JOURNAL OF COMPUTATIONAL PHYSICS 90, 1–64 (1990)

Composite Overlapping Meshes for the Solution of Partial Differential Equations

G. CHESSHIRE AND W. D. HENSHAW

IBM Research Division, Thomas J. Watson Research Centre, Yorktown Heights, New York 10598

Received January 24, 1989; revised September 6, 1989

We discuss the generation of curvilinear composite overlapping grids and the numerical solution of partial differential equations on them. A composite overlapping grid consists of a set of curvilinear component grids that cover a region and overlap where they meet. Continuity conditions (interpolation) are imposed at the overlapping boundaries. The principal advantage of composite grids is in the generation of grids for regions with complicated geometries. The grid construction program CMPGRD is used to create composite grids with any number of component grids. We describe some techniques for the solution of elliptic and time-dependent PDEs on composite meshes. Applications to the solution of the compressible Navier–Stokes equations are presented. © 1990 Academic Press, Inc.

CONTENTS

0021-9991/90 $3.00

1. INTRODUCTION

We consider the solution of partial differential equations on composite overlapping grids. For the purposes of this article the term *composite (overlapping) grid* or *composite mesh* will have a particular meaning described briefly as follows. A composite grid is a set of *component* grids. Each component gid is a logically rectangular curvilinear grid. The union of the component grids covers the region on which the PDE is to be solved. The component grids overlap where they meet and functions defined on the grids are matched by interpolation. One of the principal advantages of composite grids is the generation of grids on regions of complicated geometry. We are primarily interested in the solution of PDEs by finite difference methods on such regions. Since each component grid is logically a rectangle, composite grids are well suited to finite difference applications, although in principle a composite mesh can be triangulated and used with finite element or finite volume methods. We use the grid generation program CMPGRD [13] to construct our composite grids. CMPGRD, which is written in standard Fortran, implements an algorithm for the creation of very general composite meshes. In Fig. 1 we show various composite grids which have been generated by CMPGRD. CMPGRD has recently been extended to construct three-dimensional composite grids. This paper, however, will only consider two-dimensional grids although much of what we say applies in three dimensions. We will give a brief description of some of the features of CMPGRD and we will discuss some of the principles of grid generation. We also give a short description of the data structure we use to store the composite grid data. A set of Fortran utility programs, known collectively as the DSK package, has been written to manage this and other similar data structures. Further information regarding CMPGRD and the DSK package can be found in "Getting Started with CMPGRD, Introductory User's Guide and Reference Manual" [11], "Composite Grid Data: An explanation of the CMPGRD composite grid data structure" [10], and "The DSK Package, A Data Structure for Efficient Fortran Array Storage (Reference Guide for the DSK Package)" [14].

After beginning with an overview of grid generation methods, Section 2, we proceed to discuss the creation of composite overlapping grids with CMPGRD in Sections 3 and 4. Next we give a brief description of some techniques for the solution of time dependent and elliptic partial differential equations on composite meshes, Section 5. We describe how to interpolate between grids and how to choose the order of interpolation. Finally, Section 6, we show results from solving the compressible Navier–Stokes equations on general two-dimensional regions. We use second-order finite differences combined with implicit–explicit time stepping to discretize the equations. Examples are shown of supersonic flow past a cylinder, flow around an airfoil with multiple flaps, and the flow around the read–write head of a magnetic storage device.

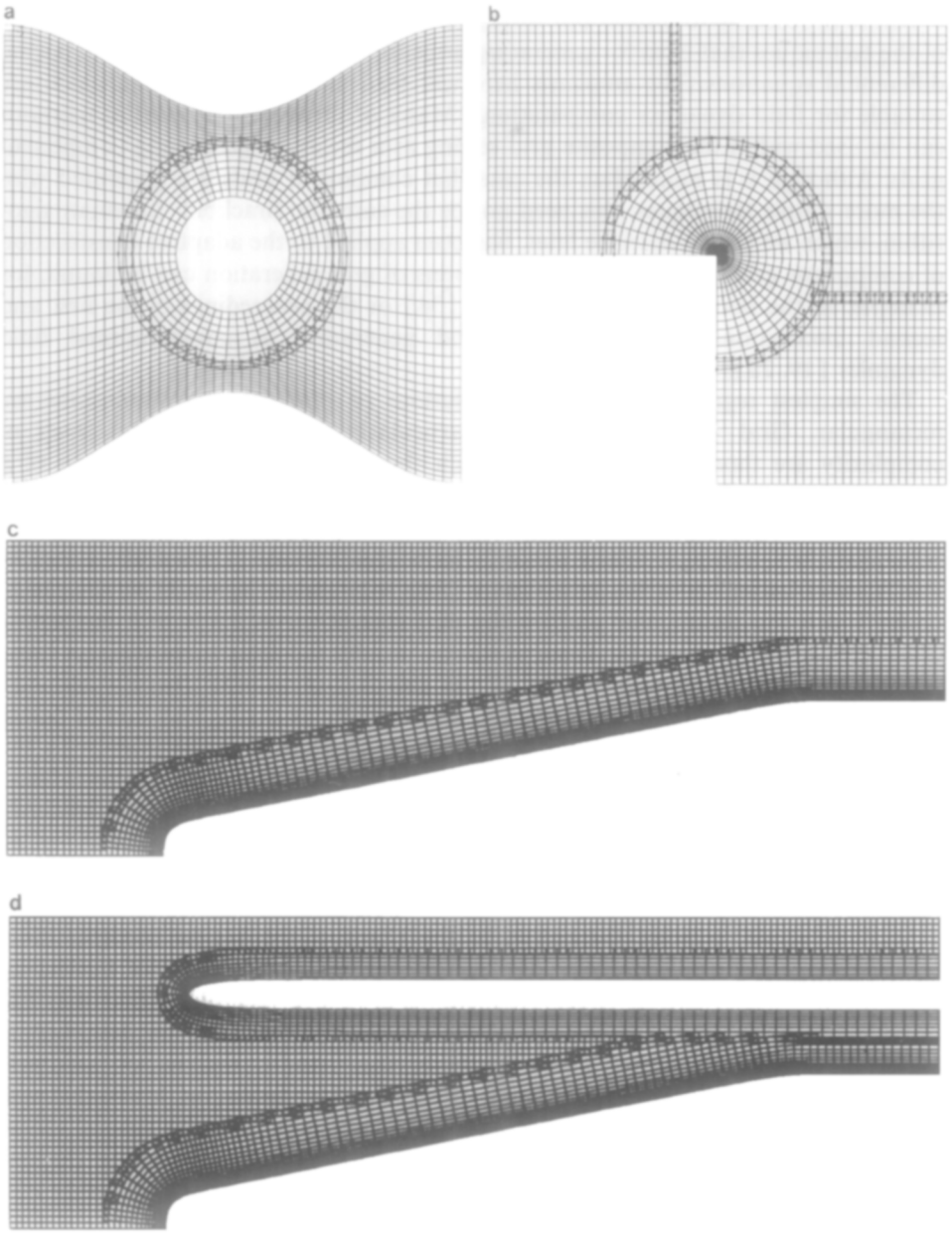


FIG. 1. Sample composite grids generated with CMPGRD.

2. Approaches to Grid Generation

Before proceeding to discuss our work, let us first give some background on some of the techniques that exist for generating grids for the numerical solution of partial differential equations. The emphasis here will tend to be towards grids which are suitable for fluid dynamics computations. Many of these grid generation methods can be used within the composite grid setting as a means of generating component grids. Although it is more often the case that component grids are generated with the more simple techniques, this need not be the case: the Brackbill–Saltzman algorithm, which we briefly describe later, has been applied to the adaptive construction of component grids. More detailed discussions of grid generation can be found, for example, in the book of Thompson *et al.* [34] or the proceedings of the first and second international conferences on grid generation in computational fluid dynamics [19, 28].

The simplest regions for computing solutions of PDEs in two dimensions are rectangular. Grid generation is then particularly easy. At the next level of complexity are those regions, **D**, for which there exists a smooth mapping of a rectangle onto **D**. The mapping can be used to generate a grid for **D**. In this case a single computer code can be written for all such regions since the grid is logically equivalent to a rectangle. Many methods have been devised to construct a one-to-one transformation from the unit square onto a bounded region **D** in the plane, under which the rectangular grid on the unit square corresponds to a boundary-fitted curvilinear grid on **D**. These methods usually fall into three classes: those which construct the transformation as the solution of a PDE (usually elliptic), those which construct the transformation algebraically as a weighted sum of points on the boundary (and in some cases other points describing curves interior to **D**), and those which construct the transformation from commonly known coordinate transformations by the appropriate choice of parameters. The latter methods readily provide transformations for such simple regions as a trapezoid or an annulus, but lack the versatility to handle regions of general shape. More sophisticated examples are the conformal Schwartz–Christoffel transformations for general polygonal regions. Generally, as the shape of the region becomes more complicated, these methods offer diminishing return for the effort of finding an appropriate transformation. Algebraic methods are more versatile in that they can provide transformations for regions whose boundaries need not be described in terms of simple functions. These methods can handle regions of relatively complicated geometries with a minimum of effort and computation. Methods based on PDEs are the most versatile and can be used for regions of quite general shape, but at the expense of the transformation being slower to compute than the one constructed by an algebraic method. Some PDE methods have the advantage of a maximum principle that ensures the transformation is one-to one.

The simplest algebraic method is the shearing transformation

$$\mathbf{d}(r, s) = (1 - r)\mathbf{c}_1(s) + r\mathbf{c}_2(s)$$

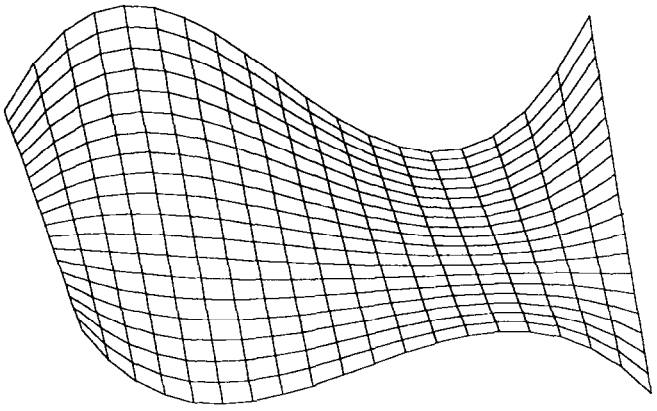


FIG. 2. Shearing transformation.

from the unit square onto the region bounded by the two curves $\mathbf{C}_k = \{\mathbf{c}_k(s) : 0 \leqslant s \leqslant 1\}$ and the line segments joining their endpoints at $s = 0$ and $s = 1$, Fig. 2. A generalization of the shearing transformation is the *Coons patch*, [15], also known as *transfinite interpolation*. The simplest form of the Coons patch is

$$\mathbf{d}(r, s) = (1-s)\mathbf{c}_1(r) + s\mathbf{c}_2(r) + (1-r)\mathbf{c}_3(s) + r\mathbf{c}_4(s) - (1-s)(1-r)\mathbf{c}_1(0) - (1-s)r\mathbf{c}_1(1) - s(1-r)\mathbf{c}_2(0) - sr\mathbf{c}_2(1),$$

which maps the unit square onto the region bounded by the four curves $\mathbf{C}_k = \{\mathbf{c}_k(s) : 0 \leqslant s \leqslant 1\}$, where it is assumed that $\mathbf{c}_1(0) = \mathbf{c}_3(0)$, $\mathbf{c}_1(1) = \mathbf{c}_4(0)$, $\mathbf{c}_2(0) = \mathbf{c}_3(1)$, and $\mathbf{c}_2(1) = \mathbf{c}_4(1)$, Fig. 3. Like the shearing transformation, the Coons patch is fast to compute. Its advantage is that it can handle a region whose boundary consists of four given curve segments. However, it has trouble with regions with convoluted boundaries; the resulting transformation may have folds where its Jacobian derivative is singular.

PDE methods for construction of coordinate transformations from the unit square onto a region $\mathbf{D}$ are generally more versatile than the algebraic methods in that they can handle regions of more complicated shape and that they offer greater control over resolution and skewness. This versatility comes at the expense of a higher computational cost. The most popular of these methods construct the transformation as the solution to an elliptic PDE boundary value problem, although methods based on hyperbolic and parabolic PDEs also exist.

The best known elliptic method for construction of a coordinate transformation is perhaps the conformal mapping. A conformal mapping can be found for any simply-connected region $\mathbf{D}$ in two dimensions, as the solution of the Cauchy–Riemann equations

$$\frac{\partial r}{\partial x} = \frac{\partial s}{\partial y}, \qquad \frac{\partial s}{\partial x} = -\frac{\partial r}{\partial y}$$

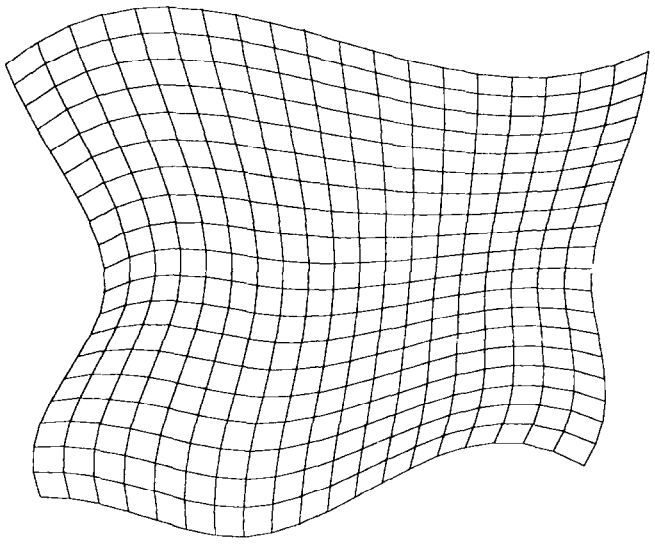


FIG. 3. Coons patch (transfinite interpolation).

in the interior of **D** with boundary geometry $(r, s) = \Psi(x, y)$ specified, where Ψ is a function (to be determined) that maps the boundary of **D** onto the boundary of the unit square. By differentiating and combining the Cauchy–Riemann equations we get Laplace's equation $\nabla^2(r, s) = 0$ in the interior of **D**. This equation can be used in the interior of **D** provided the Cauchy–Riemann equations are used as extra boundary conditions. Conformal maps offer no control over the resolution in the interior or on the boundary; grid lines near the boundary of **D** are more dense where the boundary is concave than where it is convex. Conformal maps usually have singularities on the boundary of **D** where the Jacobian derivative becomes zero or infinite. Furthermore, conformal mapping methods do not generalize to three dimensions.

As a more useful method for generating coordinate transformations, Winslow [36] proposed using Laplace's equation $\nabla^2(r, s) = 0$, as in the conformal mapping, but relaxing the boundary conditions. The boundary condition $(r, s) = \Psi(x, y)$ is imposed, with the function Ψ specified; the Cauchy–Riemann equations are not imposed at the boundary. This approach has the advantage over conformal mapping that it allows some control over the transformation near the boundary. In particular the boundary conditions can be chosen so that the corners of **D** correspond to corners of the unit square, so the transformation need not have singularities. Also, the transformation is faster to compute since it involves solving only a linear problem. An example of a grid generated from an elliptic equation (from [22]) is shown in Fig. 4. The method generalizes easily to three dimensions. A disadvantage is that the resulting grid is no longer orthogonal.

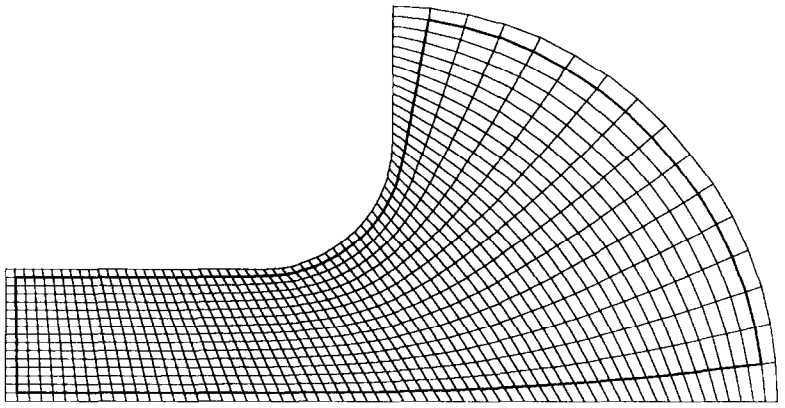


FIG. 4. Grid generated from an elliptic equation.

The use of elliptic systems for generating coordinate transformations was further generalized by Thompson, Thames, and Mastin [33] with the inclusion of forcing terms in the elliptic equations and branch cuts to allow for multiply-connected regions. The terms f and g in the equations $\nabla^2(r, s) = (f, g)$ allow greater control over the density of grid lines, not only near the boundary but also throughout the interior. The introduction of more than one branch cut, as necessary for many multiply-connected regions, results in a region **D** with more than four corners. In such cases some of the corners of **D** cannot correspond to corners of the unit square, and the transformation must have singularities there.

Brackbill and Saltzman [8] generate coordinate systems using a variational formulation. They find the transformation that maximizes a funtional $I = I_s + \lambda_v I_v + \lambda_0' I_0'$, where I_s is a global measure of the smoothness of the transformation, I_v is a global measure of resolution, and I_0' is a global measure of orthogonality. The Euler equation of this variational problem is an elliptic system of more complicated structure than that considered by Thompson. This method provides systematic global control over the resolution and other properties of the grid. This systematic control makes possible the automatic adaptive generation of grids.

There are a number of limitations which apply to the use of a single curvilinear grid on a region with a curved boundary. Even on a region with a fairly simple geometry it could be that the density of grid lines is forced to be highly nonuniform. For example, a curvilinear grid for an oblong region with a narrow waist will inevitably have a much higher density of grid lines in the narrow waist than in other parts of the region. In addition, with a single transformation any local changes to the shape of the region or local changes to the density of grid lines will have global effects on the grid. Another limitation is that, if the region is not doubly connected, any transformation must have a singularity on the boundary. For example, the Jacobian of any transformation from the unit square onto a disc (polar coordinates) will be singular at some corners of the square.

Thus, despite the many improvements in generating a grid by transforming the unit square to the computational domain there appears to be still a need for more flexible methods. One form of generalization is to allow for multiple transformations. *Patched grids* attempt to grid a region **D** with multiple curvilinear grids which join precisely along some common boundary, see, for example, [35]. Another approach, which permits great flexibility, is the use of unstructured triangular or tetrahedral grids discretizing with finite volume [25, 2] or finite element methods [16]. However, no matter how the equations are discretized, one of the most important points is the generation of a good grid. Whether one uses finite difference or finite element methods is then a matter of taste. Whether one method is better than another will often come down to details of implementation: how easy is it to program, how efficient is the method, can the code be vectorized/parallelized, how easy is it to implement higher-order methods or special techniques like upwind differencing, and so on.

The approach we consider here is that of *composite overlapping grids* (also called *overlaid grids*), where multiple grids are allowed to overlap. This approach is similar in some ways to the overlapping grids methods developed by Berger [5, 6] which have been successfully applied to a wide range of problems. The component grids in her approach are taken to be rectangles with emphasis on adaptively creating finer and finer rectangles in order to locally refine sharp features in the solution. In our case, however, the main emphasis is on generating a grid for a region with a complicated geometry, using curvilinear grids to generate boundary fitted coordinates and putting a smooth grid, without singularities, on multiply connected regions. We are also working on adaptive methods in which a curvilinear grid is automatically generated in order to resolve a feature such as a shock or layer. The idea is that a curvilinear grid can follow a curved shock using fewer grid points than a sequence of rectangular grids. Overlapping composite grids are more flexible than patched grids, since component grids are permitted to overlap as opposed to being forced to align along a particular curve. (A patched grid can in some sense be thought of as an overlapping grid with exactly zero overlap.)

The composite overlapping grid technique has been in use for some time. The method has been promoted by Professor Heinz Kreiss for a number of years. For example, in 1977, Starius [29], who was a student of Professor Kreiss, looked at the convergence of elliptic problems on two overlapping meshes using the Schwartz alternating procedure. In a later paper [30] he considered the numerical solution of hyperbolic problems. The stability of the Lax–Wendroff method was shown for a model problem on a one-dimensional overlapping mesh. Moreover, he solved the shallow water equations in a two-dimensional basin, showing that despite the overlap the mass was conserved to within a few percent. In his Ph.D. thesis, Reyna [27] obtained further stability results for the method of lines. Reyna gave examples of the accuracy of the method in solving the wave equation and also presented some calculations of flow in a circular basin where composite meshes were useful in order to remove the singularity of a polar coordinate mesh. A method for the construction of composite meshes and the solution of hyperbolic PDEs was described by

B. Kreiss [23]. The idea for the CMPGRD code came from this original program. However, the current version of CMPGRD, with its general algorithm for determining the overlapping regions, its interpolation procedures, data structures, and interactive graphics, is radically different from the code of B. Kreiss. Pärt [26] has adapted and extended an early version of CMPGRD for finite volume calculations. The current version of CMPGRD can also generate grids for finite volume computations, where variables are interpolated at the cell centers as opposed to cell vertices. Furthermore, there are extended discussions of composite grids in the theses of Henshaw [21] and Chesshire [13].

Besides the people working with the CMPGRD code there are a number of other groups currently working on composite meshes. The review article by Steger and Buning reviews some of this work [31]; also see the articles in [19, 28]. Atta and Vadyak [1], describe the use of composite grids for flows about two- and three-dimensional aircraft configurations. Steger *et al.* [12] present some very nice calculations of the three-dimensional flow around the space shuttle, external tank, and solid rocket booster assembly using composite grids which they have constructed with their *Chimera* code [3, 4, 12, 32].

The major distinguishing features between these different approaches to composite meshes lie in the grid construction algorithm, the manner of performing interpolation, the data structures, and the details of implementation. We have attempted, in our implementation, to be very flexible. Besides being able to handle any number of component grids, for example, CMPGRD can generate a composite grid which can be used for fourth- or higher-order spatial discretizations with the appropriate high-order interpolation which is needed at overlapping grid boundaries. Moreover, CMPGRD can automatically generate the sequence of coarser and coarser grids needed in the multigrid algorithm as described in "Multigrid on Composite Meshes" [20]. The creation of a general code for generating overlapping grids such as CMPGRD or the Chimera scheme mentioned above is not an easy task. The current version of CMPGRD, not including any graphics interface, is on the order of 10^4 lines long.

3. Composite Grid Construction with CMPGRD

3.1. *Step I: Creating Component Grids*

The composite grid construction program CMPGRD [13] can be used to generate very general composite overlapping grids. A composite grid can consist of any number of component grids. Each component grid is a logically rectangular curvilinear grid, which is permitted to overlap any other component grid. The idea in constructing a composite grid for a region $\mathbf{D}$ is to divide the region into simple enough subregions, $\mathbf{D}_k$, so that each subregion can be easily covered with a component mesh. With CMPGRD the component grid is thought of as a mapping $\mathbf{d}_k$ from the unit (r, s) square onto the physical domain $\mathbf{D}_k$, $\mathbf{d}_k$: $[0, 1] \times [0, 1] \rightarrow \mathbf{D}_k$. In many applications boundary fitted grids are desirable. Often the first step in

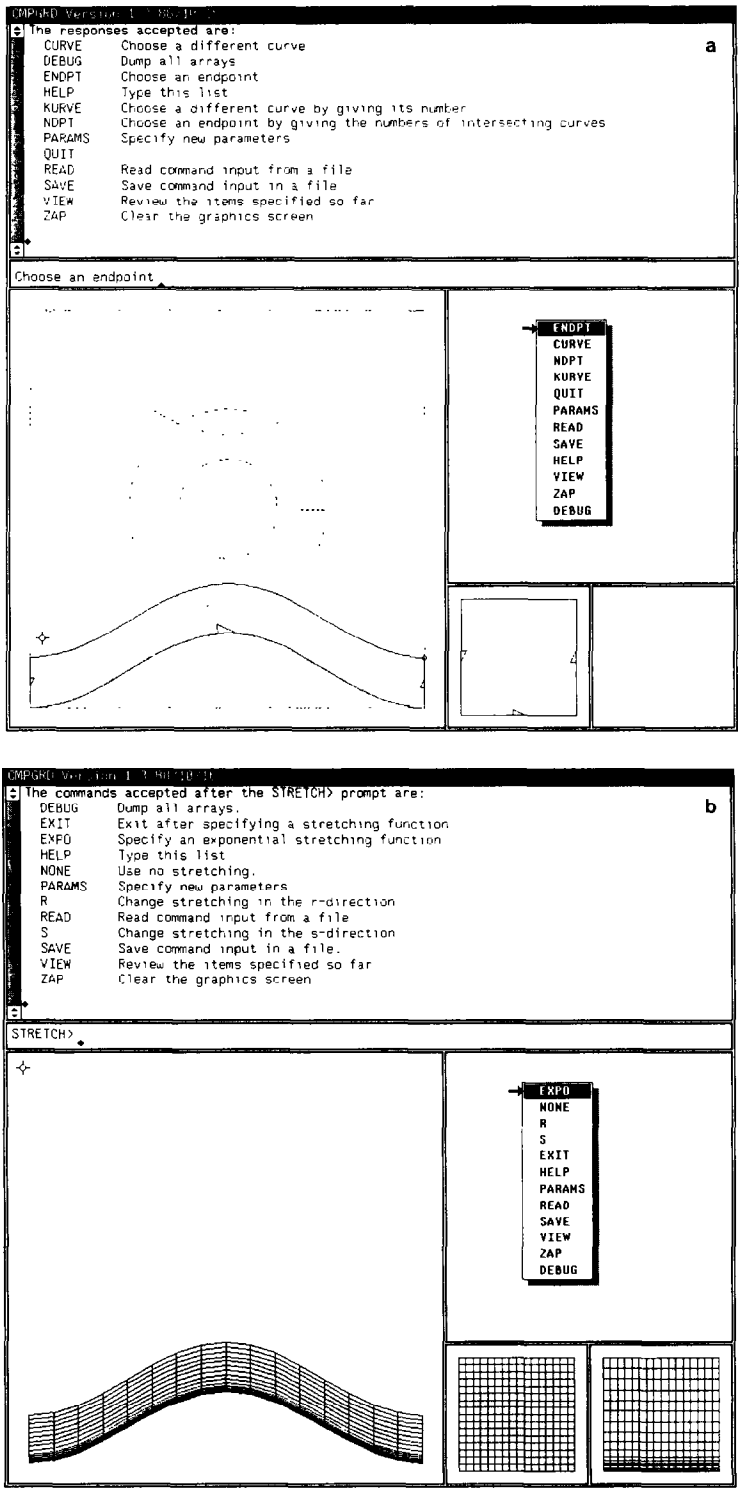


FIG. 5. Using CMPGRD to construct a composite grid.

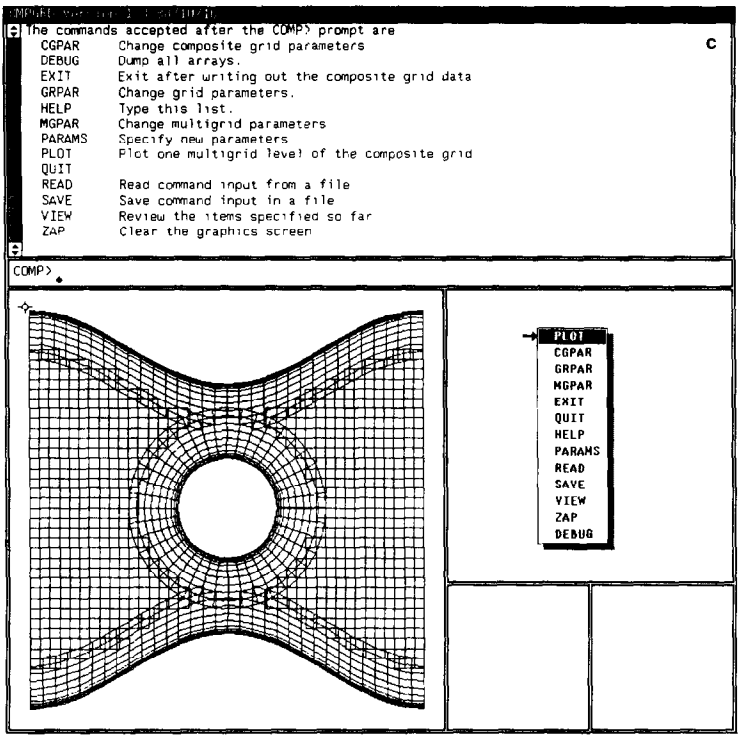


FIGURE 5—*Continued*

choosing a set of component meshes is to fit grids along the boundaries, using a smooth portion of the boundary as one side of the component grid and extending the boundary curve a short way into the interior to form the opposite side of the grid. A standard procedure here is to simply use the normals to the boundary to define one set of coordinate lines. This will work provided the width of the boundary grid is not too large compared to the curvature of the boundary. After all boundaries have been handled in this way, the interior part of the region can then be covered with a single rectangular grid, meaning that a significant portion of many domains can be discretized with a computationally efficient rectangular grid. The big advantage of composite grids is that each component grid can be generated almost independently of the other component grids. Grid lines on boundary fitted grids can be stretched next to the boundary if one anticipates needing extra resolution there. Of course, the component grids are not completely independent as their union must cover the entire region. Moreover, it is usually wise to make sure that where two component grids overlap the grid spacings are about the same on each grid. There is little sense in having a very fine grid next to a very coarse one, since the numerical solution must be smoothly represented on each grid and the fine grids points would thus be wasted.

Component grids can be specified to CMPGRD in a number of ways. One way is to supply a subroutine which defines the component grid. This means that the subroutine must supply the transformation which maps the unit (r, s) square to the (x, y) domain as well as the first derivatives of this mapping, $\partial x/\partial r$, $\partial x/\partial s$, $\partial y/\partial r$, and $\partial y/\partial s$.

Another way to create component grids is to use facilities supplied by CMPGRD. CMPGRD has an advanced graphical interface for interactively creating component grids. CMPGRD has been run on the IBM VM/CMS system, the CRAY CTSS system, the DEC VMS system, and various UNIX systems. Graphics routines have been written for a number of devices including the Tektronix 4014 and 4105, the IBM 3279 with GDDM, Sun workstations running the SunView window system, and the Apple Macintosh II. In Fig. 5 we show the screen of a graphics terminal at three stages in the construction of a composite grid. In this case the graphics is being performed on a Sun terminal running the SunView window system. There are a number of windows appearing on the screen which are used to display messages and plot results.

In the interactive mode of creating grids one must first create the curves from which the sides of the component grid will be constructed. These curves can be predefined through a user-supplied subroutine or they can be defined interactively, using the cursor to pick out points and a spline to define the curve. In Fig. 5a the predefined curves to be used for generating a grid for a converging channel with a hole in the middle are shown (lower left window). Some curves are obviously needed as they correspond to sides of the region. The other curves are added to complete sides of component grids. Each component grid is defined in terms of segments of these curves. That is, a portion of a curve will define one side of a component grid. The user is prompted, using the cursor, to pick out the segments of the curves which correspond to each of the four sides of the component grid, Fig. 5a. A pop-up menu indicates the various choices which are available for the user at any given time. Once the four sides of a component grid have been chosen the grid transformation is defined through a mapping function such as a Coons patch (Section 2). At this stage one also chooses the number of grid lines for the component grid and any stretching of grid lines, Fig. 5b. CMPGRD permits the user to choose from a variety of *stretching functions* which will cluster grid lines. The stretching is performed by first mapping the unit (r, s) square which has uniform grid spacing into another unit square, (t, u), on which the grid lines are clustered. For simplicity, the stretched coordinate t is a function of r only, while u is a function of s only. This stretched unit square is then mapped by a Coons patch to (x, y) space:

$$(r, s) \xrightarrow[\text{functions}]{\text{stretching}} (t, u) \xrightarrow[\text{functions}]{\text{mapping}} (x, y).$$

The currently available stretching functions are of two types. The first type will concentrate points into a fine cluster near a given point. This stretching function is defined as

$$r = U_i(t) = \frac{a_i}{2} \tanh b_i(t - c_i),$$

as shown in Fig. 6. Here a_i, b_i, and c_i are parameters which can be chosen to obtain suitable concentrations of grid lines. The second type of stretching functions permits the transition from one fixed mesh spacing to a second fixed mesh spacing and takes the form

$$r = V_j(t) = \frac{d_j - 1}{2e_j} \log\left(\frac{\cosh e_j(t - f_j)}{\cosh e_j(t - f_{j+1})}\right),$$

Fig. 7. In general one can choose a combination of these functions:

$$r = R(t) = \left[t + \sum_{i=1}^{N_U} (U_i(t) - U_i(0)) + \sum_{j=1}^{N_V} (V_j(t) - V_j(0))\right] C_1 + C_0,$$

where C_0 and C_1 are chosen so that $R(0) = 0$ and $R(1) = 1$. The function $U_i(t)$ is a hyperbolic tangent centered at $t = c_i$ and asymptoting to $-a_i/2$ or $a_i/2$. As b_i tends to infinity the function U tends toward a step function. The function $V_j(t)$ is a smoothed out ramp function with transitions at f_j and f_{j+1}. The slope of the ramp is $d_j - 1$. Thus d_j indicates the relative slope of the ramp compared to the linear term t which appears in $R(t)$. In other words, the grid spacing between f_j and f_{j+1} is approximately d_j times the grid spacing in the region where the linear term is dominant. By adding a correction, the stretching function $R(t)$ can be made suitable for periodic regions. In Fig. 8 we present some examples of stretching functions. The equally spaced grid in r is marked on the vertical axis while the stretched grid points are marked on the horizontal t axis. Some care is required to implement the evaluation of stretching functions and their inverses to be efficient and to avoid underflows and overflows.

3.2. *Step II: Generation of the Composite Grid*

Once all component meshes have been defined, the next step is the generation of the actual composite grid, Fig. 5c. The algorithm for the generation of a composite

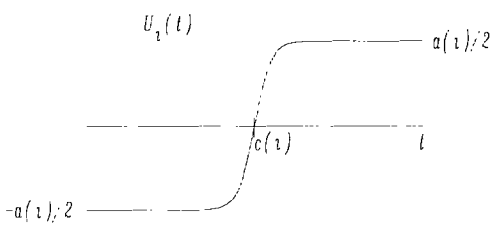


FIG. 6. Stretching function U_i.

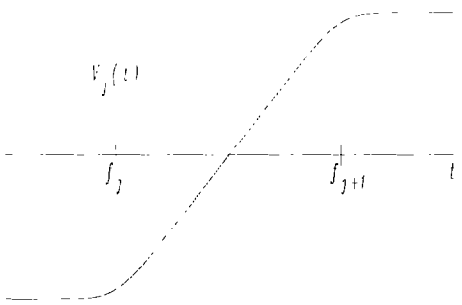


FIG. 7. Stretching function V_j.

grid must detect regions of overlap and determine the points to be used for interpolation between component grids. In addition, the algorithm should recognize regions of grid points which are not needed in the computation. The algorithm we present here is fundamentally different from those used by B. Kreiss [23] or Benek *et al.* [3]. We consider this algorithm to be, by far, the most important and most difficult result of this paper.

A valid composite grid $\mathbf{G}$ consists of a set of component grids $\mathbf{G}_k$, labelled by k, $k = 1, ..., n_g$. Each component grid is logically rectangular. We will label each point of $\mathbf{G}_k$ as either (i, j, k), or $\mathbf{x}_{ijk} = (x_{ijk}, y_{ijk})$, or (r_{ijk}, s_{ijk}), depending on whether we are considering the indices, the physical space coordinates, or the unit square coordinates. The particular composite grid which is generated will depend upon a number of parameters including

(1) the width of the interpolation formula,

(2) the width of the discretization formula,

(3) the minimum allowable overlap,

(4) the ordering of the component grids.

As the width of the interpolation formula is increased, for example, the amount of overlap will also increase, since the interpolation is required to be sufficiently centered (Section 5.3). On the other hand, as the width of the discretization formula is increased, the amount of overlap increases because the number of lines of interpolation points will increase. A centered difference formula which is three points wide will need only one interpolation point while a formula which is five points wide will need two points of interpolation. CMPGRD is quite flexible with respect to these parameters. For example, the interpolation width parameter, $iw(m, k_1, k_2)$, gives the width, in the r ($m = 1$) or s ($m = 2$) direction, of the interpolation formula for points on component grid $\mathbf{G}_{k_1}$ that interpolate from component grid $\mathbf{G}_{k_2}$. Of course, one normally does not want all this flexibility, so CMPGRD provides reasonable default values.

The algorithm to create a composite grid must start with a set of component

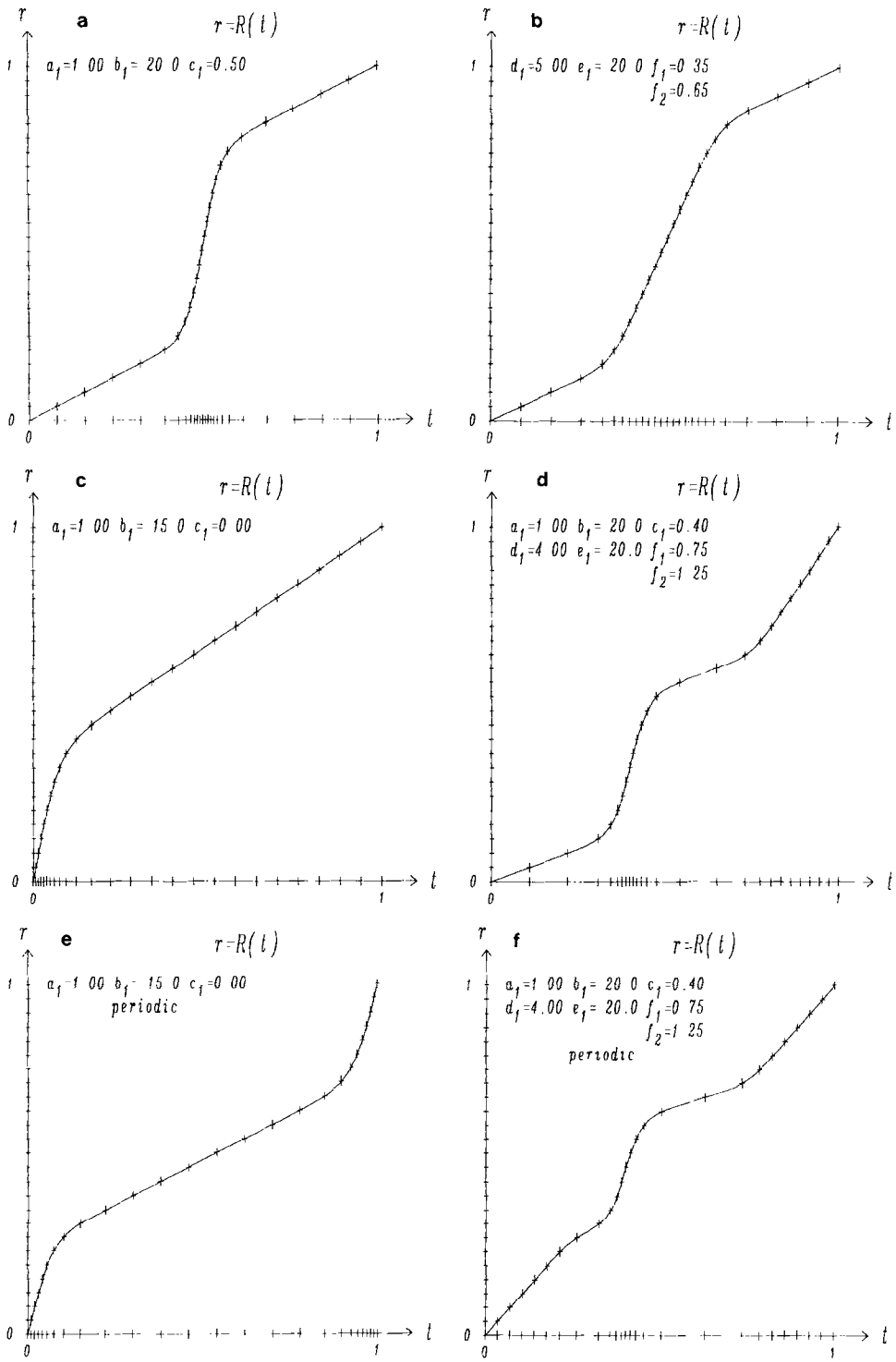


FIG. 8. Stretching functions for concentrating grid lines. The stretched grid is plotted on the t axis.

581 90/1-2

grids and a set of constraints defined by the interpolation and discretization formulae and find a valid composite grid which satisfies all the constraints. Each point (i, j, k) on a valid composite grid must be one of the following:

(1) *Discretization point.* A discretization point is either an *interior point* or a *boundary point.* Point (i, j, k) is called an interior point if it can be discretized, to the required order, in terms of points on component grid $\mathbf{G}_k$ which are interior points, boundary points, or interpolation points. The discretization is assumed to be centered so that a rectangle of points is required; the width and height of the rectangle being specified by the user. Point (i, j, k) is called a boundary point if it lies on the true boundary and can be discretized to the required order in terms of points on grid $\mathbf{G}_k$ which are interior points, boundary points, or interpolation points. (Boundary condition discretizations may be different from interior point discretizations.)

(2) *Interpolation point.* Point (i, j, k) is an interpolation point if it can be interpolated from discretization or interpolation points on another component grid $\mathbf{G}_{k'}$ with $k' \neq k$ to the required order. (We further require that the interpolation be sufficiently centered, see Section 5.2 for details.)

(3) *Exterior or unused point.* Point (i, j, k) is an exterior or unused point if it is not a discretization or interpolation point.

It is possible that a given point could belong to more than one of the above categories. For example, some discretization points might just as well be interpolation points. However, for efficiency we try to create a composite grid with a minimum number of interpolation points. It is possible that the only valid composite grid is one consisting entirely of unused points, a *null grid.* This situation can be avoided by having enough overlap between component grids.

The basic idea in our composite grid construction algorithm is to think of ordering the component grids, $k = 1, 2, ..., n_g$ so that higher-numbered grids *cover over* parts of lower-numbered component grids. Points which are removed will be in general lie *underneath* a higher numbered component grid. This is illustrated in Fig. 9, where we have shown how changing the ordering of the component grids affects the resulting composite grid. In this example one should notice, however, that all component grids have lost grid points, including the one that lies on top of all other component grids. In fact our algorithm is general enough so that if a valid composite grid is created for some ordering of component grids then a valid composite grid should result for all orderings of component grids (although we have not attempted to prove this).

We now outline the *composite grid algorithm* for constructing a composite grid from a set of component grids (Fig. 10). We combine an explanation in words with brief sections of pseudo-code. The composite grid is described by a flag array, $kr(i, j, k)$ (where *kr* stands for *koordinates*). This array contains a code for each point on the composite grid, indicating whether the point (i, j, k) is a discretization,

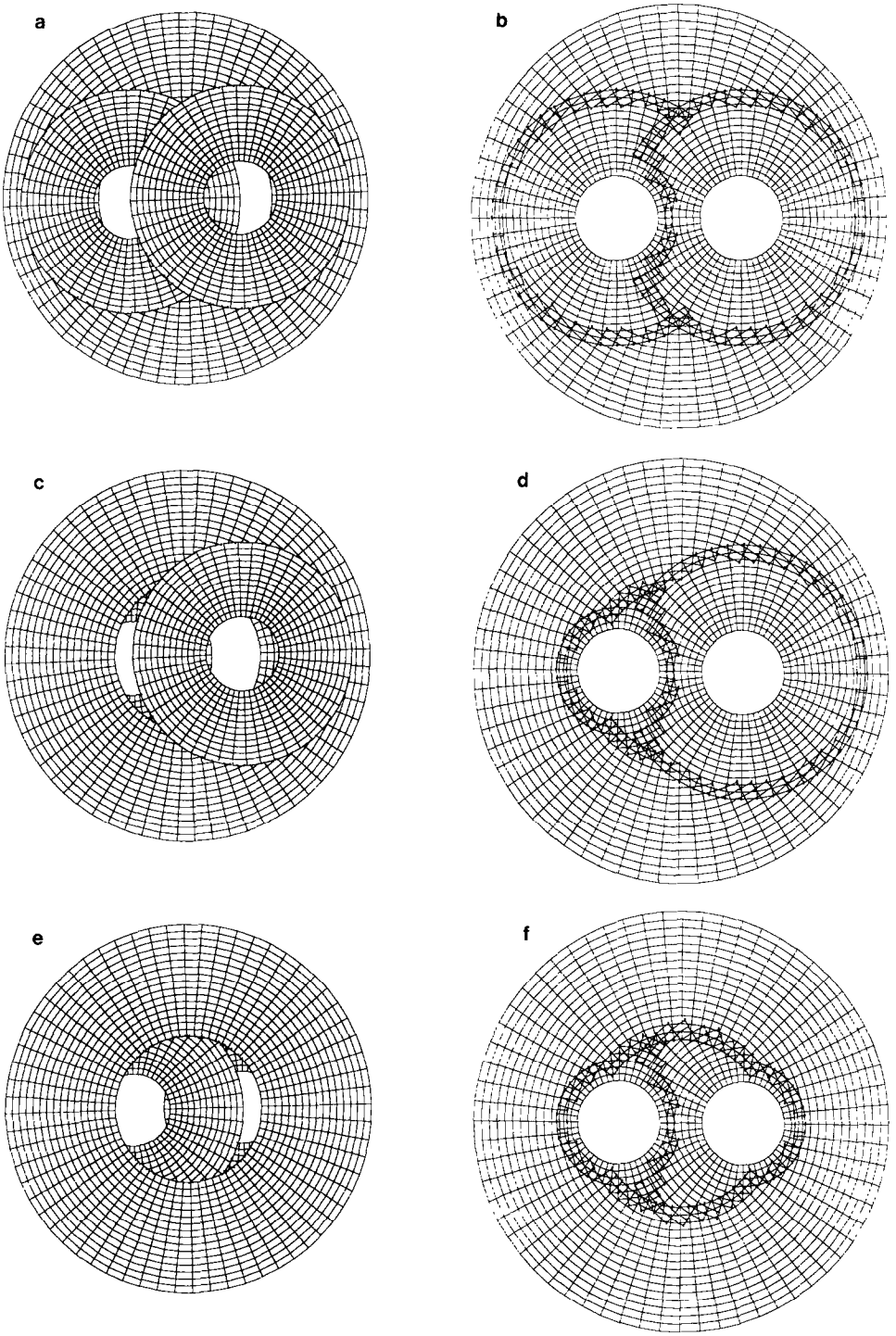


FIG. 9. A composite grid depends on the ordering of the grids.

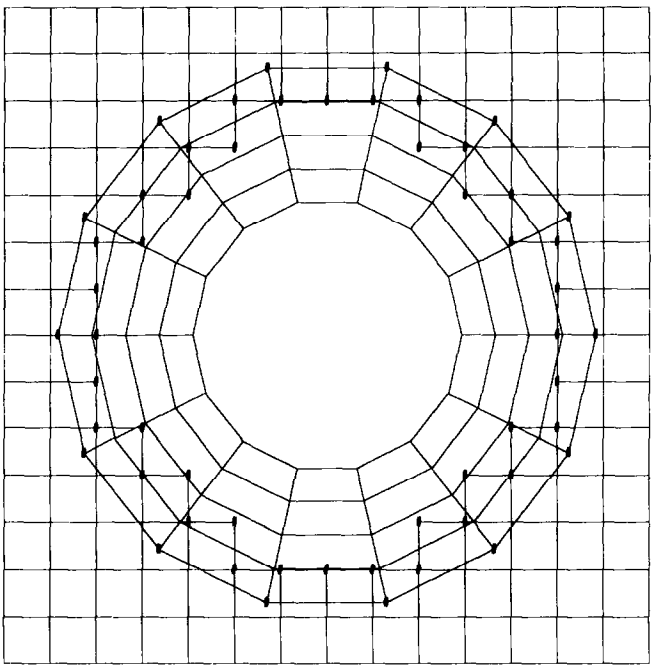


FIG. 10. Grid used to describe the composite grid algorithm.

exterior of interpolation point. By the end of the algorithm the values in this array will have the following meaning

$$kr(i, j, k) = \begin{cases} k & \text{if } (i, j, k) \text{ is a discretization point} \\ -k' & \text{if } (i, j, k) \text{ is interpolated from grid } k' \neq k \\ 0 & \text{if } (i, j, k) \text{ is an exterior point.} \end{cases}$$

The *first step* is to initialize each grid point by assigning the number of component grids n_g into each point of the flag array. The reason this is done should become clear when we describe step 3.

for $k = 1, \ldots n_g$ **do**
  for $(i, j, k) \in \mathbf{G}_k$ **do**
    $kr(i, j, k) \leftarrow n_g$
  end for
end for

The *second step* is to mark non-boundary points that lie close to a boundary side of another grid as exterior points, $kr(i, j, k) = 0$. This is done so that if a grid extends outside the region, then those of its gridpoints which lie exterior to the computational domain will be marked as exterior, as they should be. Although this

procedure will not find all exterior points, in the steps to follow the set of exterior points will expand to fill out the exterior regions:

```
for k = 1, ..., n_g do
  for every boundary point x_b ∈ G_k do
    for k' = 1, ..., n_g and k' ≠ k do
      if x_b ∈ D_k' and x(i', j', k') is the closest point of G_k' to x_b then
        kr(i', j', k') ← 0
      end if
    end for
  end for
end for
```

The *third step* is to find which grid, if any, each point on each grid can be interpolated from. Starting from the highest numbered grid and working down to the lowest numbered grid, each grid point on $\mathbf{G}_k$ is examined to find the highest $\mathbf{G}_{k'}$ with $k' > k$, if any, from which it can be interpolated. If the point cannot be interpolated from a higher grid and is not a valid discretization point then we find the highest $\mathbf{G}_{k'}$ with $k' < k$, if any, from which it can be interpolated. If the point cannot be interpolated from a lower grid either, it is marked as an exterior point. At the end of this step every point is marked either as an exterior point, $kr(i, j, k) = 0$, an interior point $kr(i, j, k) = k$, or as interpolating from grid k', $kr(i, j, k) = k' \neq k$.

```
repeat until there are no more changes
  for k = 1, ..., n_g do
    for (i, j, k) ∈ G_k do
      k' ← kr(i, j, k)
      repeat
        if k' = k then
          if(i, j, k) is not a valid discretization point then
            kr(i, j, k) ← kr(i, j, k) − 1
          end if
        elseif (i, j, k) cannot be interpolated from G_k' then
          kr(i, j, k) ← kr(i, j, k) − 1
        end if
        k' ← k' − 1
      until k' = 0 or (i, j, k) is a valid point
    end for
  end for
end repeat
```

In the *fourth step* we mark all points on lower grids which are definitely needed for interpolation by higher grids. We mark points in this way in preparation for the following step when we will be deleting unnecessary interpolation points. An interpolation point is unnecessary if it is not needed by any discretization points on the

same grid and it is not needed by any interpolation points on other grids. At the end of this step all points needed for interpolation by higher grids are marked as $kr(i, j, k) < 0$.

```
for k = 1, ..., n_g do
  for (i, j, k) ∈ G_k do
    if k' := kr(i, j, k) < k and k' > 0 then
      (Point (i, j, k) interpolates from G_k')
      for all points (i', j', k') needed to interpolate (i, j, k) do
        kr(i', j', k') ← −|kr(i', j', k')|
      end for
    end if
  end for
end for
```

In the *fifth step* we start from the lowest grid and proceed to the highest grid. For each grid we delete interpolation points which are not needed and change interpolation points into discretization points if possible. We also mark points on higher grids which are needed for interpolation by lower grids.

```
for k = 1, ..., n_g do
  for every interpolation point (i, j, k) ∈ G_k do
    if (i, j, k) is not needed then
      kr(i, j, k) ← 0
    end if
  end for
  for every interpolation point (i, j, k) ∈ G_k do
    if (i, j, k) can be an interior point then
      kr(i, j, k) ← k
    end if
  end for
  for every interpolation point (i, j, k) ∈ G_k do
    mark points on upper grids needed for interpolation:
    kr(i', j', k') − |kr(i', j', k')|
  end for
end for
```

At this point the composite mesh has been determined. The *sixth step* simply consists of changing the sign of the entries in the *kr* array so that discretization points are positive and interpolation points are negative:

```
for k = 1, ..., n_g do
  for (i, j, k) ∈ G_k do
    if |kr(i, j, k)| = k then
      (Discretization point)
      kr(i, j, k) ← |kr(i', j', k')|
```

```
    elseif |kr(i, j, k)| > 0 then
      (Interpolation point)
      kr(i, j, k) ← −|kr(i, j, k)|
    end if
  end for
end for
```

To reiterate, each point is an interior, exterior or interpolation point as indicated by

$$kr(i,j,k)=\begin{cases} k & \text{if } (i,j,k) \text{ is a discretization point} \\ -k' & \text{if } (i,j,k) \text{ is interpolated from grid } k' \neq k \\ 0 & \text{if } (i,j,k) \text{ is an exterior point.} \end{cases}$$

As an example we show the actual values in the kr array after each step in the composite grid algorithm for the composite grid shown in Fig. 11.

One of the most important operations which is performed many times in the above algorithm is the task of determining whether a given spatial point $\mathbf{x}$ can be interpolated from a given component grid. More generally if the point $\mathbf{x}$ lies somewhere in the component grid then we need to know the (r, s) coordinates of this point; that is, we must invert the transformation, $(r, s) = \mathbf{d}_k^{-1}(\mathbf{x})$ which defines the component grid. It is essential to perform this operation as quickly as possible. The algorithm we use is of the form:

(1) First check if $\mathbf{x}$ lies inside a rectangle that bounds the component grid. If not then it cannot be interpolated and we are done.

(2) If $\mathbf{x}$ lies in the rectangle then we try to invert the transformation which defines the grid. To get an initial guess for this inversion step we first find the closest grid point to $\mathbf{x}$. Let us assume we have an initial guess to the closest grid point, then check the neighbours of this point to see if any are closer. If a neighbour is closer to $\mathbf{x}$ than the current guess, make the neighbour the current guess. Continue until a local minimum is reached. If the local minimum is on the boundary, do a global search of the boundary points to determine the one with minimum distance. Now repeat the local search once again. At this point the nearest grid point to $\mathbf{x}$ will have been found (provided the transformation which defines the grid is sufficiently smooth).

(3) Now determine the (r, s) coordinates of the point by inverting the component grid transformation with a Newton iteration, using the closest point as an initial guess.

We emphasize the above point, since the construction of a composite grid requires some computation and it is important to do certain things efficiently. We estimate that the number of operations to construct the composite mesh is on the order of $(n_g)^2 N$, where n_g is the number of component grids and N is the total number of grid points. In the last section we present some timings on the creation of the grid compared with the solution of a PDE on the resulting mesh.

a

```
           kr(*,*,1)                         kr(*,*,2)
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2                2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2                2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2                2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2                2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2                2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2                2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2                2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2                2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2                2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2                2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2                2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2                2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2                2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2                2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2                2 2 2 2 2
```

b

```
           kr(*,*,1)                         kr(*,*,2)
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2                2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2                2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2                2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2                2 2 2 2 2
2 2 2 2 2 2 0 2 0 2 2 2 2 2 2                2 2 2 2 2
2 2 2 2 2 0 2 2 2 0 2 2 2 2 2                2 2 2 2 2
2 2 2 2 0 2 2 2 2 2 0 2 2 2 2                2 2 2 2 2
2 2 2 2 0 2 2 2 2 2 0 2 2 2 2                2 2 2 2 2
2 2 2 2 0 2 2 2 2 2 0 2 2 2 2                2 2 2 2 2
2 2 2 2 2 0 2 2 2 0 2 2 2 2 2                2 2 2 2 2
2 2 2 2 2 2 0 2 0 2 2 2 2 2 2                2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2                2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2                2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2                2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2                2 2 2 2 2
```

c

```
           kr(*,*,1)                         kr(*,*,2)
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1                1 2 2 2 2
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1                1 2 2 2 2
1 1 1 1 1 2 2 2 2 2 1 1 1 1 1                1 2 2 2 2
1 1 1 1 2 2 2 2 2 2 2 1 1 1 1                1 2 2 2 2
1 1 1 2 2 2 0 2 0 2 2 2 1 1 1                1 2 2 2 2
1 1 2 2 2 0 0 0 0 0 2 2 2 1 1                1 2 2 2 2
1 1 2 2 0 0 0 0 0 0 0 2 2 1 1                1 2 2 2 2
1 1 2 2 0 0 0 0 0 0 0 2 2 1 1                1 2 2 2 2
1 1 2 2 0 0 0 0 0 0 0 2 2 1 1                1 2 2 2 2
1 1 2 2 2 0 0 0 0 0 2 2 2 1 1                1 2 2 2 2
1 1 1 2 2 2 0 2 0 2 2 2 1 1 1                1 2 2 2 2
1 1 1 1 2 2 2 2 2 2 2 1 1 1 1                1 2 2 2 2
1 1 1 1 1 2 2 2 2 2 1 1 1 1 1                1 2 2 2 2
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1                1 2 2 2 2
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1                1 2 2 2 2
```

FIG. 11. *kr* array during each step of the composite grid algorithm.

d

kr(*,*,1)

```
 1  1  1  1  1 -1 -1 -1 -1 -1  1  1  1  1  1
 1  1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1  1  1
 1  1 -1 -1 -1 -2 -2 -2 -2 -2 -1 -1 -1  1  1
 1 -1 -1 -1 -2  2  2  2  2  2 -2 -1 -1 -1  1
 1 -1 -1 -2  2  2  0  2  0  2  2 -2 -1 -1  1
 1 -1 -2 -2  2  0  0  0  0  0  2 -2 -2 -1  1
-1 -1 -2  2  0  0  0  0  0  0  0  2 -2 -1 -1
-1 -1 -2  2  0  0  0  0  0  0  0  2 -2 -1 -1
-1 -1 -2  2  0  0  0  0  0  0  0  2 -2 -1 -1
 1 -1 -2 -2  2  0  0  0  0  0  2 -2 -2 -1  1
 1 -1 -1 -2  2  2  0  2  0  2  2 -2 -1 -1  1
 1 -1 -1 -1 -2  2  2  2  2  2 -2 -1 -1 -1  1
 1  1 -1 -1 -1 -2 -2 -2 -2 -2 -1 -1 -1  1  1
 1  1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1  1  1
 1  1  1  1  1 -1 -1 -1 -1 -1  1  1  1  1  1
```

kr(*,*,2)

```
 1  2  2  2  2
 1  2  2  2  2
 1  2  2  2  2
 1  2  2  2  2
 1  2  2  2  2
 1  2  2  2  2
 1  2  2  2  2
 1  2  2  2  2
 1  2  2  2  2
 1  2  2  2  2
 1  2  2  2  2
 1  2  2  2  2
 1  2  2  2  2
 1  2  2  2  2
 1  2  2  2  2
```

e

kr(*,*,1)

```
 1  1  1  1  1 -1 -1 -1 -1 -1  1  1  1  1  1
 1  1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1  1  1
 1  1 -1 -1 -1 -2 -2 -2 -2 -2 -1 -1 -1  1  1
 1 -1 -1 -1 -2  2  0  0  0  2 -2 -1 -1 -1  1
 1 -1 -1 -2  2  0  0  0  0  0  2 -2 -1 -1  1
 1 -1 -2 -2  0  0  0  0  0  0  0 -2 -2 -1  1
-1 -1 -2  0  0  0  0  0  0  0  0  0 -2 -1 -1
-1 -1 -2  0  0  0  0  0  0  0  0  0 -2 -1 -1
-1 -1 -2  0  0  0  0  0  0  0  0  0 -2 -1 -1
 1 -1 -2 -2  0  0  0  0  0  0  0 -2 -2 -1  1
 1 -1 -1 -2  2  0  0  0  0  0  2 -2 -1 -1  1
 1 -1 -1 -1 -2  2  0  0  0  2 -2 -1 -1 -1  1
 1  1 -1 -1 -1 -2 -2 -2 -2 -2 -1 -1 -1  1  1
 1  1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1  1  1
 1  1  1  1  1 -1 -1 -1 -1 -1  1  1  1  1  1
```

kr(*,*,2)

```
-1 -2 -2 -2  2
-1 -2 -2 -2  2
-1 -2 -2 -2  2
-1 -2 -2 -2  2
-1 -2 -2 -2  2
-1 -2 -2 -2  2
-1 -2 -2 -2  2
-1 -2 -2 -2  2
-1 -2 -2 -2  2
-1 -2 -2 -2  2
-1 -2 -2 -2  2
-1 -2 -2 -2  2
-1 -2 -2 -2  2
-1 -2 -2 -2  2
-1 -2 -2 -2  2
```

f

kr(*,*,1)

```
 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
 1  1  1  1  1 -2 -2 -2 -2 -2  1  1  1  1  1
 1  1  1  1 -2 -2  0  0  0 -2 -2  1  1  1  1
 1  1  1 -2 -2  0  0  0  0  0 -2 -2  1  1  1
 1  1 -2 -2  0  0  0  0  0  0  0 -2 -2  1  1
 1  1 -2  0  0  0  0  0  0  0  0  0 -2  1  1
 1  1 -2  0  0  0  0  0  0  0  0  0 -2  1  1
 1  1 -2  0  0  0  0  0  0  0  0  0 -2  1  1
 1  1 -2 -2  0  0  0  0  0  0  0 -2 -2  1  1
 1  1  1 -2 -2  0  0  0  0  0 -2 -2  1  1  1
 1  1  1  1 -2 -2  0  0  0 -2 -2  1  1  1  1
 1  1  1  1  1 -2 -2 -2 -2 -2  1  1  1  1  1
 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
```

kr(*,*,2)

```
-1  2  2  2  2
-1  2  2  2  2
-1  2  2  2  2
-1  2  2  2  2
-1  2  2  2  2
-1  2  2  2  2
-1  2  2  2  2
-1  2  2  2  2
-1  2  2  2  2
-1  2  2  2  2
-1  2  2  2  2
-1  2  2  2  2
-1  2  2  2  2
-1  2  2  2  2
-1  2  2  2  2
```

FIGURE 11—*Continued*

4. Composite Grid Output and Data Structures—the DSK Routines

All the information needed to solve PDEs on composite grids is output by CMPGRD. This information can be written to a data file or CMPGRD can be called as a subroutine and the information returned in an array. The information CMPGRD generates includes not only the grid point locations $(x(i, j, k), y(i, j, k))$ but also the derivatives of the mapping functions, a list of interpolation points and their locations, the interpolation and discretization widths, and so on. In order to more easily keep track of this information and to aid in the writing of application programs we have designed the *composite grid data structure*. We use standard ideas from computer science data structure design. To manage this data structure within the confines of Fortran and to efficiently store the large arrays we have developed some utility routines. The utility routines simply implement some of the features which appear naturally in such languages as *Pascal* or *C*. These data management routines are known as the DSK package and described in "The DSK Package, a Data Structure for Efficient Fortran Array Storage (Reference Guide for the DSK Package)" [14] and in "Getting Started with CMPGRD, Introductory User's Guide and Reference Manual" [11].

Logically the composite grid data is organized into a directory-file structure. The information for a particular component grid, for example, can be found in a particular directory, under a particular name. Figure 12 shows part of the composite grid data structure. This figure schematically shows how the data associated with a composite grid is organized. For example, the variable **ng** found in the directory **composite grid** indicates the number of component grids found in

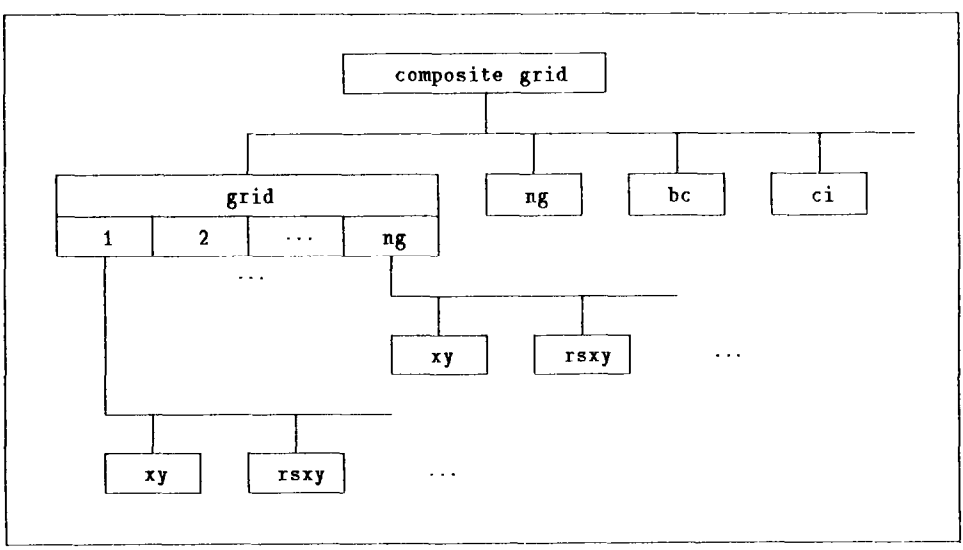


FIG. 12. Composite grid data structure.

this particular composite grid. The directory **grid** is actually an array of directories, which we denote by **grid[k]**, **k** = **1**, **2**, ..., **ng**. The directory **grid[k]** would hold the variables associated with the **k**th component grid such as the coordinate locations **xy** or the jacobian derivatives **rsxy**.

DSK utility routines permit one to locate items in this directory-file structure, add or delete items, and write out items in a convenient way. By *item* we refer to variables or arrays of type integer, real or double precision, as well as directories and pointer variables. Physically this data structure all exists on a single large array. To give a flavour of what is involved in locating an array in this data structure we present a piece of Fortran code which locates an array **xy** in component grid $\mathbf{G}_1$ and then calls a subroutine where the **xy** array is used in a standard fashion. One should assume that the DSK initialization routine has already been called and that the composite grid data structure has been created. The main point of this example is to demonstrate the use of the function **dskloc** which will return a pointer to a variable of a given name.

```
      subroutine sub1( ndra,ndrb,ndsa,ndsb,cgdir,disk )
c=============================================================
c       Find the array grid[1]/xy(ndra:ndrb,ndsa:ndsb,2)
c    Assume we know :
c      cgdir : "composite grid" directory
c      ndra,ndrb,ndsa,ndsb : array dimensions for grid[1]
c=============================================================

      integer dskloc,cgdir,gdir,pxy
      real disk(*)
           ...
c...get gdir = pointer to 'grid' directory :
      gdir=dskloc( disk,cgdir,'grid' )
c...get pxy = pointer to 'xy' in grid[1] :
      pxy =dskloc( disk,gdir,'xy' )
      call sub2( disk(pxy),ndra,ndrb,ndsa,ndsb )
             ...
      end

      subroutine sub2( xy,ndra,ndrb,ndsa,ndsb )
c...In this subroutine we treat xy as a normal array
      real xy(ndra:ndrb,ndsa:ndsb,2)
      ...
      write(6,*) 'x(i,j) =',xy(i,j,1)
      write(6,*) 'y(i,j) =',xy(i,j,2)
      ...
      return
      end
```

Of course, one needs a certain amount of experience with these routines to become compatible with them. Having all the composite grid data on a single array makes it easy to write application routines for general composite grids. The application program need only be given the large array and a pointer to the main directory and it can find any information it needs. In the next section we describe some of the programs which have been written for solving PDEs on general composite meshes, including a solver for systems of second-order elliptic equations.

5. Numerical Solution of PDEs on Composite Meshes

The principal use of composite meshes is in discretizing PDE boundary value problems. In this section we describe some methods for solving elliptic and time dependent problems. The solution of a PDE on a composite mesh can be thought of as the solution of a set of transformed PDEs on a set of unit squares. Standard discretization techniques are applied on each unit square while a system of interpolation equations couples the solutions on the unit squares. In Section 5.1 we will describe one particular way of discretization using finite differences and the *mapping method.* The *mapping method* simply consists of transforming partial derivatives in (x, y) space to partial derivatives in the (r, s) spaces using the derivatives of the transformations $\mathbf{d}_k$ which map the unit square onto the component domains $\mathbf{D}_k$. Other discretization techniques are possible such as finite volume methods [26] or finite element methods. For the purpose of obtaining accurate solutions of PDEs the individual component grids of a composite mesh should be sufficiently smooth. The smoothness of the component grids is reflected in the smoothness of the transformations $\mathbf{d}_k$. The mapping method of discretization probably requires more smoothness in the transformations than, say, a finite volume approach. However, with composite grids there is little difficulty in obtaining the required smoothness.

The grid function values on different component meshes are matched together through interpolation equations. In Sections 5.2 and 5.3 we describe a convenient way to perform this interpolation and we make some remarks regarding the order of interpolation that should be used. In Section 5.4 we describe some useful techniques for creating boundary-fitted component grids. We describe the solution of elliptic PDEs in Section 5.5 and the solution of time dependent equations in Section 5.6.

5.1. *Mapping Method of Discretization*

Consider a PDE boundary value problem

$$\mathbf{F}\left(\frac{\partial}{\partial t}, \frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \mathbf{u}\right) = 0 \qquad \text{for} \quad (x, y) \in \mathbf{D}, \tag{5.1}$$

where $\mathbf{F}$ represents the PDE in the interior of $\mathbf{D}$ and the boundary conditions on the boundary of $\mathbf{D}$, and it is understood that higher derivatives may be involved.

For each component grid $\mathbf{G}_k$ with function $\mathbf{d}_k$ which maps the unit square (r, s) into the subdomain $\mathbf{D}_k$ of $\mathbf{D}$ we can transform the PDE (5.1) into the coordinates of the unit square

$$\mathbf{F}\left(\frac{\partial}{\partial t}, \frac{\partial r}{\partial x}\frac{\partial}{\partial r}+\frac{\partial s}{\partial x}\frac{\partial}{\partial s}, \frac{\partial r}{\partial y}\frac{\partial}{\partial r}+\frac{\partial s}{\partial y}\frac{\partial}{\partial s}, \mathbf{u}_k\right)=0 \qquad \text{for} \quad (r, s)\in[0, 1]\times[0, 1], \tag{5.2}$$

where $\mathbf{u}_k(r, s)=\mathbf{u}(\mathbf{d}_k(r, s))$. The derivatives of the transformation at the grid points are supplied by CMPGRD,

$$\mathbf{rx}(i, j)=\frac{\partial r}{\partial x}(\mathbf{x}_{ij}), \qquad \mathbf{sx}(i, j)=\frac{\partial s}{\partial x}(\mathbf{x}_{ij}), \qquad \mathbf{ry}(i, j)=\frac{\partial r}{\partial y}(\mathbf{x}_{ij}), \qquad \mathbf{sy}(i, j)=\frac{\partial s}{\partial y}(\mathbf{x}_{ij}).$$

These equations (5.2) can now be easily discretized with standard methods for rectangular grids. For example, the following second-order centered difference approximations may be used for many problems. For first derivatives we use the approximations

$$\frac{\partial u}{\partial x}=\frac{\partial r}{\partial x}\frac{\partial u}{\partial r}+\frac{\partial s}{\partial x}\frac{\partial u}{\partial s}$$

$$\frac{\partial u}{\partial r}\approx\frac{u_{i+1j}-u_{i-1j}}{2h_r},$$

while for second derivatives we have

$$\frac{\partial^2 u}{\partial x^2}=\left(\frac{\partial r}{\partial x}\right)^2\frac{\partial^2 u}{\partial r^2}+2\frac{\partial r}{\partial x}\frac{\partial s}{\partial x}\frac{\partial^2 u}{\partial r\,\partial s}+\left(\frac{\partial s}{\partial x}\right)^2\frac{\partial^2 u}{\partial s^2}+\frac{\partial^2 r}{\partial x^2}\frac{\partial u}{\partial r}+\frac{\partial^2 s}{\partial x^2}\frac{\partial u}{\partial s}$$

$$\frac{\partial^2 u}{\partial r^2}\approx\frac{u_{i+1j}-2u_{ij}+u_{i-1j}}{h_r^2}$$

$$\frac{\partial^2 r}{\partial x^2}=\frac{\partial r}{\partial x}\frac{\partial}{\partial r}\left(\frac{\partial r}{\partial x}\right)+\frac{\partial s}{\partial x}\frac{\partial}{\partial s}\left(\frac{\partial r}{\partial x}\right)$$

$$\frac{\partial}{\partial r}\left(\frac{\partial r}{\partial x}\right)\approx\frac{\mathbf{rx}_{i+1j}-\mathbf{rx}_{i-1j}}{2h_r}.$$

Utility routines are available to generate the discrete coefficients corresponding to the above formulae.

5.2. *Interpolation—CGINT and CGINTE*

Grid function values at points on interior boundaries of component grids are obtained by interpolation from other component grids. CMPGRD provides all the information needed to make this interpolation step particularly easy. Consider the situation depicted in Fig. 13 in which the point $\mathbf{x}_{ijk}=(x_{ijk}, y_{ijk})$ is to be interpolated from component grid k'. CMPGRD supplies the (r', s', k') coordinates of the point

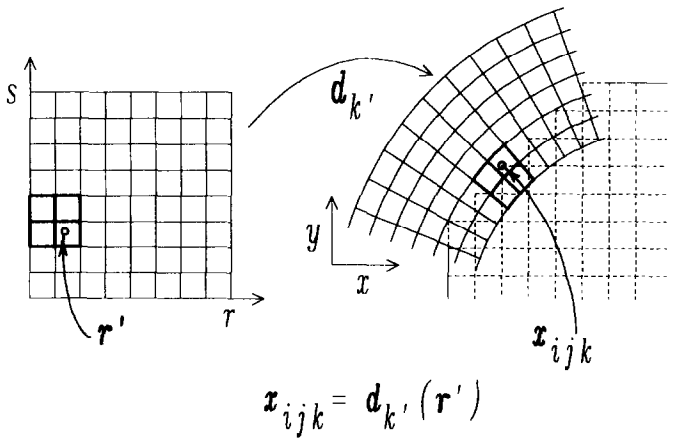


FIG. 13. Interpolation is performed in (r, s) coordinates.

(i, j, k): $\mathbf{r}' = (r', s') = \mathbf{d}_{k'}^{-1}(\mathbf{x}_{ijk})$. Therefore it is only necessary to know how to interpolate a point from a rectangular grid. Moreover, CMPGRD also supplies the information indicating the set of points (i', j', k') from which the interpolation should be performed. In particular, an interpolation formula would be of the form

$$\mathbf{u}_k(i, j) \approx \sum_{i'=i'_0}^{i'_1} \sum_{j'=j'_0}^{j'_1} \alpha(i', j', i, j)\mathbf{u}_{k'}(i', j'). \tag{5.3}$$

CMPGRD supplies the lower left corner of the interpolation stencil, (i'_0, j'_0), and the stencil widths (iw_r, iw_s), from which (i'_1, j'_1) can be calculated. The interpolation weights $\alpha(i', j', i, j)$ can be computed from (r', s') using, for example, bi-linear, bi-quadratic or, in general, two-dimensional Lagrange interpolation.

The above approach to interpolation not only permits an easy way of implementing arbitrary-order interpolation, but also makes the interpolation step less prone to error.

In general the system of interpolation equations will couple interpolation points on different component grids. That is, some of the points appearing on the right-hand side of the interpolation equation (5.3) will themselves be interpolation points. This coupling can be avoided when creating the composite grid by specifying *explicit* interpolation, in which case there will be more overlap between the grids.

When the equations are coupled a small system of equations must be solved to obtain the solution at the interpolation points in terms of the values at other points. The routine CGINT has been written to solve these equations using the Yale sparse matrix package [17]. The equations need only be factored once, implying that only a back-substitution is required each time the interpolation points are to be updated. The advantage of this *implicit* interpolation is that the amount of overlap is less and thus there are fewer grid points.

The routine CGINTE, on the other hand, solves the interpolation equations when they are explicit. This routine is more efficient than the implicit version and

can be vectorized more easily. Timing results in the Section 6.5 compare the speed of these two methods of interpolation.

5.3. *Accuracy and Order of Interpolation*

An important question to ask when using composite grids is how to choose the order of interpolation so that the overall accuracy will be as good as the accuracy of the discretization formulae. The answer to this question depends on the order of the PDE and the order of accuracy of the discretization formula. Moreover, the answer also depends on the behaviour of the region of overlap as the mesh is refined. Typically the overlap region will have a width which is approximately a constant times h, where h is a measure of the grid spacing. That is, the overlap region shrinks as the mesh is refined. *Note*: we will call an interpolant whose accuracy is $O(h^p)$ to be a pth-order interpolant. In one dimension the standard interpolant on an equally spaced mesh which uses p points is a pth-order interpolant. Thus the standard linear interpolation (two points) is second-order interpolation while quadratic interpolation (three points) is third-order interpolation.

In Henshaw [21] it was shown that for solving second-order elliptic equations to second-order accuracy it is necessary to use third-order interpolation (quadratic interpolation) if the overlap between component grids decreases with h as the grids are refined. (Second-order interpolation (linear interpolation) is sufficient if the overlap remains larger than some constant.) In this section we determine how to choose the interpolation for a more general class of problems. We consider the model problem of solving a $(2p)$th-order boundary value problem on a one-dimensional composite grid. We show that when the overlap d decreases linearly with h, $d \propto h$, then the width of the interpolation formula, q, should be $2pr+1$, where $2r$ is the order of acuracy of the discretization. Thus the width of the interpolation formula is the same as the width of the discretization formula. If, on the other hand, d is a constant independent of h then $q = pr + 1$.

To summarize the results of this section, if

$2p$: order of PDE (highest spatial derivative)

$2r$: order of accuracy of discretization

q: width of interpolation formula

d: width of overlap,

then

$q = 2pr + 1$: width of interpolation formula if $d \propto h$

$q = pr + 1$: width of interpolation formula if $d = O(1)$

pr: number interpolation points on each grid

$d + pr(h_1 + h_2)$: total overlap.

In order to see why these results hold we consider a model elliptic problem on a one-dimensional composite mesh. The model problem is the following $(2p)$th-order boundary value problem

$$\begin{aligned} \frac{\partial^{2p} u}{\partial x^{2p}} &= F(x) \qquad x \in [a, b] \\ \frac{\partial^j u}{\partial x^j}(a) &= L_j \qquad j = 0, \ldots, p-1 \\ \frac{\partial^j u}{\partial x^j}(b) &= R_j \qquad j = 0, \ldots, p-1. \end{aligned} \tag{5.4}$$

We attempt to solve this problem on a one-dimensional composite mesh which consists of the two overlapping grids, Fig. (14),

$$G^1 = \{x_i^1 \mid x_i^1 = a + (i-1)h_1,\ i = 1, \ldots, N_1\}$$
$$G^2 = \{x_i^2 \mid x_i^2 = b - (i-1)h_2,\ i = 1, \ldots, N_2\}.$$

The points of interpolation are

$$\begin{aligned} x_j &= x^1_{N_1 - j + 1} \qquad j = 1, \ldots, p \\ x_{p+j} &= x_j^2 \qquad j = 1, \ldots, p. \end{aligned}$$

We define the *overlap* d as the distance between the innermost interpolation point on G^1 to the innermost interpolation point on G^2, $d = x_p - x_{2p}$. Note that in general one wants enough overlap so that $d > 0$. This means that all interpolation points on G^1 lie to the right of all interpolation points of G^2. This amount of overlap is required since the system of interpolation equations becomes singular whenever the position of an interpolation point on G^1 coincides with the position of an interpolation point on G^2.

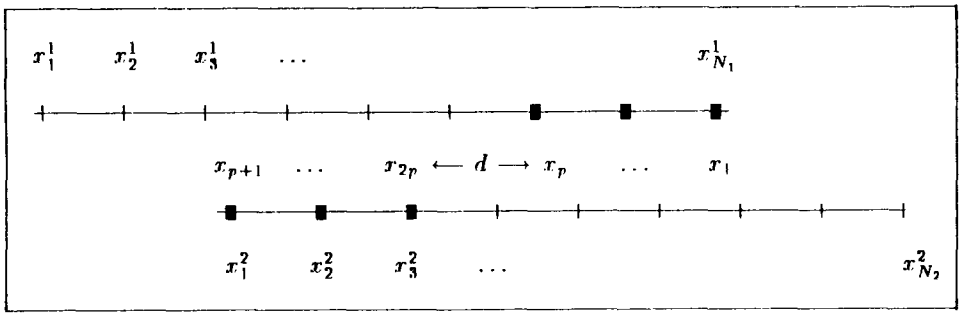


FIG. 14. One-D composite mesh showing grid points x_i^k, interpolation points x_i, and overlap d.

We discretize the equations, attempting to achieve second-order accuracy. We use q-point interpolation. The discrete system of equations we obtain is

$$\begin{aligned}
(D_+ D_-)^p v_i^k &= f_i && \text{for interior points} \\
D_L^j v_j^1 &= L_j && j = 0, \ldots, p-1 \\
D_R^j v_{N_2-j}^2 &= R_j && j = 0, \ldots, p-1 \\
v_i^k &= \sum_j \alpha_{i,j}^k v_j^{k'} && \text{for interpolation points.}
\end{aligned}$$

Here

$$(D_+ D_-) v_i^k = \frac{v_{i+1}^k - 2v_i^k + v_{i-1}^k}{h_k^2},$$

and D_L^j, D_R^j are some appropriate discrete approximations to the boundary conditions in (5.4). The difference between the true solution and the approximate solution will be denoted by

$$e_i^k := u(x_i^k) - v_i^k.$$

This error function satisfies

$$\begin{aligned}
(D_+ D_-)^p e_i^k &= O(h_k^2) && \text{for interior points} \\
D_L^j e_j^1 &= O(h_1^2) && j = 0, \ldots, p-1 \\
D_R^j e_{N_2-j}^2 &= O(h_2^2) && j = 0, \ldots, p-1 \\
e_i^k &= \sum_j \alpha_{i,j}^k e_j^{k'} + O(h_{k'}^q) && \text{for interpolation points.}
\end{aligned}$$

The truncation errors appear as forcing functions in the above equations. We can split the above equations into two new problems, one in which the truncation error in the interpolation equations is zero and one in which the truncations errors in the interior and boundary discretization formulae are zero. The total error will be the sum of the solutions of these two problems. We assume that the first of these cases (zero truncation in the interpolation equations) is well posed and causes no trouble. This was proved in [21] for the case $p = 1$. We thus consider the problem

$$\begin{aligned}
(D_+ D_-)^p e_i^k &= 0 && \text{for interior points} \\
D_L^j e_j^1 &= 0 && j = 0, \ldots, p-1 \\
D_R^j e_{N_2-j}^2 &= 0 && j = 0, \ldots, p-1 \\
e_i^k &= \sum_j \alpha_{i,j}^k e_j^{k'} + O(h_{k'}^q) && \text{for interpolation points.}
\end{aligned}$$

To simplify the discussion further, we consider the following continuous problem which is related to these discrete equations:

$$\begin{aligned}
&\frac{\partial^{2p} e^k}{\partial x^{2p}}(x) = 0 && x \in [a, b] \\
&\frac{\partial^j e^1}{\partial x^j}(a) = 0 && j = 0, \ldots, p-1 \\
&\frac{\partial^j e^2}{\partial x^j}(b) = 0 && j = 0, \ldots, p-1 \\
&e^k(x_i) = \sum_j \alpha^k_{i,j} e^{k'}(x_j) + O(h^q_{k'}) && \text{for interpolation points.}
\end{aligned} \tag{5.5}$$

We will now obtain bounds for the size of e^1 and e^2 as a function of q, h_k, and d. Both e^1 and e^2 are polynomials of degree $2p-1$ of the form

$$\begin{aligned}
e^1(x) &= c^1_1(x-a)^p + c^1_2(x-a)^{p+1} + \cdots + c^1_p(x-a)^{2p-1} \\
e^2(x) &= c^2_1(b-x)^p + c^2_2(b-x)^{p+1} + \cdots + c^2_p(b-x)^{2p-1}.
\end{aligned} \tag{5.6}$$

The interpolation equations will determine the constants, c^k_i, in this last expression. Substituting (5.6) into the interpolation equations gives

$$\begin{aligned}
e^1(x_i) &= \sum_{j=1}^{p} \alpha^1_{i,j} e^2(x_j) + O(h^q_2) \\
&= e^2(x_i) + O(h^q_2) + O(h^q_2) \\
&= \sum_{j=1}^{p} c^2_j (b - x_i)^{p+j-1} + O(h^q_2), \qquad i = 1, \ldots, p.
\end{aligned}$$

Whence

$$\sum_{j=1}^{p} c^1_j (x_i - a)^{p+j-1} = \sum_{j=1}^{p} c^2_j (b - x_i)^{p+j-1} + O(h^q_2), \qquad i = 1, \ldots, p$$

and, similarly,

$$\sum_{j=1}^{p} c^2_j (b - x_i)^{p+j-1} = \sum_{j=1}^{p} c^1_j (x_i - a)^{p+j-1} + O(h^q_1), \qquad i = p+1, \ldots, 2p.$$

These equations define a system of linear equations for the coefficients c^k_i which is of the form

$$A\mathbf{c} = O(h^q),$$

where $h = \max\{h_1, h_2\}$,

$$\mathbf{c} = [c_1^1 \quad c_2^1 \quad \cdots \quad c_p^1 \quad -c_1^2 \quad -c_2^2 \quad \cdots \quad -c_p^2]^{\mathrm{T}}$$

and

$$A = \begin{bmatrix} (x_1-a)^p & \cdots & (x_1-a)^{2p-1} & (b-x_1)^p & \cdots & (b-x_1)^{2p-1} \\ (x_2-a)^p & \cdots & (x_2-a)^{2p-1} & (b-x_2)^p & \cdots & (b-x_2)^{2p-1} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ (x_{2p}-a)^p & \cdots & (x_{2p}-a)^{2p-1} & (b-x_{2p})^p & \cdots & (b-x_{2p})^{2p-1} \end{bmatrix}.$$

We wish to determine the asymptotic behaviour for $\mathbf{c}$ as h_k tends to zero since for second-order accuracy $\mathbf{c}$ must be $O(h^2)$. Now consider solving $A\mathbf{c} = O(h^q)$ by Cramer's rule. The matrix A is similar in form to the Vandermonde matrix and we show that

$$\det(A) = C(a, b, p) \prod_{i=1}^{2p} \left\{ \prod_{\substack{j=1 \\ j \neq i}}^{2p} (x_i - x_j) \right\}, \tag{5.7}$$

where $C(a, b, p)$ is independent of x_j. This result can be shown as follows. As a function of x_j, $\det(A)$ is a polynomial of degree $2p-1$. Since $\det(A) = 0$ when $x_j = x_k$ for $k = 1, ..., j-1, j+1, ..., 2p$, it follows that $\det(A)$ must be of the form

$$\det(A) = D(x_1, ..., x_{j-1}, x_{j+1}, ..., x_{2p}, a, b, p) \prod_{\substack{k=1 \\ k \neq j}}^{2p} (x_j - x_k).$$

where D is independent of x_j. The result (5.7) follows.

The determinants of the principal submatrices of A are of the same form as the determinant of A and hence by Cramer's rule

$$c_m^k \sim \sum_{n=1}^{2p} O(h^q) \left\{ \frac{\prod_{i=1, i\neq n}^{2p} \prod_{j=1, j\neq i\neq n}^{2p} (x_i - x_j)}{\prod_{i=1}^{2p} \prod_{j=1, j\neq i}^{2p} (x_i - x_j)} \right\}. \tag{5.8}$$

We are thus led to the following conclusions:

LEMMA I. *If the overlap d is greater than or equal to a constant times h, $d \geqslant ch$, then the solution to the error equations* (5.5) *is second-order accurate provided the width of the interpolation formula, q, satisfies*

$$q \geqslant 2p + 1.$$

LEMMA II. *If the overlap d is greater than or equal to a constant independent of h, $d \geqslant c$, then the solution to the error equations* (5.5) *is second-order accurate provided the width of the interpolation formula, q, satisfies*

$$q \geqslant p + 1.$$

Proof. To estimate the right-hand side of (5.8) we need

$$|x_i - x_j| = \begin{cases} |i-j|h_1 & 1 \leqslant i, j \leqslant p \\ |i-j|h_2 & p+1 \leqslant i, j \leqslant 2p \\ d + (p-i)h_1 + (j-p-1)h_2 & 1 \leqslant i \leqslant p \text{ and } p+1 \leqslant j \leqslant 2p. \end{cases}$$

If $d \geqslant ch$ then each factor of $(x_i - x_j)$ in (5.8) is $O(h)$ and there are $\binom{2p}{2}$ terms in the product appearing in the numerator and $\binom{2p}{2}$ terms in the product of the denominator. Since $\binom{2p}{2} - \binom{2p-1}{2} = 2p - 1$, it follows that

$$c_m^k \sim \sum_{n=1}^{2p} \frac{O(h^q)}{O(h^{2p-1})} \sim O(h^{q-2p+1})$$

and thus for second-order accuracy we require $q \geqslant 2p + 1$. On the other hand, if d is larger than some constant then there are only $2\binom{p}{2}$ terms of order h in the denominator and $\binom{p}{2} + \binom{p-1}{2}$ terms of order h in the numerator and, since $2\binom{p}{2} - \{\binom{p}{2} + \binom{p-1}{2}\} = p - 1$, it follows that we require $q \geqslant p + 1$. This completes the proof of the lemmas.

Remark 1. The model problem we have considered consisted of solving a $(2p)$th-order boundary value to 2nd-order accuracy. The results can be extended to higher order accuracy since, for example, solving a 4th-order equation to 2nd-order is equivalent to solving a 2nd-order equation to 4th-order.

Remark 2. We can relate our analysis to the case when the composite grid equations degenerate to the standard central-difference approximation for a single grid. This situation occurs when $h_1 = h_2 = d$, in which case the interpolation points align exactly with grid points. Our results say that interpolant must be $(2p + 1)$-order accurate. However, in this case even a 1-point interpolation formula will be exact (and thus is $(2p + 1)$-order accurate) and satisfies the conditions of our result.

Remark 3. We contrast our conclusions to the theoretical results of Kreiss [24] or Gustafsson [18] which prove that sometimes it is sufficient to use a boundary condition of lower order and still achieve the required global accuracy. However, in the above work, it is only those non-essential boundary conditions, such as conditions for out-going characteristic variables or extra boundary conditions which must be added for higher order discretizations, that can be specified to lower order. Essential boundary conditions, such as conditions for in-going characteristic variables, must be specified to full accuracy. Our results show that the interpolation equations cannot be thought of as non-essential boundary conditions.

We confirm these theoretical results by numerically solving the discrete system and computing the convergence rate. We solve

$$\begin{aligned}
(D_+ D_-)^p v_i^k &= f_i && \text{for interior points} \\
v_j^1 &= g_j^L && j = 0, \ldots, p-1 \\
v_j^2 &= g_j^R && j = 0, \ldots, p-1 \\
v_i^k &= \sum_j \alpha_{i,j}^k v_j^{k'} && \text{for interpolation points}
\end{aligned}$$

on a one-dimensional composite grid, Fig. 14, on the interval $[0, 1]$ for $p = 1$ and $p = 2$. We choose the right-hand sides f_i, g_j^L, and g_j^R so that the true solution is

$$u_{\text{TRUE}} = \sin(\pi x) + \cos(\pi x) + \sin(2\pi x) + \cos(2\pi x).$$

We consider the cases of the overlap d being a constant times h, $d = 0.5h$, and the overlap being a constant independent of h, $d = 0.5$. We solve the problem with the number of grid points on each component grid being $(N_1, N_2) = (10, 10)$, $(15, 15)$, ..., $(130, 130)$, and make a least squares fit to the convergence rate σ assuming that the error is proportional to h^σ,

$$e \propto h^\sigma.$$

When $d = 0.5h$ we use interpolation with $q = 2p$ and $q = 2p + 1$. The analytical results predict that q must be at least $2p + 1$ for second-order accuracy ($\sigma = 2$). When $d = 0.5$ we use interpolation with $q = p$ and $q = p + 1$, with the analytical results predicting that q must be at least $p + 1$ for second-order accuracy. The results given in Table I and II confirm our analysis.

In Section (5.5) we give further results from solving an elliptic problem on two-dimensional composite grids generated by CMPGRD. These two-dimensional results also confirm our theoretical predictions.

5.4. *On Component Grid Construction for Polygons with Smoothed-out Corners*

In this section we describe some special techniques for the generation of component grids. The method we describe is particularly suited to the creation of boundary fitted grids along curves which consist of straight lines connected by

TABLE I

Computed Order of Accuracy, $e \propto h^\sigma$, for $d = 0.5$

$d = 0.5$	$p = 1$	$p = 2$
$q = 2p$	$\sigma = 1.0$	$\sigma = 0.79$
$q = 2p + 1$	$\sigma = 2.1$	$\sigma = 2.0$

TABLE II

Computed Order of accuracy, $e \propto h^{\sigma}$, for $d=0.5$

$d=0.5$	$p=1$	$p=2$
$q=p$	$\sigma=0.38$	$\sigma=1.3$
$q=p+1$	$\sigma=1.9$	$\sigma=2.1$

corners. The grids we create will actually smooth out the corners; the degree to which the corners are rounded can be varied. The corners are rounded since our applications are usually to fluid flows and we do not want to address the problem of boundary conditions at singular points such as corners.

A component mesh $\mathbf{G}_k$ is defined by a mapping $\mathbf{d}_k$ from the unit square (r, s) to the physical domain (x, y). The first step in constructing the grid is to define a curve $\mathbf{x}_1(s)=(x_1(s), y_1(s))$ which approximates the boundary curve. This curve is to approximate the polygon which passes through the points

$$\mathbf{x}_c(i)=(x_c(i), y_c(i)), \qquad i=1, ..., n_c \qquad \text{(corners of polygon)}.$$

The curve is parameterized by a pseudo-arclength s, $0 \leqslant s \leqslant 1$, with the value of s at corner i being given by

$$s(i)=\frac{\sum_{j=1}^{i-1} \|\mathbf{x}_c(j+1)-\mathbf{x}_c(j)\|}{\sum_{j=1}^{n_c-1} \|\mathbf{x}_c(j+1)-\mathbf{x}_c(j)\|}.$$

As a function of s the curves $x_1(s)$ and $y_1(s)$ should approximate the piecewise linear function which passes through the points $x_c(i)$ and $y_c(i)$, respectively.

Both $x_1(s)$ and $y_1(s)$ are defined using the same stretching functions that CMPGRD uses for clustering grid points, as described in Section 3.1. The curves are simply defined using a combination of the *ramp functions* $V_j(s)$:

$$x_1(s)=\left[s+\sum_{j=1}^{n_c} (V_j(s)-V_j(0)) \right] C_1+C_0.$$

By suitable choice of the constants d_j, e_j, f_j which appear in the definition of V_j the curve $x_1(s)$ can be made to pass exponentially close to the points $x_c(i)$, $i=1, ..., n_c$, while being almost linear in the regions between the points. The curve itself has the desirable property of being analytic in s.

Once the boundary curve has been defined it is necessary to define a boundary fitted grid. The approach we take is to define the grid by using lines in the direction normal to the curve. The normal at each point on $\mathbf{x}_1(s)=(x_1(s), y_1(s))$ is

$$\mathbf{n}(s)=(-\dot{y}_1(s), \dot{x}_1(s))/\sqrt{\dot{x}_1^2(s)+\dot{y}_1^2(s)}.$$

The grid can thus be defined as

$$\mathbf{x}(r, s) = \mathbf{x}_1(s) + rN(s)\mathbf{n}(s), \qquad 0 \leqslant r \leqslant 1, \quad 0 \leqslant s \leqslant 1,$$

where the scalar function $N(s)$ is used to define the width of the grid in the normal direction. Typically one will want to *stretch* the grid lines in both the r and s directions. Grid lines are usually stretched in the s direction to cluster points near regions of large curvature, typically near corners. Stretching in the r direction would permit a concentration of grid lines near the boundary. Thus we define two stretching transformations $t(r)$ and $u(s)$, t: $[0, 1] \to [0, 1]$, u: $[0, 1] \to [0, 1]$, again defined using the stretching functions supplied with CMPGRD. The final grid takes the form

$$\mathbf{x}(r, s) = \mathbf{x}_1(u(s)) + t(r)R(u(s))\mathbf{n}(u(s)), \qquad 0 \leqslant r \leqslant 1, \quad 0 \leqslant s \leqslant 1.$$

In Fig. 15 we show some grids which have been defined in the manner outlined above.

5.5. *Solving Elliptic PDEs—CGEL and CGMG*

For elliptic boundary value problems the composite mesh equations consist of a set of equations describing the discrete form of the PDE coupled to a set of *interpolation equations* which connect the solution between different component grids. This set of equations can be solved in many ways; both direct sparse solvers and iterative methods can be used.

For small enough meshes (or big enough computers) the equations can be solved in a direct way using a sparse Gaussian elimination routine, for example. In particular we have written the Fortran subroutine CGEL to solve systems of linear variable coefficient elliptic PDEs of the form

$$\mathbf{Cxx}(x, y)\frac{\partial^2 \mathbf{u}}{\partial x^2} + \mathbf{Cxy}(x, y)\frac{\partial^2 \mathbf{u}}{\partial x\, \partial y} + \mathbf{Cyy}(x, y)\frac{\partial^2 \mathbf{u}}{\partial y^2} + \mathbf{Cx}(x, y)\frac{\partial \mathbf{u}}{\partial x}$$

$$+ \mathbf{Cy}(x, y)\frac{\partial \mathbf{u}}{\partial y} + \mathbf{Cu}(x, y)\mathbf{u} - \mathbf{f}(x, y) = 0, \qquad \mathbf{x} \in \mathbf{D}$$

with boundary conditions expressed in (x, y) derivatives,

$$\mathbf{Bx}(x, y)\frac{\partial \mathbf{u}}{\partial x} + \mathbf{By}(x, y)\frac{\partial \mathbf{u}}{\partial y} + \mathbf{Bu}(x, y)\mathbf{u} - \mathbf{g}(x, y) = 0, \qquad \mathbf{x} \in \partial\mathbf{D},$$

or in tangential and normal derivatives,

$$\mathbf{Bt}(x, y)\frac{\partial \mathbf{u}}{\partial t} + \mathbf{Bn}(x, y)\frac{\partial \mathbf{u}}{\partial n} + \mathbf{Bu}(x, y)\mathbf{u} - \mathbf{g}(x, y) = 0, \qquad \mathbf{x} \in \partial\mathbf{D}.$$

Here $\mathbf{u}$ is a vector function with n_v components and **Cxx**, **Cxy**, ..., **Cu**, **Bx**, **By**, ..., **Bu** are $n_v \times n_v$ matrices. The routine CGEL will solve elliptic equations using second-

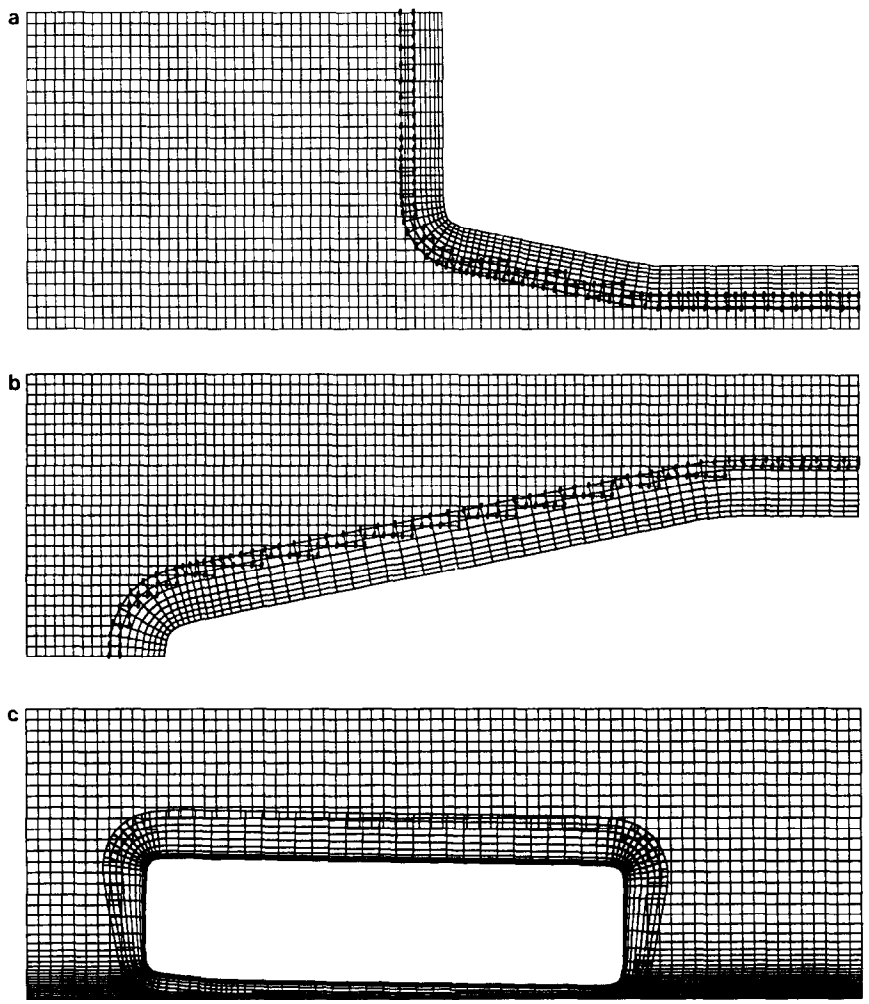


FIG. 15. Component grids generated with the stretching functions.

order or fourth-order difference methods on any CMPGRD composite mesh which has been constructed with the appropriate parameters for the widths of the discretization and interpolation formulae. Boundary conditions are discretized with one-sided derivatives. CGEL calls the Yale sparse matrix package [17] to solve the composite mesh equations.

As a simple example we solve the following Poisson equation:

$$\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}=f(x, y), \qquad (x, y)\in \mathbf{D}$$

$$u(x, y)=g(x, y), \qquad (x, y)\in \partial\mathbf{D}.$$

The forcing functions f and g are chosen so that the true solution is

$$u_{\text{true}} = \cos(2\pi x)\cos(2\pi y).$$

The equations are solved with second-order and fourth-order differences on a sequence of successively refined grids. Figure 16 shows the coarsest grids for the second-order and the fourth-order methods. Note that the fourth-order method uses two lines of interpolation. The finer grids (not shown) have 1.5, 2., and 2.5 times as many grid points in each direction as the coarsest grid. Denote the maximum errors on the four composite grids by e_m, $m = 1, ..., 4$. These errors are given in Tables III and IV along with an estimate of the convergence rate, σ. The convergence rate is determined by a least squares fit assuming that the error is proportional to h^σ.

Recall that the one-dimensional theory of Section 5.3 indicates that, loosely speaking, the width of the interpolation formula should be equal to (or greater than) the width of the discretization formula in order to achieve an overall accuracy equal to the order of discretization. This means that a 3×3 interpolation stencil (third-order or bi-quadratic interpolation) is required for second-order accuracy and a 5×5 interpolation stencil (fifth-order interpolation) is needed for a fourth-order accuracy. The tables show results for the theoretically suggested order of interpolation and for interpolation of one order less. In the latter case the convergence rate is seen to drop by one, in agreement with the theory.

The composite mesh equations generated from an elliptic PDE boundary value problem can also be solved using an iterative method such as the *multigrid method.* Multigrid uses a sequence of grids of varying coarseness to accelerate the convergence of the iteration. CMPGRD can automatically generate the sequence of grids required by the multigrid algorithm. Since CMPGRD is aware of the manner in which information is transferred between the different levels (the *prolongation* and *restriction* operators) the composite grids for multigrid can be optimally generated. We have described, in "Multigrid on Composite Meshes" [20], how the multigrid algorithm can be applied to composite meshes. We have written a program, called CGMG, to solve linear, variable coefficient elliptic boundary value problems. A wide variety of point and line smoothers are available; the user can choose different smoothers on different meshes. It was demonstrated in [20] that one can obtain convergence rates which are almost as good those obtained on a single grid. The grids of Fig. 17 were generated by CMPGRD for use with the multigrid algorithm. Note in particular that the finer grids are not just equal to the coarser grids with extra lines added.

5.6. *Time-Dependent PDEs*

A time-dependent PDE can be discretized using the *method of lines*: the partial differential equation is first discretized in space resulting in a system of time-

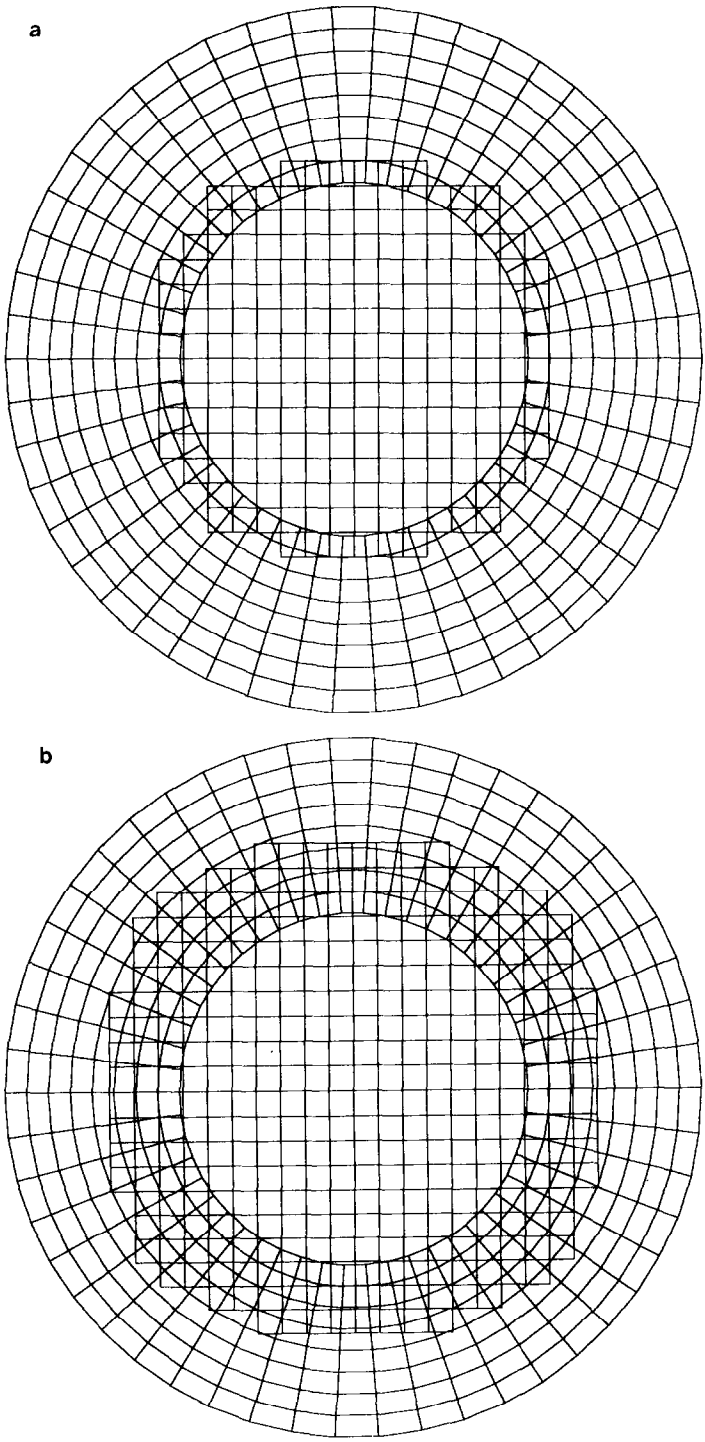


FIG. 16. Coarse grids for the accuracy tests of the elliptic solver: (a) for second-order; (b) for fourth-order.

TABLE III

Maximum Errors with Second-Order Discretization

iw	e_1	e_2	e_3	e_4	σ
2	0.11	0.12	0.053	0.053	0.98
3	0.048	0.021	0.010	0.0074	2.1

Note. $iw = 2$: second-order (bi-linear) interpolation, $iw = 3$: third-order (bi-quadratic) interpolation, σ: convergence rate, $e \propto h^{\sigma}$.

dependent ordinary differential equations. For composite grids this system will be of the form

$$\frac{d}{dt}\mathbf{u}(i, j, k; t) = \mathbf{f}(i, j, k; \mathbf{u}, t) \qquad \text{for interior points} \tag{5.9a}$$

$$B(\mathbf{u}(i, j, k; t), t) = 0 \qquad \text{for boundary points} \tag{5.9b}$$

$$\mathbf{u}(i, j, k; t) = \sum \alpha(i', j', i, j)\mathbf{u}(i', j', k; t) \qquad \text{for interpolation points.} \tag{5.9c}$$

Here $\mathbf{u}$ denotes the vector of all grid-point values and B denotes the discrete operator for the boundary conditions. With an explicit time stepping scheme the interior points can be advanced first from Eqs. (5.9a) and then the boundary and interpolation points can be updated from Eqs. (5.9b) and (5.9c). For implicit methods one must solve the implicit time stepping equations coupled with the interpolation equations. One way to solve these equations is to use the elliptic solver, CGEL, described in the previous section.

It is natural when dealing with composite meshes to consider using different time-stepping methods on different component meshes. This technique might be useful when some component meshes have much smaller mesh spacings than the other

TABLE IV

Maximum Errors with Fourth-Order Discretization

iw	e_1	e_2	e_3	e_4	σ
4	0.0053	0.0025	0.00023	0.00048	3.2
5	0.0037	0.00061	0.00018	0.000079	4.2

Note. $iw = 4$: fourth-order interpolation, $iw = 5$: fifth-order interpolation, σ: convergence rate, $e \propto h^{\sigma}$.

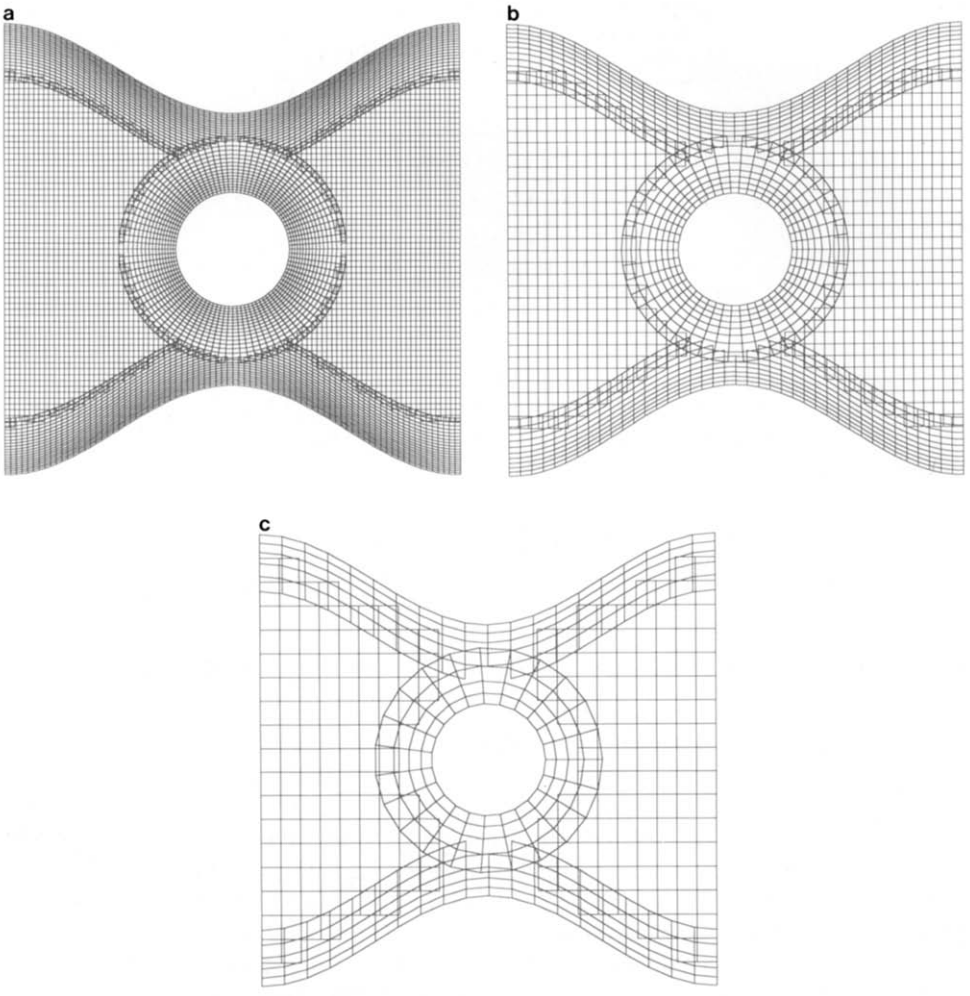


FIG. 17. CMPGRD can generate grids for the multigrid elliptic solver.

component meshes. In the numerical results presented in the final section we use an implicit–explicit scheme of the form

$$\frac{\mathbf{v}(n+1)-\mathbf{v}(n)}{\Delta t}=\frac{\mathbf{f}_I(n+1)+\mathbf{f}_I(n)}{2}+\frac{3}{2}\mathbf{f}_E(n)-\frac{1}{2}\mathbf{f}_E(n-1), \tag{5.10}$$

where $\mathbf{v}(n)$ denotes the solution at the nth time step and where the right-hand side of (5.9a) has been split into two pieces

$$\mathbf{f}=\mathbf{f}_E+\mathbf{f}_I$$

corresponding to the parts of the equation which are to be treated explicitly and the parts to be treated implicitly. This splitting can vary from component grid to component grid or even from grid point to grid point. Scheme (5.10) is a combination of the implicit trapezoidal rule (Crank–Nicolson) with a second-order Adams–Bashforth method and is overall second-order accurate in time. In general we can devise other higher-order schemes which treat part of an equation explicitly and part implicitly. Since such split schemes are not usually presented in standard texts we will briefly outline how to create them. We can devise a pth-order implicit–explicit multistep scheme of the form

$$\mathbf{v}(n+1) = \sum_{k=0}^{L} \alpha_k \mathbf{v}(n-k) + \Delta t \sum_{k=-1}^{M} \beta_k \mathbf{f}_I(n-k) + \Delta t \sum_{k=0}^{N} \gamma_k \mathbf{f}_E(n-k).$$

The method will be implicit if $\beta_{-1} \neq 0$. Although this is not the form of a standard multistep method, it can be easily derived from such schemes. For example, starting from the implicit scheme,

$$\mathbf{v}(n+1) = \sum_{k=0}^{L} \alpha_k \mathbf{v}(n-k) + \Delta t \sum_{k=-1}^{M} \beta_k \mathbf{f}(n-k),$$

one determines constants γ_k such that $\sum_{k=-1}^{M} \beta_k \mathbf{f}_E(n-k) = \sum_{k=0}^{N} \gamma_k \mathbf{f}_E(n-k) + O(\Delta t^p)$. This can always be done, for example, by extrapolating $\mathbf{f}(n+1)$:

$$\mathbf{f}(n+1) = \sum_{k=0}^{N} \delta_k \mathbf{f}(n-k) + O(\Delta t^p).$$

Of course one must consider the stability of the scheme that results. The scheme (5.10) is obtained in precisely this manner by starting from the trapezoidal rule and then extrapolating for $\mathbf{f}(n+1)$ to second order, $\mathbf{f}(n+1) = 2\mathbf{f}(n) - \mathbf{f}(n-1) + O(\Delta t^2)$, resulting in the second-order Adams–Bashforth scheme for the explicit part.

In our applications to the Navier–Stokes equations we are particularly interested in treating the second-order derivative terms in an implicit manner as these terms often determine the time-step restriction through stability considerations. However, we typically treat many more terms in an implicit fashion. The nonlinear terms are treated in a semi-implicit manner by first linearizing the equations. The approach will be to linearize with respect to functions which are independent of time. We only linearize with respect to the functions themselves as opposed to their derivatives. For example, consider the equation

$$\begin{aligned} u_t &= uu_x + vu_{xx} \\ &:= \mathbf{f}(t) \end{aligned}$$

which is the form of the Navier–Stokes equations. We split the right-hand side into $\mathbf{f} = \mathbf{f}_I + \mathbf{f}_E$, where

$$\mathbf{f}_I = U_0(x, y)u_x(x, y, t) + \nu u_{xx}$$
$$\mathbf{f}_E = (u(x, y, t) - U_0(x, y))u_x(x, y, t).$$

Typically we choose $U_0(x, y)$ to be the solution at some time t_0. The implicit time-stepping equations are solved using the elliptic solver CGEL described in an earlier section. We then integrate the equations over some time interval with this fixed linearization (and fixed factorization). After some time we update U_0 to be the current solution and refactor the matrix. One way to decide when to choose a new linearization is to wait until the relative difference between the current solution and the current linearization becomes larger than some small value. The time-stepping procedure is second-order no matter what linearization is used, although better stability conditions on the time step result if the linearization is close to the current solution.

6. Solution of the Compressible Navier–Stokes Equations on Composite Meshes

We now present some results from solving the compressible Navier–Stokes equations on composite meshes. After introducing the equations and boundary conditions we give some accuracy tests and finally show results from a number of applications. The main aim of this section is to show the flexibility of the method. Thus although we show results for flows around airfoils, we do not pretend to be experts in aerodynamics and hence we do not make comparisons of pressure profiles. We are, however, reasonably confident in the accuracy of the solutions we have computed for two main reasons: first of all, we have performed a number of nontrivial tests when a true solution is known; second, for simulations when no solution is available we try to ensure that computed grid functions are smooth with respect to the grids that we are using.

6.1 *Navier–Stokes and Scaling*

The compressible Navier–Stokes equations in two space dimensions can be written in the form:

$$\rho_t + (\rho u)_x + (\rho v)_y = 0$$
$$\rho(u_t + uu_x + vu_y) + p_x = \mu\,\Delta u + (\mu + \lambda)(u_x + v_y)_x$$
$$\rho(v_t + uv_x + vv_y) + p_y = \mu\,\Delta v + (\mu + \lambda)(u_x + v_y)_y$$
$$\rho(T_t + uT_x + vT_y) + \frac{p}{C_v}(u_x + v_y) = \frac{k}{C_v}\Delta T + \frac{\mu}{C_v}\Phi$$
$$\Phi = 2(u_x)^2 + 2(v_y)^2 + (v_x + u_y)^2 + \lambda(u_x + v_y)^2,$$

where ρ is the density, u and v are the horizontal and vertical components of the velocity, and p is the pressure. We assume the ideal gas law holds, $p = \rho RT$, and that $\lambda = -2/3\mu$. We nondimensionalize and scale the equations using a characteristic length L_0, velocity u_0, density ρ_0, absolute temperature T_0, and relative temperature variation of ΔT_0. Our nondimensional variables will be defined as

$$\tilde{x} = x/L_0, \qquad \tilde{y} = y/L_0, \qquad \tilde{t} = t(L_0/u_0), \qquad \tilde{\rho} = \rho/\rho_0, \qquad \tilde{u} = u/u_0, \qquad \tilde{v} = v/u_0$$
$$\tilde{T} = (T - T_0)/\Delta T_0, \qquad \tilde{p} = p/(\rho_0 R T_0) = \tilde{\rho}(1 + (\Delta T_0/T_0)\tilde{T}).$$

We introduce the non-dimensional parameters Re, Reynolds number, Pr, Prandtl number, Ma, Mach number, and γ the ratio of specific heats,

$$\mathrm{Re} = \frac{\rho_0 u_0 L_0}{\mu}, \qquad \mathrm{Pr} = \frac{\mu C_p}{k}, \qquad \mathrm{Ma} = \frac{u_0}{\sqrt{\gamma R T_0}}, \qquad \gamma = \frac{C_p}{C_v},$$

and after dropping the tildes on the non-dimensional variables the equations can be written as

$$\rho_t + \rho(u_x + v_y) + u\rho_x + v\rho_y = 0$$

$$u_t + uu_x + vu_y + \frac{1}{\rho}\frac{1}{\gamma \mathrm{Ma}^2}\left(1 + \frac{\Delta T_0}{T_0}T\right)\rho_x + \frac{1}{\gamma \mathrm{Ma}^2}\frac{\Delta T_0}{T_0}T_x = \frac{1}{\mathrm{Re}}\frac{1}{\rho}\left(\Delta u + \frac{1}{3}(u_x + v_y)_x\right)$$

$$v_t + uv_x + vv_y + \frac{1}{\rho}\frac{1}{\gamma \mathrm{Ma}^2}\left(1 + \frac{\Delta T_0}{T_0}T\right)\rho_y + \frac{1}{\gamma \mathrm{Ma}^2}\frac{\Delta T_0}{T_0}T_y = \frac{1}{\mathrm{Re}}\frac{1}{\rho}\left(\Delta v + \frac{1}{3}(u_x + v_y)_y\right)$$

$$T_t + uT_x + vT_y + (\gamma - 1)\left(\frac{T_0}{\Delta T_0} + T\right)(u_x + v_y) = \frac{\gamma}{\mathrm{Re}\,\mathrm{Pr}}\frac{1}{\rho}\Delta T + \frac{\gamma(\gamma - 1)\,\mathrm{Ma}^2}{\mathrm{Re}}\frac{T_0}{\Delta T_0}\frac{1}{\rho}\Phi$$

$$\Phi = 2(u_x)^2 + 2(v_y)^2 + (v_x + u_y)^2 - \frac{2}{3}(u_x + v_y)^2.$$

This will be the form of the equations which we discretize.

6.2. *Boundary Conditions*

We use boundary conditions of the form

$$\text{wall (no-slip):} \begin{cases} u = u_B \\ v = v_B \\ T + \alpha T_n = T_B \\ p\text{: extrapolated} \end{cases} \qquad \text{wall (slip):} \begin{cases} \mathbf{u}\cdot\mathbf{n} = 0 \\ (\mathbf{u}\cdot\mathbf{t})_n = 0 \\ T + \alpha T_n = T_B \\ p\text{: extrapolated} \end{cases}$$

$$\text{inflow (subsonic):} \begin{cases} \mathbf{u}\cdot\mathbf{t} = g_I^1 \\ \mathbf{u}\cdot\mathbf{n} + \rho = g_I^2 \\ (\mathbf{u}\cdot\mathbf{n} - \rho)_n = g_I^3 \\ T = g_I^4 \end{cases} \qquad \text{inflow (supersonic):} \begin{cases} \rho = g_I^1 \\ u = g_I^2 \\ v = g_I^3 \\ T = g_I^4 \end{cases}$$

$$\text{outflow (subsonic):} \begin{cases} (\mathbf{u}\cdot\mathbf{t})_n = g_O^1 \\ (\mathbf{u}\cdot\mathbf{n}+\rho)_n = g_O^2 \\ \mathbf{u}\cdot\mathbf{n}-\rho = g_O^3 \\ T_n = g_O^4 \end{cases} \qquad \text{outflow (supersonic):} \begin{cases} \rho_n = 0 \\ u_n = 0 \\ v_n = 0 \\ T_n = 0 \end{cases}$$

$$\text{symmetry:} \begin{cases} \mathbf{u}\cdot\mathbf{n} = 0 \\ (\mathbf{u}\cdot\mathbf{t})_n = 0 \\ \rho_n = 0 \\ T_n = 0. \end{cases}$$

Here $\mathbf{t}$ and $\mathbf{n}$ denote the tangent and normal vectors to the boundaries and u_n the normal derivative of u. Boundary conditions are discretized with second-order one-sided difference approximations.

6.3. *Remarks on Implementation*

With the aid of our data handling routines (the DSK routines described in Section 4), we have been able to write a Fortran code, CGNVT, to solve the Navier–Stokes equations on the general class of composite meshes which are generated through CMPGRD. All the information about the composite grid that the program needs to know is contained in the output file generated by the composite grid program. At the time the composite grid is generated with CMPGRD a boundary condition code is associated with each side of each component grid. These codes are translated by the Navier–Stokes code, CGNVT, into boundary conditions such as those presented in Section 6.2.

We discretize the compressible Navier–Stokes equations to second order in space using the mapping method described in Section 5.1. We integrate the equations in time using the second-order implicit–explicit method (5.10). The code permits one to choose which component grids should be integrated implicitly and which ones should be integrated explicitly. We use the routine CGEL (Section 5.5) to solve the system of equations that results from the implicit time-stepping, including the interpolation and boundary conditions.

To prevent numerical instabilities we add an artificial viscosity to the density equation. This artificial viscosity is applied in the transformed (r, s) coordinates and takes the form

$$\rho_t + \rho(u_x + v_y) + u\rho_x + v\rho_y = \nu_a(h_r^2\,\rho_{rr} + h_s^2\,\rho_{ss}).$$

Here h_r and h_s are the grid spacings on the unit square and ν_a is the coefficient of the artificial viscosity. This is a standard way to apply a second-order artificial viscosity and the typical claim is that the solution will remain second-order accurate provided the coefficient ν_a is taken to be order one.

The primary aim of writing the CGNVT code was to develop a flexible research program. In fact the code was written to solve a general class of (nonlinear) time

dependent equations. Thus it is not particularly optimized to be efficient with regard to either CPU time or storage.

6.4. *Numerical Accuracy and Efficiency*

We now present some results which show the accuracy and efficiency that can be expected from the scheme.

TEST 1. As a first test of accuracy a forcing function is added to each of the Navier–Stokes equations. The forcing is chosen to be that analytically derived function such that the true solution will be

$$\rho_{\text{true}} = 1.5 + \cos(2\pi x)\cos(2\pi y)\cos(2\pi t)$$
$$u_{\text{true}} = 1.25 + \cos(2\pi x)\sin(2\pi y)\cos(2\pi t)$$
$$v_{\text{true}} = \cos(\pi x)\sin(\pi y)\cos(\pi t)$$
$$T_{\text{true}} = \cos(\pi x)\sin(2\pi y)\cos(\pi t).$$

We integrated the forced equations on three grids, G_m, $m = 1, 2, 3$, which cover the same region of a cylinder in a channel, Fig. 18. Table V gives the number of mesh points on each of the three composite grids. The mesh spacings h_m were chosen to be in the ratio $(h_1 : h_2 : h_3) = (2 : \sqrt{2} : 1)$ (approximately).

The boundary conditions were no-slip on the cylinder, subsonic inflow on the left boundary, subsonic outflow at the right boundary, and symmetry on the top and bottom boundaries. The parameters used in this run were

$$\text{Re} = 50., \qquad \text{Ma} = 0.85, \qquad \text{Pr} = 1., \qquad T_0/\Delta T_0 = 1., \qquad \gamma = 1.4.$$

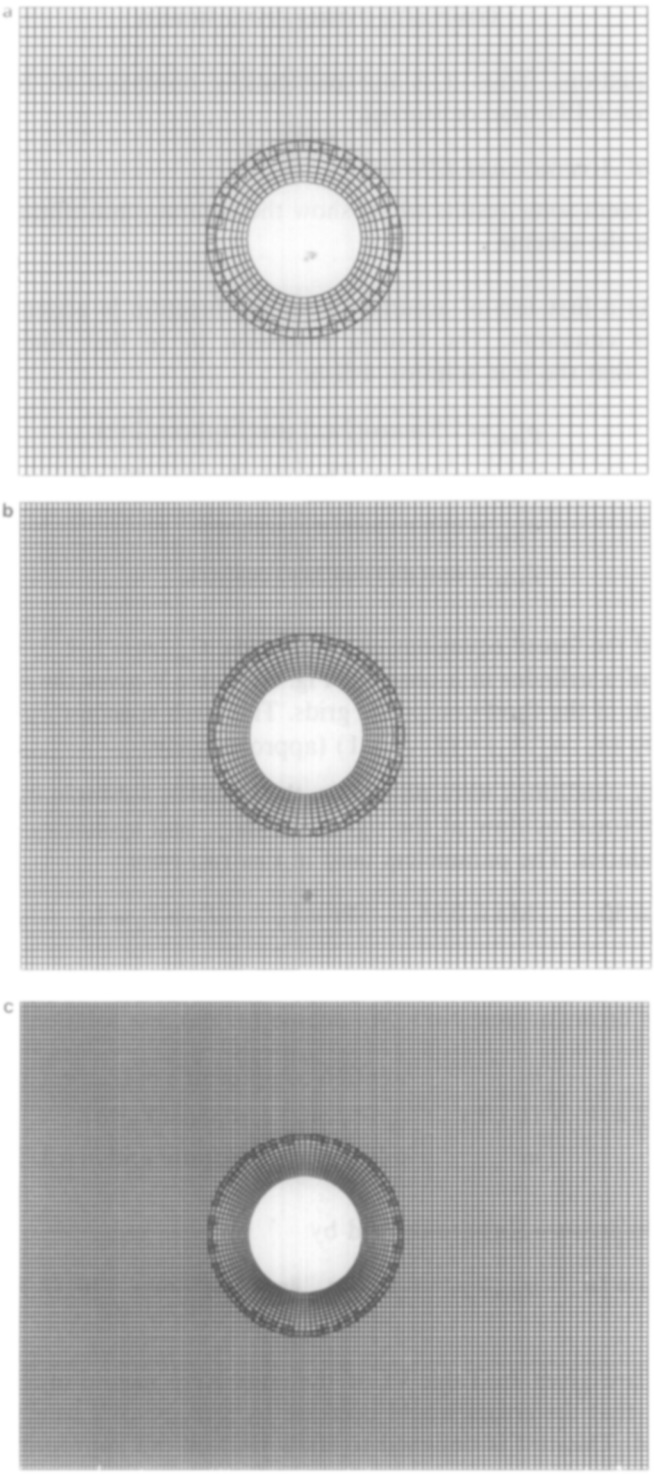
We integrated the equations starting from the exact initial conditions and computed the maximum errors at time $t = 1$. We denote the discrete solution on grid G_m by

$$\mathbf{u}_m(i, j, n, k): \begin{cases} n & \text{solution component } n = 1, \ldots, n_v \ (\rho, u, v, T) \\ (i, j, k) & \text{(grid point, component grid)} \\ m & \text{solution on composite grid } m = 1, 2, 3. \end{cases}$$

The maximum errors $e_m(n)$ are defined by

$$e_m(n) = \|\mathbf{u}_m - \mathbf{u}_{\text{true}}\|_\infty, \qquad \text{where} \quad \|\mathbf{u}_m\|_\infty = \max_{(i,j,k) \in G_m} |\mathbf{u}_m(i, j, n, k)|.$$

These errors are given in Table VI. If the code were second-order accurate, one would expect that the errors would decrease by a factor of 2 from one grid to the next finer grid. This test is designed to show that all the appropriate terms in the Navier–Stokes equations have been correctly discretized.



18. Grids G_1, G_2, and G_3 for testing the accuracy of the Navier–Stokes solver.

TABLE V

Number of Grid Points on Grids Used for Test 1 and Test 2

k	Grid G_1	Grid G_2	Grid G_3
1	70×50	99×71	139×199
2	51×7	72×10	101×13

TEST 2. As a second accuracy test we compute a supersonic flow on the grids G_m and estimate the errors. The parameters used in these runs are

$$\text{Re} = 50., \qquad \text{Ma} = 2.0, \qquad \text{Pr} = 0.71, \qquad T_0/\Delta T_0 = 1., \qquad \gamma = 1.4.$$

On the left edge of the domain we specify supersonic inflow boundary conditions. The top and bottom boundaries of the domain are specified with symmetry boundary conditions and subsonic outflow conditions are given at the right edge. The surface of the cylinder is defined as a no-slip wall with constant temperature. The equations are integrated in time with initial conditions $\rho = 1$, $u = 1$, $v = 0$, and $T = 0$ and $u = v = 0$ on the cylinder boundary (impulsively started cylinder). The solution develops a (viscous) shock which forms on the front face of the cylinder and then propagates to a position upstream. In Fig. 19 we compare the pressure contours on grids G_1, G_2, and G_3 at times $t = 1$ and $t = 3$. In Fig. 20 we show contours of ρ, u, v, and T on G_3 at $t = 10$ and in Fig. 21 we show a surface plot of the pressure on G_1 at $t = 10$. The contour plots are obtained by plotting contour lines on each component grid independently. The smooth alignment of contours between component grids is a good indication of the accuracy of the computed solution. This is a difficult problem, since as the shock crosses the interpolation boundary it is parallel to that boundary and so many interpolation points are affected by the shock at once. Recall that our discretization is not conservative; neither is the interpolation conservative. We emphasize, however, that our approach here is to have enough grid points to resolve the shock. If the computed solution is smoothly represented on the grid then the question of conservation is not crucial. Of course, if one does not want to resolve the shock then conservation

TABLE VI

Maximum Errors in Navier–Stokes Solver at $t = 1.$ for Test 1

n	$\|\mathbf{u}(n)\|_\infty$	$e_1(n)$	$e_2(n)$	$e_3(n)$	e_1/e_2	e_2/e_3
ρ	2.5	0.52	0.25	0.12	2.1	2.1
u	2.25	0.20	0.093	0.044	2.2	2.1
v	1.0	0.22	0.11	0.054	2.0	2.0
T	1.0	0.19	0.087	0.041	2.2	2.1

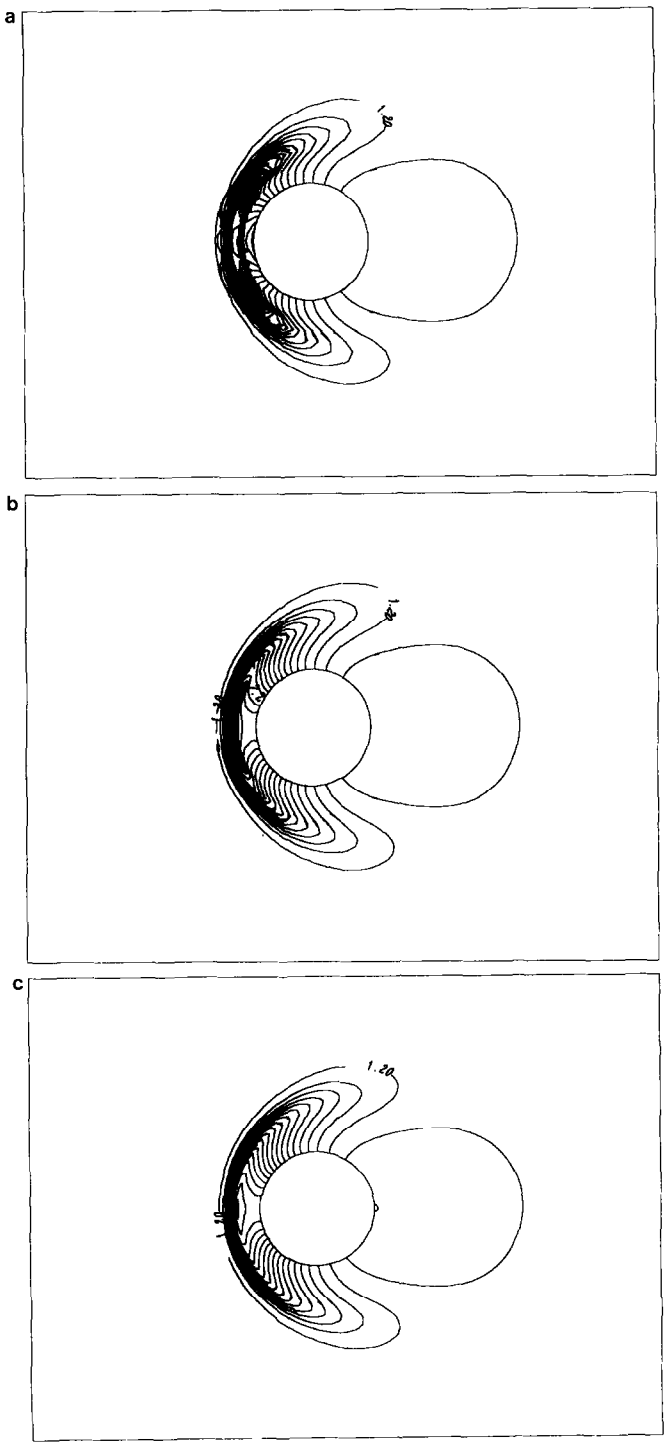


FIG. 19. Comparison of pressure contours for Test 2: (a) G_1, $t=2$; (b) G_2, $t=2$; (c) G_3, $t=2$; (d) G_1, $t=3$; (e) G_2, $t=3$; (f) G_3, $t=3$.

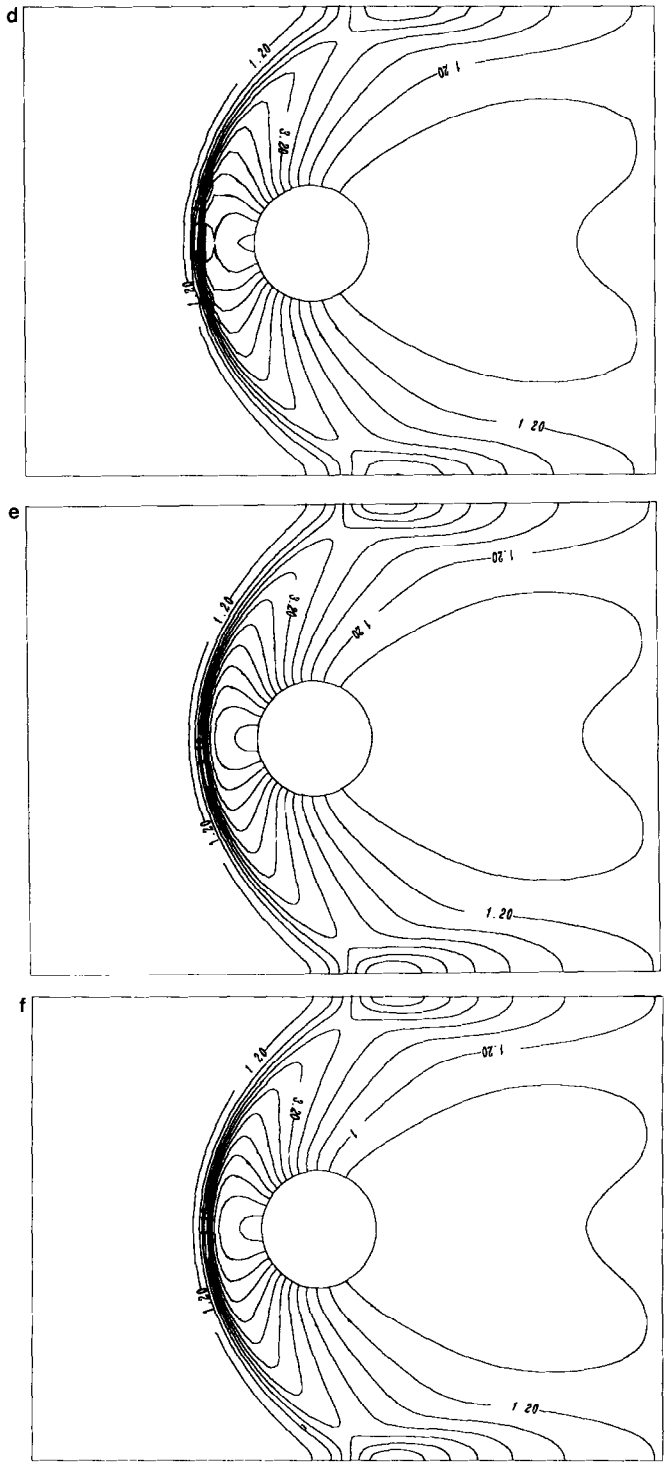


FIGURE 19—*Continued*

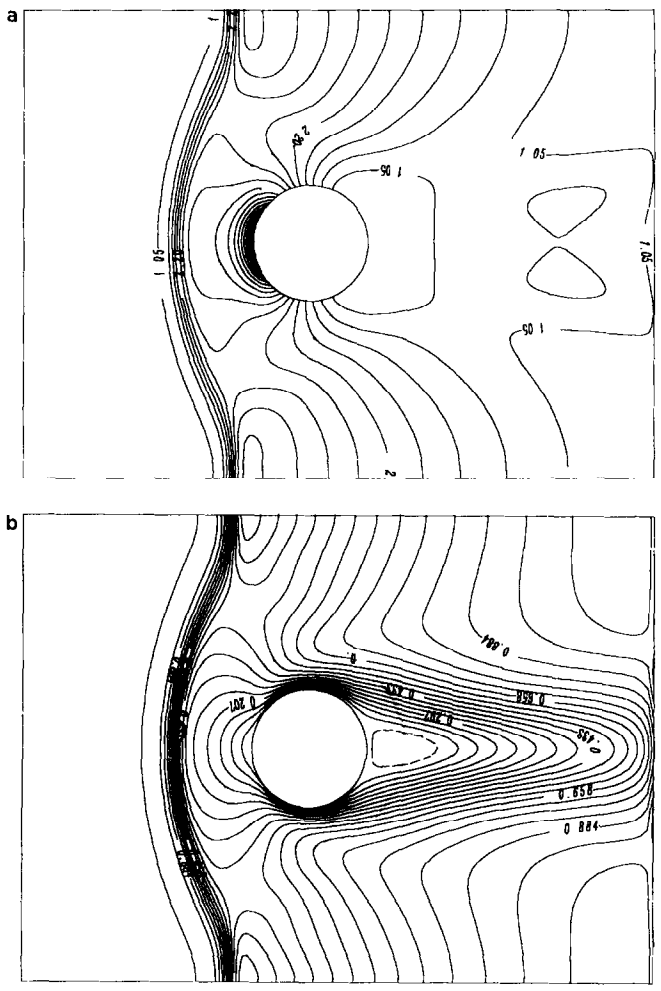


FIG. 20. Solution contours on G_3 at $t = 10$: (a) ρ; (b) u; (c) v; (d) T.

is important. The theoretical problem of conservation at the interpolation faces has been considered by Berger [7], while others [26, 3] have apparently been able to compute high Reynolds number flows without major difficulties. However, a systematic and thorough study of this question is appropriate for a future paper.

Figure 19a shows that the shock has some difficulty crossing the interpolation boundary on the coarsest grid (since the contours lines do not join up smoothly). However, the solution is well represented on the two finer grids. Define the l_2 norm of grid-function on grid G_m $\|\mathbf{u}_m\|_2$ by

$$\|\mathbf{u}_m\|_2^2 = \frac{\sum_{(i,j,k)\in G_3} |\hat{\mathbf{u}}_m(i, j, n, k)|^2}{\text{total number of grid points of } G_3}.$$

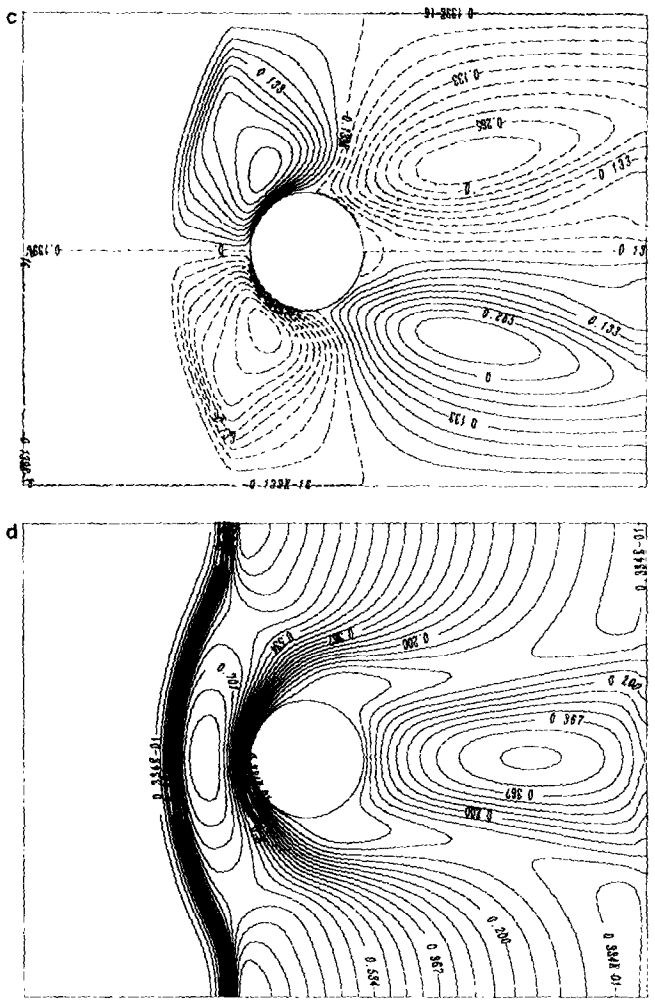


FIGURE 20—*Continued*

This norm is always computed with respect to the finest grid G_3 by interpolating the discrete solution on grids G_1 or G_2 to grid G_3 ($\hat{\mathbf{u}}_m$ denotes this interpolant). In Tables VII and VIII we show the differences between the solutions on the three composite grids at times $t = 1$ and $t = 10$. Note that if the solution were converging to second order, $\mathbf{u}_m = \mathbf{u}_{\text{true}} + Ch_m^2$, then

$$\frac{\|\mathbf{u}_1 - \mathbf{u}_3\|}{\|\mathbf{u}_2 - \mathbf{u}_3\|} \approx \frac{C(h^2 - h^2/4)}{C(h^2/2 - h^2/4)} = 3.$$

The ratios of the differences, as measured in either norm, are fairly consistent with this expected convergence rate.

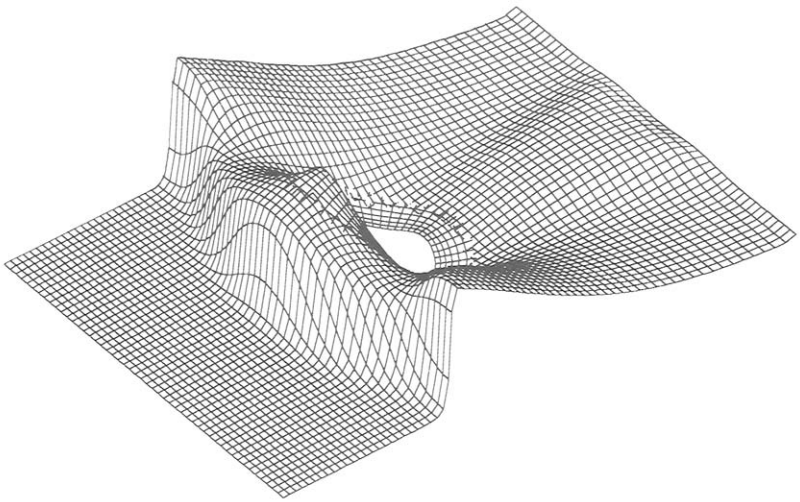


FIG. 21. Surface plot of pressure on G_1 at $t = 10$.

Efficiency. One of the possible advantages of a composite grid over an unstructured grid is that the composite grid, which consists of a set of logically rectangular grids, should be more efficient. We now make some remarks regarding this point.

Storage. The main difference in storage between a simple single-grid finite-difference time dependent code and the equivalent composite grid code which uses explicit interpolation is the storage of the flag array $kr(i, j, k)$. All other arrays are at most the dimension of the number of interpolation points. We show that the storage for these interpolation arrays and all the storage for pointers, etc. required by the DSK package is negligible compared to basic storage requirements of the time dependent code.

The CGNVT Navier–Stokes code uses quite a bit more storage than a basic scheme, since it integrates the equations implicitly, on some or all component grids,

TABLE VII

Maximum Errors in Navier–Stokes Solver at $t = 1$ and $t = 10$ for Test 2

	$\|\mathbf{u}_3\|_\infty$		$\|\mathbf{u}_1 - \mathbf{u}_3\|_\infty$		$\|\mathbf{u}_2 - \mathbf{u}_3\|_\infty$		$\|\mathbf{u}_1 - \mathbf{u}_3\|_\infty / \|\mathbf{u}_2 - \mathbf{u}_3\|_\infty$	
n	$t = 1$	$t = 10$	$t = 1$	$t = 10$	$t = 1$	$t = 10$	$t = 1$	$t = 10$
ρ	8.7	6.5	1.4	0.23	0.41	0.061	3.4	3.7
u	1.1	1.1	0.24	0.068	0.088	0.017	2.7	4.0
v	0.52	0.33	0.16	0.022	0.057	0.0067	2.8	3.3
T	1.1	0.83	0.35	0.081	0.14	0.017	2.5	4.8

TABLE VIII

l_2 Errors in Navier–Stokes Solver at $t = 1$ and $t = 10$ for Test 2

	$\|\mathbf{u}_3\|_\infty$		$\|\mathbf{u}_1 - \mathbf{u}_3\|_2$		$\|\mathbf{u}_2 - \mathbf{u}_3\|_2$		$\|\mathbf{u}_1 - \mathbf{u}_3\|_2 / \|\mathbf{u}_2 - \mathbf{u}_3\|_2$	
n	$t = 1$	$t = 10$	$t = 1$	$t = 10$	$t = 1$	$t = 10$	$t = 1$	$t = 10$
ρ	8.7	6.5	0.11	0.031	0.030	0.0078	3.6	4.0
u	1.1	1.1	0.023	0.0086	0.0076	0.0024	3.0	3.6
v	0.52	0.33	0.016	0.0035	0.0058	0.0012	2.8	2.9
T	1.1	0.83	0.035	0.0097	0.012	0.0028	3.6	3.5

using a sparse matrix solver. The storage for the sparse solver is orders of magnitude larger than all other storage. We now detail the storage used by CGNVT in order to solve a problem on grid G_1, Table IX. We list the storage by recognizable arrays and separate the storage required by the sparse solver. If the code were changed to use explicit interpolation then the storage for the sparse solver would not be required. We are able to easily determine all the storage requirements, since essentially all variables are allocated with the DSK package (Section 4) and stored on one large array; a simple call to a DSK routine displays the amount of storage allocated to real arrays, integer arrays, pointers, etc.

CPU Time. We now give sample CPU times for running the shock-cylinder test run on grid G_1. The grid itself took 1.6 s to compute with CMPGRD (grids G_2 and G_3 required 3.1 and 7.2 s, respectively, to generate). We performed two runs: *Run 1*

TABLE IX

Array Storage Required by CGNVT on Grid G_1

N = total number of grid points = $72 \times 52 + 53 \times 9 = 4221$	
1. x and y grid points = $2N$	8,442
2. $\partial r/\partial x$, derivatives of transformation = $4N$	16,884
3. Storage for explicit time stepping = $(4n_t + 6)N$	92,862
4. miscellaneous real arrays	2,511
5. flag array $kr = N$	4,221
6. miscellaneous integer arrays	947
7. DSK pointers etc.	2,904
Total for explicit time stepping portion	128,771
8. Sparse solver (integer arrays)	72,241
9. Sparse solver (real arrays)	2,601,766
Total for sparse solver	2,673,907
Grand total for implicit time stepping	2,802,678

TABLE X

CPU Time for Run 1 (Explicit Time-Stepping) and Run 2 (Implicit Time-Stepping) for CGNVT

	CPU/step	CPU_E/step	CPU_I/step	CPU/step/gridpoint	CPU/step/Δt
Run 1	0.11	0.049	0.063	$31. \times 10^{-6}$	45.9
Run 2	0.21	0.050	0.16	$59. \times 10^{-6}$	42.6

used explicit time-stepping and *Run 2* used semi-implicit time-stepping where the component grid around the cylinder was integrated implicitly. All times are for the CGNVT code running on an IBM 3090 with vector facility. We make no claims that our code is particularly fast. The CPU time given in Table X is decomposed into two portions:

1. CPU_E: time required in the explicit time-stepping of interior points.
2. CPU_I: time required by the sparse solver to solve the implicit time-stepping equations, to solve the interpolation equations and to solve the boundary conditions.

For the computation of the column CPU/step/gridpoint the number of grid points was taken as 3580; this number does not include those points which were eliminated by the composite grid algorithm. The final column in the table is the CPU time required to reach time $t = 1$. The CPU time per step for the explicit method was significantly less than the implicit method. However, since the implicit method could use a time step which was twice as large as the explicit method the overall CPU required was slightly less for the implicit method. The implicit method would have even more of an advantage if the cylinder grid had been stretched more.

It can be seen from the explicit run that solution of the interpolation and boundary conditions is taking longer than the solution of all the interior points. The basic reason for this poor performance is that the code was written to be run in an implicit time-stepping mode and is not efficient when using explicit time-stepping. A second reason is that the sparse back-solve routine we use does not run any faster when compiled in a vectorized mode.

A significant speedup per time step could probably be achieved by incorporating an efficient explicit-interpolation routine. Indeed, D. Brown reports that when using explicit interpolation in his finite volume code that the time required for interpolation is 11% of the time required to update the interior points [9]. Brown's code ran on a Cray X-MP on a composite grid similar to grid G_1.

6.5. *Numerical Results*

This section shows two examples of solving the Navier–Stokes equations on composite grids generated by CMPGRD. We present computations around an airfoil with multiple flaps and the flow around the read–write head of a magnetic

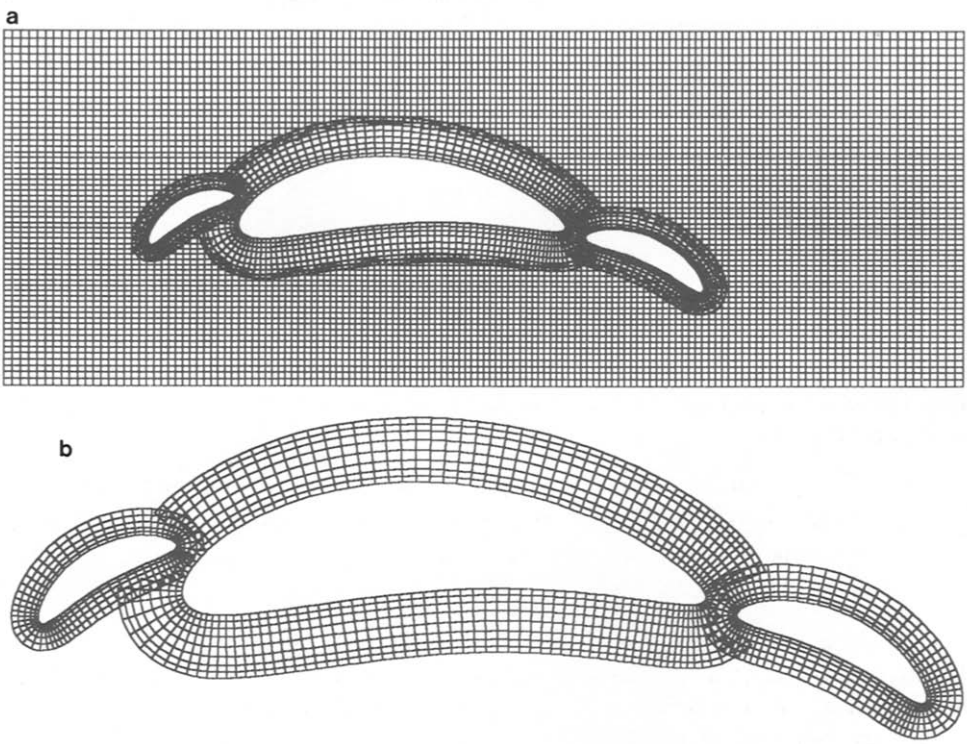


FIG. 22. Composite grid for airfoil with flaps.

storage device. We show contour and streamline plots of the solutions on the composite meshes. The contour plots are performed by plotting contours on each component grid separately, using the same contour levels. The contours align only if the solutions agree between different meshes. All CPU times stated in this section are for the IBM 3090VF.

I. *Flow past an airfoil with multiple flaps.* As a first example we consider the flow around an airfoil with two flaps. For simplicity the airfoil and flaps are defined as Joukowsky airfoils. Since each component grid can be generated independently of the other component grids it is simple matter to add as many flaps as desired, Fig. 22.

TABLE XI

CPU Time for Flow Past an Airfoil with Flaps; 8101 Gridpoints

CPU/step	CPU_E/step	CPU_I/step	CPU/step/gridpoint	CPU/step/Δt
0.725	0.11	0.60	$89. \times 10^{-6}$	145.

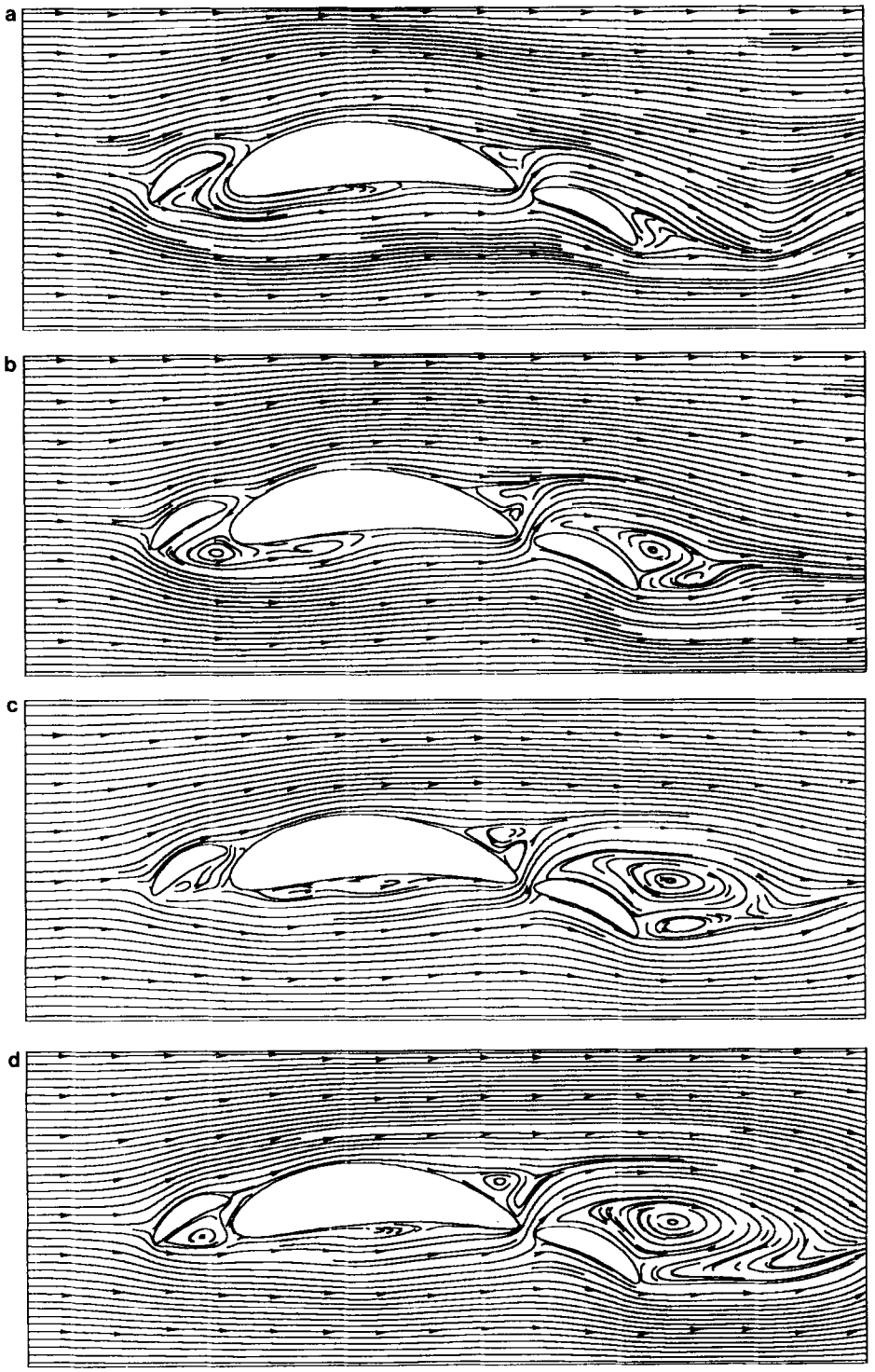


FIG. 23. Streamlines at times $t = 3, 5, 7, 9$.

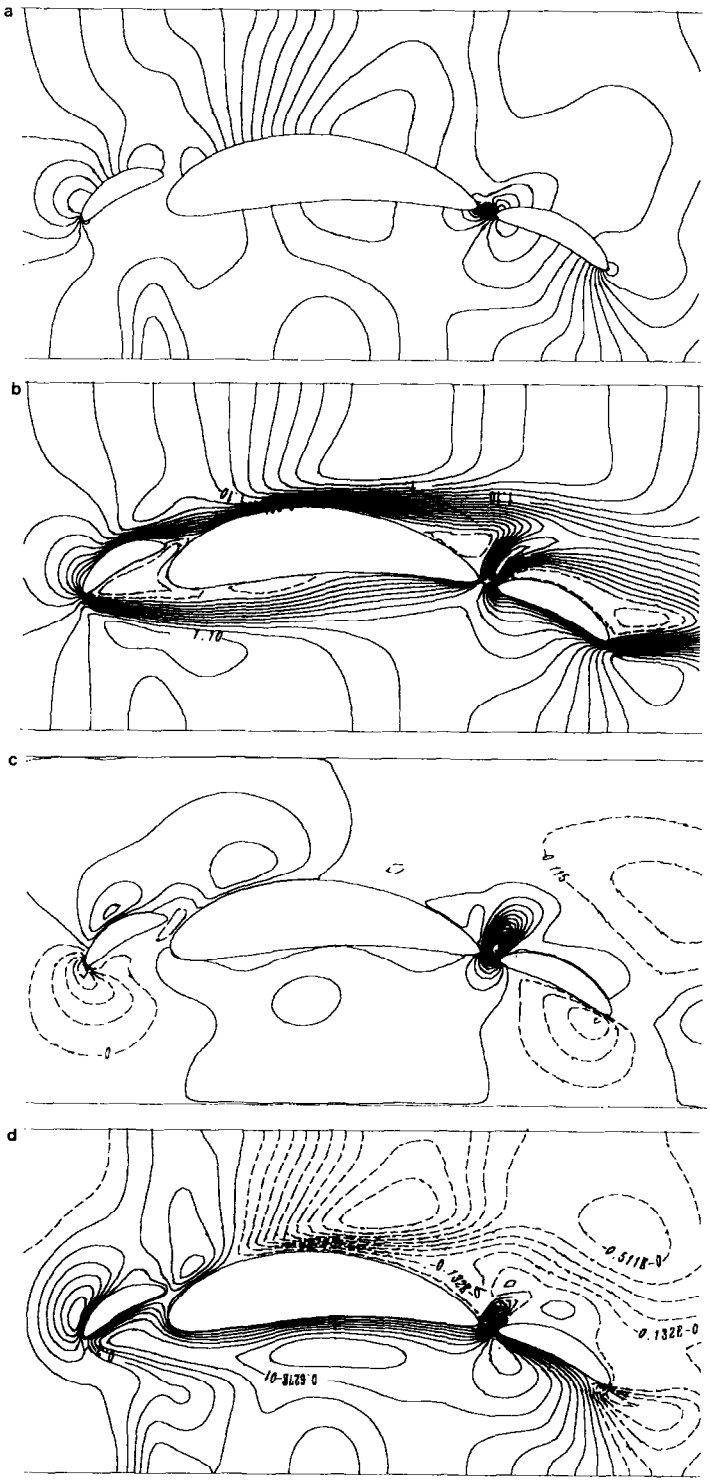


FIG. 24. Solution contours at $t = 5$: (a) p; (b) u; (c) v; (d) T.

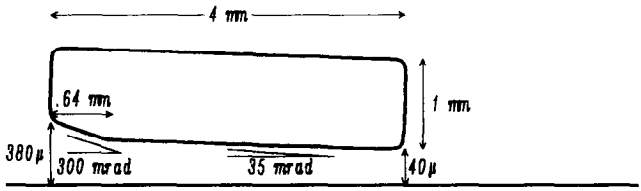


FIG. 25. Two-dimensional slider bearing.

We consider a flow of Mach number Ma = 0.5 and Reynolds number Re = 370 (based on the length of the main airfoil). The boundary conditions are taken as no-slip on the airfoil and flaps, subsonic inflow on the left edge, subsonic outflow on the right edge and symmetry conditions on the top and bottom. The equations are integrated with implicit time-stepping on the airfoil and flap grids and explicit time-stepping on the rectangular grid. The construction of the grid with CMPGRD required about 9.3 s, while a breakdown of the CPU time required for time-stepping is given in Table XI. Recall that CPU_E and CPU_I indicate the times

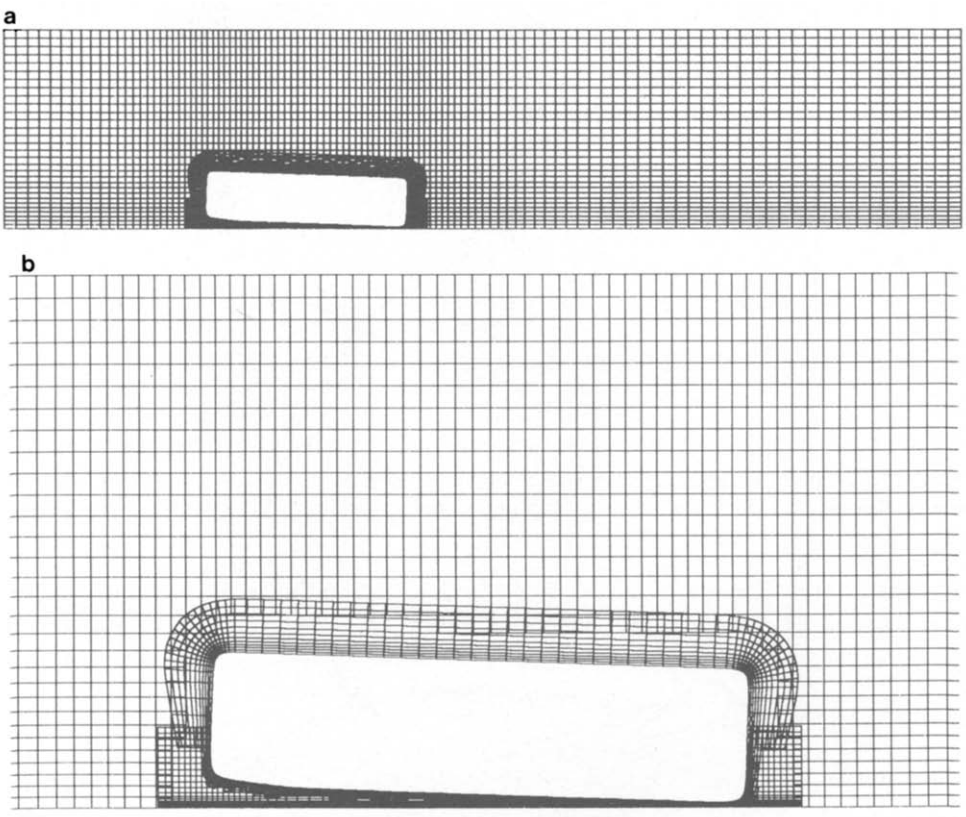


FIG. 26. Composite grid for the slider bearing and magnified views.

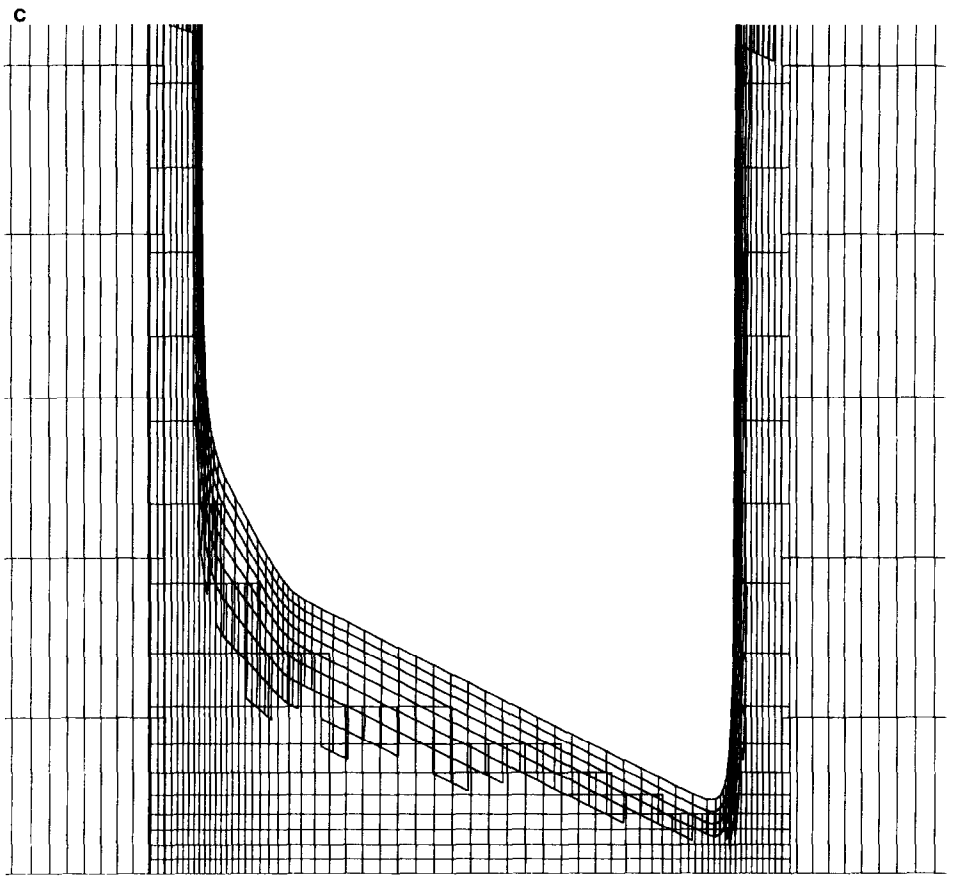


FIGURE 26—*Continued*

required for the explicit and implicit parts of the time-step while CPU/step/Δt is the time required to integrate the equations to time $t = 1$. Plots of instantaneous streamlines are shown in Fig. 23 and contours of p, u, v, T at $t = 5$. are given in Fig. 24.

II. *Flow around a disk read–write head.* As a second example we consider the flow around the body depicted in Fig. 25. This is meant to be a two-dimensional model of the body which holds the read–write head in a magnetic disk storage device. The bottom surface is the computer disk which moves from left to right at a constant speed. The head flies over the disk, using the air which flows underneath it to support it. This arrangement is usually called a slider-bearing or air-bearing. The difficult part of this problem is the fact that the gap between the head and the disk surface is so small. We generated a component grid around this body using the method outlined in Section 5.4. The composite grid is shown in Fig. 26 with

magnified views of the grid in the narrow gap. For this computation the minimum gap height is 100 times smaller than the length of the slider. The Reynolds number based on the length of the body is $Re = 200$ and the Mach number $Ma = 0.14$. The actual device flies at a much smaller gap height and a larger Reynolds number. Instantaneous streamlines of the computed time dependent flow are shown in Fig. 27, including a magnified view of the flow underneath the slider. Further details can be found in [22].

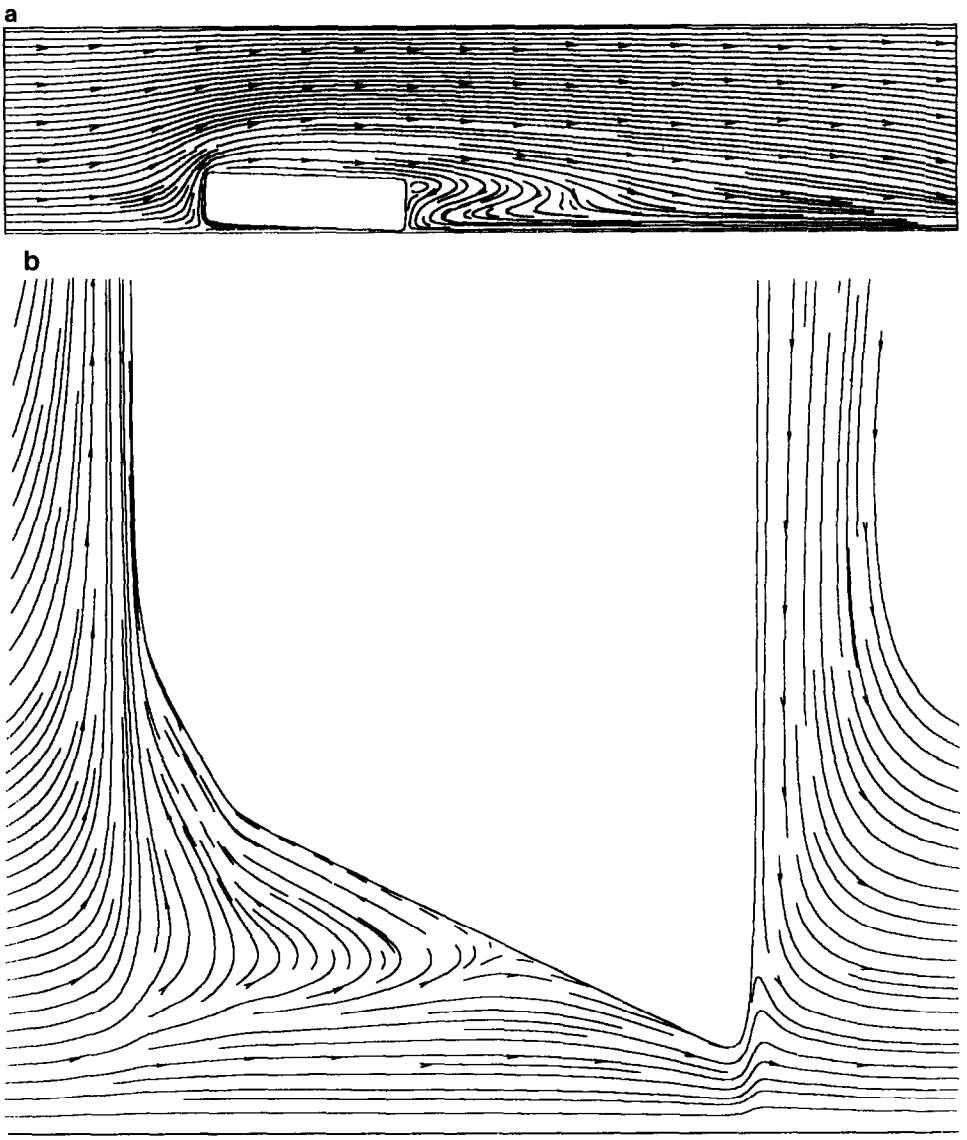


FIG. 27. Streamlines around the slider bearing at time $t = 5$ and a magnified view.

7. Conclusions

We have described how composite overlapping grids can be used to solve partial differential equations. A Fortran code, CMPGRD, has been written to construct very general composite grids. We have presented some of the features of CMPGRD and have outlined the most important part of the code: the algorithm that the program uses in order to determine the regions of overlap. We have discussed how to choose the order of accuracy of interpolation so as to maintain overall accuracy. We have shown some techniques for solving elliptic and time dependent PDEs and have displayed the accuracy of these methods.

In future work we plan to discuss a number of important issues related to the potential usefulness of composite grids. For example, we have been collaborating with David Brown on automatic methods for adaptively creating new component grids. In this regard we are working on ways to speed up CMPGRD so that it will become feasible to generate a composite grid many times in the course of a computation. There is also a need for a careful study related to the problem of conservation at the interpolation points. Since the fall of 1988, CMPGRD has been changed to construct three-dimensional composite overlapping grids. The code was extended so that a two-dimensional grid would just be a special case of a three-dimensional grid. This two/three-dimensional version of CMPGRD will be described in a forthcoming paper.

We distribute, free of charge, the source code for CMPGRD and the DSK package as well as many of the associated utility routines, including the elliptic solver, the explicit and implicit interpolation routines, a time dependent code to solve linear convection–diffusion equations, the multigrid solver, and plotting routines. Please contact one of the authors if you would like to obtain a copy of these programs.

Acknowledgments

The authors thank Professor Heinz-Otto Kreiss for encouraging the use of composite meshes and Barbro Kreiss for her original composite grid program which served as a basis for CMPGRD. Thanks also to Dr. David Brown and Dr. Don Schwendeman for their help in testing the programs. Plotting was performed with the aid of the NCAR graphics routines and the surface plotting routine, GRAFIC, of Melvin Prueitt.

References

1. E. H. Atta and J. Vadyak, *AIAA J.* **21**, No. 9, 1271 (1983).
2. T. J. Baker, in *Numerical Grid Generation in Computational Fluid Mechanics '88*, edited by S. Sengupta *et al.* (Pineridge Press, Swansea, Great Britain, 1988), p. 675.
3. J. A. Benek, J. L. Steger, and F. C. Dougherty, AIAA Paper No. 83-1944, 1983 (unpublished).
4. J. A. Benek, P. G. Buning, and J. L. Steger, "A 3-D Chimera Grid Embedding Technique," AIAA Paper No. 85-1523, 1985 (unpublished).
5. M. J. Berger and J. Oliger, *J. Comput. Phys.* **53**, 484 (1984).
6. Berger, M. J., "Adaptive Finite Difference Methods in Fluid Dynamics," Courant Mathematics and Computing Laboratory Report DOE/ER/03077-277, 1987 (unpublished).

7. M. J. BERGER, *SIAM J. Numer. Anal.* **24**, No. 5, 967 (1987).
8. J. U. BRACKBILL AND J. S. SALTZMAN, *J. Comput. Phys.* **46**, 342 (1982).
9. D. L. BROWN, Computing and Communications Division, Los Alamos National Laboratory, Los Alamos, NM, private communication (1989).
10. D. L. BROWN, G. CHESSHIRE, AND W. D. HENSHAW, "Composite Grid Data: An Explanation of the CMPGRD Composite Grid Data Structure," IBM internal report RC 14354, 1988 (unpublished).
11. D. L. BROWN, G. CHESSHIRE, AND W. D. HENSHAW, "Getting Started with CMPGRD, Introductory User's Guide and Reference Manual," Los Alamos National Laboratory Report LA-UR-89-1294, 1989 (unpublished).
12. P. G. BUNING, I. T. CHIU, S. OBAYASHI, Y. M. RIZK, AND J. L. STEGER, "Numerical Simulation of the Integrated Space Shuttle Vehicle in Ascent," AIAA Paper No. 88-4359-CP, 1988 (unpublished).
13. G. CHESSHIRE, "Composite Grid Construction and Applications," Ph.D. thesis, California Institute of Technology, 1986 (unpublished).
14. G. CHESSHIRE AND W. D. HENSHAW, "The DSK Package, A Data Structure for Efficient Fortran Array Storage (Refence Guide for the DSK Package)," IBM internal report RC 14353, 1988 (unpublished).
15. S. A. COONS, "Surfaces for Computer Aided Design of Space Forms," Thesis, Dept. of Mech. Eng., Massachusetts Inst. of Tech., Cambridge, 1964 (unpublished).
16. T. J. CHUNG AND G. R. KARR (Eds.), *Finite Element Analysis in Fluids, Proceedings of the Seventh International Conference on Finite Element Methods in Flow Problems* (UAH Press, Huntsville, AL, 1989).
17. S. C. EISENSTAT, M. C. GURSKY, M. H. SCHULTZ, AND A. H. SHERMAN, "Yale Sparse Matrix Package," Research Reports 112 and 114, Yale University, Dept. of Computer Science, 1977 (unpublished).
18. B. GUSTAFSSON, *Math. Comput.* **29**, 396 (1975).
19. J. HÄUSER AND C. TAYLOR (Eds.), *Numerical Grid Generation in Computational Fluid Dynamics* (Pineridge Press, Swansea, Great Britain, 1986).
20. W. D. HENSHAW AND G. CHESSHIRE, *SIAM J. Sci. Stat. Comput.* **8**, No. 6, 914 (1987).
21. W. D. HENSHAW, Thesis, Department of Applied Mathematics, California Institute of Technology, 1985 (unpublished).
22. W. D. HENSHAW, L. G. REYNA, AND J. A. ZUFIRIA, "Compressible Navier Stokes Computations for Slider Air-Bearings," IBM internal report RC 13931, 1988 (unpublished).
23. B. KREISS, *SIAM J. Sci Stat. Comput.* **4**, No. 2, 270 (1983).
24. H. O. KREISS, *Math. Comput.* **26**, 605 (1972).
25. D. MAVRIPLIS, Thesis, Dept. of Mechanical and Aerospace Engineering, Princeton University, 1988 (unpublished).
26. E. PÄRT, Uppsala University, Department of Scientific Computing, Internal Report No. 88-07, 1988 (unpublished).
27. L. G. M. REYNA, Thesis, California Institute of Technology, 1986 (unpublished).
28. S. SENGUPTA, J. HÄUSER, P. R. EISEMAN AND J. F. THOMPSON, *Numerical Grid Generation in Computational Fluid Mechanics '88* (Pineridge Press, Swansea, Great Britain, 1988).
29. G. STARIUS, *Numer. Math.* **28**, 243 (1977).
30. G. STARIUS, *Numer. Math.* **35**, 241 (1980).
31. J. L. STEGER AND P. G. BUNING, *Developments in the Simulation of Compressible Inviscid and Viscous Flow on Supercomputers, Progress and Supercomputing in Computational Fluid Dynamics*, edited by E. M. Murman and S. S. Abarbanel (Birkhäuser, Boston, 1985), p. 67.
32. J. L. STEGER AND J. A. BENEK, *Comput. Methods Appl. Mech. Eng.* **64**, 301 (1987).
33. J. F. THOMPSON, F. C. THAMES, AND C. W. MASTIN, *J. Comput. Phys.* **15**, 299 (1974).
34. J. F. THOMPSON, (Ed.), *Numerical Grid Generation* (Elsevier, New York, 1982).
35. J. F. THOMPSON, *AIAA J.* **26**, 271 (1988).
36. A. M. WINSLOW, *J. Comput. Phys.* **54**, 115 (1984).