JOURNAL OF COMPUTATIONAL PHYSICS 57, 251–284 (1985)

Symbolic Manipulation and Computational Fluid Dynamics*

STANLY STEINBERG

Department of Mathematics and Statistics, University of New Mexico, Albuquerque, New Mexico 87106

AND

PATRICK J. ROACHE

Ecodynamics Research Associates, Inc., Albuquerque, New Mexico 87108

Received April 29, 1983; revised May 2, 1984

The problem of numerically integrating general elliptic differential equations in irregular two and three dimensional regions is discussed. The method used numerically computes a transformation of the given region into a rectangular region. The numerical coordinate transformation is determined by requiring that the components of the transformation satisfy inhomogeneous Laplace or more general equations. The transformation is then used to transform the differential equation and the boundary conditions to the rectangular region. The boundary value problem in the rectangular region is integrated using one of the standard methods for general elliptic equations. The use of the existing software reduces the problem to analytically transforming the given differential equation and the Laplacian to general coordinate frame and then writing subroutines that will tabulate the coefficients of these differential equations using the tabulated coordinate transformation. This method has been successfully used in two dimensions so we are concerned with the three dimensional extension of the existing codes where the major problem encountered is the volume of algebra and coding required to complete the method. To overcome these difficulties, a symbol manipulation program in VAXIMA is written that has as input the formula for the given differential equation in some natural coordinates and has as output the required FORTRAN subroutines. Because of the complexity of the resulting code, code validation was performed by systematic truncation error testing. The paper concludes with a discussion of the problems encountered in using a symbol manipulator to write large FORTRAN codes.

* This work was partially supported by the U.S. Air Force Office of Scientific Research, by the U.S. Army Research Office, and by the National Science Foundation Grant MCS-8102683.

0021-9991/85 $3.00

581/57/2-7

1. Introduction

The last decade has seen a virtual explosion in the use of boundary fitted, generally non-orthogonal, coordinate systems for solving partial differential equations in complex geometries using finite difference methods. The wide range of physical problems treated include fluid dynamics, heat transfer, electric fields, and structures [2, 9, 15, 22]. The "modern era" of grid generation can be dated from the 1974 paper by Thompson, Thames and Mastin [21]. The automatic generation of such boundary fitted coordinate systems has become a significant research area in its own right, as evidenced by conferences and publications devoted entirely to this subject, e.g. [2, 9, 22]. The advantages and importance of this approach have been thoroughly discussed in, e.g. [2, 9, 22].

We begin this paper by presenting the analytic ideas behind a method for numerically integrating boundary value problems for elliptic partial differential equations in connected n-dimensional regions. We are mostly interested in 3-dimensional regions; however, we present the n-dimensional case because it requires essentially no extra effort. The analytic formulation that we present is then used to write a symbol manipulation program that uses, as input, the given differential equation and produces, as output, complete FORTRAN subroutines. These subroutines are combined with standard software to produce a program that is now being used to study certain laser and fluid dynamics problems.

One important aspect of these methods is that they are specifically designed to efficiently handle irregular regions, e.g., Fig. 1; in fact, the region is not required to be simply connected.

We will be interested in general elliptic problems. In the linear case we set

$$Lf = \sum_{i,j=1}^{n} a_{ij} \frac{\partial^2}{\partial x_i \partial x_j} f + \sum_{i=1}^{n} b_i \frac{\partial}{\partial x_i} f + cf + d, \tag{1.1}$$

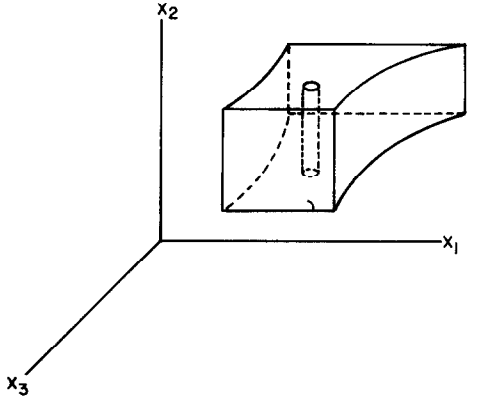

FIGURE 1

where

$$\begin{aligned} a_{ij} &= a_{ij}(\mathbf{x}), \qquad 1 \leqslant i, j \leqslant n, \\ b_i &= b_i(\mathbf{x}), \qquad 1 \leqslant i \leqslant n, \\ c &= c(\mathbf{x}), \qquad d = d(\mathbf{x}), \\ \mathbf{x} &= (x_1, x_2, \ldots, x_n). \end{aligned} \tag{1.2}$$

Given a_{ij}, b_i, c, d and appropriate boundary conditions, the problem that interests us is to determine a numerical approximation to $f = f(\mathbf{x})$ which is to be a solution of

$$Lf = 0. \tag{1.3}$$

Here, all functions and the boundary of the region are assumed to be smooth enough to guarantee that the boundary value problem, the differential equation with the boundary conditions, has a smooth solution. To study nonlinear problems we allow the coefficients of the differential equation to depend on f or the derivatives of f. We will discuss the linear case and note the changes necessary to study nonlinear problems.

The general idea of the approach that is considered in this paper is to transform the given problem into a problem that is easier to handle numerically. Because there is a substantial advantage to using finite difference methods on rectangular regions, the given region will be transformed to a rectangular region. The transformation will be written $\xi = \xi(\mathbf{x})$, and we will refer to the $\mathbf{x}$ coordinates as physical space and the ξ coordinates as the transformed space (see Fig. 2). Because of the complexity of the regions that interest us, it will be impossible to determine formulas for the transformation. However, such a coordinate transformation can be determined numerically by using one of several standard methods [4–6, 12, 21, 22].

Once the coordinate transformation is found, it is necessary to transform the

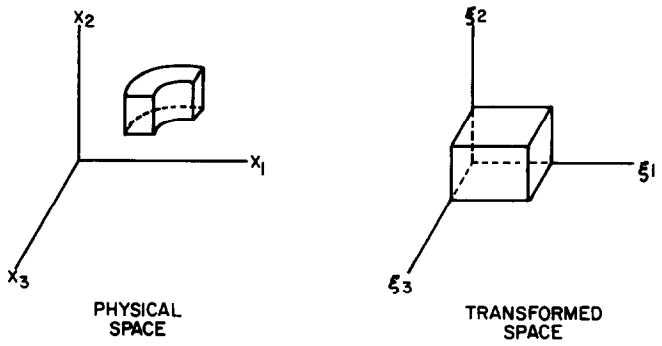

FIGURE 2

given boundary value problem to the new coordinates. Because no analytic formula for the transformation is found, it is necessary to transform the given differential equation into a general coordinate frame. Centered differences are used to convert the differential equation in the transformed space to finite difference form. The coefficients of the finite difference equation are called the *stencil* of the difference equation. We use an iterative hopscotch SOR algorithm to solve the difference equations [16]. The SOR program requires, as input, the arrays that tabulate the stencil of the finite difference equations. Thus, one of the subroutines that the symbol manipulation program writes does the tabulation of the stencil when it is given a tabulation of the coordinate transformation. Currently, we are using only Dirichlet boundary conditions, which are hand coded.

The method that we use to determine the coordinate transformation requires each component of the transformation to be the solution of an inhomogeneous Laplace equation in physical space [10]. The requirement that the boundary of the given region is mapped to the boundary of the rectangular region provides the boundary conditions for the Laplace equations. Experience shows that this formulation has the advantage of producing good transformations over a wide range of problems but has the disadvantage of being formulated in physical space. To take advantage of the rectangular region, the Laplacian must be transformed to the rectangular region using the transformation that is yet to be determined. This results in a set of coupled nonlinear elliptic equations that must be solved for the transformation. Again, centered differences are used to reduce the differential equations to finite difference form and the resulting finite difference equations are solved by using an iterative hopscotch SOR procedure [16]. As before, the SOR solver requires, as input, a stencil that tabulates the coefficients of the difference scheme. Thus, the symbol manipulation program produces another subroutine that tabulates the stencil that is used in determining the coordinate transformation.

In this paper we will focus our attention on the problems of transforming the given differential equation and the Laplacian to a general coordinate frame, converting the resulting differential equations to finite difference form, and then using the resulting formulas to write symbol manipulation programs that will produce the required FORTRAN subroutines. We also describe a procedure for validating the resulting code.

In Section 2, we set up our notation and review some properties of transformations. In the applications which we have in mind it is inappropriate to assume that the coordinate transformation preserves volume (has constant Jacobian) or that the transformation is orthogonal. We begin by reviewing some standard formulas for transforming derivatives and then compute the coefficients of the given differential equation in the new coordinate frame in a form convenient for symbol manipulation. Here, we use the formulas for the derivative of the inverse of a matrix function to greatly simplify our results. Instead of assuming that the components of the coordinate transformation satisfy Laplace's equation, we assume they satisfy a general elliptic equation, calculate the form of this equation in the new frame, and then reduce these formulas to the case we need. It is hoped that the general for-

mulas will be of use in some problems that we are considering. We also discuss the transformation of derivatives of parameters that may appear in the coefficients.

In Section 3 we use centered differences to reduce a general elliptic differential equation to finite difference form. Again, we do a bit of algebra to put the formulas for the stencil in a form convenient for symbol manipulation. We also observe that the stencil contains many symmetries, and we use this information to simplify the formulas.

In Section 4 we list and discuss the symbol manipulation program used to write the subroutines. Currently, the symbol manipulation program is restricted to 2 or 3 dimensions but could be easily extended to higher dimensions if we had applications for the extension. The subroutine that computes the stencil for the given differential equation is called SETM2 or SETM3 (depending on the dimension) while the subroutine that calculates the stencil for determining the coordinate transformation is called SETGM2 or SETGM3. The acronyms are for "set matrix dimension" and "set grid matrix dimension."

We use the symbol manipulation program VAXIMA that is running on a VAX 11/780 computer. This program is a VAX version of the symbol manipulation program MACSYMA [12]. The choice of this symbol manipulator was dictated by the experience of the first author. However, we believe that our programs could be implemented in most general purpose symbol manipulators. It is important to note that our analytic formulation eliminates the need for the symbol manipulation program to know the multivariate form of the chain rule. It is also important to note that the completion of this project in a reasonable time while using reasonable effort was greatly facilitated by the large number of procedures available in VAXIMA. References [1, 7, 11, 14, 23] describe some work related to our work.

In Section 5 we give a listing of SETM3 and discuss some of the advantages and disadvantages of machine written code. In Section 6 we describe the testing procedure that was used to validate the complete solver.

We began this project with a substantially different approach than the final version. In Section 7 we describe the old approach, which was successful, and discuss why the new approach is better. We do not consider that this project is complete. However, this report represents what we believe is significant and useful progress, and we hope that others will find the techniques described herein useful. We also describe some of the additional work that needs to be done.

In this paper we will concentrate on symbol manipulation problems and will refer the reader to the literature for discussions of numerical problems.

2. Coordinate Transformations

In this section we review some standard material on coordinate transformations and derive some formulas appropriate for our symbol manipulation programs. In

what follows we will freely change between abstract vector and matrix notation, and component notation. Thus, if $\mathbf{x}$ is a vector, then we write

$$\mathbf{x} = (x_i) = (x_1, x_2, \ldots, x_n)$$

and if M is a matrix, then we write $M = (M_{ij})$. We think of $\mathbf{x}$ as a column vector and j as the row index of the matrix and then

$$M\mathbf{x} = \left(\sum_{j=1}^{n} M_{ij} x_j \right). \tag{2.1}$$

We also assume that, unless otherwise noted, all sums run from 1 to n. We proceed by assuming that we are given a problem in the physical variables $\mathbf{x}$ and that we will transform the problem to the transformed variables ξ. The physical variables may be Cartesian, spherical, cylindrical or other coordinates. In our applications we will be doing numerical calculations in the transformed space so we will want to write all of our formulas in terms of the ξ variables. Thus we write

$$\mathbf{x} = \mathbf{x}(\xi), \tag{2.2}$$

and choose the Jacobian matrix J to be

$$J = (J_{ij}) = \left(\frac{\partial x_j}{\partial \xi_i} \right) = \begin{Bmatrix} \dfrac{\partial x_1}{\partial \xi_1} & \dfrac{\partial x_2}{\partial \xi_1} & \cdots \\ \dfrac{\partial x_1}{\partial \xi_2} & \cdots & \\ \cdots & & \cdots \end{Bmatrix} \tag{2.3}$$

and K to be the cofactor matrix of J. Thus if $\Delta = \textit{determinant}\ (J)$, then

$$J^{-1} = \frac{K}{\Delta}. \tag{2.4}$$

The chain rule gives

$$\frac{\partial}{\partial \xi_i} = \sum_j J_{ij} \frac{\partial}{\partial x_j} \tag{2.5}$$

and multiplying by K/Δ gives

$$\frac{\partial}{\partial x_i} = \frac{1}{\Delta} \sum_j K_{ij} \frac{\partial}{\partial \xi_j}. \tag{2.6}$$

The formula for $\partial/\partial x_i$ can be used to compute the second derivatives,

$$\frac{\partial^2}{\partial x_i\,\partial x_j}=\sum_{rs}\frac{1}{\Delta}K_{ir}\frac{\partial}{\partial\xi_r}\left(\frac{1}{\Delta}K_{js}\frac{\partial}{\partial\xi_s}\right)$$
$$=\sum_{rs}\frac{1}{\Delta^2}K_{ir}K_{js}\frac{\partial^2}{\partial\xi_r\,\partial\xi_s}+\sum_{rs}\frac{1}{\Delta}K_{ir}\frac{\partial}{\partial\xi_r}\left(\frac{1}{\Delta}K_{js}\right)\frac{\partial}{\partial\xi_s}. \tag{2.7}$$

If ∂ is any derivative, then the rule for differentiating the inverse of a matrix, $\partial A^{-1}=-A^{-1}\,\partial A A^{-1}$, gives

$$\frac{\partial}{\partial\xi_r}\left(\frac{1}{\Delta}K_{js}\right)=-\frac{1}{\Delta^2}\sum_{uv}K_{ju}\frac{\partial}{\partial\xi_r}J_{uv}K_{vs}, \tag{2.8}$$

and thus

$$\frac{\partial^2}{\partial x_i\,\partial x_j}=\frac{1}{\Delta^2}\sum_{rs}K_{ir}K_{js}\frac{\partial^2}{\partial\xi_r\,\partial\xi_s}-\frac{1}{\Delta^3}\sum_{rsuv}K_{ir}K_{ju}K_{vs}\left(\frac{\partial}{\partial\xi_r}J_{uv}\right)\frac{\partial}{\partial\xi_s}. \tag{2.9}$$

Recall that the original differential equation has the form

$$Lf=\sum_{ij}a_{ij}\frac{\partial^2}{\partial x_i\,\partial x_j}f+\sum_i b_i\frac{\partial}{\partial x_i}f+cf+d. \tag{2.10}$$

In the new coordinates the equation will become

$$\tilde{L}\tilde{f}=\sum_{ij}\tilde{a}_{ij}\frac{\partial^2}{\partial\xi_i\,\partial\xi_j}\tilde{f}+\sum_i\tilde{b}_i\frac{\partial}{\partial\xi_i}\tilde{f}+\tilde{c}\tilde{f}+\tilde{d}. \tag{2.11}$$

Here a_{ij}, b_i, c, d and f depend on x while $\tilde{a}_{ij}$, $\tilde{b}_i$, $\tilde{c}$, $\tilde{d}$ and $\tilde{f}$ depend on ξ. We also assume that $a_{ij}=0$ and $\tilde{a}_{ij}=0$ for $i<j$. Substituting (2.6) and (2.9) into (2.10) and comparing to (2.11) gives (δ_{ij} is the Kronecker delta)

$$\tilde{a}_{ij}=\frac{1}{\Delta^2}(2-\delta_{ij})\sum_{rs}a_{rs}K_{ri}K_{sj}, \tag{2.12}$$

$$\tilde{b}_i=-\frac{1}{\Delta^3}\sum_{jkurv}a_{jk}K_{ui}K_{jr}K_{kr}\frac{\partial}{\partial\xi_v}J_{ru}+\frac{1}{\Delta}\sum_j b_jK_{ji}, \tag{2.13}$$

$$\tilde{c}=c,\qquad \tilde{d}=d. \tag{2.14}$$

The formula for $\tilde{b}$ can be simplified using the formula for $\tilde{a}$,

$$\tilde{b}_i=-\frac{1}{\Delta}\sum_{ruv}\tilde{a}_{vr}K_{ui}\frac{\partial}{\partial\xi_v}J_{ru}+\frac{1}{\Delta}\sum_j b_jK_{ji}. \tag{2.15}$$

To help with the symbol manipulation we set up (recall that $J_{ru} = \partial x_u / \partial \xi_r$)

$$S_{ij} = \sum_{rs} a_{rs} K_{ri} K_{sj}, \qquad 1 \leqslant i, j \leqslant n, \tag{2.16}$$

$$U_i = \sum_{ruv} S_{vr} K_{ui} \frac{\partial^2 x_u}{\partial \xi_v \, \partial \xi_r}, \qquad 1 \leqslant i \leqslant n, \tag{2.17}$$

$$T_i = \sum_{j} b_j K_{ji}, \qquad 1 \leqslant i \leqslant n. \tag{2.18}$$

Note that $S_{ij} = S_{ji}$. These definitions give

$$\tilde{a}_{ij} = \frac{1}{\Delta^2} (2 - \delta_{ij}) S_{ij}, \qquad 1 \leqslant i \leqslant j \leqslant n, \tag{2.19}$$

$$\tilde{b}_i = -\frac{1}{\Delta^3} U_i + \frac{1}{\Delta} T_i, \qquad 1 \leqslant i \leqslant n, \tag{2.20}$$

$$\tilde{c} = c, \qquad \tilde{d} = d. \tag{2.21}$$

This completes the calculation of the transformed equation. Note that in nonlinear problems, where the coefficients of L depend on f, the same formulas apply.

We now turn to the problem of computing the coordinate transformation. The method that we use for this computation starts by assuming that

$$\xi = \xi(\mathbf{x}) \tag{2.22}$$

and that the components of ξ satisfy some elliptic equation in the $\mathbf{x}$ variables,

$$L\xi_k = P_k, \qquad 1 \leqslant k \leqslant n, \tag{2.23}$$

where P_k is given and, as before, L will have the form

$$L = \sum a_{ij} \frac{\partial^2}{\partial x_i \, \partial x_j} + \sum b_i \frac{\partial}{\partial x_i} + c. \tag{2.24}$$

Here a_{ij}, b_i, c and P_k may depend on x and ξ. We start with a general equation so that we will have the formulas available for studying hyperbolic grids. Soon we will restrict L to be the Laplacian. The requirement that the boundary of the physical region be mapped one to one and onto the boundary of the transformed region generates boundary conditions that, along with the differential equations (2.23), determine the mapping $\xi = \xi(\mathbf{x})$.

The differential equation for the ξ_i can be transformed to the ξ frame using the, as of yet, undetermined transformation. Because the transformation has not yet been determined, the resulting transformed equations will be nonlinear. From (2.6) we have

$$\frac{\partial \xi_k}{\partial x_i} = \frac{K_{ik}}{\Delta} \tag{2.25}$$

and from (2.9) we have

$$\frac{\partial^2 \xi_k}{\partial x_i \partial x_j} = -\frac{1}{\Delta^3} \sum_{ruv} K_{ir} K_{ju} K_{vk} \frac{\partial^2 x_v}{\partial \xi_r \partial \xi_u}. \tag{2.26}$$

To transform the differential equations for ξ_k to a differential equation for x_k we apply (2.25) and (2.26) with the definition

$$\alpha_{ru} = \sum_{ij} a_{ij} K_{ir} K_{ju} \tag{2.27}$$

to Eq. (2.23) to obtain

$$\tilde{L}\tilde{\xi}_k = -\frac{1}{\Delta^3} \sum_{ruv} \alpha_{ru} K_{vk} \frac{\partial^2 x_v}{\partial \xi_r \partial \xi_u} + \frac{1}{\Delta} \sum_u b_u K_{uk} + c\xi_k. \tag{2.28}$$

Now multiply the equations $\tilde{L}\xi_k = P_k$ by J to obtain

$$-\frac{1}{\Delta^2} \sum_{ru} \alpha_{ru} \frac{\partial^2 x_i}{\partial \xi_u \partial \xi_r} + b_i + c \sum_k J_{ki} \xi_k = \sum_k J_{ki} P_k, \qquad 1 \leqslant i \leqslant n. \tag{2.29}$$

In our applications we choose $c = b_i = 0$ and thus have to solve

$$\sum_{ru} \alpha_{ru} \frac{\partial^2 x_j}{\partial \xi_r \partial \xi_u} + \Delta^2 \sum_k J_{ki} P_k = 0, \tag{2.30}$$

$1 \leqslant n$, for the functions $x(\xi)$.

In our applications we assume that the operator L is a Laplacian and this simplifies the formulas even further. Thus if J is replaced by its definition, then

$$\alpha_{ru} = \sum_i K_{ir} K_{ju}, \qquad 1 \leqslant r \leqslant u \leqslant n, \tag{2.31}$$

$$\alpha_{ru} = 0, \qquad 1 \leqslant u < r \leqslant n, \tag{2.32}$$

$$\sum_{ru} \alpha_{ru} (2 - \delta_{ru}) \frac{\partial^2 x_i}{\partial \xi_r \partial \xi_u} + \Delta^2 \sum_r P_k \frac{\partial x_i}{\partial \xi_k} = 0, \qquad 1 \leqslant i \leqslant n. \tag{2.33}$$

Remark. The α_{ru} and Δ depend on the first derivatives of $\mathbf{x}(\xi)$ so that the above equations are quasi-linear and coupled. Also note that if the transformation is

orthogonal in the inner product determined by the matrix (a_{ij}), then the matrix (α_{ij}) is the identity and then the equations become uncoupled Laplacians.

Another situation that arises frequently is that the derivatives of some parameters will appear in the coefficients of the differential equation. In particular, if the original equation is in divergence form, say

$$\sum_{ij} \frac{\partial}{\partial x_i} \sigma_{ij} \frac{\partial}{\partial x_j} f = 0, \qquad \sigma_{ij} = \sigma_{ij}(\mathbf{x}), \tag{2.34}$$

then we rewrite the equation as

$$\sum_{ij} \sigma_{ij} \frac{\partial^2}{\partial x_i \partial x_j} f + \sum_j \left(\sum_i \frac{\partial}{\partial x_i} \sigma_{ij} \right) \frac{\partial}{\partial x_j} f = 0. \tag{2.35}$$

Now our previous formulas can be used to change the derivatives of σ_{ij} to the new coordinates. The case that we have used is where σ_{ij} is a scaler multiple of the identity matrix, that is, the equation under consideration is

$$\sum_i \frac{\partial}{\partial x_i} \sigma \frac{\partial}{\partial x_j} f = 0. \tag{2.36}$$

3. Difference Equations

After the coordinate transformation has been made, the resulting differential equation must be converted to finite difference form. Here we replace f by g. In our application we are using the simplest centered differences. Thus we convert derivatives to differences using

$$\frac{\partial g}{\partial \xi_1}(\xi_1, \xi_2, \xi_3) \to \frac{g(\xi_1 + \Delta\xi_1, \xi_2, \xi_3) - g(\xi_1 - \Delta\xi_1, \xi_2, \xi_3)}{2\Delta\xi_1}, \tag{3.1}$$

$$\frac{\partial^2 g}{\partial \xi_1^2}(\xi_1, \xi_2, \xi_3) \to \frac{g(\xi_1 + \Delta\xi_1, \xi_2, \xi_3) - 2g(\xi_1, \xi_2, \xi_3) + g(\xi_1 - \Delta\xi_1, \xi_2, \xi_3)}{\Delta\xi_1^2}, \tag{3.2}$$

and so forth. The mixed partial derivatives are obtained by combining the first differences. All of the formulas that we use are given in the function "difference" in the VAXIMA listing in Section 4. This procedure will convert the differential equation to difference equation of the form

$$\sum_{|i|+|j|+|k|\leq 3} c_{i,j,k}(\xi_1, \xi_2, \xi_3)\, g(\xi_1 + i\,\Delta\xi_1, \xi_2 + j\,\Delta\xi_2, \xi_3 + k\,\Delta\xi_3) = R(\xi_1, \xi_2, \xi_3). \tag{3.3}$$

Here i, j, k each run over the set of values $\{-1, 0, 1\}$. The coefficients $c_{i,j,k}$ are called a stencil. For convenience, we view the elements of the stencil as points on cube on three space and then label the elements as in Fig. 3. Thus

$$c1 = c_{-1,1,0}, \qquad c2 = c_{0,1,0}, \qquad c3 = c_{1,1,0},$$
$$c4 = c_{-1,0,0}, \qquad c5 = c_{0,0,0}, \qquad c6 = c_{1,0,0},$$
$$c7 = c_{-1,-1,0}, \qquad c8 = c_{0,-1,0}, \qquad c9 = c_{1,-1,0},$$

while the cf and cb elements are equal to some c coefficient with the same first two subscripts, as indicated in the above formulas, and the third subscript is $+1$ for cf and -1 for cb.

Because of the simplicity of the finite differences used to represent derivatives, the stencil has several symmetries. First note that the stencil contains 27 elements. We use the value of $|i| + |j| + |k|$ to distinguish four cases. Again, we view this stencil as points on a cube in three space, and note that 8 corner elements have value zero,

$$|i| + |j| + |k| = 3 \Rightarrow c_{i,j,k} = 0. \tag{3.4}$$

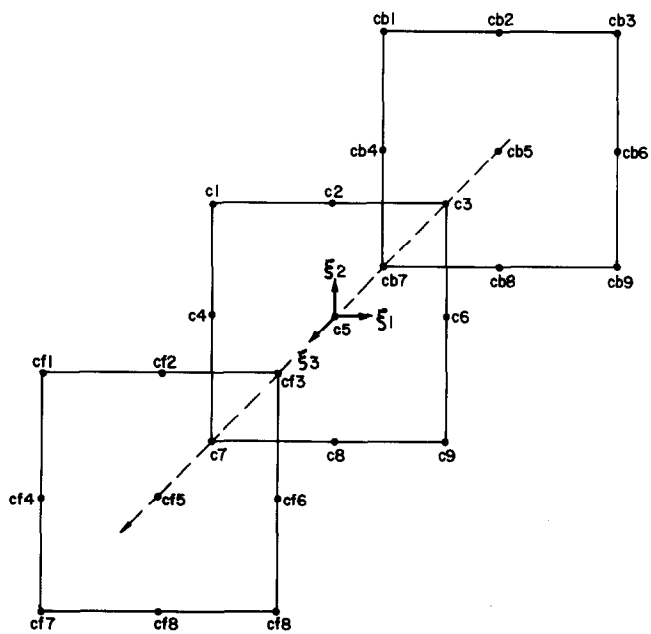

FIGURE 3

Next, the 12 elements at the center of an edge have an odd symmetry:

$$|i|+|j|+|k|=2 \Rightarrow \begin{cases} c_{-i,j,k} = (-1)^i c_{i,j,k} \\ c_{i,-j,k} = (-1)^j c_{i,j,k}. \\ c_{i,j,-k} = (-1)^k c_{i,j,k} \end{cases} \tag{3.5}$$

The 6 elements at a center of a face are a sum of two terms,

$$c_{i,j,k} = a_{i,j,k} + b_{i,j,k}, \tag{3.6}$$

when $a_{i,j,k}$ and $b_{i,j,k}$ have, respectively, even and odd symmetries:

$$|i|+|j|+|k|=1 \Rightarrow \begin{cases} a_{-i,j,k} = a_{i,j,k} \\ a_{i,-j,k} = a_{i,j,k} \\ a_{i,j,-k} = a_{i,j,k} \end{cases} \tag{3.7}$$

$$|i|+|j|+|k|=1 \Rightarrow \begin{cases} b_{-i,j,k} = (-1)^i b_{i,j,k} \\ b_{i,-j,k} = (-1)^j b_{i,j,k}. \\ b_{i,j,-k} = (-1)^k b_{i,j,k} \end{cases} \tag{3.8}$$

Of course there is the one remaining center element where $|i|+|j|+|k|=0$ which is the sum of three terms.

These symmetries reduce the computation of the 27 stencil elements to the computation of 10 stencil elements grouped as follows:

Case	Number
$\|i\|+\|j\|+\|k\|=3$	0
$\|i\|+\|j\|+\|k\|=2$	3
$\|i\|+\|j\|+\|k\|=1$	6
$\|i\|+\|j\|+\|k\|=0$	1

Using the non-subscripted notation we have identities

$$\begin{aligned} cf1 = cf3 = cf7 = cf9 = 0, \\ cb1 = cb2 = cb7 = cb9 = 0, \\ c1 = -c3 = -c7 = c9, \\ cf2 = -cb2 = -cf8 = cb8, \\ cf4 = -cb4 = -cf6 = cf6. \end{aligned} \tag{3.9}$$

If we set $h_i = \Delta\xi_i$, then the elements of the stencil are given by sums of multiples of coefficients of various derivatives in the differential equation as indicated in the following table.

Stencil	Factor	Derivative	Factor	Derivative
$c1$	$\dfrac{-2}{4h_1h_2}$	$\dfrac{\partial^2 g}{\partial\xi_1\,\partial\xi_2}$		
$cf2$	$\dfrac{2}{4h_2h_3}$	$\dfrac{\partial_2 g}{\partial\xi_2\,\partial\xi_3}$		
$cb4$	$\dfrac{2}{4h_1h_3}$	$\dfrac{\partial^2 g}{\partial\xi_1\,\partial\xi_3}$		
$c4$	$\dfrac{1}{h_1^2}$	$\dfrac{\partial^2 g}{\partial\xi_1^2}$	$\dfrac{-1}{2h_1}$	$\dfrac{\partial g}{\partial\xi_1}$
$c6$	$\dfrac{1}{h_1^2}$	$\dfrac{\partial^2 g}{\partial\xi_1^2}$	$\dfrac{1}{2h_1}$	$\dfrac{\partial g}{\partial\xi_1}$
$c2$	$\dfrac{1}{h_2^2}$	$\dfrac{\partial^2 g}{\partial\xi_2^2}$	$\dfrac{-1}{2h_2}$	$\dfrac{\partial g}{\partial\xi_2}$
$c8$	$\dfrac{1}{h_2^2}$	$\dfrac{\partial^2 g}{\partial\xi_2^2}$	$\dfrac{-1}{2h_2}$	$\dfrac{\partial g}{\partial\xi_2}$
$cf5$	$\dfrac{1}{h_3^2}$	$\dfrac{\partial^2 g}{\partial\xi_3^2}$	$\dfrac{1}{h_3}$	$\dfrac{\partial g}{\partial\xi_3}$
$cb6$	$\dfrac{1}{h_3^2}$	$\dfrac{\partial^2 g}{\partial\xi_3^2}$	$\dfrac{-1}{h_3}$	$\dfrac{\partial g}{\partial\xi_3}$
$c5$	$\dfrac{-2}{h_i^2}$	$\dfrac{\partial^2 g}{\partial\xi_i^2}$	1	g

Ideas similar to the above apply to the calculation of the coordinate transformation. The differential equation under consideration is then

$$\sum_{ru} \alpha_{ru} \frac{\partial^2 x_i}{\partial\xi_r\,\partial\xi_u} + \Delta^2 \sum_k P_k \frac{\partial x_i}{\partial\xi_k} = 0. \tag{3.10}$$

Using the previous table we can now easily write down the formulas for the stencil defining this differential equation. This last formula is given in the last section of Mastin and Thompson [13].

4. The Symbol Code

A partial listing of the VAXIMA code used to write the subroutines SETM3 and SETGM3 is given at the end of this section. A partial listing of the subroutine SETM3 is given in the next section. We have not provided the full listings (at the suggestions of the editors) of the symbol and FORTRAN code because these listing are lengthy and redundant. We will provide complete listings with the reprints. Every effort was made to keep the symbol code straightforward. In particular, the computation of SETM3 and SETGM3 were kept uncoupled (and consequently redundant) to facilitate the use of the symbol code for new problems.

Because the VAXIMA code closely follows the mathematical development of the previous section, we hope that the reader may follow the program using the MACSYMA manual [12]. As we indicate in the comments in the symbol program, the problem is broken up into several pieces represented by VAXIMA function calls. Each function accumulates, in lists, some of the formulas needed to write the FORTRAN code. At the end these lists are processed to produce the subroutines.

Before we turn to the listing we would like to mention a few points. To avoid conflicts between indices in the VAXIMA code and the FORTRAN code we have used double characters, *nn*, *ii*, *jj*,..., for integer variables in VAXIMA and reserved single characters, *n*, *i*, *j*,..., for use in the FORTRAN code. In many places where one would expect array elements, say $a(1)$, one finds instead function call of the form

$$concat(a, 1). \tag{4.1}$$

This call produces an atomic variable of the form $a1$ rather than the array reference $a(1)$. This allows us to substantially reduce the number of array references than might otherwise appear in the FORTRAN code.

Although we are mainly interested in the three dimensional case, the VAXIMA program does both the two and three dimensional case. This greatly facilitated checking the VAXIMA programs because the two dimensional case can be done by hand.

The original VAXIMA code was quite different from the version given here. The approach of the first version was to implement the chain rule and then let the chain rule do its work. This produced large expressions that were difficult to handle in VAXIMA. A more detailed description of the problems encountered using this technique is given in Section 7 of this paper.

4.1. *VAXIMA Listings*

The following comments occur at the beginning of the VAXIMA listings.

These programs start with a differential equation, transform the differential equation to general coordinates, introduce finite differences and notation appropriate for FORTRAN and then write two FORTRAN subroutines called *SETM2* and *SETGM2* or two subroutines called *SETM3* and *SETGM3*. These

subroutines can be used with elliptic equation solvers and coordinate generating codes to solve elliptic boundary value problems in irregular regions.

The functions in this file are called:

special( )	special 2 or 3 dimensional case
general( )	general 2 or 3 dimensional case
change( )	change variables
notate( )	atomic notation for derivatives
notation(exp,vari)	primitive atomic notation
scheme( )	introduce differences of unknown
difference(u,f,exp)	primitive differences
myFORTRAN( )	write the FORTRAN code

The functions *special* and *general* are test drivers. The special test driver transforms a Laplacian with variable diffusivity given in rectangular coordinates in 2 or 3 dimensions while the general test driver transforms a general elliptic equation in 2 or 3 dimensions. Both drivers return FORTRAN subroutines suitable for use with elliptic equation solvers.

The program *change* converts an arbitrary second order differential equation in *nn* variables to an arbitrary coordinate frame in the variables *xi[i]*. This program also computes some formulas used to transform the Laplacian to the *xi* variables.

The programs *notate* and *notation* introduce atomic variable notation for the derivatives of the coordinate change.

The programs *scheme* and *difference* introduce the differences of the unknown functions and then collect the coefficients of these differences which are used to calculate the stencil used to drive the elliptic equation solver. These programs also calculate the stencil for the code that calculates the coordinate transformations.

The program *myFORTRAN* takes the information computed by the other programs and writes the FORTRAN subroutines *SETMN* and *SETGMN* into files called *SETM3* and *SETGM3*. It also writes a file called *formulas* that contains algebraic forms of all of the formulas used in the subroutines in a form convenient for human reading.

Here are the variables used to store the information needed to write the FORTRAN subroutine SETM:

real	variables used by the subroutine
differ	difference equations
inter	Jacobian and other intermediate expressions
stencil	formulas for the stencil
calling	notation for the coefficients

In the VAXIMA code we use the variables:

nn	the spatial dimension
x,y,z	the independent variables
ind	a list of the independent variables
xi[i]	the new independent variables
f	the dependent variable
g	the new dependent variable
eqn	the differential equation
neweqn	the transformed differential equation
sigma	the variable diffusivity
s	sigma in the new variables
aij,bi,c,d	coefficients in the differential equation
dep	list of dependent variables

Now we turn to listing some of the VAXIMA functions. Variables with a name of the form *name*_g are used to write *SETGMN* rather than *SETMN*.

4.1.1. *special*

```
special(n) := block(
/* The special 2 dimensional case of Laplacian with variable
   diffusivity. */
/* Declare some flags and variables. */
   showtime:all, /* display computation times */
   gcprint:false, /* don't display garbage collections */
   dep:[f,sigma], /* the dependent variables */
   ind:[x,y,z], /* the dependent variables */
   depends(dep,ind), /* declare the dependencies */
/* Here is the Laplacian with variables diffusivity sigma. */
   eqn:sum(diff(sigma*diff(f,ind[i]),ind[i]),i,1,nn),
/* Call the functions that write the subroutines. */
   change( ),
   notate( ),
   scheme( ),
   myFORTRAN( ),
   end)$
```

4.1.2. *general*

This function has been omitted but is much like the previous function.

4.1.3. *change*

```
/* The next program changes a given differential equation to an  */
/* arbitrary coordinate frame. The notation vj is used to denote */
/* values related to the Jacobian while the notation vk is used to */
/* denote values related to the inverse of the Jacobian. The arrays */
/* tt, aa, bb are used as temporary storage. */
change( ):= block([new,temp1,temp2,jacmat,det],
  kill(labels), /* clear some memory */
/* Construct a list of the new variables. */
  new:[ ], /* initialize a list for the new independent variables */
  for ii thru nn do new:endcons(xi[ii],new),
  depends(ind,new), /* declare the dependencies */
/* Define the Jacobian and related quantities. */
  tt[ii,jj]:= diff(ind[ii],xi[jj]), /* tt is a VAXIMA array */
  jacmat:transpose(genmatrix(tt,nn,nn)), /* the Jacobian matrix */
  remarray(tt), /* save memory */
  det:expand(determinant(jacmat)), /* the Jacobian */
  inter:[vj = det], /* start a list of formulas for SETM */
  inter_g:inter, /* same for SETGM */
  inter:endcons(vk = 1/vj,inter), /* reciprocal of the Jacobian */
  inter_g:endcons(vj2 = vj^2,inter_g), /* powers of above */
  inter:endcons(vk2 = vk/vj,inter),
  inter:endcons(vk3 = vk2/vj,inter),
  real:[vj,vk,vk2,vk3], /* start a list of variables for SETM */
  real_g:[vj,vj2], /* start a list of variables for SETGM */
/* Collect the coefficients of the given differential equation. */
  neweqn:ev(eqn,diff,expand), /* put the PDE in expanded form */
/* Collect the coefficients of the second derivatives. */
  for ii thru nn do for jj:ii thru nn do (
    temp1:bothcoeff(neweqn,diff(f,ind[ii],1,ind[jj],1)),
    aa[ii,jj]:first(temp1),
    neweqn:last(temp1),
    end),
/* Collect the coefficients of the first derivatives. */
  for ii thru nn do (
    temp1:bothcoeff(neweqn,diff(f,ind[ii])),
    bb[ii]:first(temp1),
    neweqn:last(temp1),
    end),
```

```
/* Transform the derivatives of the parameter sigma appearing in the */
/* coefficients of the differential equation. This uses the formulas */
/* derived in the above mentioned paper. s is the name for sigma */
/* in the new variables. */
  depends(s,new), /* more dependencies */
  for ii thru nn do for jj thru nn do
    bb[ii]:subst(
      sum(
      concat(vk,jj,kk)*diff(s,xi[kk]*vk
      ,kk,1,nn),
      diff(sigma,ind[jj]),bb[ii]),
  for ii thru nn do for jj:ii thru nn do
    aa[ii,jj]:subst(s,sigma,aa[ii,jj]),
/* Calculate the intermediate expressions that determine the transformed */
/* differential equation. For a derivation of these formulas see the */
/* above mentioned paper. Here we are simply recording those formulas. */
/* Compute the entries the cofactor matrix of the Jacobian. */
  for ii thru nn do for jj thru nn do (
    temp1:concat(vk,ii,jj),
    temp2:(-1)^(ii+jj)*determinant(minor(jacmat,jj,ii)),
    temp2:expand(temp2),
    if nn=2 then (temp1::temp2) else (
    real:endcons(temp1,real),
    real_g:endcons(temp1,real_g),
    inter:endcons(temp1=temp2,inter),
    inter_g:endcons(temp1=temp2,inter_g),
    end),
  end),
/* Record the formula for S. */
  for ii thru nn do for jj:ii thru nn do (
    temp1:concat(S,ii,jj),
    real:endcons(temp1,real),
    temp2:sum(sum(
      aa[kk,ll]*
      concat(vk,kk,ii)*
      concat(vk,ll,jj),
      ll,kk,nn),kk,1,nn),
    temp2:factorsum(temp2),
  inter:endcons(temp1=temp2,inter),
  end),
  for ii thru nn do for jj:1 thru ii-1 do (
    concat(S,ii,jj)::concat(S,jj,ii),
  end),
```

```
/* Record the formulas for U. */
  for ii thru nn do (
    temp1:concat(U,ii),
    real:endcons(temp1,real),
    temp2:sum(sum(sum(
      concat(S,jj,kk)*
      concat(vk,ll,ii)*
      diff(ind[ll],xi[jj],1,xi[kk],1)
      ,jj,1,nn),kk,1,nn),ll,1,nn),
    inter:endcons(temp1 = temp2,inter),
  end),

/* Record the formulas for T. */
  for ii thru nn do (
    temp1:concat(T,ii),
    real:endcons(temp1,real),
    temp2:sum(bb[jj]*concat(vk,jj,ii),jj,1,nn),
    temp2:factorsum(temp2),
    inter:endcons(temp1 = temp2,inter),
  end),
  remarray(aa,bb), /* clear some memory */
  inter:ev(inter), /* evaluate the temporary variables */

/* Record the formulas for SETGMN. */

/* Record the formulas for alpha. */
  for ii thru nn do for jj:ii thru nn do (
    temp1:concat(alp,ii,jj),
    temp2:sum(
      concat(vk,kk,ii)*concat(vk,kk,jj)
      ,kk,1,nn),
    real_g:endcons(temp1,real_g),
    inter_g:endcons(temp1 = temp2,inter_g),
  end),
  for ii thru nn do for jj:1 thru ii - 1 do (
    concat(alp,ii,jj)::concat(alp,jj,ii),
  end),
  inter_g:ev(inter_g),

/* Introduce the hosted differential equation in the new coordinates. */

  depends(g,[xi[1],xi[2],xi[3]]),
  neweqn:subst(g,f,neweqn),
  neweqn:neweqn +
    sum(sum(concat(S,ii,jj)*diff(g,xi[ii],1,xi[jj],1)*vk2,
      ii,jj,nn),jj,1,nn)
```

```
    +sum((-concat(U,ii)*vk3+concat(T,ii)*vk)*diff(g,xi[ii]),
      ii,1,nn),
  neweqn:ev(neweqn,expand),
/* Introduce the differential equation that determines the coordinate */
/* transformation. */
  neweqn_g:sum(sum(
    concat(alp,ii,jj)*diff(g,xi[ii],1,xi[jj],1)
    ,ii,1,nn),jj,1,nn)
    +vj2*sum(concat(vp,ii)*diff(g,xi[ii],ii,1,nn),
  neweqn_g:ev(neweqn_g,expand),
  end)$
```

4.1.4. *notate*

This functions applies the next function to various lists.

4.1.5. *notation*

```
notation(exp,vari):=block(
/* Make the substitution of atomic variables for derivatives. */
  for ii thru nn do for jj thru nn do
  exp:subst(concat(vari,ii,jj),diff(vari,xi[ii],1,xi[jj],1),exp),
  for ii thru nn do exp:subst(concat(vari,ii),diff(vari,xi[ii]),exp),
  exp)$
```

4.1.6. *scheme*

```
/* The next program sets up the calculation of the derivatives of the */
/* coordinate change and any parameters in the coefficients of the */
/* the list differ. */
scheme():=block([temp1,temp2,temp3,args],
/* Add the step sizes to the real lists */
  for ii thru nn do real:endcons(concat(h,ii),real),
  for ii thru nn do real_g:endcons(concat(h,ii),real_g),
/* The array arguments depend on the dimension. */
  if nn=2 then args:"(i,j)" else args:"(i,j,k)",
  temp1:concat(s,args),
  differ:[s0=temp1],
  differ_g:[],
/* Write the formulas for the derivatives of s. */
  for ii thru nn do (
    temp1:concat(s,ii),
    temp2:difference(s,s,diff(s,x[ii])),
    differ:endcons(temp1=temp2,differ),
  end),
```

```
/* Write the formulas for calculating p, the forcing terms. */
  for ii thru nn do (
    temp1:concat(vp,ii),
    temp2:concat(p,ii,args),
    differ_g:endcons(temp1 = temp2,differ_g),
  end),

/* Write the formulas for the first derivatives of the independent */
/* variables. */
  for ii thru n do for kk thru nn do (
    temp1:concat(ind[kk],ii),
    temp2:difference(ind[kk],ind[kk],diff(ind[kk],xi[ii])),
    differ:endcons(temp1 = temp2,differ),
    differ_g:endcons(temp1 = temp2,differ_g),
  end),

/* Write the formulas for the second derivatives of the independent */
/* variables. */
  for ii thru nn do for jj thru ii do for kk thru nn do (
    temp1:concat(ind[kk],jj,ii),
    temp2:difference(
      ind[kk],ind[kk],diff(ind[kk],xi[ii],1,xi[jj],1)),
    differ:endcons(temp1 = temp2,differ),
  end),

/* Introduce the formulas for calculating the stencil for SETMN. */
/* See the above mentioned paper for a derivation of the formulas. */
  stencil:[ ],calling:[ ], /* initialize the lists */

/* Now create the formulas. */
  calling:endcons("c1",calling),
  temp1:concat("c1",args),
  temp2:-coeff(neweqn,diff(g,xi[1],1,xi[2],1))/2*h1*h2),
  stencil:endcons(temp1 = temp2,stencil),

  calling:endcons("c2",calling),
  temp1:concat("c2",args),
  temp2:coeff(neweqn,diff(g,xi[2],2))/(h2^2),
  temp3:coeff(neweqn,diff(g,xi[2],1))/(2*h2),
  stencil:endcons(temp1 = temp2 + temp3,stencil),

  calling:endcons("c8",calling),
  temp1:concat("c8",args),
  stencil:endcons(temp1 = temp2 - temp3,stencil),
```

Now omit the calculation of the remaining stencil elements in SETNM and all of the stencil calculations in SETGMN. Finish up *scheme*.

```
/* Try to improve the form of the formulas in stencil_g. */
   stencil_g:factorsum(stencil_g),
   end)$
```

4.1.7. *difference*

This function substitutes standard difference formulas for derivatives and is consequently omitted.

4.1.8. *myFORTRAN*

This functions uses the previously computed formulas to write the FORTRAN subroutines. The steps in this function essentially parallel those that would be used by a human writing the same subroutine so we omit this function. The reader should refer to the *fortran* function that is described in the MACSYMA manual.

5. The Fortran Code

Portions of the FORTRAN subroutine SETM3 written by VAXIMA are listed at the end of this section. As the reader will notice, the subroutine is a bit different from what a human would write. We would now like to explain these differences and discuss their advantages and disadvantages.

One of the first things to notice is that there is considerable redundant arithmetic in the formulas that compute the stencil elements. We attempted to use the VAXIMA functions "optimize" and "factorsum" to improve the form of these expressions. Unfortunately, the expression for $u1$ is left unchanged by each of these VAXIMA functions. It is certainly possible to write programs to reorganize the formulas in the FORTRAN subroutines for the particular case we studied here. This is also easily done with a text editor. We decided that a general solution of this problem is beyond the scope of the present paper but we hope to work on this problem in the near future. We note that it will be important to make the solution of this part of the problem compatible with the optimizers in various FORTRAN compilers.

Also, we made no attempt to make the formulas in the FORTRAN code readable. Instead, as an independent part of the VAXIMA code, we write all of the formulas in VAXIMAs planar format that is designed for human reading.

It is worth nothing that, in our applications, the computations done by our subroutines represent a small percentage of the total computations so that optimizing the operation count in the subroutines is not critical. It is also easy to set up a test case for the subroutines, use an editor to modify the subroutines and then compare the results to the original subroutine to guard against errors. Also, in some applications, it may be desirable to have some part of the code compute the arrays of derivatives of the coordinate functions and then pass these arrays to any parts of the program. Again, these are easy changes to make in the subroutines.

If one looks at the mathematical formulation of the change of variables given in

Section 2, then it seems natural to introduce arrays to represent T_i, U_i and S_{ij}. In our programs such arrays are not introduced and instead atomic variables of the form

$$T1,\ T2,\ T3,\ U1,\ U2,\ U3,\ S11,\ S12,\ldots,\ S33.$$

are introduced. This represents a substantial savings in the run time of the program because of the elimination of the need to look up array elements. A similar procedure is used to represent the derivatives of the coordinate functions.

Here is a listing of the FORTRAN subroutine written by our VAXIMA program.

5.1. *FORTRAN Listing for SETM3*

The header and comments will help in understanding the purpose of the subroutine:

```
      SUBROUTINE SETM3(IL,JL,KL,ILD,JLD,S,X,Y,Z,H1,H2,H3,C1,C2,C8,C4,
     C6
     1   ,C5,CF2,CB4,CF5,CB5)
C
C     ILD,JLD,KLD ARE THE DIMENSIONS OF ARRAYS IN THE CALLING
C     PROGRAM.
C     IL,JL,KL GIVE THE SIZE OF THE PROBLEM.
C     S IS THE INPUT ARRAY OF DENSITIES.
C     H1,H2,H3 ARE INPUT DIFFERENCES.
C     X,Y,Z ARE THE INPUT ARRAYS OF COORDINATES.
C
C     C1,C2,...,CF1,CF2,...,CB1,CB2 REFER TO THE ELEMENTS OF THE
C     STENCIL THAT ARE RETURNED BY THIS SUBROUTINE. THESE
C     ELEMENTS SATISFY CERTAIN IDENTITIES SO THAT WE ONLY
C     COMPUTE A SUBSET CONSISTING OF 10 OF THE ELEMENTS. THE
C     FORMULAS FOR THE REMAINING ELEMENTS ARE:
C        CF1 = CF3 = CF7 = CF9 = 0
C        CB1 = CB2 = CB7 = CB9 = 0
C        C1 = -C3 = -CF8 = CB8
C        CF4 = -CB4 = -CF6 = CB6
```

The variable declarations, loop counters, and some preliminary computations are now omitted. Here are some of the finite difference computations:

```
      X3 = (X(I,J,K+1)-X(I,J,K-1))/H3/2.0
      Y3 = (Y(I,J,K+1)-Y(I,J,K-1))/H3/2.0
      Z3 = (Z(I,J,K+1)-Z(I,J,K-1))/H3/2.0
      X11 = (X(I+1,J,K)-2*X(I,J,K)+X(I-1,J,K))/H1**2
```

```
      X12 = (X(I+1,J+1,K)-X(I+1,J-1,K)-X(I-1,J+1,K)+
                                          X(I-1,J-1,K))/(H1*H
     1  2)/4.0
```

Now skip to the evaluation of the Jacobian and its minors:

```
      VJ = X1*Y2*Z3-X2*Y1*Z3-X1*Y3*Z2+X3*Y1*Z2+X2*Y3*Z1-
                                                X3*Y2*Z1
      VK = 1/VJ
      VK2 = VK/VJ
      VK3 = VK2/VJ
      VK11 = Y2*Z3-Y3*Z2
      VK12 = Y3*Z1-Y1*Z3
```

Now skip to some of the calculations of some of intermediate quantities:

```
      S33 = SO*(VK33**2+VK23**2+VK13**2)
      U1 = S33*VK31*Z33+2*S23*VK31*Z23+S22*VK31*Z22+2*S13*VK31*
                                                Z13+2*S1
     1  2*VK31*Z12+S11*VK31*Z11+S33*VK21*Y33+2*S23*VK21*Y23+S22*
                                                VK21*Y2
     2  2+2*S13*VK21*Y13+2*S12*VK21*Y12+S11*VK21*Y11+S33*VK11*
                                                X33+2*S23
     3  *VK11*X23+S22*VK11*X22+2*S13*VK11*X13+2*S12*VK11*
                                                X12+S11*VK11*X
     4  11
```

Now skip some of the intermediate calculations and go to a point just before the arrays for the stencils are calculated:

```
      T3 = VK*(S3*VK33**2+S2*VK33+S1*VK31*VK33+S3*VK23**2+S2*
                                                VK22*
     1  VK23+S1*VK21*VK23+S3*VK13**2+S2*VK12*VK13+S1*VK11*
                                                VK13)
      C1(I,J,K) = -S12*VK2/(H1*H2)/2.0
      C2(I,J,K) = -(H2*U2*VK3-2*S22*VK2-H2*T2*VK)/H2**2/2.0
```

The final part of the code fills the remaining stencil elements and then closes the loops.

6. Numerical Validation

The potential for errors in either the problem formulation or the encoding procedure always exists in complex codes such as three dimensional boundary fitted coordinate codes. The need for validation was emphasized in the present work

because symbolic manipulation was used; the resulting "psychological distance" from the work made it less likely to be satisfied with superficial plausibility exercises based on intuitive ideas of acceptable levels of absolute error.

In the present work, we validate the codes by performing numerical tests. The approach is to choose a continuum solution for the class of problems treated by the code, and to validate the convergence of the results to this continuum solution by systematic truncation error convergence testing over a sequence of grid sizes. This procedure, though well known in theory, is seldom followed in practice.

6.1. *Hosted Equation Convergence Testing*

To test the equation $L(F)=Q$, we test a modified equation whose solution is chosen to be

$$Fs=(x_1^{m+1}x_2^{m+2}x_3^{m+3})/Fn \qquad on \quad R,$$

where

$$m=nl+na-1$$

and $nl=$ *degree of* L (e.g., $nl=2$ for $L=$ Laplacian) and $na=$ order of accuracy of the finite-difference expression (e.g., $na=2$ for centered differences). Fn is the normalizing value of Fs. The motivation in the selected form of the solution is of course to insure that all the derivative terms in L are exercised, and that there is non-zero truncation error for finite h_i, even without the transformation of coordinates. (For example, a parabolic solution will show no truncation error using second-order accurate solutions with the identity transformation. Several published "validations" of the accuracy of upwind differencing are inadequate because the chosen solution structure does not exercise the meaningful terms in the truncation error, giving a false indication of accuracy.)

The modified problem to be tested is then

$$L(F)=Qs \qquad on \quad R,$$

where

$$Qs=Q+L(Fs).$$

The boundary equations for the modified problem are

Qs can be obtained in x by elementary operations as

$$Qs=Q+Fs[m(m+1)/x_1^2+(m+1)(m+2)/x_2^2+(m+2)(m+3)/x_3^2].$$

The solution is obtained in a stretched coordinate system $\xi=(\xi_1, \xi_2, \xi_3)$, where the ξ_i values are linear from 0 to 1, $\xi_1=h1*(i-1)$, etc. The chosen coordinate transformation is given by

$$x_i=\xi s+\xi_i+\tanh(d_i\xi_1\xi_2\xi_3),$$

ξs is a shift, necessary to avoid a singularity in the chosen solution at the origin. The parameters d_i control the severity of the coordinate stretching; for $d_i = 0$, there is no stretch in x_i direction. The form of the coordinate transformation is chosen to exercise all terms in the transformed equations, notably the cross-derivative terms, which would be zero if the coordinate transformations were independent in each direction. For non-zero d_i, the tanh term in the transformation assures non-zero values for all derivatives.

The above solution and transformation appear to generate non-trivial (non-zero and non-repeating or canceling) derivatives of all orders, including cross-derivatives.

The discrete solution *Fd* is then obtained on a family of grids, using any standard method. (We used the spatial marching methods in the GEM codes [16–20] for 2*D*, and hopscotch SOR in 3*D*.) It is important that iterative convergence criteria be stringent, so as not to confuse the incomplete iteration (residual) error with the truncation error $te = Fs - Fd$. We required iterative convergence to essentially the single precision accuracy of the 32-bit VAX computer used, about 7–8 decimal figures. We also forced a minimum number of iterations to be performed, somewhat greater than the maximum grid index, to be sure that a false indicator of iterative convergence was not obtained due to good initial conditions available at the fine grid spacing.

The truncation error convergence is then monitored as the grid is systematically refined. Theoretically, the local values of $p = te/h^2$ should become constant as the grid size is refined. For a uniformly *na*-th order finite difference discretization ("uniformly" implying at all points for all derivatives) the truncation error can be written as

$$te = C1\, h1^{na} + C2\, h2^{na} + C3 h3^{na} + h.o.t.$$

We define the index of predicted order of truncation error

$$p = N^{na}.$$

With $h1 = \xi_{1\max}/N = 1/N$, etc. we have

$$p = C1\, \xi_{1\max}^{na} + C2\, \xi_{1\max}^{na} + C3\, \xi_{1\max}^{na} + h.o.t.,$$

which is constant to order *na* approximation.

We then monitor p as N increases, using p at specified points (e.g., pcr at the center of the grid) and $pmx = \max_{i,j,k} p$. If all $p = constant$ as N increases, we have verified (1) the equation transformation, (2) the order of finite difference expressions, and (incidentally)(3) the solution procedure.

This procedure does not require a grid halving sequence, $N = 2^m$, although this sequence does make the judgment easier. Also, we note from experience, even on trivial 1D problems, that $p =$ "*constant*" really means as much as a 50% variation as the grid is halved. (The results presented herein are much more convincing, as will be seen.)

For a coarse-mesh solution, it would be advantageous to eliminate the iterative residual from the calculation of *te* by using a direct method, i.e., Guassian elimination. However, in 3D this will fail for large N by round-off error and/or CPU time demands; likewise for the marching methods. Iterative methods are dangerous in that their convergence depends on L, and on the coordinate transformation. Note, however, that it is not necessary, nor even advisable, to test the small-parameter (high Reynolds number) problem. Errors will show up more readily when all coefficients in L are of the same order. Once the above procedure has verified the algebra and coding accuracy, one can proceed with alternate solutions techniques for the small-parameter problem.

Note that this procedure can be followed for mixed-order finite difference equations (e.g., upwind differencing on first derivative terms) with the modification that we look only for convergence in the sense of zero error as N increases, to verify the coding algebra and coding accuracy. This modified procedure will not verify the order of accuracy of the mixed-order equations, and will probably require even finer grid solutions for first-order equations since the convergence of these is difficult to judge.

6.2. *3D Hosted Equation Convergence Results*

Table I presents the results of testing for the constant-coefficient Laplacian equation in 3D, using moderate stretching parameters $d_i = 0.1$. The mesh size was successively halved three times, from 5^3 to 33^3. The numerically calculated Jacobian for the 5^3 grid ranged from 1.0187 to 1.1684, and for the 33^3 grid ranged from 1.0003 to 1.2792. The table presents the maximum truncation error over the grid, *temax*, and the grid location at which it occurred; the truncation error at the center of the grid, *tectr*; and the corresponding values $pmx = temax/h^2$ and $pcr = tectr/h^2$.

The near-constancy of the coefficient $pmx = temax/h^2$ indicates that the entire solution, including the coordinate transformation and the finite difference expressions, is second order accurate. The erratic behavior of *pcr* at the finest grid indicates round-off error problems.

Table II presents results of testing for the variable-coefficient Laplacian equation in 3D, using both moderate stretching $d_i = 0.1$ and strong stretching $d_i = 10$. The hosted equation solved is

$$\nabla\sigma \cdot \nabla F = 0,$$

$$\sigma = \sigma_0 + \sigma_m \sin(b),$$

$$b = x_1^n x_2^n x_3^n,$$

$$x_{1n} = (x_1 - x_{1\min})/(x_{1\max} - x_{1\min}), \text{ etc.}$$

The code, of necessity, used the expanded (non-conservation form),

$$\sigma\nabla^2 F + \nabla\sigma \cdot \nabla F = 0.$$

TABLE I

3D Hosted Equation Convergence Testing for $\nabla^2 F = 0$.
The Stretching Parameters $d_i = 0.1$

grid	*temax*	at	*tectr*	*pmx*	*pcr*
5^3	.402e−03	4,4,4	.832e−05	.646e−02	.133e−03
9^3	.172e−03	8,8,8	.650e−06	.111e−01	.416e−04
17^3	.492e−04	15,15,15	.340e−07	.126e−01	.870e−05
33^3	.134e−04	29,29,29	−.169e−06	.137e−01	−.173e−03

(We have not yet developed symbolic manipulation codes with the ability to retain the unexpanded or conservation form of the equation.) The solution chosen was

$$Fs = x_1^{m+1} x_2^{m+2} x_3^{m+3}/Fn$$

giving the forcing term

$$\begin{aligned} Qs = [&\sigma m(m+1)\, x_1^{m-1} x_2^{m+2} x_3^{m+3} + \sigma(m+1)(m+2)\, x_1^{m+1} x_2^m x_3^{m+3} \\ &+ \sigma(m+2)(m+3)\, x_1^{m+1} x_2^{m+2} x_3^{m+1} + \sigma_x(m+1)\, x_1^m x_2^{m+2} x_3^{m+3} \\ &+ \sigma_y(m+2)\, x_1^{m+1} x_2^{m+1} x_3^{m+3} + \sigma_z(m+3)\, x_1^{m+1} x_2^{m+2} x_3^{m+2}]/Fn \end{aligned}$$

with

$$\sigma_x = \sigma_m \cos(b)\, x_2^n x_3^n/bd,$$

$$\sigma_y = \sigma_m \cos(b)\, x_1^n x_3^n/bd,$$

$$\sigma_z = \sigma_m \cos(b)\, x_1^n x_2^n/bd,$$

$$bd = (x_{1\max} - x_{1\min})(x_{2\max} - x_{2\min})(x_{3\max} - x_{3\min}).$$

Only the values of *pmx* are shown in Table II. The results show that the validation occurs even at the grid 17^3 for moderate stretching with $d_i = 0.1$, but that there is some variation in *pmx* even at the grid 33^3 for strong stretching with $d_i = 10$. The convergence is clear, however.

The behavior of the truncation error for the strong stretching is of interest. The values of *temax* for just the coarse grid are plotted in Table III for a range of stretching parameters d_i.

For this coarse grid, $d_i = 0.1$ minimizes the maximum truncation error. The table shows that large and inappropriate stretching values do indeed increase the truncation error by almost two orders of magnitude compared to the best transformation used. However, Table II shows that the results are still $O(h^2)$ accurate. By

TABLE II

3D Hosted Equation Convergence Testing for $\nabla \cdot \sigma \nabla F = 0$

	grid =	5^3	9^3	17^3	33^3
$d_i = 00.1$,	*pmx* =	.390e−02	.877e−02	1.00e−02	
$d_i = 10.0$,	*pmx* =	.433	.691	1.21	1.53

TABLE III

3 Hosted Equation Convergence Testing for $\nabla \cdot \sigma \nabla F = 0$ in the Coarse Grid $= 5^3$

$d_i =$	0.	0.1	0.5	1.0	10.0
$temax =$	.830e−03	.244e−03	.489e−02	.737e−02	.271e−01

definition, we mean that the method is second order accurate if a reduction in h by a factor of 1/2 will (asymptotically) reduce the truncation error by a factor of 1/4.

We also validated this retention of second-order accuracy for very strong $1d$ coordinate stretching. Using double precision calculations, we experimented with a range of stretching parameters for reasonable coordinate transformations based on hyperbolic tangent and exponential functions, and even unreasonable transformations based on exponentials of exponentials. In the extreme case, we used a transformation of

$$x = \xi + (\exp(d1 * (f(xi) - 1)) - \exp(-d1))/(1 - \exp(d1)),$$

where

$$f(xi) = (\exp(d1 * (xi - 1)) - \exp(-d1))/(1 - \exp(-d1))$$

with $d1$ ranging from 0 to 100. For $d1 = 10$, $pmx = temax/n**2$ reaches its asymptotic value of 225 to two significant figures at $n = 257$. For $d1 = 100$, the truncation error is 6 orders of magnitude higher than the no transformation case; nevertheless, pmx is constant to at least the first figure at $n = 8193$ and 16385.

These results are at variance with analyses which focus on the coefficients of the Taylor series terms in the original ("physical") independent variables. The present results also validate the claim in [16], that an analytical transformation of the equations does not change the order of the accuracy. If the coefficients of the transformation are also evaluated to $O(h^2)$ accuracy, as well as the hosted equation, then the overall results will be $O(h^2)$ accurate.

6.3. *3D Grid Generation Results*

A similar procedure is followed to validate the grid generation codes. As described earlier, the grid generation method involves solving a set of three elliptic equations for the new coordinates $\xi(\mathbf{x})$, written in the original ("physical") coordinates as

$$L(\xi_i) = P_i,$$

where L is the Laplacian operator in $\mathbf{x}$, and the P_i may be chosen to give some control of grid position in the interior. Since the system is transformed to ξ space, resulting in a set of three coupled nonlinear (quasilinear) equations, the validation procedure is more complicated. We chose the "solution" to be the transformation

$$\xi = x_i + \tanh(d_i x_1 x_2 x_3)$$

(which is the observe of the transformation used for the hosted equation testing

values of x_1, x_2, x_3 at each grid point. This nonlinear 3×3 system is solved by coupled 3×3 Newton–Raphson iteration at each grid point to establish the continuum grid solution. Again, tight convergence criteria were used, assuring accuracy to essentially the single precision of the machine.

The P_i necessary to produce this transformation are then obtained by operating on the solution ξ with the Laplacian L, giving

$$P_i = -2d_i^2[x_2^2(x_3^2 + x_1^2) + x_1^2 x_3^2] \operatorname{sech}(d_i x_1 x_2 x_3)^2 \tanh(d_i x_1 x_2 x_3)].$$

The numerical solution is obtained using hopscotch SOR to solve linearized equations sequentially for x_1, x_2, and x_3, followed by Picard outer iterations to update the linearized coefficients. A relaxation factor of 1/2 was used in the outer (nonlinear) iterations, and loose iterative convergence criteria (actually, a limit on maximum number of inner iterations) were used in the early nonlinear stages. Again, overall iterative convergence was tight, essentially to the single precision of the machine, to clarify the truncation error behavior. The cost of solving the nonlinear 3×3 system for the grid test was more expensive than solving the scalar linear equation for the hosted equation test, but was still reasonably obtained because the exact continuum solution was available for use as initial conditions.

The result for an early test in two grids is given in Table IV, for strong stretching values $d_i = 3$. The values of *temax* seem small, and in fact decreased by almost a factor of 2 with the mesh size halving. This test could likely pass for a validation. The test was actually inconclusive and, in fact, the test driver code was in error.

The false result is included here to emphasize that systematic convergence testing over a sequence of grids is required for validation. The error was in fact discovered with such systematic testing, the results of which are shown in Table V, for both the original and corrected test drivers.

The complete, systematic convergence test clearly indicate a persistent error in the original test, and clearly indicate second-order convergence in the corrected test.

6.4. *Discussion of Validation Procedure*

We note that the validation procedure can be complicated by the use of first-order differencing, since this slows convergence and makes it difficult to judge. Also,

TABLE IV

Grid Generation Tests, $d_i b = b3$, with an Error in Coding in the Test Driver

grid =	5^3	9^3
temax =	.394e−02	.205e−02

TABLE V

Complete Grid Generation Tests, $d_i = 3$

	grid =	5^3	9^3	17^3	33^3
original	*temax*100 =	.394	.205	.169	.160
corrected	*temax*100 =	.304	.0821	.0208	.00533
original	*pmx* =	.0630	.131	.431	1.646
corrected	*pmx* =	.0486	.0525	.0533	.0545

the use of mixed first- and second-order differencing (boundary conditions, advection terms at high Reynolds numbers, etc.) and/or any sort of conditional differencing (e.g., upstream differencing) will complicate the validation procedure. The technique is applicable to validating the hosted equation codes as long as the method of grid generation can systematically refine the grid, which is the case for elliptic grid generating systems themselves; particularly, the technique is applicable to validating the extension of the elliptic grid technique recently developed by Brackbill and Saltzmann [3], which provides control over the grid properties of smoothness and orthogonality.

As described above, the validation procedure simply gives a "yes" or "no" answer to the accuracy of the entire code: the transformation of the hosted equation(s), the substitution of the finite difference forms, the ensuing algebraic grouping, the FORTRAN encoding, the solution procedure for the discretized equations, and the correct formulation of the test problem. If the validation procedure produces a negative result, it is conceivable that the only information obtained is that there is an error, somewhere. For example, an error in the formulation could destroy the diagonal dominance of the matrix equation, resulting in an unstable or extremely slowly converging numerical procedure.

In actual experience, the procedure has been readily modified, on an ad hoc basis, and has successfully helped in isolating and identifying errors. By judiciously building up or modifying the test problem, one can selectively turn off the cross-derivative terms, perform the trivial identity transformation, etc. Direct solvers for the hosted equation testing may also be used if iterative convergence is not attainable. We have found also that printing and inspection of all values of the stencil at one internal grid point aided de-bugging and validation.

The cost of computer time to do the systematic convergence testing requiring can be significant, especially in 3D problems. However, it should be noted that the kind of code validation described here does not address the small parameter (high Reynolds number) problem, so the costs and difficulties associated with it do not contribute to the expense. In fact, designing the test problems, one specifically avoids large or small parameters, since these could mask errors in the treatment of the negligible terms. Also, if the code and the validation code are indeed correct, the use of the correct continuum solution as the initial condition for the iterative discrete solution will speed the testing process. (The test driver must require some minimal number of passes through an iterative solver, to avoid a false indication of

iterative convergence.) Thus, the computer CPU time to perform these validation computations in the finest grid will be much less than that required to obtain a realistic (unknown) problem solution in that grid, when iterative solution methods are used.

The results obtained strongly confirm that strong and inappropriate coordinate stretching can indeed increase the size of the truncation error, as is well known, but that the asymptotic order of the accuracy, as indicated by the reduction in truncation error resulting from systematic reduction in grid size, remains $O(h^2)$ if centered differences are used in the transformed equations.

7. Comments

When we first started on this project we decided to implement the symbol manipulation code at as "high" a level as was possible. By high level we mean that we would give the symbol code the differential equation in some natural coordinates and then let the chain rule do its work. This produces a surprisingly large equation in a general coordinate frame. The equation

$$\sum_{i=1}^{n} \frac{\partial}{\partial x_i} \sigma \frac{\partial}{\partial x_i} f = 0, \tag{7.1}$$

in fully expanded form, contains the number of terms indicated in the following table.

σ	n	No. of terms
constant	2	42
constant	3	1611
variable	2	50
variable	3	1710

Because of the size of this expression the remainder of the VAXIMA program became quite tricky. One of the earliest versions of the VAXIMA program for the $n = 3$ and variable σ case, ran about 60 cpu hours on a VAX 11/780 running under UNIX and produced a SETM3 subroutine of approximately 1800 lines of dense FORTRAN code. There were some minor problems in getting the subroutine to compile. However, as soon as the subroutine compiled, it ran and produced correct results. (Surprisingly, round-off error was not a problem, despite the arithmetical complexity.) This was accomplished in less than a month of full-time work. It is, we believe, essentially impossible to duplicate the above results by hand in any amount of time.

At this point it seemed that the run time of the symbol program was out of line with the amount of work being done by the symbol code. It was found that during the replacement of derivatives by atomic variables the simplifier was rearranging

the terms in the differential equations in an attempt to find simplification where none existed. At this point a code was written to break up all of the large expressions into pieces that contain at most a few terms. This prevented a major portion of the wasted algebra and reduced the run time of the VAXIMA code to under 3 cpu hours.

The current approach teaches VAXIMA something about the structure of the intermediate expressions occurring in the chain rule. This produced the code listed in this paper and has a running time under 8 cpu minutes.

The results in Section 2 which allowed us to write the faster VAXIMA programs are based on a formula for the derivative of the inverse of a matrix function. Thus if $m(t)$ is a matrix whose entries depend on t, then

$$\frac{d}{dt} m^{-1}(t) = -m^{-1}(t) \frac{dm(t)}{dt} m^{-1}(t). \tag{7.2}$$

It is interesting to note that VAXIMA does not know this formula. On the other hand, VAXIMA does know the derivative of the square of a matrix function,

$$\frac{d}{dt} m^2(t) = m(t) \frac{dm(t)}{dt} + \frac{dm(t)}{dt} m(t). \tag{7.3}$$

As we noted in the introduction, we consider this paper an interim report on this project. We believe what has been accomplished so far provides a useful tool for constructing FORTRAN code. However, there are many problems still left. One important problem is to extend the VAXIMA code to handle general boundary conditions. Simple boundary conditions are now hand coded. It is also important to solve the problem of setting up intermediate expressions so that the evaluation of our formulas is more efficient.

We believe that it is now possible to extend the ideas of this paper to more general equations including systems and evolution problems. It should be possible to handle other methods of generating coordinates including hyperbolic generation and to use higher order difference schemes. With the experience we have gained it is now possible to set up a package of symbol manipulation programs that would be useful to others working on problems similar to the problem studied here. Potential future uses include the following: deferred corrections for higher order accuracy and/or nonlinear terms in single and multiple equations; combination of perturbation methods and numerical methods; constitutive equation testing, in areas like turbulence modeling, non-newtonian fluids, soil mechanics, flow in porous media, gravitational theory; generation and analysis of new discrete forms via finite difference, finite element, last squares, etc. methodologies; and application of coordinate transformations in conjunction with all the above.

We anticipate that the practice of disciplines like computational fluid dynamics will be revolutionized in the next decade as the power of symbolic manipulation becomes widly recognized and applied.