An Introduction to the Piecewise Parabolic Method (PPM) Library I. Interpolation and Hydrodynamics Routines: Part 3a-c

Paul Ralph Woodward,[†] *B Edgar,*
and S. E. Anderson,[†]

Laboratory for Computational Science and Engineering
Department of Astronomy, School of Physics and Astronomy
116 Church St. SE.
University of Minnesota
Minneapolis, Minnesota 55455

3. Hydrodynamics

a Introduction

The approach to hydrodynamics embodied in PPMLIB is based upon the ideas of Godunov (1959). In the simplest of Godunov type schemes, the data are represented by zone-averages (piecewise-constant). The zones are allowed to interact only at zone interfaces, where discontinuities can exist between adjacent zones. These discontinuities are resolved by finding solutions to Riemann Shock Tube Problems at each zone interface. These solutions, which can be solved exactly, incorporate the non-linear interactions between two constant hydrodynamic states, resulting in a set of time-averaged fluxes at zone interfaces by which the zone averaged quantities may be advanced one timestep. Such a scheme, based upon piecewise constant values, is first order accurate.

For the hydrodynamics routines available in PPMLIB the ideas of van Leer (1979) are used to extend the above method to a higher order of accuracy. The piecewise constant representations are replaced by piecewise parabolic interpolations. These interpolations are then integrated over the appropriate domains of dependence (those portions of each zone which may physically interact at the zone interface during a given timestep). In this way, the initial states for the Riemann problems more properly reflect the information producing the time-averaged fluxes at each zone interface.

The multi-dimensional Eulerian flow equations may be approximated by applying a one-dimensional Eulerian scheme in alternation according to the directional-splitting algorithm of Strang (1968). For example, in two dimensional Cartesian geometry (x,y) this would be performed as a sequence of one-dimensional passes, $x - y - y - x$. This symmetrized order provides second-order accuracy. In this way, multidimensional hydrodynamics reduces to constructing approximate solutions to Euler's equations in one spatial variable, say r.

Two classes of hydrodynamical routines exist in PPMLIB. The first consists of basic routines which generally perform only one function or closely related group of functions. The user will usually have little need or desire to examine or manipulate the source to these routines, all of which champion the prefix "**PPM98_**" (or "**D_PPM98_**" for double precision versions). The interpolation routines of section 2 are examples of this class of routines.

The other class of routine is introduced in this section - a "template" routine. It would not be feasible (or desirable) to attempt a definitive set of hydrodynamical routines. There would always be that one (vocal) user whose pet problem was overlooked. Instead, it is hoped that the ability to construct a wide variety of hydrodynamical applications from the individual PPMLIB modules can be demonstrated by a series of graded example templates, the complete source for which are provided. Each template is part of a series ordered roughly from simple to complex. Later templates build upon the foundation set by the earlier templates. In this way, the user is shown how the pieces fit together and how they can be rearranged for a variety of problems. This should empower the user to create his own customized routines, either from scratch or by modifying one of the templates. The template routines brandish the prefix "**Do_PPM**", or "**D_Do_PPM**" for double precision. Their source is available at the PPMLIB web site www.lcse.umn.edu//PPMlib.

Routines exist in PPMLIB for two approaches to hydrodynamics. One is a two step process. The hydrodynamics is computed in Lagrangian coordinates, with the results mapped from the Lagrangian grid back to the Eulerian grid. The other approach is a direct Eulerian calculation. The separation of the hydrodynamical interactions from the advective computation in the Lagrangian + remap approach permits relatively simple hydrodynamic routines to be constructed, especially when the hydrodynamics is only part of the overall problem. The cost however is greater computation. The state variables must be interpolated twice for each pass. This also doubles the boundary zone requirement. A full Lagrangian plus remap step requires 9 fake zones to be appended to both ends of the arrays. The direct Eulerian requires only 5. This can become a consideration in multi-dimensions where the "surface to volume" ratio of fake zones to real zones can become expensive, both in CPU time and memory. The choice, of course, is ultimately up to the programmer. Unfortunately, the PPMLIB hydrodynamical routines do not lend themselves to isolated study as easily as did the interpolation routines of the previous section. Therefore these routines will be discussed only in the context of the templates.

Because the Direct Eulerian formulation requires fewer operations, it will be considered first. Following the brief description of Eulerian hydrodynamics, is a tutorial consisting of several specific examples of computing hydrodynamics with the PPMLIB templates. In the tutorial, an attempt has been made to arrange the examples from simplest to most difficult, a subjective ranking at best. In any case, each subsequent example assumes that the reader has at least some familiarity with the preceding examples, better yet, has compiled and run them. It will also be assumed that the reader has a working knowledge of the terms and examples in the preceding sections of this guide. A discussion of Lagrangian hydrodynamics follows the Eulerian tutorial. Following that is a Lagrangian tutorial, which, like the Direct Eulerian tutorial, consists of a series of graduated examples evolving from simple to complex, again using PPMLIB templates.

b Eulerian Hydrodynamics

When written in Eulerian coordinates in one dimension, Euler's equations in conservation form, are

$$\begin{aligned} \frac{\partial}{\partial t}\rho + \frac{\partial}{\partial V}(r^{\alpha}\rho u) &= 0 \\ \frac{\partial}{\partial t}\rho u + \frac{\partial}{\partial V}(r^{\alpha}\rho u^2) + \frac{\partial}{\partial r}p &= \rho f \\ \frac{\partial}{\partial t}E + \frac{\partial}{\partial V}[r^{\alpha}(\rho u E + up)] &= \rho u f \end{aligned} \tag{3.1}$$

where p is the pressure, u is the velocity, E is the total energy per unit volume, f is a body force, and V is a volume coordinate

$$V(r) = \frac{r^{\alpha+1}}{(\alpha+1)} \tag{3.2}$$

with $\alpha = 0, 1, 2$ for planar, cylindrical, or spherical symmetry, respectively.

These equations must be augmented with an *equation of state* (EOS) which specifies the relationship between the state variables. Two equations of state are supported. One is a gamma law EOS

$$p = (\gamma - 1)\rho\varepsilon \tag{3.3}$$

where γ is the ratio of specific heats of the gas and ε is the specific internal energy. The later is related to the total energy by the relation

$$\varepsilon = E - \frac{u^2}{2}\,. \tag{3.4}$$

The other EOS is a generalized or metal equation of state

$$p = \Phi + g\,\rho\,\varepsilon \tag{3.5}$$

where Φ and g are two constants. Equation 3.5 implies that the square of the Eulerian sound speed is

$$c^2 = g(\varepsilon + \frac{p}{\rho}), \tag{3.6}$$

hence

$$c^2 = \frac{[(g+1)p] - \Phi}{\rho}\,. \tag{3.7}$$

The constants in the equation of state must be consistent with the relation

$$c^2 = \left.\frac{\partial p}{\partial \rho}\right]_{\text{constant }\varepsilon} + \left.\frac{\partial p}{\partial \varepsilon}\right]_{\text{constant }\rho} \tag{3.8}$$

where

$$\left.\frac{\partial p}{\partial \rho}\right]_{\text{constant }\varepsilon}$$

is positive definite and

$$= \left.\frac{\partial p}{\partial \varepsilon}\right]_{\text{constant }\rho}$$

is nonnegative. Negative pressures, however, are allowed. The reader should note that when $\Phi = 0$ and $g = (\gamma - 1)$, the gamma law EOS (eqn. 3.3) is obtained.

Beginning with the zone averages of the conserved quantities in each zone at time t^n,

$$U_j^n = \frac{1}{\Delta V_i}\int_{r_i-1/2}^{r_i+1/2} U(r, t^n)\,dV,$$

$$U = \begin{pmatrix} \rho \\ u \\ E \end{pmatrix}, \tag{3.9}$$

A finite difference approximation to eqns (3.1) may be found by integrating them over a space-time volume ΔV by Δt. This results in the preliminary updated zone averages

$$\rho_j^{n+1} = \rho_j^n + \Delta t\left(\frac{A_{j-1/2}\,(\overline{\rho u})_{j-1/2} - A_{j+1/2}\,(\overline{\rho u})_{j+1/2}}{\Delta V_j}\right)$$

$$(\rho u)_j^{n+1} = (\rho u)_j^n + \Delta t\left(\frac{A_{j-1/2}\,(\overline{\rho u^2})_{j-1/2} - A_{j+1/2}\,(\overline{\rho u^2})_{j+1/2})}{\Delta V_j}) + \frac{(\bar{p}_{j-1/2} - \bar{p}_{j+1/2})}{\Delta r_j} + 1/2\left((\rho f)_j^n + (\rho f)_j^{n+1}\right)\right. \tag{3.10}$$

$$(\rho E)_j^{n+1} = (\rho E)_j^n + \Delta t \left(\frac{A_{j-1/2}(\overline{\rho u E} + \overline{up})_{j-1/2} - A_{j+1/2}(\overline{\rho u E} + \overline{up})_{j+1/2}}{\Delta V_j} \right) + 1/2 \left((\rho u f)_j^n + (\rho u f)_j^{n+1} \right).$$

Here the bars indicate time averaged quantities.

For a complete hydrodynamical computation, a diffusion step is also necessary. First, zone centered Eulerian velocities are calculated, v_i, which are large in shocks but zero elsewhere. From these a zone-interface diffusion velocity is found

$$\hat{v}_{j-1/2} = \max(v_j, v_{j-1}). \tag{3.11}$$

This allows the determination of the diffusive zone interface fluxes of mass, momentum, and total energy:

$$f^{\rho}_{j-1/2} = \hat{v}_{j-1/2}(\rho_i^n - \rho_{i-1}^n)\Delta t$$

$$f^{u}_{j-1/2} = \hat{v}_{j-1/2}(\rho u)_i^n - (\rho u)_{i-1}^n)\Delta t \tag{3.12a}$$

$$f^{E}_{j-1/2} = \hat{v}_{j-1/2}(\rho E)_i^n - (\rho E)_{i-1}^n)\Delta t$$

It will prove useful later to rewrite these equations as

$$f^{\rho}_{j-1/2} = \upsilon_{j-1/2}(\rho_i^n - \rho_{i-1}^n)$$

$$f^{u}_{j-1/2} = \mu^L_{j-1/2} u_i^n - \mu^R_{j-1/2} u_{i-1}^n) \tag{3.12b}$$

$$f^{E}_{j-1/2} = \mu^L_{j-1/2} E_i^n - \mu^R_{j-1/2} E_{i-1}^n)$$

where $\upsilon_{j-1/2}$ is the diffusive volume flux at the lefthand zone interface

$$\upsilon_{j-1/2} = \hat{v}_{j-1/2}\Delta t, \tag{3.13a}$$

$\mu^L_{j-1/2}$ is the diffusive mass flux from zone $i-1$ to zone i,

$$\mu^L_{j-1/2} = \upsilon_{j-1/2}\rho_{i-1}^n, \tag{3.13b}$$

and $\mu^R_{j-1/2}$ is the diffusive mass flux from zone i to $i-1$

$$\mu^R_{j-1/2} = \upsilon_{j-1/2}\rho_i^n. \tag{3.13c}$$

Then the final updated zone averages are found by conservatively differencing these diffusive fluxes:

$$\rho'^{n+1}_{j-1/2} = \rho^{n+1}_{j-1/2} + \frac{\left(A_{j-1/2} f^{\rho}_{j-1/2} - A_{j+1/2} f^{\rho}_{j+1/2} \right)}{\Delta V_j}$$

$$(\rho u)'^{n+1}_{j-1/2} = (\rho u)^{n+1}_{j-1/2} + \frac{\left(A_{j-1/2} f^{u}_{j-1/2} - A_{j+1/2} f^{u}_{j+1/2} \right)}{\Delta V_j} \tag{3.14}$$

$$(\rho E)'^{n+1}_{j-1/2} = (\rho E)^{n+1}_{j-1/2} + \frac{\left(A_{j-1/2} f^{E}_{j-1/2} - A_{j+1/2} f^{E}_{j+1/2} \right)}{\Delta V_j}$$

In the following discussion, the above procedure to find the new zone averages of the conserved quantities at time $t^{n+1} = t^n + \Delta t$ is be divided into the following steps using routines from the PPMLIB library:

1) Compute piecewise parabolic distributions of the dependent variables in each zone using interpolation routines described in the previous chapter. This produces a global distribution of the dependent variables which is parabolic within each zone, but which may have discontinuities at the zone boundaries.

2) Determine effective left and right states for a Riemann problem at each zone interface by averaging the above interpolations of the of the various domains of dependence. This generally results in discontinuities at zone boundaries.

3) Calculate the time averaged zone interface values of density ($\bar{\rho}_{j-1/2}$), pressure ($\bar{p}_{j-1/2}$), velocity ($\bar{u}_{j-1/2}$), and total energy ($\bar{E}_{j-1/2}$), through the application of a *Riemann solver*. This calculates the solution to Riemann's shock tube problem, which is the resolution of an arbitrary hydrodynamic discontinuity.

4) Calculate the preliminary updated zone averages by by differencing these fluxes using the difference equations (3.10).

5) Determine zone centered Eulerian diffusion velocities in those zones which are likely to generate numerical noise, such as strong shocks or shocks moving slowly with respect to the numerical grid.

6) Determine zone interface diffusion fluxes of mass, momentum, and total energy from eqns. (3.11-12).

7) Calculate the final updated zone averages by conservatively differencing the diffusion velocities using eqns (3.14).

These steps will now be described in more detail with the simplifying assumptions of planar symmetry ($\alpha = 0$) and of no body forces.

Step 1. Interpolation of distributions

It is necessary to find interpolations of density, pressure, and velocity. In contrast to the classic PPM algorithm (Colella and Woodward 1984) which interpolated these quantities directly, in modern versions of PPM they are interpolated indirectly through *Riemann invariants* . In smooth flow, certain quantities (known as Riemann invariants) are conserved along characteristic paths. (In hydrodynamics, these characteristic paths are the paths of sound waves in space-time.) These invariants are, in general, only expressible as inexact differentials.

In characteristic form, eqns (3.1) are

$$du \pm \frac{dp}{C} = 0 \quad \text{along} \quad dx = (u \pm c)dt, \tag{3.15}$$

and the streamline characteristic is

$$d\rho \pm \frac{dp}{c_2} = 0 \quad \text{along} \quad dx = u\,dt, \tag{3.16}$$

where C is the Lagrangian sound speed

$$C = \sqrt{\gamma p \rho}, \tag{3.17}$$

and c the Eulerian sound speed,

$$c = \frac{C}{\rho} \tag{3.18}$$

The compatibility equations describe the quantities conserved along the characteristic paths,

$$dR^{\pm} = du \pm \frac{dp}{C}\,, \tag{3.19a}$$

$$dR^{0} = d\rho - \frac{dp}{c^2}\,, \tag{3.19b}$$

where R^{+}, R^{-}, and R_0 are the three Riemann invariants. The derivation of the equations is straightforward (cf. Courant and Friedricks 1948).

The interpolation procedure for u and p begins by finding left differences of zone averages of the Riemann invariants, $R^{\pm}$:

$$dR^{+}_{i-1/2} = (u_i - u_{i-1}) + \frac{(p_i - p_{i-1})}{1/2\,(C_i + C_{i-1})} \tag{3.20a}$$

and

$$dR^{-}_{i-1/2} = (u_i - u_{i-1}) - \frac{(p_i - p_{i-1})}{1/2\,(C_i + C_{i-1})} \tag{3.20b}$$

These differences may then be interpolated, resulting in a lefthand and a righthand zone interface value in each zone (relative to a vanishing zone average) for the Riemann invariants: R_i^{+L} , R_i^{+R} , R_i^{-L} , and R_i^{-R} . These may then be combined to reveal the implied left and right hand zone interface values for pressure and velocity:

$$\begin{aligned} u_i^L &= u_i + 1/2\,(R_i^{+L} + R_i^{-L}) \\ p_i^L &= p_i + \frac{C_i}{2}\,(R_i^{+L} - R_i^{-L}) \\ u_i^R &= u_i + 1/2\,(R_i^{+R} + R_i^{-R}) \\ p_i^R &= p_i + \frac{C_i}{2}\,(R_i^{+R} - R_i^{-R}) \end{aligned} \tag{3.21}$$

Finally, monotonicity constraints must be applied to these velocity and pressure interpolations.

This interpolation of the pressure and velocity using Riemann invariants is performed by **PPM98_INTERP0_UP** for uniform grids and **PPM98_INTERP_UP** for nonuniform grids.

The density interpolation procedure begins with finding left differences of zone averages of the adiabatic Riemann invariant, R_0:

$$dR^{0}_{i-1/2} = \rho_i - \rho_{i-1} + \frac{(p_i - p_{i-1})}{1/2\,(c_i^2 + c_{i-1}^2)} \tag{3.22}$$

These are then interpolated, producing a lefthand and a righthand zone interface value in each zone (again, relative to a vanishing zone average) for the Riemann invariant: R_i^{0L} and R_i^{0R}. The underlying density interpolation can then be determined:

$$\begin{aligned} \rho_i^L &= \rho_i + R_i^{0L} + \frac{(p_i^L - p_i)}{1/2\,(c_i^2 + c_{i-1}^2)}\,, \\ \rho_i^R &= \rho_i + R_i^{0R} + \frac{(p_i^R - p_i)}{1/2\,(c_i^2 + c_{i-1}^2)}\,, \end{aligned} \tag{3.23}$$

where p_i^L and p_i^R are the left and right interpolated zone interface pressure values found from eqn (3.21). Note that this requires prior completion of the the pressure interpolation. Finally monotonicity constraints are applied to the density interpolations. The density interpolation using the streamline Riemann invariant is performed by **PPM98_INTERP0_RHO** for uniform grids and **PPM98_INTERP_RHO** for nonuniform grids.

The rather baroque procedure described above to find the velocity, pressure, and density interpolations permits monotonicity constraints to be applied directly to the Riemann invariants. Experience has shown that this improves results, particularly in pathological cases such as weak shock waves moving slowly with respect to the computational grid.

Step 2. Calculating the Effective States

The effective states for the Riemann problems are found by finding the spatial averages over the *domains of dependence* for each zone interface. The plus domain of dependence may be approximated by tracing back the paths of all sound waves travelling in the positive direction with respect to the fluid (that is, those waves travelling with a velocity of u+c in the laboratory frame of reference) which may impinge upon the zone interface from the zone to the left of the interface during the timestep. Likewise, the minus domain of dependence may be approximated by tracing back the paths of all sound waves travelling in the negative direction with respect to the fluid (that is, those waves travelling with a velocity of u-c in the laboratory frame of reference) which may impinge upon the zone interface from the zone to the right of the interface during the timestep. Finally, the streamline characteristic is traced backwards. It should be noted however, that this requires knowledge of the time averaged velocity at the zone interface, complicating the process somewhat. The maximum extents of the domains of dependence may be approximated as follows: for the plus domain of dependence

$$\max[0, (u_{i-1} + c_{i-1})\Delta t]$$

to the left of the zone interface. The maximum extent of the minus domain of dependence is

$$-\min[0, (u - c_i)\Delta t]$$

to the right of the zone interface. The maximum extent to the streamline domain of dependence is

$$\bar{u}_{j-1/2}\,\Delta t.$$

This is illustrated in figure 13. The required states consist of averages of velocity and pressure over the plus and minus domains of dependence and averages of density and pressure over the streamline domain of dependence unsing eqns. (2.27-28). These averages of pressure and velocity over the plus and minus Eulerian domains of dependence are found by **PPM98_EULER_DOMAIN_AVG** and **PPM98_EULER0_DOMAIN_AVG**, both of which incorporate the interpolations of Step 1.

Step 3. Solving the Riemann Problems

The determination of effective states for a zone interface usually results in a discontinuity at that interface, which must now be resolved. The break up of an arbitrary fluid discontinuity is described by a solution to Riemann's shock tube problem. In hydrodynamics, this solution usually consists of three states, the effective left and right states, U_L and U_R, and a middle state U_*, separated by waves which may be shocks or rarefactions (Fig. 14). The solution of Riemann's shock tube problem is discussed in Courant and Friedrichs (1948), van Leer (1979), and Colella and Glaz (1985).

Positive definite nonlinear wave speeds $W^{\pm}$ for discontinuous waves traveling to the right (+) or to the left (-) may be determined from the shock jump equations and the integral form of the conservation laws. The desired time averaged values for pressure and velocity then satisfy the following equations:

$$\begin{aligned}
&\frac{\bar{p} - p_L}{W^+} + (\bar{u} - u_L) = 0 \\
&W^+ = C_L \sqrt{1 + \frac{\gamma + 1}{2\gamma}\left(\frac{\bar{p} - p_L}{p_L}\right)} \\
&\frac{\bar{p} - p_R}{W^-} - (\bar{u} - u_R) = 0 \\
&W^- = C_R \sqrt{1 + \frac{\gamma + 1}{2\gamma}\left(\frac{\bar{p} - p_R}{p_R}\right)}.
\end{aligned} \qquad (3.24)$$

In the above equations, it is assumed that all quantities are referenced to the lefthand zone interface. Therefore the subscript "$i - 1/2$" has been suppressed in the interest of clarity. These nonlinear equations may be solved by the iterative procedure given in van Leer (1979). These solutions will hold in rarefactions if rarefaction waves are approximated by discontinuous jumps.

Once the time averaged velocity at the interface has been determined, it is possible to determine the averages of density and pressure over the streamline domain of dependence. The time averaged density may then be found from eqn (3.18b). The time averages of density, pressure, and velocity at the zone averages are computed by **PPM98_E_RIEMANN_GAMMA** for calculations assuming a gamma law EOS and **PPM98_E_RIEMANN** for general EOS calculations. With this calculation of the Eulerian fluxes, it is now possible to begin the update of the zone averages.

Step 4. Preliminary Zone Update

With the assumptions of planar symmetry and no body forces, eqns (3.10) become

$$\rho_j^{n+1} = \rho_j^n + \Delta t \left(\frac{(\overline{\rho u})_{j-1/2} - (\overline{\rho u})_{j+1/2})}{\Delta r_j} \right)$$

$$(\rho u)_j^{n+1} = (\rho u)_j^n + \Delta t \left(\frac{(\overline{\rho u^2})_{j-1/2} - (\overline{\rho u^2})_{j+1/2})}{\Delta r_j}) + \frac{(\bar{p}_{j-1/2} - \bar{p}_{j+1/2})}{\Delta r_j} \right) \tag{3.25}$$

$$(\rho E)_j^{n+1} = (\rho E)_j^n + \Delta t \left(\frac{(\overline{\rho u E} + \overline{up})_{j-1/2} - (\overline{\rho u E} + \overline{up})_{j+1/2}}{\Delta r_j} \right).$$

These equations, combined with the results from the Riemann solver determine the preliminary updated zone averages of the conserved quantities. These preliminary averages can be found using **PPM98_DEUL_UPDATE_CART**.

Step 5. Calculating Diffusion Velocities

In some pathological cases, such as thin strong shocks or shocks moving slowly with respect to the numerical grid, it becomes necessary to add additional numerical dissipation. Care is taken however, to ensure that it is added only to those zones which can benefit from it. First, shock detection is performed. By looking at the zone averages surrounding a given zone, the jumps in pressure, velocity, and density across the zone are found. If these jumps are consistent with the internal structure of shocks, a pressure jump ratio is then calculated

$$\delta p_i = \frac{p_- - p_+}{p_+} \tag{3.26}$$

where p_+ and p_- are the pre and post shock pressure values, resp.

To prevent numerical noise from registering as shocks, a zone is considered to be in a shock only if the pressure jump is greater than some threshold J_0. An indicator, S_i, signalling whether or not a given zone is in a shock is then constructed:

$$S_i = \begin{cases} 1 & \text{if} \quad \delta p_i > J_0 \\ 0 & \text{otherwise.} \end{cases} \tag{3.27}$$

The default value for J_0 is 0.25.

For zones thus determined to be in shocks, a preliminary diffusion velocity is determined from the difference in nearby velocity zone averages,

$$v'_i = V_0(u_{i+2} - u_{i-2}), \tag{3.28}$$

with the default value for V_0 being 0.5.

However, not all shocks require the same amount of additional diffusion. Short wavelength noise is quickly damped by the hydrodynamic routines of PPMLIB, therefore only shocks likely to emit long wavelength noise are contentious. Therefore, the fundamental wavelength of the noise emitted by the shock is determined. The diffusion velocity is smoothly reduced as the fundamental wavelength of the noise decreases by constructing a coefficient multiplying the diffusion velocity in equation 3.28. this coefficient is

$$\xi_i = 0.1 + 0.9\zeta_i \tag{3.29}$$

where

$$\zeta_i = \frac{\psi_i}{\psi_i + W_0},$$

$$\psi_i = \begin{cases} (\lambda_{noise} - 2)^3 & \text{if} \quad \lambda_{noise} > 2 \\ 0 & \text{otherwise.} \end{cases}$$

In the above equation, λ_{noise} is the fundamental wavelength of the noise emitted by the shock, and W_0 is a parameter regulating the rapidity of the change in ξ_i. The default value of W_0 is 1. With this default, the diffusion velocity reaches half its asymptotic value when λ_{noise} is 3, and it reaches 82% of its asymptotic value when λ_{noise} is 4. Thus the final diffusion velocity is

$$v_i = \xi_i v_i' \tag{3.30}$$

Step 6. Calculating Diffusion Fluxes Now that zone centered diffusion velocities have been found, they may be combined with eqns (3.12) to produce the diffusion interface fluxes:

$$\begin{aligned} f^{\rho}_{j-1/2} &= \upsilon_{j-1/2}(\rho_i^{n+1} - \rho_{i-1}^{n+1}) \\ f^{u}_{j-1/2} &= \mu^L_{j-1/2} u_i^{n+1} - \mu^R_{j-1/2} u_{i-1}^{n+1}) \\ f^{E}_{j-1/2} &= \mu^L_{j-1/2} E_i^{n+1} - \mu^R_{j-1/2} E_{i-1}^{n+1}) \end{aligned} \tag{3.12b}$$

These diffusion fluxes are found by **PPM98_DIFUSEFLUX_HYDRO_CART**, which also returns the diffusive volume and mass fluxes of eqns (3.13a-c). These latter fluxes will be useful when there are other quantities in the calculations, such as the passively advected quantities described later.

Step 7. Updating Zone Averages

Finally, the diffusive fluxes are combined with eqn (3.14) to produce the final updated zone averages:

$$\begin{aligned} \rho'^{n+1}_{j-1/2} &= \rho^{n+1}_{j-1/2} + \frac{\left(f^{\rho}_{j-1/2} - f^{\rho}_{j+1/2}\right)}{\Delta r_j} \\ (\rho u)'^{n+1}_{j-1/2} &= (\rho u)^{n+1}_{j-1/2} + \frac{\left(f^{u}_{j-1/2} - f^{u}_{j+1/2}\right)}{\Delta r_j} \\ (\rho E)'^{n+1}_{j-1/2} &= (\rho E)^{n+1}_{j-1/2} + \frac{\left(f^{E}_{j-1/2} - f^{E}_{j+1/2}\right)}{\Delta r_j} \end{aligned} \tag{3.31}$$

where in the above equations the assumption of planar symmetry has been assumed.

c Direct Eulerian Tutorial

In this tutorial, the application of the hydrodynamical routines of PPMLIB will be demonstrated by examining their use in the PPMLIB templates. It is not possible to anticipate the exact requirements for every problem that any given user may wish to solve. Therefore, this set of templates has been developed to illustrate how the modules fit together to form complete hydrodynamic routines. They may serve as useful hydrodynamics routines in and of themselves, but each is intended to serve as template or framework from which the user starts, customizing as necessary to suit the requirements of the user's problem. The current Direct Eulerian templates are:

1) Do_PPM_DE0_1DC_GAMMA: This is the simplest template of all. It computes one timestep of a one-dimensional problem assuming a uniform grid and planar symmetry. A gamma law equation of state is also assumed.

2) Do_PPM_DE_1DC_GAMMA: This is the same as 1, except the assumption of a uniform grid is relaxed.

3) Do_PPM_DE0_1DC: This is is the same as 1, but a gamma law equation of state is no longer assumed.

4) Do_PPM_DE0_1DC_PAQ_GAMMA: Again, this begins with the basic program of 1, but now adds the concept of a passively advected quantity (PAQ). A passively advected quantity is something which is merely transported with the flow. It does not actively influence the hydrodynamics, or if it does, its effects can be isolated and treated in another step. For example, multi-dimensions are treated using directional splitting, in which changes in the flow are computed in only one direction at a time. The transverse momenta are merely passively advected during each individual directional pass.

5) Do_PPM_DE0_2DC_GAMMA: This uses the concept of a passively advected quantity introduced in the last template to compute two dimensional flows.

6) Do_PPM_DE0_3DC_GAMMA: This extends the last program to three dimensions.

These will now be discussed in greater detail. It is not possible to reproduce the entire source code to each of these templates here, only the highlights will be stressed. The user is encouraged to study the source to the templates which can be found at the PPMLIB web site. A simple driver is provided for each of these templates. a *driver* is the program which performs the basic housekeeping chores which, at the very least, must include

i) Initializing the problem by setting the initial values of all zone averages and constants upon which the program depends.

ii) Impose boundary conditions by setting the fake zones to values appropriate to the problem boundary conditions.

iii) Execute the hydrodynamic calculation.

iv) Produce output as desired.

v) Terminate execution at end of the problem.

The 1-dimension drivers provided for these templates are similar, so only the driver "Driverdirecte1dg0c.EXA" will be discussed in detail. This initializes and runs a one-dimensional shock tube problem on a uniform Cartesian grid, assuming a gamma law ga. The complete source for all the drivers can be found at http://www.lcse.umn.edu/PPMlib/Examples/Hydrodynamics. This source code is intended to be preprocessed by the "m4" macro processor to create either single precision or double precision examples. This discussion will be further restricted to single precision. The changes for double precision are left as an exercise for the reader.

```
c
      program Driverdirecte1dg0c
c
c     some ability to change the program parameters is provided via:
c
      parameter (NZONES = 128)
      parameter (MAXSTEPS = 100)
      parameter (EOSGAM = 1.4)
      parameter (DT = 0.0025)
      parameter (XMAX = 1.0)
c     Note that because of the fixed timestep, N, XMAX, and DT are not
c               independent.
c
c      NZONES = number of real computational zones in the calculation
c      MAXSTEPS = maximum number of timpsteps to perform.
c      EOSGAM = the equation of state gamma
c      DT = the fixed timestep
c      XMAX = the righthand edge value of the grid.  The grid runs from (0,XMAX)c
c
c     Do not change this one:
      parameter (NBDY = 5)
c
      dimension    xl(1-NBDY:NZONES+NBDY+1)
      dimension   rho(1-NBDY:NZONES+NBDY),   p(1-NBDY:NZONES+NBDY)
      dimension     u(1-NBDY:NZONES+NBDY)
      dimension rhonu(1-NBDY:NZONES+NBDY), unu(1-NBDY:NZONES+NBDY)
      dimension   pnu(1-NBDY:NZONES+NBDY)

cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c     SET constant parameters                                          c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
```

```
c   For round off ...
      smlrho  = 1.e-06
      smalle = 1.e-06
      smallp = 1.e-06
      smallu = 1.e-06

      gamma = EOSGAM
      xlimit = XMAX

      dtime = DT
      tmax = float( MAXSTEPS ) * dtime
      n = nzones

cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c                                                                      c
c     SETUP initial state:                                             c
c                                                                      c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
      k = 1-nbdy
      call setup( xl(k), rho(k), p(k), u(k), xlimit, gamma, n, nbdy )
c
c   Initialize counters  for total time and number of timestep:
      time = 0.0
      nsteps = 0
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c                                                                      c
c     The 1010 loop is the hydro loop.  One iteration is one timestep  c
c        Each timestep is divided into three substeps:                 c
c          1:  Impose boundary conditions                              c
c          2:  Call a PPM hydro package to produce new zone averages   c
c          3:  Replace old zone averages with new in preparation       c
c               for the next timestep.                                 c
c          4:  Produce output if desired                               c
c                                                                      c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c

1010   continue

c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c                                                                      c
c     Step 1: Boundary Condition Step                                  c
c       Boundary conditions are set at the beginning of each timestep  c
c       by placing appropriate values of XL, RHO, P, and U in the fake c
c       zones on the left and right ends of the arrays.                c
c                                                                      c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c

c      Impose reflecting boundaries:
      k=1-nbdy
      call boundary_refl(xl(k), rho(k), u(k), p(k), n, nbdy)

c
c     clear old maximum Courant number:

      courmx = 0.0
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c                                                                      c
c     Step 2: Perform a single timestep direct eulerian hydro update   c
c                                                                      c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
      nn = N
      nnbdy = NBDY
      k = 1-nbdy

      call do_ppmde0_1dc_gamma (xl(k), rho(k), p(k), u(k), rhonu(k),
     ^   pnu(k), unu(k), gamma, dtime, smlrho, smallp, smallu, smalle,
     ^     courmx, nn, nnbdy)
c
c   Increment counters:
       time = time + dtime
       nsteps = nsteps+1

c    Rename for the next timestep (It is assumed that XL is unchanged)
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c                                                                      c
c    Step 3:   Replace old zone averages with new zone averages        c
c                  i preparation of the next timestep.                 c
c                                                                      c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc

      do i=1,n
       rho(i) = rhonu(i)
       u(i) = unu(i)
       p(i) = pnu(i)
      enddo

cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c                                                                      c
c     Step 4: Output                                                   c
c                                                                      c
```

```
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c  A:  print values of updated quantities to screen each timestep

      print *,' '
      print *,' '
      print *,' '
      print *,' Updated results are:'
      print *,' i   xl(i)        rho(i)       p(i)         u(i)    '
      do 9999 i=1,n
        write(*,666) i,xl(i),rho(i),p(i),u(i)
666     format (i3, 1h ,1p, 8(1x,e12.5))
9999  continue
      print *,' ******  NSTEP TIME COURMX  **** = ',nsteps,time,courmx

c  B:  Create disk data files:
c   create some output files for mongo , one every timestep,
c    one every 10 timesteps, one every 25 timesteps:

      ido10 =  nsteps - 10*(nsteps/10)
      ido25 =  nsteps - 25*(nsteps/25)
     write(101,666) i,xl(1), 0.0, 0.0, 0.0
     if (ido10 .eq. 0) write(110,666) i,xl(1), 0.0, 0.0, 0.0
     if (ido25 .eq. 0) write(125,666) i,xl(1), 0.0, 0.0, 0.0
      do i=1,n
      write(101,666) i,xl(i),rho(i),p(i),u(i)
     if (ido10 .eq. 0) write(110,666) i,xl(i),rho(i),p(i),u(i)
     if (ido25 .eq. 0) write(125,666) i,xl(i),rho(i),p(i),u(i)
      enddo
     write(101,666) i,xl(n+1), 0.0, 0.0, 0.0
     if (ido10 .eq. 0) write(110,666) i,xl(n+1), 0.0, 0.0, 0.0
     if (ido25 .eq. 0) write(125,666) i,xl(n+1), 0.0, 0.0, 0.0



c     Go back to 1010 if further timesteps are to be computed:
c
c     if(time .lt. tmax) goto 1010
      if(nsteps .lt. MAXSTEPS) goto 1010

c     close the disk files and stop

      close(101)
      close(110)
      close(125)
c
      stop
      end
c
```

The main program, as provided, is designed to complete 100 timesteps at a fixed timestep of a 128 zone problem. If the reader wished to change these parameters, he should be aware that due to the fixed timestep, `DT`, `NZONES`, and `XMAX` cannot be altered independently. The principle arrays are `XL` for the locations of the lefthand zone interfaces, `RHO` holding the density zone averages, `U` for the velocity zone averages, and `P` for the pressure zone averages. There are also arrays for the updated zone average values returned by the `do_ppmde0_1dc_gamm` template. After initializing several quantities, including what are to be consiered trivial values of density (`SMLRHO`), energy (`SMALLE`), pressure (`SMALLP`),and velocity (`SMALLU`) (used to establish the level of round off and protect agains divides by zero), The arrays are initialized by a call to `SETUP`.

```
c
      subroutine setup( xl, rho, p, u, xmax, gamma, n, nbdy )
c
c   This routine sets up a 1d shock tube problem on a uniform grid. Said
c   grid ranges from xl=0 to xl=xmax.
c
c   xl = locations of lefthand zone interfaces
c  rho = zone averaged density
c    p = zone averaged pressure
c    u = zone averaged velocity
c
c    n = number of real zones in calculation
c nbdy = number of boundary zones added to each end of 1d arrays to
c        implement boundary conditions.
c

      dimension  xl(1-nbdy:n+nbdy+1)
      dimension rho(1-nbdy:n+nbdy)
      dimension   p(1-nbdy:n+nbdy)
      dimension   u(1-nbdy:n+nbdy)

ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c                                                                     c
c     SET initial state:                                              c
c     sod 1-D shock tube                                              c
```

```
c     p left = 1.0     p right = 0.1                                      c
c   rho left = 1.0   rho right = 0.125                                    c
c     u left = 0.0     u right = 0.0                                      c
c     gammaa = 1.4                                                        c
c                                                                         c
c                                                                         c
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc


      deex = xmax / float(N)
      DO I=1,N+1
     xl(i) = deex*float(i-1)
      ENDDO
      DO I=1,N
       IF ( I .LE. N/2 ) THEN
          P(I) = 1.0
          RHO(I) = 1.0
          U(I) =  0.0
        ELSE
          P(I) = 0.1
          RHO(I) = 0.125
          U(I) =  0.0
       ENDIF
      ENDDO
c
      return
      end
```

The problem is a one dimensional shocktube problem in which the gas is initially at rest, with a discontinuity in density and pressure in the middle of the grid. The left state has $\rho = p = 1.0$ while for the right state, $\rho = 0.125$ and $p = 0.1$ (Sod 1978). A uniform computational grid extending from zero to XMAX is also created. Typical results at time 0.025 are shown in figure 15.

After problem initialization the program enters the main hydrodynamics portion of the driver. This consists of an iterative loop comprising four steps:

1) Impose reflecting boundary conditions by filling the fake array elements with values appropriate to a reflecting wall.

2) Perform one timestep by calling the do_ppmde0_1dc_gamma template. This will produce zone averages of density, pressure, and velocity advanced one timestep.

3) replace the values of the old zone average density, pressure and velocity arrays with the results of the hydrodynamic calculation in preparation for the next timestep.

4) Display the results on the screen and store in files on disk.

```
c
      subroutine boundary_refl(xl, rho, u, p, n, nbdy)
c
c   This routine imposed reflecting boundary conditions on
c   1d strips
c
c   xl = locations of lefthand zone interfaces
c  rho = zone averaged density
c    p = zone averaged pressure
c    u = zone averaged velocity
c
c    n = number of real zones in calculation
c nbdy = number of boundary zones added to each end of 1d arrays to
c        implement boundary conditions.
c
      dimension  xl(1-nbdy:n+nbdy+1)
      dimension rho(1-nbdy:n+nbdy)
      dimension   p(1-nbdy:n+nbdy)
      dimension   u(1-nbdy:n+nbdy)

c   Impose reflecting boundaries:
c
c   Left:
        DO I=1-NBDY, 0
        XL(I) = 2.0 * XL(1)  - XL(2 - I)
           RHO(I) = RHO(1-I)
           U(I) = -U(1-I)
           P(I) = P(1-I)
        ENDDO
c
c   Right:
        DO I = N+1, N+NBDY
           XL(I+1) = 2.0 * XL(N+1)  - XL(2*N+1 - I)
           RHO(I) = RHO(2*N+1-I)
           U(I) = -U(2*N+1-I)
           P(I) =  P(2*N+1-I)
        ENDDO

      return
      end
```

The subroutine boundary_refl is very similar to the boundary routine described in the section on interpolation. Boundary conditions are imposed by filling the required number of fake zones abutting each end of the true computational domain with values appropriate to a reflecting wall. The reader should note the change in sign on the velocity.

The ppm template do_ppmde0_1dc_gamma will be described shortly. Its results, new zone averages of density, pressure, and velocity are returned in the arrays RHONU, PNU, and UNU, resp. Note that because the input zone averages are not overwritten during the calculation, this must be done explicitly by the driver in preparation for the next timestep. These new zone averages are also displayed on the screen each timestep, and saved in three disk file, one each timestep, once each ten timesteps, and once each 25 timesteps. Now, highlights from the ppm template will be examined.

Do_PPMDE0_1DC_GAMMA

```
      subroutine do_ppmde0_1dc_gamma
     ^              (xl,      rho,      p,      u,  rhonu,
     ^                pnu,    unu, eosgam,     dt, smlrho,
     ^             smallp, smallu, smalle, courmx,
     ^          n_real_zones_in, nbdy_set)
c
c
```

Input to this template consists of the locations of the lefthand zone interfaces (xl), zone averages of density (rho), pressure (p), and velocity (u). Other input consists of the EOS parameter (eosgam), the time step (dt), and trivial values for density (smlrho), pressure (smallp), velocity (smallu), and energy (smalle). The number of real zones in the calculation is n_real_zones_in and the number of fake zones appended to each end of the input arrays is nbdy_set. It returns updated zone averages of density (rhonu), pressure (pnu), and velocity (unu). Also returned is the maximum Courant number (stability parameter) courmx.

This template calculates one timestep of a direct Eulerian calculation assuming a gamma law gas, uniform grid and planar symmetry. It requires a minimum of 5 fake zones added to each end of the input arrays to account for the boundary zones. This is reflected in the dimensions of the arrays:

```
      dimension       xl(1-nbdy_set:n_real_zones_in + nbdy_set+1),
     ^               rho(1-nbdy_set:n_real_zones_in + nbdy_set),
     ^                 p(1-nbdy_set:n_real_zones_in + nbdy_set),
     ^                 u(1-nbdy_set:n_real_zones_in + nbdy_set),
     ^             rhonu(1-nbdy_set:n_real_zones_in + nbdy_set),
     ^               pnu(1-nbdy_set:n_real_zones_in + nbdy_set),
     ^               unu(1-nbdy_set:n_real_zones_in + nbdy_set)
```

Here, the input parameter nbdy_set must be at least 5. After the allocation of scratch arrays and the calculation of a few necessary quantities, the first PPMLIB module is called.

```
      call  ppm98_Euler0_domain_avg
     ^       (  dx(K),   rho(K),      p(K),      u(K),   ceul(K),
```

```
^       ullrp(K), pllrp(K),  urlrm(K),  prlrm(K),  clagl(K),
^        rhol(K),  rhor(K),   drho(K),   rho6(K),     ul(K),
^          ur(K),    du(K),     u6(K),     pl(K),     pr(K),
^          dp(K),    p6(K), unsmad(K), stpadb(K), unsmup(K),
^      courno(K),       dt,         N,  nnbdy_set)
```

This routine, **PPM98_EULER0_DOMAIN_AVG** uses as input the zone widths (`dx`), zone averages of density (`rho`), pressure (`p`), velocity (`u`) Eulerian sound speed (`ceul`), and the timestep (`dt`). Note that the arrays have an offset equal to `1-nbdy_set`, pointing to the the first fake zone. The number of fake zones sent to **PPM98_EULER0_DOMAIN_AVG** (`nnbdy_set`) is the number appropriate to the subroutine being called *NOT* the number of fake zones required by `Do_PPMDE0_1DC_GAMMA`. Thus the number of zones to process (`N`) is also different to reflect this. The results of this routine are the domain averages of pressure, and velocity over the R^{+} domain of dependence: `ullrp, pllrp` and over the R^{-} domain of dependence: `urlrm, prlrm` and a lefthand zone interface value of the Lagrangian sound speed (`clagl`). Note that since these quantities are all lefthand zone interface values, there will be `N+1` of them. Also returned are the piecewise parabolic interpolation coefficients for density, pressure, and velocity as well as the results of the smoothness calculations and the discontinuity detection calculation results.

This provides a convenient break point at which geometrical terms (not non-planar symmetries) or external force terms may to added to the domain averages following the prescription in Colella and Woodward (1984). The next step is to solve the Riemann problems.

```
 call  PPM98_E_RIEMANN_GAMMA
^       (  dx(K),  ullrp(K),   pllrp(K), urlrm(K), prlrm(K),
^       clagl(K),   rhol(K),     rho(K),  rhor(K),  drho(K),
^        rho6(K),     pl(K),       p(K),    pr(K),    dp(K),
^          p6(K), rhoavl(K),    pavl(K),  uavl(K), dvoll(K),
^      dmassl(K),  dmoml(K),  detotl(K),    gamma,   smlrho,
^         smallp,        dt,  niterl(K), n, nnbdy)
```

The reader will no doubt have noted that array offsets have been adjusted to account for the loss of boundary zones that occurred during the process of determining the domain averages. One of the array inputs to this routine is an array called `niterl`. Recall that the Riemann problems are solved using an iterative process. Using this array, the user may specify the number of iterations at each zone interface. Generally, in smooth regions no iterations are required. Even for strong shocks it is unlikely that more than one or two iterations will be required. The results of the smoothness calculation could be used to decide where extra iterations are to be taken. No extra iterations are taken here because it is assumed that additional numerical diffusion will be added later to such zones anyway, destroying the work of any such additional iterations. The reader is cautioned however, that without additional diffusion of some kind in shocks, additional iterations must be taken, in general.

In addition to the time-averaged values for density (`rhoavl`), pressure (`pavl`), and velocity (`uavl`) found from the solutions to the Riemann problems, **PPM98_E_RIEMANN_GAMMA** also returns the lefthand zone interface fluxes of volume (`dvoll`), mass (`dmassl`), momentum (`dmoml`), and total energy (`detotl`). These fluxes are used to find the preliminary updated zone averages of density, pressure and total energy by a call to the routine **PPM98_DEUL_UPDATE_CART**.

```
      call ppm98_deul_update_cart
     ^     (     dx(K),   dvol(K),      dm(K),    rho(K),      u(K),
     ^      e_total(K), dmassl(K),   dmoml(K),   pavl(K), detotl(K),
     ^        rhonu(K),    unu(K), e_totnu(K),   dmnu(K),    smlrho,
     ^          smallp,    smallu,     smalle,        dt,    nzones)
```

Now, the diffusion section begins. First, the zone centered Eulerian velocities are found by **PPM98_EUL_DIFUSE_1D**.

```
      call ppm98_eul_difuse_1d
     ^            (   dx(k-2), dx(k-1), dx(K), dx(k+1),  dx(k+2),
     ^                 u(k-2), u(k-1), u(K), u(k+1),  u(k+2),
     ^                 p(k-2), p(K), p(k+2), rho(k-2),   rho(K), rho(k+2),
     ^                ceul(k-2), ceul(K), ceul(k+2),
     ^       difuse_vel(K),  smallu,        nzones)
```

The calling statement may at first appear unusual. In order to determine the existence of shocks, this routine must look two zones to the left and two zones to the right. These offset arrays are passed explicitly. Input includes zone widths (`dx`), zone averages of pressure (`p`), velocity (`u`), density (`rho`), and Eulerian sound speed (`ceul`). It should be noted that this input consists of zone averages at the initial time, not the preliminary zone average just determined. From these zone centered diffusion velocities (`difuse_vel`), diffusion fluxes are found:

```
      call ppm98_difuseflux_hydro_cart
     ^    (        dx(K), difuse_vel(K),
     ^        rhonu(K),        unu(K), e_totnu(K),
     ^        dvoll(K),     dmasll(K),  dmasrl(K),
     ^       dmassl(K),      dmoml(K),  detotl(K),
     ^       courno(K),            dt,     nzones,
     ^        nbdy_difuse_fluxes )
```

The fluxes that **PPM98_DIFUSEFLUX_HYDRO_CART** computes are the volume flux (`dvoll`) and the left and right mass fluxes of eqns (3.13a-c) (`dmasll` and `dmasrl`) as well as the net fluxes of mass (`dmassl`), momentum (`dmoml`), and total energy (`detotl`). After this, the final zone average update is done explicitly:

```
      do 9100   i = KSTARTDF2,KENDDF2
c
          dmnew(i) = dmnu(i)
c           this saves a flop or two:
          if ((dvoll(i) + dvoll(i+1)) .ne. zero)   then
             dmnu(i) = dmnew(i) + dmassl(i) - dmassl(i+1)
             rhonu(i) = dmnu(i) * dvolinv(i)
             dmnui = one / dmnu(i)
             unu(i) = (unu(i) * dmnew(i)  +  dmoml(i) - dmoml(i+1))  *  dmnui
             e_totnu(i) = (e_totnu(i) * dmnew(i) +
     ^                                     detotl(i) - detotl(i+1)) * dmnui
9099         continue
         endif
9101  continue
```

Here, in the interests of efficiency, where there are no left or right zone interface diffusive fluxes (that is `dvoll(i)` and `dvoll(i+1)` both are zero) there is no change to the zone average and the arithmetical operations are skipped.

After the maximum Courant number has been determined, the calculation is complete and control returns to the calling routine. The next example shows how this program may be changed to accommodate a non-uniform grid.

Do_PPMDE_1DC_GAMMA

In this template, the requirement of a uniform grid assumed by **Do_PPMDE0_1DC_GAMMA** is relaxed. There is only one significant change. The call to **PPM98_EULER0_DOMAIN_AVG** is replaced by calls to two routines:

```
c   precompute the nonuniform grid interpolation factors

      call ppm98_interpolation_factors
     ^   (   dx(K), dalfac(K), darfac(K),  facda(K), fcdazl(K),
     ^   facdal(K), fa6dal(K), fa6dar(K), frrdal(K), frrdar(K),
     ^   flldal(K), flldar(K), N,nnbdy_set)


      call  ppm98_Euler_domain_avg
     ^       (  dx(K),   dalfac(K),      darfac(K),      facda(K),
     ^      fcdazl(K),   facdal(K),      fa6dal(K),     fa6dar(K),
     ^      flldal(K),   flldar(K),       frrdal(K),    frrdar(K),
     ^         rho(K),        p(K),            u(K),      ceul(K),
     ^       ullrp(K), pllrp(K),  urlrm(K),  prlrm(K),  clagl(K),
     ^        rhol(K),  rhor(K),   drho(K),   rho6(K),     ul(K),
     ^          ur(K),    du(K),     u6(K),     pl(K),     pr(K),
     ^          dp(K),    p6(K), unsmad(K), stpadb(K), unsmup(K),
     ^      courno(K),       dt,        N,  nnbdy_set)
```

The routine **PPM98_INTERPOLATION_FACTORS**, was encountered in the previous section on interpolations. It exists to collect together the various geometrical factors which arise when interpolations on non-uniform grids are performed. By placing them into one routine, efficiency is gained when more than one interpolation is performed on the same nonuniform grid. As is seen here, it is called with the same array offsets as **PPM98_EULER_DOMAIN_AVG**, the nonuniform grid version of **PPM98_EULER0_DOMAIN_AVG**. The calculation then proceeds as before.

The sample driver for this template is `Driverdirecte1dgc`. The only significant difference between this driver and the last is in the subroutine `setup` which now uses a sine function to create a non uniform grid:

```
c
      sinamp = 0.2
      sinfac = 6.28318530717958 / float (N)
      deex = xmax / float(N)
      xl(1) = 0.0
      DO I=2,N+1
        xl(i) = xl(i-1) + deex * ( 1 + sinamp * sin(sinfac*float(i)) )
      ENDDO
```

Do_PPM_DE0_1DC

```
      subroutine do_ppmde0_1dc
     ^        (      xl,     rho,      p,       u,   e_total,    ceul, rhonu,
     ^          e_totnu,     unu,     dt, smlrho,    smallu, smalle, courmx,
     ^          n_real_zones_in, nbdy_set)
```

This template uses the generalized equation of state of eqn (3.5) rather than a gamma law equation of state. As used in this template, the generalized equation of state is

$$p_i = \Phi_i + g_i\, \rho_i\, \varepsilon_i, \tag{3.32}$$

where p is the zone averaged pressure, ρ is the zone averaged density, and ε is the zone averaged internal energy. The two equation of state constants, Φ_i and g_i, must be determined for each zone from the initial data. Input to this template consists of the locations of the lefthand zone interfaces (`xl`), zone averages of density (`rho`), pressure (`p`), velocity (`u`), total energy (`e_total`), and the Eulerian sound speed (`ceul`). Other input consists of the time step (`dt`), and trivial values for density (`smlrho`), velocity (`smallu`), and energy (`smalle`). The number of real zones in the calculation is `n_real_zones_in` and the number of fake zones appended to each end of the input arrays is `nbdy_set`. It returns updated zone averages of density (`rhonu`), total energy (`e_totnu`), and velocity (`unu`). It should be noted that new updated zone-averages of pressure are *not* returned. Also returned is the maximum Courant number (stability parameter) `courmx`.

There are several significant changes from **Do_PPMDE0_1DC_GAMMA**. First, additional input is required in the form of the zone averages of the total energy `e_total` and the Eulerian sound speed `ceul`. These are necessary in order to calculate the generalized equation of state parameters.

```
          cesq(i) = ceul(i) * ceul(i)
          ek = 0.5 * ( u(i)*u(i) )
          ei = e_total(i) - ek
          gamma1(i) = cesq(i) / (ei + p(i)/rho(i) )
```

```
      p00(i) = p(i)  -  gamma1(i) * rho(i) * ei
```

Here `gamma1(i)` and `p00(i)` correspond to g_i and Φ_i in eqn. (3.32), resp. These generalized equation of state parameters are used in the general EOS Riemann Problem solver, **PPM98_E_RIEMANN**.

```
      call  PPM98_E_RIEMANN
     ^       (  dx(K),  ullrp(K),   pllrp(K),  urlrm(K), prlrm(K),
     ^       clagl(K),   rhol(K),     rho(K),   rhor(K),  drho(K),
     ^        rho6(K),     pl(K),       p(K),     pr(K),    dp(K),
     ^          p6(K),    p00(K),  gamma1(K), rhoavl(K),   pavl(K),
     ^        uavl(K),  dvoll(K),  dmassl(K),  dmoml(K), detotl(K),
     ^         smlrho,      dt,   niterl(k),      n, nnbdy)
```

Once gas from neighboring zones has mixed, the general equation of state parameters calculated above are no longer valid. Thus it is not possible to calculate a zone averaged pressure to return to the calling program. It is assumed that the calling program has access to the actual equation of state, and thus able to calculate a new zone averaged pressure and Eulerian sound speed for each zone in preparation for the next timestep.

The driver for this template, `Driverdirecte1d0c` reflects these changes. Additional arrays for the zone averages of Eulerian sound speed and total energy have been added to the existing zone average arrays. Also, an additional step is required after the template has been called to access the true equation of state.

```
      call ifelse(ifdouble,1,d_,)do_ppmde0_1dc (xl(k), rho(k), p(k), u(k), e_total(K), ceul(K),
     ^   rhonu(k), e_totalnu(k), unu(k), dtime, smlrho, smallu, smalle,
     ^     courmx, nn, nnbdy)
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c                                                                  c
c   step 3:                                                        c
c   At the end of DO_PPMDE0_1DC,  new zone averaged pressures and  c
c   sound speeds require  a reference to the actual equation of    c
c   state. In this single gamma-law gas example, this is a trivial c
c   operation. In general, the user would have to supply a routine c
c   which returns new pressures and sound speeds from the given    c
c   new densities and energies.                                    c
c                                                                  c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc

      call eqn_of_state( rhonu(K), unu(k), e_totalnu(k), pnu(k),
     ^                          ceulnu(k), gamma, n, nbdy)
c
c   Increment counters:
       time = time + dtime
       nsteps = nsteps+1

c
c    Rename for the next timestep
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c                                                                      c
c   step 4:  Replace old zone averages with new zone averages          c
c                 i preparation of the next timestep.                  c
c                                                                      c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
```

```
c   At the end of DO_PPMDE0_1DC,  new zone averaged pressures and  c
c   sound speeds require  a reference to the actual equation of    c
      do i=1,n
         rho(i) = rhonu(i)
         u(i) = unu(i)
         p(i) = pnu(i)
         ceul(i) = ceulnu(i)
         e_total(i) = e_totalnu(i)
      enddo
```

After the template, a reference to the true equation of state of the gas is required in order to obtain new zone averages of pressure and Eulerian sound speed from the known averages of density, velocity, and total energy. Here, this is a trivial operation because a gamma law gas was assumed, but this must be a user supplied routine in general. This allows the user to include his own unique knowledge and valuable expertise in the calculation, as only he may do.

Do_PPM_DE0_1DC_PAQ_GAMMA

```
      subroutine do_ppmde0_1dc_paq_gamma
     ^               (xl,      rho,      p,      u,    paq,  rhonu,
     ^               pnu,      unu,  paqnu, eosgam,     dt, smlrho,
     ^            smallp,   smallu, smalle, courmx, nbyvol, nbymass,
     ^   n_real_zones_in, nbdy_set)
c
```

In this template, the concept of passively advected quantities (PAQ) is introduced. A passively advected quantity is some quantity which does not effect the hydrodynamics and is merely transported along with the flow (e.g. smoke or dye) or whose effects can be separated (e.g. transverse momenta). There are two types of passively advected quantities: mass weighted and volume weighted, both of which are allowed.

This template has as input the locations of the lefthand zone interfaces (`xl`), zone averages of density (`rho`), pressure (`p`), velocity (`u`) and an array which contains the zone averages of *all* of the passively advected quantities (`paq`) to be advected. Other input consists of the equation of state parameter (`eosgam`), the time step (`dt`), and trivial values for density (`smlrho`), pressure (`smallp`), velocity (`smallu`), and energy (`smalle`).

As stated above, two classes of passively advected quantities exist: volume weighted and mass weighted. The number of of volume weighted passively advected quantities is `nbyvol` and the number of mass weighted passively advected quantities is `nbymass`. The number of real zones in the calculation is `n_real_zones_in` and the number of fake zones appended to each end of the input arrays is `nbdy_set`. Returned quantities are the updated zone averages of density (`rhonu`), pressure (`pnu`), velocity (`unu`) and the passively advected quantities (`paqnu`).

The initial zone averages of all passively advected quantities are passed as a single two-dimensional array. The fastest running index is the grid index, the second index is the “paq” index. In this way, each PAQ is contiguous in memory. Thus one may think of this as an array of arrays of zone-averaged data. Furthermore, it is assumed that the volume weighted passively advected quantities (if any) precede the mass weighted passively advected quantities (if any). Specifically, the assumed dimensions are

```
[1-NBDY_SET:N_REAL_ZONES_IN+NBDY_SET, 1:(NBYVOL+NBYMASS)]
```

where `NBYVOL` is the number of volume weighted passively advected quantities and `NBYMASS` is the number of mass weighted passively advected quantities. Similarly, the updated PAQ zone averages are returned in a single array, `paqnu`. When using this template, the user must adhere to this convention, or undesirable behaviour may result. The hydrodynamics of this template are essentially those of `Do_PPM_DE0_1DC_GAMMA`. Therefore the discussion here will focus on the passively advected quantities.

In order to advect the PAQs, several quantities must be precomputed. The volume weighted PAQs are advected first, so the fractional volumes, `dvolfl`, are computed.

```
c   Obtain advected volume fractions
c
         do i=KSTART2,KEND2+1
             if( dvoll(i) .le. 0.0 ) then
                dvolfl(i) = -dvoll(i) * dvolinv(i)
             else
                dvolfl(i) = dvoll(i) * dvolinv(i-1)
             endif
                dvolfl(i) = min (1.0, dvolfl(i))
          enddo
```

These may be combined with the (as yet not determined) piecewise parabolic PAQ interpolations to construct fluxes of the passively advected quantities using eqn (2.27-28). The actual preliminary PAQ update is performed by using the advected volume fractions in a call to **PPM98_ADVECT_PASSIVE0** as in the example `ADVECT_2` from the previous section in interpolation.

```
            call ppm98_advect_passive0
     ^              (dvolfl(K),    dvoll(K),    dvol(K),
     ^              dvolinv(K), paq(K,ipaq), smallda(K),
     ^              unsmuta(K),   steepa(K),     dal(K),
     ^           paqnu(K,ipaq),      nzones,      nbdy)
```

The program then proceeds to the mass weighted PAQs by constructing the advected mass fractions, `dmasfl`.

```
c   Obtain advected mass fractions
c
         do i=KSTART2,KEND2+1
             if( dmassl(i) .le. 0.0 ) then
                dmasfl(i) = -dmassl(i) * dminv(i)
             else
                dmasfl(i) = dmassl(i) * dminv(i-1)
             endif
             dmasfl(i) = min (1.0, dmasfl(i))
         enddo
```

The mass weighted PAQs are then updated in the same way as the volume weighted PAQs, also by a call to

PPM98_ADVECT_PASSIVE0.

```
             call ppm98_advect_passive0
     ^          (dmasfl(K),   dmassl(K),      dm(K),
     ^          dmnuinv(K), paq(K,ipaq), smallda(K),
     ^          unsmuta(K),   steepa(K),     dal(K),
     ^       paqnu(K,ipaq),      nzones,        nbdy )
```

The reader should note that in this call the volume terms have been replaced by appropriate mass terms.

Diffusion must also be applied to the passively advected quantities where the hydrodynamical quantities have had additional diffusion applied. Again, this is divided into volume weighted and mass weighted operations. First, the volume diffusion fluxes are supplied by a call to **PPM98_DIFUSEFLUX_PASSIVE**.

```
c   PPM98_DIFUSEFLUX_PASSIVE used the diffusion volumes DVOLL
c     computed above  to compute lefthand zone interface volume
c      weighted interface diffusion fluxes DAL.

          call ppm98_difuseflux_passive
     ^        (paqnu(K,ipaq),  dvoll(K),  dvoll(K), dal(K),
     ^               nzones,      nbdy)
```

Here, input consists of the preliminary updated PAQ zone averages `paqnu` and the diffusive volume flux weighting factors `dvoll`. **PPM98_DIFUSEFLUX_PASSIVE** allows for differing left and right flux weighting factors, as will be needed to the mass weighted PAQs (cf ens. 3.12a-c). Here, because this is a volume weighted quantity, both weighting vectors are identical. The final zone average update is computed explicitly:

```
          do 6800   i = KSTARTDF,KENDDF
              ghoti = paqnu(i,ipaq)
             if ((dvoll(i) + dvoll(i+1)) .ne. zero)   then
               paqnu(i,ipaq) = paqnu(i,ipaq) + (dal(i) - dal(i+1)) * dvolinv(i)
             endif
6800      continue
       enddo
```

The mass weighted PAQs are next. Their fluxes are also found by a call to **PPM98_DIFUSEFLUX_PASSIVE** using the left and right mass flux weighting factors `dmasll` and `dmasrl` returned by **fBPPM98_DIFUSEFLUX_HYDRO_CART**.

```
            call ppm98_difuseflux_passive
     ^      (paqnu(K,ipaq), dmasll(K), dmasrl(K), dal(K), nzones,  nbdy)

        KSTARTDF = KSTARTDF1 + nbdy
```

The final updated zone averages of the mass weighted passively advected diagnostics are then determined.

```
            do 7800   i = KSTARTDF,KENDDF
           ghoti = paqnu(i,ipaq)
               if ((dvoll(i) + dvoll(i+1)) .ne. zero)   then
                  dmnui = one / dmnu(i)
                  paqnu(i,ipaq) = (paqnu(i,ipaq) * dmnew(i) +
     ^                                  dal(i) - dal(i+1)) * dmnui
            endif
7800        continue
c
         enddo
```

As before, only those zones which have diffusive fluxes are modified.

This template is used in the driver `Driverdirecte1dgpaq0c` which is based on the basic driver `Driverdirecte1dg0c`.

The differences involve the addition of a "paq" array to hold the passively advected quantities. This is a *two*-dimensional array. This fist dimension is the spatial dimension `x`, while the second dimension is the "paq" dimension. That is

```
c
      dimension paqnu(1-NBDY:NZONES+NBDY,1:(NBYV+NBYM))
```

where `NZONES` and `NBDY` are the number of real computational zones and the number of fake zones, resp. and `NBYV` and `NBYM` are the number of spatially weighted and inertially weighted paqs, resp. This may be thought of as an array of arrays contiguous in memory.

This driver initializes two paqs of each type. One is a "picket fence" diagnostic consisting of alternating bands of zero and one. The other is a Gaussian centered on the mid point of the computational grid. The parrays are initialized by a call to the subroutine `setpaq.`

```
c
      subroutine  setpaq( xl, paq, nbyv, nbym, n, nbdy )
      ifelse(ifdouble,1,implicit real*8 (a-h,o-z), )
      dimension  xl(1-nbdy:n+nbdy+1)
      dimension  paq(1-nbdy:n+nbdy,nbyv+nbym)
c
c    This routine initializes the zone averages of the passively advected quantities
c    all paq averages are stored in one array, volume weighted paws preceding mass
c    weighted paqs.  The paqs are in a single two dimensional array, with the
```

```
c    second dimension being the "paq" dimension.  Volume weighted paqs are first,
c    mass weighted paqs are second.
c
c    Two paqs of each type are initialized:  A "picket fence" paq which consists of
c    alternating bands of zero and 1. The second is a gaussian centered at the midpoint
c    of the grid.

        xlength = xl(n+1) - xl(1)
        x0 = xl(1) + 0.5*xlength
        gfac = 3.0
        npicket = 16

        if ( nbyv .le. 0 ) goto 500
        do i=1,n
          xm = 0.5*(xl(i+1)+xl(i)) - x0
          paq(i,1) = exp (-(xm*gfac)**2)
        enddo
        if ( nbyv .eq. 1 ) goto 500
         do ipaq = 2,nbyv
             picket = 1.0
             do i=1,n
                if (mod((i-1),npicket) .eq. 0) picket = 1.0 - picket
                paq(i,ipaq) = picket
             enddo
         enddo
c       if ( nbyv .le. 0 ) goto 500
c    the first paq  is a gaussian
500     continue
        if ( nbym .le. 0 ) return
        do i=1,n
          xm = 0.5*(xl(i+1)+xl(i)) - x0
          paq(i,nbyv+1) = exp (-(xm*gfac)**2)
        enddo
        if ( nbym .eq. 1 ) return
         do ipaq = 2,nbym
             picket = 1.0
             do i=1,n
                if (mod((i-1),npicket) .eq. 0) picket = 1.0 - picket
                paq(i,ipaq+nbyv) = picket
             enddo
         enddo
       return
       end
```

The initial values of the picket fence and Gaussian diagnostics are shown in figure 16. A representative plot of these diagnostics at the end of 100 timesteps is shown in figure 17. This dramatically illustrates the ability of the steepening operation to maintain sharp features. It is a suggested exercise for the reader to try this without discontinuity detection applied to the passively advected quantities.

Do_PPM_DE0_2DC_GAMMA

This template uses the concept of passively advected diagnostics introduced in the previous template to construct an algorithm suitable for two-dimensional hydrodynamical calculations. Recall that multi-dimensions may be computed using the technique of directional splitting. In this technique, only changes in the direction of a pass (the longitudinal direction) are computed. Changes in the orthogonal (transverse) directions are computed in subsequent passes, transverse momenta being passively advected during the longitudinal computation.

```
       subroutine do_ppmde0_2dc_gamma
      ^    (  xl, rhozbb,    rho, rhoztt,   pzbb,      p, pztt,
      ^     uzbb,      u,   uztt,  utzbb,   utzb,     ut, utzt,
      ^    utztt,  rhonu,    pnu,    unu,   utnu, eosgam,   dt,
      ^   smlrho, smallp, smallu, smalle, courmx, n_real_zones_in, nbdy_set)
```

This template requires considerably more information than previous templates. In addition to the usual arrays of zone averages of density (`rho`), pressure (`p`), and longitudinal velocity (`u`), zone averages of the transverse velocity (`ut`) are also required. The diffusion calculation, which needs to determine the orientation of any shocks which may be present requires zone averages of these quantities two rows above (suffix `ztt`) and two rows below (suffix `zbb`) the row being updated, as well as zone averages of the transverse velocity one row above (suffix `zt`) and one row below `zb`).

The reader should note carefully that because the velocity has been split into longitudinal and transverse terms, the kinetic energy has also been so split.

```
      gamm1i = 1.0 / ( eosgam - 1.0)
      do i=KSTART2 ,KEND2
         ekl = 0.5 * u(I)*u(I)
         eint = gamm1i * p(i) / rho(i)
         e_total(i) = eint + ekl
      enddo
```

Here the array `e_total` holds only the internal energy (`eint`) and the portion of the kinetic energy due to the longitudinal velocity (`ekl`). Thus the call to **PPM98_DEUL_UPDATE_CART** will only partially update the total energy. The transverse kinetic energy flux terms cannot be determined until the transverse momenta advective fluxes have been determined. This requires first constructing the advected fractions over which the piecewise parabolic interpolations of the transverse velocity will be integrated. Historically, `PPM` has used volume fractions for this purpose rather than mass fractions as would be expected. Volume fractions are used here to maintain compatibility with these earlier versions. The actual advection will be performed using mass weightings however, as the call to **PPM98_ADVECT_PASSIVE0** `shows.`

```
        call ppm98_advect_passive0
     ^            (dvolfl(K),    dmassl(K),    dm(K),
     ^            dmnuinv(K), ut(K), smalldu(K),
     ^            unsmutut(K),   steeput(K),     dmomtl(K),
     ^         utnu(K),      nzones,      nbdy )
```

In addition to providing the preliminary updates of the transverse velocity `utnu`, this also provides the transverse momenta fluxes `dmomtl` which are used to compute the transverse kinetic energy terms to add to the preliminary updated total energy zone averages found previously.

```
      do i=KSTART2 ,KEND2+1
         if (dmassl(i) .eq. 0.0) then
            dektl(i) = 0.0
         else
            dektl(i) = 0.5 * dmomtl(I)*dmomtl(I) / dmassl(i)
         endif
      enddo

      do i=KSTART2 ,KEND2
         ekt = 0.5 * ut(I)*ut(I)
         ektnu = ( ekt * dm(i) + dektl(i) - dektl(i+1))*dmnuinv(i)
```

```
   e_totnu(i) =  e_totnu(i) + ektnu
enddo
```

The diffusion calculation is computationally expensive, so only those zones which are in unsmooth portions of the data (and hence in which it is possible to harbor a discontinuity such as a shock) are checked, that is, wherever the unsmoothness calculation is non-zero. However, in multi dimensions, it is necessary to combine the unsmoothness calculations of the longitudinal and transverse velocities to ensure that those zones in which may roost shock fronts parallel to the direction of the pass will also be checked.

```
do i=KSTART2 ,KEND2
   unsmooth(i) =  unsmutut(i) + unsmup(i)
enddo
```

The zone centered Eulerian diffusion velocities are then found by a call to **PPM98_EUL_DIFUSE0_GAMMA_2D**

```
      call ppm98_eul_difuse0_gamma_2d
     ^           (     uzbb(K),          u(k-2),    u(k-1),   u(K),    u(k+1),
     ^       u(k+2),   uztt(k),      utzbb(K),   utzb(k),    ut(k-2), ut(K),
     ^      ut(k+2),   utzt(k),      utztt(k),   pzbb(K),     p(k-2),  p(K),
     ^        p(k+2),   pztt(k),     rhozbb(K),    rho(k-2), rho(K),  rho(k+2),
     6   rhoztt(k), unsmooth(K), difuse_vel(K),  shockd(K),  eosgam, smallu,
     ^     nzones)
```

This routine uses information above and below the row being updated in order to be able to determine the position and orientation with respect to the grid of any shocks which may be present. Using these zone centered velocities, the hydrodynamic quantities are updated by **PPM98_DIFUSEFLUX_HYDRO_CART**. The reader should note that at this point the total energy does contain both the longitudinal and transverse kinetic energy terms. Therefore, the total energy diffusive fluxes returned by this subroutine are all that are needed to update the total energy zone averages. The transverse momentum diffusive fluxes are found by a call to **PPM98_DIFUSEFLUX_PASSIVE**. These diffusive fluxes are then used for the final updates of the zone averages.

```
      do 9100   i = KSTARTDF,KENDDF
c
          dmnew(i) = dmnu(i)
          if ((dvoll(i) + dvoll(i+1)) .ne. zero)   then
c            dmnew(i) = rhonu(i) * dvol(I)
             dmnu(i) = dmnew(i) + dmassl(i) - dmassl(i+1)
             rhonu(i) = dmnu(i) * dvolinv(i)
```

```
            dmnui = one / dmnu(i)
            unu(i) = (unu(i) * dmnew(i)  +  dmoml(i) - dmoml(i+1))  * dmnui
            utnu(i) = (utnu(i) * dmnew(i)  +  dmomtl(i) - dmomtl(i+1)) * dmnui
            e_totnu(i) = (e_totnu(i) * dmnew(i) +
     ^                                         detotl(i) - detotl(i+1)) * dmnui
         endif
         if ((unu(i) * unu(i)) .lt. ssmlu2)   unu(i) = 0.0
         if ((utnu(i) * utnu(i)) .lt. ssmlu2)   utnu(i) = 0.0
         eknu = 0.5 * (unu(i) * unu(i)  +  utnu(i) * utnu(i))
         e_intnu = max (smalle, (e_totnu(i) - eknu))
         pnu(i) = gamma1 * rhonu(i) * e_intnu
9100  continue
```

The use of this template is illustrated in the basic driver `Driverdirect2dg0c`.

```
      program Driverdirect2dg0c
c
      parameter (NXZONES = 256)
      parameter (NYZONES = 256)
      parameter (MAXSTEPS = 500)
      parameter (EOSGAM = 1.4)
      parameter (DT = 0.001100)
      parameter (XMAX = 1.0)
      parameter (YMAX = 1.0)
      parameter (N2DUMP = 20)
      parameter (T2DUMP = 99999.)
c     Note that because of the fixed timestep, N, XMAX, and DT are not
c               independent.
c
c      NXZONES = number of real computational zones in the x-direction, set
c                by an m4 definition at beginning of code.
c      NYZONES = number of real computational zones in the y-direction, set
c                by an m4 definition at beginning of code.
c        NXZONES and NYZONES should be changed by changing the values of
c         256, 256, and 256 at beginning.
c      MAXSTEPS = maximum number of timpsteps to perform.
c      EOSGAM = the equation of state gamma
c      DT = the fixed timestep
c      XMAX = the righthand edge x-value of the grid.
c                                  The x-grid runs from (0,XMAX)
c      YMAX = the righthand edge y-value of the grid.
c                                  The y-grid runs from (0,YMAX)
c      N2DUMP is the frequency of image dumps in iterations
c      T2DUMP is the frequency of image dumps in program time.
c
c     Do not change this 1.:
      parameter (NBDY = 5)
c
      dimension  xl( 1-NBDY:NXZONES+NBDY+1)
      dimension  yl( 1-NBDY:NYZONES+NBDY+1)
      dimension ddd( 1:4, 1-NBDY:NXZONES+NBDY, 1-NBDY:NYZONES+NBDY )
c
c
c
c     ddd    = the 3-D array into which the 5 principal 2-D arrays of
c              basic physical variables are interleaved in order to
c              optimize memory bandwidth on microprocessor systems.
c
c      xl    = the locations of the lefthand zone interfaces
c      yl    = the locations of the bottom zone interfaces
c     ddd    = the 3-D array into which the 4 principal 2-D arrays of
c              basic physical variables are interleaved in order to
c              optimize memory bandwidth on microprocessor systems.
c     ddd(1,i,j) = rrho(i,j)
c     ddd(2,i,j) = pp(i,j)
c     ddd(3,i,j) = uux(i,j)
c     ddd(4,i,j) = uuy(i,j)
c     rrho   = the zone-averaged densities
c     pp     = the zone-averaged pressures
c     uux    = the zone-averaged x-velocities
c     uuy    = the zone-averaged y-velocities
c
c
c     smlrho = a trivial value of density, used to protect divides.
c     smalle = a trivial value of energy, used to protect divides.
c     smallp = a trivial value of pressure, used to protect divides.
c     smallu = a trivial value of velocity, used to protect divides.
c
      smlrho = 1.0e-06
      smalle = 1.0e-06
```

```
      smallp = 1.0e-06
      smallu = 1.0e-06
c
c     initialize the grid:
      xlimit = XMAX
      ylimit = YMAX
c
      call setgrid(  xl,  yl, xlimit, ylimit, nxzones, nyzones, nbdy )
c
c     initialize the hydrodynamic quantities:
c
      call set2dshock( ddd, nxzones, nyzones, nbdy )
c
      nstop = MAXSTEPS
      tstop = float (MAXSTEPS) * DT
      ndump = N2DUMP
      tdump = T2DUMP

c          nstop is the maximum number of iterations to take for the entire
c             calculation
c          tstop is the maximum program time for the entire calculation, here
c             assuming a constant timestep DT
c          ndump is the frequency of image dumps in iterations
c          tdump is the frequency of image dumps in program time.
c
c    CONTROL performs the entire hydrodynamic calculation.
c
      dtime = DT
      gamma = EOSGAM

      call cntrol ( xl,  yl, ddd, gamma, dtime, smlrho, smalle, smallp,
     ^             smallu, ndump, tdump, nstop, tstop, nxzones, nyzones, nbdy)

      stop
      end
```

The main program, `Driverdirect2dg0c`, is deliberately simple. It performs a specified number of timesteps of a two-dimensional problem, with a fixed timestep and problem size. There are three arrays which are allocated in the main program. Two define the computational grid by specifying the locations of the left- and right-hand zone interfaces, `xl` and `yl`, resp. The remaining array contains the zone averages of all the computed quantities which, for this problem, are densiy, pressure, x-, and y-velocity. These are stored in the array `ddd` in *interleaved order*, that is, the zone averages of density, pressure, and the two velocities for a zone *(i,j)* are contiguous in memory. making `ddd` a *three*-dimensional array, with the first dimension being the number of zone-averaged quantities in the array. For example, the density, pressure, x- and y-velocity for zone *(i,j)* are `ddd(1,i,j)`, `ddd(2,i,j)`, `ddd(3,i,j)`, `ddd(4,i,j)`, resp. This interleaving provides greater efficiency on cache-based computers.

```
      dimension  xl( 1-NBDY:NXZONES+NBDY+1)
      dimension  yl( 1-NBDY:NYZONES+NBDY+1)
      dimension ddd( 1:4, 1-NBDY:NXZONES+NBDY, 1-NBDY:NYZONES+NBDY )
```

The number of actual computational zones in the x- and y- directions are given by the parameters `NXZONES` and `NYZONES`, resp. An additional `NBDY` zones are added to the left and right of each dimension to allow for the imposition of boundary conditions. Consider that in order to produce the updated zone averages for any given zone, the hydrodynamic subroutine requires information from surrounding zones. Ordinarily this would mean that the zones along the edges of the computational domain could not be updated due to the lack of information corresponding to the zones not present. Recall that this is avoided by appending a sufficient number of fake zones, as if a picture frame, around the computational domain. These fake zones will be filled with values appropriate to the user's chosen boundary conditions. This allows zones on the boundary of the *problem* to the treated as internal zones.

The arrays are initialized in two steps. First the computational grid is set by a call to the subroutine `setgrid`. This creates a uniform computational grid running from 0.0 to 1.0 in both dimensions. The reader should note that a computational grid of `N` zones would require `N+1` lefthand zone interface locations.

```
      subroutine setgrid(  xl,  yl, xmax, ymax, nx, ny, nbdy )
c
c
c
```

```
c   This routine initializes the locations of the lefthand zone interfaces in
c     both the x- and y-directions, assuming a uniform grid.  The grid consists
c     of NX zones (defined by NX+1 lefthand zone interfaces) in the x-direction
c     ranging from 0.0 to XMAX, with NBDY zones added to each end to allow for
c     boundary conditions.  there are NY zones (defined by NY+1 lefthand zone
c     interfaces) in the y-direction ranging from 0.0 to YMAX, with NBDY zones
c     added to each end to allow for boundary conditions.
c
c      xl    = the x-locations of the lefthand zone interfaces.
c      yl    = the y-locations of the lefthand zone interfaces.
c
c     xmax  = The real (non boundary) portion of the grid is
c               XXL(1)=0.0 to XXL(NX+1)=XMAX.
c     ymax  = The real (non boundary) portion of the grid is
c               YYL(1)=0.0 to YYL(NY+1)=YMAX.
c
c    nx = number of real zones in in the x-direction
c    ny = number of real zones in in the y-direction
c  nbdy = number of boundary zones added to each end of the arrays to
c        implement boundary conditions.

      dimension   xl(1-nbdy:nx+nbdy+1)
      dimension   yl(1-nbdy:ny+nbdy+1)

c
      deex = xmax / float(NX)
      DO I=1,NX+1
         xl(i) = deex*float(i-1)
      ENDDO

      deey = ymax / float(NY)
      DO I=1,NY+1
         yl(i) = deey*float(i-1)
      ENDDO

      return
      end
```

Next, the zone averages are initialized by `set2dshock`. The initial conditions are those of a "two-dimensional shocktube", which is a rectangular computational domain divided into four smaller rectangles or subdomains, with all four meeting at the point (nxdisc, nydisc) near the center. Each subdomain consists of a constant hydrodynamic state. After problem initialization, the program flow proceeds to `cntrol` which will perform the rest of the calculation.

```
      subroutine set2dshock( ddd, nx, ny, nbdy )
c
c     This routine sets the initial values of all hydrodynamic
c     quantities for BKE's 2d shock tube problem.  The initial
c     state consists of 4. different constant states:
c
c   rhoq2 = 2.0        rhoq1 = 1.0
c     pq2 = 2.0          pq1 = 1.0
c    uxq2 = 0.0         uxq1 = 0.0
c    uyq2 = 0.0         uyq1 = 0.0
c
c   rhoq3 = 0.5        rhoq4 = 0.1
c     pq3 = 0.5          pq4 = 0.1
c    uxq3 = 0.0         uxq4 = 0.0
c    uyq3 = 0.0         uyq4 = 0.0
c        .....................................................
c        .                       .                           .
c        .                       .                           .
c        .                       .                           .
c        .      Q2               .            Q1             .
c        .                       .                           .
c        .                       .                           .
c        .....................................................  < nydisc
c        .                       .                           .
c        .                       .                           .
c        .                       .                           .
c        .                       .                           .
c        .      Q3               .            Q4             .
c        .                       .                           .
c        .                       .                           .
c        .                       .                           .
c        .                       .                           .
c        .....................................................
c                                ^
c                              nxdisc
c
c   assuming a gamma law equation of state: p = (gamma-1)*rho*ei
c
c
c     ddd    = the 3-D array into which the 4 principal 2-D arrays of
c              basic physical variables are interleaved in order to
c              optimize memory bandwidth on microprocessor systems.
c     ddd(1,i,j) = rrho(i,j)
```

```
c      ddd(2,i,j) = pp(i,j)
c      ddd(3,i,j) = uux(i,j)
c      ddd(4,i,j) = uuy(i,j)
c
       dimension ddd( 1:4, 1-NBDY:NX+NBDY, 1-NBDY:NY+NBDY )
c

       parameter( rhoq1 = 1.0)
       parameter(   pq1 = 1.0)
       parameter(  uxq1 = 0.0)
       parameter(  uyq1 = 0.0)

       parameter( rhoq2 = 2.0)
       parameter(   pq2 = 2.0)
       parameter(  uxq2 = 0.0)
       parameter(  uyq2 = 0.0)

       parameter( rhoq3 = 0.5)
       parameter(   pq3 = 0.5)
       parameter(  uxq3 = 0.0)
       parameter(  uyq3 = 0.0)

       parameter( rhoq4 = 0.1)
       parameter(   pq4 = 0.1)
       parameter(  uxq4 = 0.0)
       parameter(  uyq4 = 0.0)

c
c    Location of discontinuity in the x-direction:
       nxdisc = nx/4
c    Location of discontinuity in the y-direction:
       nydisc = (4*ny)/5

       do 100   k = nydisc,ny+nbdy
       do 100   i = nxdisc,nx+nbdy
       ddd(1,i,k) = rhoq1
       ddd(2,i,k) = pq1
       ddd(3,i,k) = uxq1
       ddd(4,i,k) = uyq1
100    continue
c
       do 200   k = nydisc,ny+nbdy
       do 200   i = -nbdy+1,nxdisc-1
       ddd(1,i,k) = rhoq2
       ddd(2,i,k) = pq2
       ddd(3,i,k) = uxq2
       ddd(4,i,k) = uyq2
200    continue
c
       do 300   k = 1-nbdy,nydisc-1
       do 300   i = -nbdy+1,nxdisc-1
       ddd(1,i,k) = rhoq3
       ddd(2,i,k) = pq3
       ddd(3,i,k) = uxq3
       ddd(4,i,k) = uyq3
300    continue
c
       do 400   k = 1-nbdy,nydisc-1
       do 400   i = nxdisc,nx+nbdy
       ddd(1,i,k) = rhoq4
       ddd(2,i,k) = pq4
       ddd(3,i,k) = uxq4
       ddd(4,i,k) = uyq4
400    continue
c
c
c
c
       return
       end
```

The subroutine, `cntrol` is essentially an iterative loop consisting of two steps. First, the hydrodynamical calculation is advanced two time steps by a call to `runhyd`, which will be discussed shortly. After `runhyd` has updated the zone averages, an image file is created by a call to `dump`, when appropriate. This latter routine constructs a 2 by 2 composite image of density, pressure, and the x- and y-components of velocity, with each quantity independently scaled, and writes it to disk as a binary file, one byte per pixel. Hence its size is 4x`NX`x`NY` bytes. This is suitable for viewing by *xraz* or the user may convert it to his own format of choice. More flexible modes of data output are described in volume II of this series entitled, ‘‘I/O and Data Analysis.’’

```
       subroutine cntrol ( xl,  yl, ddd, gamma, dtime, smlrho, smalle,
      ^       smallp, smallu, ndump, tdump, nstop, tstop, nx, ny, nbdy)
c
```

```
      dimension  xl( 1-NBDY:NX+NBDY+1)
      dimension  yl( 1-NBDY:NY+NBDY+1)
      dimension ddd( 1:4, 1-NBDY:NX+NBDY, 1-NBDY:NY+NBDY )
c
c
c     This routine runs the entire hydro calculation.
c
c
c
c     ddd    = the 3-D array into which the 5 principal 2-D arrays of
c              basic physical variables are interleaved in order to
c              optimize memory bandwidth on microprocessor systems.
c     nx     = the number of real zones in the x-dimension for this
c              tile of the computational domain.
c     ny     = the number of real zones in the y-dimension for this
c              tile of the computational domain.
c     nbdy   = the number of fake zones needed to implement boundary
c              conditions so that in the hydrodynamical calculation
c              all zones can be treated as if they were interior zones.
c
c
c     dtime  = the time step.
c     courmx = the maximum value of the Courant number for this tile
c              of the computational domain.
c     gamma  = the value of the ratio of specific heats, as in the
c              gamma-law equation of state:  p = (gamma-1)*rho*ei
c
c     smlrho = a trivial value of density, used to protect divides.
c     smalle = a trivial value of energy, used to protect divides.
c     smallp = a trivial value of pressure, used to protect divides.
c     smallu = a trivial value of velocity, used to protect divides.
c
c      xl    = the locations of the lefthand zone interfaces
c      yl    = the locations of the bottom zone interfaces
c     ddd    = the 3-D array into which the 5 principal 2-D arrays of
c              basic physical variables are interleaved in order to
c              optimize memory bandwidth on microprocessor systems.
c     ddd(1,i,j) = rrho(i,j)
c     ddd(2,i,j) = pp(i,j)
c     ddd(3,i,j) = uux(i,j)
c     ddd(4,i,j) = uuy(i,j)
c     rrho   = the zone-averaged densities for this tile.
c     pp     = the zone-averaged pressures for this tile.
c     uux    = the zone-averaged x-velocities for this tile.
c     uuy    = the zone-averaged y-velocities for this tile.
c
c
c    Create an image of the initial state. The initial state will be
c     sequence number 0.

       numdmp = 0
      call dump (ddd, numdmp, nx, ny, nbdy)
c          numdmp = the sequential number of the image 0->whatever, numdmp
c          will be automatically incremented after write the image.
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c                                                                     c
c      The 1000 loop advances the calculation by 2. timesteps per    c
c                            iteration                                c
c                                                                     c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c   initialize counters
c
      ncycle = 0
      time = 0.0
      ntodmp = ndump
      ttodump = tdump
c
1000  continue
c
c   clear the old maximum courant number in preparation of the next pair of
c       timesteps:
      courmx = 0.
c
c   RUNHYD performs a symmetrized pair of passes x-y-y-x which
c          equals 2. complete timesteps.
c
      call runhyd ( xl,  yl, ddd, dtime, courmx, gamma,
     1             smlrho, smalle, smallp, smallu,
     2             nx, ny, nbdy)
c
c
c
c   update counters
c
      ncycle = ncycle + 2
      ntodmp = ntodmp - 2
      time = time + dtime + dtime
c
      write (6,1710)   ncycle, time, courmx, dtime
c
c   Check to see if time to write am image file.  An image is
c   written every NDUMP steps or at intervals of DTDUMP in
c   problem time.
c
      if (ntodmp)   1600,1600,1500
1500  if (time - tdump)   1700,1600,1600
```

```
1600  ntodmp = ndump
      ttodump = ttodump + tdump

      call dump (ddd, numdmp, nx, ny, nbdy)
c     numdmp = the sequential number of the image 0->whatever, numdmp
c           will be automatically incremented after write the image.

      call flush (6)
c
1700  continue
1710  format (1h ,3hn =,i5,4x,3ht =,
     1        1pe12.5,4x,13hcourant no. =,e12.5,4x,4hdt =,e12.5)
c
c   Check to see if time to stop.  Progam execution ends when either
c    the maximum number of timesteps, NSTOP,  has been computed or
c    when the maximum problem time, TSTOP, has been reached.
c
      if (ncycle .ge. nstop) stop
      if (time .ge. tstop) stop
      goto 1000
      end
```

In dump a file name is constructed based upon two supplied one character strings and the sequential number of the image. The desired quantities are extracted from ddd, without the fake zones, individually scaled, and composited in a one byte array igray.. At this point the bytes comprising the composite image are now contiguous in memory in this array of bytes. The disk file is opened using **ppm98_binopen**, and the composite image values written to it by **ppm98_binwrite**, after which the file is closed by **ppm98_binclose**. The image sequence number is then incremented in preparation for the next call.

```
      subroutine dump (ddd, numdmp, nx, ny, nbdy )
c
c   we are counting on the F77 compiler to automatically allocate/deallocate
      character*1 igray(2*nx,2*ny)
      character*1 acharc, acharc2
      character*8 namdmp
c
c
c     ddd    = the 3-D array into which the 4 principal 2-D arrays of
c              basic physical variables are interleaved in order to
c              optimize memory bandwidth on microprocessor systems.
c     ddd(1,i,j) = rrho(i,j)
c     ddd(2,i,j) = pp(i,j)
c     ddd(3,i,j) = uux(i,j)
c     ddd(4,i,j) = uuy(i,j)
c     rrho   = the zone-averaged densities.
c     pp     = the zone-averaged pressures.
c     uux    = the zone-averaged x-velocities.
c     uuy    = the zone-averaged y-velocities.
c
c     numdmp = the sequential number of the image 0->whatever, numdmp
c           will be automatically incremented after write the image.
c      nx = number of real computational zones in the x-direction, set
c              by an m4 definition at beginning of code.
c      ny = number of real computational zones in the y-direction, set
c              by an m4 definition at beginning of code.
c
c      nbdy = the number fake zones necessary for boundary conditions.
c
      dimension ddd( 1:4, 1-NBDY:NX+NBDY, 1-NBDY:NY+NBDY )
c
c
      acharc = 'PREFIX'
      acharc2 = 'e'
c
c     determine individual min/max for scaling
c
      rhomin = ddd(1,1,1)
      rhomax = ddd(1,1,1)
      pmin = ddd(2,1,1)
      pmax = ddd(2,1,1)
      uxmin = ddd(3,1,1)
      uxmax = ddd(3,1,1)
      uymin = ddd(4,1,1)
      uymax = ddd(4,1,1)
      do 1000   j = 1,ny
      do 1000   i = 1,nx
      ii = ii + 1
       rhomin = min( rhomin,ddd(1,i,j))
       rhomax = max( rhomax,ddd(1,i,j))
       pmin = min( pmin,ddd(2,i,j))
       pmax = max( pmax,ddd(2,i,j))
       uxmin = min( uxmin,ddd(3,i,j))
```

```
      uxmax = max( uxmax,ddd(3,i,j))
      uymin = min( uymin,ddd(4,i,j))
      uymax = max( uymax,ddd(4,i,j))
1000  continue
c
c   create scaling factors
c
      denom = (rhomax - rhomin)
      if( denom .ne. 0.0 ) denom = 1.0/denom
      rhofac = 255.0 * denom
      denom = (pmax - pmin)
      if( denom .ne. 0.0 ) denom = 1.0/denom
      pfac = 255.0 * denom
      denom = (uxmax - uxmin)
      if( denom .ne. 0.0 ) denom = 1.0/denom
      uxfac = 255.0 * denom
      denom = (uymax - uymin)
      if( denom .ne. 0.0 ) denom = 1.0/denom
      uyfac = 255.0 * denom
c
      irhooff = 0
      ipoff = nx
      iuxoff = 2*nx*ny
      iuyoff = 2*nx*ny + nx
      do j = 1,2*ny
         do i = 1,2*nx
            igray(i,j) = char(100)
         enddo
         enddo
      ii = 0
      do j = 1,ny
         do i = 1,nx
         ii = ii + 1
            rthyng = rhofac * (ddd(1,i,j) - rhomin)
            pthyng = pfac * (ddd(2,i,j) - pmin)
            uxthyng = uxfac * (ddd(3,i,j) - uxmin)
            uythyng = uyfac * (ddd(3,i,j) - uymin)
            igray(i,j) = char(int(rthyng))
            igray(i+nx,j) = char(int(pthyng))
            igray(i,j+ny) = char(int(uxthyng))
            igray(i+nx,j+ny) = char(int(uythyng))
         enddo
      enddo
c
c
c
c   Construct name of image file and store in NAMDMP:
c
      write (namdmp,1004)   acharc,  acharc2, numdmp
1004  format (a1,a1,'im',i4.4)
       print *,' namdmp=',namdmp
c

c
c   now write it to file namdmp
      iunit = 34
      nbytes = 4*nx*ny
      call ppm98_binopen (namdmp, iunit)
      call ppm98_binwrite (iunit, igray, nbytes)
      call ppm98_binclose (iunit)
c


      numdmp = numdmp + 1

c
c
      return
      end
```

The routine runhyd performs the actual work of the hydrodynamical calculation in a sequence of four passes or iterative loops, first an x-pass, two y-passes, and a final x-pass to complete a symmetrized pair of timesteps. Each pass consists of two parts: application of boundary conditions followed by the actual hydrodynamic calculation. These passes are similar in nature, so the discussion will be restricted to the first x-pass.

```
      subroutine runhyd ( xl,  yl, ddd, dt,
     1                   courmx, gamma, smlrho, smalle, smallp, smallu,
     2                   nx, ny, nbdy)
c

      dimension  xl( 1-NBDY:NX+NBDY+1)
      dimension  yl( 1-NBDY:NY+NBDY+1)

      dimension    ddd(4,1-nbdy:nx+nbdy,1-nbdy:ny+nbdy)
c   we are counting on the F77 compiler to automatically allocate/deallocate
```

```
c
c   in these arrays, so the dimension nbdy+nx+ny is
c   a little wasteful, but ensures that enough space has been
c   allocated.
      dimension
     1      rhonu(1-nbdy:nbdy+nx+ny),
     ^        pnu(1-nbdy:nbdy+nx+ny),
     ^        u1nu(1-nbdy:nbdy+nx+ny),
     ^        u2nu(1-nbdy:nbdy+nx+ny)
c
      dimension
     1      xxl(1-nbdy:nbdy+nx+ny+1),
     ^        dddslab(1-nbdy:nbdy+nx+ny,5,4)
c
c
c
c     nbdy   = the number of fake zones needed to implement boundary
c              conditions so that in the hydrodynamical calculation
c              all zones can be treated as if they were interior zones.
c     nx     = the number of real zones in the x-dimension for this
c              tile of the computational domain.
c     ny     = the number of real zones in the y-dimension for this
c              tile of the computational domain.
c     iy0x1  = a flag which is 0 in the y-pass and 1 in the x-pass.
c
c
c     dt     = the time step.
c     courmx = the maximum value of the Courant number for this tile
c              of the computational domain.
c     gamma  = the value of the ratio of specific heats, as in the
c              gamma-law equation of state:  p = (gamma-1)*rho*ei
c
c     smlrho = a trivial value of density, used to protect divides.
c     smalle = a trivial value of energy, used to protect divides.
c     smallp = a trivial value of pressure, used to protect divides.
c     smallu = a trivial value of velocity, used to protect divides.
c
c      xl    = the x-grid for this tile.
c      yl    = the y-grid for this tile.
c     rrho   = the zone-averaged densities for this tile.
c     pp     = the zone-averaged pressures for this tile.
c     uux    = the zone-averaged x-velocities for this tile.
c     uuy    = the zone-averaged y-velocities for this tile.
c     ddd    = the 3-D array into which the 5 principal 2-D arrays of
c              basic physical variables are interleaved in order to
c              optimize memory bandwidth on microprocessor systems.
c     ddd(1,i,j) = rrho(i,j)
c     ddd(2,i,j) = pp(i,j)
c     ddd(3,i,j) = uux(i,j)
c     ddd(4,i,j) = uuy(i,j)
c
c
c
c
10    continue
c
      courmx = 0.
      maxdim = max (nx, ny)
c
c
c     NOW BEGIN LOOP OVER 2 TIME STEPS.
c     These 2 time steps are symmetrized;  they both use the same
c     time step value, and they perform the x- and y-sweeps in
c     opposite orders.
c
      iy0x1 = 1
c     Set fake zones according to boundary conditions.
      call bdrys ( xl,  yl, ddd, nx, ny, nbdy, iy0x1)
c
c
c(
c
c=====================================================================72
c                       -----  x-pass  -----
c
c   This is an m4 macro definition which will construct a pass
c   in the x direction.  It is invoked by, for example,
c    P_A_S_S( x, y )  (without the "_")
c    in which case, all instances of "x" will be replaced with "x", and
c    all "y  " by "y".
c
c   Why has this been done? In order to update any given row, PPM  requires
c     information from up to 2. rows above and below the row of interest.
c     This information must be from the same computational time, however,
c     as the computation progresses, the large storage array DDD becomes
c     a mixture of old timestep information and new timestep information
c     as updated zone averages (rhonu, pnu, unu, etc) are stored in DDD.
c     Therefore, a small working scratch space of 5. rows (a slab) will
c     be created. Consider an X-pass, then the slab would be dimensioned
c     (1-nbdy:nx+nbdy,5):
c
c              ----------------------------------------
c      islab=5 |                                      >
c              ----------------------------------------
c      islab=4 |                                      >
c              ----------------------------------------
c      islab=3 |                                      >
c              ----------------------------------------
```

```
c      islab=2 |                                       >
c              ---------------------------------------
c      islab=1 |                                       >
c              ---------------------------------------
c
c   initially, islab=1 is taken from ddd(*,*,-1)
c   initially, islab=2 is taken from ddd(*,*,0)
c   initially, islab=3 is taken from ddd(*,*,1)
c   initially, islab=4 is taken from ddd(*,*,2)
c   initially, islab=5 is taken from ddd(*,*,3)
c
c   Pointers to the correct slab rows (slabbb, slabb, slab, slabt, slabtt)
c   (for  2rows below, 1 row below, the row, 1. row above, 2 rows above)
c   which would have the values (1,2,3,4,5).  After the call to ppm and the
c   updated values stored in DDD, the counters are cycled to (2,3,4,5,1)
c   and the required data from DDD is written into the row pointed to
c   by the new value of slabtt.
c
c   The user should also note that the velocities switch roles.  That is,
c   during an X-pass, u1 is the x-velocity and u2 is the y-velocity while
c   during a Y-pass, u1 is the y-velocity and u2 is the x-velocity.
c
c   This reduces both the scratch space requirement and the amount of
c   data copying required.
c
c   the xvelocity is element #3 in DDD
c   the yvelocity is element #4 in DDD
      kvx = 3
      kvy = 4

      do i = 1-nbdy, nx+nbdy+1
      xxl(i) = xl(i)
      enddo
c
c   initialize first 4 rows in slab
c
c     ((do 100)
      do islab = 1, 4
       iy   = islab-2
            do ix = 1-nbdy, nx+nbdy
              dddslab(ix, islab,1) =  ddd(   1, ix, iy)
              dddslab(ix, islab,2) =  ddd(   2, ix, iy)
              dddslab(ix, islab,3) =  ddd(kvx, ix, iy)
              dddslab(ix, islab,4) =  ddd(kvy  , ix, iy)
            enddo
      enddo
c     (do 100))

c
c     ((do 400)
      do iy   = 1, ny
c        find pointers to proper rows in the slab to where
c         the row 2. zones below are stored (slabbb) 1.
c         row below (slabb), etc
         islabbb = 1+mod(iy  -1,5)
         islabb  = 1+mod(iy  ,  5)
         islab   = 1+mod(iy  +1,5)
         islabt  = 1+mod(iy  +2,5)
         islabtt = 1+mod(iy  +3,5)
c
c   At this point, we are ALWAYS able to fill position slabtt from ddd
         ly   = iy  +2
            do ix = 1-nbdy, nx+nbdy
            lx = ix
              dddslab(ix, islabtt,1) =  ddd(   1, lx, ly)
              dddslab(ix, islabtt,2) =  ddd(   2, lx, ly)
              dddslab(ix, islabtt,3) =  ddd(kvx, lx, ly)
              dddslab(ix, islabtt,4) =  ddd(kvy  , lx, ly)
            enddo

         ip = 1-nbdy
             d2inv = 4. / (y  l(iy  +1)+y  l(iy  +2)-y  l(iy  -1)-y  l(iy  ))
         xcourmx = 0.
      call  do_ppmde0_2dc_gamma( xxl(ip),
     ^ dddslab(ip,islabbb,1),  dddslab(ip,islab,1),   dddslab(ip,islabtt,1),
     ^ dddslab(ip,islabbb,2),  dddslab(ip,islab,2),   dddslab(ip,islabtt,2),
     ^ dddslab(ip,islabbb,3),   dddslab(ip,islab,3),    dddslab(ip,islabtt,3),
     ^  dddslab(ip,islabbb,4),   dddslab(ip,islabb,4),    dddslab(ip,islab,4),
     ^             dddslab(ip,islabt,4),   dddslab(ip,islabtt,4),
     ^     rhonu(ip),     pnu(ip),      u1nu(ip),    u2nu(ip),
     ^   gamma,         dt,        smlrho,      smallp,     smallu,
     ^       smalle,    xcourmx,            nx, nbdy)
c     call do_ppmde0_2dc_gamma ( xl(L)
c    ^ rrho(L,k-2),  rrho(L,k),  rrho(L,k+2),
c    ^   pp(L,k-2),    pp(L,k),    pp(L,k+2),
c    ^  uux(L,k-2),   uux(L,k),   uux(L,k+2),
c    ^  uuy(L,k-2), uuy(L,k-1),      uuy(L,k), uuy(L,k+1), uuy(L,k+2),
c    ^    rhonu(L),      pnu(L),       uxnu(L),    uynu(L),
c    ^       gamma,          dt,        smlrho,      smallp,     smallu,
c    ^      smalle,    xcourmx,            nx, nbdy)

c     call  do_ppmlr0_2dc_gamma ( xxl(ip),
c    ^ dddslab(ip,islabbb,1),  dddslab(ip,islab,1),   dddslab(ip,islabtt,1),
c    ^ dddslab(ip,islabbb,2),  dddslab(ip,islab,2),   dddslab(ip,islabtt,2),
c    ^ dddslab(ip,islabbb,3),   dddslab(ip,islab,3),    dddslab(ip,islabtt,3),
c    ^  dddslab(ip,islabbb,4),   dddslab(ip,islabb,4),    dddslab(ip,islab,4),
c    ^             dddslab(ip,islabt,4),   dddslab(ip,islabtt,4),
c    ^     rhonu(ip),     pnu(ip),      u1nu(ip),    u2nu(ip),
```

```
c    ^   gamma,         dt,       smlrho,     smallp,     smallu,
c    ^      smalle,    xcourmx,           nx, nbdy)

c     print *,' iy  , xcourmx=', iy  , xcourmx

c
      courmx = max ( courmx, xcourmx )
c
         do ix = 1, nx
         ddd(   1,ix,iy) = rhonu(ix)
         ddd(   2,ix,iy) =  pnu(ix)
         ddd(kvx,ix,iy) = u1nu(ix)
         ddd(kvy  ,ix,iy) = u2nu(ix)
         enddo
c

      enddo
c     (do 400))

c
c
c
c )
c
66    format(1h ,i3,1p,4(1x,e10.3))

       iy0x1 = 0
c     Set fake zones according to boundary conditions.
          call bdrys ( xl,  yl, ddd, nx, ny, nbdy, iy0x1)
c
c
c(
c
c=====================================================================72
c                      -----  y-pass  -----
c
c
c   the xvelocity is element #3 in DDD
c   the yvelocity is element #4 in DDD
      kvx = 3
      kvy = 4

      do i = 1-nbdy, ny+nbdy+1
      xxl(i) = yl(i)
      enddo
c
c   initialize first 4 rows in slab
c
c     ((do 100)
      do islab = 1, 4
       ix   = islab-2
            do iy = 1-nbdy, ny+nbdy
              dddslab(iy, islab,1) =  ddd(   1, ix, iy)
              dddslab(iy, islab,2) =  ddd(   2, ix, iy)
              dddslab(iy, islab,3) =  ddd(kvy, ix, iy)
              dddslab(iy, islab,4) =  ddd(kvx  , ix, iy)
            enddo
      enddo
c     (do 100))

c
c     ((do 400)
      do ix   = 1, nx
c        find pointers to proper rows in the slab to where
c         the row 2. zones below are stored (slabbb) 1.
c         row below (slabb), etc
         islabbb = 1+mod(ix  -1,5)
         islabb  = 1+mod(ix  ,  5)
         islab   = 1+mod(ix  +1,5)
         islabt  = 1+mod(ix  +2,5)
         islabtt = 1+mod(ix  +3,5)
c
c   At this point, we are ALWAYS able to fill position slabtt from ddd
          lx   = ix  +2
            do iy = 1-nbdy, ny+nbdy
             ly = iy
              dddslab(iy, islabtt,1) =  ddd(   1, lx, ly)
              dddslab(iy, islabtt,2) =  ddd(   2, lx, ly)
              dddslab(iy, islabtt,3) =  ddd(kvy, lx, ly)
              dddslab(iy, islabtt,4) =  ddd(kvx  , lx, ly)
            enddo

         ip = 1-nbdy
             d2inv = 4. / (x  l(ix  +1)+x  l(ix  +2)-x  l(ix  -1)-x  l(ix  ))
         ycourmx = 0.
      call  do_ppmde0_2dc_gamma( xxl(ip),
     ^ dddslab(ip,islabbb,1),  dddslab(ip,islab,1),   dddslab(ip,islabtt,1),
     ^ dddslab(ip,islabbb,2),  dddslab(ip,islab,2),   dddslab(ip,islabtt,2),
     ^ dddslab(ip,islabbb,3),   dddslab(ip,islab,3),    dddslab(ip,islabtt,3),
     ^  dddslab(ip,islabbb,4),   dddslab(ip,islabb,4),    dddslab(ip,islab,4),
     ^             dddslab(ip,islabt,4),   dddslab(ip,islabtt,4),
     ^     rhonu(ip),     pnu(ip),      u1nu(ip),    u2nu(ip),
     ^   gamma,         dt,       smlrho,     smallp,     smallu,
     ^      smalle,    ycourmx,           ny, nbdy)
c     call do_ppmde0_2dc_gamma ( xl(L)
c    ^ rrho(L,k-2),  rrho(L,k),  rrho(L,k+2),
c    ^   pp(L,k-2),    pp(L,k),    pp(L,k+2),
c    ^  uux(L,k-2),   uux(L,k),   uux(L,k+2),
```

```
c     ^  uuy(L,k-2), uuy(L,k-1),      uuy(L,k), uuy(L,k+1), uuy(L,k+2),
c     ^    rhonu(L),     pnu(L),      uxnu(L),    uynu(L),
c     ^       gamma,         dt,       smlrho,     smallp,     smallu,
c     ^      smalle,    xcourmx,            nx, nbdy)

c      call  do_ppmlr0_2dc_gamma ( xxl(ip),
c     ^ dddslab(ip,islabbb,1),  dddslab(ip,islab,1),   dddslab(ip,islabtt,1),
c     ^ dddslab(ip,islabbb,2),  dddslab(ip,islab,2),   dddslab(ip,islabtt,2),
c     ^ dddslab(ip,islabbb,3),   dddslab(ip,islab,3),    dddslab(ip,islabtt,3),
c     ^  dddslab(ip,islabbb,4),   dddslab(ip,islabb,4),    dddslab(ip,islab,4),
c     ^            dddslab(ip,islabt,4),   dddslab(ip,islabtt,4),
c     ^     rhonu(ip),     pnu(ip),      u1nu(ip),    u2nu(ip),
c     ^   gamma,          dt,        smlrho,      smallp,     smallu,
c     ^      smalle,    ycourmx,            ny, nbdy)

c      print *,' ix  , ycourmx=', ix  , ycourmx

c
       courmx = max ( courmx, ycourmx )
c
          do iy = 1, ny
          ddd(   1,ix,iy) = rhonu(iy)
          ddd(   2,ix,iy) =  pnu(iy)
          ddd(kvy,ix,iy) = u1nu(iy)
          ddd(kvx  ,ix,iy) = u2nu(iy)
          enddo
c

       enddo
c      (do 400))

c
c
c
c )
c
c
        iy0x1 = 0
c      Set fake zones according to boundary conditions.
           call bdrys ( xl,  yl, ddd, nx, ny, nbdy, iy0x1)
c
c
c(
c
c======================================================================72
c                        -----  y-pass  -----
c
c
c   the xvelocity is element #3 in DDD
c   the yvelocity is element #4 in DDD
       kvx = 3
       kvy = 4

       do i = 1-nbdy, ny+nbdy+1
       xxl(i) = yl(i)
       enddo
c
c   initialize first 4 rows in slab
c
c      ((do 100)
       do islab = 1, 4
       ix   = islab-2
             do iy = 1-nbdy, ny+nbdy
               dddslab(iy, islab,1) =  ddd(   1, ix, iy)
               dddslab(iy, islab,2) =  ddd(   2, ix, iy)
               dddslab(iy, islab,3) =  ddd(kvy, ix, iy)
               dddslab(iy, islab,4) =  ddd(kvx  , ix, iy)
             enddo
       enddo
c      (do 100))

c
c      ((do 400)
       do ix   = 1, nx
c         find pointers to proper rows in the slab to where
c          the row 2. zones below are stored (slabbb) 1.
c          row below (slabb), etc
          islabbb = 1+mod(ix  -1,5)
          islabb  = 1+mod(ix  ,  5)
          islab   = 1+mod(ix  +1,5)
          islabt  = 1+mod(ix  +2,5)
          islabtt = 1+mod(ix  +3,5)
c
c   At this point, we are ALWAYS able to fill position slabtt from ddd
          lx   = ix  +2
             do iy = 1-nbdy, ny+nbdy
             ly = iy
               dddslab(iy, islabtt,1) =  ddd(   1, lx, ly)
               dddslab(iy, islabtt,2) =  ddd(   2, lx, ly)
               dddslab(iy, islabtt,3) =  ddd(kvy, lx, ly)
               dddslab(iy, islabtt,4) =  ddd(kvx  , lx, ly)
             enddo

          ip = 1-nbdy
              d2inv = 4. / (x  l(ix  +1)+x  l(ix  +2)-x  l(ix  -1)-x  l(ix  ))
          ycourmx = 0.
       call  do_ppmde0_2dc_gamma( xxl(ip),
      ^ dddslab(ip,islabbb,1),  dddslab(ip,islab,1),   dddslab(ip,islabtt,1),
```

```
     ^ dddslab(ip,islabbb,2),  dddslab(ip,islab,2),   dddslab(ip,islabtt,2),
     ^ dddslab(ip,islabbb,3),   dddslab(ip,islab,3),    dddslab(ip,islabtt,3),
     ^  dddslab(ip,islabbb,4),   dddslab(ip,islabb,4),    dddslab(ip,islab,4),
     ^              dddslab(ip,islabt,4),   dddslab(ip,islabtt,4),
     ^     rhonu(ip),     pnu(ip),      u1nu(ip),    u2nu(ip),
     ^   gamma,          dt,       smlrho,     smallp,     smallu,
     ^      smalle,    ycourmx,            ny, nbdy)
c     call do_ppmde0_2dc_gamma ( xl(L)
c    ^ rrho(L,k-2),  rrho(L,k),  rrho(L,k+2),
c    ^   pp(L,k-2),    pp(L,k),    pp(L,k+2),
c    ^  uux(L,k-2),   uux(L,k),   uux(L,k+2),
c    ^  uuy(L,k-2), uuy(L,k-1),     uuy(L,k), uuy(L,k+1), uuy(L,k+2),
c    ^    rhonu(L),     pnu(L),      uxnu(L),    uynu(L),
c    ^       gamma,          dt,       smlrho,     smallp,     smallu,
c    ^      smalle,    xcourmx,            nx, nbdy)

c     call  do_ppmlr0_2dc_gamma ( xxl(ip),
c    ^ dddslab(ip,islabbb,1),  dddslab(ip,islab,1),   dddslab(ip,islabtt,1),
c    ^ dddslab(ip,islabbb,2),  dddslab(ip,islab,2),   dddslab(ip,islabtt,2),
c    ^ dddslab(ip,islabbb,3),   dddslab(ip,islab,3),    dddslab(ip,islabtt,3),
c    ^  dddslab(ip,islabbb,4),   dddslab(ip,islabb,4),    dddslab(ip,islab,4),
c    ^              dddslab(ip,islabt,4),   dddslab(ip,islabtt,4),
c    ^     rhonu(ip),     pnu(ip),      u1nu(ip),    u2nu(ip),
c    ^   gamma,          dt,       smlrho,     smallp,     smallu,
c    ^      smalle,    ycourmx,            ny, nbdy)

c     print *,' ix  , ycourmx=', ix  , ycourmx

c
      courmx = max ( courmx, ycourmx )
c
         do iy = 1, ny
         ddd(   1,ix,iy) = rhonu(iy)
         ddd(   2,ix,iy) =  pnu(iy)
         ddd(kvy,ix,iy) = u1nu(iy)
         ddd(kvx  ,ix,iy) = u2nu(iy)
         enddo
c

      enddo
c     (do 400))

c
c
c
c )
c
c
      iy0x1 = 1
c     Set fake zones according to boundary conditions.
      call bdrys ( xl,  yl, ddd, nx, ny, nbdy, iy0x1)
c
c
c(
c
c=====================================================================72
c                        -----  x-pass  -----
c
c
c   the xvelocity is element #3 in DDD
c   the yvelocity is element #4 in DDD
      kvx = 3
      kvy = 4

      do i = 1-nbdy, nx+nbdy+1
      xxl(i) = xl(i)
      enddo
c
c   initialize first 4 rows in slab
c
c     ((do 100)
      do islab = 1, 4
      iy   = islab-2
            do ix = 1-nbdy, nx+nbdy
              dddslab(ix, islab,1) =  ddd(   1, ix, iy)
              dddslab(ix, islab,2) =  ddd(   2, ix, iy)
              dddslab(ix, islab,3) =  ddd(kvx, ix, iy)
              dddslab(ix, islab,4) =  ddd(kvy  , ix, iy)
            enddo
      enddo
c     (do 100))

c
c     ((do 400)
      do iy   = 1, ny
c        find pointers to proper rows in the slab to where
c         the row 2. zones below are stored (slabbb) 1.
c         row below (slabb), etc
         islabbb = 1+mod(iy  -1,5)
         islabb  = 1+mod(iy  ,  5)
         islab   = 1+mod(iy  +1,5)
         islabt  = 1+mod(iy  +2,5)
         islabtt = 1+mod(iy  +3,5)
c
c   At this point, we are ALWAYS able to fill position slabtt from ddd
          ly   = iy  +2
            do ix = 1-nbdy, nx+nbdy
             lx = ix
```

```
            dddslab(ix, islabtt,1) =  ddd(   1, lx, ly)
            dddslab(ix, islabtt,2) =  ddd(   2, lx, ly)
            dddslab(ix, islabtt,3) =  ddd(kvx, lx, ly)
            dddslab(ix, islabtt,4) =  ddd(kvy  , lx, ly)
          enddo

         ip = 1-nbdy
             d2inv = 4. / (y  l(iy  +1)+y  l(iy  +2)-y  l(iy  -1)-y  l(iy  ))
         xcourmx = 0.
      call  do_ppmde0_2dc_gamma( xxl(ip),
     ^ dddslab(ip,islabbb,1),  dddslab(ip,islab,1),   dddslab(ip,islabtt,1),
     ^ dddslab(ip,islabbb,2),  dddslab(ip,islab,2),   dddslab(ip,islabtt,2),
     ^ dddslab(ip,islabbb,3),   dddslab(ip,islab,3),    dddslab(ip,islabtt,3),
     ^  dddslab(ip,islabbb,4),   dddslab(ip,islabb,4),    dddslab(ip,islab,4),
     ^             dddslab(ip,islabt,4),   dddslab(ip,islabtt,4),
     ^     rhonu(ip),     pnu(ip),       u1nu(ip),     u2nu(ip),
     ^   gamma,          dt,        smlrho,      smallp,      smallu,
     ^      smalle,    xcourmx,            nx, nbdy)
c
      courmx = max ( courmx, xcourmx )
c
         do ix = 1, nx
         ddd(   1,ix,iy) = rhonu(ix)
         ddd(   2,ix,iy) =  pnu(ix)
         ddd(kvx,ix,iy) = u1nu(ix)
         ddd(kvy  ,ix,iy) = u2nu(ix)
         enddo
c

      enddo
c     (do 400))

c
c
c
c )
c
c
c
c
      return
      end
```

The reader will have undoubtedly noticed that the boundary zone requirements for the ppm templates are asymmetric. That is, a NBDY zone are needed at each end of the arrays in the logical direction of the pass, but only two zones are required in the directions transverse. As it would be wasteful to set values in unrequired zones, a flag is passed to the boundary routine which is zero for a y-pass and 1 for an x-pass.

```
c
      iy0x1 = 1
c     Set fake zones according to boundary conditions.
      call bdrys ( xl,  yl, ddd, nx, ny, nbdy, iy0x1)
```

```
      subroutine bdrys ( xl,  yl, ddd, nx, ny, nbdy, iy0x1)
c
      dimension     xl(-nbdy+1:nx+nbdy+1),  yl(-nbdy+1:ny+nbdy+1)
      dimension  ddd(4,1-nbdy:nx+nbdy,1-nbdy:ny+nbdy)
c
c
c
c     nbdy   = the number of fake zones needed to implement boundary
c              conditions so that in the hydrodynamical calculation
c              all zones can be treated as if they were interior zones.
c     nx     = the number of real zones in the x-dimension for this
c              tile of the computational domain.
c     ny     = the number of real zones in the y-dimension for this
c              tile of the computational domain.
c     iy0x1  = a flag which is 0 in the y-pass and 1 in the x-pass.
c
c      xl    = the x-grid for this tile.
c      yl    = the y-grid for this tile.
c
c     rrho   = the zone-averaged densities for this tile.
c     pp     = the zone-averaged pressures for this tile.
c     uux    = the zone-averaged x-velocities for this tile.
c     uuy    = the zone-averaged y-velocities for this tile.
c
c     ddd    = the 3-D array into which the 4 principal 2-D arrays of
c              basic physical variables are interleaved in order to
c              optimize memory bandwidth on microprocessor systems.
c     ddd(1,i,j) = rrho(i,j)
c     ddd(2,i,j) = pp(i,j)
c     ddd(3,i,j) = uux(i,j)
c     ddd(4,i,j) = uuy(i,j)
c
c
c
c
c     Apply boundary conditions, Periodic only.
```

```
c
      iy1x0 = 1 - iy0x1
      nxp1 = nx + 1
      ntoset = iy0x1 * nbdy  +  iy1x0 * 2
c
c     Periodic boundary conditions:
c
      gridsz =  xl(nxp1) -  xl(1)
      do 190   i = 1-nbdy,0
       xl(i) =  xl(i+nx) - gridsz
190   continue
c
      do 200   i = -ntoset+1,0
      do 200   k = 1,ny
      ddd(1,i,k) = ddd(1,i+nx,k)
      ddd(2,i,k) = ddd(2,i+nx,k)
      ddd(3,i,k) = ddd(3,i+nx,k)
      ddd(4,i,k) = ddd(4,i+nx,k)
200   continue
c
      do 290   i = nxp1+1,nxp1+nbdy
       xl(i) =  xl(i-nx) + gridsz
290   continue
c
      do 300   i = nxp1,nx+ntoset
      do 300   k = 1,ny
      ddd(1,i,k) = ddd(1,i-nx,k)
      ddd(2,i,k) = ddd(2,i-nx,k)
      ddd(3,i,k) = ddd(3,i-nx,k)
      ddd(4,i,k) = ddd(4,i-nx,k)
300   continue
c
900   continue
c
c
      nyp1 = ny + 1
      ntoset = iy1x0 * nbdy  +  iy0x1 * 2
c
c     Periodic boundary conditions:
c
      gridsz =  yl(nyp1) -  yl(1)
      do 1190   i = 1-nbdy,0
       yl(i) =  yl(i+ny) - gridsz
1190  continue
c
      do 1200   i = -ntoset+1,0
      do 1200   k = -nbdy+1,nx+nbdy
      ddd(1,k,i) = ddd(1,k,i+ny)
      ddd(2,k,i) = ddd(2,k,i+ny)
      ddd(3,k,i) = ddd(3,k,i+ny)
      ddd(4,k,i) = ddd(4,k,i+ny)
1200  continue
c
      do 1290   i = nyp1+1,nyp1+nbdy
       yl(i) =  yl(i-ny) + gridsz
1290  continue
c
      do 1300   i = nyp1,ny+ntoset
      do 1300   k = -nbdy+1,nx+nbdy
      ddd(1,k,i) = ddd(1,k,i-ny)
      ddd(2,k,i) = ddd(2,k,i-ny)
      ddd(3,k,i) = ddd(3,k,i-ny)
      ddd(4,k,i) = ddd(4,k,i-ny)
1300  continue
c
c
c
      return
      end
```

This dependence on data transverse to the pass also requires some care during the sequential calls to the ppmlib template which updates a row of data. If one merely extracted the required five rows from `ddd` and then stored in `ddd` the new zone averaged in their proper location within that array, the information sent to the template would quickly become a mixture of data at the old and new timesteps as the loop progressed up the `ddd` array, consuming old strips of data laying ahead of it and depositing new strips of data immediately behind. This is avoided here by using a five strip temporary array, `dddslag` and constructing the pointers `slabbb`, `slabb`, `slab`, `slabt`, and `slabtt`, which point to the location of the data for the row two rows below, one row below the row being updated, the row being updated, one row above, and two rows above, resp. Representative results at four different times are shown in figure 18.

Do_PPM_DE0_3DC_GAMMA

This template extends the previous example to three dimensions.

```
      subroutine do_ppm_de0_3dc_gamma (xl, rhozbb, rho, rhoztt, rhozmm, rhozpp,
     ^      pzbb, p, pztt, pzmm, pzpp,  u1zbb, u1, u1ztt, u1zmm, u1zpp, u2zbb,
     ^      u2zb, u2, u2zt, u2ztt, u2zmm, u2zpp, u3zmm, u3zm, u3,
     ^      u3zp, u3zpp, u3zbb, u3ztt, rhonu,pnu, u1nu, u2nu,
     ^      u3nu, eosgam, dt, smlrho, smallp, smallu, smalle, courmx,
     ^      n_real_zones_in, nbdy_set)
c
```

Now in addition to the information above and below the row being updated as in the previous example, information is needed two planes in front of the row being updated (suffix `zpp`) and two planes behind the row being updated (suffix `zmm`) for density, pressure, longitudinal velocity and both transverse velocities. Additionally, zone averages one plane in front (suffix `zp`) and one plane behind (suffix `zm`) for the second transverse velocity are required. It is assumed that the directions of the velocities form a righthand coordinate system, with the direction of `u1` (the longitudinal velocity) pointing in the direction of increasing x, the direction of the first transverse velocity (`u2`) pointing in the direction from ‘zbb’’ to ‘‘ztt’’, and the direction of the second transverse velocity (`u3`) pointing in the direction from ‘‘zmm’’ to ‘‘zpp.’’

The calculation proceeds as with the previous example, with the notable exception that the zone centered Eulerian diffusion velocities are now found from **PPM98_EUL_DIFUSE0_GAMMA_3D**.

```
      call ppm98_eul_difuse0_gamma_3d (
     ^               u1zbb(K),   u1(k-2),  u1(k-1),   u1(K), u1(k+1), u1(k+2),
     ^     u1ztt(k),  u1zmm(k),  u1zpp(k), u2zbb(K), u2zb(k),   u2(K-2), u2(k),
     ^     u2(k+2),  u2zt(k), u2ztt(K),  u2zmm(k), u2zpp(k), u3zbb(K), u3(k-2),
     ^        u3(K), u3(k+2), u3ztt(k),  u3zmm(K),  u3zm(k), u3zp(k), u3zpp(k),
     ^     pzbb(K),    p(k-2), p(K), p(k+2), pztt(k), pzmm(K), pzpp(k),
     ^    rhozbb(K), rho(k-2), rho(K), rho(k+2), rhoztt(k),rhozmm(K),rhozpp(k),
     ^       unsmooth(K), difuse_vel(K),  shockd(K), eosgam, smallu,
     ^              nzones)
```

This routine uses information in three dimensions in order to be able to determine the position and orientation with respect to the grid of any shocks which may be present. Using these zone centered velocities, the hydrodynamic quantities are updated by **PPM98_DIFUSEFLUX_HYDRO_CART**, with the remaining calculation proceeding template **fBPPM98_EUL_DIFUSE0_GAMMA_2D**. `It is used in the sample driver Driverdirect3dg0c.`

```
      program Driverdirect2dg0c
c

c
      parameter (NXZONES = 128)
      parameter (NYZONES = 128)
      parameter (NZZONES = 128)
      parameter (MAXSTEPS = 500)
      parameter (EOSGAM = 1.4)
      parameter (DT = 0.00200)
      parameter (XMAX = 1.0)
      parameter (YMAX = 1.0)
```

```
      parameter (ZMAX = 1.0)
      parameter (N2DUMP = 4)
      parameter (T2DUMP = 99999.)
c     Note that because of the fixed timestep, N, XMAX, and DT are not
c               independent.
c
c      NXZONES = number of real computational zones in the x-direction, set
c                by an m4 definition at beginning of code.
c      NYZONES = number of real computational zones in the y-direction, set
c                by an m4 definition at beginning of code.
c      NZZONES = number of real computational zones in the z-direction, set
c                by an m4 definition at beginning of code.
c        NXZONES and NYZONES should be changed by changing the values of
c         128, 128, and 128 at beginning.
c      MAXSTEPS = maximum number of timpsteps to perform.
c      EOSGAM = the equation of state gamma
c      DT = the fixed timestep
c      XMAX = the righthand edge x-value of the grid.
c                                    The x-grid runs from (0,XMAX)
c      YMAX = the righthand edge y-value of the grid.
c                                    The y-grid runs from (0,YMAX)
c      ZMAX = the righthand edge y-value of the grid.
c                                    The z-grid runs from (0,ZMAX)
c      N2DUMP is the frequency of image dumps in iterations
c      T2DUMP is the frequency of image dumps in program time.
c
c     Do not change this 1.:
      parameter (NBDY = 5)
c
c


c======================================================================
c
c     The constant  128  gives the maximum length of a row in the
c     x-direction which will be encountered.
c     The constant  128  gives the maximum length of a column in the
c     y-direction which will be encountered.
c
c
c======================================================================
c
c

      dimension  xl( 1-NBDY:NXZONES+NBDY+1)
      dimension  yl( 1-NBDY:NYZONES+NBDY+1)
      dimension  zl( 1-NBDY:NYZONES+NBDY+1)
      dimension ddd( 1:5,
     ^                   1-NBDY:NXZONES+NBDY,
     ^                       1-NBDY:NYZONES+NBDY,
     ^                           1-NBDY:NZZONES+NBDY )
c
c
c
c     ddd    = the 4-D array into which the 5 principal 3-D arrays of
c              basic physical variables are interleaved in order to
c              optimize memory bandwidth on microprocessor systems.
c
c      xl    = the locations of the lefthand zone interfaces
c      yl    = the locations of the bottom zone interfaces
c      zl    = the locations of the front zone interfaces
c     ddd    = the 4-D array into which the 5 principal 3-D arrays of
c              basic physical variables are interleaved in order to
c              optimize memory bandwidth on microprocessor systems.
c     ddd(1,i,j) = rrho(i,j)
c     ddd(2,i,j) = pp(i,j)
c     ddd(3,i,j) = uux(i,j)
c     ddd(4,i,j) = uuy(i,j)
c     ddd(5,i,j) = uuz(i,j)
c     rrho   = the zone-averaged densities
c     pp     = the zone-averaged pressures
c     uux    = the zone-averaged x-velocities
c     uuy    = the zone-averaged y-velocities
c     uuz    = the zone-averaged z-velocities
c
c
c     smlrho = a trivial value of density, used to protect divides.
c     smalle = a trivial value of energy, used to protect divides.
c     smallp = a trivial value of pressure, used to protect divides.
c     smallu = a trivial value of velocity, used to protect divides.
c
      smlrho = 1.0e-06
      smalle = 1.0e-06
      smallp = 1.0e-06
      smallu = 1.0e-06
c
c

c
c     initialize the grid:
      xlimit = XMAX
      ylimit = YMAX
      zlimit = ZMAX

      nx = nxzones
      ny = nyzones
      nz = nzzones
c
```

```
      call setgrid(  xl,  yl, zl, xlimit, ylimit, zlimit,
     ^                         nx, ny, nz, nbdy )
c
c
c     initialize the hydrodynamic quantities:
c
      call set3dshock( xl, yl, zl, ddd, 5, nx, ny, nz, nbdy)
c
c
c
      nstop = MAXSTEPS
      tstop = float (MAXSTEPS) * DT
      ndump = N2DUMP
      tdump = T2DUMP

c          nstop is the maximum number of iterations to take for the entire
c             calculation
c          tstop is the maximum program time for the entire calculation, here
c             assuming a constant timestep DT
c          ndump is the frequency of image dumps in iterations
c          tdump is the frequency of image dumps in program time.

c
c   CONTROL performs the entire hydrodynamic calculation.
c
      dtime = DT
      gamma = EOSGAM

      call cntrol ( xl,  yl, zl, ddd, gamma, dtime, smlrho, smalle, smallp,
     ^             smallu, numdmp, ndump, tdump, nstop, tstop, nx, ny,
     ^               nz, nbdy)


      stop
      end
```

Again, the main program, is deliberately simple, performing a specified number of timesteps of a two-dimensional problem, with a fixed timestep and problem size. There are four arrays which are allocated in the main program. Three define the computational grid by specifying the locations of the zone interfaces, `xl`, `yl`, and `zl`. As usual remaining array, `ddd`, contains the zone averages of all the computed quantities which, for this problem, are density, pressure, x-, y-, and z-velocity, stored in interleaved order.

```
      dimension  xl( 1-NBDY:NXZONES+NBDY+1)
      dimension  yl( 1-NBDY:NYZONES+NBDY+1)
      dimension  zl( 1-NBDY:NYZONES+NBDY+1)
      dimension ddd( 1:5,
     ^                   1-NBDY:NXZONES+NBDY,
     ^                       1-NBDY:NYZONES+NBDY,
     ^                           1-NBDY:NZZONES+NBDY )
```

The number of actual computational zones in the x-, y-, and z-directions are given by the parameters `NXZONES`, `NYZONES`, and `NZZONES`, resp. An additional `NBDY` zones are added to the left and right of each dimension to allow for the imposition of boundary conditions.

As with the two-dimensional example, the arrays are initialized in two steps. First the computational grid is set by a call to a subroutine `setgrid`, creating a uniform computational grid running from 0.0 to 1.0 in each dimension.

```
      call setgrid(  xl,  yl, zl, xlimit, ylimit, zlimit, nx, ny, nz, nbdy )
c
c     initialize the hydrodynamic quantities:
c
      call set3dshock( xl, yl, zl, ddd, 5, nx, ny, nz, nbdy)
c
c
      nstop = MAXSTEPS
      tstop = float (MAXSTEPS) * DT
      ndump = N2DUMP
      tdump = T2DUMP

c          nstop is the maximum number of iterations to take for the entire
c             calculation
c          tstop is the maximum program time for the entire calculation, here
c             assuming a constant timestep DT
c          ndump is the frequency of image dumps in iterations
c          tdump is the frequency of image dumps in program time.
```

```
c
c   CONTROL performs the entire hydrodynamic calculation.
c
      dtime = DT
      gamma = EOSGAM

      call cntrol ( xl,  yl, zl, ddd, gamma, dtime, smlrho, smalle, smallp,
     ^              smallu, numdmp, ndump, tdump, nstop, tstop, nx, ny,
     ^                nz, nbdy)


      stop
      end
```

The zone averages are initialized by `set3dshock`. The initial conditions are those of a "three-dimensional shocktube", which is a cubical computational domain divided into eight subdomains, with all eight meeting at the point (nxdisc, nydisc, nzdisc) near the center. Each subdomain consists of a constant hydrodynamic state. This is illustrated in figure 19. After problem initialization, the program flow proceeds to `cntrol` which will perform the rest of the calculation.

```
      subroutine set3dshock( xl, yl, zl, ddd, nvar, nx, ny, nz, nbdy)
c  ***********************************************************************
c  *                        THIS CODE IS D36PPM3D                        *
c  * THIS CODE WAS WRITTEN BY:  B. Kevin Edgar  ALL RIGHTS RESERVED      *
c  ***********************************************************************
      dimension xl(1-nbdy:nx+nbdy+1)
      dimension yl(1-nbdy:ny+nbdy+1)
      dimension zl(1-nbdy:nz+nbdy+1)
      dimension ddd(1:nvar,1-nbdy:nx+nbdy,
     ^                     1-nbdy:ny+nbdy,
     ^                     1-nbdy:nz+nbdy)
c

      xdisc = 0.49
      ydisc = 0.49
      zdisc = 0.49

      xlength = xl(nx+1) - xl(1)
      ylength = yl(ny+1) - yl(1)
      zlength = zl(nz+1) - zl(1)
      xdisc = xl(1) + xdisc * xlength
      ydisc = yl(1) + ydisc * ylength
      zdisc = zl(1) + zdisc * zlength
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c  This routine sets the pressure, density, and velocity fields for    c
c  a BKE 3d shocktube.                                                 c
c--------------------------------------------------------------------c
c
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccc

      rhoo8 = 2.5
        po8 = 2.5
       vxo8 = 0.0
       vyo8 = 0.0
       vzo8 = 0.0

      rhoo7 = 1.25
        po7 = 1.25
       vxo7 = 0.0
       vyo7 = 0.0
       vzo7 = 0.0

      rhoo6 = 0.125
        po6 = 0.125
       vxo6 = 0.0
       vyo6 = 0.0
       vzo6 = 0.0

      rhoo5 = 0.635
        po5 = 0.635
       vxo5 = 0.0
       vyo5 = 0.0
       vzo5 = 0.0

c
      rhoo1 = 0.375
        po1 = 0.375
       vxo1 = 0.0
       vyo1 = 0.0
       vzo1 = 0.0

      rhoo2 = 0.075
        po2 = 0.075
       vxo2 = 0.0
```

```
        vyo2 = 0.0
        vzo2 = 0.0

      rhoo3 = 0.75
        po3 = 0.75
       vxo3 = 0.0
       vyo3 = 0.0
       vzo3 = 0.0

      rhoo4 = 1.5
        po4 = 1.5
       vxo4 = 0.0
       vyo4 = 0.0
       vzo4 = 0.0

c
c               y
c               ^
c               |
c               *******************
c              *   6      *  5   **
c             *          *      * *
c            *******************  *
c           *          *      **  *
c          *          *      * *  *
c         *******************  * **
c         *          *      *  ** *
c         *    2     *   1  *  *  *
c         *          *      * **8 *
c         *          *      ** *  *-->x
c         *******************  * *
c         *          *      *  **
c         *    3     *  4   *  *
c         *          *      * *
c         *          *      **
c         *******************
c        /
c       z
c

       do 611 k=1, nz

       if ( zl(k) .ge. zdisc ) then
      rrhoq1 = rhoo1
      rrhoq2 = rhoo2
      rrhoq3 = rhoo3
      rrhoq4 = rhoo4
      ppq1 = po1
      ppq2 = po2
      ppq3 = po3
      ppq4 = po4
      vvxq1 = vxo1
      vvxq2 = vxo2
      vvxq3 = vxo3
      vvxq4 = vxo4
      vvyq1 = vyo1
      vvyq2 = vyo2
      vvyq3 = vyo3
      vvyq4 = vyo4
      vvzq1 = vzo1
      vvzq2 = vzo2
      vvzq3 = vzo3
      vvzq4 = vzo4
       else
      rrhoq1 = rhoo5
      rrhoq2 = rhoo6
      rrhoq3 = rhoo7
      rrhoq4 = rhoo8
      ppq1 = po5
      ppq2 = po6
      ppq3 = po7
      ppq4 = po8
      vvxq1 = vxo5
      vvxq2 = vxo6
      vvxq3 = vxo7
      vvxq4 = vxo8
      vvyq1 = vyo5
      vvyq2 = vyo6
      vvyq3 = vyo7
      vvyq4 = vyo8
      vvzq1 = vzo5
      vvzq2 = vzo6
      vvzq3 = vzo7
      vvzq4 = vzo8
       endif
       do 611 j=1, ny
       if ( yl(j) .ge. ydisc ) then
      rhol = rrhoq2
      pl = ppq2
      vxl = vvxq2
      vyl = vvyq2
      vzl = vvzq2
      rhor = rrhoq1
      pr = ppq1
      vxr = vvxq1
      vyr = vvyq1
      vzr = vvzq1
```

```
         else
        rhol = rrhoq3
        pl = ppq3
        vxl = vvxq3
        vyl = vvyq3
        vzl = vvzq3
        rhor = rrhoq4
        pr = ppq4
        vxr = vvxq4
        vyr = vvyq4
        vzr = vvzq4
         endif
         do 611 i=1, nx
         if ( xl(i) .le. xdisc ) then
             ddd(1,i,j,k) = rhor
             ddd(2,i,j,k) = pr
             ddd(3,i,j,k) = vxr
             ddd(4,i,j,k) = vyr
             ddd(5,i,j,k) = vzr
         else
             ddd(1,i,j,k) = rhol
             ddd(2,i,j,k) = pl
             ddd(3,i,j,k) = vxl
             ddd(4,i,j,k) = vyl
             ddd(5,i,j,k) = vzl
         endif

611      continue

        return
        end
```

The subroutine, `cntrol` is an iterative loop consisting of two steps. First, the hydrodynamical calculation is advanced two time steps by a call to `runhyd3`. After `runhyd3` has updated the zone averages, an image file is created by a call to `dump3d`, when appropriate. This latter routine constructs a nx by ny by nz brick of bytes from the density zone averages, scaled from zero to one. This brick may be view by the **bob** “brick o bytes” viewer. Other modes of data output may be found in volume II, “I/O and Data Analysis.”

```
      subroutine cntrol ( xl,  yl, zl, ddd, gamma, dtime, smlrho, smalle,
     ^       smallp, smallu, numdmp, ndump, tdump, nstop, tstop, nx, ny, nz, nbdy)
c
      dimension  xl( 1-NBDY:NX+NBDY+1)
      dimension  yl( 1-NBDY:NY+NBDY+1)
      dimension  zl( 1-NBDY:NZ+NBDY+1)
      dimension ddd( 1:5, 1-NBDY:NX+NBDY, 1-NBDY:NY+NBDY, 1-NBDY:NZ+NBDY )
c
c
c     This routine runs the entire hydro calculation.
c
c
c
c     ddd    = the 4-D array into which the 5 principal 3-D arrays of
c              basic physical variables are interleaved in order to
c              optimize memory bandwidth on microprocessor systems.
c     nx     = the number of real zones in the x-dimension for this
c              tile of the computational domain.
c     ny     = the number of real zones in the y-dimension for this
c              tile of the computational domain.
c     nz     = the number of real zones in the z-dimension for this
c              tile of the computational domain.
c     nbdy   = the number of fake zones needed to implement boundary
c              conditions so that in the hydrodynamical calculation
c              all zones can be treated as if they were interior zones.
c
c
c     dtime  = the time step.
c     courmx = the maximum value of the Courant number for this tile
c              of the computational domain.
c     gamma  = the value of the ratio of specific heats, as in the
c              gamma-law equation of state:  p = (gamma-1)*rho*ei
c
c     smlrho = a trivial value of density, used to protect divides.
c     smalle = a trivial value of energy, used to protect divides.
c     smallp = a trivial value of pressure, used to protect divides.
c     smallu = a trivial value of velocity, used to protect divides.
c
c      xl    = the locations of the lefthand zone interfaces
c      yl    = the locations of the bottom zone interfaces
c      zl    = the locations of the front zone interfaces
c     ddd    = the 3-D array into which the 5 principal 2-D arrays of
c              basic physical variables are interleaved in order to
c              optimize memory bandwidth on microprocessor systems.
c     ddd(1,i,j) = rrho(i,j)
```

```
c     ddd(2,i,j) = pp(i,j)
c     ddd(3,i,j) = uux(i,j)
c     ddd(4,i,j) = uuy(i,j)
c     ddd(5,i,j) = uuz(i,j)
c     rrho   = the zone-averaged densities for this tile.
c     pp     = the zone-averaged pressures for this tile.
c     uux    = the zone-averaged x-velocities for this tile.
c     uuy    = the zone-averaged y-velocities for this tile.
c     uuz    = the zone-averaged z-velocities for this tile.
c
c
c    Create an image of the initial state. The initial state will be
c     sequence number 0.

       numdmp = 0
      call dump3d (ddd, numdmp, nx, ny, nz, nbdy)

c           numdmp = the sequential number of the image 0->whatever, numdmp
c           will be automatically incremented after write the image.
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c                                                                    c
c      The runhyd advances the calculation by 2. timesteps per    c
c                             iteration                              c
c                                                                    c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c   initialize counters
c
       ncycle = 0
       time = 0.0
       ntodmp = ndump
       ttodump = tdump
c
1000  continue
c
c   clear the old maximum courant number in preparation of the next pair of
c       timesteps:
       courmx = 0.
c
c   RUNHYD3 performs a symmetrized pair of passes x-y-z-z-y-x which
c          equals 2. complete timesteps.
c
       call runhyd3 ( xl,  yl, zl, ddd, dtime, courmx, gamma,
     1              smlrho, smalle, smallp, smallu,  numdmp,
     2              nx, ny, nz, nbdy)
c
c
c
c   update counters
c
       ncycle = ncycle + 2
       ntodmp = ntodmp - 2
       time = time + dtime + dtime
c
       write (6,1710)   ncycle, time, courmx, dtime
c
c   Check to see if time to write am image file.  An image is
c   written every NDUMP steps or at intervals of DTDUMP in
c   problem time.
c
       if (ntodmp)   1600,1600,1500
1500  if (time - tdump)   1700,1600,1600
1600  ntodmp = ndump
       ttodump = ttodump + tdump

       call dump3d (ddd, numdmp, nx, ny, nz, nbdy)

c     numdmp = the sequential number of the image 0->whatever, numdmp
c           will be automatically incremented after write the image.

      call flush (6)
c
1700  continue
1710  format (1h ,3hn =,i5,4x,3ht =,
     1        1pe12.5,4x,13hcourant no. =,e12.5,4x,4hdt =,e12.5)
c
c   Check to see if time to stop.  Progam execution ends when either
c    the maximum number of timesteps, NSTOP,  has been computed or
c    when the maximum problem time, TSTOP, has been reached.
c
      if (ncycle .ge. nstop) stop
      if (time .ge. tstop) stop
      goto 1000
      end
```

The routine `runhyd` performs the actual work of the hydrodynamical calculation in a sequence of six passes or iterative loops, first an x-pass, a y-pass, two z-passes, another y-pass, and a final x-pass to complete a symmetrized pair of timesteps. Each pass consists of two parts: application of boundary conditions followed by the actual hydrodynamic calculation. These passes are similar in nature, so the discussion will be restricted to the first x-pass.

```
      subroutine runhyd3 ( xl,  yl, zl, ddd, dt,
     1                     courmx, gamma, smlrho, smalle, smallp, smallu,
     2                     numdmp, nx, ny, nz, nbdy)
c
      dimension  xl( 1-NBDY:NX+NBDY+1)
      dimension  yl( 1-NBDY:NY+NBDY+1)
      dimension  zl( 1-NBDY:NZ+NBDY+1)

      dimension    ddd(5,1-nbdy:nx+nbdy,1-nbdy:ny+nbdy,1-nbdy:nz+nbdy)
c   we are counting on the F77 compiler to automatically allocate/deallocate
c
      dimension
     1      rhonu(1-5:5+128),
     ^        pnu(1-5:5+128),
     ^        u1nu(1-5:5+128),
     ^        u2nu(1-5:5+128),
     ^        u3nu(1-5:5+128),
     ^        crmx(1-5:5+128)
c
      dimension
     1      xxl(1-nbdy:nbdy+nx+ny+1),
     ^        dddslab(1-5:5+128,1-5:5+128,5,5)
c
c
c
c     nbdy   = the number of fake zones needed to implement boundary
c              conditions so that in the hydrodynamical calculation
c              all zones can be treated as if they were interior zones.
c     nx     = the number of real zones in the x-dimension for this
c              tile of the computational domain.
c     ny     = the number of real zones in the y-dimension for this
c              tile of the computational domain.
c     nz     = the number of real zones in the z-dimension for this
c              tile of the computational domain.
c
c
c     dt     = the time step.
c     courmx = the maximum value of the Courant number for this tile
c              of the computational domain.
c     gamma  = the value of the ratio of specific heats, as in the
c              gamma-law equation of state:  p = (gamma-1)*rho*ei
c
c     smlrho = a trivial value of density, used to protect divides.
c     smalle = a trivial value of energy, used to protect divides.
c     smallp = a trivial value of pressure, used to protect divides.
c     smallu = a trivial value of velocity, used to protect divides.
c
c      xl    = the x-grid for this tile.
c      yl    = the y-grid for this tile.
c      zl    = the z-grid for this tile.
c     rrho   = the zone-averaged densities for this tile.
c     pp     = the zone-averaged pressures for this tile.
c     uux    = the zone-averaged x-velocities for this tile.
c     uuy    = the zone-averaged y-velocities for this tile.
c     uuz    = the zone-averaged y-velocities for this tile.
c     ddd    = the 3-D array into which the 5 principal 2-D arrays of
c              basic physical variables are interleaved in order to
c              optimize memory bandwidth on microprocessor systems.
c     ddd(1,i,j) = rrho(i,j)
c     ddd(2,i,j) = pp(i,j)
c     ddd(3,i,j) = uux(i,j)
c     ddd(4,i,j) = uuy(i,j)
c     ddd(5,i,j) = uuz(i,j)
c
c
c
c
10    continue
c
      courmx = 0.
      maxdim = max (nx, ny,nz)
c
c
c     NOW BEGIN LOOP OVER 2 TIME STEPS.
c     These 2 time steps are symmetrized;  they both use the same
c     time step value, and they perform the sweeps in
c     opposite orders.
c
c     Set fake zones according to boundary conditions.
      call bdrys ( xl,  yl, zl, ddd, 5, 5, 5, nx, ny, nz, nbdy )
c
c
c(
c
c=====================================================================72
c                         -----  x-pass  -----
c
c   This is an m4 macro definition which will construct a pass
c   in the x direction.  It is invoked by, for example,
c    P_A_S_S( x, y,z )  (without the "_")
c    in which case, all instances of "x" will be replaced with "x", and
```

```
c     all "y" by "y", and all "z " by "z".
c
c    Why has this been done? Believe it or not, as an attempt at simplification.
c      In order to update any given row, PPM  requires information from up to
c      2. rows above below, fore, and aft of  the row under intense scrutiny.
c
c         Consider first just above and below.
c
c      This information must be from the same computational time, however,
c      as the computation progresses, the large storage array DDD becomes
c      a mixture of old timestep information and new timestep information
c      as updated zone averages (rhonu, pnu, unu, etc) are stored in DDD.
c      Therefore, a small working scratch space 5. rows deep (a slab) will
c      be created. Consider an X-pass, then the slab would be dimensioned
c      at least (1-nbdy:nx+nbdy,,-1,ny+2,5):
c
c               ----------------------------------------
c       islab=5 |                                      >
c               ----------------------------------------
c       islab=4 |                                      >
c               ----------------------------------------
c       islab=3 |                                      >
c               ----------------------------------------
c       islab=2 |                                      >
c               ----------------------------------------
c       islab=1 |                                      >
c               ----------------------------------------
c
c    initially, islab=1 is taken from ddd(*,*,-1,*)
c    initially, islab=2 is taken from ddd(*,*,0,*)
c    initially, islab=3 is taken from ddd(*,*,1,*)
c    initially, islab=4 is taken from ddd(*,*,2,*)
c    initially, islab=5 is taken from ddd(*,*,3,*)
c
c    Pointers to the correct slab planes (islab1bb, islab1b, islab1, islab1t,
c    islab1tt)
c    (for  2 planes aft, 1 plane aft, the row, 1. row fore, 2 rows fore)
c    which would have the values (1,2,3,4,5).  After the call to do_ppm and the
c    updated values stored in DDD, the counters are cycled to (2,3,4,5,1)
c    and the relevent required data from DDD is written into the row pointed to
c    by the new value of islabtt.
c
c    The user should also note that the velocities switch roles.  That is,
c    during an X-pass, u1 is the x-velocity and u2, u3 are the y- and
c    z-velocities, resp. while during a Y-pass, u1 is the y-velocity and u2, u3
c    are the z- and x-velocities, resp.  One should maintain a right-hand
c    coordinate system.  I don't believe that this is required (now) but
c    who knows what future perils await thee.
c
c
c    This reduces both the scratch space requirement and the amount of
c    data copying required. Believe it or not.
c
c    the xvelocity is element #3 in DDD
c    the yvelocity is element #4 in DDD
c    the zvelocity is element #5 in DDD
       kvx = 3
       kvy = 4
       kvz = 5

       do i = 1-nbdy, nx+nbdy+1
       xxl(i) = xl(i)
       enddo
c
c    initialize the first  slab
c
c     islab = 5 is done below at plane islabtt
c
c      ((do 100)
       do islab = 1, 4
        iz  = islab-2
       do iy = 1-nbdy,ny+nbdy
             do ix = 1-nbdy, nx+nbdy
               dddslab(ix, iy,islab,1) =  ddd(   1, ix, iy,iz)
               dddslab(ix, iy,islab,2) =  ddd(   2, ix, iy,iz)
               dddslab(ix, iy,islab,3) =  ddd(kvx, ix, iy,iz)
               dddslab(ix, iy,islab,4) =  ddd(kvy, ix, iy,iz)
               dddslab(ix, iy,islab,5) =  ddd(kvz , ix, iy,iz)
             enddo
       enddo
       enddo
c      (do 100))

       islabbb = 0
       islabb  = 1
       islab   = 2
       islabt  = 3
       islabtt = 4

c
c      ((do 410)
       do iz  = 1, nz
c            (slabbb) 2. in front (slabtt), etc.
c         islabbb = 1+mod(iz -1,5)
c         islabb  = 1+mod(iz ,  5)
c         islab   = 1+mod(iz +1,5)
c         islabt  = 1+mod(iz +2,5)
c         islabtt = 1+mod(iz +3,5)
```

```
         islabbb = 1+mod(islabbb,5)
         islabb  = 1+mod(islabb, 5)
         islab   = 1+mod(islab,  5)
         islabt  = 1+mod(islabt, 5)
         islabtt = 1+mod(islabtt,5)

c   At this point, we are ALWAYS able to fill plane islabtt from ddd
           lz  = iz +2
            do ly = 1-nbdy, ny+nbdy
            do lx = 1-nbdy, nx+nbdy
              dddslab(lx, ly,islabtt,1) =  ddd(   1, lx, ly,lz)
              dddslab(lx, ly,islabtt,2) =  ddd(   2, lx, ly,lz)
              dddslab(lx, ly,islabtt,3) =  ddd(kvx, lx, ly,lz)
              dddslab(lx, ly,islabtt,4) =  ddd(kvy, lx, ly,lz)
              dddslab(lx, ly,islabtt,5) =  ddd(kvz , lx, ly,lz)

            enddo
            enddo
c     ((do 400)
      do iy = 1, ny
c

         ip = 1-nbdy
         xcourmx = 0.
             call do_ppm_de0_3dc_gamma( xl(ip),
     ^ dddslab(ip,iy-2,islab,1),
     ^           dddslab(ip,iy,islab,1),
     ^                 dddslab(ip,iy+2,islab,1),
     ^                     dddslab(ip,iy,islabbb,1),
     ^                         dddslab(ip,iy,islabtt,1),
     ^ dddslab(ip,iy-2,islab,2),
     ^           dddslab(ip,iy,islab,2),
     ^                 dddslab(ip,iy+2,islab,2),
     ^                     dddslab(ip,iy,islabbb,2),
     ^                         dddslab(ip,iy,islabtt,2),
     ^ dddslab(ip,iy-2,islab,3),
     ^           dddslab(ip,iy,islab,3),
     ^                 dddslab(ip,iy+2,islab,3),
     ^                     dddslab(ip,iy,islabbb,3),
     ^                         dddslab(ip,iy,islabtt,3),
     ^ dddslab(ip,iy-2,islab,4),
     ^ dddslab(ip,iy-1,islab,4),
     ^           dddslab(ip,iy,islab,4),
     ^                 dddslab(ip,iy+1,islab,4),
     ^                 dddslab(ip,iy+2,islab,4),
     ^                     dddslab(ip,iy,islabbb,4),
     ^                         dddslab(ip,iy,islabtt,4),
     ^ dddslab(ip,iy,islabbb,5),
     ^    dddslab(ip,iy,islabb,5),
     ^       dddslab(ip,iy,islab,5),
     ^          dddslab(ip,iy,islabt,5),
     ^             dddslab(ip,iy,islabtt,5),
     ^                dddslab(ip,iy-2,islab,5),
     ^                dddslab(ip,iy+2,islab,5),
     ^           rhonu(ip),  pnu(ip), u1nu(ip), u2nu(ip), u3nu(ip),
     ^           gamma, dt, smlro, smallp, smallu, smalle,
     ^           xcourmx, nx, nbdy )
c

      print *,' iy, iz , xcourmx=', iy, iz , xcourmx

c
      courmx = max ( courmx, xcourmx )
c
         do ix = 1, nx
         ddd(   1,ix,iy,iz) = rhonu(ix)
         ddd(   2,ix,iy,iz) =  pnu(ix)
         ddd(kvx,ix,iy,iz) = u1nu(ix)
         ddd(kvy,ix,iy,iz) = u2nu(ix)
         ddd(kvz ,ix,iy,iz) = u3nu(ix)
         enddo
c

      enddo
c     (do 400))

      enddo
c     (do 410))

c
c
c
c )
c
      print *,' pass x, y, z 1 courmx = ',courmx
66    format(1h ,i3,1p,4(1x,e10.3))

c     Set fake zones according to boundary conditions.
      call bdrys ( xl,  yl, zl, ddd, 5, 5, 5, nx, ny, nz, nbdy )
c
c
c(
c
c======================================================================72
c                      -----  y-pass  -----
c
c   This is an m4 macro definition which will construct a pass
c   in the y direction.  It is invoked by, for example,
```

```
c     P_A_S_S( x, y,z )  (without the "_")
c     in which case, all instances of "y" will be replaced with "x", and
c     all "z" by "y", and all "x  " by "z".
c
c    Why has this been done? Believe it or not, as an attempt at simplification.
c      In order to update any given row, PPM  requires information from up to
c      2. rows above below, fore, and aft of  the row under intense scrutiny.
c
c         Consider first just above and below.
c
c      This information must be from the same computational time, however,
c      as the computation progresses, the large storage array DDD becomes
c      a mixture of old timestep information and new timestep information
c      as updated zone averages (rhonu, pnu, unu, etc) are stored in DDD.
c      Therefore, a small working scratch space 5. rows deep (a slab) will
c      be created. Consider an X-pass, then the slab would be dimensioned
c      at least (1-nbdy:nx+nbdy,,-1,ny+2,5):
c
c               ----------------------------------------
c       islab=5 |                                      >
c               ----------------------------------------
c       islab=4 |                                      >
c               ----------------------------------------
c       islab=3 |                                      >
c               ----------------------------------------
c       islab=2 |                                      >
c               ----------------------------------------
c       islab=1 |                                      >
c               ----------------------------------------
c
c    initially, islab=1 is taken from ddd(*,*,-1,*)
c    initially, islab=2 is taken from ddd(*,*,0,*)
c    initially, islab=3 is taken from ddd(*,*,1,*)
c    initially, islab=4 is taken from ddd(*,*,2,*)
c    initially, islab=5 is taken from ddd(*,*,3,*)
c
c    Pointers to the correct slab planes (islab1bb, islab1b, islab1, islab1t,
c    islab1tt)
c    (for  2 planes aft, 1 plane aft, the row, 1. row fore, 2 rows fore)
c    which would have the values (1,2,3,4,5).  After the call to do_ppm and the
c    updated values stored in DDD, the counters are cycled to (2,3,4,5,1)
c    and the relavent required data from DDD is written into the row pointed to
c    by the new value of islabtt.
c
c    The user should also note that the velocities switch roles.  That is,
c    during an X-pass, u1 is the x-velocity and u2, u3 are the y- and
c    z-velocities, resp. while during a Y-pass, u1 is the y-velocity and u2, u3
c    are the z- and x-velocities, resp.  One should maintain a right-hand
c    coordinate system.  I don't believe that this is required (now) but
c    who knows what future perils await thee.
c
c
c    This reduces both the scratch space requirement and the amount of
c    data copying required. Believe it or not.
c
c    the xvelocity is element #3 in DDD
c    the yvelocity is element #4 in DDD
c    the zvelocity is element #5 in DDD
      kvx = 3
      kvy = 4
      kvz = 5

      do i = 1-nbdy, ny+nbdy+1
      xxl(i) = yl(i)
      enddo
c
c    initialize the first  slab
c
c     islab = 5 is done below at plane islabtt
c
c      ((do 100)
      do islab = 1, 4
       ix   = islab-2
      do iz = 1-nbdy,nz+nbdy
            do iy = 1-nbdy, ny+nbdy
              dddslab(iy, iz,islab,1) =  ddd(   1, ix, iy,iz)
              dddslab(iy, iz,islab,2) =  ddd(   2, ix, iy,iz)
              dddslab(iy, iz,islab,3) =  ddd(kvy, ix, iy,iz)
              dddslab(iy, iz,islab,4) =  ddd(kvz, ix, iy,iz)
              dddslab(iy, iz,islab,5) =  ddd(kvx  , ix, iy,iz)
            enddo
      enddo
      enddo
c      (do 100))

      islabbb = 0
      islabb  = 1
      islab   = 2
      islabt  = 3
      islabtt = 4

c
c      ((do 410)
      do ix   = 1, nx
c           (slabbb) 2. in front (slabtt), etc.
c        islabbb = 1+mod(ix  -1,5)
c        islabb  = 1+mod(ix  ,  5)
c        islab   = 1+mod(ix  +1,5)
```

```
c        islabt  = 1+mod(ix  +2,5)
c        islabtt = 1+mod(ix  +3,5)

         islabbb = 1+mod(islabbb,5)
         islabb  = 1+mod(islabb, 5)
         islab   = 1+mod(islab,  5)
         islabt  = 1+mod(islabt, 5)
         islabtt = 1+mod(islabtt,5)

c   At this point, we are ALWAYS able to fill plane islabtt from ddd
           lx   = ix  +2
            do lz = 1-nbdy, nz+nbdy
            do ly = 1-nbdy, ny+nbdy
              dddslab(ly, lz,islabtt,1) =  ddd(   1, lx, ly,lz)
              dddslab(ly, lz,islabtt,2) =  ddd(   2, lx, ly,lz)
              dddslab(ly, lz,islabtt,3) =  ddd(kvy, lx, ly,lz)
              dddslab(ly, lz,islabtt,4) =  ddd(kvz, lx, ly,lz)
              dddslab(ly, lz,islabtt,5) =  ddd(kvx  , lx, ly,lz)

            enddo
            enddo
c     ((do 400)
      do iz = 1, nz
c

         ip = 1-nbdy
         ycourmx = 0.
             call do_ppm_de0_3dc_gamma( xl(ip),
     ^ dddslab(ip,iz-2,islab,1),
     ^           dddslab(ip,iz,islab,1),
     ^                 dddslab(ip,iz+2,islab,1),
     ^                     dddslab(ip,iz,islabbb,1),
     ^                         dddslab(ip,iz,islabtt,1),
     ^ dddslab(ip,iz-2,islab,2),
     ^           dddslab(ip,iz,islab,2),
     ^                 dddslab(ip,iz+2,islab,2),
     ^                     dddslab(ip,iz,islabbb,2),
     ^                         dddslab(ip,iz,islabtt,2),
     ^ dddslab(ip,iz-2,islab,3),
     ^           dddslab(ip,iz,islab,3),
     ^                 dddslab(ip,iz+2,islab,3),
     ^                     dddslab(ip,iz,islabbb,3),
     ^                         dddslab(ip,iz,islabtt,3),
     ^ dddslab(ip,iz-2,islab,4),
     ^ dddslab(ip,iz-1,islab,4),
     ^           dddslab(ip,iz,islab,4),
     ^                 dddslab(ip,iz+1,islab,4),
     ^                 dddslab(ip,iz+2,islab,4),
     ^                     dddslab(ip,iz,islabbb,4),
     ^                         dddslab(ip,iz,islabtt,4),
     ^ dddslab(ip,iz,islabbb,5),
     ^    dddslab(ip,iz,islabb,5),
     ^       dddslab(ip,iz,islab,5),
     ^          dddslab(ip,iz,islabt,5),
     ^             dddslab(ip,iz,islabtt,5),
     ^                dddslab(ip,iz-2,islab,5),
     ^                dddslab(ip,iz+2,islab,5),
     ^           rhonu(ip),  pnu(ip), u1nu(ip), u2nu(ip), u3nu(ip),
     ^           gamma, dt, smlro, smallp, smallu, smalle,
     ^           ycourmx, ny, nbdy )
c

      print *,' iz, ix  , ycourmx=', iz, ix  , ycourmx

c
      courmx = max ( courmx, ycourmx )
c
         do iy = 1, ny
         ddd(   1,ix,iy,iz) = rhonu(iy)
         ddd(   2,ix,iy,iz) =  pnu(iy)
         ddd(kvy,ix,iy,iz) = u1nu(iy)
         ddd(kvz,ix,iy,iz) = u2nu(iy)
         ddd(kvx  ,ix,iy,iz) = u3nu(iy)
         enddo
c

      enddo
c     (do 400))

      enddo
c     (do 410))

c
c
c
c )
c
      print *,' pass y, z, x 1 courmx = ',courmx
c
c     Set fake zones according to boundary conditions.
      call bdrys ( xl,  yl, zl, ddd, 5, 5, 5, nx, ny, nz, nbdy )
c
c
c(
c
c======================================================================72
c                       -----  z-pass  -----
c
```

```
c   This is an m4 macro definition which will construct a pass
c   in the z direction.  It is invoked by, for example,
c    P_A_S_S( x, y,z )  (without the "_")
c    in which case, all instances of "z" will be replaced with "x", and
c    all "x" by "y", and all "y  " by "z".
c
c   Why has this been done? Believe it or not, as an attempt at simplification.
c     In order to update any given row, PPM  requires information from up to
c     2. rows above below, fore, and aft of  the row under intense scrutiny.
c
c        Consider first just above and below.
c
c     This information must be from the same computational time, however,
c     as the computation progresses, the large storage array DDD becomes
c     a mixture of old timestep information and new timestep information
c     as updated zone averages (rhonu, pnu, unu, etc) are stored in DDD.
c     Therefore, a small working scratch space 5. rows deep (a slab) will
c     be created. Consider an X-pass, then the slab would be dimensioned
c     at least (1-nbdy:nx+nbdy,,-1,ny+2,5):
c
c              ----------------------------------------
c      islab=5 |                                      >
c              ----------------------------------------
c      islab=4 |                                      >
c              ----------------------------------------
c      islab=3 |                                      >
c              ----------------------------------------
c      islab=2 |                                      >
c              ----------------------------------------
c      islab=1 |                                      >
c              ----------------------------------------
c
c   initially, islab=1 is taken from ddd(*,*,-1,*)
c   initially, islab=2 is taken from ddd(*,*,0,*)
c   initially, islab=3 is taken from ddd(*,*,1,*)
c   initially, islab=4 is taken from ddd(*,*,2,*)
c   initially, islab=5 is taken from ddd(*,*,3,*)
c
c   Pointers to the correct slab planes (islab1bb, islab1b, islab1, islab1t,
c   islab1tt)
c   (for  2 planes aft, 1 plane aft, the row, 1. row fore, 2 rows fore)
c   which would have the values (1,2,3,4,5).  After the call to do_ppm and the
c   updated values stored in DDD, the counters are cycled to (2,3,4,5,1)
c   and the relavent required data from DDD is written into the row pointed to
c   by the new value of islabtt.
c
c   The user should also note that the velocities switch roles.  That is,
c   during an X-pass, u1 is the x-velocity and u2, u3 are the y- and
c   z-velocities, resp. while during a Y-pass, u1 is the y-velocity and u2, u3
c   are the z- and x-velocities, resp.  One should maintain a right-hand
c   coordinate system.  I don't believe that this is required (now) but
c   who knows what future perils await thee.
c
c
c   This reduces both the scratch space requirement and the amount of
c   data copying required. Believe it or not.
c
c   the xvelocity is element #3 in DDD
c   the yvelocity is element #4 in DDD
c   the zvelocity is element #5 in DDD
      kvx = 3
      kvy = 4
      kvz = 5

      do i = 1-nbdy, nz+nbdy+1
      xxl(i) = zl(i)
      enddo
c
c   initialize the first  slab
c
c    islab = 5 is done below at plane islabtt
c
c      ((do 100)
      do islab = 1, 4
       iy   = islab-2
      do ix = 1-nbdy,nx+nbdy
            do iz = 1-nbdy, nz+nbdy
              dddslab(iz, ix,islab,1) =  ddd(   1, ix, iy,iz)
              dddslab(iz, ix,islab,2) =  ddd(   2, ix, iy,iz)
              dddslab(iz, ix,islab,3) =  ddd(kvz, ix, iy,iz)
              dddslab(iz, ix,islab,4) =  ddd(kvx, ix, iy,iz)
              dddslab(iz, ix,islab,5) =  ddd(kvy  , ix, iy,iz)
            enddo
      enddo
      enddo
c      (do 100))

      islabbb = 0
      islabb  = 1
      islab   = 2
      islabt  = 3
      islabtt = 4

c
c      ((do 410)
      do iy   = 1, ny
c           (slabbb) 2. in front (slabtt), etc.
c        islabbb = 1+mod(iy  -1,5)
```

```
c        islabb  = 1+mod(iy  ,  5)
c        islab   = 1+mod(iy  +1,5)
c        islabt  = 1+mod(iy  +2,5)
c        islabtt = 1+mod(iy  +3,5)

         islabbb = 1+mod(islabbb,5)
         islabb  = 1+mod(islabb, 5)
         islab   = 1+mod(islab,  5)
         islabt  = 1+mod(islabt, 5)
         islabtt = 1+mod(islabtt,5)

c   At this point, we are ALWAYS able to fill plane islabtt from ddd
           ly   = iy  +2
            do lx = 1-nbdy, nx+nbdy
            do lz = 1-nbdy, nz+nbdy
              dddslab(lz, lx,islabtt,1) =  ddd(   1, lx, ly,lz)
              dddslab(lz, lx,islabtt,2) =  ddd(   2, lx, ly,lz)
              dddslab(lz, lx,islabtt,3) =  ddd(kvz, lx, ly,lz)
              dddslab(lz, lx,islabtt,4) =  ddd(kvx, lx, ly,lz)
              dddslab(lz, lx,islabtt,5) =  ddd(kvy  , lx, ly,lz)

            enddo
            enddo
c     ((do 400)
      do ix = 1, nx
c

         ip = 1-nbdy
         zcourmx = 0.
             call do_ppm_de0_3dc_gamma( xl(ip),
     ^ dddslab(ip,ix-2,islab,1),
     ^           dddslab(ip,ix,islab,1),
     ^                 dddslab(ip,ix+2,islab,1),
     ^                     dddslab(ip,ix,islabbb,1),
     ^                         dddslab(ip,ix,islabtt,1),
     ^ dddslab(ip,ix-2,islab,2),
     ^           dddslab(ip,ix,islab,2),
     ^                 dddslab(ip,ix+2,islab,2),
     ^                     dddslab(ip,ix,islabbb,2),
     ^                         dddslab(ip,ix,islabtt,2),
     ^ dddslab(ip,ix-2,islab,3),
     ^           dddslab(ip,ix,islab,3),
     ^                 dddslab(ip,ix+2,islab,3),
     ^                     dddslab(ip,ix,islabbb,3),
     ^                         dddslab(ip,ix,islabtt,3),
     ^ dddslab(ip,ix-2,islab,4),
     ^ dddslab(ip,ix-1,islab,4),
     ^           dddslab(ip,ix,islab,4),
     ^                 dddslab(ip,ix+1,islab,4),
     ^                 dddslab(ip,ix+2,islab,4),
     ^                     dddslab(ip,ix,islabbb,4),
     ^                         dddslab(ip,ix,islabtt,4),
     ^ dddslab(ip,ix,islabbb,5),
     ^    dddslab(ip,ix,islabb,5),
     ^       dddslab(ip,ix,islab,5),
     ^          dddslab(ip,ix,islabt,5),
     ^             dddslab(ip,ix,islabtt,5),
     ^                dddslab(ip,ix-2,islab,5),
     ^                dddslab(ip,ix+2,islab,5),
     ^           rhonu(ip),  pnu(ip), u1nu(ip), u2nu(ip), u3nu(ip),
     ^           gamma, dt, smlro, smallp, smallu, smalle,
     ^           zcourmx, nz, nbdy )
c

      print *,' ix, iy  , zcourmx=', ix, iy  , zcourmx

c
      courmx = max ( courmx, zcourmx )
c
         do iz = 1, nz
         ddd(   1,ix,iy,iz) = rhonu(iz)
         ddd(   2,ix,iy,iz) =  pnu(iz)
         ddd(kvz,ix,iy,iz) = u1nu(iz)
         ddd(kvx,ix,iy,iz) = u2nu(iz)
         ddd(kvy  ,ix,iy,iz) = u3nu(iz)
         enddo
c

      enddo
c     (do 400))

      enddo
c     (do 410))

c
c
c
c )
c
      print *,' pass z, x, y 1 courmx = ',courmx
c
c     Set fake zones according to boundary conditions.
      call bdrys ( xl,  yl, zl, ddd, 5, 5, 5, nx, ny, nz, nbdy )
c
c
c(
c
c=====================================================================72
```

```
c                         -----  z-pass  -----
c
c   This is an m4 macro definition which will construct a pass
c   in the z direction.  It is invoked by, for example,
c    P_A_S_S( x, y,z )  (without the "_")
c    in which case, all instances of "z" will be replaced with "x", and
c    all "x" by "y", and all "y  " by "z".
c
c   Why has this been done? Believe it or not, as an attempt at simplification.
c     In order to update any given row, PPM  requires information from up to
c     2. rows above below, fore, and aft of  the row under intense scrutiny.
c
c        Consider first just above and below.
c
c     This information must be from the same computational time, however,
c     as the computation progresses, the large storage array DDD becomes
c     a mixture of old timestep information and new timestep information
c     as updated zone averages (rhonu, pnu, unu, etc) are stored in DDD.
c     Therefore, a small working scratch space 5. rows deep (a slab) will
c     be created. Consider an X-pass, then the slab would be dimensioned
c     at least (1-nbdy:nx+nbdy,,-1,ny+2,5):
c
c               ----------------------------------------
c      islab=5 |                                       >
c               ----------------------------------------
c      islab=4 |                                       >
c               ----------------------------------------
c      islab=3 |                                       >
c               ----------------------------------------
c      islab=2 |                                       >
c               ----------------------------------------
c      islab=1 |                                       >
c               ----------------------------------------
c
c   initially, islab=1 is taken from ddd(*,*,-1,*)
c   initially, islab=2 is taken from ddd(*,*,0,*)
c   initially, islab=3 is taken from ddd(*,*,1,*)
c   initially, islab=4 is taken from ddd(*,*,2,*)
c   initially, islab=5 is taken from ddd(*,*,3,*)
c
c   Pointers to the correct slab planes (islab1bb, islab1b, islab1, islab1t,
c   islab1tt)
c   (for  2 planes aft, 1 plane aft, the row, 1. row fore, 2 rows fore)
c   which would have the values (1,2,3,4,5).  After the call to do_ppm and the
c   updated values stored in DDD, the counters are cycled to (2,3,4,5,1)
c   and the relavent required data from DDD is written into the row pointed to
c   by the new value of islabtt.
c
c   The user should also note that the velocities switch roles.  That is,
c   during an X-pass, u1 is the x-velocity and u2, u3 are the y- and
c   z-velocities, resp. while during a Y-pass, u1 is the y-velocity and u2, u3
c   are the z- and x-velocities, resp.  One should maintain a right-hand
c   coordinate system.  I don't believe that this is required (now) but
c   who knows what future perils await thee.
c
c
c   This reduces both the scratch space requirement and the amount of
c   data copying required. Believe it or not.
c
c   the xvelocity is element #3 in DDD
c   the yvelocity is element #4 in DDD
c   the zvelocity is element #5 in DDD
      kvx = 3
      kvy = 4
      kvz = 5

      do i = 1-nbdy, nz+nbdy+1
      xxl(i) = zl(i)
      enddo
c
c   initialize the first  slab
c
c    islab = 5 is done below at plane islabtt
c
c      ((do 100)
      do islab = 1, 4
       iy   = islab-2
      do ix = 1-nbdy,nx+nbdy
            do iz = 1-nbdy, nz+nbdy
              dddslab(iz, ix,islab,1) =  ddd(   1, ix, iy,iz)
              dddslab(iz, ix,islab,2) =  ddd(   2, ix, iy,iz)
              dddslab(iz, ix,islab,3) =  ddd(kvz, ix, iy,iz)
              dddslab(iz, ix,islab,4) =  ddd(kvx, ix, iy,iz)
              dddslab(iz, ix,islab,5) =  ddd(kvy  , ix, iy,iz)
            enddo
      enddo
      enddo
c      (do 100))

      islabbb = 0
      islabb  = 1
      islab   = 2
      islabt  = 3
      islabtt = 4

c
c      ((do 410)
      do iy   = 1, ny
```

```
c           (slabbb) 2. in front (slabtt), etc.
c        islabbb = 1+mod(iy  -1,5)
c        islabb  = 1+mod(iy  ,  5)
c        islab   = 1+mod(iy  +1,5)
c        islabt  = 1+mod(iy  +2,5)
c        islabtt = 1+mod(iy  +3,5)

         islabbb = 1+mod(islabbb,5)
         islabb  = 1+mod(islabb, 5)
         islab   = 1+mod(islab,  5)
         islabt  = 1+mod(islabt, 5)
         islabtt = 1+mod(islabtt,5)

c   At this point, we are ALWAYS able to fill plane islabtt from ddd
           ly   = iy  +2
            do lx = 1-nbdy, nx+nbdy
            do lz = 1-nbdy, nz+nbdy
              dddslab(lz, lx,islabtt,1) =  ddd(   1, lx, ly,lz)
              dddslab(lz, lx,islabtt,2) =  ddd(   2, lx, ly,lz)
              dddslab(lz, lx,islabtt,3) =  ddd(kvz, lx, ly,lz)
              dddslab(lz, lx,islabtt,4) =  ddd(kvx, lx, ly,lz)
              dddslab(lz, lx,islabtt,5) =  ddd(kvy  , lx, ly,lz)

            enddo
            enddo
c     ((do 400)
      do ix = 1, nx
c

         ip = 1-nbdy
         zcourmx = 0.
             call do_ppm_de0_3dc_gamma( xl(ip),
     ^ dddslab(ip,ix-2,islab,1),
     ^           dddslab(ip,ix,islab,1),
     ^                 dddslab(ip,ix+2,islab,1),
     ^                     dddslab(ip,ix,islabbb,1),
     ^                         dddslab(ip,ix,islabtt,1),
     ^ dddslab(ip,ix-2,islab,2),
     ^           dddslab(ip,ix,islab,2),
     ^                 dddslab(ip,ix+2,islab,2),
     ^                     dddslab(ip,ix,islabbb,2),
     ^                         dddslab(ip,ix,islabtt,2),
     ^ dddslab(ip,ix-2,islab,3),
     ^           dddslab(ip,ix,islab,3),
     ^                 dddslab(ip,ix+2,islab,3),
     ^                     dddslab(ip,ix,islabbb,3),
     ^                         dddslab(ip,ix,islabtt,3),
     ^ dddslab(ip,ix-2,islab,4),
     ^ dddslab(ip,ix-1,islab,4),
     ^           dddslab(ip,ix,islab,4),
     ^                 dddslab(ip,ix+1,islab,4),
     ^                 dddslab(ip,ix+2,islab,4),
     ^                     dddslab(ip,ix,islabbb,4),
     ^                         dddslab(ip,ix,islabtt,4),
     ^ dddslab(ip,ix,islabbb,5),
     ^    dddslab(ip,ix,islabb,5),
     ^       dddslab(ip,ix,islab,5),
     ^          dddslab(ip,ix,islabt,5),
     ^             dddslab(ip,ix,islabtt,5),
     ^                dddslab(ip,ix-2,islab,5),
     ^                dddslab(ip,ix+2,islab,5),
     ^           rhonu(ip),  pnu(ip), u1nu(ip), u2nu(ip), u3nu(ip),
     ^           gamma, dt, smlro, smallp, smallu, smalle,
     ^           zcourmx, nz, nbdy )
c

      print *,' ix, iy  , zcourmx=', ix, iy  , zcourmx

c
      courmx = max ( courmx, zcourmx )
c
         do iz = 1, nz
         ddd(   1,ix,iy,iz) = rhonu(iz)
         ddd(   2,ix,iy,iz) =  pnu(iz)
         ddd(kvz,ix,iy,iz) = u1nu(iz)
         ddd(kvx,ix,iy,iz) = u2nu(iz)
         ddd(kvy  ,ix,iy,iz) = u3nu(iz)
         enddo
c

      enddo
c     (do 400))

      enddo
c     (do 410))

c
c
c
c )
c
      print *,' pass z, x, y 2 courmx = ',courmx
c

c     Set fake zones according to boundary conditions.
      call bdrys ( xl,  yl, zl, ddd, 5, 5, 5, nx, ny, nz, nbdy )
c
c
```

```
c(
c
c=====================================================================72
c                       -----  y-pass  -----
c
c   This is an m4 macro definition which will construct a pass
c   in the y direction.  It is invoked by, for example,
c    P_A_S_S( x, y,z )  (without the "_")
c    in which case, all instances of "y" will be replaced with "x", and
c    all "z" by "y", and all "x  " by "z".
c
c   Why has this been done? Believe it or not, as an attempt at simplification.
c     In order to update any given row, PPM  requires information from up to
c     2. rows above below, fore, and aft of  the row under intense scrutiny.
c
c        Consider first just above and below.
c
c     This information must be from the same computational time, however,
c     as the computation progresses, the large storage array DDD becomes
c     a mixture of old timestep information and new timestep information
c     as updated zone averages (rhonu, pnu, unu, etc) are stored in DDD.
c     Therefore, a small working scratch space 5. rows deep (a slab) will
c     be created. Consider an X-pass, then the slab would be dimensioned
c     at least (1-nbdy:nx+nbdy,,-1,ny+2,5):
c
c               ----------------------------------------
c      islab=5 |                                        >
c               ----------------------------------------
c      islab=4 |                                        >
c               ----------------------------------------
c      islab=3 |                                        >
c               ----------------------------------------
c      islab=2 |                                        >
c               ----------------------------------------
c      islab=1 |                                        >
c               ----------------------------------------
c
c   initially, islab=1 is taken from ddd(*,*,-1,*)
c   initially, islab=2 is taken from ddd(*,*,0,*)
c   initially, islab=3 is taken from ddd(*,*,1,*)
c   initially, islab=4 is taken from ddd(*,*,2,*)
c   initially, islab=5 is taken from ddd(*,*,3,*)
c
c   Pointers to the correct slab planes (islab1bb, islab1b, islab1, islab1t,
c   islab1tt)
c   (for  2 planes aft, 1 plane aft, the row, 1. row fore, 2 rows fore)
c   which would have the values (1,2,3,4,5).  After the call to do_ppm and the
c   updated values stored in DDD, the counters are cycled to (2,3,4,5,1)
c   and the relavent required data from DDD is written into the row pointed to
c   by the new value of islabtt.
c
c   The user should also note that the velocities switch roles.  That is,
c   during an X-pass, u1 is the x-velocity and u2, u3 are the y- and
c   z-velocities, resp. while during a Y-pass, u1 is the y-velocity and u2, u3
c   are the z- and x-velocities, resp.  One should maintain a right-hand
c   coordinate system.  I don't believe that this is required (now) but
c   who knows what future perils await thee.
c
c
c   This reduces both the scratch space requirement and the amount of
c   data copying required. Believe it or not.
c
c   the xvelocity is element #3 in DDD
c   the yvelocity is element #4 in DDD
c   the zvelocity is element #5 in DDD
      kvx = 3
      kvy = 4
      kvz = 5

      do i = 1-nbdy, ny+nbdy+1
      xxl(i) = yl(i)
      enddo
c
c   initialize the first  slab
c
c    islab = 5 is done below at plane islabtt
c
c      ((do 100)
      do islab = 1, 4
       ix   = islab-2
      do iz = 1-nbdy,nz+nbdy
            do iy = 1-nbdy, ny+nbdy
              dddslab(iy, iz,islab,1) =  ddd(   1, ix, iy,iz)
              dddslab(iy, iz,islab,2) =  ddd(   2, ix, iy,iz)
              dddslab(iy, iz,islab,3) =  ddd(kvy, ix, iy,iz)
              dddslab(iy, iz,islab,4) =  ddd(kvz, ix, iy,iz)
              dddslab(iy, iz,islab,5) =  ddd(kvx  , ix, iy,iz)
            enddo
      enddo
      enddo
c      (do 100))

      islabbb = 0
      islabb  = 1
      islab   = 2
      islabt  = 3
      islabtt = 4
```

```
c
c     ((do 410)
      do ix   = 1, nx
c           (slabbb) 2. in front (slabtt), etc.
c        islabbb = 1+mod(ix  -1,5)
c        islabb  = 1+mod(ix  ,  5)
c        islab   = 1+mod(ix  +1,5)
c        islabt  = 1+mod(ix  +2,5)
c        islabtt = 1+mod(ix  +3,5)

         islabbb = 1+mod(islabbb,5)
         islabb  = 1+mod(islabb, 5)
         islab   = 1+mod(islab,  5)
         islabt  = 1+mod(islabt, 5)
         islabtt = 1+mod(islabtt,5)

c   At this point, we are ALWAYS able to fill plane islabtt from ddd
           lx   = ix  +2
            do lz = 1-nbdy, nz+nbdy
            do ly = 1-nbdy, ny+nbdy
              dddslab(ly, lz,islabtt,1) =  ddd(   1, lx, ly,lz)
              dddslab(ly, lz,islabtt,2) =  ddd(   2, lx, ly,lz)
              dddslab(ly, lz,islabtt,3) =  ddd(kvy, lx, ly,lz)
              dddslab(ly, lz,islabtt,4) =  ddd(kvz, lx, ly,lz)
              dddslab(ly, lz,islabtt,5) =  ddd(kvx  , lx, ly,lz)

            enddo
            enddo
c     ((do 400)
      do iz = 1, nz
c

         ip = 1-nbdy
         ycourmx = 0.
             call do_ppm_de0_3dc_gamma( xl(ip),
     ^ dddslab(ip,iz-2,islab,1),
     ^           dddslab(ip,iz,islab,1),
     ^                 dddslab(ip,iz+2,islab,1),
     ^                     dddslab(ip,iz,islabbb,1),
     ^                         dddslab(ip,iz,islabtt,1),
     ^ dddslab(ip,iz-2,islab,2),
     ^           dddslab(ip,iz,islab,2),
     ^                 dddslab(ip,iz+2,islab,2),
     ^                     dddslab(ip,iz,islabbb,2),
     ^                         dddslab(ip,iz,islabtt,2),
     ^ dddslab(ip,iz-2,islab,3),
     ^           dddslab(ip,iz,islab,3),
     ^                 dddslab(ip,iz+2,islab,3),
     ^                     dddslab(ip,iz,islabbb,3),
     ^                         dddslab(ip,iz,islabtt,3),
     ^ dddslab(ip,iz-2,islab,4),
     ^ dddslab(ip,iz-1,islab,4),
     ^           dddslab(ip,iz,islab,4),
     ^                 dddslab(ip,iz+1,islab,4),
     ^                 dddslab(ip,iz+2,islab,4),
     ^                     dddslab(ip,iz,islabbb,4),
     ^                         dddslab(ip,iz,islabtt,4),
     ^ dddslab(ip,iz,islabbb,5),
     ^    dddslab(ip,iz,islabb,5),
     ^       dddslab(ip,iz,islab,5),
     ^          dddslab(ip,iz,islabt,5),
     ^             dddslab(ip,iz,islabtt,5),
     ^                dddslab(ip,iz-2,islab,5),
     ^                dddslab(ip,iz+2,islab,5),
     ^           rhonu(ip),  pnu(ip), u1nu(ip), u2nu(ip), u3nu(ip),
     ^           gamma, dt, smlro, smallp, smallu, smalle,
     ^           ycourmx, ny, nbdy )
c

      print *,' iz, ix  , ycourmx=', iz, ix  , ycourmx

c
      courmx = max ( courmx, ycourmx )
c
         do iy = 1, ny
         ddd(   1,ix,iy,iz) = rhonu(iy)
         ddd(   2,ix,iy,iz) =  pnu(iy)
         ddd(kvy,ix,iy,iz) = u1nu(iy)
         ddd(kvz,ix,iy,iz) = u2nu(iy)
         ddd(kvx  ,ix,iy,iz) = u3nu(iy)
         enddo
c

      enddo
c     (do 400))

      enddo
c     (do 410))

c
c
c
c )
c
      print *,' pass y, z, x 2 courmx = ',courmx
c
c     Set fake zones according to boundary conditions.
      call bdrys ( xl,  yl, zl, ddd, 5, 5, 5, nx, ny, nz, nbdy )
```

```
c
c
c(
c
c=====================================================================72
c                       -----  x-pass  -----
c
c   This is an m4 macro definition which will construct a pass
c   in the x direction.  It is invoked by, for example,
c    P_A_S_S( x, y,z )  (without the "_")
c    in which case, all instances of "x" will be replaced with "x", and
c    all "y" by "y", and all "z " by "z".
c
c   Why has this been done? Believe it or not, as an attempt at simplification.
c     In order to update any given row, PPM  requires information from up to
c     2. rows above below, fore, and aft of  the row under intense scrutiny.
c
c        Consider first just above and below.
c
c     This information must be from the same computational time, however,
c     as the computation progresses, the large storage array DDD becomes
c     a mixture of old timestep information and new timestep information
c     as updated zone averages (rhonu, pnu, unu, etc) are stored in DDD.
c     Therefore, a small working scratch space 5. rows deep (a slab) will
c     be created. Consider an X-pass, then the slab would be dimensioned
c     at least (1-nbdy:nx+nbdy,,-1,ny+2,5):
c
c              ----------------------------------------
c      islab=5 |                                      >
c              ----------------------------------------
c      islab=4 |                                      >
c              ----------------------------------------
c      islab=3 |                                      >
c              ----------------------------------------
c      islab=2 |                                      >
c              ----------------------------------------
c      islab=1 |                                      >
c              ----------------------------------------
c
c   initially, islab=1 is taken from ddd(*,*,-1,*)
c   initially, islab=2 is taken from ddd(*,*,0,*)
c   initially, islab=3 is taken from ddd(*,*,1,*)
c   initially, islab=4 is taken from ddd(*,*,2,*)
c   initially, islab=5 is taken from ddd(*,*,3,*)
c
c   Pointers to the correct slab planes (islab1bb, islab1b, islab1, islab1t,
c   islab1tt)
c   (for  2 planes aft, 1 plane aft, the row, 1. row fore, 2 rows fore)
c   which would have the values (1,2,3,4,5).  After the call to do_ppm and the
c   updated values stored in DDD, the counters are cycled to (2,3,4,5,1)
c   and the relavent required data from DDD is written into the row pointed to
c   by the new value of islabtt.
c
c   The user should also note that the velocities switch roles.  That is,
c   during an X-pass, u1 is the x-velocity and u2, u3 are the y- and
c   z-velocities, resp. while during a Y-pass, u1 is the y-velocity and u2, u3
c   are the z- and x-velocities, resp.  One should maintain a right-hand
c   coordinate system.  I don't believe that this is required (now) but
c   who knows what future perils await thee.
c
c
c   This reduces both the scratch space requirement and the amount of
c   data copying required. Believe it or not.
c
c   the xvelocity is element #3 in DDD
c   the yvelocity is element #4 in DDD
c   the zvelocity is element #5 in DDD
      kvx = 3
      kvy = 4
      kvz = 5

      do i = 1-nbdy, nx+nbdy+1
      xxl(i) = xl(i)
      enddo
c
c   initialize the first  slab
c
c    islab = 5 is done below at plane islabtt
c
c      ((do 100)
      do islab = 1, 4
       iz  = islab-2
      do iy = 1-nbdy,ny+nbdy
            do ix = 1-nbdy, nx+nbdy
              dddslab(ix, iy,islab,1) =  ddd(   1, ix, iy,iz)
              dddslab(ix, iy,islab,2) =  ddd(   2, ix, iy,iz)
              dddslab(ix, iy,islab,3) =  ddd(kvx, ix, iy,iz)
              dddslab(ix, iy,islab,4) =  ddd(kvy, ix, iy,iz)
              dddslab(ix, iy,islab,5) =  ddd(kvz , ix, iy,iz)
            enddo
      enddo
      enddo
c      (do 100))

      islabbb = 0
      islabb  = 1
      islab   = 2
      islabt  = 3
```

```
      islabtt = 4

c
c     ((do 410)
      do iz  = 1, nz
c           (slabbb) 2. in front (slabtt), etc.
c         islabbb = 1+mod(iz -1,5)
c         islabb  = 1+mod(iz ,  5)
c         islab   = 1+mod(iz +1,5)
c         islabt  = 1+mod(iz +2,5)
c         islabtt = 1+mod(iz +3,5)

          islabbb = 1+mod(islabbb,5)
          islabb  = 1+mod(islabb, 5)
          islab   = 1+mod(islab,  5)
          islabt  = 1+mod(islabt, 5)
          islabtt = 1+mod(islabtt,5)

c   At this point, we are ALWAYS able to fill plane islabtt from ddd
            lz  = iz +2
             do ly = 1-nbdy, ny+nbdy
             do lx = 1-nbdy, nx+nbdy
               dddslab(lx, ly,islabtt,1) =  ddd(   1, lx, ly,lz)
               dddslab(lx, ly,islabtt,2) =  ddd(   2, lx, ly,lz)
               dddslab(lx, ly,islabtt,3) =  ddd(kvx, lx, ly,lz)
               dddslab(lx, ly,islabtt,4) =  ddd(kvy, lx, ly,lz)
               dddslab(lx, ly,islabtt,5) =  ddd(kvz , lx, ly,lz)

             enddo
             enddo
c     ((do 400)
      do iy = 1, ny
c

          ip = 1-nbdy
          xcourmx = 0.
              call do_ppm_de0_3dc_gamma( xl(ip),
     ^ dddslab(ip,iy-2,islab,1),
     ^           dddslab(ip,iy,islab,1),
     ^                 dddslab(ip,iy+2,islab,1),
     ^                     dddslab(ip,iy,islabbb,1),
     ^                         dddslab(ip,iy,islabtt,1),
     ^ dddslab(ip,iy-2,islab,2),
     ^           dddslab(ip,iy,islab,2),
     ^                 dddslab(ip,iy+2,islab,2),
     ^                     dddslab(ip,iy,islabbb,2),
     ^                         dddslab(ip,iy,islabtt,2),
     ^ dddslab(ip,iy-2,islab,3),
     ^           dddslab(ip,iy,islab,3),
     ^                 dddslab(ip,iy+2,islab,3),
     ^                     dddslab(ip,iy,islabbb,3),
     ^                         dddslab(ip,iy,islabtt,3),
     ^ dddslab(ip,iy-2,islab,4),
     ^ dddslab(ip,iy-1,islab,4),
     ^           dddslab(ip,iy,islab,4),
     ^                 dddslab(ip,iy+1,islab,4),
     ^                 dddslab(ip,iy+2,islab,4),
     ^                     dddslab(ip,iy,islabbb,4),
     ^                         dddslab(ip,iy,islabtt,4),
     ^ dddslab(ip,iy,islabbb,5),
     ^    dddslab(ip,iy,islabb,5),
     ^       dddslab(ip,iy,islab,5),
     ^          dddslab(ip,iy,islabt,5),
     ^             dddslab(ip,iy,islabtt,5),
     ^                dddslab(ip,iy-2,islab,5),
     ^                dddslab(ip,iy+2,islab,5),
     ^           rhonu(ip),  pnu(ip), u1nu(ip), u2nu(ip), u3nu(ip),
     ^           gamma, dt, smlro, smallp, smallu, smalle,
     ^           xcourmx, nx, nbdy )
c

      print *,' iy, iz , xcourmx=', iy, iz , xcourmx

c
      courmx = max ( courmx, xcourmx )
c
          do ix = 1, nx
          ddd(   1,ix,iy,iz) = rhonu(ix)
          ddd(   2,ix,iy,iz) =  pnu(ix)
          ddd(kvx,ix,iy,iz) = u1nu(ix)
          ddd(kvy,ix,iy,iz) = u2nu(ix)
          ddd(kvz ,ix,iy,iz) = u3nu(ix)
          enddo
c

      enddo
c     (do 400))

      enddo
c     (do 410))

c
c
c
c )
c
      print *,' pass x, y, z 2 courmx = ',courmx
c
```

```
      return
      end
```

The the boundary zone requirements for the ppm templates are asymmetric. That is, a NBDY zone are needed at each end of the arrays in the logical direction of the pass, but only two zones are required in the directions transverse. As it would be wasteful to set values in unrequired zones so the actual number of boundary zones in each direction is passed to the boundary routine.

```
c
c     Set fake zones according to boundary conditions.
      call bdrys ( xl,  yl, zl, ddd, 5, 5, 5, nx, ny, nz, nbdy )
```

```
      subroutine bdrys ( xl,  yl, zl, ddd, nxtoset, nytoset, nztoset,
     ^                                       nx, ny, nz, nbdy )

      dimension     xl(-nbdy+1:nx+nbdy+1),  yl(-nbdy+1:ny+nbdy+1)
      dimension     Zl(-nbdy+1:nZ+nbdy+1)
      dimension  ddd(5,1-nbdy:nx+nbdy,
     ^                 1-nbdy:ny+nbdy,
     ^                 1-nbdy:nz+nbdy)
c
c
c
c     nbdy   = the number of fake zones needed to implement boundary
c              conditions so that in the hydrodynamical calculation
c              all zones can be treated as if they were interior zones.
c     nxtoset= the number of boundary zones to set at each end in
c               the x-dimension for this pass.
c     nytoset= the number of boundary zones to set at each end in
c               the y-dimension for this pass.
c     nztoset= the number of boundary zones to set at each end in
c               the z-dimension for this pass.
c     nx     = the number of real zones in the x-dimension for this
c              tile of the computational domain.
c     ny     = the number of real zones in the y-dimension for this
c              tile of the computational domain.
c     nz     = the number of real zones in the z-dimension for this
c              tile of the computational domain.
c
c      xl    = the x-grid for this tile.
c      yl    = the y-grid for this tile.
c      zl    = the z-grid for this tile.
c
c     rrho   = the zone-averaged densities for this tile.
c     pp     = the zone-averaged pressures for this tile.
c     uux    = the zone-averaged x-velocities for this tile.
c     uuy    = the zone-averaged y-velocities for this tile.
c     uuz    = the zone-averaged z-velocities for this tile.
c
c     ddd    = the 4-D array into which the 5 principal 3-D arrays of
c              basic physical variables are interleaved in order to
c              optimize memory bandwidth on microprocessor systems.
c     ddd(1,i,j) = rrho(i,j)
c     ddd(2,i,j) = pp(i,j)
c     ddd(3,i,j) = uux(i,j)
c     ddd(4,i,j) = uuy(i,j)
c     ddd(5,i,j) = uuz(i,j)
c
c     Apply boundary conditions, reflecting only.
c
c     Reflecting boundary conditions:
c
      do i = 1-nxtoset,0
       xl(i) =  2.0*xl(1) - xl(2-i)
      enddo
      do i = nx+2,nx+nxtoset+1
       xl(i) =  2.0*xl(nx+1) - xl(2*nx+2-i)
      enddo
c
      do i = 1-nytoset,0
       yl(i) =  2.0*yl(1) - yl(2-i)
      enddo
      do i = ny+2,ny+nytoset+1
       yl(i) =  2.0*yl(ny+1) - yl(2*ny+2-i)
      enddo
c
      do i = 1-nztoset,0
       zl(i) =  2.0*zl(1) - zl(2-i)
      enddo
      do i = nz+2,nz+nztoset+1
       zl(i) =  2.0*zl(nz+1) - zl(2*nz+2-i)
      enddo
c
c
c     reflecting boundary conditions:
c
c
      do 1200   k = 1-nztoset,0
      do 1200   j = 1,ny
      do 1200   i = 1,nx
```

```
      ddd(1,i,j,k) =  ddd(1,i,j,1-k)
      ddd(2,i,j,k) =  ddd(2,i,j,1-k)
      ddd(3,i,j,k) =  ddd(3,i,j,1-k)
      ddd(4,i,j,k) =  ddd(4,i,j,1-k)
      ddd(5,i,j,k) = -ddd(5,i,j,1-k)
1200  continue
c
      do 1210   k = nz+1,nz+nztoset
      do 1210   j = 1,ny
      do 1210   i = 1,nx
      ddd(1,i,j,k) = ddd(1,i,j,2*nz+1-k)
      ddd(2,i,j,k) = ddd(2,i,j,2*nz+1-k)
      ddd(3,i,j,k) = ddd(3,i,j,2*nz+1-k)
      ddd(4,i,j,k) = ddd(4,i,j,2*nz+1-k)
      ddd(5,i,j,k) =-ddd(5,i,j,2*nz+1-k)
1210  continue
c
      do 1300   k = 1-nztoset,nz+nztoset
      do 1300   j = 1-nytoset,0
      do 1300   i = 1,nx
      ddd(1,i,j,k) = ddd(1,i,1-j,k)
      ddd(2,i,j,k) = ddd(2,i,1-j,k)
      ddd(3,i,j,k) = ddd(3,i,1-j,k)
      ddd(4,i,j,k) =-ddd(4,i,1-j,k)
      ddd(5,i,j,k) = ddd(5,i,1-j,k)
1300  continue
c
      do 1310   k = 1-nztoset,nz+nztoset
      do 1310   j = ny+1,ny+nytoset
      do 1310   i = 1,nx
      ddd(1,i,j,k) = ddd(1,i,2*ny+1-j,k)
      ddd(2,i,j,k) = ddd(2,i,2*ny+1-j,k)
      ddd(3,i,j,k) = ddd(3,i,2*ny+1-j,k)
      ddd(4,i,j,k) =-ddd(4,i,2*ny+1-j,k)
      ddd(5,i,j,k) = ddd(5,i,2*ny+1-j,k)
1310  continue
c
      do 1400   k = 1-nztoset,nz+nztoset
      do 1400   j = 1-nytoset,ny+nytoset
      do 1400   i = 1-nxtoset,0
      ddd(1,i,j,k) = ddd(1,1-i,j,k)
      ddd(2,i,j,k) = ddd(2,1-i,j,k)
      ddd(3,i,j,k) =-ddd(3,1-i,j,k)
      ddd(4,i,j,k) = ddd(4,1-i,j,k)
      ddd(5,i,j,k) = ddd(5,1-i,j,k)
1400  continue
c
      do 1410   k = 1-nztoset,nz+nztoset
      do 1410   j = 1-nytoset,ny+nytoset
      do 1410   i = nx+1,nx+nxtoset
      ddd(1,i,j,k) = ddd(1,2*nx+1-i,j,k)
      ddd(2,i,j,k) = ddd(2,2*nx+1-i,j,k)
      ddd(3,i,j,k) =-ddd(3,2*nx+1-i,j,k)
      ddd(4,i,j,k) = ddd(4,2*nx+1-i,j,k)
      ddd(5,i,j,k) = ddd(5,2*nx+1-i,j,k)
1410  continue
c
c
c
c
      return
      end
```

Only reflecting boundary conditions are allowed in the example.

This dependence on data transverse to the pass also requires some care during the sequential calls to the ppmlib template which updates a row of data. If one merely extracted the required five rows from `ddd` and then stored in `ddd` the new zone averaged in their proper location within that array, the information sent to the template would quickly become a mixture of data at the old and new timesteps as the loop progressed through the `ddd` array. This is avoided here by using a 5x5 strip temporary array, `dddslag` and constructing the pointers `slabbb`, `slabb`, `slab`, `slabt`, and `slabtt`, which point to the location of the data for the plane two planes below, one plane below the plane being updated, the plane being updated, one plane above, and two planes above, resp. Representative results at four different times are shown in figure 20.