Symbolic and Numerical Computation for Artificial Intelligence

edited by

Bruce Randall Donald
Department of Computer Science
Cornell University, USA

Deepak Kapur
Department of Computer Science
State University of New York, USA

Joseph L. Mundy
AI Laboratory
GE Corporate R&D, Schenectady, USA

Academic Press
Harcourt Brace Jovanovich, Publishers
London San Diego New York
Boston Sydney Tokyo Toronto

ACADEMIC PRESS LIMITED
24–28 Oval Road
London NW1

US edition published by
ACADEMIC PRESS INC.
San Diego, CA 92101

This book is printed on acid-free paper

A catalogue record for this book is available from the British Library

ISBN 0-12-220535-9

Printed and Bound in Great Britain by
The University Press, Cambridge

Mathematical Modeling, Symbolic and Numerical Systems

Chapter 11

Basic Requirements for the Automatic Generation of FORTRAN Code

Stanly Steinberg[†]

Department of Mathematics and Statistics

University of New Mexico

Albuquerque, New Mexico 87131

The design of a numerical algorithm has a critical impact on the possibility of using computer symbol manipulation techniques to implement the algorithm in a numerical programming language. This paper describes a finite-difference discretization of a general elliptic partial-differential operator that is used to approximate a general boundary-value problem involving such operators. The boundary-value problems are posed in complex regions in three dimensions. Samples of generated FORTRAN code, some of the basic symbol manipulation tools, and portions of the special purpose symbol code used for generating the FORTRAN are given. This technology was used to produce thirty pages of FORTRAN code for a three-dimensional problem; the resulting code contained only one error.

1. Introduction

Part of a project to build a modeling program for porous media flow in a complex three-dimensional underground region required the generation of FORTRAN code for solving a boundary value problem (BVP) for a linear, second-order, symmetric, elliptic, partial differential equation (PDE) with Robin (mixed) boundary conditions (BCs). Because of the nature of the porous media, the coefficients of the second derivatives in the PDE form a general symmetric matrix with discontinuous entries. The problem that was modeled required a mixture of Dirichlet and Neumann boundary conditions so, to cover all cases, full Robin boundary conditions were implemented. A significant part of the FORTRAN code was generated using symbol manipulation (MACSYMA Reference Manual, 1988). The size of the symbol manipulation effort can be estimated from the fact that the generated code for the three-dimensional case is just under thirty pages long.

The complexity in this project comes from the general coefficients of the PDE, the

[†] This work was partially supported by the Office of Naval Research, Sandia National Laboratories, the Army Research Office, and the Air Force Weapons Laboratory.

general boundary conditions, and the underlying geometry. Numerical grid-generation techniques (Thompson *et al.*, 1985; Steinberg and Roache, 1986a) are used to handle the complex geometry. The new code was integrated into an existing modeling program, so the modules from that program which handle the input of the geometry, input of parameters for the PDE and BCs, output of numerical results, and graphics were used in this project. Most of the remaining code was written using MACSYMA.

Both numerical algorithm design and the building of symbol manipulation tools (Steinberg and Roache, 1987) played a critical role in this project, so both will be described here. The numerical algorithm has the following major parts:

1 generate a transformation of the given region to a unit box using boundary conforming coordinates (Thompson *et al.*, 1985; Steinberg and Roache, 1986a);
2 transform the PDE and BCs to the new coordinate system;
3 calculate the stencils for (coefficients of) the difference approximations of the PDE and BCs; and
4 solve the resulting system of algebraic equations.

The grid-generation problem in step 1 has been solved previously (Steinberg and Roache, 1986a) and the required FORTRAN code was generated using MACSYMA (Steinberg and Roache, 1987). Some previous work had been done on generating finite-difference code (Steinberg and Roache, 1988); a main point of this project was to extend this work to finite-volume formulations and general boundary conditions and to generate code for steps 2 and 3. An extensive theoretical background for this work is given in Steinberg and Roache (1991). In step 4, the algebraic equations are solved using a semi-coarsening multigrid algorithm of Dendy *et al.* (1989) (SOR and line SOR are also used). In the papers (Steinberg and Roache, 1986b, 1987, 1988; Florence *et al.*, 1987; Liska and Drska, 1990), related work on the use of symbol manipulators to generate FORTRAN is presented.

The numerical algorithms are derived by first transforming the BVP to boundary-conforming coordinates and then discretizing the resulting PDE and the BCs. Two ideas are used to derive difference schemes: one is a finite-difference approach based on nodal positions for the unknowns; and the other is a finite-volume approach based on cell-centered positions for the unknowns. The second approach is the most difficult to program, so this paper only considers that approach.

The stencil loader is the code that computes the coefficients of the difference equations for the PDE and BCs. The stencil loader has code for: evaluating metrics; transforming the coefficients of the PDE and BCs; averaging of coefficients; calculating stencils; calculating the residue; and evaluating fluxes. In general, the symbol-manipulation programs are designed to write FORTRAN code in any spatial dimension. This is important because it helps in debugging both the symbol and FORTRAN code. Unfortunately, we were not able to accomplish this for some of the boundary code.

Although the use of grid generation greatly simplifies the geometry, there are still significant programming problems left. In three dimensions, the geometry becomes a unit cube. In the finite volume approach, the unit cube is divided into cells and then quantities in the interior of the region can be located at cell centers, cell face centers, cell edge centers, and cell corners. In addition, the boundary conditions are given on the six faces of the unit cube, and auxiliary conditions must be given along the twelve

edges and at the eight corners of the unit cube. The FORTRAN code will contain loops for calculating quantities at each class of points. It is a unique feature of this work that symbol-manipulation tools were developed for writing such loops.

This project required a substantial symbol-manipulator programming effort. However, this approach was very successful, producing essentially error-free FORTRAN code with a reasonable programming effort. The FORTRAN code produced is now being incorporated into groundwater-flow modeling programs. (This will be discussed further in section 11.)

What is important in this effort is that the numerical algorithm was designed so that a symbol manipulator could be used to handle all of the programming complexities, that is, the numerical grid-generation technology along with transforming the BVP reduces all geometric problems to algebraic problems. The user must still must describe the geometry, that is, provide points on the boundary of the region (the user interface and grid-generation codes help with this task). The algorithm is independent of the boundary description. However, transforming the BVP to a unit cube and the use of general boundary conditions requires the FORTRAN code to contain many loops and many types of loops, all of which are written using a symbol manipulator. Moreover, the symbol code that writes loops for computations on the boundary must keep track of which direction points to the interior of the cube. As far as we know, this is the first time a symbol manipulator has been used to construct such complex numerical-programming logic.

This paper uses concepts from the theory of PDEs, numerical analysis and symbolic computing, and thus may present some difficulty for the reader who is not familiar with all of these areas. The text by Birkhoff and Lynch (1984) provides excellent background material on PDEs and the numerical analysis used in this paper. A MACSYMA Reference Manual (1988) is needed for reading the latter part of this paper.

2. The Boundary Value Problem

The goal is to solve the boundary value problem

$$Lf = g \quad \text{in} \quad \Omega,$$

$$\alpha \frac{\partial f}{\partial n} + \beta f = \gamma \quad \text{on} \quad \partial\Omega$$

for f where Ω is a region in m dimensional Euclidean space $\mathbf{R}^m$ and $\partial\Omega$ is the boundary of Ω (see figure 1). The operator L is defined by

$$Lf(x) = \sum_{i,j=1}^{m} \frac{\partial}{\partial x_i} \left(\sigma_{i,j}(x) \left(\frac{\partial}{\partial x_j} f(x) \right) \right)$$

and the normal flux is defined by

$$\frac{\partial f}{\partial n} = \sum_{i,j=1}^{m} n_i \left(\sigma_{i,j}(x) \left(\frac{\partial}{\partial x_j} f(x) \right) \right)$$

where the matrix σ is symmetric ($\sigma_{i,j} = \sigma_{j,i}$) and $\vec{n} = (n_1, n_2, \cdots, n_m)$ is the unit outer normal to $\partial\Omega$. As usual $x = (x_1, x_2, \cdots, x_m) \in \mathbf{R}^m$, $f = f(x)$, $g = g(x)$, $h = h(x)$, $\alpha = \alpha(x)$, $\beta = \beta(x)$, $\gamma = \gamma(x)$ and so forth.

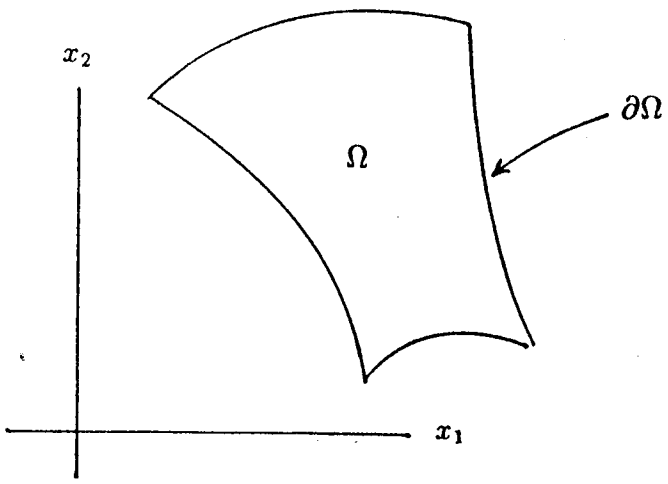

Figure 1. The physical region.

In the development of numerical algorithms, it is useful to introduce the flux vector

$$\vec{F} = (F_1, F_2, \cdots, F_m)$$

where

$$F_i = F_i(x) = \sum_{j=1}^{m} \left(\sigma_{i,j}(x) \left(\frac{\partial}{\partial x_j} f(x) \right) \right) .$$

Now, the differential operator can be written as a divergence of the flux vector:

$$Lf(x) = \sum_{i=1}^{m} \frac{\partial}{\partial x_i} F_i(x)$$

and the normal flux is the inner product of the flux vector with the normal

$$\frac{\partial f}{\partial n} = \sum_{i=1}^{m} n_i F_i .$$

3. The Transformation

The first step in the solution algorithm requires the BVP to be transformed to general coordinates. The new coordinates are called logical variables, and are labeled $\xi = (\xi_1, \xi_2, \cdots, \xi_m)$. It will be assumed that the transformation is boundary conforming and

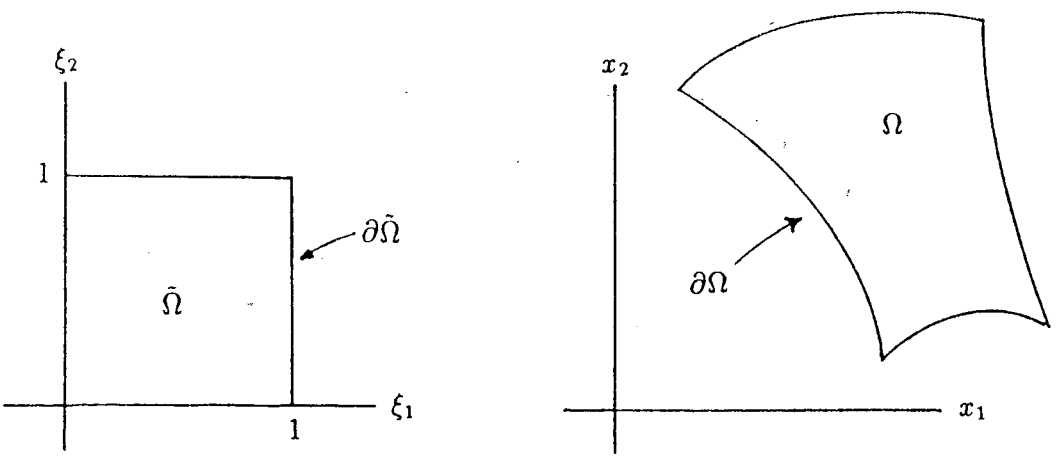

Figure 2. The transformation.

that a unit box $\tilde{\Omega}$ is transformed to the physical region Ω (see figure 2). Thus, the transformation has the form

$$x = x(\xi)$$

and the boundaries of the physical region are given by the union of the images of the sets where $\xi_i = 0$ or $\xi_i = 1$, $1 \leq i \leq m$. It is important for the algorithm that all quantities that involve the transformation be written as explicit functions of ξ. For more details, see Steinberg and Roache (1991).

The algorithm uses several metric quantities: the Jacobian matrix

$$\mathcal{J} = (J_{i,j}) = (\frac{\partial x_j}{\partial \xi_i}),$$

the Jacobian

$$J = \text{determinant}(\mathcal{J}),$$

and the cofactor matrix

$$\mathcal{K} = (K_{i,j}) = J\,\mathcal{J}^{-1}$$

of the Jacobian matrix. Thus $K_{i,j}$ is $(-1)^{i+j}$ times the determinant of the minor of $J_{j,i}$. The cofactor matrix plays a central role in transforming the BVP.

The chain rule can now be used to transform the BVP to the logical variables. However, the form of the transformed BVP is not unique because of an identity, called the metric

identity (Steinberg and Roache, 1991), that allows the chain rule to be written in two different forms, usually called the chain rule form and the conservation form. To maintain symmetry of the differential operator, both forms of the chain rule are used.

4. The Transformed BVP

The transformed BVP has the same form as the original BVP:

$$\tilde{L}\tilde{f} = \tilde{g} \quad \text{in} \quad \tilde{\Omega},$$

$$\tilde{\alpha}\frac{\partial \tilde{f}}{\partial n} + \tilde{\beta}\tilde{f} = \tilde{\gamma} \quad \text{on} \quad \partial\tilde{\Omega}$$

where

$$\tilde{L}\tilde{f} = \sum_{i,j} \frac{\partial}{\partial \xi_i}\left(s_{i,j}\left(\frac{\partial}{\partial \xi_j}\tilde{f}\right)\right),$$

and

$$s_{i,j} = \frac{1}{J}\sum_{l,m} K_{l,i}\sigma_{l,m}K_{m,j},$$

and where $\tilde{f}(\xi) = f(x(\xi))$, and $\tilde{g}(\xi) = J\, g(x(\xi))$. The formula for $\tilde{\alpha}$ is given below; $\tilde{\beta} = \beta$ and $\tilde{\gamma} = \gamma$.

As stated above, it is convenient to introduce a flux vector

$$\vec{\tilde{F}} = (\tilde{F}_1, \tilde{F}_2, \cdots, \tilde{F}_m),$$

where

$$\tilde{F}_i = \sum_j s_{i,j}\left(\frac{\partial}{\partial \xi_j}\tilde{f}\right)$$

and then the differential operator can be again written as divergence:

$$\tilde{L}\tilde{f} = \sum_i \frac{\partial}{\partial \xi_i}\tilde{F}_i.$$

Because the logical region is a box in m-dimensional space, its boundary is a union of $2m$ faces given by $\xi_i = 0$ or $\xi_i = 1$ (see figures 2 and 4). Thus, the boundary condition can be written on each face. The i-th coordinate planes are given by $\xi_i = \text{const}$, so the unit normal $\vec{n}_i$ to the i-th plane has components all zero, except that the i-th component is one. Consequently, the flux normal to the i-th coordinate plane is given by $\tilde{F}_i$.

To write the boundary conditions, introduce the coefficients α_i^k, β_i^k, and γ_i^k, where $k = 0, 1$ and $1 \le i \le m$. These coefficients have the same values as α, β, and γ on the part of $\partial\Omega$ where $\xi_i = k$. The outward flux at the boundary $\xi_i = k$ is given by

$$\frac{\partial \tilde{f}}{\partial \vec{n}_i} = (-1)^{k+1}\tilde{F}_i.$$

Now, the transformed boundary condition can be written

$$\tilde{\alpha}_i^k \frac{\partial \tilde{f}}{\partial \vec{n}_i} + \tilde{\beta}_i^k \tilde{f} = \tilde{\gamma}_i^k ,$$

where

$$\tilde{\alpha}_i^k = \frac{\alpha_i^k}{\|\vec{S}_i\|} , \quad \tilde{\beta}_i^k = \beta_i^k , \quad \tilde{\gamma}_i^k = \gamma_i^k$$

and where $\vec{S}_i$ is a column of K,

$$\vec{S}_i = (K_{1,i}, \cdots, K_{m,i}) .$$

Note that the transformed BVP depends only on the cofactor matrix and the Jacobian.

5. Finite Differences

The transformation, the PDE, and the BCs are discretized using standard central differences. The differencing of the transformation is discussed in the next section. The difference scheme for the PDE was designed so that the approximation is:

1 a second-order approximation to the continuum;
2 symmetric;
3 nearest neighbor; and
4 constants as solutions of the homogeneous problem.

The symmetry is for the PDE only, and does not include the boundary conditions. The first three properties involve only the differencing of the transformed operator, while the fourth property involves the metric identity (for more information, see Steinberg and Roache, 1991). The differencing of the PDE near the boundary and the differencing of the BCs involve ghost points. The term *ghost* is used to indicate that the points lie outside the physical region; this terminology is also applied to cells.

The differences in this section are given in continuum form, while in the next section they are given for discrete data. This is an important aspect of coupling the numerical algorithm with the symbol manipulation program. For example, the continuum notation is appropriate for doing a Taylor series analysis of truncation error, while the discrete formulation is used for writing FORTRAN code. MACSYMA utilities were written to translate between the continuum and discrete formulations.

Both half-step and full-step differences and averages are needed in the algorithm. The definitions are given in two variables (the generalization to m variables is clear). The half-step differences and averages are

$$\delta_\xi f(\xi, \eta) = \frac{f(\xi + \Delta\xi/2, \eta) - f(\xi - \Delta\xi/2, \eta)}{\Delta\xi} ,$$

$$\mu_\xi a(\xi, \eta) = \frac{a(\xi + \Delta\xi/2, \eta) + a(\xi - \Delta\xi/2, \eta)}{2} ,$$

while full-step differences and averages are

$$\mathcal{D}_\xi g(\xi,\eta) = \frac{g(\xi+\Delta\xi,\eta) - g(\xi-\Delta\xi,\eta)}{2\Delta\xi},$$

$$\mathcal{A}_\xi a(\xi,\eta) = \frac{a(\xi+\Delta\xi,\eta) + a(\xi-\Delta\xi,\eta)}{2}.$$

The partial differential operator contains two distinct types of terms: diagonal and off-diagonal. It is assumed that the off-diagonal terms occur in symmetric pairs. Diagonal terms have the form

$$\frac{\partial}{\partial\xi}\left(a(\xi,\eta)\frac{\partial}{\partial\xi}(f(\xi,\eta))\right),$$

while the off-diagonal terms have the form

$$\frac{\partial}{\partial\xi}\left(b(\xi,\eta)\frac{\partial}{\partial\eta}(g(\xi,\eta))\right) + \frac{\partial}{\partial\eta}\left(b(\xi,\eta)\frac{\partial}{\partial\xi}(g(\xi,\eta))\right).$$

The diagonal terms are differenced using

$$\frac{\partial}{\partial\xi}\left(a(\xi,\eta)\frac{\partial}{\partial\xi}(f(\xi,\eta))\right) \approx \delta_\xi\left((\mu_\xi a(\xi,\eta))(\delta_\xi f(\xi,\eta))\right),$$

while the diagonal terms are differenced using

$$\frac{\partial}{\partial\xi}\left(b(\xi,\eta)\frac{\partial}{\partial\eta}(g(\xi,\eta))\right) \approx \mathcal{D}_\xi\left(b(\xi,\eta)(\mathcal{D}_\eta g(\xi,\eta))\right).$$

6. Discrete Data

Two types of grids are used in this work: node on boundary and cell edge on boundary. The first grid corresponds to a finite-difference approach to the discretization, while the second grid corresponds to a finite-volume approach to the discretization. Both approaches produce finite-difference schemes. The second approach is somewhat more complicated, from the symbol manipulation point of view, than the first approach, so the second approach is described below. The given and computed quantities can have nodal, cell-centered, or cell face-centered positions.

To obtain an edge on boundary grid, the unit box is divided into cells (see figure 3 for a two-dimensional example). To simplify the notation, this discussion is restricted to two dimensions (again, the generalization to m dimensions is clear). If the unit cube is divided into cells M by N cells, then the nodes are given by

$$\xi_i = \frac{i}{M}, \quad \eta_j = \frac{j}{N}, \quad 0 \le i \le M, \quad 0 \le j \le N.$$

To include the ghost cells, extend the index ranges to $-1 \le i \le M+1$ and $-1 \le j \le N+1$. The cell centers (including ghosts) are given by

$$\xi_i = \frac{i+\frac{1}{2}}{M}, \quad \eta_j = \frac{j+\frac{1}{2}}{N}, \quad -1 \le i \le M, \quad -1 \le j \le N.$$

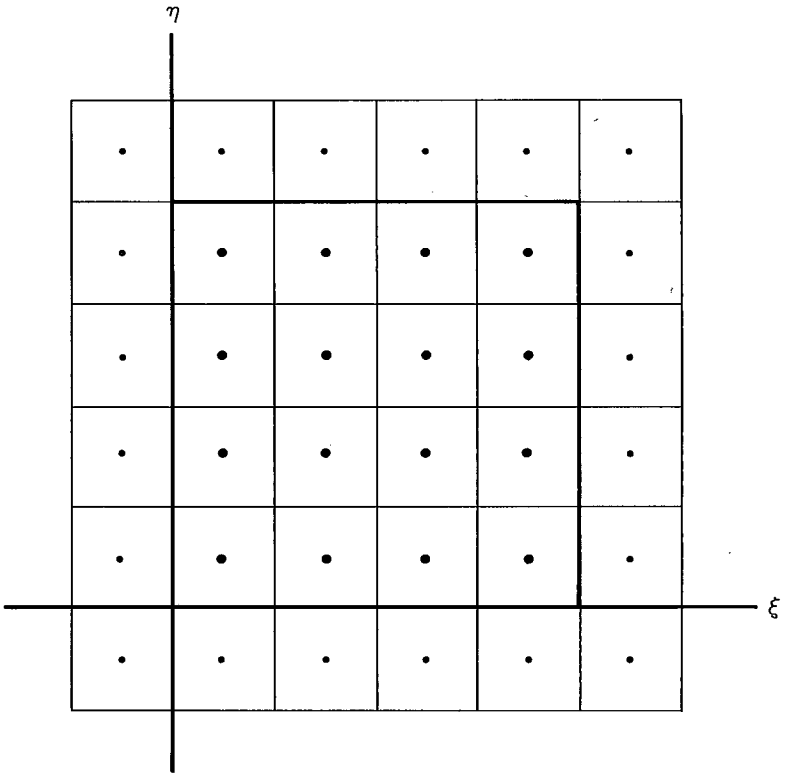

Figure 3. Edge on boundary grid.

The horizontal face centers (no ghosts) are given by

$$\xi_i = \frac{i+\frac{1}{2}}{M}, \quad \eta_j = \frac{j}{N}, \quad 0 \le i \le M-1, \quad 0 \le j \le N,$$

while the vertical face centers are given by

$$\xi_i = \frac{i}{M}, \quad \eta_j = \frac{j+\frac{1}{2}}{N}, \quad 0 \le i \le M, \quad 0 \le j \le N-1.$$

To translate the difference algorithms to FORTRAN, the given and computed quantities must be stored in arrays. The following conventions are used. The coordinates of the transformation are taken as nodal quantities

$$x(i,j) \leftarrow x(\frac{i}{M}, \frac{j}{N}),$$

while the solution values are taken as cell centered quantities

$$f(i,j) \leftarrow \tilde{f}(\frac{i+\frac{1}{2}}{M}, \frac{j+\frac{1}{2}}{N}).$$

The fluxes are face centered, and thus come in two types:

$$F_1(i,j) \leftarrow \tilde{F}_1(\frac{i+\frac{1}{2}}{M}, \frac{j}{N})$$

and

$$F_2(i,j) \leftarrow \tilde{F}_2(\frac{i}{M}, \frac{j+\frac{1}{2}}{N}).$$

Differences are now computed using the formulas given in the previous section. For example, the ξ derivative of the x coordinate at cell centers is given by

$$\frac{\partial x}{\partial \xi} \approx \delta_\xi \mu_\eta x(\xi,\eta),$$

while the vertical face-centered derivatives of f are computed using

$$\frac{\partial \tilde{f}}{\partial \xi} \approx \delta_\xi \tilde{f}(\xi,\eta)$$

and

$$\frac{\partial \tilde{f}}{\partial \eta} \approx \mu_\xi \mathcal{D}_\eta \tilde{f}(\xi,\eta).$$

The horizontal face-centered differences have similar formulas. Also, the ξ derivative of $\tilde{F}_1$ and the η derivative of $\tilde{F}_2$ are cell-centered quantities given by half-step differences. Complete formulas for differencing the transformation, PDE and BCs are given in Steinberg and Roache (1991). MACSYMA utilities are used to translate from the continuum difference formulas just given and the required formulas on discrete data stored in arrays.

It is important to note that the ghost points play a role in both the boundary and interior differences. For example, consider the lower horizontal boundary $\eta = 0$, that is, $j = 0$ in two dimensions. The flux at a point $(i+\frac{1}{2}, 0)$ involves both points with $j = \frac{1}{2}$ and $j = -\frac{1}{2}$. The points with negative j values are ghost points. This flux is used in the boundary condition and the flux-balance equations for the first interior cell. It is even more interesting to note that the flux on the vertical cell wall at $(i, \frac{1}{2})$ involves the ghost points $(i \pm \frac{1}{2}, -\frac{1}{2})$. Finally, the values of $\tilde{f}$ on the boundary are computed using central averages of values at a ghost point and a first interior point. The boundary conditions are second order accurate.

7. Global Geometry

The global geometry is relatively simple in one and two dimensions, but substantially more complicated in three dimensions, so the three-dimensional case is described below (see figure 4). Note that a cube in three dimensions has:

1	interior;
6	faces;
12	edges; and
8	corners.

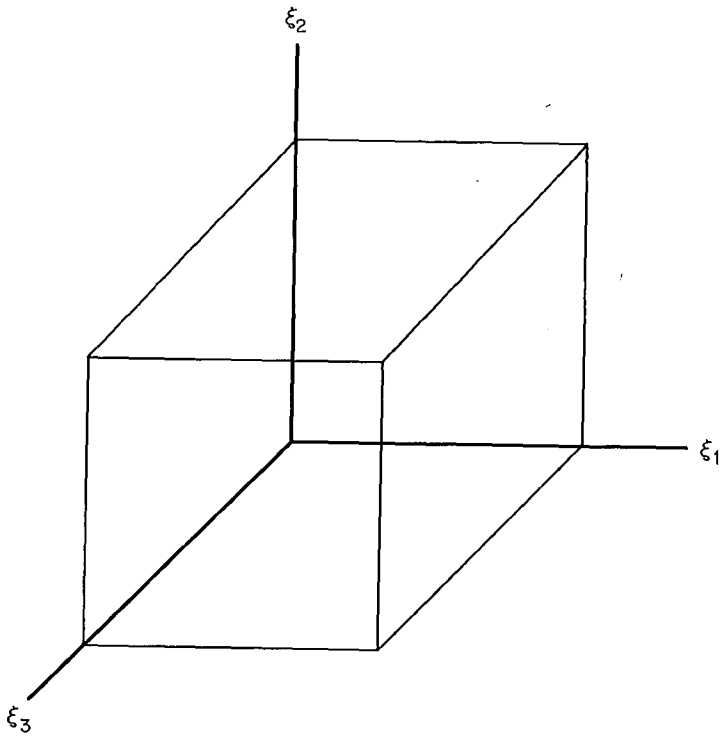

Figure 4. Logical space in 3D.

The stencil loader part of the FORTRAN code calculates various quantities using loops that pass over all interior nodes, interior cell-face centers, or cell corners. Similar loops must be constructed for the boundary faces, boundary edges, and boundary corners (see figure 4). Some of the loops use the ghost points while others do not. Another unique feature of the symbol code written for this project is that it has the capabilities of writing such loops. Section 8 gives examples of such loops.

8. FORTRAN Samples

In this section, some samples of the three-dimensional FORTRAN code generated by MACSYMA are presented; the samples are presented in a special font, essentially as MACSYMA produced them (some editing has been done for compactness of printing). The techniques used to generate the code are discussed in section 9. The three-dimensional code has 125 declarations: 32 real; 75 array; and 14 integer (all variables are declared). Also, the code contains 39 loops. The generation of code this size is, clearly, a complex task.

There are a number of useful things that MACSYMA can do besides compute formulas. First, certain parts of comment statements can be computed along with the formulas.

This is very helpful for human readers of the code. Second, it is easy to keep track of all variables introduced by MACSYMA and then use these in the declaration statements. When this is combined with the FORTRAN default variable type being *undeclared*, errors are much easier to detect. Also, MACSYMA computes the upper and lower values for array dimensions, and then it builds array declarations using this information. The following FORTRAN fragments illustrate these points. For example, the upper and lower limits on the i index have been computed by MACSYMA.

```
c     The arrays of grid points.
c         Stored at corners: i1-1 <= i <= il+2, etc.
      real x(i1d:ild, j1d:jld, k1d:kld)
```

A numerical grid gives a correspondence between points in physical space and the indices that lie in a box in a space which has the same number of dimensions as the physical space. In the finite-volume approach, physical space is divided into cells and the coordinates of the transformation are tabulated at the corners of the cells. The solution values are tabulated at the centers of cells in the interior of the region, while the fluxes are tabulated at cell face centers. In the interior of the region, loops must be constructed for calculations over cell corners, cell centers, and cell face centers. Similar loops must be generated for the faces, edges, and corners of the box. Samples of such loops follow:

```
c         Loop is over cell centers.
      do  1   k  =  k1 - 1  , kl + 1
      do  1   j  =  j1 - 1  , jl + 1
      do  1   i  =  i1 - 1  , il + 1

c         Loop is over face centers.
      do 21 k = k1-1,kl+1
      do 21 j = j1-1,jl+1
      do 21 i = i1,il+1

c         Loop is over face centers.
      i  =  i1
      do  611 j  =  j1 ,jl
      do  611 k  =  k1 ,kl

c         Loop is over edge centers.
      j  =  j1 - 1
      k  =  k1 - 1
      do  9111 i  =  i1, il
```

Formulas for derivatives of the transformation, computing metrics, averaging, evaluating the stencils, etc. must be produced. A few samples are listed below:

```
c The derivatives of the coordinate change.
      x1 = ( x(i+1,j+1,k+1)+x(i+1,j+1,k)+x(i+1,j,k+1)+x(i+1,j,k)-
     1       x(i,j+1,k+1)-x(i,j+1,k)-x(i,j,k+1)-x(i,j,k)  )/h1/4.0

c Compute the metric quantities.
      vk11 = y2*z3-y3*z2
      vjac = vk13*x3+vk12*x2+vk11*x1

c The transformed coefficients.
```

```
      a11(i,j,k) = ( vk31**2*vsig33+2*vk21*vk31*vsig23+vk21**2*vsig22
     1    +2*vk11*vk31*vsig13+2*vk11*vk21*vsig12+vk11**2*vsig11)/vjac


c The stencil formulas.
c The coefficient of  f(i - 1, j, k)
      stn011(i,j,k) = ah11(i,j,k)/h1**2
c The coefficient of  f(i - 1, j - 1, k)
      stn001(i,j,k) = (a12(i,j-1,k)+a12(i-1,j,k))/(h1*h2)/4.0
c The coefficient of  f(i, j, k)
      stn111(i,j,k) = -stn211(i,j,k)-stn121(i,j,k)-stn112(i,j,k)
     1                -stn110(i,j,k)-stn101(i,j,k)-stn011(i,j,k)
```

9. Basic Symbol Tools

There is no symbol manipulator which provides all of the basic tools needed for FORTRAN code generation and analysis. In particular, tools are needed for manipulating finite-difference equations (FDEs) in both the continuum and discrete settings. Simple tools were built for: (i) basic finite difference formulas; (ii) transforming the PDE and BCs; (iii) converting a PDE to a continuum FDE; (iv) converting a continuum FDE to a discrete FDE; (v) collecting coefficients and building stencil formulas; and (vi) writing FORTRAN code. Here is an example of a central difference formula. The lines labeled with c_ are MACSYMA input lines, those labeled with d_n are MACSYMA output lines.

```
(c_7) cd(f(x),x,1,1/2);
```

$$(d_7) \qquad \frac{f\left(x+\frac{dx}{2}\right)-f\left(x-\frac{dx}{2}\right)}{dx}$$

Tools are needed for both algorithm generation and algorithm checking. The most elementary check is a truncation error analysis. The code generated by MACSYMA was checked by this method; it took about six cpu hours on a Sun 3 workstation to check the three-dimensional code. The Taylor series computation, used in the check, requires the use of the *newdiff* option from the Partial Differentiation Package in MACSYMA (see the MACSYMA Reference Manual, 1988). An example of a truncation error calculation follows:

```
(c_11) expression : cd(f(x),x,1,1/2)-diff(f(x),x,1);
```

$$(d_{11}) \qquad \frac{f\left(x+\frac{dx}{2}\right)-f\left(\left(x-\frac{dx}{2}\right)\right)}{dx}-\frac{\partial f}{\partial x}(x)$$

```
(c_12) taylor(expression,dx,0,4);
```

$$(d_{12}) \qquad \frac{\frac{\partial^3 f}{\partial x^3}(x)\,dx^2}{24}+\frac{\frac{\partial^5 f}{\partial x^5}(x)\,dx^4}{1920}+\cdots$$

The conclusion is that the central difference is a second-order approximation of the first derivative.

One of the basic tools needed for stencil generation is a list of all of the nearest neighbors of a point in n-dimensions. The function *displace* generates a list of the differential displacements of the nearest neighbors of a point.

```
(c_1) displace([x,y],1);
```

$$(\mathrm{d}_1) \quad [[-dx,-dy],[-dx,0],[-dx,dy],[0,-dy],[0,0],[0,dy],[dx,-dy],[dx,0],[dx,dy]]$$

It is particularly useful to develop algorithms in continuum notation (as is used in the previous example) and then convert this notation to an index notation. The function *index_note* was written to do this.

```
(c_14) expression : expand(ca(cd(f(x,y),x,1,1),y,1,1));
```

$$(\mathrm{d}_{14}) \quad \frac{f(x+dx,y+dy)}{4\,dx}+\frac{f(x+dx,y-dy)}{4\,dx}-\frac{f(x-dx,y+dy)}{4\,dx}-\frac{f(x-dx,y-dy)}{4\,dx}$$

```
(c_15) index_note(expression,[x,y],[i,j],1);
```

$$(\mathrm{d}_{15}) \quad \frac{f(i+1,j+1)}{4\,h1}+\frac{f(i+1,j-1)}{4\,h1}-\frac{f(i-1,j+1)}{4\,h1}-\frac{f(i-1,j-1)}{4\,h1}$$

The *fortran* function can be used to convert the previous expression to FORTRAN syntax. A more compact form of the expression is obtained by using the *factor* or *ratsimp* functions.

10. The Symbol Programs

The symbol manipulation programs follow the theoretical discussion given in the previous sections of this paper; see Steinberg and Roache (1991) for more details. Thus, once the basic tools are understood, the main symbol manipulation programs are easily understood. In this section, several basic utility programs are listed and described. (The comments in the symbol programs have been deleted and replaced by descriptions.) The symbol manipulation programs use three list of variables. In three dimensions they are: `physical` $= [x,y,z]$, `logical` $= [\xi,\eta,\zeta]$, and `index` $= [i,j,k]$. Because the index variables i,j,and k are used in the FORTRAN code, the symbol code uses the variables `ii`, `jj`, and so forth, to index loops.

The block of code listed below is from the *transformation* function, which is used to generate FORTRAN code for the evaluation of derivatives of the coordinate transformation. Here *cd* is a central difference, *ca* is a central average, and *lt* is a left translate. All other functions are standard MACSYMA utilities. The variable *transformation* has previously been constructed from the `physical`, `logical` lists and has as its value a list:

$$[x(\xi,\eta,\zeta),y(\xi,\eta,\zeta),z(\xi,\eta,\zeta)]$$

```
real : [],
formulas : [],
for ii thru nn do for jj thru nn do (
   temp1 : concat(physical[ii], jj),
   real : endcons(temp1, real),
   temp2 : cd(transformation[ii], logical[jj], 1, 1/2),
   for kk thru nn do if kk # jj then
      temp2 : ca(temp2, logical[kk], 1, 1/2),
   for kk thru nn do
      temp2 : lt(temp2, logical[kk], 1/2),
              /* Simplify the derivatives. */
      temp2 : factor(temp2),
   formulas : endcons(temp1 = temp2, formulas),
end_ii_jj_do),
```

The `formulas` list is used to hold the formulas that are later converted to FORTRAN. To understand the first loop, assume that `ii` = 1, `jj` = 1, and `physical` = $[x, y]$. Then `temp1` = $x1$, `real` = $[x1]$, and `temp2` has as its value the central-difference formula (see above) for the derivative of the x coordinate with respect to its first argument. Because of the nature of the finite-volume formulation, the derivative must be averaged in all variables, except the one in which it is differenced, and then all variables must be translated by one half-step. This is accomplished in the two `kk` loops. After *transformations* is called, the list of formulas in `formulas` is translated to index notation, then to FORTRAN notation, and finally the code is printed into a file. This produces the formula for `x1` given in a previous example.

MACSYMA is used to convert the differential equation to a difference equation by substituting differences for derivatives. In the finite-volume formulation, different differences are used for the diagonal and off-diagonal terms in the PDE, so this is not totally trivial. In our programs, the solution of the PDE is called f, so the previous operations produce a finite-difference operator called *expression*. In one dimension, *expression* is a linear combination of $f(\xi - d\xi)$, $f(\xi)$, and $f(\xi + d\xi)$. It is the job of the *make_stencils* function to collect the coefficients of the various f values in *expression*, which are the formulas for the stencils. After translation to index notation and to FORTRAN, formulas such as those for `stn011`, `stn001`, and `stn111`, as shown above, are produced. A listing of part of the *make_stencil* program follows:

```
/* Create a list of all possible differential displacements. */
   list : displace(variables, width),
/* Collect the coefficients of the differences of
                              function in the expression. */
   expression : expand(expression),
   for jj thru length(list) do (
      place : function,
      for ii thru length(variables) do
         place : subst(variables[ii]+list[jj][ii], variables[ii], place),
      parts : bothcoeff(expression, place),
      full : name,
      for ii thru length(variables) do (
```

```
        temp : width + coeff(list[jj][ii], concat(d, variables[ii])),
        full : concat(full, temp),
     end_ii_do),
     full : apply(full, variables),
     parts[1] : ratsimp(parts[1]),
     if not(parts[1] = 0) then (
        real : endcons(full , real),
        stencils : endcons([full, parts[1], place], stencils)
     ) else (
        stn_zero : endcons([full, place], stn_zero) ),
  expression : expand(parts[2]),
  end_jj_do),
```

This is a computation in logical space, therefore, in two dimensions `variables` $= [\xi, \eta]$ and for a nearest neighbor stencil `width` $= 1$. The action of *displace* was given above; a typical entry in `list` is a list of two displacements. So, assume `list[jj]` $= [-d\xi, -d\eta]$ and `function` $= f(\xi, \eta)$. After the first pass through the `ii` loop `place` $= f(\xi - d\xi, \eta - d\eta)$. Then `parts` is a list containing the coefficient of `place` in `expression` and the rest of `expression`.

Next, assume `full` = `stn`. After the `ii` loop `full` = `stn00`, that is, the name for the lower left stencil. Next `full` $=$ `stn00`(ξ, η). The conditional checks to see if the coefficient is zero; if not, add `full` to the list of real variables and, if COEFFICIENT is the coefficient just computed, then add

$$[stn00(\xi, \eta), \text{COEFFICIENT}, f(\xi - d\xi, \eta - d\eta)]$$

to the list of stencils. This list is converted to index notation and then to FORTRAN, finally resulting in the following comment and code being inserted into the FORTRAN code.

```
c The coefficient of  f(i - 1, j - 1)
      stn00(i,j) = (a12(i,j-1)+a12(i-1,j))/(h1*h2)/4.0
```

11. Conclusions

When numerical algorithms are appropriately designed, then the use of a symbol manipulation program to convert the algorithm to FORTRAN code is effective and efficient. For some time, symbol manipulators have been used to derive formulas for numerical computations and convert such formulas to FORTRAN notation. This technology was used to great advantage in this project. The use of a symbol manipulator to construct variable declarations, comments, and loops is a novel aspect of this project.

The formulas in the FORTRAN code that calculate the stencils were read back into the symbol manipulator where they were used to reconstruct the finite-difference operator. A truncation-error analysis was then performed on the difference operator, confirming that it was second order accurate. Such an analysis essentially eliminates the possibilities of errors in the formulas (and none were ever found). In this process, some information from the FORTRAN comments was used, so even they were checked.

The first symbol manipulation programs were used to write FORTRAN code based

on the finite-difference formulation. Most of the symbol code was developed for the n-dimensional case, but it was tested for increasing dimension. At the point when the correct two-dimensional FORTRAN was finally generated, substantially more development time was required via the symbol manipulation route than would be required for hand coding the FORTRAN. However, now that much of the work for the three-dimensional case was done, by the time the three-dimensional FORTRAN was running, the symbol manipulation route was more efficient than hand coding. The next project was to write a finite-volume formulation code. Much of the symbol code that was written before was reusable, so the symbol manipulation route was substantially more efficient than hand coding. The final symbol code is about one and one-half times as large as the three-dimensional finite-volume FORTRAN code.

Certainly, accuracy is more important than efficiency. An important aspect of using a symbol manipulator to write FORTRAN is that errors in the symbol code typically produce FORTRAN code that is obviously incorrect, and thus, errors are easy to detect. For example, a sign error in the symbol code will typically appear in many places in the FORTRAN code, making it far easier to detect than a single typo. Thus, when one believes that the symbol manipulator is generating correct FORTRAN code, there are very few errors left.

Of course, all of the codes were rigorously tested numerically (Roache *et al.*, 1990) and some additional errors were found. The testing is done one dimension at a time. For the finite-volume approach, during the numerical testing, no errors were found in the one dimensional code, three errors were found in the two dimensional code, and one error was found in the three dimensional code. All errors were easy to find and of the type where a FORTRAN statement is not in the correct loop or a loop index had an incorrect limit.

Because the FORTRAN code is written symbolically, then symbolically tested using a truncation error analysis, and then thoroughly tested numerically, we believe that the probability of any remaining errors is extremely small.

This project has emphasized the construction of *mathematically* correct code. What is really needed is *numerically* correct code. Certainly, having no mathematical or programming errors is a critical first step. However, it is also important to have the formulas arranged so that numerical errors and operation counts are minimized. There is an excellent operation-count optimizer in MAPLE. Char *et al.* (1988) and van Hulzen *et al.* (1989) have produced operation-count optimizers for Reduce. The optimization of the operation count generally reduces numerical errors. We do not know of any symbol-manipulation work that directly addresses the problem of numerical errors. In the applications of this work, the generated FORTRAN code accounts for a small percentage of the execution time of a much larger program, so little effort was put into optimizing the code, aside from keeping the generated code simple and organizing the loops to take advantage of the way FORTRAN stores arrays. The numerical testing was done using approximately six decimal-digit arithmetic. Over a wide range of problems, the floating-point errors were in agreement with what would be predicted from the number of operations performed, and were well below the level of the numerical errors in other parts of the program.

Acknowledgments

Closely related material has been presented at: The Second International Conference on Expert Systems for Numerical Computing, Purdue University, West Lafayette, April 1990; The IV International Conference on Computer Algebra and its Applications in Physical Research, Joint Institute for Nuclear Research, Dubna, Russia, May, 1990; International Seminar on Computer Algebra and Applications in Mechanics, Novosibirsk, August, 1990; The Symposium on Symbolic Computations and Their Impact on Mechanics at the ASME Annual Winter Meeting, Dallas, November 1990; ACM-SIGNUM, Albuquerque, November, 1990; and in the minisymposium: The Generation of Numerical Code Using Symbol Manipulation, 13th IMACS World Congress on Computation and Applied Mathematics, Trinity College, Dublin, Ireland, July, 1991.

The author wishes to thank Dr. Patrick Roache and the company, Ecodynamics Research Associates, Inc., of which he is president, for the opportunity to apply the techniques described in this paper to "real" problems. Dr. Roache and the author had many important interactions as these applications were worked out.

References

G. Birkhoff and R.E. Lynch (1984), *Numerical Solution of Elliptic Problems*, SIAM, Philadelphia.

B.W. Char, K.O. Geddes, G.H. Gonnet, M.B. Monagan and S.M. Watt (1988), *MAPLE Reference Manual: Fifth Edition*, WATCOM Publications Limited, Waterloo, Ontario, Canada.

J.E. Dendy, S.F. McCormick, J.W. Ruge, T.F. Russell and S. Schaffer (1989), "Multigrid methods for three-dimensional petroleum reservoir simulation", SPE paper 18409, presented at the 10th SPE Symp. on Reservoir Simulation, Houston, TX.

M. Florence, S. Steinberg and P.J. Roache (1987), "Generating subroutine codes with MACSYMA", *Proc. 6th Int. Conf. Math. Modeling*, St. Louis, MO.

J.A. van Hulzen, B.J.A. Hulshof, B.L. Gates and M.C. van Heerwaarden (1989), "A code optimization package for REDUCE", *Proc. Int. Symp. Symbolic Algebraic Computation*, G.H. Gonnet, ed., ACM-Press, NY, 163–176.

R. Liska and L. Drska (1990), "FIDE: a REDUCE package for automation of Finite difference method for solving PDE", *Proc. Int. Symp. Symbolic Algebraic Computation*, Tokyo, 169–176.

MACSYMA Reference Manual (1988), version 13, Symbolics, Inc.

P.J. Roache, P.M. Knupp, R.L. Blaine and S. Steinberg (1990), "Experience with benchmark test cases for groundwater flow", *Proc. Forum on Benchmark Test Cases for Computational Fluid Dynamics*, ASME Fluids Eng. Division Spring Conf., Toronto, Canada.

S. Steinberg and P.J. Roache (1986a), "Variational grid generation", *Num. Math. P. Diff. Eqns.* **2**, 71–96.

S. Steinberg and P.J. Roache (1986b), "Using MACSYMA to write FORTRAN subroutines", *J. Symbolic Computation*, **2**, 213–216.

S. Steinberg and P.J. Roache (1987), "A toolkit of symbolic manipulation programs for variational grid generation", *Proc. Coupling Symbolic and Numeric Computing in Knowledge-Based Systems Workshop*, Boeing Comput. Services, Seattle, WA, 103–116.

S. Steinberg and P.J. Roache (1988), "Automatic generation of finite-difference code", *Symbolic Computation in Fluid Mechanics and Heat Transfer*, HTD-105, AMD- 97, H.H. Bau, T. Herbert and M.M. Yovanovich, eds., The Heat Transfer Division and the Applied Mechanics Division, ASME, Chicago, IL.

S. Steinberg and P.J. Roache (1991), "Discretizing symmetric operators in general coordinates", in preparation.

J.F. Thompson, A.U.A. Warsi and C.W. Mastin (1985), *Numerical Grid Generation*, North Holland, NY.