by the application of γ FAS-cycles with level l as highest level;
3. if the solution on level l is not sufficiently accurate, then introduce level $l + 1$ by (locally) refining the grid and apply the FMG-cycle for level $l + 1$;

In the experiments presented at the end of this section we use $\sigma = 1$ (V-cycles), $\nu_1 = \nu_2 = 1$ (a single pre-relaxation and a single post-relaxation) and $\gamma = 1$ (a single V-cycle, before starting a higher level). The prolongation $\tilde{P}^{l+1}_l$ used in the FMG-algorithm is bilinear interpolation.

1.1.4 Iterative defect correction

In the previous section, we showed how the first order accurate discrete system (12) is solved. Generally, first order accuracy is not sufficient, and we want to solve a second order accurate discretisation. The system of equations with second-order accuracy is *not* efficiently solved by the same technique as the first order system. The reason is that the second-order discrete system has less numerical dissipation, and hence a sufficiently efficient relaxation procedure cannot be found. The second-order system is solved by iterative defect correction (ItDeC) [2, 7]. Let the set of our equations at level l be given by

$$N^l_{\rm II}(q^l) = r^l. \tag{14}$$

The ItDeC-algorithm makes use of the fact that an efficient technique does exist for the lower-order system $N^l_{\rm I}(q^l) = r^l$. An initial approximation for the ItDeC-process is obtained by application of the FAS-algorithm to (12). In the ItDeC-process the right-hand side r^l depends on the defect of the higher-order accurate equations, through

$$r^l_{i,j} = \begin{cases} \left\{N^l_{\rm I}(q^l)\right\}^l_{i,j} - \left\{N^l_{\rm II}(q^l)\right\}^l_{i,j}, \\ \qquad \text{if } \Omega^l_{i,j} \text{ is a cell on the finest grid,} \\ \left\{N^l_{\rm I}(R^l_{l+1} q^{l+1})\right\}^l_{i,j} \\ -\left\{\overline{R}^l_{l+1}\left(N^{l+1}_{\rm I}(q^l) - r^{l+1}\right)\right\}^l_{i,j}, \\ \qquad \text{if a refinement exists for cell } \Omega^l_{i,j}\ . \end{cases} \tag{15}$$

Upon convergence of the ItDeC-scheme, equation (14) is satisfied.

In [16] it is shown that one nonlinear multigrid cycle per defect-correction cycle is sufficient and most efficient. In our experiments at the end of this section we use a single nonlinear multigrid cycle per defect-correction cycle indeed.

Before any local grid refinement is introduced, first the solution is approximated, sufficiently accurate, on a certain level l_b. This is done by the application of the nested iteration FMG, one or two FAS-cycles to approximately solve the first-order discretisation and then a sufficient number of ItDeC-cycles for second-order accuracy.

1.2 Refinement cycles

Solution-adaptive grid refinement involves automatic grid refinement at some stage in the solution process. Based on an a-posteriori estimate of relevant quantities appearing in the refinement criterion, the grid is refined where these quantities exceed a pre-set or solution-dependent threshold value (see e.g. [27]).

A computation that uses local grid refinement starts with the FMG-algorithm and subsequent iterative defect correction. It yields first an approximate solution for the uniform grid on some basic level l_b. Introduction of local grid refinements is accomplished by the following refinement algorithm, where l is the highest available level:

1. determine which cells on level l should be refined (or may be deleted), based on the refinement criterion and on an a-posteriori estimate of the relevant quantities used in the refinement criterion;
2. decide whether a grid on level $l+1$ should be created, denote the (new) highest level by L;
3. refine the grid and delete obsolete cells on all levels, from l_b up to and including level $L-1$;
4. initialise the approximate solution of the newly created refinements by the application of the prolongation $\tilde{P}_m^{m+1}$, for $m = l_b, \ldots, L-1$ (similar to the FMG-algorithm);
5. improve the solution on all levels by the application of ρ FAS iterations (first-order discretisation), or by ρ ItDeC-cycles (second-order discretisation) on the complete sequence of grids;
6. either apply a refinement cycle on the new system, or solve the present system of equations by a sufficient number of (FAS or ItDeC) iteration;

The decision in step (2) of the refinement algorithm may be determined by the answer to the question whether the grids on all currently present levels have sufficiently converged, or whether the highest level allowed has already been reached. Notice that for newly created cells, the refinement cycle actually is an application of the nested iteration algorithm FMG, as introduced in the previous section. For the prolongation $\tilde{P}_l^{l+1}$ a bilinear interpolation is used for newly created cells. In second-order computations, after initialisation of the solution for newly created cells, defect correction is continued, without first applying the nonlinear multigrid scheme to the first-order accurate system (12). The number of iterations ρ before a new refinement cycle is started, in step (5) of the refinement algorithm, determines to a large extent the efficiency of the adaptive grid-refinement method. However, using an insufficient number of iterations in step (5) may yield a grid too much distorted by the insufficiently converged numerical solution. In practice, $\rho = 1$ or $\rho = 2$ for a first-order discretisation appears to yield a grid virtually the same as the grid obtained by using a fully converged solution. For a second-order discretisation $\rho = 4$ or $\rho = 5$ appears to be sufficient.

1.3 A scalar implementation

In order to perform Euler-flow computations, accelerated by multigrid with solution-dependent local grid refinement, a computer code has been developed in portable Fortran 77. This code consists of two modules. One module, called BASIS, is entirely devoted to set up and maintain the data structure. It is described in [13]. The second part, called EULER, consists of routines related to the adaptive multigrid Euler-flow computations. This module is described in [24]. Recently, it appeared that the efficiency of the code could benefit essentially from vectorisation [21]. This resulted in an additional module, called EUVEL, which is presented in Section 2.

In the code, the (sequence of locally refined) grids over the domain that is discretised, is composed of so-called *patches*. A patch consists of a corner point, and (if not at a top or right side of a grid) a vertical cell edge, a horizontal cell edge and a cell interior. If a patch contains a cell, sufficient neighbour patches exist to provide the necessary edges and vertices. On the other hand, each corner point, each edge and each cell of the geometric structure belong to some patch. The data in the structure are stored and referenced through these patches. By the nature of the refinements, the patches in the data structure are related in a quad-tree structure. The tree of cells is a genuine subtree of the tree of patches.

In the linked list that implements the quad-tree structure, nine *pointers* are used for each patch. One pointer is used for the parent of a patch, four pointers for the kid patches and four pointers for the neighbours of each patch. In the Fortran implementation each patch has a unique number, and the use of pointers in the linked list is emulated by a large, two-dimensional array of type `integer`. For each patch the patch-numbers of its parent, its possible kids and its possible neighbours are stored column-wise in the integer array. Furthermore, for each patch a set of properties is kept, which identify the *type* of the patch: whether the patch contains a cell, whether the horizontal cell edge or vertical cell edge is part of the inter-grid boundary or the boundary of the domain, whether the cell contained in the patch should be refined at the earliest possible occasion, etc.. These properties are stored column-wise in a two-dimensional array of type `logical`, where the column number corresponds with the unique number of the patch, and the row with the specific property. Finally, the numerical data for the problem are kept in another two-dimensional array of type `real` (or `double precision`). These data are also addressed through the unique patch number. For each patch a total of 18 real numbers is stored. The data structure handled by BASIS has a much wider range of applications than Euler-flow computations or cell-centred discretisation schemes.

The actions on the data in the data structure are performed through a depth-first traversal of the quad-tree. The subroutines performing the necessary numerical actions work by the application of this tree traversal algorithm.

For each patch visited through this algorithm, a subroutine is called to perform some action on the data in the structure. The quad-tree structure and the use of such a traversal algorithm to perform any task, is very well-suited for the implementation of a multigrid algorithm with adaptive mesh refinement. However, automatic vectorisation (i.e., vectorisation by a compiler) of a code of this nature does not gain any performance. Therefore, in the vector extension library presented in Section 2, subroutines are provided that collect pointers to patches contained in the geometric structure of a single level of refinement. These pointers are placed in an appropriate order in a separate array of pointers. The array of pointers makes the algorithm fit for vectorised processing. Essentially, by using indirect addressing via those pointers, instead of tree-traversing, the original subroutines that were called for a single patch to perform some numerical action, have been replaced by subroutines that work on multiple patches. Details follow in Section 2.

1.4 Numerical results

1.4.1 Shock reflection

The problem In this section we consider a shock reflection problem. This is a gas dynamics problem of a supersonic flow along a flat solid surface. The domain of definition is $\Omega = \{(x,y) \mid 0 < x < 4,\ 0 < y < 1\}$, and the flat surface is located at $y = 0$. A shock is impinging from the point $(0,1)$, at an angle of 29° with the positive x-direction. At the inflow boundary $x = 0$, the boundary conditions for this solution are given by

$$\begin{array}{rcll} u(0,y) &=& 1, & (x\text{-velocity}) \\ v(0,y) &=& 0, & (y\text{-velocity}) \\ M(0,y) &=& 2.9, & (\text{Mach number}) \\ \rho(0,y) &=& 1 & (\text{density}). \end{array} \tag{16}$$

At the inflow boundary $y = 1$, the flow perpendicular to the horizontal boundary is subsonic, and we impose three conditions. These are approximately given by

$$\begin{array}{rcl} u(x,1) &=& 0.90322141, \\ v(x,1) &=& -0.17459319, \\ M(x,1) &=& 2.37807192. \end{array} \tag{17}$$

The boundary $y = 0$ is the solid wall, and we impose impermeability, given by

$$v(x,0) \;=\; 0.$$

The shock is reflected at the solid wall, at an angle of about 23.279°. The exact solution is known from shock relations. It is a piecewise constant function. The impinging and reflected shocks form the discontinuities of this function.

Refinement The domain Ω is rectangular. The coarsest grid used, level $l = 0$, is a 6×2 grid. The basic level is $l_b = 1$. Since, away from the shock, the exact solution is a constant function for both a first-order discretisation and a second-order discretisation, the local discretisation error is zero away from the shock. For an adaptive computation, it is sufficient *for this problem* to use only the variation of the solution as the refinement criterion. Grids are refined on the basis of the first undivided difference of a solution component. According to research on the use of undivided differences as a general refinement criterion, it is found that of any component of the solution, the first undivided difference of density gives the best results [5].

Results For this problem, away from the shock, the discretisation yields equations with local discretisation error equal to zero. The accuracy of the results will be determined to a large extent by the *resolution* provided by the grid used. In the following paragraphs we describe the first and the second order discretisation, respectively.

Table 1: Final number of cells used for shock reflection problem; first-order discretisation.

	locally refined		uniform	
L	composite	total	composite	total
4	1533	2040	3072	4092
5	3582	4772	12288	16380
6	7797	10392	49152	65532

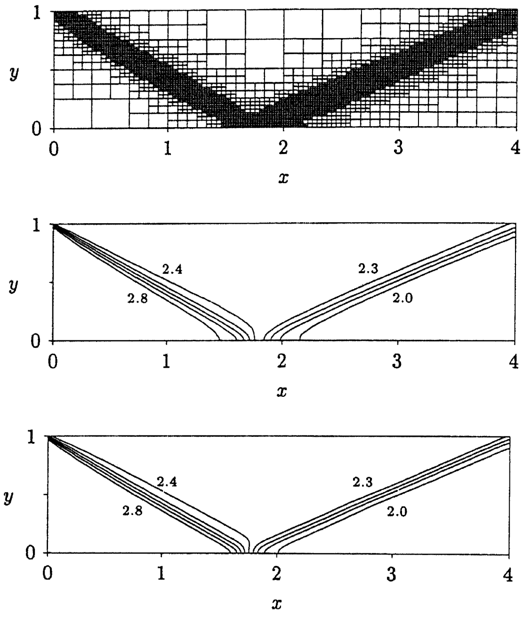


Figure 1: Locally refined grid and iso-lines of the Mach number for the shock reflection problem on the locally refined and the uniform grid (from top to bottom). First-order discretisation; $L = 5$.

First-order discretisation The equations resulting from the first-order discretisation, are solved on an adaptively refined grid. For the highest level L we take successively $L = 4, 5, 6$ to study the convergence behaviour. The number of FAS-iterations (V-cycles) for each refinement cycle is two ($\gamma = 2$). A cell is refined if the absolute value of the first undivided difference in either x-direction or y-direction exceeds 0.05. We consider refinements to have become obsolete if the absolute value of the first undivided difference of density drops under 0.025. In Table 1 the number of cells used are given for both the locally refined and the uniform grids. Note that the number of cells doubles approximately when going from L to $L + 1$. Figure 1 shows for $L = 5$ the grid obtained by local refinement, with iso-plots of the Mach number on that grid as well as on the corresponding uniform grid. In Figure 2 the convergence histories are given for both the adaptive method and the uniform method. Along the vertical axis, these figures show the logarithm of the mean of the four discrete L_1 norms of the scaled residual of the first-order discretisation, defined by $(A^l_{i,j})^{-1}\{N^l_{\text{I}}(q^l) - r^l\}^l_{i,j}$. Along the horizontal

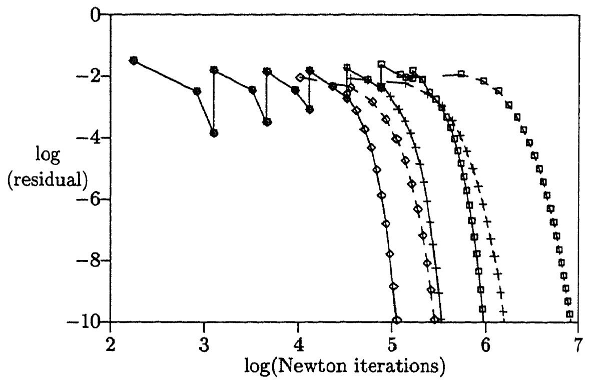


Figure 2: Residual versus amount of work: convergence histories for adaptively refined and uniform grids; first-order discretisation; ◇: $L = 4$; +: $L = 5$; □: $L = 6$; ——: locally refined; — —: uniform.

Table 2: Final number of cells used for shock reflection problem; second-order discretisation.

	locally refined		uniform	
L	composite	total	composite	total
4	924	1228	3072	4092
5	2004	2668	12288	16380
6	4707	6272	49152	65532

axis the figures show the logarithm of the number of elementary Newton iteration steps performed (i.e., the approximate solution of the nonlinear 4×4-system used in the point relaxation). For $L = 6$ the number of Newton iteration steps to convergence up to machine precision for the adaptive method is about nine times less than the number of iterations needed when a uniform grid is used, while virtually the same solution is obtained (see Figure 1 and Figure 2). For $L = 5$ the number of iterations for the adaptive method is about five times less and for $L = 4$ this is about 2.5 times less.

Second-order discretisation We use the second-order discretisation N^l_{II}, with the Van Albada limiter [1], and third-order accurate virtual states as defined in [23, Chapter 2]. The refinement decision is the same as for the first-order discretisation. The number of defect-correction iterations in each refinement cycle is five. It appears that, after five defect-correction cycles, possible wiggles in the 'initial' solution have vanished. The final locally refined grid and iso-lines of the Mach number for $L = 5$ are shown in Figure 3. The number of cells for local refinement with this second-order discretisation is shown in Table 2. Notice that the number of cells for levels $L = 4, 5, 6$ is much smaller for the adaptive computation with second-order discretisation than for the computation with first-order discretisation. For second-order discretisation some extra refinements may be introduced, apart from the refinements introduced by the refinement criterion itself. These extra refinements are introduced in order to let virtual states for the discretisation on level l depend only on the solution on levels l and $l - 1$, and not on the

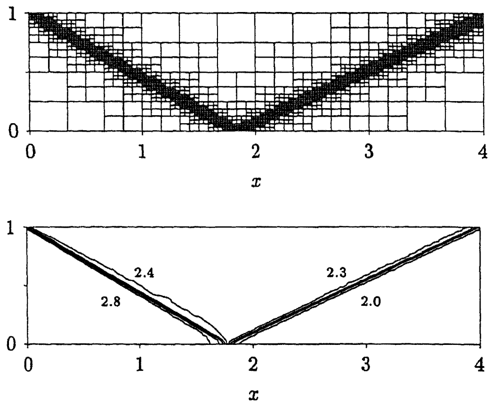


Figure 3: Grid and iso-lines of the Mach number for the shock reflection problem on a locally refined grid; second-order discretisation; $L = 5$.

level $l - 2$.

Convergence histories for locally refined and uniform grids are given in Figure 4. This figure shows the logarithm of the mean of the four discrete L_1 norms of the second-order discretisation versus the logarithm of the number of Newton iteration steps. We did not consider $L = 6$ and a uniform grid. (The latter problem is so large that it caused our workstation to start swapping pieces of memory to disk, resulting in a very large processing time.)

Apparently, the defect-correction process does not converge for uniform grids. The reason for this is possibly the following. On a uniform grid with finest level L, many more Fourier modes can be represented than on the refined grid with finest level L. Especially low-frequency modes can be represented very well on the uniform grid, better than on the locally refined grid. In [6] an amplification factor $g \approx 1$ for low-frequency Fourier modes is found, in case of the linear convection problem in two space dimensions. However, it should be stressed that for this linear convection problem this high amplification factor corresponds to functions that are constant in the characteristic direction of the problem.

The defect-correction algorithm for locally refined grids does converge. For second-order discretisation and defect correction, discretisation on a locally refined grid yields a more robust algorithm for this problem.

Execution time In order to get an idea of the execution time, for this problem we give CPU-times of our scalar Fortran research code on an SGI IRIS INDIGO XS workstation. Optimisation was done automatically by the compiler. In Table 3 we give the average CPU-time per Newton iteration step. Note that these Newton iterations are again *local* Newton iterations, used in the nonlinear point Gauss-Seidel relaxation. Table 3 also shows the average CPU-time for another (scalar) multigrid code, developed to work with uniform grids only. This non-adaptive code, called EULER7, implements the same multigrid and defect-correction algorithms as used in the code for adaptive computations (see e.g. [16]). The FAS-algorithm on a locally refined grid appears to be only three percent more expensive than on a uniform grid with the adaptive code. The iterative defect correction appears to be about 18% more expensive. For the FAS-algorithm, the

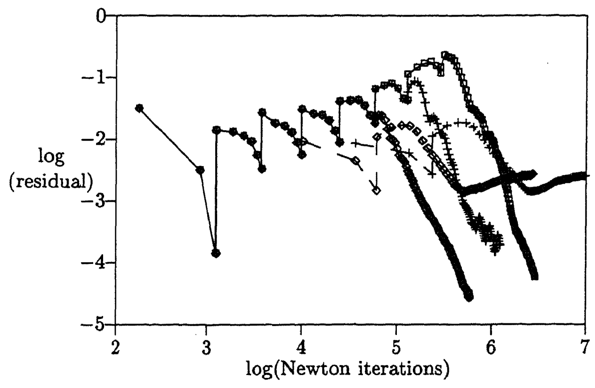


Figure 4: Residual versus amount of work: convergence histories for defect correction and second-order discretisation on uniform and locally refined grids; ◇: $L = 4$; +: $L = 5$; □: $L = 6$; ——: locally refined; — —: uniform.

Table 3: CPU-time required per Newton iteration step for the shock reflection problem. The numbers denote CPU-time: ms/iteration.

	adaptive-grid code		uniform grid code
	locally refined	uniform	
FAS	0.96	0.93	0.84
ItDeC/FAS	1.14	1.03	0.84

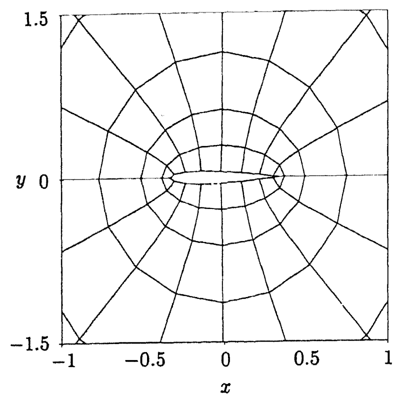


Figure 5: Uniform grid of level $l = 1$, around NACA0012 airfoil.

adaptive code *with* local grid refinement, appears to be about 14% more expensive than the non-adaptive code EULER7. For iterative defect correction, the adaptive-grid code is about 34% more expensive than EULER7.

1.4.2 Transonic airfoil flow

In this section we consider transonic flow around the NACA0012-airfoil. The flow conditions at the far-field boundary are: $M_\infty = 0.8$, angle of attack $\alpha = 1.25°$, $\rho_\infty = 1$ and $(u^2 + v^2)_\infty = 1$. The computational domain extends to about 100 chords to all sides.

As second-order operator, N^l_{II}, we use the Van Albada limiter scheme [28, 1]. Again, third-order accurate computation of virtual states is applied (see [23, Chapter 2] for details). The limiter scheme is used because spurious wiggles in the solution are expected if a non-limited, second-order scheme is used.

In the refinement criterion we use first undivided differences of the density, in both streamwise direction and the perpendicular direction. Two thresholds are used, one for each direction. This prevents the algorithm from refining in the neighbourhood of a shock only. It allows the algorithm also to find the contact discontinuity, and to resolve the expansion region. Then, we not only get a good resolution of the shock, but also a good resolution of the expansion. This in turn is important for the accurate computation of the lift and drag coefficients. The use of a single threshold value only (i.e., the same threshold for both criteria) would be inefficient for a small threshold value (too many refinements). On the other hand, a larger threshold value only refines at strong discontinuities.

The grid used is an O-type grid. The coarsest grid is a 5×8 grid. The highest level is $L = 5$. The uniform grid at level $l = 1$ is shown in full and in detail in Figure 5. A cell is refined if the first undivided difference of density in flow direction is larger than 0.02, or if this difference in the direction perpendicular to the flow is larger than 0.004. The final adaptively refined grid, with $L = 5$, is shown in Figure 6. In Figure 7 the Mach number distributions are shown both for an adaptively refined grid and a uniform grid. The pressure distributions for the uniform grid and for the locally refined grid are shown in Figure 8. For the lift and drag coefficient on the adaptively generated composite grid we find $c_l = 0.3480$, $c_d = 0.0235$. On the non-adaptive grid we find $c_l = 0.3512$ and $c_d = 0.0235$. The difference between these values is less than 10% of the scatter found

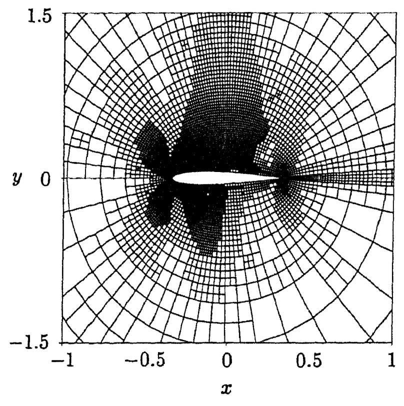
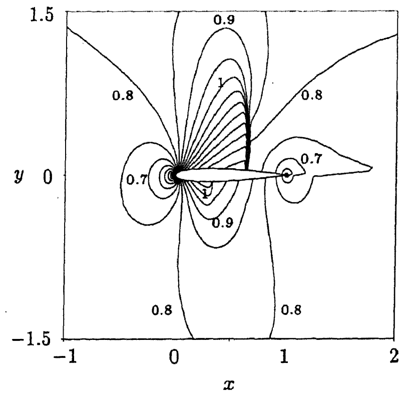


Figure 6: Locally refined grid with $L = 5$, around NACA0012 airfoil.



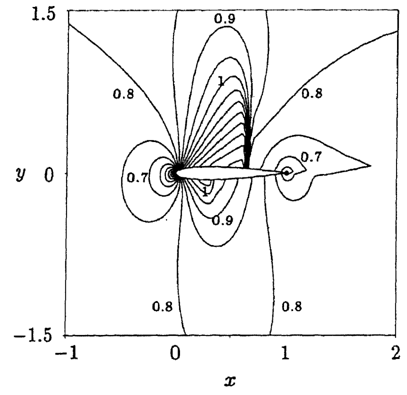


Figure 7: Iso-line plots of the Mach number for the transonic flow around a NACA0012 airfoil; $\alpha = 1.25°$; $M_\infty = 0.8$; locally refined grid: $L = 5$; uniform grid: $L = 4$.

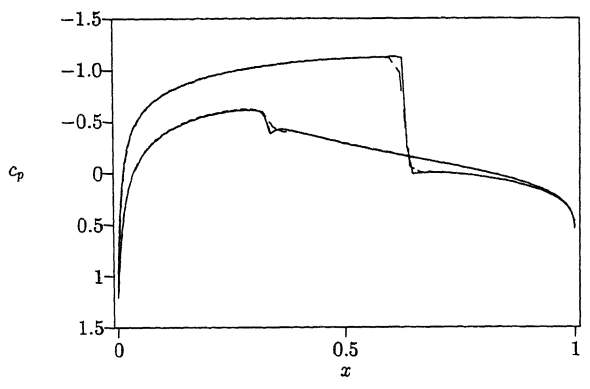


Figure 8: Pressure distribution for adaptively refined and uniform grids for transonic flow around a NACA0012 airfoil; $\alpha = 1.25°$; $M_\infty = 0.8$; ——: locally refined grid, $L = 5$; — —: uniform grid, $L = 4$.

between different reference results listed in [31]. This reference gives $c_l = 0.3632$ and $c_d = 0.0230$, obtained on a grid of 20480 cells by Schmidt and Jameson [31]. The number of cells on the adaptively generated composite grid is 7876 and a total number of 10488 cells was used in the computation. The non-adaptive grid uses 10240 cells on the finest grid and a total number of cells of 13640. The convergence histories of both the adaptive and non-adaptive case are shown in Figure 9. The adaptive computation takes about three times less work than the computation on the non-adaptive grid.

The solution-adaptive multigrid method appears to be an efficient tool for detailed studies of singular flow phenomena as well (see [19, 25]).

2 EUVEL: An EULER Vector Extension Library

In this section we describe the EUler Vector Extension Library (EUVEL), which contains a set of vectorised subroutines for the same algorithm as used for the original subroutines in BASIS and EULER. Since the implementation of the original subroutines in BASIS and EULER is inherently scalar and since the greatest performance gain on a machine as the Cray Y-MP is expected from vectorisation and not from parallelisation, we focus our attention on vectorisation of the original EULER code, using the same data structure. By the typical tree structure of the data structure used, it can be expected that parallelisation of the code will be more straightforward. In Section 2.1 we describe an extension of BASIS to facilitate the vectorisation. In Section 2.2 we describe the vectorisation of some of the EULER subroutines. To enhance the vectorisation of the relaxation process, Osher's flux-difference splitting scheme [26] was replaced by Van Leer's flux-splitting scheme [20].

2.1 Extensions to BASIS

A strict definition of the data structure is found in [13]. In the following subsections we briefly summarise the data structure, possibly repeating some information given in preceding sections. The computational domain Ω is partitioned into a finite number of

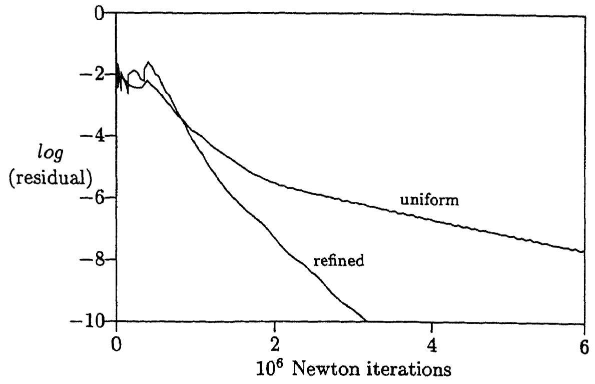


Figure 9: Residual versus amount of work: convergence histories of defect correction and second-order discretisation for NACA0012 airfoil flow; $\alpha = 1.25°$; $M_\infty = 0.8$; locally refined grid: $L = 5$; uniform grid: $L = 4$.

quadrilateral cells. For the adaptive multigrid algorithm different levels of refinement of the grid are used. 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. We call the cell on Ω^{l-1} the parent cell and the four smaller cells of the subdivision on Ω^l the NE-kid, SE-kid, SW-kid, and NW-kid cell, respectively (using the compass to denote the direction). The data structure is ordered by *patches*: the union of a point with possibly a horizontal cell edge, a vertical cell edge and a cell interior. The data structure consists of a quad-tree of such patches (i.e., at most four new branches at each node). With each cell (or node), a patch as well as a number of pointers, properties, coordinates and numerical values are associated.

The kernel of EULER is a BASIS subroutine called `Scan`, which accomplishes a depth-first quad-tree traversal, invoking an external subroutine (`DoIt`), which is passed as argument to `Scan`, for every patch found. Another argument of `Scan` is an `INTEGER` array `Order(1:4)` which contains a permutation of the directions NE, SE, SW, NW: the order in which the four kid cells (branches in the tree) are visited. Since `DoIt` is called separately for every patch found we hardly find any performance gain by automatic vectorisation. On the other hand, the data structure is ideally suited for a self-adapting mesh. So, we decided to leave the data structure unchanged and to add a pointer structure in order to allow vectorisation.

Each patch has an 'arbitrary' identification number. Therefore, two successive patch numbers can be related to patches that have nothing in common. All information concerning patches and their relations is stored in three large two-dimensional arrays:

`PNTR`	- an `INTEGER` array,
`PPTY`	- a `LOGICAL` array,
`DATA`	- a `REAL` (or `DOUBLE PRECISION`) array.

These arrays have `MNOP` (`M`aximum `N`umber `O`f `P`atches) columns. To prevent tree-wise scanning of the patches each time, we operate on patches of which the pointers are listed in various index arrays. Which specific index array is used depends on the kind of action.

First of all, pointers to patches that belong to the same level are gathered and stored into an array `LevelW(0:MNOP)`. Array `LevelW` (combined with `INTEGER` array

`NLev(0:MNOL)`, where `MNOL` denotes the `Maximum Number Of Levels`), tells us where we can find patches of level `k`: the array segment

```
LevelW(1:NLev(0))
```

contains pointers to patches on level $l = 0$, whereas the segment

```
LevelW(NLev(k-1)+1:NLev(k))
```

contains pointers to patches on level $l = \mathtt{k}$.

The collective symmetric point Gauss-Seidel relaxation can be vectorised by using a diagonal ordering. For this purpose, the elements of `LevelW`, referring to patches, are stored such that their diagonal number (being the sum of the ξ- and η-coordinates) is in monotonously non-descending order. The `INTEGER` array `NDiag(0:MNOD, 0:MNOL, 0:2)`[1], informs about the position of the diagonal elements within the array `LevelW`. The patches on diagonal `d` of level `k` can be found in the range

```
NDiag(d,k,0) : NDiag(d,k,1)
```

of `LevelW`. These index arrays are all generated by a subroutine, `SortLv`, and they must be updated only when the cell structure (the mesh) has changed.

2.2 Vectorisation of some of the EULER subroutines

We have restricted vectorisation to the most time-consuming subroutines. All subroutines adapted were originally implemented using `Scan`.

2.2.1 Computation of the flux, transport and their derivatives

To enhance the performance of the vectorised relaxation routines, it appeared that the P-variant of Osher's flux-difference splitting scheme ([15]) could be replaced by Van Leer's flux-splitting scheme. The relaxation process is responsible for approximately 80–90% of the total CPU-time. The vector length used in the relaxation will be at most the number of cells in a diagonal of the finest grid. However, since we are dealing with an adaptively refined grid, in many cases this vector length will not be very large. The motive for preferring Van Leer's scheme above Osher's scheme is its smaller number of branchings. In Osher's scheme there are 16 different possibilities, whereas in Van Leer's scheme there appear to be only 4. In fact, the main problem in vectorising is not the number of conditions, but the small number of patches that satisfy a particular condition in the upwind scheme. Moreover, this number of patches in a diagonal decreases during the relaxation process, because the number of Newton-steps needed on the individual patches may differ. It is clear that the 16 possibilities in Osher's scheme would result in smaller vector lengths than the 4 possibilities in Van Leer's scheme.

The original subroutine `FTA` computes the `F`lux, the `T`ransport and the left and right transport derivatives on a single patch. We created a generic source (to be pre-processed by cpp, the C language pre-processor) with the functionality of `FTA`. It operates on multiple patches (passed as an array of patch numbers), and it has Van Leer's flux-splitting scheme completely inlined. With cpp six different versions are generated, working on multiple patches, but with different variants for horizontal and vertical cell edge; in addition, left or right derivatives are computed if necessary. It saves us from passing some original arguments, and —more important— it saves us from the resulting conditions in the generated subroutines. For example, one of the variants, `FTAHLS`, replaces `FTA` on

[1]Here `MNOD` denotes the `Maximum Number Of Diagonals`,

Horizontal cell edge computing Left derivatives and it operates on multiple patches (a Subset). Inside an FTA variant, lists of patch numbers are pre-computed satisfying the subsonic and supersonic flow conditions considered in the Van Leer scheme; this is done completely vectorised. Next, the flux, the transport and the derivatives are computed (completely vectorised) using indirect addressing via the pre-computed lists.

2.2.2 Right-hand side computation

Originally, the right-hand sides were constructed using Scan over all patches that build up the grid. Vectorisation by looping over all patches on one level is not feasible because too many actions must be performed for each patch. Fluxes must be calculated and sent to the memory locations in DATA. Then, the residuals of the kids are collected, or, when there are no kids, a source term is evaluated. For the vectorisation it is better to split up the actions and to perform them on all patches residing on the same level. The new structure of the subroutine MkRhs, which computes the right-hand side, looks as follows:

```
call ZRhsV( ... )
if (lev .eq. TopLev) then
   call MkRhsTV( ... )
else
   call MkRhsKV( ... )
   call MkRhsHV( ... )
   call MkRhsVV( ... )
end if
```

These subroutines operate on multiple patches, listed in an index array, and they perform one of the following subtasks:

ZRhsV	– initialises the right-hand sides in DATA on a given level;
MkRhsTV	– evaluates the source term on the highest level and assigns the value to the corresponding DATA memory location;
MkRhsKV	– adds the right-hand sides of the kids or, when there are no kids, a source term is evaluated and assigned;
MkRhsHV	– calculates the horizontal fluxes and adds them to corresponding DATA memory locations of the cells and/or their (southern) neighbour;
MkRhsVV	– calculates the vertical fluxes and adds them to corresponding DATA memory locations of the cells and/or their (western) neighbours.

The computation of the fluxes is bounded by many restrictions: distinction should be made between green cell edges (i.e. edges at fine-coarse grid interfaces), boundary edges, and ordinary edges. Moreover, it should be known whether the patches have kids or not. Fortunately, the computation of the horizontal and vertical fluxes can be done independently, minimising the number of conditions for a patch.

Again it turns out to be convenient to use index arrays in order to save tests on properties of patches and to make vectorisation easier.

The flux transport computation across ordinary cell edges has been completely vectorised. For the green and boundary cell edges the flux computation required a slightly different approach. However, as soon as the left and right states have been computed for all green and boundary edges, the process can proceed analogous to the 'ordinary' case, i.e., completely vectorised. The fluxes are computed by the vectorised subroutines described in Section 2.2.1. Finally, sending of the flux values to the patches and/or their neighbours can be done in a straightforward way.

2.2.3 The residual

The computation of the residual on the composite grid (i.e., the grid consisting of all cells that have not been refined) corresponds to the previously discussed computation of the right-hand sides of the equations. First-order fluxes in horizontal and vertical direction must be computed. Again, it is possible to separate the horizontal and vertical part. For both directions index arrays are generated for the composite grid. The computational complexity for the residual computation and the right-hand side evaluation is roughly the same.

Originally, after the residual was computed, another `Scan` through the data structure was made to construct the L_1 and L_∞-norms of the residual fields. The weighting is done by a multiplication of the residual by a factor 4^{-l}, where l is the level on which the cell resides. This implies, that for each patch its weighting factor must be computed. Operating along the index array `LevelW` has the additional advantage of a constant weighting factor for each level.

Both norms can be calculated at vector speed using the BLAS subroutines `SASUM` and `ISAMAX`, respectively. Originally, the norms were computed simultaneously; in the modified code these norms are computed separately for each residual field.

2.2.4 The relaxation

Collective symmetric point Gauss-Seidel relaxation is used as the relaxation procedure in EULER. *Point* refers to the property that during the update of a state vector on a patch all other state vectors are kept fixed. *Collective* refers to the property that the update of the state vector on a patch is done for all of its four components simultaneously. *Symmetric* means that after a relaxation sweep a new sweep is made with the reverse ordering. For each cell visited during a relaxation sweep a system of four non-linear equations is approximately solved by Newton iteration, the differential operator being $(\partial/\partial u, \partial/\partial v, \partial/\partial c, \partial/\partial z)^T$; see [15] for definitions and details. We consider two possibilities for vectorising the relaxation:

1. Replacing symmetric Gauss-Seidel by a red-black ordering. This leads to an essentially worse convergence factor (based on other experiments we expect to loose a factor 5).
2. Staying with symmetric Gauss-Seidel, but using a diagonal ordering (since the non-linear systems on a grid-diagonal are completely decoupled) and replacing Osher's scheme by Van Leer's scheme. This results in longer vectors, because the number of choices in the upwind scheme reduces from 16 to 4.

A comparison of both choices shows a better efficiency for the latter approach. This means that, except for the replacement of Osher's flux-difference splitting scheme by Van Leer's flux-splitting scheme, we keep exactly the algorithm as used in the original EULER code. It can be easily seen that the original relaxation, using `Scan` with (SW, NW, SE, NE)-ordering, corresponds to the lexicographical ordering, which in turn is identical to the diagonal ordering because all non-linear systems on a diagonal are completely decoupled. In Section 2.1, we already described how an additional pointer structure has been added to facilitate working on diagonals.

We still have to describe how the decoupled non-linear systems on a diagonal are solved. First, the linearised (4×4) systems are constructed by use of the previously described flux, transport and derivatives subroutines. Next, a single Newton iteration step is performed, in which a 4×4 linear systems must be solved. For this, a subroutine was developed which contains one completely vectorisable loop, in which the loop body solves a 4×4 linear system using Gaussian elimination with partial pivoting. In order to vectorise this loop and to enhance the vector and even the scalar performance, the Gaussian elimination code for the solution of the 4×4 system is completely unrolled. A problem is that one or more of the linear systems might be singular. This is taken care of by replacing the matrix-diagonal elements by '1' and separately marking the systems singular. Finally, we notice that we do not know beforehand, how many Newton iteration steps are to be performed. After a Newton step on the patches in a grid-diagonal, we test whether the specified accuracy is reached, and then the patch numbers are collected of the non-linear systems that did not reach the required accuracy. The Newton process is continued on this probably much smaller subset. The iteration process is finished if either the subset becomes empty or a certain number of Newton steps has been performed.

2.2.5 Some other vectorised subroutines

Finally, we discuss some other interesting subroutines that have been vectorised, viz., `RstSol`, `BckUp` and `AddP1`. The first, `RstSol` computes a restriction of the solution on a given level, and sends it to the memory locations of the solution on another level. `BckUp` makes a copy from the solution. `AddP1` interpolates and adds the correction from one level to another. The `Scan` approach originally used being too expensive, in the new implementation the actions are performed directly for multiple patches and can be vectorised, including the `IF`-tests on the patch properties. Additional index arrays are not necessary.

2.3 Vector Performance

We consider the same two test problems that were used in Section 1.4. In this section, our main interest is the performance improvement obtained by vectorisation.

2.3.1 Shock reflection

The first test problem is the shock reflection problem considered in Section 1.4.1. Here the finest grid has dimensions 128×64, so the maximal diagonal length (being the maximal vector length) in the relaxation process is 64. The vector speed-up, measured for one FAS-cycle, can be found in Table 4. The total cost of maintaining the additional data structure, which allow vectorisation is 0.0142 CPU seconds, being only 2.5% of the total execution time.

2.3.2 Transonic airfoil flow

The second problem is the transonic flow problem around the NACA0012-airfoil, shown in Section 1.4.2. Beside the replacement of Osher's flux-difference splitting scheme by Van Leer's, the solution method differs for this problem due to the O-type grid. In this case the diagonal ordering no longer corresponds with the original lexicographical ordering. For the solution a cylindrical grid (a rectangular grid with coinciding lower

Table 4: Speed-up achieved with EUVEL on shock reflection problem. The CPU time is given in seconds.

routine	CPU time original	CPU time EUVEL	Speed up
AddP1	0.0107	0.0009	12
BckUp	0.0079	0.0002	40
MkRhs	0.2070	0.0279	7.4
Relax	2.1587	0.5294	4.1
RstSol	0.0106	0.0006	18
FAS	2.3949	0.5590	4.3
Res	0.1707	0.0208	8.2
NormRP	0.0275	0.0008	34

Table 5: Speed-up achieved with EUVEL on airfoil problem. The CPU time is given in seconds.

subroutine	CPU time original	CPU time EUVEL	Speed Up
AddP1	0.0229	0.0019	12
BckUp	0.0167	0.0004	40
MkRhs	0.4193	0.0641	6.5
Relax	4.4489	0.9914	4.5
RstSol	0.0427	0.0025	17
FAS Total	4.9510	1.0605	4.7
Res	0.3396	0.0347	9.8
NormRP	0.0500	0.0015	33

and upper boundaries) is used: the d-th diagonal, having d as sum of the ξ- and η-coordinates, extends over the lower boundary to the diagonal in the rectangular grid having $d + \mathtt{NY}$ as sum of the ξ- and η-coordinates. (NY is the number of points in the η-direction). In fact, the diagonals on the rectangular grid render into spirals on the cylindrical grid. In Table 5, we can find the vector speed-up for this problem.

2.4 Conclusions

Whereas, at first sight, the quad-tree data structure used for the self-adaptive algorithm seems not suitable for use on a vector computer, we have vectorised the EULER code, obtaining a vector speed-up factor of 4–5. To judge this speed-up we must bear in mind, first, the indirect addressing, necessary for the adaptive grid, and second, the relatively small average vector length due to the adaptive grid. Furthermore, the use of adaptive grids instead of uniformly refined grids has already decreased the amount of computational work by a factor 5–10 for realistic problems. We could obtain slightly higher vector speeds by changing the refinement criterion. However, in that case, the increase of computational work is not compensated by the increased vector speed.

The price to be paid for vectorising the original Euler code is the replacement of Osher's flux-difference splitting scheme by the somewhat less accurate Van Leer flux-splitting scheme. This is a relatively low price, because it is possible to compensate for this by applying defect correction with Osher's flux-difference splitting scheme.

As indicated in the introduction, vectorisation instead of parallelisation of the code was a well-considered approach; the greatest performance gain on a Cray Y-MP was expected from vectorisation. Parallelisation is still possible using domain decomposition or, chopping up the quad-tree. Compared to our vectorisation efforts, parallelisation should be relatively easy.

Requests for the modules BASIS, EULER and EUVEL can be directed to one of the e-mail addresses: barry@cwi.nl or walter@cwi.nl.

References

[1] G.D. van Albada, B. van Leer, and W.W. Roberts. A comparative study of computational methods in cosmic gas dynamics. *Astron. Astrophys.*, 108:76–84, 1982.

[2] K. Böhmer, P.W. Hemker, and H.J. Stetter. The defect correction approach. In K. Böhmer and H.J. Stetter, editors, *Defect Correction Methods: Theory and Applications*, volume 5 of *Comput. Suppl.*, pages 1–32. Springer, Wien, 1984.

[3] A. Brandt. Multilevel adaptive computations in fluid dynamics. *AIAA J.*, 18:1165–1172, 1980.

[4] A. Brandt. Guide to multigrid development. In W. Hackbusch and U. Trottenberg, editors, *Multigrid Methods*, number 960 in Lecture Notes in Mathematics, pages 220–312, Berlin, 1982. Springer.

[5] J.F. Dannenhoffer, III. Adaptive grid embedding for complex two-dimensional flows. In J.E. Flaherty, P.J. Paslow, M.S. Shephard, and J.D. Vasilakis, editors, *Adaptive Methods for Partial Differential Equations*, pages 68–82, Philadelphia, 1989. SIAM.

[6] J.-A. Désidéri and P.W. Hemker. Analysis of the convergence of iterative implicit and defect-correction algorithms for hyperbolic problems. *SIAM J. Sci. Comput.*, 16:88–118, 1995.

[7] W. Hackbusch. *Multi-Grid Methods and Applications.* Number 4 in Springer Series in Computational Mathematics. Springer, Berlin, 1985.

[8] W. Hackbusch and U. Trottenberg, editors. *Multigrid Methods II, Proceedings of the 2nd European Conference on Multigrid Methods, Cologne, 1985*, number 1228 in Lecture Notes in Mathematics, Berlin, 1986. Springer.

[9] P.W. Hemker. Defect correction and higher order schemes for the multi grid solution of the steady Euler equations. In Hackbusch and Trottenberg [8], pages 149–165.

[10] P.W. Hemker. On the order of prolongations and restrictions in multigrid procedures. *J. Comput. Appl. Math.*, 32:423–429, 1990.

[11] P.W. Hemker and G.M. Johnson. Multigrid approaches to the Euler equations. In S.F. McCormick, editor, *Multigrid Methods*, volume 3 of *Frontiers in Applied Mathematics*, pages 57–72, Philadelphia, PA, USA, 1987. SIAM.

[12] P.W. Hemker, B. Koren, W.M. Lioen, M. Nool, and H.T.M. van der Maarel. Multigrid for steady gas dynamics problems. In M. Hafez and K. Oshima, editors, *Computational Fluid Dynamics Review 1995*, pages 477–494, Chichester, 1995. Wiley.

[13] P.W. Hemker, H.T.M. van der Maarel, and C.T.H. Everaars. BASIS: A data structure for adaptive multigrid computations. Technical Report NM-R9014, CWI, Amsterdam, 1990.

[14] P.W. Hemker and S.P. Spekreijse. Multigrid solutions of the steady Euler equations. In D. Braess, W. Hackbusch, and U. Trottenberg, editors, *Notes on Numerical Fluid Mechanics*, volume 11, pages 33–44, Braunschweig, 1985. Vieweg.

[15] 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.*, 2:475–493, 1986.

[16] B. Koren. Defect correction and multigrid for the efficient and accurate computation of airfoil flows. *J. Comput. Phys.*, 77:183–206, 1988.

[17] B. Koren. Euler flow solutions for transonic shock wave – boundary layer interaction. *Int. J. Numer. Meth. Fluids*, 9:59–73, 1989.

[18] B. Koren. *Multigrid and Defect Correction for the Steady Navier-Stokes Equations, Application to Aerodynamics.* Number 74 in CWI Tracts. CWI, Amsterdam, 1990.

[19] B. Koren and H.T.M. van der Maarel. On steady, inviscid shock waves at continuously curved, convex surfaces. *Theor. Comput. Fluid Dyn.*, 4:177–195, 1993.

[20] B. van Leer. Flux-vector splitting for the Euler equations. In E. Krause, editor, *Proceedings Eighth International Conference on Numerical Methods in Fluid Dynamics*, number 170 in Lecture Notes in Physics, pages 507–512, Berlin, 1982. Springer.

[21] W.M. Lioen and M. Louter-Nool. EUVEL: An EULER Vector Extension Library. Technical Report NM-R9318, CWI, Amsterdam, 1993.

[22] H.T.M. van der Maarel. Adaptive multigrid for the steady Euler equations. *Comm. Appl. Numer. Meth.*, 8:749–760, 1992.

[23] H.T.M. van der Maarel. *A Local Grid Refinement Method for the Euler Equations.* CWI Tracts. CWI, Amsterdam, to appear.

[24] H.T.M. van der Maarel, P.W. Hemker, and C.T.H. Everaars. EULER: An adaptive Euler code. Technical Report NM-R9015, CWI, Amsterdam, 1990.

[25] H.T.M. van der Maarel and B. Koren. Spurious, zeroth-order entropy generation along a kinked wall. *Int. J. Numer. Meth. Fluids*, 13:1113–1129, 1991.

[26] S. Osher and F. Solomon. Upwind difference schemes for hyperbolic systems of conservation laws. *Math. Comput.*, 38:339–374, 1982.

[27] K.G. Powell, M.A. Beer, and G.W. Law. An adaptive embedded mesh procedure for leading-edge vortex flows. *AIAA Paper*, 89–0080, 1989.

[28] S.P. Spekreijse. Multigrid solution of monotone second-order discretizations of hyperbolic conservation laws. *Math. Comput.*, 49:135–155, 1986.

[29] S.P. Spekreijse. Second order accurate upwind solutions of the 2D steady Euler equations by the use of a defect correction method. In Hackbusch and Trottenberg [8], pages 285–300.

[30] S.P. Spekreijse. *Multigrid Solution of the Steady Euler Equations.* Number 46 in CWI Tracts. CWI, Amsterdam, 1988.

[31] H. Viviand. Numerical solutions of two-dimensional reference test cases. In *Test Cases for Inviscid Flow Field Methods*, number 211 in AGARD advisory report, pages 6-1–6-68. AGARD, Neuilly sur Seine, 1985.

[32] P. Wesseling. *An Introduction to Multigrid Methods.* Wiley, Chichester, 1991.