The VH-1 User's Guide

The Virginia Bulls

April 3, 1995

Contents

1 Preface **3**

2 PPM: A Brief Review **3**

3 VH-1 **5**

3.1 One-dimensional hydrodynamics 6

3.1.1 **sweepx2d** 6

3.1.2 **boundary.f** 7

3.1.3 **fictforces** 7

3.1.4 **gravity** 7

3.1.5 **ppm** 8

3.1.6 **flatten** 8

3.1.7 **paraset** 8

3.1.8 **parabola** 9

3.1.9 **states** 9

3.1.10 **riemann** 10

3.1.11 **evolve** 11

3.1.12 **remap** 11

3.2 The rest of the program 13

3.2.1 **mainNd.f** 13

3.2.2 **equalgrd** 13

3.2.3 **initoned, inittwod, initthreed** 14

3.2.4 **timestep** 14

3.2.5 **printNd.f** 14

3.2.6 **dumpNd.f** 14

3.3 Running VH-1 15

3.4 The Godunov Code 17

4 Test Problems **17**

4.1 Sod Shock Tube 18

4.2 1D Advection of a step in density 18

4.3 Double Mach Reflection 18

4.4 Mach 3 Wind Tunnel with a Step 19

4.5 Supersonic Flow Past a Cylinder/Sphere 19

4.6 Hawley-Zabusky 2D Angled Shock Tube 19

4.7 1D Hydrostatic Equilibrium in a Polytropic Star 19

4.8 1D Supernova in a power-law star 20

4.9 1D Supernova in a polytropic star 20

4.10 Geometry test problems 20

5 Dissipation **20**
5.1 Shock Flattening 21
5.2 Grid Wiggling 21
5.3 Artificial Viscosity 22

6 Boundary Conditions **22**
6.1 Reflecting 22
6.2 Fixed Inflow 23
6.3 Inflow/Outflow 23
6.4 Other Boundary Conditions 24

7 Optimization and Efficiency **24**

1 Preface

VH-1 was written by the numerical astrophysics group at the University of Virginia Institute of Theoretical Astrophysics in 1990–1991, and is based on the description of the piecewise parabolic method (PPM) given in Colella and Woodward (1984, hereafter C&W). While many individuals at UVa contributed to the development of VH-1, the bulk of the code writing and testing was done by John Hawley, John Blondin, Greg Lindahl, and Eric Lufkin. This User's Guide was written by John Blondin and Greg Lindahl. John Blondin is responsible for all the correct parts, while Greg Lindahl inserted bugs.

Our motivation for writing VH-1 is to provide the astrophysics community with a freely-redistributable implementation of PPM that can be adapted by someone, already familiar with finite differencing, to their own research. VH-1 is **NOT** idiot proof. No attempt has been made to provide a universal program that can be blindly applied to any problem in hydrodynamics. To that end, we expect the VH-1 user to be able to change the program to suit their own problem, and to devise test problems to ensure that the program does not have bugs which we have not caught in our testing for our particular problems. In addition, we assume that the reader of this Guide and user of this program is Fortran-literate, and that they will know how to compile, link, and run VH-1 on a local machine.

This guide to VH-1 will not repeat the detailed description of PPM given in C&W, but will merely provide a brief overview of the algorithm in § 2. It is imperative that the VH-1 user read and understand C&Win addition to reading this manual.

The following sections provide a brief overview of PPM, a detailed description of VH-1, and instructions on how to run it. We also discuss some general features of the program and algorithm that we have found necessary to pay careful attention to, and that therefore should be of interest to the VH-1 user. These include dissipation techniques and coordinate singularities. A few test problems are included with the program. A description of each of these problems and an example of VH-1 output for each problem is given in § 4.

2 PPM: A Brief Review

VH-1 uses finite difference techniques to solve the equations of ideal, inviscid, compressible fluid flow. These equations, written in conservative Eulerian form, are:

$$\partial_t \rho + \nabla \cdot (\rho \mathbf{u}) = 0,$$

$$\partial_t (\rho \mathbf{u}) + \nabla \cdot (\rho \mathbf{u}\mathbf{u}) + \nabla p = \mathbf{F},$$

$$\partial_t (\rho \mathcal{E}) + \nabla \cdot (\rho \mathcal{E} \mathbf{u}) + \nabla (p\mathbf{u}) = G + \rho \mathbf{u} \cdot \mathbf{F}$$

where $\mathcal{E} = u^2/2 + (\gamma - 1)^{-1} p/\rho$ is the total specific energy, ρ is the mass density, p the pressure, and $\mathbf{F}$ and G are momentum and energy source terms (e.g., gravity, radiative cooling), and ∂_t is the partial derivative with respect to time. VH-1 does not include an energy source term, but such a term can be readily added using an operator splitting method. Because a force term is already present in the form of fictitious forces for a curvilinear grid, we have included an arbitrary body force term in VH-1.

VH-1 is written as a Lagrangian hydrodynamics program coupled with a remap onto the original Eulerian grid. This implementation of PPM is referred to as PPM-LR. Our primary reason for using PPM-LR rather than the direct Eulerian method (PPM-DE) is to avoid the subtleties associated with constructing the correct input states for the Riemann problem in PPM-DE (see pages 190-191 in C&W). Despite the fact that PPM-LR requires a second call to the interpolation routines for the remap step, the computation of the input states is simplified, and the Courant constraint on the timestep is less restrictive, being the minimum of $(\Delta x/u_x, \Delta x/c_s)$ rather than $\Delta x/(u_x + c_s)$. In practice we find PPM-LR to be less dissipative than PPM-DE (not always a good thing), and better at maintaining contact discontinuities without the aid of a contact steepener.

The following is an outline of the PPM-LR algorithm. For specific details see C&W. As an illustration, let us consider the equations of ideal hydrodynamics in Lagrangian coordinates in one dimension, planar geometry and no source terms. Written in conservative form, these equations are:

$$\partial_t V - \partial_m u = 0$$

$$\partial_t u + \partial_m p = 0$$

$$\partial_t \mathcal{E} + \partial_m up = 0$$

where V, u, p, and $\mathcal{E}$ are the specific volume, velocity, pressure, and total energy of the gas, ∂_t is the partial derivative with respect to time, and ∂_m is the partial derivative with respect to the mass coordinate. The gas density ρ and internal energy e are related to these quantities via the relations

$$\rho = 1/V, \qquad e = \mathcal{E} - \frac{1}{2}u^2, \qquad p = (\gamma - 1)\rho e.$$

The conservation equations can be finite differenced as:

$$r^{n+1}_{j+\frac{1}{2}} = r^n_{j+\frac{1}{2}} + \Delta t \bar{u}_{j+\frac{1}{2}}$$

$$u^{n+1}_j = u^n_j + \frac{\Delta t}{\Delta m_j}(\bar{p}_{j-\frac{1}{2}} - \bar{p}_{j+\frac{1}{2}})$$

$$\mathcal{E}^{n+1}_j = \mathcal{E}^n_j + \frac{\Delta t}{\Delta m_j}(\bar{u}_{j-\frac{1}{2}}\bar{p}_{j-\frac{1}{2}} - \bar{u}_{j+\frac{1}{2}}\bar{p}_{j+\frac{1}{2}})$$

where the subscript j refers to zone-averaged values, the subscripts $j - \frac{1}{2}$ and $j + \frac{1}{2}$ refer to values at the left and right-hand interfaces of the zone, and the superscript is the timestep. The variables $\bar{u}$ and $\bar{p}$ are the time-averaged values of the velocity and pressure at the (Lagrangian) zone interfaces. The challenge is to obtain accurate, stable estimates of $\bar{u}$ and $\bar{p}$.

The approach of Godunov's method is to obtain these time-averaged quantities by approximating the flow at each zone interface during each timestep with a Riemann shock tube problem. At the beginning of the timestep, the zone interface is modeled as a discontinuity separating two uniform states given by the zone averages on the left and right side of the zone interface (zones $j - 1$ and j). This constructed Riemann problem is then solved to find the time-averaged value of the velocity and pressure ($\bar{u}$ and $\bar{p}$) at this discontinuity (zone interface). The solution of the non-linear Riemann problem typically requires some kind of iterative procedure, as described in van Leer (1979) and in Woodward (1986).

PPM improves upon Godunov's method by using more accurate guesses for the input states to the Riemann problem (the values on either side of the interface). Using a quadratic (parabolic) interpolation of the fluid variables in each zone, the Riemann input states are taken to be the average over that part of the zone that can be reached by a sound wave in a time dt, i.e., the characteristic domain of dependence.

Given the time-averaged velocity and pressure, the hydrodynamic equations are differenced to obtain the values of the fundamental variables (x, u, and e) at $t + dt$. Then the fluid variables are instantaneously remapped from the Lagrangian coordinate system to the stationary Eulerian grid. This remap step uses the same quadratic interpolation method that was used in the hydrodynamics step.

This description makes PPM sound relatively simple, but there are many key ingredients that we have skipped over, including the parabolic interpolation and monotonicity constraints, the iterative Riemann solver, and additional dissipation mechanisms. We will give a brief discussion of these first two topics in the appropriate subsections of § 3 and postpone the discussion of dissipation until § 5 (see also C&W).

3 VH-1

The following description of the program assumes the reader is familiar with finite differencing in general and with PPM in particular. For specific details we will occasionally refer the reader to various references on PPM. In the following two subsections, we provide a detailed description of the program, subroutine by subroutine. But first, we give a general introduction to VH-1. Subroutine names are referred to in bold type, **ppm**, and generally the subroutine **foo** can be found in the file **foo.f**. Variable names are given in a typewriter typeface, `ncycle`.

VH-1 is currently written as a 1D, 2D, or 3D program, with each versions employing the same subroutines for the hydrodynamics calculation. The only difference is that the 2D program calls the 1D hydro sweep for both the X and the Y directions, while the 1D program only calls the hydro sweep for one dimension. Extension of the program to 3D is relatively trivial, given the 2D program, but we recommend becoming familiar with the 1D and 2D versions before attempting 3D computations. There are separate main programs, **main1d.f**, **main2d.f**, and **main3d.f**. In general, files specific to the 1D or 2D versions end with the characters **1d** or **2d**. Initialization routines have names like **initfoo1d.f** or **initbar2d.f**.

The variables needed to run this program (ignoring for the moment the local variables in the lower subroutines) are stored in three common blocks. **global.h** contains all the "global" quantities needed throughout the program, such as the current time, Courant parameter, and polytropic index γ. **Sweep.h** contains all the 1D arrays needed for the one dimensional sweeps, and is included in all subroutines below **sweepx.f** in the calling tree. **sweepglobal.h** contains a parameter `maxsweep` which governs the size of scratch arrays in the sweep, plus declarations for the `sweepglobals` common block, which contains variables used for debugging. **zone1d.h** and **zone2d.h** contain the arrays for the fluid variables (density, pressure, and velocities) and grid coordinates.

The input used to run the program is written in a namelist file, **indat**.

Our convention for output files is that each machine-readable "dump" of the program's state is written into files with names of the form **dump1d.00001**, where the number is the current cycle. These dumpfiles are used to save the complete program state so that the computation can be restarted at a later date. Human-readable output is periodically written to a single file named **print1d**, and a record of useful parameters and events are written into a single file named **history1d**. VH-1 will avoid overwriting existing files by adding the letter "A" to a filename as many times as needed to create a unique filename (see **makeunique**).

A schematic of the program is shown in Figure 1. The program naturally separates into two components: the main driver and the 1D hydro sweep. The main driver performs all of the peripheral activities such as input, output, and timestep control, while the hydro is computed within the 1D hydro sweeps. For multidimensional problems, this is done by calling the 1D hydro sweep in alternating directions. The following two subsections will describe these two primary components of VH-1.

3.1 One-dimensional hydrodynamics

In this subsection we will describe how VH-1 performs a 1D hydro step (see **sweep1d**, **sweepx2d**, or **sweepy2d**). We will defer discussion of the peripheral routines, such as **equalgrd**, initialization routines, outputs, and drivers to the next subsection, and focus on the actual hydrodynamics calculation in this subsection. The convention in this program is to use the index `j` for one-dimensional situations, and `(i,j)` or `(i,j,k)` for multidimensional arrays, so the following discussion will only use index `j`. In addition, on a coordinate axis, the smaller coordinate is to the left, i.e. if x coordinate runs from 0 to 1, then the left boundary is at 0 and the right boundary is at 1.

3.1.1 sweepx2d

The hydrodynamics calculation begins by copying the variables from one slice of the global 2D variables (i.e. zrho) into the sweep 1D work arrays (e.g. rho). The 1D equivalent to **sweepx2d**, **sweep1d**, also copies from 1D global arrays into 1D work arrays for the sweep. This is unnecessary, but wastes a tiny amount of CPU time and simplifies the code quite a bit.

The sweep routines understand up to 2 quantities which are "transverse velocities": `vt1` and `vt2`. These velocities are advected along with the density, and are only used to compute the total energy in a zone. A one-dimensional computation would use no transverse velocities, while 2D and 3D would use 1 and 2, respectively. For example, in a 2D sweep along the x direction, v_y would be a transverse velocity.

The sweepNd routine subsequently calls several subroutines. The first of these is a call to **bc_fundamental**, which creates two ghost zones on either end of the 1D array. These ghost zones are needed to construct parabolas at the edge of the grid. The values in the ghost zones depend on our boundary conditions. The file **boundary.f** contains the routine **bc_fundamental**.

Next, we compute some geometric constants for our parabolic interpolation, using the routine **paraset**, in the file **parabola.f**.

Next, a call to **fictforces** computes fictitious forces, and a call to **gravity** computes the real forces for this sweep A few sample implementations of these functions can be found in **fictforces.f**, **gravitynone.f**, **gravityself.f**, and so on.

A call to **ppm** performs the entire Lagrangian hydrodynamics step (see following sections), updating the 1D array with new fluid variable values and coordinates. Subroutines used by **ppm** include **flatten, parabola, states, riemann,** and **evolve**.

The ghost zones are reset according to the new, updated variables with another call to **bc_fundamental** in **boundary.f**. The routine **remap** does just what its name implies, and remaps the new fluid quantities from the new, evolved Lagrangian grid given by the `xa`'s, to the old, Eulerian grid, given by the `xa0`'s.

Finally, the modified 1D arrays are copied back into the multidimensional arrays.

sweepy2d is similar to **sweepx2d** except that it loops over a different index and uses the appropriate coordinates for the y direction (e.g., `xa` = `zya`).

3.1.2 boundary.f

This file contains several subroutines that set boundary conditions. **bc_fundamental** is called twice per sweep to set up the fundamental variables and coordinates in the two ghost zones on either end of a 1D sweep. These values are needed to compute parabolas at the left and right edges of the grid. **bc_average** is used by **ppm** to apply boundary conditions to the zone-of-influence averages. **bc_parabola** applies boundary conditions to the parabolae so that the remap can move material on and off the grid.

3.1.3 fictforces

This subroutine is called from **evolve** to calculate zone-centered and face-centered fictitious forces at the beginning and end of the timestep for use in the finite difference equations. It uses the values of `ngeom` to determine which loop to use. Fictitious forces appear whenever the grid is not cartiesian; e.g. in a 2D problem in $(r,\ \theta)$ coordinates, then the fictitious forces would be centrifugal and Coriolis forces generated by angular momentum.

3.1.4 gravity

This subroutine is called from **evolve** to calculate zone-centered and face-centered gravitational forces at the beginning and end of the timestep for use in the finite difference equations. It uses the values of `ngeom` to determine which loop to use. If no gravity (or other external force) is used in the problem, use **gravitynone.f**. If you are doing a 2D problem with self-gravity, the proper way to deal with these forces would be to compute gravity before you entered the sweep, and have **gravity** merely copy the necessary information into the appropriate arrays.

3.1.5 ppm

This routine performs the Lagrangian evolution of the fundamental variables. First, **flatten** is called to get the array of flattening coefficients, `flat`. **parabola** is called for each fluid variable needed for the hydro calculation: pressure, density, and velocity. These parabolic profiles (specified by sets of variables like `pl`, `p6`, and `dp`) are then passed to **states** to compute the input states for the Riemann problem to be solved at each zone interface. **states** returns the value of these variables averaged over the domain of influence of the characteristics, giving the left and right states at each zone interface (e.g., `plavg`, `pravg`). `plavg(j)` is the average pressure over the domain of influence on the left side of the zone boundary separating the $j-1$ zone and the jth zone. `pravg(j)` is the average pressure over the domain of influence on the right side of the same zone boundary.

Next, **bc_average** is called to compute the averages on the edges of the vector by applying the boundary conditions.

These input states are passed to **riemann** which calculates the time averaged value of the pressure and velocity at each zone face, from `jmin` to `jmax+1`: `umid` and `pmid`.

Finally, **evolve** is called to update the fluid variables using the time-averaged quantities `umid` and `pmid`.

3.1.6 flatten

Post-shock oscillations are generated by a variation of the width of the shock structure as a shock moves across the grid. To reduce these oscillations, PPM generally uses a mixture of 3rd order and 1st order zone profiles at shocks. This broadens the shock and reduces the variation. For more discussion, see § 5.

The subroutine **flatten** loops over the 1D arrays and looks for strong, narrow shocks. If it finds one, it sets the variable **flat** to a value (between 0 and 1) that will be used to flatten out the parabolic profiles in that zone. If the shock is very strong, `flat` $= 1$, and the routine **parabola** will use a first-order (flat) profile for that variable. Weaker shocks will have a smaller value of `flat` corresponding to a mixture of first and third order. If no shocks are present in (or near) a given zone, `flat`$= 0$ and the parabolic profile derived in **parabola** will be used as-is.

The variable `shock` is positive if a shock is in the zone, or negative if there is no shock, based on the pressure gradient. If the zone is undergoing compression, as determined by the sign of the velocity gradient (which distinguishes shocks from rarefaction waves), then the steepness of the pressure gradient is stored in `steep`. Finally, the `flat` parameter is set to the highest value of `steep` in that zone and the two neighboring zones, ie, a zone is flattened if a strong shock is in that zone or only one zone away. The parameters used in this routine are chosen based on extensive testing and the advice of C&Wand W.

3.1.7 paraset

This routine computes some geometric constants used by the **parabola** routine. These values remain the same for a given geometry, and can be reused for different fluid variables at the same time. In addition, the values computed at the start of the sweep can be used at the start of succeeding sweeps because the grid is always remapped back to the same location.

The array `para` is used to store these constants. It is dimensioned to the size (9, `maxsweep`). The planar version of **paraset** only uses the first five. For a derivation of these coefficients, see C&W.

The last four vectors are used in the more detailed parabolic interpolation for dealing with coordinate singularities. This curvilinear version is considerably more complex and slower, and is only advantageous near coordinate singularities. For that reason, these corrections can be turned off by setting the variable `axiscorrection` = 0 (see § 3.3).

3.1.8 parabola

This subroutine calculates the parabolic profiles of the variable `a` and returns the coefficients `deltaa`, `a6`, and `al` that describe this parabola:

$$a(\xi) = al + x(\delta a + a6(1-x))$$

$$x = \frac{\xi - \xi_j}{\xi_{j+1} - \xi_j}, \qquad 0 < x < 1.$$

In the one case of remapping the total energy, the values of `al` and `deltaa` are already known, and we only need calculate `a6` and apply the monotonicity constraints. Otherwise, we calculate the `ar`'s (note that `al(j+1) = ar(j)`) according to C&W. In calculating the `ar`'s we apply a simpler limitation than that given by eq. (1.8) in C&W. We require `ar(j)` to be in the range of `a(j)` and `a(j+1)`. This is important in the vicinity of shocks where a steep pressure profile might lead to an overshoot in `ar(j)`, giving a negative pressure just in front of the shock. While our constraint will not steepen sharp gradients as would Eq. (1.8) of C&W, it can be applied to a correct curvilinear interpolation scheme as described in Blondin & Lufkin (1993).

If `iflat = 1` (which is true when we are making parabolae for **ppm**, but not for **remap**) then parabola uses the passed array of flattening coefficients, `flat`, to smear out steep shocks.

Finally, **parabola** applies the monotonicity constraints. These constraints are very important, because they damp out oscillations on a zone-to-zone length scale, which cannot be accurately represented by the algorithm and must be eliminated. We apply two different constraints: If the zone average value `a` is a local extremum, flatten the profile in that zone: `ar(j)` $\rightarrow$ `a(j)`. If the derived parabola introduces a new extremum, i.e., if the parabola exceeds `ar` or `al`, then reset `ar` or `al` so that the slope $\rightarrow 0$ at the side closest to where the parabola exceeded the boundary value (see C&W). Monotonicity constraints help prevent noise on one-zone length scales, which cannot be resolved by the algorithm, from growing.

3.1.9 states

This subroutine takes the parabolic profiles of pressure, density, and velocity given by (`al,da,a6`) and computes the left and right states of each variable for input to the Riemann solver. First, it calculates the fraction of the zone to be averaged using the average sound speed in that zone times the timestep, divided by the width of the zone. A factor of 1/2 is included in the definition of `Cdtdx` to save multiplications later on. If the coordinate is angular, the zone width must be multiplied by the variable `radius` to get a physical dimension.

The averaged states are then computed using the formulae in Eq. (1.12) of C&W, with the addition of force corrections to the velocity states. These corrections (of $g\Delta t/2$) represent the time-averaged acceleration of the fluid to first order only. If geometry corrections are required (currently implemented for radial, cylindrical and spherical), the velocity states would be altered according to eq. (2.9) of C&W, but this version of the program does not have these corrections. Note that other corrections may be desired at this point to account for other source terms, including forces and energy sinks and sources.

3.1.10 riemann

The Riemann solver uses the left and right input states at each zone boundary and solves the Riemann problem to obtain the velocity and pressure at the (Lagrangian) zone boundary. You can think of the Riemann problem as a tiny Sod shock tube, starting with a discontinuity in both pressure and density at the zone boundary. The Lagrangian zone boundary will move with the contact (density) discontinuity. The Riemann problem has a nonlinear solution and is generally solved by using some form of iteration. This particular subroutine can be replaced by other Riemann solvers that may be more accurate or more efficient.

The nonlinear equations to be solved at each zone interface look like (C&W, van Leer 1979):

$$\frac{\bar{p} - p_L}{W_L} + (\bar{u} - u_L) = 0$$

$$W_L^2 = (\gamma * p_L * \rho_L)\left(1 + \frac{\gamma + 1}{2\gamma}(\bar{p}/p_L - 1)\right)$$

$$\frac{\bar{p} - p_R}{W_R} - (\bar{u} - u_R) = 0$$

$$W_R^2 = (\gamma * p_R * \rho_R)\left(1 + \frac{\gamma + 1}{2\gamma}(\bar{p}/p_R - 1)\right).$$

where $\bar{p}$ and $\bar{u}$ are the pressure and velocity of the contact (density) discontinuity, and p_L, p_R, ρ_L, and ρ_R are the initial pressure and density on the left and right sides. In this situation the initial states come from averages over the zone of influence. We have assumed that the equation for the Lagrangian wave speed of a shock is a sufficiently valid approximation for a rarefaction wave so that only one wave speed need be calculated. To start the iteration, the wave speeds W_L, W_R are approximated by the adiabatic Lagrangian sound speeds, i.e., $C_L^2 = \gamma\ p_L/\rho_L$, and the above equations are solved for $\overline{p}$. In a smooth flow this is all that is required. In the vicinity of discontinuities, we must iterate to converge on the correct value of $\overline{p}$. First, we find the current guesses for $\bar{u}$ based on the current guess for $\overline{p}$ and the left and right wave equations above. Let us call these u_L^* and u_R^*. We then use the slope of the left and right adiabat (Z_L, Z_R) in the u—p plane at these two points ($\bar{p}$,u^*_L) and ($\bar{p}$,u^*_R),

$$Z_L = \frac{2 * W_L^2}{W_L^2 \ + \ C_L^2} \ W_L$$

$$Z_R = \frac{2 * W_R^2}{W_R^2 \ + \ C_R^2} \ W_R$$

to extrapolate the tangents to the point of intersection, from which we find our next guess for $\bar{p}$:

$$\overline{p}^{n+1} = \overline{p}^n - \frac{Z_L * Z_R}{Z_L \ + \ Z_R}(u_R \ - \ u_L).$$

This new guess can then be used to start another iteration (compute wave speeds, solve for $\overline{p}$) until the desired accuracy is obtained. In practice, we always iterate five times on all zones, although more efficient algorithms are certainly possible. Once a converged value for $\overline{p}$ has been found, the value of $\overline{u}$ can be found by combining the above equations into an expression for $\overline{u}$:

$$\overline{u} = \frac{W_L \ u_L + W_R \ u_R - (p_R - p_L)}{W_L + W_R}.$$

3.1.11 evolve

This subroutine uses `umid` and `pmid`, as determined by the Riemann solver, to solve the finite difference form of the Lagrangian equations of hydrodynamics given by Eq. (2.10) in C&W. The first step is to evolve the locations of the grid faces, `xa`, which move at the velocity `umid` over the whole timestep. Next, the new zone volumes are computed according to the geometry flag passed from **ppm**. At the same time the average area of the zone face, `amid`, is computed to account for non-planar geometry in the hydrodynamic equations.

Evolution of the density is particularly trivial. Because the total mass in the zone remains constant in a Lagrangian calculation, the change in the zone-averaged density is inversely proportional to the change in zone volume. The velocity is updated according to the pressure gradient and the forces, both fictitious and external. The total energy is updated according to the enthalpy gradient and the external forces only. The pressure is found by subtracting the kinetic energy from the new total energy.

Note that fictitious forces do not change the total energy, but rather redistribute the momentum between the different coordinate directions. If included in the energy equation, the change due to fictitious forces in one direction should exactly cancel the change found in the orthogonal direction.

3.1.12 remap

This subroutine remaps the new fluid variables from the evolved Lagrangian grid back onto the original, fixed Eulerian grid. The same subroutines used in **ppm** are used to compute parabolic profiles of the conserved quantities, and the total amount of those variables found in overlapping subshells are returned to the Eulerian zone. Note, however, that no flattening at shocks is used in the remap step.

The first order of business is to compute the new mass elements `dmu` and the width of the overlapping subshells, `delta`. Note that `delta` can be positive or negative, depending on the sign of the velocity at the zone interface. As in the hydro step, we call **paraset** to get the parabolic coefficients, `paracoff`. Since the new Lagrangian coordinates are always different, we must make this call every time. When remapping, no flattening is desired, so we pass 0 for the flattening flag along with a dummy array in place of `flat`. Next we make individual calls to **parabola** to get the profiles of the variables to be remapped. The variables in the ghost zones have already been set before the Eulerian evolution step.

There are two possible forms of remap: the density and velocity are always remapped, but we can choose to remap either the total energy or the internal energy. The value of the variable `mremap` is used to pick one of the two methods. The reason that we implement both methods is that each has its weak points. Remapping the total energy (`mremap = 0`) means that the internal energy must be computed by subtracting the kinetic energy, which is formed from the separately remapped density and velocities, from the total energy. Since each of these terms has independent errors, the internal energy might end up negative if the kinetic energy dominates the flow. Remapping the internal energy (`mremap = 1`), on the other hand, will not strictly conserve total energy.

As discussed in C&W, a "fix" must be implemented for the remap of the total energy to try to account for the fact that total energy contains the square of the velocity. A parabola fit to the total energy will not match the square of the parabola fit to the velocity; i.e., although we have $E = P/\rho + \frac{1}{2}u^2$ for the zone averages, the same will not be true for each point on the respective parabolic interpolations. To compensate for this possible discrepancy, C&Wsuggest using the left and right values of the velocity and internal energy to construct the total energy profile. In this way the different parabolae will at least satisfy the above energy equation at the zone boundaries, which is where the calculation of the subfluxes originate. (In the limit of small grid motion, we will use only the quantities very near the zone interfaces.) Loop 100 does the construction of `el` and `de` as described above, and the subsequent call to **parabola** passes a flag to tell **parabola** that the left and right values are already determined.

Next, a new boundary condition problem raises its head: if there is any motion of the ghost zones onto the grid at the boundaries of the grid, we need to know the variable profiles in the first ghost zone, `jmin-1` and `jmax+1`. We set these values based by calling the routine **bc_parabola**.

We remap the conserved quantities in two steps: First, we compute the integral of the conserved quantities in the overlapping subshell using the parabolic profiles calculated above, and second, we update the zone averages by adding and subtracting the appropriate quantities in the subshells.

The total quantity of a given variable in a subshell, or "flux," is stored in an array; `fluxr` (total mass), `fluxu`, `fluxv1`, `fluxv2` (velocities), and `fluxe` (total energy) equal to the integral of that variable over the volume of the subshell. If `delta` > 0 the flux is positive and equal to the integral from the right side of zone $j-1$ back to the original zone boundary, `xa0(j)`. If `delta` < 0 the flux is negative and equal to the integral from the left side of zone j forward to the original zone boundary, `xa0(j)`. The advection is then just a matter of redistributing these subshell fluxes to the appropriate zone:

The quantities to be remapped are (1) the total mass, `fluxr`, (2) the momentum, `fluxr` $\times$ `fluxu`, (3) either the total energy, `fluxr` $\times$ `fluxe`, or the total internal energy, and (4) the transverse velocities `fluxv1` and `fluxv2`.

The remap routine in VH-1 is not capable of a completely general remap from any grid to any other grid. It is designed to be used to remap from a grid to another grid which has not moved any grid point more than one zone, i.e. one in which a single timestep of hydro constrained by a Courant condition has taken place.

3.2 The rest of the program

The main driver for VH-1 is **main1d.f**, **main2d.f**, or **main3d.f**. This program reads the input namelist file, sets up the grid (**equalgrd.f**) and initializes it (by calling some **initfooNd.f** file), prints output using (**prin.f** and **dumpNd.f**), constrains the timestep (**timestepNd.f**), calls the appropriate **sweep** routines, and other general managing tasks. In this section we will describe the driver and each of the peripheral subroutines.

3.2.1 mainNd.f

The input of data and general management is performed the same way for 1D, 2D, and 3D simulations. First some defaults are set, and the two main namelist files are read out of **indat**. The contents of these namelist files are described in §3.3. A quick check is performed to make sure the declared arrays are large enough to handle the problem. A history file is opened and a summary of the parameters of the run is written to it. **main2d.f** and **main3d.f** allow a previous run to be restarted, and if this is a continuation, the dump file is read. If the run is not a continuation, the appropriate init file is called. The initial timestep is determined by a call to **timestep**.

The main loop consists of calling the appropriate sweep subroutines and **dtcon**, incrementing loop variables such as `ncycle` and `time`, and performing dumps and human-readable prints based on the settings of the `ncycp, ncycd, timep,` and `timed` variables. In 2D runs the cycles are split into even or odd values of `ncycle` so that the order of sweep calls is XYYX (to approach second order accuracy). When the program has reached either the time or iteration limit, it takes a final dump and exits.

Whenever a time barrier, either the end of the run or a timed dump or print, is reached, the timestep is temporarily reduced to the timestep necessary to exactly reach the desired time. Since the timestep routine uses the previous timestep as a limitation to only allow the new timestep to increase slowly, we save the actual timestep in the variable `savedt`, and restore it before computing a new timestep. If we neglected this detail, the timestep could drop dramatically and take quite a while to recover if a dump-time was a very small time from the ending time of some timestep.

3.2.2 equalgrd

This utility routine generates an equally spaced grid, and is used to build the coordinate axes. It does not generate anything for ghost zones, because ghost zones only exist inside the sweep routines.

3.2.3 initoned, inittwod, initthreed

This family of subroutines initializes the desired problem. It must generate values for all the fundamental variables in all the grid cells, and may read a namelist out of **indat** to find parameters. Note that the variables must be averaged over the volume of the zone, which can be significantly different from the value at the center of the zone in some geometries and situations. Many of our examples do not practice such averaging, but you should beware unusual effects from using zone-center values.

3.2.4 timestep

This subroutine loops over the grid and calculates the limiting timestep based on the CFL (Courant-Friedrichs-Lewy) constraint using the parameter `courant`, set in the `fundamental` namelist (see § 3.3). For multidimensional grids with angular coordinates, angular coordinates must be multiplied by the radius to get physical dimensions, and hence physical times. The limiting time step is determined by

$$dt = \texttt{courant} * \max(\Delta x/u_x, \Delta y/u_y, \Delta/c_s),$$

where $\Delta = \max(\Delta x, \Delta y)$. This timestep can be much larger than the timestep used by a direct Eulerian program that must use the sum of the fluid velocity and the sound speed, $\Delta/(u_x + c_s)$. An additional constraint is implemented that does not allow the timestep to grow too quickly, in this case limiting the new timestep to 1.1 times the old timestep. If source terms are included in the calculation, they may impose additional constraints on the timestep. A value of `courant` of 0.7 or less is recommended to ensure stability. The 1D Sod shock tube test problem provides a good example of the error introduced at a shock front when a too large or too small a value of `courant` is used.

If the timestep drops more than 50% in a single timestep, a warning is printed. This often indicates a problem, such as a timestep dropping to zero, or a situation such as beginning of the Sod Shock Tube, where large velocities are generated in one timestep and so the initial timestep was not small enough.

As a special case, a negative input timestep will result in no timestep increase limitation, and no warning. This is used to initialize the timestep the first time through.

3.2.5 printNd.f

This subroutine should be altered to output the fluid variables in a format appropriate for your individual use. We use this output for debugging and simple graphics.

3.2.6 dumpNd.f

This subroutine writes out an unformatted but sophisticated dump of the state of the entire program, sufficient to restart the computation at the time of dump. Version information is written into the file, so that it is possible to restore old versions into new programs. We also have IDL and FORTRAN programs that read these dumps to make graphical output. Finally, these dumps can be converted to ASCII to be transported over the network between dissimilar machines.

3.3 Running VH-1

VH-1 must be customized before it can run on a new problem. Normally this involves (1) including the proper input data in the file **indat**, (2) setting up the initialization subroutine, **initfooNd.f**, and (3) possibly altering the output subroutines to write out the data in the user's desired format. This is a minimal list of required changes, and more difficult problems may involve more changes to the program for proper results. Such changes might involve additional body forces, switching to Lagrangian mode in 1D, adding new boundary conditions or incorporating additional source terms. Often the user will add parameters to **global.h** for use on a specific problem.

The file **initfooNd.f** will contain the code needed to set up the initial conditions. This code is responsible for setting the initial values of the `rho, p,` and the velocities. If desired, this routine may read a namelist out of the **indat** file.

The information needed for a specific run is written in the file **indat**. The user is encouraged to adapt this file to suit their needs. **indat** contains a series of namelists, most of which have default values and need not be changed.

The `fluff` namelist contains the information that will be reset every time a given run is started or restarted from a dump file. `fluff` contains the following variables:

```
&fluff
  dumpfile    = '( ''dump2d.'', i5.5 )',
  printfile   = 'print2d',
  historyfile = 'history2d',
  ncycend     = 2000,
  nprint      = 10000,
  ndump       = 20000,
  tprint      = 1.e7,
  tdump       = 2.e7,
  endtime     = 2.4e8
&end
```

The file parameters are the names that will be used for the dump, print, and history files. The dumpfile specification can either be a constant string (e.g. `'foo'`) or a format expression which will be used with the cycle number. The variables beginning with `n` are expressed in cycles, and tell when prints, dumps, or the end of the run will take place. The variables beginning with `t` specify the same things in units of time. To generate a movie of a run, for example, you will often want to dump at specific times. To make a run take a given amount of CPU time, on the other hand, you want to run to a given number of cycles. Notice in the example above that `ncycend` < `ndump`, so we are in fact dumping only at a specific time, and never at a specific number of cycles.

The `fundamental` namelist contains information that should remain the same through an entire run, even if it is restarted. The `fundamental` namelist contains:

```
&fundamental
  iorder   = 3,
```

```
   gam      = 1.4,
   nzones   = 100,
   xmin     = 0.0,
   xmax     = 100.0,
   ngeom    = 0,
   nleftbc  = 0,
   nrightbc = 0,
   courant  = 0.7,
   tinitial = 1.0
 &end
```

The order, 1 or 3, determines if the Godunov or PPM algorithm is used. `gam` is the polytropic index gamma. `nzones` specifies the number of physical zones. In 2D, there are `nxzones` and `nyzones`. `xmin` and `xmax` give the physical dimensions of the grid. `ngeom` gives the geometry. The following geometries exist in the current version:

```
0 = cartesian
1 = cylindrical radial
2 = spherical radial
3 = cylindrical theta
4 = spherical theta
5 = spherical phi
```

In order to compute a 2D spherical polar geometry, for example, `nxgeom = 1` and `nygeom = 3` would be appropriate.

The left and right boundary conditions for each dimension are next. Recall that we define "left" to be the boundary at the smaller coordinate position. The current possibilities are:

```
0 = reflecting
1 = inflow/outflow
2 = fixed inflow
```

The Courant condition should be around 0.7. `tinitial` is a constant which is multiplied into the initial timestep. It is used for situations where the gas on the grid is motionless at the start of a computation, but will quickly end up moving supersonically. In this case, it is much better to use `tinitial` to reduce the intial timestep than to let a cycle be done with too large a timestep, which could result in the courant condition being violated.

The `sweep_globals` common contains information that is used by the sweep. This is mostly debugging information, but a few physical constants are here as well. `sweep_globals` contains:

```
 &sweep_globals
   scalar        = 0,
   debugremapx   = 0,
```

```
   debugremapp    = 0,
   debugcourant   = 0,
   debugevolvep   = 0,
   debugevolverho = 0,
   mremap         = 0,
   minimump       = 0.,
   minimumrho     = 0.,
   maximumv       = 0.,
   smallp         = 1.d-40,
   axiscorrection = 1
 &end
```

The debug variables should be set to 1 to turn on the appropriate debugging tests. `mremap` should be 0 for a total energy remap, and 1 for an internal energy remap. The `minimum` and `maximum` variables are used to make sure that various quantities have legal values, namely that the velocity should not exceed the speed of light, and pressure and density should be positive. `smallp` should be set to a pressure smaller than the smallest pressure expected to be physically meaningful, and is used in **riemann**. Finally, `axiscorrection` determines whether or not the coordinate corrections are applied to the interpolated parabolas, as explained in Blondin & Lufkin (1993). 1 means apply them, 0 means do not.

The `units` common contains information that is used to convert from code units to physical units. For example, if you use $G = c = 1$ units within the code but would like cgs units in your print files, you can enter appropriate conversion factors here.

The program will test to make sure that the various parameters are consistent. For example, the global array declarations must be large enough for the number of zones.

Other forms of customization may be necessary for other problems. Occasionally boundary conditions are dependent upon the position within the computational grid. This problem arises in two of the test problems provided with VH-1: the oblique shock and the Mach 3 wind tunnel with step. In this case, **sweepx2d.f** and **sweepy2d.f** are altered to set the value of the boundary condition flag as a function of grid location.

3.4 The Godunov Code

In addition to the PPM implementation, this program also includes a first-order Godunov implementation, which represents the variables as constant values within zones (as opposed to PPM's parabolae), but still uses a Riemann-solving technique for evolution. The Godunov subroutines can be found in files such as `remap1.f`, while the PPM subroutines are in files such as `remap3.f`. The Godunov version is useful for code comparisons such as Woodward and Colella (1984).

4 Test Problems

Several test problems are included with the distributed version of VH-1. These problems serve two specific purposes. First, they provide the new user with immediate, working examples with which they can begin to learn how to operate the program. It is highly recommended that the new user begin by running all of the enclosed test problems to ensure that the current program is working properly for a variety of boundary conditions and geometries. The second purpose of these sample problems is to provide a series of tests that can be rerun each time the program is altered. In addition to the problems presented here, it will prove useful to add problems that are similar to the types of flow problems the user intends to work on with VH-1. In general, the closer the test problem is to the actual working problem, the less chance for errors to arise through "insignificant" changes in the program.

4.1 Sod Shock Tube

The Sod shock tube has become a standard test problem in numerical hydrodynamics (XXX ref.). The initial conditions are very simple: A contact discontinuity separates two regions with different pressures and densities. In our standard case, the density and pressure on the left are equal to 1, and the density on the right side of the contact is 0.125 and the pressure is 0.1. As the evolution begins, a shock propagates to the right while a rarefaction wave travels to the left. In the standard case these waves are relatively weak, and most hydrodynamics programs produce good results. A more demanding test is to increase the density and pressure ratios by an order of magnitude.

Figure 2 shows the flow variables for the standard Sod shock tube problem at $t = 23.0$, along with the analytic solution drawn as a solid line. The shock and contact discontinuity are both very sharp.

This problem can be expanded to multiple dimensions by placing the discontinuity across a 2D or 3D grid. For the 2D test, we have positioned the contact at a 45^o angle so that the shock and rarefaction wave propagate diagonally across the grid. Figure 3 shows the density structure of this 2D shock tube problem where the initial high density gas was in the lower left corner. The image is very close to symmetric about the diagonal, but some small differences can be seen. This image is at $t = 0.871$. The shock wave has reflected off the far walls and is converging back towards the original corner. The contact discontinuity is beginning to ripple after the recent passage of the reflected shock. This is the Richtmyer-Meshkov instability.

4.2 1D Advection of a step in density

This test is a very simple yet interesting test in which the pressure is equal everywhere, but a step-function in density moves steadily across the grid. Because the program contains numerical dissipation, the step will gradually be smeared out. Varying the Courant number and numerical technique between PPM and Godunov is interesting.

4.3 Double Mach Reflection

This problem is taken from Woodward & Colella (1984), and is used extensively as a test problem because of the complicated structures generated and the availability of experimental solutions to compare with numerical results. A shock is set at an angle of 30^o to the grid, and is allowed to reflect off the lower boundary. The shock is anchored at the lower left corner by applying outflow boundary conditions for the first few zones on the lower boundary (see Woodward & Colella (1984) for details). Figure 4 shows the density structure at $t = 0.2$. This figure can be compared with Figure 9 of Woodward and Colella.

4.4 Mach 3 Wind Tunnel with a Step

This test problem is again taken from Woodward & Colella (1984). In this case a uniform grid is initialized with planar supersonic flow from the left. A step is put into the flow downstream from the left boundary by adjusting the location of the lower reflecting boundary with the **sweepx.f** and **sweepy.f** subroutines. The simulation is stopped at a time $t = 4.0$, and the results are shown in Figure 5. This simulation can also be compared with the results of Woodward and Colella (see their figure 7). We have not implemented the corrections to the grid zones around the step corner as described in Woodward and Colella. This affects the flow solution along the surface of the step, and as such this region differs from the solutions published in Woodward and Colella. We have run this problem with such corrections and obtained good results, but have not incorporated the added complexity into the distributed test version. The contact discontinuity along the top of the flow ($y \sim 0.8$) is Kelvin-Helmholtz unstable, and slight wiggles can be seen even in this low resolution run. The seed perturbations for this instability are attributed to postshock noise (see § 5).

4.5 Supersonic Flow Past a Cylinder/Sphere

We have included a generic test problem of a plane parallel flow past a rigid sphere or cylinder (for `ngeom` = 2 or 1, respectively). This test problem verifies the geometry dependent terms and fictitious forces by evolving a plane parallel flow on a curvilinear grid. The inner radial boundary condition is reflecting (the solid sphere/cylinder), and the outer radial boundary is set for free inflow/outflow. The angular boundaries ($\theta = 0, \pi$) are set for reflecting conditions. The grid is initialized with a uniform flow at Mach number `umach` set in **init.f**. The resulting simulations compare favorably with shadow-graph images of similar experiments in fluid flow. In particular, the presence of the slip line and a shock reflecting off the axis behind the sphere agree well with supersonic flow experiments. Figure 6 shows a Mach 3 flow past a sphere. Similar results are obtained for flow past a cylinder.

4.6 Hawley-Zabusky 2D Angled Shock Tube

This 2-dimensional problem consists of a shock hitting a contact discontinuity at an angle. Net vorticity is generated by such a situation, and the integrated vorticity can be computed given a few assumptions (ref HZ.) It is interesting to vary the resolution and compare the integrated vorticity value (or is it?)

4.7 1D Hydrostatic Equilibrium in a Polytropic Star

One easy technique for modeling a star is to use a polytropic equation of state ($P = \rho^\gamma$). This test sets up a 1D star in hydrostatic equilibrium, and watches it sit still. Actually, because the initialization routines are not exact, the star will oscillate a bit. The interesting question then becomes: is the star in equilibrium on the sorts of timescales we are interested in? For example, a supernova computation needs a 10 $M_\odot$ star to be steady on a timescale of 1 hour.

4.8 1D Supernova in a power-law star

This problem is computed numerically by Mueller *et. al.* (1989) and Lindahl (1992), and an analytic discussion is provided in Chevalier (1976). We find that if we use flattening at the shock, we have an evacuated center, which is predicted by theory. The program Prometheus, used by Mueller *et. al.*, does not have an evacuated center.

4.9 1D Supernova in a polytropic star

This test problem is also computed numerically in Mueller *et. al.* (1989) and Lindahl (1992).

4.10 Geometry test problems

We have also included some test problems which illustrate the effects of the geometric corrections discussed in Blondin & Lufkin (1993).

5 Dissipation

PPM is a very nice method for compressible hydrodynamics, but it does have an Achilles heel. When strong shocks propagate slowly across the numerical grid, low-frequency noise is generated in the post-shock region. This noise is a consequence of the narrow shock structure generated by PPM. Because the shock is resolved into only one or two zones, the exact profile of the shock is dependent on the relative location of this narrow profile with respect to the zone faces. When the shock front crosses a zone face, the profile necessarily changes. If the shock is moving relatively fast with respect to the grid, these changes result in high frequency noise that is quickly damped. Slowly-moving shocks, on the other hand, produce low frequency noise that the highly accurate PPM algorithm is able to accurately follow without much dissipation. By page count, C&Wspend 1/3 of their paper discussing dissipation mechanisms to try and minimize this problem!

This problem is illustrated in Figure 7, where we show the post shock structure of a Mach 10 shock moving at 1% of the shock speed with respect to the numerical grid. Intuitively one could imaging trying to suppress this instability by either (a) broadening the shock structure so that it does not change appreciably while crossing the grid, (b) increasing the relative velocity of the shock so that the generated noise is of sufficiently high frequency that it is effectively damped, or (c) adding some sort of artificial dissipation at shock fronts to dampen any high frequency component. These are in fact the three options suggested by C&W: flattening, grid wiggling, and artificial viscosity. The first of these options, shock flattening, has proven to be simple, robust, and unobtrusive. The second of these, grid wiggling, is found to be beneficial in some circumstances, but can be rather cumbersome.

The last option, artificial viscosity, has not been shown to work with VH-1, despite the fact that it is beneficial in many other versions of PPM. This remains a mystery to us. We will discuss these three techniques below, but we have only implemented the first of these options in the version of VH-1 we are distributing.

5.1 Shock Flattening

This technique involves a reshaping of the interpolated parabolae in the vicinity of shock waves so that the shock is effectively broadened over a couple zones. This is accomplished in two steps, (1) the calculation of a flattening parameter based on the presence and strength of a shock (for VH-1 this is done in **flaten.f**), and (2) the mixing of the interpolated parabola and the zone average (for VH-1 this is done in **parabola.f**). In the extreme limit of everything being flattened, this method would transform to the original Godunov scheme, using only zone averages to compute the input states for the Riemann problem. In Figure 7 we also show the same problem computed with `flat(j)` = 1 in both the hydro and in the remap step, i.e., a Godunov scheme. The postshock oscillations are indeed removed, but the shock is now spread out over many zones (cf. C&W, who find some noise remains with complete flattening).

C&Wdescribe several stages of complexity in computing the flattening coefficient. We have found that the more complex methods do not show significantly better results than the simplest method in our slowly moving strong shock problem. We have therefore included only this simple approach. Figure 7 also shows the shock structure when flattening is applied as it is implemented in the release version of VH-1. While there is still some postshock noise, it is of substantially smaller amplitude than the noise in the unflattened solution.

One variation that we have not included is detection of a shock transverse to the current sweep. This variation could conceivably make a difference in cases where a strong shock is sitting nearly parallel (but not exactly parallel) to the numerical grid. In this case the postshock structure is flattened perpendicular to the shock front but not along the direction of the shock front. Instituting a multidimensional flattening routine is much more cumbersome than in 1D, and we have not found the results sufficiently encouraging to warrant such an addition to the standard program. In these difficult cases we find that grid wiggling provides a more effective suppression of post-shock noise.

5.2 Grid Wiggling

The easiest way to avoid this whole problem is to only work on problems where the shocks are rapidly moving across the numerical grid. Unfortunately, it is often the case that the interesting thing one is studying is associated with the shock, and it is desirable to have the shock in one location on the grid throughout the simulation. If you cannot move the shock with respect to the grid, move the grid with respect to the shock. This is the essence of the grid wiggling technique described in C&W. At each timestep one can move the grid back and forth, effectively smearing out the steep shock profile over more than one zone. One can imagine going about this in several ways:

(1) As in C&W, wiggle the grid at each zone location according to the amount of dissipation

needed and then wiggle back to the original grid on the next timestep.

(2) Wiggle the entire grid a fixed fraction of a zone each timestep, alternating directions of the wiggle each time.

(3) Wiggle the grid on the remap step and then re-remap the variables back onto the original grid at the same timestep.

In more than one dimension the grid must be orthogonal inbetween timesteps, so that if option (1) were used there must be two X sweeps (wiggle forward, wiggle backward) followed by two Y sweeps. In practice this is done anyway to minimize cross-term errors in the directional splitting. We find this technique to be most effective when there is a strong shock almost parallel to the zone gridding. In this case small changes in the shock/zone orientation from row to row can produce large differences in the postshock entropy, generating substantial postshock noise along the inside surface of the shock. If the grid is wiggled *along* the shock front (rather than across it), these postshock variations from zone to zone are effectively smoothed out.

Option (3) requires an extra remap step and hence is less efficient, but we include it here because there may be circumstances where it is the only feasible option.

While using a fixed-length wiggle will add dissipation where it is not needed (away from shocks), this added dissipation is not significantly more than what is already present in the algorithm (which is very small). We have not found any significant degradation when a fixed- length wiggle is used rather than a selective wiggle as described in C&W, although we have not performed any quantitative testing of these different possibilities.

5.3 Artificial Viscosity

Artificial viscosity has been used effectively to dampen short wavelength oscillations in several versions of PPM (e.g., programs written by Fryxell, Balsara, and Stone). We have not been able to formulate an artificial viscosity term that improves the performance of VH-1, and so have not included it with this program. The reason why VH-1 behaves in this manner compared with other implementations of PPM is not clear.

6 Boundary Conditions

Choosing good boundary conditions for a problem can sometimes be somewhat subtle. This section explains the types of boundary conditions commonly used in VH-1, and the implications of PPM to these boundary conditions.

6.1 Reflecting

One of the simplest boundary conditions is the reflecting boundary condition: Waves hitting such a boundary should reflect, and should conserve energy and mass. This goal is accomplished by choosing variables in the ghost zones — the zones on the other side of the boundary – which have equal density and pressure but opposite velocities to the zones approaching the boundary. In essence, we hit each approaching zone with an equal and opposite ghost zone. The resulting collision results in no boundary motion and lots of hot, compressed, stationary gas. This gas will then expand, and that provides the reflection off the boundary.

If there was just one ghost zone, we would implement such a boundary by setting the pressure and density in the ghost zone equal to the zone adjacent to the boundary, and the velocity equal to but opposite in sign to the velocity in the zone adjacent to the boundary. However, since PPM uses 2 zones to the left and right of each zone to compute the profile of the variables in each zone, it is also necessary to consider what we would like the parabolae to look like at the boundary: we want them to be symmetrical. This means that the pressure and density will have a zero derivative at the boundary. This also means that it is important that the zone widths be set up correctly. If the boundary is at the coordinate origin and you have a logarithmically-increasing grid, the ghost zones will be at negative coordinates, and the ghost zone farther away from the boundary should be wider than the ghost zone closer to the grid.

6.2 Fixed Inflow

In some problems, such as the advection of a step function of density (4.2) or the Hawley-Zabusky 2D angled shock tube (4.6), one boundary has totally specified behavior, i.e. inflow of material with a given density, pressure, and velocity. Such a boundary is easily implemented by giving the ghost zones the specified values. A wave propagating up to such a boundary will simply disappear from the grid, and not cause any long-term consequences.

Note that the implementation of fixed inflow in VH-1 is quite limited, suitable for the test problems mentioned in this manual.

6.3 Inflow/Outflow

Boundaries at which material should be able to simply "leave the grid" can be very troublesome in any hydrodynamics program. If the boundary is supersonic outflow, information from the boundary generally cannot propagate back onto the grid, so this one case is easy. Cases in which the outflow is subsonic, or at which you wish to have mass enter the grid are difficult.

Imagine a simple scheme in which we set the variables in the ghost zones equal to the zone on the boundary. Then you might expect that waves would be able to propagate off the grid. They can, but the boundary is not necessarily stable.

For the subsonic inflow case, a wave propagating upstream to the boundary will cause a change in the inflow which will persist at later times. Once the values in the boundary zone are changed, they will remain changed, because the material from the ghost zones will remember and reinforce the new value. For example, if the density is reduced in the boundary zone, the inflow will be reduced, tending to keep the density low. This is why we used fixed inflow boundary conditions for test problems X and X. For the outflow case, other sorts of problems appear. For example, consider a spherically-symmetric situation in which material is accelerating off the outer edge of the grid. The density should be falling as r^{-2}, but a straightforward boundary implementation would have the density in the ghost zones equal to the density in the final zone. Therefore, mass flowing off the grid would bounce a bit in the boundary zone, and waves might propagate back onto the grid.

There is no general solution to this problem. We recommend watching subsonic inflow/outflow boundaries carefully for reflected waves, or use of other forms of boundary conditions.

6.4 Other Boundary Conditions

If you implement other forms of boundary conditions, keep in mind that PPM's parabolic interpolation routine uses information from 2 adjacent zones in each direction to construct parabolae in a given zone. Also keep in mind that it is a parabolic interpolation, which simply cannot accurately fit some functions. For example, a variable which has an initial r^{-1} distribution is quite difficult to deal with at $r = 0$, because it is singular and does not have a zero first derivative. Even distributions which integrate to finite values in the first zone might not have zero first derivatives. A typical trick for dealing with such a boundary condition would be to add a turn-over in the innermost 10 zones, such that the distribution becomes flat in the center.

7 Optimization and Efficiency

The version of the code presented here is optimized for either scalar or vector computers. A simple but extensive rewrite is needed to make the code suitable for current parallel computers and compilers using Fortran-90 array notation.

On a scalar machine, there are few pitfalls. The guts of the code is the sweep, which takes a set of input vectors of length N, and performs around 1,000N floating point calculations. This entire computation will fit into the cache on most modern workstations; the maximum storage used is around 60 `maxsweep` words, which is a modest number of bytes for `maxsweep` = 512. Busting the on-chip cache on an Alpha costs us about a factor of 2 in performance.

Inside the sweep, there are 2 important routines: parabola and riemann. The Riemann solver is generally limited by the speed of floating square root and divide, so we have left it in a form which doesn't minimize the loads and stores. The parabola routine has a lot of logical operations, in addition to quite a few loads and stores. Most of these loads hit the cache. You may be able to speed this routine up a bit by fiddling with it.

Since the code has good locality of reference, most scalar machines perform very well on any size problem, as long as the global arrays fit into physical memory and the sweep arrays fit into cache. Many codes from supercomputers have unfortunate cache and TLB thrashing characteristics, or require large main memory bandwidth. VH-1 does not.

On a vector machine, all the inner loops in the sweep should vectorize. No use is made of scatter/gather. While the MFLOPS for some routines, namely parabola, look low, this is because of the large numbers of logical operations. It is possible to rewrite parabola to have higher MFLOPS, but this would not change the overall runtime significantly. The true measure of the speed of the code is the work performed, which we express in zone-cycles per second, which is the number of zones in a test problem multiplied by the number of cycles and divided by the CPU time consumed.

If you wish to run this code on a parallel machine, several strategies are possible. On a data-parallel machine, clever compiler could determine that the sweep is a pure 1-

dimensional operator, and parallelize around it. As far as we know, no such compiler exists. The most common workaround is to produce separate versions of the sweep for each direction. For example, for a 3D version of the code, 3 versions of the sweep would be needed. The sweep would work directly on the 3D global arrays. The first would change all the 1D loops in the sweep into 3D loops which work in the X direction. The second and third versions would work in the Y and Z directions, respectively. The drawback with this technique is that the 1D temporary vectors in sweep become 3D arrays, and the memory usage of the code balloons. This memory is being wasted, however, in the case where there are many virtual processors on each physical processor, which is the case with machines such as the CM-5 and Cray T3D. There is no easy work-around for this problem, short of a compiler which is capable of blocking the entire sweep.

Alternately, the arrays could be explicitly divided into cubes and the code could use messsage-passing. This would not be that difficult; the riemann solver is trivially parallel, so only the parabola, paraset, evolve, and remap routines would have to pass information. The communication should overlap quite well with computations.

For more modest levels of parallelism, on machines like the few-CPU boxes from Sun, SGI, and DEC, we suggest creating a bunch of threads to run one sweep each. A 3D problem 128^3, for example, has 128^2 sweeps which could be done in parallel. This method only works for shared-memory machines. You might also need to reorganize the global arrays for better locality. Another technique would be to use the allegedly highly-portable fork()/mmap() method.

I should also discuss Bruce's clever MasPar parallelization by sections, and my mildly-clever transpose f90 optimization.