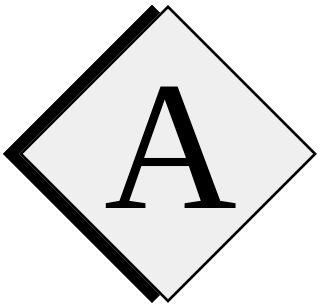

Appendix A. Use of Operators in ARPS

The methodology for solving the equations of hydrodynamics in either differential or integral form using grid-point techniques (finite difference, finite volume, finite element) can be described in four basic steps. In the first step, the appropriate continuous equation or set of equations is prescribed, as shown in Eq. (A.1) for a simple scalar conservation law

$$\frac{\partial T}{\partial t} = - \frac{\partial (uT)}{\partial x} - \frac{\partial (vT)}{\partial y} - \frac{\partial (wT)}{\partial z} . \tag{A.1}$$

Here, $T = T(x,y,z,t)$ is a scalar, t is time, and u, v, and w are the velocity components in the x,y, and z directions, respectively. This equation possesses considerable parallel structure in that all terms on the right hand side (RHS) are similar in form but differ in direction. This is more evident if the RHS is written using tensor notation

$$\frac{\partial T}{\partial t} = - \frac{\partial (u_i T)}{\partial x_i} , \tag{A.2}$$

where a summation $i = 1,2,3$ is implied such that $x_1 = x$, $x_2 = y$, $x_3 = z$ and $u_1 = u$, $u_2 = v$, $u_3 = w$. In an explicit numerical scheme, and in some implicit schemes, the three (elementwise matrix) multiplications of u_i and T, as well as the subsequent differentiations, can be performed independently.

In the second step of the solution process, the appropriate continuous equations are recast into discrete form, often through the use of differencing and averaging operators as a notational convenience, in a manner appropriate for a given computational mesh and solution methodology. Consider, for example, the centered operator notation used by Lilly (1965)

$$\overline{A(x)}^{nx} = \frac{A\left(x + n\Delta x / 2\right) + A\left(x - n\Delta x / 2\right)}{2} \tag{A.3}$$

$$\delta_{nx} A(x) = \frac{A\left(x + n\Delta x/2\right) - A\left(x - n\Delta x/2\right)}{n\Delta x} \tag{A.4}$$

where A is the dependent variable at an arbitrary location x, which is the independent variable, n is a positive integer, and Δx is the grid spacing. Eq. (A.4) is the discrete representation of a continuous first-order derivative $\partial A/\partial x$, while (A.3) represents an averaging operator that has no continuous counterpart. Other forms of these operators with varying definitions are widely used (*e.g.*, van Leer, 1977). As described in the next section, Eqs. (A.3) and (A.4), along with other fundamental operators (*e.g.*, one-sided spatial difference), may be used in combination to create a variety of higher-order expressions provided that certain commutative and associative properties, similar but not identical to those in the calculus, are obeyed.

Applying (A.3) and (A.4) to (A.1) using second-order quadratically conservative spatial differences on the Arakawa C-grid (*e.g.*, Arakawa and Lamb, 1977), along with a second-order leapfrog time discretization, yields

$$\delta_{2t} T^n_{i,j,k} = -\delta_x\left(u\,\bar{T}^x\right)^n_{i,j,k} - \delta_y\left(v\,\bar{T}^y\right)^n_{i,j,k} - \delta_z\left(w\,\bar{T}^z\right)^n_{i,j,k} \tag{A.5}$$

where the subscripts i, j, and k correspond to the x, y, and z directions, respectively (*e.g.*, $x_i = i\Delta x$), the superscript n indicates the time level, and $t = n\Delta t$ where Δt is the time step. At this point, the full structure of the governing equation (A.1) remains intact.

In the third step, Eq. (A.5) is expanded using the rules given in (A.3) and (A.4) to yield a set of linear algebraic equations:

$$\begin{aligned}\frac{T^{n+1}_{i,j,k} - T^{n-1}_{i,j,k}}{2\Delta t} = &-\frac{1}{\Delta x}\left[u^n_{i,j,k}\frac{(T^n_{i+1,j,k} + T^n_{i,j,k})}{2} - u^n_{i-1,j,k}\frac{(T^n_{i,j,k} + T^n_{i-1,j,k})}{2}\right] \\ &-\frac{1}{\Delta y}\left[v^n_{i,j,k}\frac{(T^n_{i,j+1,k} + T^n_{i,j,k})}{2} - v^n_{i,j-1,k}\frac{(T^n_{i,j,k} + T^n_{i,j-1,k})}{2}\right] \\ &-\frac{1}{\Delta z}\left[w^n_{i,j,k}\frac{(T^n_{i,j,k+1} + T^n_{i,j,k})}{2} - w^n_{i,j,k-1}\frac{(T^n_{i,j,k} + T^n_{i,j,k-1})}{2}\right].\end{aligned} \tag{A.6}$$

Finally, these algebraic equations are cast in an appropriate computer language using the subscripted arrays. For the example shown here, the code might be written in Fortran as

```
dimension u(nx,ny,nz),v(nx,ny,nz),w(nx,ny,nz),T(nx,ny,nz)

do k = 1,nz
do j = 1,ny
do i = 1,nx
  T(i,j,k,future) = T(i,j,k,past)
:    -0.5*rdx*dt*(u(i,j,k,now)*(T(i+1,j,k,now)+T(i  ,j,k,now))-
:                 u(i-1,j,k,now)*(T(i,j,k,now)+T(i-1,j,k,now)))
:    -0.5*rdy*dt*(v(i,j,k,now)*(T(i,j+1,k,now)+T(i,j  ,k,now))-
:                 v(i,j-1,k,now)*(T(i,j,k,now)+T(i,j-1,k,now)))
:    -0.5*rdz*dt*(w(i,j,k,now)*(T(i,j,k+1,now)+T(i,j,k  ,now))-
:                 w(i,j,k-1,now)*(T(i,j,k,now)+T(i,j,k-1,now)))
end do
end do
end do                                                          (A.7)
```

where future, now, and past are the time indices for levels n+1, n, and n-1; $rdx = 1/\Delta x$, $rdy = 1/\Delta y$, $rdz = 1/\Delta z$; $dt = \Delta t$; and nx, ny, and nz are the number of grid-points in each coordinate direction. In moving from Eq. (A.5) to the FORTRAN code, much of the functional (*i.e.*, mathematical operation) parallel structure has been lost. For example, it is no longer possible to perform independently or to separate the three products of the scalar T with the velocities u, v, and w. The same is true for the three differentiations. Furthermore, resemblance to the governing equation (A.1) has been greatly diminished, and the code is inherently prone to error due to the number of array index manipulations involved. Extension of this code to different or higher order schemes, though perhaps straightforward in theory, is cumbersome and prone to error, presenting formidable challenges to users and greatly reducing the utility of the code.

The obvious solution to the problems illustrated above is to *eliminate the operator expansion step (step 3) and make use of the operator constructs in the computer code itself.* Consider, therefore, a set of subroutines, the details of which are described in the next section, that perform the discrete operations shown in Eqs. (A.3) and (A.4). Each routine receives on input a dependent variable and perhaps information concerning its dimensionality and location within the grid, and returns on output a transformed variable according to the operation performed.

For example, let AVGX(input_var, output_var) and DIFFX(input_var, output_var) be the Fortran counterparts of Eqs. (A.3) and (A.4), respectively, for the x direction (similar routines exist for the y and z directions), and let routine AAMULT(input1, input2, output) return as output the element-wise

product of two input matrices input1 and input2. Assuming that the appropriate nested DO-loops are contained within each operator routine, and for clarity neglecting other arguments passed to this routine, the operator-based code for solving Eq. (A.1) can be written as

```
dimension u(nx,ny,nz), v(nx,ny,nz), w(nx,ny,nz), T(nx,ny,nz)
dimension temp1(nx,ny,nz), temp2(nx,ny,nz),temp3(nx,ny,nz)

call avgx(T, temp1)            ! temp1 contains  T̄^x
call avgy(T, temp2)            ! temp2 contains  T̄^y
call avgz(T, temp3)            ! temp3 contains  T̄^z

call aamult(u, temp1, temp1)   ! second temp1 contains U T̄^x
call aamult(v, temp2, temp2)   ! second temp2 contains V T̄^y
call aamult(w, temp3, temp3)   ! second temp3 contains W T̄^z

call diffx(temp1, temp1)    ! second temp1 contains δ_x (U T̄^x)
call diffy(temp2, temp2)    ! second temp2 contains δ_y (V T̄^y)
call diffz(temp3, temp3)    ! second temp3 contains δ_z (W T̄^z)

do k = 1,nz
do j = 1,ny
do i = 1,nx
 T(i,j,k,future) = T(i,j,k,past)
:   -2.*dt*(temp1(i,j,k,now)+temp2(i,j,k,now)+temp3(i,j,k,now))
end do
end do
end do                                                          (A.8)
```

A number of important points are worth noting about this code relative to that found in (A.7). First, the fundamental mathematical structure of the three types of operations (averaging, matrix multiplying, and differencing) is clearly evident. Second, it is clear that, although the *types* of operations must be performed sequentially for each pairing of variables (*e.g.*, the matrix multiply of *u* and the average of *T* cannot occur until the average of *T* is available), computations *within* each type (*e.g.*, all averaging operations) can be performed independently and simultaneously. In distributing this parallelism or granularity, one might choose a *functional decomposition* for a shared memory computer or workstation cluster in which *avgx* is executed on one processor, *avgy* on another and *avgz* on yet another. No communication would ever be required among the processors since each has a copy of the en-

tire array *T*. In a *data decomposition* mode for a distributed memory parallel computer, one could load the *T* array across all processing nodes and then simply perform each averaging operation independently, first in *x*, then in *y*, then in *z*. Again, no communication is required after the computations begin.

Third, the code in Eq. (A.8) bears more of a resemblance to the governing equation (A.1) or its discrete operator representation (A.3) than does (A.7) because (A.8) is written in a manner analogous to the continuous problem using functionally similar constructs. As a result, once the operators are known to be correct and properly applied, the code can be verified largely by inspection, thereby facilitating its debugging, maintenance, and *correct* usage by those unfamiliar with it. Implementing higher order or more complex numerical schemes, virtually all of which can be expressed using some forms of operator notation, simply involves using other operators or different combinations thereof.

Fourth, the complexity of the array index manipulations, all of which are extremely basic and occur in the operator routines (all of which contain less than 10 lines of executable code), is hidden from the user. By redefining a particular operator, one can change literally hundreds of lines of code in a matter of minutes. Finally, the storage costs associated with the operator methodology range from one to two times that of a more standard code, depending upon the use of temporary arrays and overlaid storage. It is our experience that the benefits associated with the operators and some use of temporary storage far outweigh the associated overhead, although this issue is somewhat problematic.

The obvious simplicity and apparent universality of the discrete operator methodology suggests its application to broad classes of problems involving differential or integral equations. In much the same way that graphics primitives are used to build images of complex objects, the discrete operators can, we believe, serve as the building blocks of complex numerical models.