LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

SIMPLE: Part I
An Exercise in future Scientific Programming

Computation Structures Group Memo 273

July 1987

Kattamuri Ekanadham
IBM T.J. Watson Research Center
Yorktown Heights, New York

Arvind
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts

Simultaneously published as IBM/T.J. Watson Research Center Research Report 12686.

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology and at IBM. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-84-K-0099.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

SIMPLE: Part I

An exercise in future scientific programming

Kattamuri Ekanadham
IBM T.J.Watson Research Center
Yorktown Heights, New York

Arvind
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts

July 15, 1987

Abstract

Ideally, a high-level language should provide a way of writing abstractions which are as close to the problem domain as possible, as well as facilitate efficient implementations of these abstractions lest a user try to "get underneath" the abstractions. With the advent of parallel machines, a language such as FORTRAN fails on both counts. Functional and other declarative languages allegedly offer relief on both counts. The use of higher order functions, including the free use of curried forms, can dramatically raise the level of programming. In addition, such languages often have straight-forward operational semantics which admits tremendous opportunities for parallel execution. Programs in declarative languages, thus eliminate the problem of "detecting parallelism". In this report we have examined the first part of this claim by writing an application known as the SIMPLE code in a language called Id Nouveau [5]. The issues of parallelism will be examined in a companion report, Part II, to be published later. We have presented a high-level description of the algorithms used in the hydrodynamics problem, called SIMPLE, as described in [4]. Corresponding Id Nouveau program fragments are derived from the mathematical descriptions. The emphasis is on the style of programming, and not on algorithmic cleverness. The resulting program has 550 lines and runs successfully in ID WORLD [6], which is a compiler for Id Nouveau together with an abstract simulation facility for a dataflow machine. This report presumes the knowledge of Id Nouveau, functional programming and I-structures. Readers can obtain sufficient knowledge of Id Nouveau by reading [1]. A more thorough discussion of I-structures, a novel data structuring facility, may be found in [2].

1. Introduction

Fortran, in spite of its lack of support for user-defined abstractions, has remained the overwhelming choice of scientists and engineers. This may have to do with the availability of excellent Fortran compilers on a variety of computers, notably supercomputers, and the inertia of users and manufacturers. However, the introduction of parallel machines has exacerbated the programming problem enough that users may be willing to accept radical alternatives to Fortran. The alternative that is being put forth by the Computation Structures Group at M.I.T. is a declarative language called *Id Nouveau* [5]. *Id Nouveau* is the preferred language for programming the dataflow machine under development at M.I.T. In addition to the requirement of generating good code for parallel machines, *Id Nouveau* is also supposed to embody the advantages of declarative programming, that is, clear and concise code that is easy to understand and reason about. Functional languages are said to offer some of these advantages and indeed, very compelling arguments in favor of them have been made by Backus [3] and Turner [7]. However, functional languages have traditionally lacked good facilities for manipulating arrays and matrices; simulating such structures using traditional functional data structures often results in excessive storage demand or unnecessarily sequential code [2]. *Id Nouveau* is a functional language augmented with a novel array-like data structure called *I-Structures*. The objective of this report is to examine the coding of a non-trivial application in *Id Nouveau*. We want to evaluate the use of higher-order functions and *I-structures* in writing scientific programs; do such features bring clarity and succinctness to programming? At the same time we want to examine the efficiency of the resulting program in terms of storage use and parallelism.

The application we have chosen for our study is a hydrodynamics and heat conduction simulation program known as the SIMPLE code [4]. The SIMPLE document [4], along with the associated Fortran program, was developed as a benchmark (unclassified) to evaluate various high performance machines and compilers, including dataflow machines then under study at M.I.T. and elsewhere. Though SIMPLE is supposed to reflect some "real application", it is contrived to reflect a more complex mix of numerical methods than the usual problems in that class. For example, the hydrodynamics part uses an explicit method so that the new values for a zone depend only upon the previous values of that zone and its 6 neighbors. Whereas the conduction part uses an implicit method, so that the new temperature of a zone depends on the previous values of all zones in the mesh. In our study, we are neutral to the physics of the problem and even to the numerical algorithms chosen to solve the problem. Our goal is to see if the ideas can be expressed easily in *Id Nouveau* and if the resulting code preserves the parallelism of the algorithms employed. A good test of the succinctness of the code would be the ease with which a discriminating physicist can adjust the algorithms and code to his/her own specifications.

It is worth pointing out that, prior to starting this work, we were very familiar with the SIMPLE code, having written several versions of it in several earlier versions of *Id*. However, previous versions of SIMPLE in *Id* were written with the primary motivation of generating dataflow graphs and measuring parallelism and other quantities of interest from the dataflow architecture point of view. The program in this report, Part I, represents our first attempt at trying to write SIMPLE at a fairly high level of abstraction. Part II of this report, which is still under preparation, will document the parallelism of this program and contrast it with the parallelism measured in the earlier and lower-level

versions of the program. This report presumes the knowledge of *functional* languages, *I-structures* and *Id Nouveau* syntax. It is assumed that the reader has read [1] which provides an introduction to *Id Nouveau* and a critical analysis of imperative style programming. A comprehensive treatment of *Id Nouveau* may be found in [2,5].

1.1. Problem description

The problem is to simulate the behavior of a fluid in a sphere, using a Lagrangian formulation of equations. To simplify the problem, only a semi-circular cross-sectional area is considered for simulation, as shown in Figure 1. The area is divided into parcels by radial and axial lines as shown. Each parcel is delimited by four corners as illustrated in Figure 1. The corners are called *nodes*. The regions enclosed by 4 nodes are called *zones* (illustrated by the shaded area in Figure 1). In the Lagrangian formulation, the nodes are identified by mapping them onto a 2-dimensional logical grid, in which the grid points have coordinates (k, l) for some $kmin \leq k \leq kmax$, $lmin \leq l \leq lmax$. A zone is identified by specifying its *southeast* corner node. The product, $kmax * lmax$, is often referred to as the *grid size* of the problem. The following quantities are considered in the simulation. Note that the first two quantities are associated with nodes and the remaining are associated with zones.

1. $\vec{v}$: velocity components, u, w, of node (k,l) in the R-Z plane.
2. $\vec{x}$: coordinates, r and z, of node (k,l) in the R-Z plane.
3. a : Area of zone (k,l).
4. s : Volume of revolution per radian of zone (k,l).
5. ρ : Density of zone (k,l).
6. p : Pressure of zone (k,l).
7. q : Artificial Viscosity of zone (k,l).
8. ϵ : Specific internal energy within zone (k,l).
9. θ : Temperature of zone (k,l).

The main computation involves the determination of changes to the values of the above listed attributes as time progresses in discrete steps. The length of each time step varies and is determined from some of the attributes of the previous step.

1.2. Boundary Considerations

In order to incorporate appropriate boundary conditions, a fictitious layer of zones, called *ghost zones*, is added to the cross section, as depicted by the shaded area in Figure 2. Behavior of ghost zones is governed by desired boundary conditions. Each quantity is associated permanently with one of the following 3 types of boundary conditions. Each time new boundary values for a quantity are computed according to the associated criterion.

1. Constant boundary: The temperature of the boundary zones is kept at a constant value (heat source).
2. Continuous boundary: The ghost zones assume the area, volume, pressure and artificial viscosity of the adjoining zones, so that the effect of an infinite sequence of zones is created.
3. Reflected boundary: The positional coordinates of the nodes of the outer boundary of ghost zones are computed so that the ghost zone is a mirror image of the adjoining zone.

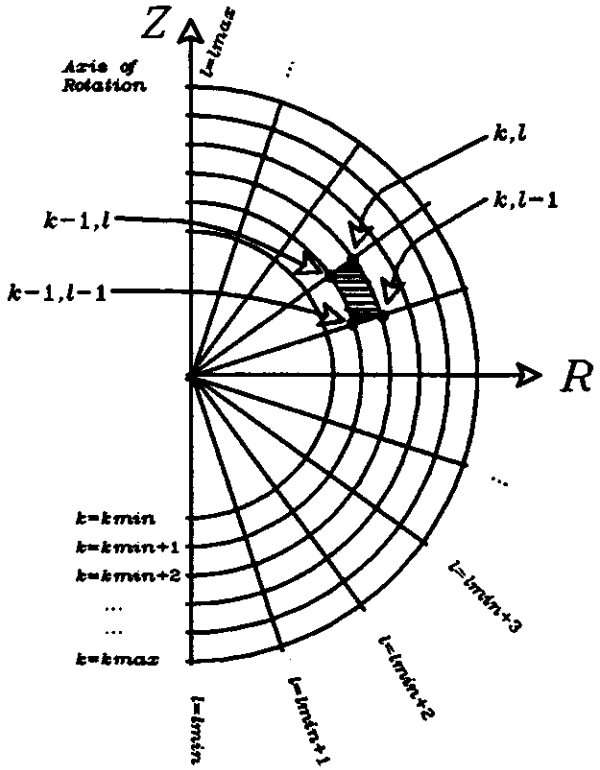

Figure 1: Fluid parcels in a 2-dimensional cross-section

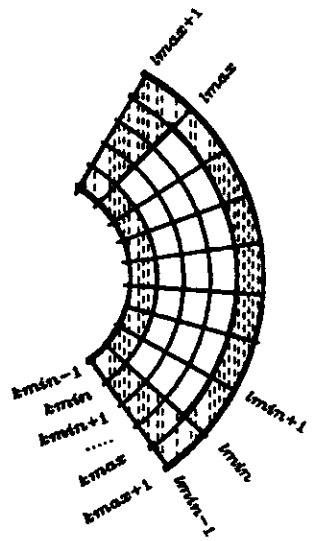

Figure 2: Ghost zones added for boundary conditions

2. Some Basic Programming Abstractions

In this section, we describe very briefly some of the basic abstractions that are used repeatedly. For subtleties about automatic unravelling of loop iterations, I-structure operations and execution semantics, the reader should refer to [1,5,2]. The notation used in this paper is actual *Id Nouveau* syntax, except at times for brevity (and to relate to the SIMPLE document [4]) greek symbols are used as identifiers and exponents are written as a^b, instead of $a \wedge b$. They can easily be translated into appropriate syntax. The text between a percentage sign, %, and the end of that line is a comment. Key words and some standard function names are shown in bold face. The key word ***fun*** is used to define a function in place - a kind of λ-*expression*. Thus for example, { **fun** x $x+1$ } denotes the successor function.

2.1. Neighborhood Abstractions

Here we define functions frequently used in the SIMPLE program to refer to the nodes and zones that surround a given node. In [4], the neighbor nodes are numbered 1 through 8. Here we define direction using names such as *north, south, etc.* for convenience. The neighbor zones are defined as specified in [4].

Neighboring nodes: The neighboring nodes of a given node, *(k,l)*, are referred to by the names shown in Figure 3. The following functions define them.

(1)

```
north (k,l) = (k-1,l);      farnorth (k,l) = (k-2,l);      northwest (k,l) = (k-1,l-1);
south (k,l) = (k+1,l);      farsouth (k,l) = (k+2,l);      southwest (k,l) = (k+1,l-1);
east  (k,l) = (k,l+1);      fareast  (k,l) = (k,l+2);      northeast (k,l) = (k-1,l+1);
west  (k,l) = (k,l-1);      farwest  (k,l) = (k,l-2);      southeast (k,l) = (k+1,l+1);
```

Neighboring zones: A zone is logically referred by giving the indices of its southeast corner. Thus for example, the zones $(i, lmin)$, $(i, lmax+1)$, $kmin \le i \le kmax+1$ and $(kmin, j)$, $(kmax+1, j)$, $lmin \le j \le lmax+1$ are the ghost zones. For each node, the 4 neighboring zones are named *A, B, C, D,* as shown in Figure 3. For example, zone B of node *(k,l)* is referred to by the indices of its southeast corner, namely *(k+1,l)*. For convenience, we define the following functions:

(2)

```
zone_a  (k,l) = (k,l);          zone_b  (k,l) = (k+1,l);
zone_c  (k,l) = (k+1,l+1);      zone_d  (k,l) = (k,l+1);
```

Index ranges for all nodes and zones: The computations involve calculations repeated over selected regions of the matrices. We define the following names to represent the index ranges of interest. Thus for example, when we set up a loop so that *(i,j)* iterate over the range *all_nodes*, the iteration will take place over all the nodes of the grid specified by the limits in the following definition.

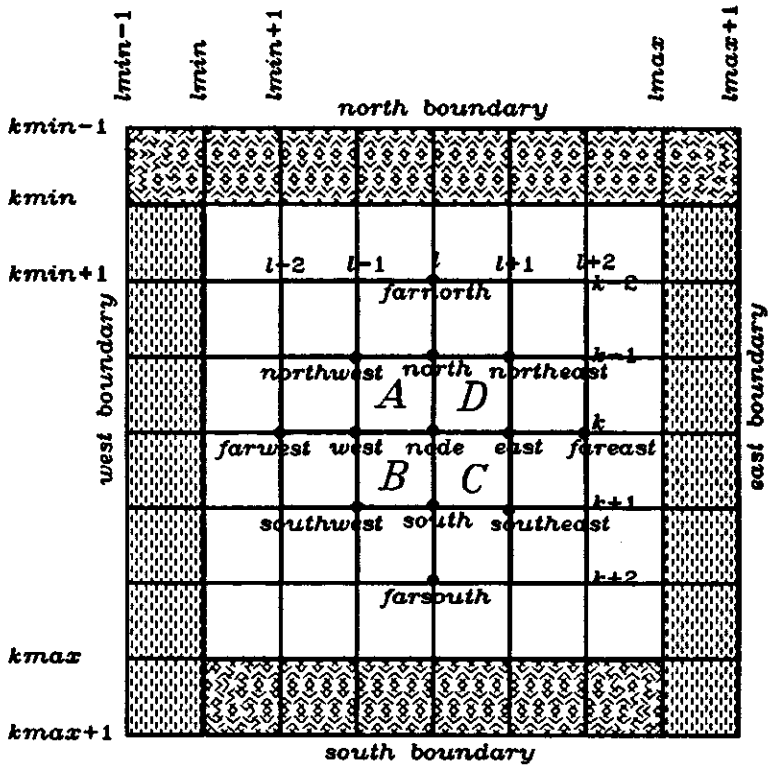

Figure 3: Logical naming of fluid parcels and their relative positions

(3)

all_nodes =	*((kmin-1, kmax+1), (lmin-1,lmax+1));*
interior_nodes =	*((kmin, kmax), (lmin,lmax));*
all_zones =	*((kmin, kmax+1), (lmin,lmax+1));*
interior_zones =	*((kmin+1, kmax), (lmin+1,lmax));*

Index ranges for boundary zones: There are many ways one can view the boundary zones. Here we define the segments for reflecting the neighboring zone properties, as indicated by the shading patterns in Figure 3. Other ways of looking at boundary zones are defined in the programs later, as needed.

(4)

west_boundary_zones =	*((kmin+1, kmax+1), (lmin,lmin));*
east_boundary_zones =	*((kmin+1, kmax+1), (lmax+1,lmax+1));*
north_boundary_zones =	*((kmin, kmin), (lmin,lmax+1));*
south_boundary_zones =	*((kmax+1, kmax+1), (lmin+1,lmax));*

2.2. Functions to manipulate matrices

A matrix is created using a generating function as follows:

```
make_matrix ((l1, u1), (l2, u2)) generate =                    (5)
    { A = matrix ((l1, u1), (l2, u2));
      {for i from l1 to u1 do
        {for j from l2 to u2 do
          A [i, j] = generate (i, j) }};
    in A};
```

Sometimes we create two matrices together as shown below, when the computations for corresponding elements share some common expressions. The generator is a function that returns a pair of values.

```
make_2_matrices ((l1, u1), (l2, u2)) generate =                (6)
    { A = matrix dimensions;
      B = matrix dimensions;
      {for i from l1 to u1 do
        {for j from l2 to u2 do
          A [i, j], B [i, j] = generate (i, j) }};
    in A, B};
```

Some times we create matrices using one generating function for boundary nodes and another for interior nodes. This abstraction is defined by the following functions.

```
fill_matrix A ((l1, u1), (l2, u2)) f =                         (7)
    {for i from l1 to u1 do
      {for j from l2 to u2 do
          A [i, j] = f (i, j) }};

make_matrix_2_ranges dimensions ((range1, f1), (range2, f2)) =  (8)
    { A = matrix dimensions;
      call fill_matrix A range1 f1;
      call fill_matrix A range2 f2;
    in A};
```

One can generalize the above notions and define *make_n_matrices*, *make_matrix_k_ranges* and *make_n_matrices_k_ranges*, for various values of n and k. All the matrix functions used in this report are particular cases of these abstractions, with suitable values for n and k.

The Following are some functions to perform reduction on matrices.

```
fold_row operator (i, l1, u1) generate =                                    (9)
    { s = generate (i, l1);
    in { for j from l1 + 1 to u1 do
           next s = operator s (generate (i, j));
         finally s } };

fold_range operator ((l1, u1), (l2, u2)) generate =                         (10)
    { s = fold_row operator (l1, l2, u2) generate;
    in { for i from l1 + 1 to u1 do
           next s = operator s (fold_row operator (i, l2, u2) generate);
         finally s } };

fold_2_ranges operator ((range1, f1), (range2, f2)) =                       (11)
    operator (fold_range operator range1 f1) (fold_range operator range2 f2);

accumulate = fold_range plus;                                               (12)
accumulate_2_ranges = fold_2_ranges plus;
minimum = fold_range min;
maximum = fold_range max;
```

Similarly one can generalize the above abstractions and define *accumulate_k_ranges, minimum_k_ranges* and *maximum_k_ranges* for various values of k. Finally we give a few simple abstractions:

```
determinant (a, b) (c, d) = a * d - b * c;   % returns the value of |a c|   (13)
                                                                    |b d|
constant x y = x;   % returns first argument
select direction A node = A [ direction node ] ;   % returns neighboring element
select_3 direction (A, B, C) node =
    A [ direction node ], B [ direction node ], C [ direction node ];
```

3. SIMPLE code

We now describe the details of the hydrodynamics problem and develop the corresponding *Id Nouveau* programs. First we provide an overview of the main computation which gives the main body of the program that is iterated many times. Later, we discuss the details of each of the computational steps and derive the corresponding definitions.

3.1. Overview of the computation

Equation (14) on page 9 shows the skeleton of the main computation. The comments indicate subcomputations. The symbol ϕ stands for some function of the indicated variables. A complete program is formed by replacing the boxes with the appropriately numbered definitions.

From the main loop body in equation (14), we can see that each step needs some data from the preceding step. In spite of this, one should note that the program has plenty of parallelism, as the use of I-structures gives the flexibility to return an array descriptor before the array is actually filled in. The evaluation of certain attributes are grouped as shown, to take advantage of shared computations in them. The steps are as follows:

1. The new velocity at each interior node is determined by computing the acceleration at that node and correspondingly incrementing the old velocity. The velocity vector, v, is represented as a pair of matrices, u and w, giving the two component velocities.

2. Similarly, the new positional coordinates are obtained by incrementing the old coordinates. The increment is the product of the corresponding new velocity and δt. The position vector, x, is represented as a pair of matrices, r and z, giving the two component coordinates.

3. The new area, volume and density of each zone are determined by geometrical approximations from the new coordinates of the nodes. The three computations share common subcomputations and hence are performed together.

4. The artificial viscosity is a fictitious quantity, to take into account the internal fluid properties of the mass in motion. It is a function of position, velocity and density.

5. The pressure, temperature and internal energy are interrelated by equations of state. For accuracy, they are computed twice, first by an approximation and then refined by substituting their values in the equations to yield better approximations. Once again, these three quantities share common expressions and hence are computed together.

6. The final temperature of the zones is computed from the heat conduction equations. This involves solving a system of linear difference equations. For the sake of efficiency a number of intermediate values are stored in temporary matrices during this heavy computation.

7. As a check for balance of energy, the change in energy is computed by determining the work done, heat lost and changes in internal and kinetic energies. The balance is checked to be within tolerance limits. (This check is not shown here.)

8. Finally δt for the next time step is computed. This is determined by taking a minimum over the temperature changes at each zone and by limiting it so that a sound signal cannot cross any zone within that time step.

% *Main program* (14)

{ % *basic abstractions*
equations (1)-(13), (21)-(22), (24), (33)-(35)

% *Initialization code not provided here*
% *constants = index ranges, polynomial coefficients, mass, conductivity, heat source*
% *variables* = δt, *u, w, r, z, p, q, s, error,* α, ρ, ε, θ

in { ***for*** *i* ***from*** *1* ***to*** *maximum_iterations* ***do***

% *select components for convenient access*
u, w = v; r, z = x; new_u, new_w = ***next*** *v; new_r, new_z =* ***next*** *x;*

% *velocity computation:* $new\_v = \phi(x, \rho, \alpha, p, q, v, \delta t)$
next *v =* { *equations* (15)-(19) ***in*** *new_v*} ;

% *positional coordinates:* $new\_x = \phi(x, new\_v, \delta t)$
next *x =* { *equations* (20), (23) ***in*** *new_x*} ;

% *area, volume, density:* $new\_\alpha, new\_s, new\_\rho = \phi(new\_x, \rho, s)$
next α, ***next*** *s,* ***next*** ρ = { *equations* (25)-(26) ***in*** new_α, *new_s, new_*ρ } ;

% *artificial viscosity:* $new\_q = \phi(new\_x, new\_\rho, new\_v, p)$
next *q =* { *equations* (27)-(28) ***in*** *new_q*} ;

% *pressure, temperature, energy:* new_ε, $interim\_\theta$, $new\_p = \phi(\rho, new\_\rho, new\_q, p, \varepsilon, \theta)$
next ε, $interim\_\theta$, ***next*** *p =* { *equations* (29)-(32) ***in*** new_ε, $interim\_\theta$, *new_p* };

% *heat conduction:* new_θ, $\bar{r}\_k$, $\bar{r}\_l = \phi(new\_\alpha, new\_x, interim\_\theta, \delta t)$
next θ, $\bar{r}\_k$, $\bar{r}\_l$ = { *equations* (36)-(41) ***in*** new_θ, $\bar{r}\_k$, $\bar{r}\_l$} ;

% *energy check:* $error = \phi(new\_x, new\_v, new\_p, new\_q, new\_\varepsilon, new\_\theta, \delta t, \bar{r}\_k, \bar{r}\_l)$
next *error =* { *equations* (42)-(51) ***in*** *error*} ;

% *time* ***step:*** $new\_\delta t = \phi(new\_x, new\_p, new\_\rho, new\_\alpha, new\_\varepsilon, \theta, new\_\theta, \delta t)$
next δt = { *equations* (52)-(56) ***in*** $new\_\delta t$ } ;

finally *x, v, p, q,* α, *s,* ρ, ε, θ δt, *error* }};

3.2. Velocity Computation

The velocity at each node is computed by first computing the acceleration at that node during the time step and incrementing the old velocity by the product of time and acceleration. The acceleration is obtained from the following equation for conservation of momentum:

$$\rho \frac{d\vec{v}}{dt} + \frac{\partial(p+q)}{\partial r}\vec{r} + \frac{\partial(p+q)}{\partial z}\vec{z} = 0$$

where, $\vec{r}$ and $\vec{z}$ are respectively the unit vectors in the r and z directions. Using Green's theorem the partial derivatives are approximated as line integrals shown below:

$$\rho \frac{d\vec{v}}{dt} + \frac{\oint (p+q)dz}{\oint r dz}\vec{r} - \frac{\oint (p+q)dr}{\oint r dz}\vec{z} = 0$$

We can rewrite the equation to give the two components of the acceleration as follows:

$$\frac{d\vec{v}}{dt} = \left[-\frac{\oint (p+q)dz}{\rho \oint r dz}, \frac{\oint (p+q)dr}{\rho \oint r dz} \right] = \left[\frac{-\oint p\,dz - \oint q\,dz}{\rho \oint r dz}, \frac{\oint p\,dr + \oint q\,dr}{\rho \oint r dz} \right]$$

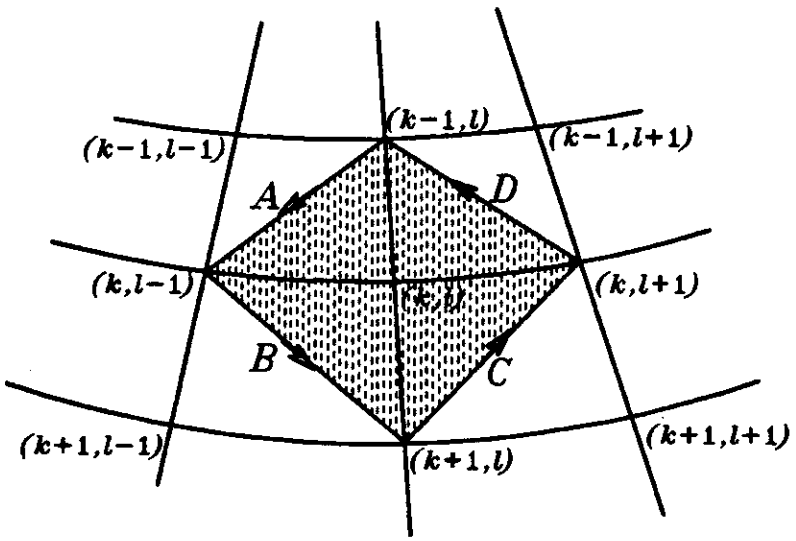

Figure 4: Mass and acceleration at a node

In the numerator, the line integrals are to be taken over the boundary line of the shaded region around node *(k,l)* shown in Figure 4. The line has four segments, *A, B, C, D,* one in each of the neighboring zones of node *(k,l)*. While integrating, it is assumed that each zone quantity is constant along the line segment within that zone. Thus for example, suppose we are integrating p with respect to z, along line A. Its value is given by the product of p in zone A and the difference of z between the two end points of line segment A. That is, $p[k,l] * (z[k,l-1] - z[k-1,l])$. In general, if we have two quantities represented as matrices f and g, and we want to integrate f with respect to g along the shaded region around some *node*, we can define the following line integral function:

```
line_integral f g node =                                              (15)
    f[zone_a node] * (g[west node] - g[north node]) +
    f[zone_b node] * (g[south node] - g[west node]) +
    f[zone_c node] * (g[east node] - g[south node]) +
    f[zone_d node] * (g[north node] - g[east node]);
```

In the denominator, the line integral $\oint rdz$, gives the area of the shaded area around the node *(k,l)* in Figure 4. The product of this area and the density ρ gives the mass in the shaded region. Instead of computing the line integral, this mass around the node is obtained in an alternative way, by approximating it as one half of the total mass of the 4 zones meeting at node *(k,l)*. The mass of each zone is taken as the product of its density and area. Thus, we can define the following function for computing the denominator:

```
nodal_mass node =                                                     (16)
    0.5 * (ρ[zone_a node] * α[zone_a node] +
           ρ[zone_b node] * α[zone_b node] +
           ρ[zone_c node] * α[zone_c node] +
           ρ[zone_d node] * α[zone_d node]);
```

Substituting appropriate applications of the above functions for the numerator and denominator, we obtain the following function to compute the two components of the acceleration at a node. We choose to define one acceleration function to yield a pair of values, (as opposed to a separate function for each component), so that recomputation of the common denominator is avoided.

```
acceleration node =                                                   (17)
    { d = nodal_mass node;
      n1 = -(line_integral p z node)-(line_integral q z node);
      n2 =  (line_integral p r node) + (line_integral q r node);
    in  (n1 / d,  n2 / d)};
```

The velocity at a node is now given by:

```
velocity node =                                                       (18)
    { u_dot, w_dot = acceleration node;
    in  u[node] + δt * u_dot,  w[node] + δt * w_dot};
```

The new velocity matrix is now created and filled by:

$$new\_v = make\_2\_matrices\ \ interior\_nodes\ \ velocity; \tag{19}$$

Note that *new_v* is a pair of matrices, *(u, w)*, representing the two component velocities. Alternatively one could build *new_v* as one matrix in which each element is a pair of values giving the two component velocities. However, in this case each time a component value has to be used, first the pair must be selected in the matrix and then the component in the pair must be selected. We prefer building two matrices because, later, we use each component matrix separately in many computations. The velocities at nodes on the ghost boundary are never used and hence we do not compute them.

3.3. Positional Coordinates

For the interior nodes, change in position is given by the equation: $\frac{d\vec{x}}{dt} - \vec{v} = 0$. Thus, the new coordinates of *interior* nodes are given by:

$$position\ \ node = r[\,node\,] + \delta t * new\_u[\,node\,],\ \ z[\,node\,] + \delta t * new\_w[\,node\,]; \tag{20}$$

Strictly speaking, the time intervals δt in the two definitions (18) and (20) are different. The acceleration, velocity and positional coordinates are computed at staggered intervals of time as depicted in Figure 5. Position and acceleration are computed at discrete time instants $t_{n-1}, t_n, t_{n+1} \ldots$, whereas velocity is computed at intermediate time instants $t_{n-1/2}, t_{n+1/2}, \ldots$ The relationships are given by the equations:

$$u_{n+1/2} = u_{n-1/2} + \Delta t_n * \dot{u}_n$$

$$r_{n+1} = r_n + \Delta t_{n+1/2} * u_{n+1/2}$$

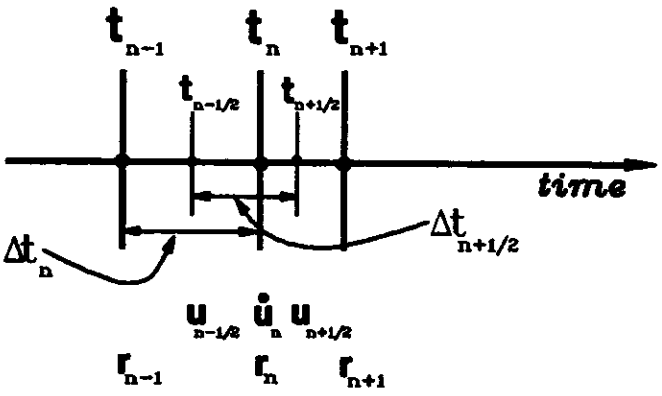

Figure 5: Staggered time instants for computing velocity and position

The value for $\Delta t_{n+1/2}$ is computed using the functions described later in section 3.9. The value of Δt_n is taken to be the average of $\Delta t_{n-1/2}$ and $\Delta t_{n+1/2}$. However, for the sake of simplicity in this report, we ignore the distinction between Δt_n and $\Delta t_{n+1/2}$ and use δt in place of both of them.

The coordinates of the nodes on the boundary are set in a special manner. In each step, the ghost boundary zone is set up so that each node on the ghost boundary is a reflection of the adjacent interior node with respect to the boundary line as shown in Figure 6. The shaded zone is a ghost zone. B is a node on the ghost zone and A is a corresponding node on the adjacent interior zone. Nodes O and C are on the boundary line between A and B. The coordinates of B are set such that B is a mirror image of A with respect to the line OC. From the construction of Figure 6, we can see that

$$\vec{OB} + \vec{OA} = \vec{OD} = 2 * \vec{OE}$$

$$\vec{OE} = (\textit{projection of } \vec{OA} \textit{ onto } \vec{OC}) * (\textit{unit vector along } \vec{OC})$$

$$\vec{OB} + \vec{OA} = 2 * \frac{(\vec{OA} \cdot \vec{OC})}{|\vec{OC}|} * \frac{\vec{OC}}{|\vec{OC}|}$$

$$\vec{OB} = -\vec{OA} + \left(\frac{2 * (\vec{OA} \cdot \vec{OC})}{|\vec{OC}|^2} \right) * \vec{OC}$$

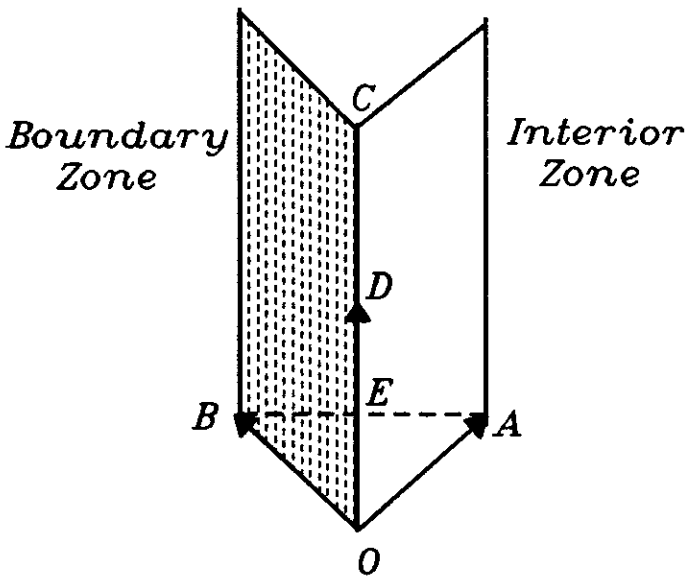

Figure 6: Coordinates on the boundary by node reflection

If (R_a, Z_a), (R_b, Z_b), (R_c, Z_c) and (R_o, Z_o) denote the coordinates of A, B, C and O respectively, then we can express the vectors as

$$\vec{OA} = \begin{bmatrix} R_a - R_o \\ Z_a - Z_o \end{bmatrix} \qquad \vec{OC} = \begin{bmatrix} R_c - R_o \\ Z_c - Z_o \end{bmatrix} \qquad \vec{OB} = \begin{bmatrix} R_b - R_o \\ Z_b - Z_o \end{bmatrix}$$

Substituting for the difference vectors and rearranging the terms, we can define the following function to compute the coordinates of the node on the boundary zone:

$$\begin{aligned} &\textit{boundary\_position}\ (R_o, Z_o)\ (R_c, Z_c)\ (R_a, Z_a) = \\ &\{\ \omega = 2 * \frac{(R_a - R_o) * (R_c - R_o) + (Z_a - Z_o) * (Z_c - Z_o)}{(R_c - R_o)^2 + (Z_c - Z_o)^2}\ ; \\ &\quad R_b = R_o - (R_a - R_o) + \omega * (R_c - R_o)\ ; \\ &\quad Z_b = Z_o - (Z_a - Z_o) + \omega * (Z_c - Z_o)\ ; \\ &\textit{in}\ \ (R_b,\ Z_b)\ \}; \end{aligned} \tag{21}$$

The manner in which the nodes O, C and A are chosen is different for each boundary node B. The circles around Figure 7 schematically show the patterns in which nodes O, C and A are chosen to compute the coordinates of each B. For example, consider the top left corner pattern. Node B is the grid point *(kmin-1,lmin-1)*. Corresponding nodes O, C and A are the grid points *(kmin,lmin)*, *(kmin,lmin-1)*, *(kmin+1,lmin-1)* respectively. We can abstract this relationship using directions: If *(k,l)* are the indices of node B, then the indices of nodes O, C and A are given by *southeast (k,l)*, *south (k,l)* and *farsouth (k,l)* respectively. Thus, the three functions *southeast*, *south* and *farsouth* completely specify how reflection should be done to obtain the coordinates of the top left corner node. In general, for each boundary node we associate three functions, f_o, f_c and f_a, which characterize the reflection pattern for that node. When they are applied to the indices of a boundary node B, they give the indices of the corresponding nodes, O, C and A. Using them the boundary coordinates are defined by:

$$\begin{aligned} &\textit{reflect}\ (r, z)\ f_o\ f_c\ f_a\ \textit{node} = \\ &\quad (r[f_o\ \textit{node}],\ z[f_o\ \textit{node}]),\ (r[f_c\ \textit{node}],\ z[f_c\ \textit{node}]),\ (r[f_p\ \textit{node}],\ z[f_p\ \textit{node}]); \end{aligned} \tag{22}$$

Thus for example, the coordinates of node B in the top left pattern in Figure 7 are given by

reflect (r,z) southeast south farsouth (kmin-1,lmin-1)

From Figure 7 we can see that there are 12 different patterns for boundary reflection (4 on the north boundary, 4 on the south bondary, 2 on east boundary and 2 on west boundary). The new positional coordinates can be defined as follows. The name *using* is defined below just for convenience to avoid repeating the partial application over and over. Its arguments suggest that the newly computed coordinates must be used in the reflection.

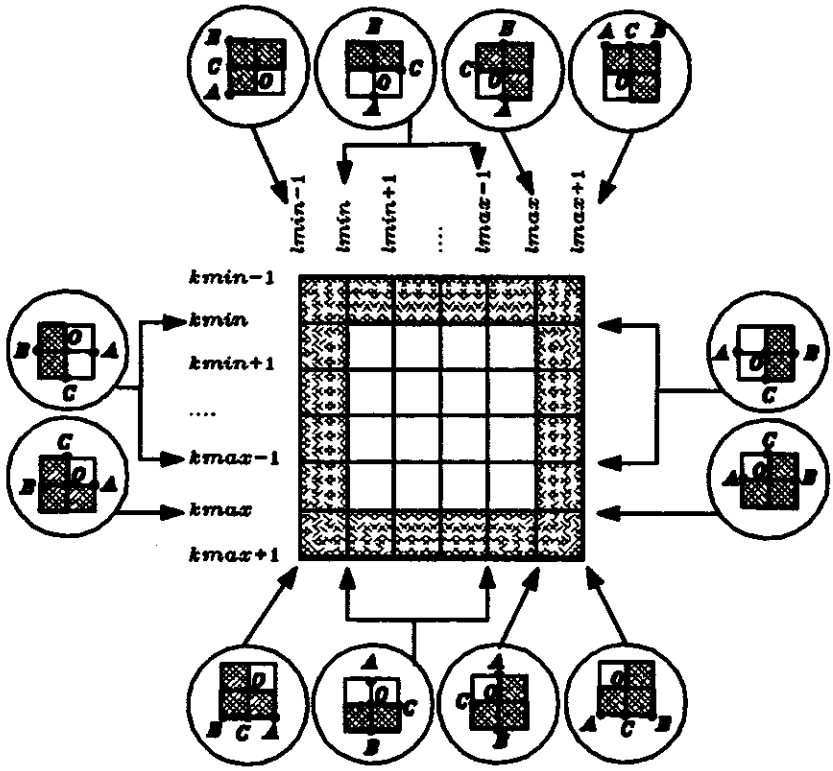

Figure 7: Reflection patterns for computing boundary node coordinates

```
using = reflect  new_x ;                                                    (23)
new_x = make_2_matrices_13_ranges  all_nodes
    ( (interior_nodes,                          position),
    % on the north boundary
      (((kmin-1,kmin-1), (lmin-1,lmin-1)),      using   southeast   south     farsouth),
      (((kmin-1,kmin-1), (lmin,lmax-1)),        using   south       southeast farsouth),
      (((kmin-1,kmin-1), (lmax,lmax)),          using   south       southwest farsouth),
      (((kmin-1,kmin-1), (lmax+1,lmax+1)),      using   southwest   west      farwest),
    % on the south boundary
      (((kmax+1,kmax+1), (lmin-1,lmin-1)),      using   northeast   east      fareast),
      (((kmax+1,kmax+1), (lmin,lmax-1)),        using   north       northeast farnorth),
      (((kmax+1,kmax+1), (lmax,lmax)),          using   north       northwest farnorth),
      (((kmax+1,kmax+1), (lmax+1,lmax+1)),      using   northwest   west      farwest),
    % on the east boundary
      (((kmin,kmax-1), (lmax+1,lmax+1)),        using   west        southwest farwest),
      (((kmax,kmax), (lmax+1,lmax+1)),          using   west        northwest farwest),
    % on the west boundary
      (((kmin,kmax-1), (lmin-1,lmin-1)),        using   east        southeast fareast),
      (((kmax,kmax), (lmin-1,lmin-1)),          using   east        northeast fareast) );
```

Notice that the above definitions are mutually recursive, as *using* and *new_x* are used in each other's definitions.

3.4. Area, Volume and Density

The area of each interior zone is approximated by the two shaded triangles shown in Figure 8. We know that the area of the triangle is half the magnitude of the cross product of the two vectors representing two sides of the triangle. The magnitude of the cross product is the determinant of the matrix formed by the two vectors. Thus for example, the area of the lower triangle in Figure 8 is given by half of the determinant

$$\begin{vmatrix} (R[k,l] - R[k,l-1]) & (R[k-1,l] - R[k,l]) \\ (Z[k,l] - Z[k,l-1]) & (Z[k-1,l] - Z[k,l]) \end{vmatrix}$$

The function to compute the area and volume of a given zone is shown below. Given the coordinates of the three vertices, the function, *triangle* computes the area of the triangle. The next three equations compute the area as the sum of the areas of the two triangles. The subsequent equations compute the zone volume *per radian*, when the zone revolves around the Z-axis. Again this is the sum of the volumes generated when the two triangles revolve. The volume generated by a triangle, is approximated by $2\pi \bar{r} \Delta$, where $\bar{r}$ is the average of the R-coordinates of the three vertices of the triangle and Δ is the area of the triangle.

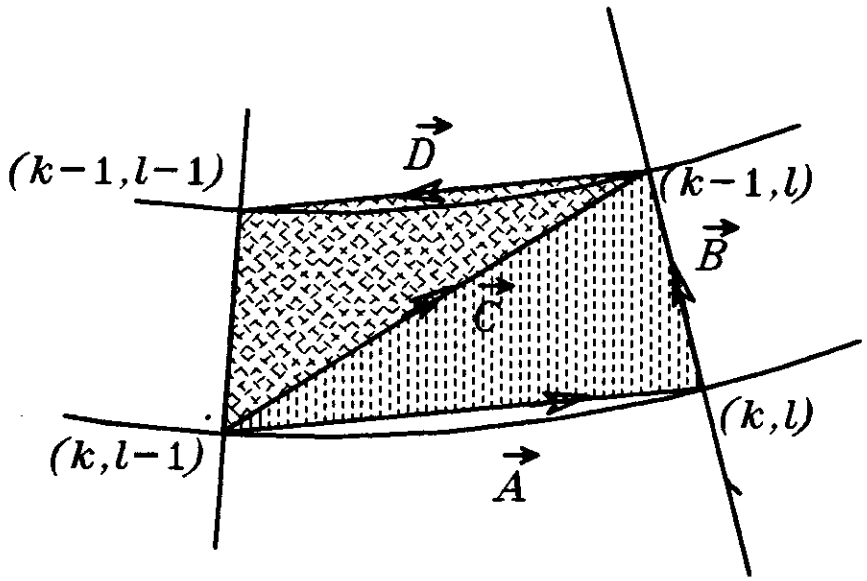

Figure 8: Approximation of zone area and volume

```
area_volume (r, z) node =                                                    (24)
    { difference p1 p2 = (r[p2] - r[p1], z[p2] - z[p1])
      triangle v1 v2 v3 = 0.5 * (determinant (difference v1 v2) (difference v2 v3));

      lower_area = triangle (west node) node (north node);
      upper_area = triangle (west node) (north node) (northwest node);
      area = upper_area + lower_area;

      lower_radius = (r[west node] + r[node] + r[north node]) / 3;
      upper_radius = (r[west node] + r[north node] + r[northwest node]) / 3;
      lower_volume = lower_area * lower_radius;
      upper_volume = upper_area * upper_radius;
      volume = upper_volume + lower_volume;

    in area, volume };
```

The density of a zone is computed from the mass conservation equation: $\frac{d(\rho V)}{dt} = 0$. The new density is given by *density * volume / new_volume*. For a ghost zone, all the three quantities, area, volume and density are set the same as those of the adjoining zone. Hence we get:

```
area_volume_density node =                                                   (25)
    { area, volume = area_volume new_x node;
      density = ρ[node] * s[node] / volume;
    in area, volume, density };
```

```
new_αsρ = new_α, new_s, new_ρ;                                               (26)
new_α, new_s, new_ρ = make_3_matrices_5_ranges all_zones

    ( (interior_zones,          area_volume_density),
      (north_boundary_zones,    select_3 south new_αsρ),
      (south_boundary_zones,    select_3 north new_αsρ),
      (west_boundary_zones,     select_3 east new_αsρ),
      (east_boundary_zones,     select_3 west new_αsρ) );
```

3.5. Artificial Viscosity

The artificial viscosity is given by the formula

$$\frac{1}{4}\rho C_0^2 (4\overline{\delta u}^2 + 4\overline{\Delta u}^2) + \frac{1}{2}\rho C_A C_l \sqrt{4\overline{\delta u}^2 + 4\overline{\Delta u}^2}$$

where $C_0^2 = 1.5$, $C_1 = 0.5$, C_A is the speed of sound in a zone given by γp rho / and other quantities

are as defined by the following equations:

```
viscosity node =                                                    (27)
    { mean_k_difference f = 0.5 * ( (f[ node ] - f[ north node ]) +
                                    (f[ west node ] - f[ northwest node ]) );
      mean_l_difference f = 0.5 * ( (f[ node ] - f[ west node ]) +
                                    (f[ north node ] - f[ northwest node ]) );
      Δr = mean_k_difference new_r;        Δz = mean_k_difference new_z;
      Δu = mean_k_difference new_u;        Δw = mean_k_difference new_w;
      δr = mean_l_difference new_r;        δz = mean_l_difference new_z;
      δu = mean_l_difference new_u;        δw = mean_l_difference new_w;
      ζ = Δr * Δr + Δz * Δz ;              η = δr * δr + δz * δz ;
      δa = determinant (Δr, Δz) (δu, δw) ; Δa = determinant (Δu, Δw) (δr, δz) ;
      δb = if δa < 0 then δa * δa else 0 ; Δb = if Δa < 0 then Δa * Δa else 0 ;
      δu~ = δb / ζ ;                       Δu~ = Δb / η ;
      d = δu~ + Δu~ ;                      γ = 1.666 ;
      c_a = γ * p [ node ] / new_ρ [ node ] ;  c = 1.5 * d + 0.5 * c_a * (sqrt d) ;
    in new_ρ [ node ] * c };
```

As before, we simply copy the viscosity for the ghost zones from the neighboring zones and get the following matrix for viscosity:

```
new_q = make_matrix_5_ranges all_zones                              (28)
        ( (interior_zones,          viscosity),
          (north_boundary_zones,    select south new_q),
          (south_boundary_zones,    select north new_q),
          (west_boundary_zones,     select east new_q),
          (east_boundary_zones,     select west new_q) );
```

3.6. Pressure, Temperature and Energy

The three quantities p, θ and ε are related by the following equations:

$$\frac{d\varepsilon}{dt} + (p+q)\frac{d\tau}{dt} = 0 \qquad \varepsilon = \sum_{i=0}^{2}\sum_{j=0}^{2} A_{ij}\,\rho^i\,\theta^j \qquad p = \sum_{i=0}^{2}\sum_{j=0}^{2} B_{ij}\,\rho^i\,\theta^j$$

where τ is the reciprocal of ρ and A_{ij} and B_{ij} are coefficients of two polynomials in ρ and θ. These

are used to compute energy and pressure, respectively. Given initial p, q, ε, ρ, the values of ε, θ, p are updated as follows:

1. Energy is updated using the difference approximation to the first equation, namely, $\tilde{\varepsilon} - \varepsilon = -(p+q) * d\tau$.
2. Then the new temperature is obtained by rewriting the second equation as

$$g(\theta) = \varepsilon - \sum_{i=0}^{2} \sum_{j=0}^{2} A_{ij} \rho^i \theta^j = 0$$

The new temperature is the zero of the above equation, which is obtained by the Newton-Raphson iterative method where the successive increments to θ are computed from

$$\theta_{n+1} = \theta_n - \frac{g(\theta_n)}{g'(\theta_n)} \qquad g'(\theta) = \frac{g(\theta + increment) - g(\theta)}{increment}$$

3. Finally, by substituting the above value of temperature, the polynomial in the third equation is evaluated to yield the new pressure.

The following function performs this computation at any node:

```
etp_triple q ρ dτ ε p θ =                                          (29)
    { ε̃ = ε − (p + q) * dτ ;
      θ̃ = inverse_polynomial EP ε̃ ρ θ;
      p̃ = polynomial PP ρ θ̃ ;
    in (ε̃, θ̃, p̃) };
```

where EP and PP are constants that yield the coefficients of the two polynomials. The functions *polynomial* and *inverse_polynomial* and the constants EP and PP are described later. For better accuracy, a predictor-corrector method is used, so that these three quantities are estimated first and recomputed as defined by the following function:

```
energy_temperature_pressure node =                                 (30)
    { h = etp_triple new_q[node] new_ρ[node] (1 / new_ρ[node] − 1 / ρ[node]);
      ε̃, θ̃, p̃ = h ε[node] p[node] θ[node];
      p̄ = (p[node] + p̃) / 2 ;
      ε̂, θ̂, p̂ = h ε̃ p̄ θ̃;
    in (ε̂, θ̂, p̂) }
```

For the boundary zones, the pressure is copied from the adjacent interior nodes, as before. The internal energy of boundary zones is never used, but we assign it zero value (so that we do not have to write separate equations for them - it does not have to be this way). The boundary temperature is kept constant, to simulate the effect of a constant source (so that no heat is lost due to finite boundary being simulated). Hence, we have the following definitions:

```
boundary_etp direction node =                                          (31)
    (0, constant_heat_source, new_p [ direction node ] );

new_ε, interim_θ, new_p = make_3_matrices_5_ranges all_zones           (32)
        ( (interior_zones,          energy_temperature_pressure),
          (north_boundary_zones,    boundary_etp south),
          (south_boundary_zones,    boundary_etp north),
          (west_boundary_zones,     boundary_etp east),
          (east_boundary_zones,     boundary_etp west) );
```

3.6.1. Polynomials

The polynomials are piecewise continuous and hence a set of polynomials are maintained. The ρ-θ plane is divided into regions as shown in Figure 9 and each region is associated with a separate polynomial. The regions are represented by two tables *ρ-table* and *θ-table*. For instance, given a table, we can determine the region of a given value by the function shown in definition (33):

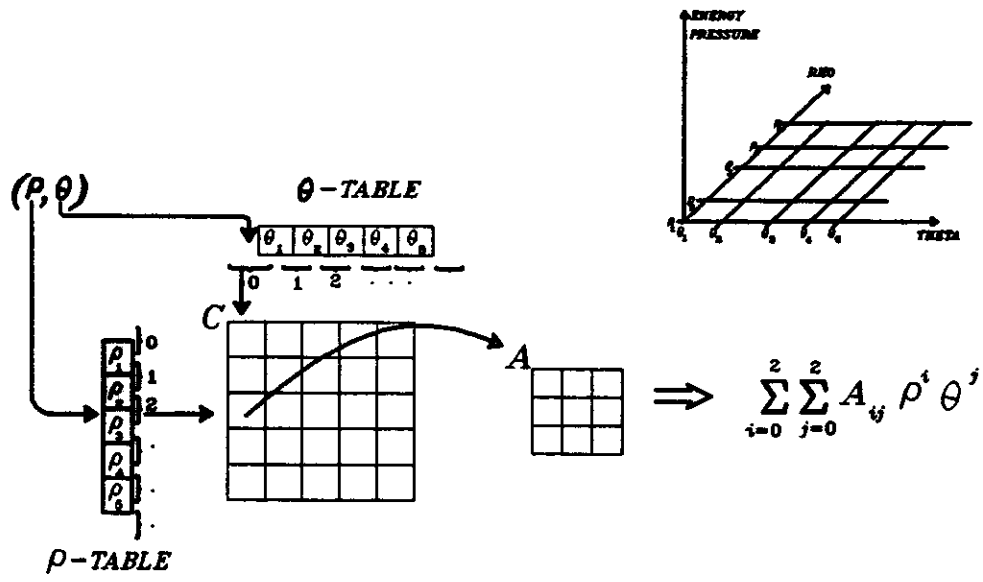

Figure 9: Representation of polynomials

```
region table value =                                                    (33)
    { low, high = bounds table;
    in if value > table [ high ] then high + 1
          else { while (high > low) and (value ≤ table [ high − 1 ]) do
                    next high = high − 1
                finally high } };
```

Here *bounds* is a function that yields the lower and upper bounds of the argument table. Using this representation, the polynomials for all the regions in the ρ-θ plane can be obtained from the tuple (*C*, *ρ-table*, *θ-table*), where *C* is a matrix of matrices, so that $C[i,j]$ gives the coefficient matrix, *A*, corresponding to the *(i,j)-th* region in the ρ-θ plane. Thus, *EP* and *PP* are constant tuples of the above form, giving the corresponding values for the polynomials for energy and pressure respectively. A polynomial and its inverse can be computed as follows:

```
polynomial (C, ρ_table, θ_table) ρ θ =                                  (34)
    { ρ_selector = region ρ_table ρ;
      θ_selector = region θ_table θ;
      A = C [ ρ_selector, θ_selector ];
      powers = ((0, 2), (0, 2));
    in accumulate ((powers, {fun (i, j) A [ i, j ] * ρ^i * θ^j} )) };
```

```
inverse_polynomial ω ε ρ θ =                                            (35)
    { g θ = ε − (polynomial ω ρ θ);
      error = g θ;    tolerance = 1.0E − 6;    increment = 1.0E − 6;
    in { while error > tolerance do
            next error = g (θ + increment);
            next θ = θ − error * increment / (next error − error);
         finally θ } };
```

3.7. Heat Conduction

The heat conduction equation in cylindrical coordinates is

$$\frac{\partial u}{\partial \theta} \frac{\partial \theta}{\partial t} = \frac{1}{\rho} \left[\frac{1}{r} \frac{\partial}{\partial r} \left(r \kappa \frac{\partial \theta}{\partial r} \right) + \frac{\partial}{\partial z} \left(\kappa \frac{\partial \theta}{\partial z} \right) \right]$$

where κ (kappa) is the heat conductivity. Transforming into Lagrangian coordinates and omitting the cross-derivative terms we get

$$\frac{\partial e}{\partial \theta}\ \frac{\partial \theta}{\partial t} = \frac{1}{\rho\ r\ \alpha}\left[\frac{\partial}{\partial k}\left(\frac{r\ |\delta \vec{x}\ |^2\ \kappa}{\alpha}\ \frac{\partial \theta}{\partial k}\right) + \frac{\partial}{\partial l}\left(\frac{r\ |\Delta \vec{x}\ |^2\ \kappa}{\alpha}\ \frac{\partial \theta}{\partial l}\right)\right]$$

where $|\Delta \vec{x}\ |^2 = \frac{\partial \vec{x}}{\partial k} \cdot \frac{\partial \vec{x}}{\partial k}$, $|\delta \vec{x}\ |^2 = \frac{\partial \vec{x}}{\partial l} \cdot \frac{\partial \vec{x}}{\partial l}$

and $\vec{x}$ *is the position vector and* α *is the area-jacobian of the zone.*

To obtain a difference formulation, first we replace the derivative $\partial\theta / \partial t$ with $(\hat{\theta} - \theta) / \delta t$, where δt is the time step and θ is the new value of θ. New temperature is computed in two steps. The first step is called *k-sweep* in which l is kept constant and k is varied. We use the notation, f_k to denote the value of the function f at (k,l), l being a constant in the *k-sweep*. Thus, for the *k-sweep*, in the above equation the second term (the derivative with respect to l) becomes zero. Assuming that the *k-difference* ∂k approximates to 1, we get:

$$\left(\frac{\partial e}{\partial \theta}\right)_k\ \frac{\hat{\theta}_k - \theta_k}{\delta t} = \frac{1}{\rho\ r\ \alpha}\left[\left(\frac{r\ |\Delta \vec{x}\ |^2\ \kappa}{\alpha}\right)_k (\hat{\theta}_{k+1} - \hat{\theta}_k) - \left(\frac{r\ |\Delta \vec{x}\ |^2\ \kappa}{\alpha}\right)_{k-1} (\hat{\theta}_k - \hat{\theta}_{k-1})\right]$$

By making the following abbreviations and substitutions,

$m = \rho\ r\ \alpha$ = *density * radius * area = zonemass per radian*

$\bar{r}_k = \left(\frac{r\ |\Delta \vec{x}\ |^2\ \kappa}{\alpha}\right)_k$ = *computed as described later*

$\sigma = \frac{m}{\delta t}\left(\frac{\partial e}{\partial \theta}\right)_k = \frac{\text{mass} * \text{conductivity}}{\delta t}$, *assumed constant for both k-sweep and l-sweep*

we obtain $\sigma\ (\hat{\theta}_k - \theta_k) = \bar{r}_k\ (\hat{\theta}_{k+1} - \hat{\theta}_k) - \bar{r}_{k-1}\ (\hat{\theta}_k - \hat{\theta}_{k-1})$

Substituting for $\hat{\theta}_{k-1}$, a solution of the form $\hat{\theta}_{k-1} = A_{k-1} * \hat{\theta}_k + B_{k-1}$, we get the equation

$$\sigma\ (\hat{\theta}_k - \theta_k) = \bar{r}_k\ (\hat{\theta}_{k+1} - \hat{\theta}_k) - \bar{r}_{k-1}\ (\hat{\theta}_k - (A_{k-1} * \hat{\theta}_k + B_{k-1}))$$

Rearranging the terms and solving for the coefficients, A and B we get, $\forall\ 1 < k \leq kmax$,

$$A_k = \frac{\bar{r}_k}{\sigma + \bar{r}_k + \bar{r}_{k-1}\ (1 - A_{k-1})},\quad B_k = \frac{\bar{r}_{k-1} B_{k-1} + \sigma\ \theta_k}{\sigma + \bar{r}_k + \bar{r}_{k-1} * (1 - A_{k-1})},\quad A_1 = 0,\quad B_1 = \text{old value of } \theta$$

The term, $\bar{r}_k$, is expressed as a product of two terms $\overline{r1}_k * \overline{r2}_k$. The first term, $\overline{r1}_k$ is defined through an averaging process at each interface in terms of a matrix *cc* as shown below. The second term is the cross sectional area of the interface between two zones and is expressed as the product of the mean radius at the center of the interface and the square of the length of the interface segment, as shown below:

$$\bar{r}_k = \left(\frac{r \, |\Delta \vec{x}|^2 \kappa}{\alpha} \right)_k = \left(\frac{\kappa}{\alpha}\right) \left(r \, |\Delta \vec{x}|^2\right) = \overline{r1}_k * \overline{r2}_k$$

$$\kappa[node] = .0001 \, \theta[node]^{\frac{5}{2}}, \qquad cc[node] = \frac{\kappa[node]}{\alpha[node]}$$

$$\overline{r1}_k = \frac{2 * cc_k * cc_{k+1}}{cc_k + cc_{k+1}}, \qquad \overline{r2}_k = \frac{r_k + r_{k-1}}{2} * \left((r_k - r_{k-1})^2 + (z_k - z_{k-1})^2 \right), \quad \forall l$$

The expressions for the *l-sweep* are similar, except that subscripts *k* should be replaced by *l*. Figure 10 schematically shows the dependence of the various quantities in the computation of the final temperature. σ and *cc* are pointwise computations and are used in both the sweeps. Hence, we first compute them and store them in matrices as shown by definitions (36) thru (38).

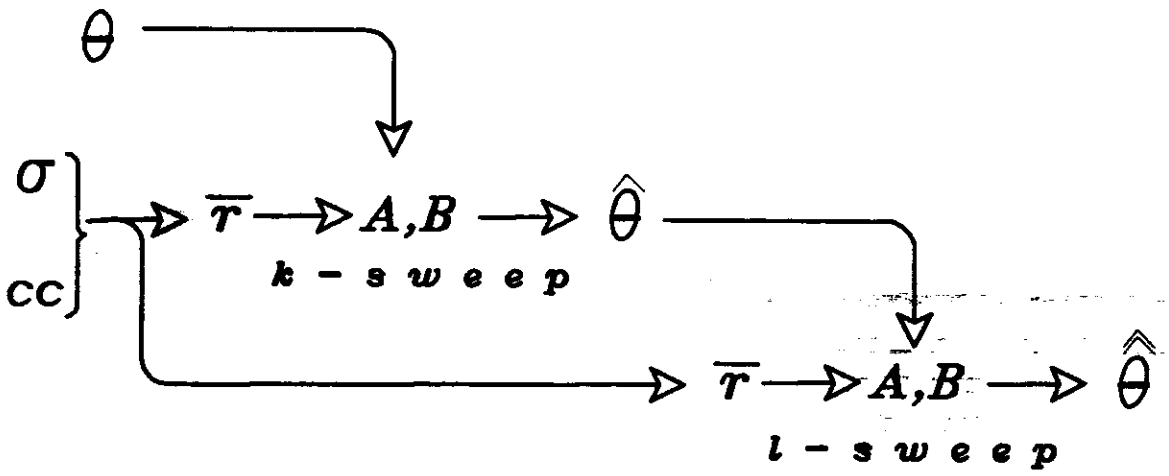

Figure 10: Quantities computed in heat conduction

$$\sigma = make\_matrix\ \ interior\_zones\ \ \{\textbf{fun}\ \ node\ \ \ mass[\,node\,] * conductivity[\,node\,] / \delta t\};\quad (36)$$

$$interior\_cc\ \ node = .0001 * interim\_\theta[\,node\,]^{5/2} / new\_\alpha[\,node\,]\ ;\quad (37)$$

$$\begin{aligned} cc = make\_matrix\_5\_ranges\ \ all\_zones & \quad (38)\\ (\ (interior\_zones, & \quad interior\_cc),\\ (north\_boundary\_zones, & \quad select\ south\ \ cc),\\ (south\_boundary\_zones, & \quad select\ north\ \ cc),\\ (west\_boundary\_zones, & \quad select\ east\ \ cc),\\ (east\_boundary\_zones, & \quad select\ west\ \ cc)\ \ );\end{aligned}$$

To abstract the repeated computation of $\bar{r}, A, B, \theta$, consider the expressions for $\bar{r}$ in the two sweeps:

k-sweep:
$$\frac{r[k,l]+r[k-1,l]}{2} * ((r[k,l]-r[k-1,l])^2+(z[k,l]-z[k-1,l])^2)\ * \ some\_mean\_of\ \ (cc[k,l], cc[k,l+1])$$

l-sweep:
$$\frac{r[k,l]+r[k,l-1]}{2} * ((r[k,l]-r[k,l-1])^2+(z[k,l]-z[k,l-1])^2)\ * \ some\_mean\_of\ \ (cc[k,l], cc[k+1,l])$$

The expressions are computed for each node and involve the position coordinates of the node and its *preceding* node. Also the values of *cc* of this node and *adjacent* node are used. (Later we use the attributes of a *successor* node also.) The notions of *preceding, succeeding* and *adjacent* are relative to the direction of sweep as shown in Figure 11. Let *pred, succ* and *adj* denote the functions that give the indices of the preceding, succeeding and adjacent positions of a node, respectively, in a given sweep.

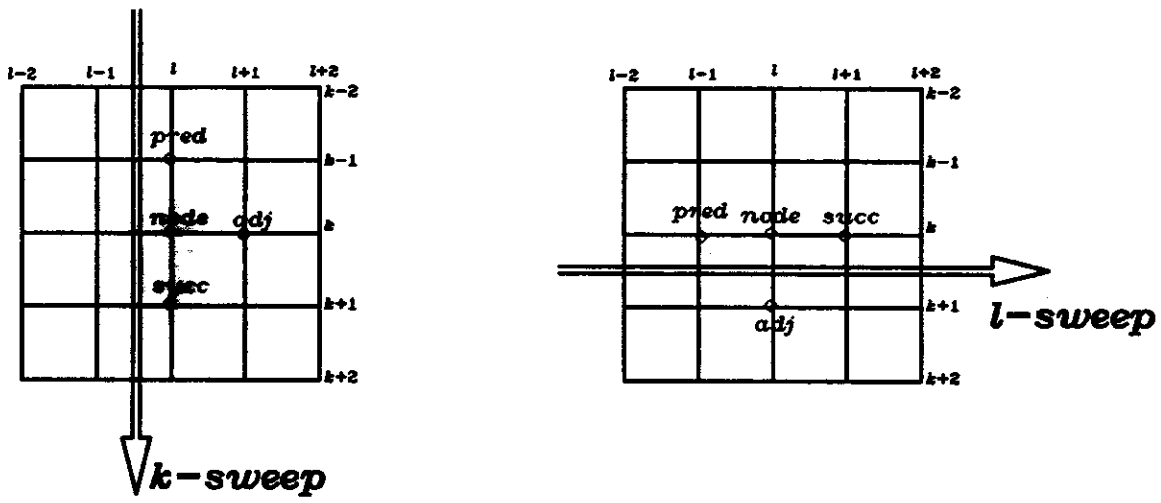

Figure 11: Abstracting the directional features of a sweep

For example, in the k-sweep, $pred\ (i,j) = (i-1,j)$. Using these functions, we can write the following expression for $\bar{r}$ common to both sweeps:

$$\frac{r[node] + r[pred\ \ node]}{2} * ((r[node] - r[pred\ \ node])^2 + (z[node] - z[pred\ \ node])^2) * some\_mean\_of\ \ (cc[node],\ cc[adj\ \ node])$$

Using this convention, conductance in any sweep is defined as:

```
conductance sweep node =                                              (39)
    { pred, succ, adj = sweep;
      cross_section = 0.5 * (r[node] + r[pred node]) *
        ( (r[node] - r[pred node])^2 + (z[node] - z[pred node])^2 );
      c1 = cc[node];
      c2 = cc[adj node];
      specific_conductivity = 2 * c1 * c2 / (c1 + c2);
    in cross_section * specific_conductivity }
```

Now the quantities $\bar{r}, A, B$ and θ for each sweep and the temperatures can be computed as follows.

```
temperature sweep old_θ =                                             (40)
    { pred, succ, adj = sweep;
      coefficients node =
          { d = σ[node] + r̄[node] + r̄[pred node] * (1 - A[pred node]);
            n1 = r̄[node];
            n2 = r̄[pred node] * B[pred node] + σ[node] * old_θ[node];
          in (n1 / d, n2 / d) }

      r̄ = make_matrix_5_ranges all_zones
          ( (interior_zones,          conductance sweep),
            (north_boundary_zones,    {constant 0}),
            (south_boundary_zones,    {constant 0}),
            (west_boundary_zones,     {constant 0}),
            (east_boundary_zones,     {constant 0}) );
      A, B = make_2_matrices_5_ranges all_zones
          ( (interior_zones,          coefficients),
            (north_boundary_zones,    {fun node (0, old_θ[node])} ),
            (south_boundary_zones,    {fun node (0, old_θ[node])} ),
            (west_boundary_zones,     {fun node (0, old_θ[node])} ),
            (east_boundary_zones,     {fun node (0, old_θ[node])} ) );
```

```
interior_temperature node = A [ node ] * θ̂ [ succ node ] + B [ node ] ;
θ̂ = make_matrix_5_ranges all_zones
    ( (interior_zones,          interior_temperature),
      (north_boundary_zones,    {constant heat_source}),
      (south_boundary_zones,    {constant heat_source}),
      (west_boundary_zones,     {constant heat_source}),
      (east_boundary_zones,     {constant heat_source}) );
in  θ̂, r̄ }
```

```
k_sweep = north, south, east;                                        (41)
l_sweep = west, east, south;
local_θ, r̄_k = temperature k_sweep interim_θ;
new_θ, r̄_l = temperature l_sweep local_θ;
```

3.8. Energy Balance

The book keeping of energy is to check the balance of the internal and kinetic energies in the interior zones and the work done and heat lost at the boundaries. At each time step, the balance is calculated to ensure that the error is within tolerance limits. We now describe the computation of each of these quantities.

3.8.1. Internal energy

Internal energy is given by the formula: $\sum_{\forall\ zones}$ (*energy* * *mass*). This is computed by

```
internal_energy node = mass [ node ] * new_ε [ node ];               (42)
EI = accumulate interior_zones internal_energy;
```

3.8.2. Kinetic Energy

Kinetic energy at all interior nodes is given by: $\sum_{\forall\ nodes} \frac{1}{2}\ \overline{M}\ |\ \vec{v}\ |^2$, where $\overline{M}$ is the average mass of 4 neighboring zones around the node. This is computed by

```
kinetic_energy node =                                                (43)
    { m̄ = 0.25 * (mass [ zone_a node ] + mass [ zone_b node ] +
                  mass [ zone_c node ] + mass [ zone_d node ] );
      v_square = new_u [ node ]^2 + new_w [ node ]^2 ;
    in 0.5 * m̄ * v_square }

EK = accumulate interior_nodes kinetic_energy;                       (44)
```

3.8.3. Boundary work

The work done on the boundary is computed by considering the movement of the boundary line. For example, Figure 12 shows the boundary line segment *12*. The shaded area (the parallelogram formed by the average velocity vector and position difference vector) shows the amount by which it moved during a time step. The work done by its movement is the product of the force and the amount of its displacement. The calculations are summarized below:

$$work = force * \delta volume, \quad force = pressure + artificial\ viscosity$$

$$\delta volume = \delta area * average\ radius\ of\ revolution = \delta area * \frac{r_1 + r_2}{2}$$

$$\delta area \cong boundary\ line * velocity * time = \delta t \left| \vec{x}_1 - \vec{x}_2 \times \frac{\vec{v}_1 + \vec{v}_2}{2} \right| = \frac{\delta t}{2} \begin{vmatrix} r_1 - r_2 & u_1 + u_2 \\ z_1 - z_2 & w_1 + w_2 \end{vmatrix}$$

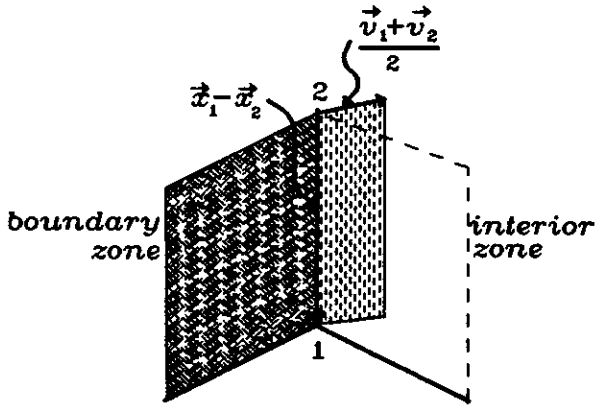

Figure 12: Displacement at boundary

The work done must be accumulated for all the difference vectors shown in Figure 13. We first define the index ranges for this traversal as follows:

$$\begin{aligned} &north\_work\_boundary = ((kmin, kmin), (lmin + 1, lmax)); \\ &east\_work\_boundary = ((kmin + 1, kmax), (lmax, lmax)); \\ &south\_work\_boundary = ((kmax, kmax), (lmin + 1, lmax)); \\ &west\_work\_boundary = ((kmin + 1, kmax), (lmin + 1, lmin + 1)); \end{aligned} \tag{45}$$

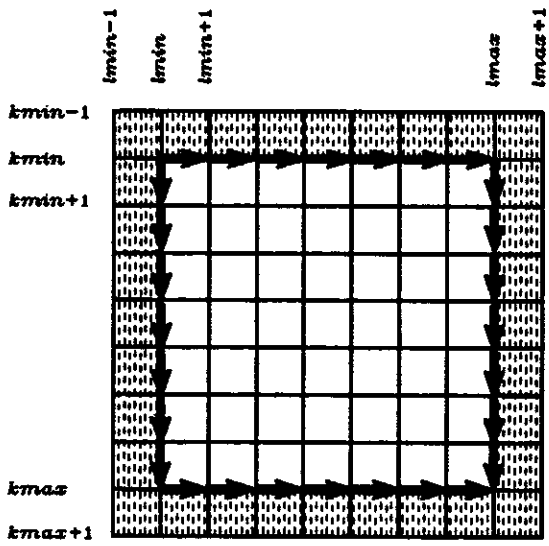

Figure 13: Work done at the boundary

The work done at each boundary line is given by

```
work_done pred node =                                                    (46)
    { force = 0.5 * (new_p[node] + new_p[pred node] +
                     new_q[node] + new_q[pred node]);
      radius = 0.5 * (new_r[node] + new_r[pred node]);
      area = 0.5 * determinant
                   (new_r[node] - new_r[pred node],
                   new_z[node] - new_z[pred node])
                   (new_u[node] + new_u[pred node],
                   new_w[node] + new_w[pred node]);
    in force * radius * area * δt };
```

The accumulated work done is:

```
W = accumulate_4_ranges                                                  (47)
        ( (north_work_boundary,      work_done west),
          (south_work_boundary,      work_done west),
          (west_work_boundary,       work_done north),
          (east_work_boundary,       work_done north) );
```

3.8.4. Boundary heat

The amount of heat flowing across the boundary is given by

$$Boundary\ Heat\ Flow = H = \delta t * \sum_{boundary} conductance * \delta\theta.$$

The heat flows from the outer layer of the interior zones to the ghost zones as shown in Figure 14. The amount of heat flow at each junction is the product of the temperature difference between the two zones and the conductance of the interior zone. Hence we need the conductance matrix, $\bar{r}$, computed in *k-sweep* and *l-sweep* for the corresponding zones. For better efficiency, one should really store the two $\bar{r}$ matrices and use them. But here we ignore this and recompute the conductance on the boundary. Since the heat flows along the arrows indicated in Figure 14, we must aggregate the heat flow over the zones in the inner shaded band, whose southeast corners are indicated by the bullets and squares in Figure 14. First, we define the following index ranges for these zones:

(48)

$north\_heat\_boundary = ((kmin + 1, kmin + 1),\ (lmin + 1, lmax));$
$east\_heat\_boundary = ((kmin + 2, kmax),\ (lmax, lmax));$
$south\_heat\_boundary = ((kmax, kmax),\ (lmin + 2, lmax - 1));$
$west\_heat\_boundary = ((kmin + 2, kmax),\ (lmin + 1, lmin + 1));$

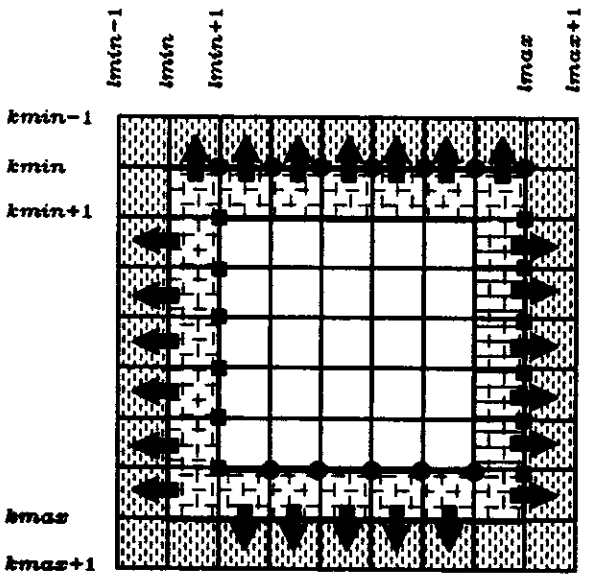

Figure 14: Heat flow across the boundary

For each zone, we also need the direction of heat flow, so that the temperature difference can be calculated in the appropriate direction. The following function defines the heat flow for a given zone:

```
heat_flow r̄ direction node =                                          (49)
    δt * (new_θ[node] − new_θ[direction node]) * r̄[node];
```

The accumulated heat flow is obtained by:

```
H = accumulate_4_ranges                                                (50)
        ( (north_heat_boundary,     heat_flow r̄_k north),
          (east_heat_boundary,      heat_flow r̄_l east),
          (south_heat_boundary,     heat_flow r̄_k south),
          (west_heat_boundary,      heat_flow r̄_l west) );
```

Finally we can define the energy check error as:

$$error = EI + EK - W - H; \tag{51}$$

3.9. Time step calculation

The time step increment for the next iteration is controlled by three criteria described below.

Courant condition: A sound signal must not be able to cross any zone in one time step. To ensure this, the next time step is limited by half of the time taken by sound to cross any zone. The time taken for a sound signal to cross a zone is approximated by the ratio of the average width of the zone to the speed of sound given by:

```
courant_delta node =                                                   (52)
    { thrust = new_p[node] / new_ρ[node];
      speed_square = thrust * (1 + thrust / new_ε[node]);
      sound_speed = sqrt speed_square;
      Δr = 0.5 * ( (new_r[node] − new_r[west node]) +
                                  (new_r[north node] − new_r[northwest node]) );
      δr = 0.5 * ( (new_r[node] − new_r[north node])
                                  (new_r[west node] − new_r[northwest node]) );
      average_length = (sqrt Δr² + δr²);
    in 0.5 * new_α[node] / sound_speed / average_length };

δt_hydro = minimum interior_zones courant_delta;                       (53)
```

Relative change in temperature: The next time step should not be larger than the relative temperature change in any zone since the last time step. To obtain the maximum relative change in temperature we define:

```
relative_change_in_temperature node =                                    (54)
    abs (θ [ node ] − new_θ [ node ]) / θ [ node ];
```

```
δt_conduct = maximum interior_zones relative_change_in_temperature;      (55)
```

Finally, the next time step cannot exceed a predefined maximum value. Hence we define:

```
new_δt = min (min δt_hydro δt_conduct) δt_maximum;                       (56)
```

4. Concluding Remarks

Ideally, a high-level language should provide a way of writing abstractions which are as close to the problem domain as possible, as well as facilitate efficient implementations of these abstractions lest a user try to "get underneath" the abstractions. With the advent of parallel machines, a language such as *Fortran* fails on both counts. It was never very good for expressing high-level abstractions and, because it forces the user to specify a sequential order of evaluation, it also makes it very difficult to compile good code for a parallel machine. In the latter deficiency, Fortran is not alone; all high-level languages in wide spread use today force the user to over-specify the algorithm. *Functional* and other *declarative* languages allegedly offer relief on both counts. The use of *higher order functions*, including the free use of *curried* forms, can dramatically raise the level of programming. In addition, such languages often have straight-forward operational semantics which admits tremendous opportunities for parallel execution. Programs in *declarative* languages, thus eliminate the problem of "detecting parallelism"; however, the problem of managing resources for parallel execution remains. In this report we have examined the first part of this claim by writing an application known as the SIMPLE code in a language called *Id Nouveau*. The issues of parallelism will be examined in a companion report to be published later.

We have presented a high-level description of the algorithms used in the hydrodynamics problem, called SIMPLE, as described in [4]. Corresponding *Id Nouveau* program fragments are derived from the mathematical descriptions. A few basic abstractions are central to the ease with which the resulting program can be expressed. One of these is, *make_n_matrices_k_ranges*, which is a generalization of the abstraction to create a matrix A by specifying k functions, f_m, and associated ranges, r_m, such that $A[i,j]$ is $f_m(i,j)$, if (i,j) falls in range r_m. Other heavily used abstractions are *north*, *south*, *zone_a*, *etc.*, to refer to the neighboring nodes and zones, and *interior_nodes*, *boundary_zones*, *etc.* to refer to ranges of nodes and zones. These abstractions, coupled with the ability to treat functions and their *curried* forms as *first class* objects, render a very high-level program which is close to the mathematical specification of the problem. We believe that a physicist or mathematician can easily alter our program - for example, to employ an *explicit* method rather than an *implicit* method

in the heat conduction phase, or to compute the inverse of the energy and pressure polynomials by a different method, or to alter the table search strategy so that each time the search commences with the index chosen last time etc.

The main program as given in definition (14) is a complete program for the SIMPLE code. In order to run it on our current implementation, *(Id World Release April 1987)*, the following amendments had to be made.

1. The current implementation of the *Id Nouveau* compiler does not do type checking. It treats A in A [exp] as a one dimensional array and if *exp* evaluates to a tuple at run time, the program generates an error. Since it does not permit a matrix to be indexed by a tuple, we must write A [i,j] instead of A [*node*] where *node=(i,j)*. We also cannot write, for example, A [*north node*]. To overcome this shortcoming, for the time being, the neighbor abstractions have been changed slightly. For example, we define north as *north* A $(k,l) = A[k-1,l]$ instead of *north* $(k,l) = (k-1,l)$. We think this solution still captures the spirit of the neighbor abstractions.

2. The definition for the *make_n_matrices_k_ranges* abstraction is given as a schema for various specific values of n and k. Note that making n and k into parameters will result in inefficiency because the matrix names and ranges etc. will have to be made into lists and a loop program which will traverse this list and call the appropriate program fragment will have to be written. An optimizing compiler which knew how to unfold loops may be able to remove these inefficiencies. Another possibility is to include macros in *Id Nouveau* and define the *make_n_matrices_k_ranges* schema as a macro. For the time being, we have included separate definitions for the desired values of n and k.

3. The scoping of names needs some care. When a function is defined within a scope, it is usually simple, as all the names in the surrounding scope can be used without having to specify them as parameters. For example, in definitions (16) through (18), the names p, q, r, z, ρ, α are inherited from the surrounding scope, which is the main body of the loop in definition (14). Therefore we did not have to specify them explicitly as parameters in the functions *nodal_mass*, *acceleration* and *velocity*. However, this implies that we must always compile these functions along with the body of the main loop. It would be convenient to compile these functions separately. To do this, we must provide complete *top level* definitions with all the parameters completely specified. As another example, sometimes the functions may be invoked from different contexts. The function *reflect* in definition (22) illustrates the point. We specified the parameters r and z, although they are available from the surrounding context, which is the main body of the loop. However, during initialization, which is not shown in this document, we need to invoke *reflect* to create the initial boundary and it uses the initial values of r, z. Hence we explicitly λ-*lifted* all the parameters. In our program, we separately compiled and tested each of the steps in the body of the main loop (14). For this purpose, we defined each step as a function indicated in the associated comment in the definition (14). All the arguments listed there, as well as all the constants are passed as explicit parameters in the actual code.

The resulting program has 550 lines, of which the matrix-related abstractions take 120 lines. It runs successfully in *Id World*, a graph interpreting facility developed at MIT. The next question is how the performance of this program compares with equivalent programs in conventional languages running on conventional machines. This issue will be discussed in part II.

Acknowledgements: We would like to thank all the members of the Computation Structures Group of the Laboratory for Computer Science at MIT, whose numerous contributions to *Id World* helped this work. In particular, we are extremely grateful for the heroic efforts of Ken Traub in providing an ace compiler for *Id Nouveau*, without which it would have been impossible to run the SIMPLE program presented here. We are thankful to Rishiyur Nikhil for helpful comments on style and to Olaf Lubeck for clarifying some of the equations in describing the problem.

5. Bibliography

[1] Arvind and K.Ekanadham, "Future Scientific Programming on Parallel Machines", To appear in proceedings of International Conference on Supercomputing, Athens, Greece (June 8-12, 1987).

[2] Arvind, R.S.Nikhil and K.K.Pingali, "I-structures: Data structures for Parallel Computing", To appear in proceedings of Workshop on Graph Reduction, Santa Fe, New Mexico (Sept 28 - Oct 1, 1986).

[3] J.Backus, "Can programming be liberated from the von Neumann style? A functional style and algebra of programs", Communications of the ACM, vol 21, No 8. (Aug 1978).

[4] W.P.Crowley, C.P.Hendrickson and T.E.Rudy, "The SIMPLE code", UCID 17715, Lawrence Livermore Laboratory (Feb 1978).

[5] R.S.Nikhil, "Id Nouveau: Reference Manual, Part I: Syntax", CSG Memo , MIT Laboratory for Computer Science,Cambridge, Mass. (April 1987).

[6] R.S.Nikhil, "Id World Reference Manual", CSG Memo , MIT Laboratory for Computer Science,Cambridge, Mass. (April 1987).

[7] D.A.Turner, "The Semantic elegance of applicative languages, Proc. of ACM conf. on Functional programming languages and Computer architecture, Portsmouth, New Hampshire (Oct 1981).