Chapter 5

Classic hydrodynamics solvers

5.1 Basic approach

Armed with the knowledge on how to numerically advect functions over a grid, we can now start to build our first numerical hydrodynamics solvers. The knowledge of the structure of hyperbolic equations described in chapter 2 would lead us to want to diagonalize the system of equations locally and then apply the numerical advection schemes. This is in fact precisely what *approximate Riemann solvers* do (see Chapter **??**). But historically the equations were approached in a somewhat different manner, and those methods also have some advantages over the Riemann solvers of later chapters and for that reason are still heavily in use today. The main term in the Euler equations that tends to mix the eigenvectors of the Jacobian (and thus makes the problem not globally diagonalizable) is the ∇P term. If we extract this term from the left-hand-side of the momentum equation and put it to the right-hand-side, then the equation becomes an advection equation of momentum with advection velocity u. A similar trick can be done with the energy equation. The result is:

$$\begin{aligned} \partial_t \rho + \nabla \cdot (\rho \vec{u}) &= 0 && (5.1) \\ \partial_t (\rho \vec{u}) + \nabla \cdot (\rho \vec{u} \vec{u}) &= -\nabla P && (5.2) \\ \partial_t (\rho e_{\rm tot}) + \nabla \cdot (\rho e_{\rm tot} \vec{u}) &= -\nabla \cdot (P \vec{u}) && (5.3) \end{aligned}$$

The right-hand-side can be regarded as a source term. If we now regard $\vec{u}$ at any time step as a given quantity of the previous time step, then the left-hand-side of the above set of equations is already in *globally diagonal form* with three equal eigenvalues: $\lambda_{1,2,3} = u$ for the x-direction and $\lambda_{1,2,3} = v$ for the y-direction and $\lambda_{1,2,3} = w$ for the z-direction. The advection now has to be done in ρ, ρu, ρv, ρw, and ρe, all with the same advection velocity: these are five fully independent advection problems. This is what is sometimes called *consistent transport*.

There are a number of advantages of this approach over the full characteristic approach we'll cover in Chapter **??**. One of the main advantages of this approach is that the advection problem is much simpler than in the case when we diagonalize the full Jacobian with gas pressure. In fact, the above approach is very similar to Burger's equation, but with source terms. The source terms take care of the pressure forces and adiabatic compression/expansion of the gas. One can now apply a standard advection routine for the advection of conserved scalar functions (see Chapters 3 and 4) to all hydrodynamic conserved quantities ρ, ρu and $\rho e_{\rm tot}$, and at the end of each time step add the source terms to the momentum- and energy equation.

Apart from the fact that this is easy to do, another advantage of this is that one can apply algorithms of arbitrary high order. Of course, this high-order advection only applies to fluid mo-

tion and not to the sound waves, so the high-precision advection does not make the propagation of sound waves more accurate in this approach. But for very sub-sonic flows the propagation of sound waves is anyway of lesser interest, and the fluid motions are then anyway the most important flow patterns, which can thus be advected with any precision one likes by choosing the right advection algorithm. If the gradient of the pressure (i.e. the source term on the rhs of the momentum and energy equations) is also evaluated with a higher-order accurate derivative, then the entire algorithm can be made higher order.

A final advantage of this approach is that hydrostatic solutions of fluid flows with external body forces can be easily 'recognized' by the algorithm. This is because all forces are handled as source terms on the right-hand-side of the equations. In equilibrium all these forces cancel exactly, and will not produce numerical disturbances to the advection part of the equation. This approach is very well suited for problems that are near some hydrostatic equilibrium, such as planetary atmospheres or fluid motions in a rotating system.

5.2 A simple 1-D hydrodynamics algorithm

Let us jump right into making an algorithm, but let us do this for the moment for a simplified case: the problem of 1-D isothermal hydrodynamics. The two conserved quantities to advect are $q_1 = \rho$ and $q_2 = \rho u$. The pressure is $P = \rho c_s^2$ where c_s^2 is a global constant. The equations to solve are:

$$\partial_t q_1 + \partial_x(q_1 u) = 0 \tag{5.4}$$
$$\partial_t q_2 + \partial_x(q_2 u) = -\partial_x P \tag{5.5}$$

5.2.1 The algorithm

We will approach this using operator splitting:

1. First we do the advection without source term,

$$q_1^{n+1/2} = q_1^n - \Delta t \partial_x(q_1^n u) \tag{5.6}$$
$$q_2^{n+1/2} = q_2^n - \Delta t \partial_x(q_2^n u) \tag{5.7}$$

2. Then we will add the source term:

$$q_1^{n+1} = q_1^{n+1/2} \tag{5.8}$$
$$q_2^{n+1} = q_2^{n+1/2} - \partial_x P^{n+1/2} \tag{5.9}$$

   where $P^{n+1/2} = q_1^{n+1/2} c_s^2$.

For the advection, let us for the moment simply choose the donor-cell algorithm. The q_1 and q_2 are located at the grid cell centers, so that we have $q_{1,i}$ and $q_{2,i}$ with $0 \leq i \leq N-1$. To produce an advective flux at the cell interfaces, we need to calculate first the cell interface value of the velocity. We do this simply by averaging the velocity over the two adjacent cells:

$$u_{i+1/2} = \frac{1}{2}\left(\frac{q_{2,i}}{q_{1,i}} + \frac{q_{2,i+1}}{q_{1,i+1}}\right) \tag{5.10}$$

Then we define the flux:

$$f_{1,i+1/2} = \begin{cases} q^n_{1,i} u_{i+1/2} & \text{for} \quad u_{i+1/2} > 0 \\ q^n_{1,i+1} u_{i+1/2} & \text{for} \quad u_{i+1/2} < 0 \end{cases} \tag{5.11}$$

and the same for $f_{2,i+1/2}$. We update q as

$$q^{n+1/2}_{1,i} = q^n_{1,i} - \Delta t \frac{f_{1,i+1/2} - f_{1,i-1/2}}{x_{i+1/2} - x_{i-1/2}} \tag{5.12}$$

and the same for $q^{n+1/2}_{2,i}$. This is the donor-cell advection. The $q^{n+1/2}$ does not mean that we do the advection for only half a time step. It only means that, by operator splitting, we still have another operator to go.

For this second operator, the addition of the source, we simply take the approximation:

$$\left.\frac{\partial P}{\partial x}\right|_i \rightarrow \frac{P^{n+1/2}_{i+1} - P^{n+1/2}_{i-1}}{x_{i+1} - x_{i-1}} = c_s^2 \frac{\rho^{n+1/2}_{i+1} - \rho^{n+1/2}_{i-1}}{x_{i+1} - x_{i-1}} \tag{5.13}$$

and we perform the following update, this time only for $q_{2,i}$

$$q^{n+1}_{2,i} = q^{n+1/2}_{2,i} - c_s^2 \frac{\rho^{n+1/2}_{i+1} - \rho^{n+1/2}_{i-1}}{x_{i+1} - x_{i-1}} \tag{5.14}$$

In principle this is it, but we must take special care at the boundary. We must decide which kind of boundary condition to take. In Section 5.3 we shall go into more detail, but for now let us assume a reflective boundary condition in which the advective velocities at the left interface of the left boundary cell and the right interface of the right boundary cell are assumed to be zero, and in which the pressure outside of the domain is assumed to be equal to the pressure in the boundary cell. If $i = 0$ is the first cell, and $i = N - 1$ is the last cell, then we write $u_{-1/2} = u_{N-1/2} = 0$ for the advection. For the pressure source term in the momentum equation we take (only the left boundary condition as an example; right goes similar):

$$q^{n+1}_{2,0} = q^{n+1/2}_{2,0} - \frac{1}{2} c_s^2 \frac{\rho^{n+1/2}_1 - \rho^{n+1/2}_0}{x_1 - x_0} \tag{5.15}$$

The factor 1/2 comes in because actually the $\Delta x = x_1 - x_{-1}$, but since without ghost cells x_{-1} does not exist, we do it with a factor 1/2. Note, then, that ghost cells may indeed be handy. We will use them later.

5.2.2 The computer implementation

For the computer implementation we must first identify the indices of the interfaces, because a computer cannot address $i + 1/2$ in an array. As usual we *always* take the interface to be *left* of the cell with the same number: $u_{i-1/2} \equiv u[i]$. We make an array of N values for $q_1 \equiv \rho$ and $q_2 \equiv \rho u$, and any interface quantities will be arrays of $N + 1$ values, starting with the left interface of the leftmost cell and ending with the right interface of the rightmost cell.

Now let us create a subroutine for doing one single time step of the hydrodynamics:

```
pro hydroiso_cen,x,xi,rho,rhou,e,gamma,dt
nx = n_elements(x)
```

```
;
; Compute the velocity at the cell interfaces
;
ui = dblarr(nx+1)
for ix=1,nx-1 do begin
   ui[ix] = 0.5 * ( rhou[ix]/rho[ix] + rhou[ix-1]/rho[ix-1] )
endfor
;
; Compute the flux for rho
;
fluxrho = dblarr(nx+1)
for ix=1,nx-1 do begin
   if ui[ix] gt 0. then begin
      fluxrho[ix] = rho[ix-1] * ui[ix]
   endif else begin
      fluxrho[ix] = rho[ix] * ui[ix]
   endelse
end
;
; Update the density
;
for ix=0,nx-1 do begin
   rho[ix] = rho[ix] - (dt/(xi[ix+1]-xi[ix]))*$
                  (fluxrho[ix+1]-fluxrho[ix])
endfor
;
; Compute the flux for rho u
;
fluxrhou = dblarr(nx+1)
for ix=1,nx-1 do begin
   if ui[ix] gt 0. then begin
      fluxrhou[ix] = rhou[ix-1]^2 / rho[ix-1]
   endif else begin
      fluxrhou[ix] = rhou[ix]^2 / rho[ix]
   endelse
end
;
; Update the momentum
;
for ix=0,nx-1 do begin
   rhou[ix] = rhou[ix] - (dt/(xi[ix+1]-xi[ix]))*$
                  (fluxrhou[ix+1]-fluxrhou[ix])
endfor
;
; Compute the pressure
;
p    = (gamma-1.d0)*rho*e
```

```
;
; Now add the pressure force, for all cells
; except the ones near the boundary
;
for ix=1,nx-2 do begin
   rhou[ix] = rhou[ix] - dt*(p[ix+1]-p[ix-1])/(x[ix+1]-x[ix-1])
endfor
;
; Now do the boundary cells, assuming mirror
; symmetry in the boundaries
;
rhou[0]    = rhou[0]    - 0.5*dt*(p[1]-p[0])/(x[1]-x[0])
rhou[nx-1] = rhou[nx-1] - 0.5*dt*(p[nx-1]-p[nx-2])/(x[nx-1]-x[nx-2])
;
; Done
;
end
```

This contains everything we discussed above: the advection, the computation of the new pressure, the implementation of the boundary conditions.

Now let us define a test problem.

```
nx         = 100
nt         = 1000
x0         = 0.d0
x1         = 100.d0
xmid       = 0.5 * (x0+x1)
dt         = 0.25
cfl        = 0.5
x          = x0 + (x1-x0)*(dindgen(nx)/(nx-1.d0))
gamma      = 7./5.
rho        = dblarr(nx,nt+1)
rhou       = dblarr(nx,nt+1)
e          = dblarr(nx)+1.d0
time       = dblarr(nt+1)
dg         = 0.1*(x1-x0)
rho[*,0]   = 1.d0+0.3*exp(-(x-xmid)^2/dg^2)
```

And produce the xi array which is needed, as well as a dx array.

```
;
; Now some additional arrays are set up
;
xi            = dblarr(nx+1)
xi[1:nx-1]    = 0.5 * ( x[1:nx-1] + x[0:nx-2] )
xi[0]         = 2*xi[1] - xi[2]
xi[nx]        = 2*xi[nx-1] - xi[nx-2]
dx            = ( xi[1:nx] - xi[0:nx-1] )
```

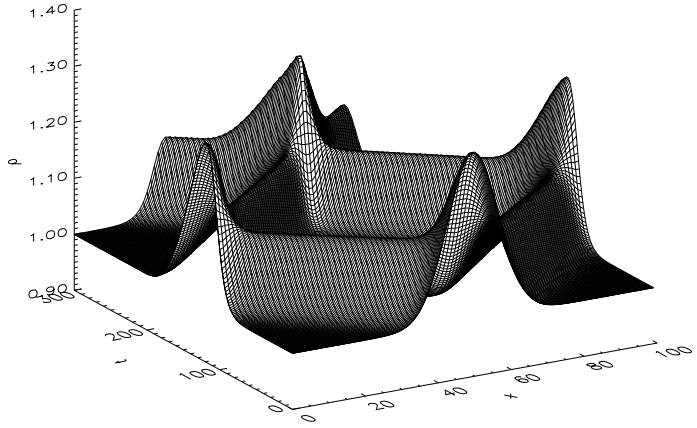

Figure 5.1. The density as a function of space x and time t with initial condition of a gaussian perturbation, solved with the simple first-order hydrodynamics algorithm of Section 5.2.

We now want to compute 1000 time steps of the hydrodynamics. We must recompute the smallest allowed Δt each time step. We define a dimensionless *Courant number* which tells how much smaller than the formal maximum we wish to take the time step. And here we go:

```
;
; Now the hydro is done
;
for it=1,nt do begin
   qrho       = rho[*,it-1]
   qrhou      = rhou[*,it-1]
   cs         = sqrt(gamma*(gamma-1)*e)
   dum        = dx/(cs+abs(qrhou/qrho))
   dt         = cfl*min(dum)
   time[it]   = time[it-1]+dt
   ;;
   print,'Time step ',it,', Time = ',time[it],', Dt = ',dt
   ;;
   hydroiso_cen,x,xi,qrho,qrhou,e,gamma,dt
   ;;
   rho[*,it]  = qrho
   rhou[*,it] = qrhou
endfor
```

The results are shown in Fig. 5.1. It is seen that the Gaussian blob in the density splits up in a left- and a right-moving blob, which both reflect off the walls of the domain and return toward the center of the domain. By that time, due to the non-linearity of the equations and the strength of the pertubation (30% of the background value), the wave has steepened into something that looks like a shock, but is still spread over multiple cells. This wave pattern repeats many times and the shock front steepens, but still remains smeared over a number of cells. Nevertheless, it seems that we have produced our first succesful hydrodynamics code.

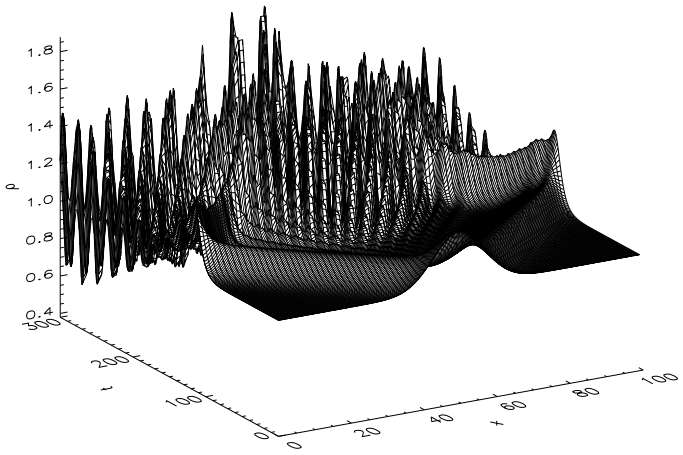

Figure 5.2. Same as Fig. 5.1, but now using always the start-of-the-time-step variables for the update of the state. A numerical instability appears.

5.2.3 The order of evaluation matters...

The above exercise may have not seemed too difficult. But still we only have a first order algorithm and of course we have been lucky that we have not fallen into the many pitfalls that can exist. One possible problem appears if the order of the evaluations is done in a different way. Suppose that the evaluation of the pressure is done before the advection of the density. In effect we then get

$$q_{2,i}^{n+1} = q_{2,i}^{n+1/2} - c_s^2 \frac{\rho_{i+1}^n - \rho_{i-1}^n}{x_{i+1} - x_{i-1}} \tag{5.16}$$

instead of Eq. 5.14. This is only a very minor change in the algorithm. It simply means that we use the values at the start of the time step for any of the operations we do (advection, source addition). It may not seem as a mistake, but Fig. 5.2 shows the result. In the beginning the wave propagates in the way that was expected, but soon a serious instability appears and wrecks the entire solution. This shows that when sources like the pressure source are added, a true operator splitting is required: first the advection, then recompute the pressure from the new variables, and then add the pressure source.

5.3 Boundary conditions, ghost cells

We have seen already in chapter 4 that the use of ghost cells at the boundaries can be very useful to easily implement boundary conditions. There are a number of different kinds of boundary conditions one can install in this way. Here is a list of common ones (where we mainly focus on the left boundary).

1. Periodic boundary conditions: $\rho[0] = \rho[N_x]$, $u[0] = u[N_x]$ (and similarly $\rho[N_x+1] = \rho[1]$, $u[N_x + 1] = u[1]$).

2. Reflective boundary conditions: $\rho[0] = \rho[1]$, $u[0] = -u[1]$.

3. Free outflow/inflow: $\rho[0] = \rho[1]$, $u[0] = u[1]$.

4. Free outflow, no inflow: $\rho[0] = \rho[1]$, $u[0] = -\mathrm{abs}(u[1])$.

5. Given-state boundary condition: $\rho[0] = \rho_l(t)$, $u[0] = u_l(t)$.

and similar for the right boundary. In 2-D and 3-D hydrodynamics these types of boundary conditions are the same, but then including also the other two velocity components.

Periodic boundary conditions have many uses. First of all, sometimes the system we wish to model is clearly periodic (such as if we wish to model the atmosphere of the Earth where going East eventually brings you back where you came from). But also for other applications can periodic boundary conditions be useful: if we wish, for instance, to model the nature of turbulence, then we are anyway interested in flow patterns with scales that are much smaller than the computational domain. But we do wish to ensure that there are no artificial damping or forcing effects caused by imposing some arbitrary boundary condition. A periodic boundary condition basically removes beforehand any artificial boundary effects because the turbulence can now not feel the boundary at all. The only way that the turbulence can 'feel' the fact that the computational domain is limited is that it cannot grow modes larger than the size of the computational domain.

Reflective boundary conditions are also very often used. The advantage is that no mass nor energy can flow off the grid. Energy and mass are therefore globally conserved, or in other words: the system is closed. There are no external influences that may affect the solution because the problem is perfectly self-contained. A disadvantage of reflective boundary conditions is that any waves that are spawned by the interesting part of the flow pattern that we are modeling are also perfectly reflected, and typically return to the area of interest and start interfering with that area. For instance, if we want to model how a rotating irregular body in a volume of gas stirrs waves and how these waves transport away energy and damp the rotation of the body, then the reflective boundary conditions bring back the waves to the body and start affecting its motion. This is then clearly undesired.

A free outflow/inflow boundary condition has the advantage that any waves that are produced in the model are simply advected off the grid. The waves do not return anymore. This is clearly desirable in many cases. But the problem is that if for some reason the velocity $u[1]$ becomes inflowing ($u[1] > 0$), then the state at cell i will determine the influx of matter. This can typically cause completely arbitrary influx of matter. One can see why this is the case: one of the two characteristics is pointing inward into the grid ($\lambda_+ = u + c_s$). So there is always in inflow of information into the grid. But if the state in the ghost cell is purely given as a function of the state at $i = 1$, then the state at $i = 1$ determines itself what kind of information it wishes to receive from grid point $i = 0$ (ghost cell) through that characteristic. Clearly this is unphysical because the inward-pointing characteristic ($\lambda_+ = u + c_s$) should transport information only toward positive x and *never* receive any information from larger x. This boundary condition is therefore quite dangerous. Only if we have a *supersonic* outflowing motion toward this boundary can we be sure that this boundary condition will never cause problems. The reason is that in the case of supersonic outflow both characteristics point out of the grid: $\lambda_- = u - c_s < 0$ and $\lambda_+ = u + c_s < 0$. Then by copying the state to the ghost cell no information is propagated in the upstream direction.

The outflow-but-no-inflow condition is an attempt to solve the problem of arbitrary inflow in subsonic cases. It clearly does not allow unphysical inflow of matter, but it allows matter to flow off the grid. It is, so to speak, a sink. It is not ideal, because there is still the problem of propagating information upstream, but if the goal of imposing this boundary condition is to get rid of any matter than comes near the boundary, then this is sometimes used.

With the given-state boundary condition one can do many things. For instance, one can force waves, or impose some influx of matter at the boundary. However, with this method this does not always work. Suppose we impose an influx of $f = \rho[0] * u[0] = 1$ at the boundary by putting $\rho[0] = 1$, $u[0] = 1$, but the state in the modeling domain is $\rho[1] = 1000$, $u[1] = 1$. We may think that we imposed $f = 1$ at the boundary, but instead the pressure in cell 1 is completely overwhelming the pressure in ghost cell 0, and a negative flux of matter will result. So the imposed-state boundary condition works only if the influx has a momentum flux that clearly overcomes the pressure in the modeling domain. We could, instead, decide to impose a flux at boundary $i = 1/2$, i.e. the left wall of the boundary cell, and completely skip the ghost cell approach. That would allow a perfect and strict imposition of a flux. A disadvantage here is that one does not know if this flux is physical in this situation.

5.3.1 Ghost cells for schemes with a 5-point stencil

A single ghost cell on each boundary is sufficient if the stencil of the advection scheme is symmetric and 3-point. For linear advection schemes such as Donor-cell and Lax-Wendroff this is the case. However, some other schemes such as Beam-warming or Fromm require, for the update at point i also information of the cells at $i \pm 2$ in addition to the $i \pm 1$ points. Also, when the flux limiter method is used with a non-linear flux limiter recipe, then often the $i \pm 2$ are used. Such schemes are therefore in principle 5-point schemes, even if, for a given velocity direction, only 3 points are used ($i-2, i-1, i$ or $i, i+1, i+2$) because we do not know a-priori which direction the velocity points. For such schemes a single ghost cell on each boundary is not sufficient. We must implement a double layer of ghost cells. For the direct flux from these cells into the physical domain the first ghost cell (the one next to the physical domain) is of most importance, because that is the donor of the flux. But the second ghost cell will, in the case of flux-limiters, determine what the value of the flux limiter is, and can thereby significantly affect the solution nonetheless. The implementation of the boundary condition goes, for the rest, similar to the single-ghost cell case.

5.4 Hydrodynamics with ghost cells

So now that we see that ghost cells are extremely useful to implement boundary conditions, let us start all over again and produce a new algorithm in which two ghost cells are used on each side. Also, let us implement a standardized advection subroutine which we shall call `advect()`, which advects any conserved quantity with a given velocity given at the cell interfaces, and a standardized boundary condition routine `boundary()` that setst he ghost cells to the proper values consistent with the kind of boundary condition used. Let us first show the routine `advect()`, but note that a more sophisticated version of `advect()` (with more choices of flux limiters) will be made available for the exercises:

```
pro advect,x,xi,q,ui,dt,fluxlim,nghost
nx     = n_elements(xi)-1
if n_elements(x) ne nx then stop
if n_elements(xi) ne nx+1 then stop
if n_elements(q) ne nx then stop
if n_elements(ui) ne nx+1 then stop
if nghost lt 1 then stop
;
```

```
; Determine the r_{i-1/2} for the flux limiter
;
r = dblarr(nx+1)
for i=2,nx-2 do begin
   dq = (q[i]-q[i-1])
   if abs(dq) gt 0.d0 then begin
      if(ui[i] ge 0.d0) then begin
         r[i] = (q[i-1]-q[i-2])/dq
      endif else begin
         r[i] = (q[i+1]-q[i])/dq
      endelse
   endif
endfor
;
; Determine the flux limiter
; (many other flux limiters can be implemented here!)
;
case fluxlim of
   'donor-cell': begin
      phi = dblarr(nx+1)
   end
   'superbee': begin
      phi = dblarr(nx+1)
      for i=1,nx-1 do begin
         a = min([1.d0,2.d0*r[i]])
         b = min([2.d0,r[i]])
         phi[i] = max([0.d0,a,b])
      endfor
   end
   else: stop
endcase
;
; Now construct the flux
;
flux = dblarr(nx+1)
for i=1,nx-1 do begin
   if ui[i] ge 0.d0 then begin
      flux[i] = ui[i] * q[i-1]
   endif else begin
      flux[i] = ui[i] * q[i]
   endelse
   flux[i] = flux[i] + 0.5 * abs(ui[i]) * $
                       (1-abs(ui[i]*dt/(x[i]-x[i-1]))) * $
                       phi[i] * (q[i]-q[i-1])
endfor
;
; Update the cells, except the ghost cells
```

```
;
for i=nghost,nx-1-nghost do begin
   q[i] = q[i] - dt * ( flux[i+1]-flux[i] ) / ( xi[i+1] - xi[i] )
endfor
;
end
```

Now the boundary condition implementation routine, which is only necessary for imposing periodic boundary conditions or mirror boundary conditions, is:

```
pro boundary,rho,rhou,periodic=periodic,mirror=mirror
;
; Get the number of grid points including the ghost cells
;
nx = n_elements(rho)
;
; If periodic, then install periodic BC, using two ghost cells
; on each side (two are required for the non-lin flux limiter)
;
if keyword_set(periodic) then begin
   rho[0]     = rho[nx-4]
   rho[1]     = rho[nx-3]
   rho[nx-2]  = rho[2]
   rho[nx-1]  = rho[3]
   rhou[0]    = rhou[nx-4]
   rhou[1]    = rhou[nx-3]
   rhou[nx-2] = rhou[2]
   rhou[nx-1] = rhou[3]
endif
;
; If mirror symmetry, then install mirror BC, using two ghost cells
; on each side (two are required for the non-lin flux limiter)
;
if keyword_set(mirror) then begin
   rho[0]     = rho[3]
   rho[1]     = rho[2]
   rho[nx-2]  = rho[nx-3]
   rho[nx-1]  = rho[nx-4]
   rhou[0]    = -rhou[3]
   rhou[1]    = -rhou[2]
   rhou[nx-2] = -rhou[nx-3]
   rhou[nx-1] = -rhou[nx-4]
endif
end
```

The actual hydrodynamics subroutine makes use of the above routines. It reads:

```
pro hydrostep,x,xi,rho,rhou,e,gamma,dt,periodic=periodic,$
```

```
         mirror=mirror,fluxlim=fluxlim,nrvisc=nrvisc
;
; Check for conflicting settings
;
if keyword_set(mirror) and keyword_set(periodic) then stop
;
; Use 2 ghost cells on each side
;
nghost = 2
;
; If not defined, install default flux limiter
;
if not keyword_set(fluxlim) then fluxlim='donor-cell'
;
; Get the number of grid points including the ghost cells
;
nx = n_elements(x)
;
; Impose boundary conditions
;
boundary,rho,rhou,periodic=periodic,mirror=mirror
;
; Compute the velocity at the cell interfaces
;
ui    = dblarr(nx+1)
for ix=1,nx-1 do begin
   ui[ix] = 0.5 * ( rhou[ix]/rho[ix] + rhou[ix-1]/rho[ix-1] )
endfor
;
; Advect rho
;
advect,x,xi,rho,ui,dt,fluxlim,nghost
;
; Advect rho u
;
advect,x,xi,rhou,ui,dt,fluxlim,nghost
;
; Re-impose boundary conditions
;
boundary,rho,rhou,periodic=periodic,mirror=mirror
;
; Compute the pressure
;
p    = (gamma-1.d0)*rho*e
;
; Now add the pressure force, for all cells except the ghost cells
;
```

```
for ix=2,nx-3 do begin
   rhou[ix] = rhou[ix] - dt*(p[ix+1]-p[ix-1])/(x[ix+1]-x[ix-1])
endfor
;
; Re-impose boundary conditions a last time (not
; strictly necessary)
;
boundary,rho,rhou,periodic=periodic,mirror=mirror
;
; Done
;
end
```

This routine makes sure to always re-implement the boundary conditions for the ghost cells. This is done without checking which variables have to be updated, so it is a bit inefficient, but it is safe. Note, also, that in the above routines `nx` denotes the number of cells including the ghost cells. With the above routines one can again do experiments and see how the solutions behave for different flux limiters and boundary conditions. One true advantage of the above implementation is that it is now very easy to impose periodic boundary conditions, because we only copy the state variables from the left to the right boundary and from the right to the left boundary.

→ **Exercise:** Implement the above routines and create a setup with a sound wave moving from left to right. Impose periodic boundary conditions. Tip: Make sure that the sine-wave is such that `rho[0]=rho[nx-4]` and `rho[1]=rho[nx-3]`, so that the periodicity of the wave is perfect, in the presence of the two ghost cells on each side. Describe how the sound wave steepens, sheds smaller waves and forms a shock.

5.5 Now including the energy equation

So far we have neglected the energy equation because we assumed that the temperature was constant at all times. However, most interesting applications do not have this property. We therefore must include the energy equation, including the work source term due to the pressure force:

$$\partial_t(\rho e_{\rm tot}) + \partial_x(\rho e_{\rm tot} u) = -\partial_x(Pu) \tag{5.17}$$

We define a third conserved quantity $q_3 \equiv \rho e_{\rm tot}$. As in the case of the momentum equation we apply the method of operator splitting here: we first solve the equation $\partial_t q_3 + \partial_x(q_3 u) = 0$ for one time step, and then solve (with the new variables) the equation $\partial_t q_3 = -\partial_x(Pu)$ for the same time step. The advection of the total energy is done in exactly the same way as for the density and the momentum. In fact, we can likewise use the subroutine `advect()` for it.

The difficulty lies in the work term $-\partial_x(Pu)$. First of all we need to calculate the pressure P from the three conserved quantities $q_1 \equiv \rho$, $q_2 \equiv \rho u$ and $q_3 \equiv \rho e_{\rm tot}$. In fact, we also need this P for the source term in the momentum equation. The way this can be done is to first compute the thermal energy $e_{\rm th}$ from the total energy:

$$u = q_2/q_1 \tag{5.18}$$

$$e_{\rm tot} = q_3/q_1 \tag{5.19}$$

$$e_{\rm kin} = u^2/2 \tag{5.20}$$

$$e_{\rm th} = e_{\rm tot} - e_{\rm kin} \tag{5.21}$$

Once we know this, we compute the pressure according to $P = (\gamma - 1)\rho e_{\rm th}$.

Then we use P for both the source term in the momentum equation (the force) and for the source term in the energy equation (the work). For the former we write

$$\frac{q_{2,i}^{n+1} - q_{2,i}^{n+1/2}}{\Delta t} = -\frac{P_{i+1} - P_{i-1}}{2\Delta x} \tag{5.22}$$

where $q_{2,i}^{n+1/2}$ denotes the update of $q_{2,i}^{n}$ due to the advection. For the latter we can write:

$$\frac{q_{3,i}^{n+1} - q_{3,i}^{n+1/2}}{\Delta t} = -\frac{P_{i+1}u_{i+1} - P_{i-1}u_{i-1}}{2\Delta x} \tag{5.23}$$

These new elements can be readily built in into the algorithm of Section 5.4, and this should produce a working algorithm.

5.5.1 Are these equations conservative?

As was mentioned in Chapter 4 it can be very important to make sure that conserved quantities actually remain conserved in the numerical simulation. In the current approach, however, we have put the pressure force and work to the right-hand-side of the conservation equation, as a source term. This raises the question: are the equations still conservative? The answer is: in principle yes, if the force and work terms are written in a proper way. Define the momentum flux through interface $i - 1/2$ to be $f_{p,i-1/2} = \frac{1}{2}(P_{i-1} + P_i)$, and similar for $i + 1/2$. Then we obtain:

$$q_{2,i}^{n+1} = q_{2,i}^{n+1/2} - \Delta t \frac{f_{p,i+1/2} - f_{p,i+1/2}}{x_{i+1/2} - x_{i-1/2}} = q_{2,i}^{n+1/2} - \frac{\Delta t}{2}\frac{P_{i+1} - P_{i-1}}{x_{i+1/2} - x_{i-1/2}} \tag{5.24}$$

and similar for the energy equation. For a regular grid this becomes equal to the expressions of Eqs.(5.22,5.23). This shows that implicitly the equations are numerically conservative, even though we have not explicitly put them in a flux-conserved form. For non-regular gridding one should stick to the presently derived expression, so that flux conservation is guaranteed.

5.6 Shock waves and the Von Neumann - Richtmyer artificial viscosity

We have seen in the examples above that waves tend to steepen and form shocks. Let us take a closer look, by simulating a true right-moving shock wave and analyzing its behavior. Let us assume that a shock wave is moving from the left boundary into a non-moving medium. We can define the *Mach number* $\mathcal{M}$ to be the ratio of the shock speed u_s to the sound speed C_s on the pre-shock medium. Let us call the pre-shock medium the 'right domain' and the post-shock region the 'left domain', because the shock moves from left to right. We therefore define $\mathcal{M} \equiv u_s/C_{s,r}$. According to the conservation of density, momentum and energy over the shock (Rankine-Hugoniot) we can write:

$$\rho_l = \rho_r \frac{(\gamma+1)\mathcal{M}^2}{(\gamma-1)\mathcal{M}^2 + 2} \tag{5.25}$$

$$P_l = P_r \frac{2\gamma\mathcal{M}^2 - (\gamma-1)}{\gamma+1} \tag{5.26}$$

$$u_s = \mathcal{M}\sqrt{\gamma\frac{P_r}{\rho_r}} \tag{5.27}$$

$$u_l = u_s \frac{\rho_l - \rho_r}{\rho_l} \tag{5.28}$$

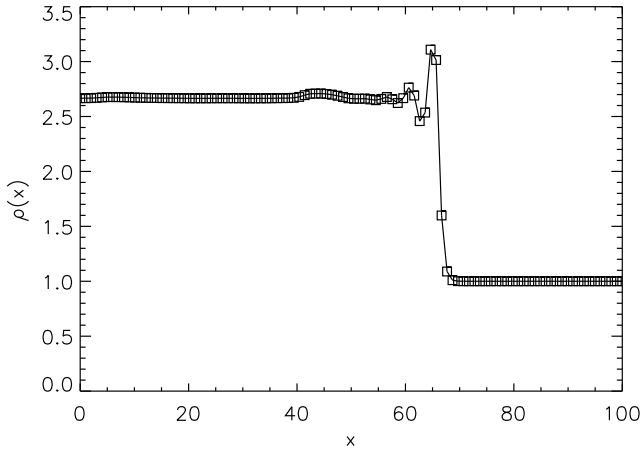

Figure 5.3. The result of a test problem with a $\mathcal{M} = 2$ shock wave moving into a non-moving medium of $\rho = 1$, $P = 0.1$, modeled using advection with a superbee flux limiter.

This is then the state that we plug into the ghost cells on the left boundary. It should, in principle, produce a clean shock front moving from the left boundary to the right. Now let us see what we get if we use advection according to the superbee flux limiter for a shock of Mach 2. This is depicted in Fig. 5.3 What one sees is that in principle the shock wave is well reproduced, but there are some oscillations at the shock front. This has a natural explanation. The equations for the hydrodynamics solver were derived from the continuous Euler equations. These equations are valid for smooth flow, but break down for shock fronts, so we should not expect the hydrodynamics solver to be able to handle shocks. In fact, a shock is the only location where non-viscous hydrodynamics can increase the entropy of the gas flow. The way that Nature does this is that at very small spatial scales the molecular viscosity becomes important, and this viscosity produces entropy in the shock front. Basically one can say that in Nature the shock front has a small but non-negligible width L such that the Reynolds number Re$= \frac{uL}{\nu}$ is of order unity in this shock front. Kinetic energy can then be dissipated to heat in this shock, which increases the entropy of the gas.

In numerical hydrodynamics the grid cell size is usually orders of magnitude larger than the true width of the shock. And the equations that we used for the hydrodynamics solver did not explicitly include any entropy-generating viscosity terms. Therefore we *expect* that our algorithm should produce errors in the flow near the shock front, in particular downstream of the shock front. The interesting thing is, in fact, that the method already does surprisingly well, given these thoughts (see, however, Section 5.10.2 for what happens if a different energy equation is used). Far downstream of the shock front the state is reasonably correct and the entropy increase, expected from the shock, is indeed there. So one should ask oneself two questions:

1. Why does our solver handle the shock reasonably well, even though we never inserted any entropy-generating viscous terms by hand?

2. How can we do better, so that we do not get the wiggles behind the shock?

To answer question 1, the answer lies in the fact that due to the intrinsic diffusivity of the advection algorithm, any oscillations are smeared out quickly. The question remains then, how does the code know how much entropy to generate? This can be traced back to the fact that the algorithm is in conservative form. If the post-shock state variables are smoothed out and all

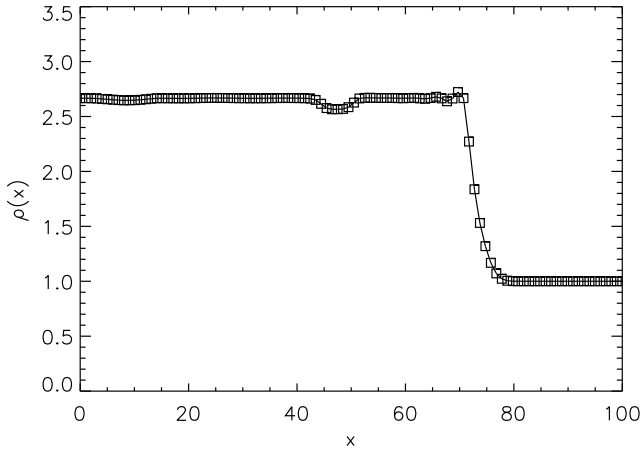

Figure 5.4. As Fig. 5.3, but now with von Neumann-Richtmyer artifi cial viscosity with $\xi = 3$.

oscillations have gone, then there *is only one set of state variables* consistent with the post-shock density, momentum and energy flux. So diffusing oscillations out, but keeping the algorithm perfectly conservative, must automatically produce the right state and therefore the right increase in entropy compared to the pre-shock state. Diffusion is therefore a good thing in this respect. In fact, if we would use the donor-cell algorithm for the advection, which has intrinsically more numerical diffusion, then the post-shock oscillations are already quite low. But the drawback of donor-cell algorithm is that *any* flow feature is smeared out. This can therefore not be the final solution.

To answer question 2: we need to find a way to increase the diffusivity near the shock front, but not elsewhere. In fact, we can follow a physically motivated path by introducing an *artificial viscosity* (as opposed to artificial diffusivity) which sort of mimics the physical shock viscosity, but then smeared out over a region the size of a few grid cells instead of the unresolvable true shock width. The most popular artificial viscosity recipe for handling shocks is the *von Neumann-Richtmyer artificial viscosity*. This is a bulk viscosity which acts as an additional pressure:

$$\Pi_i = \begin{cases} \frac{1}{4}\xi^2(u_{i+1} - u_{i-1})^2\rho_i & \text{if} \quad u_{i+1} \leq u_{i-1} \\ 0 & \text{if} \quad u_{i+1} > u_{i-1} \end{cases} \tag{5.29}$$

where ξ is a tuning parameter which specifies over how many grid cells a shock should be spread out. It is typically chosen to be of the order of 2 or 3. To implement it in the algorithm one replaces any instance of the pressure P with $P + \Pi$. This takes care of the force as well as the work done by this bulk viscosity. For the above test problem, the result for $\xi = 3.0$ is shown in Fig. 5.4 One sees that the post-shock oscillations have gone. The penalty is that the viscosity affects the pre-shock region and the shock is therefore not as sharp anymore as we had before. Also there is a little dip in the density profile where no dip should be. This has formed in the very initial phase as the shock started to propagate into the domain. Since the shock as modeled by the numerical algorithm is not infinitely sharp, the intial conditions were not in agreement with the "numerical form of the shock", and startup oscillations were present. They produced a bit too much entropy in this region, causing the density (for the same pressure) to be a bit lower than should be.

Von Neumann-Richtmyer artificial viscosity has the good property that it only becomes strong when its presence is required, and becomes negligible when the flow is smooth. It is

therefore much better than the uncontrolled numerical diffusivity of for instance the donor-cell algorithm. The von Neumann-Richtmyer artificial viscosity, and varieties thereof, is used in a great many numerical hydrodynamics codes.

5.7 Odd-even decoupling

The numerical algorithms we have constructed so far have an interesting problematic property that is only seldomly serious, but might be useful to keep in mind.

Suppose we start with a situation in which the velocity is everywhere zero: $u(x) = 0$. We assume that the specific thermal energy ($\propto$ temperature) is also constant, for simplicity. Let us take it: $e_{\rm th}(x) = 1$. But we let the density vary as a 'sawtooth' profile:

$$\rho_i = \begin{cases} 1 & \text{if} i = 1, 3, 5, 7 \cdots \\ 2 & \text{if} i = 2, 4, 6, 8 \cdots \end{cases} \tag{5.30}$$

The pressure in these cells is then $P_i = 1$ for $i = 1, 3, 5, \cdots$ and $P_i = 2$ for $i = 2, 4, 6, \cdots$. So in principle the even cells are over-pressurized compared to their neighbors. Physically these cells should quickly depressurize by moving part of their content to their under-pressurized neighbors. This should give a high-frequency oscillation, and with the usual numerical viscosity this should then automatically damp out. However, for this case the momentum flux through cell interfaces $i - 1/2$ and $i + 1/2$ is

$$F^{(1)}_{i-1/2} = \frac{1}{2}(P_{i-1} + P_i) \qquad F^{(1)}_{i+1/2} = \frac{1}{2}(P_i + P_{i+1}) \tag{5.31}$$

The update of the momentum at cell center i is:

$$\frac{\rho_i^{n+1} u_i^{n+1}}{\Delta t} = -\frac{F^{(1)}_{i+1/2} - F^{(1)}_{i-1/2}}{\Delta x} = -\frac{1}{2\Delta x}(P_{i+1} - P_{i-1}) = 0 \tag{5.32}$$

So because the P_i cancels out, the update of the momentum is zero. This is also logical, since the fluxes of momentum in both interfaces are the same, and therefore their differences cancel. This is a potential problem because the code leaves these oscillations there without damping them out, i.e. it leaves an unphysical solution. In fact, it is fundamentally impossible to solve this problem if momentum, density and pressure are located on the same gridpoints, unless artificial diffusion is invoked. This phenomenon is called *odd-even decoupling*.

One sees that the odd-even decoupling does not involve an exponetially growing unstable mode. But there is also no damping. So any numerical noise or other causes of errors may simply get 'stuck' in an odd-even mode where they do not get amplified, but they do not get damped either. Such 'sawtooth' modes are clearly an unwanted effect. One simple way to get rid of them would be diffusion, but that would also diffuse out flow patterns that we wish to resolve. A better way, which also automatically increases the order of the algorithm, is to use *staggered grids*.

5.8 Staggered grids

A numerical hydrodynamics scheme with a *staggered grid* places any vectorial components, such as the momentum density $q_2 \equiv \rho u$, not on the cell centers but on the cell interfaces. In such a scheme one would have $q_1 \equiv \rho$ and $q_3 \equiv \rho e_{\rm tot}$ located at the cell centers and ρu located on the

cell interfaces. But since q_2 is also a conserved quantity we must also define a cell around q_2. The location of the cell boundaries for q_1 and q_3 act as cell centers for q_2, whereas the locations of the cell centers for q_1 and q_3 act as cell boundaries for q_2.

Using staggered grids improves the accuracy of the algorithm and it also solves the odd-even decoupling problem. It is also more stable numerically than cell-centered algoriths. Where cell-centered algorithms sometimes crash when the algorithm is stretched too far, a staggered grid algorithm usually survives those conditions.

→ **Exercise:** Show, using the example above, that a staggered grid does not have the odd-even decoupling problem.

A drawback of staggered grids is that it requires a much more complex book-keeping of the cells and interfaces, especially when one goes to multiple dimensions. This also could make it slightly more difficult to implement adaptive mesh refinement methods and such.

A number of hydrodynamics codes use staggered grids. For instance, the famous ZEUS code, often used in astrophysics, is based on staggered grids. But also many code are based on cell-centered schemes. Both approaches have their special advantages and disadvantages.

5.9 External gravity force

It possible, without much effort, to add an external force to the hydrodynamics scheme we have produced so far. For instance, if we wish to model the flow of gas in a gravity field, we need to solve the following equations (1-D for simplicity):

$$\partial_t \rho + \partial_x(\rho u) = 0 \tag{5.33}$$

$$\partial_t(\rho u) + \partial_x(\rho u^2 + P) = -\rho \partial_x \Phi \tag{5.34}$$

$$\partial_t(\rho e_{\rm tot}) + \partial_x[(\rho e_{\rm tot} + P)u] = -\rho u \partial_x \Phi \tag{5.35}$$

where $\Phi(x)$ is the gravitational potential. The $-\rho\partial_x\Phi$ term on the rhs of the momentum equation is the force, whereas the $-\rho u\partial_x\Phi$ term on the rhs of the energy equation is the work done by the gravity on the *total* energy. The idea behind the latter term is that even though the gravity force does not directly affect the thermal energy of the gas, it *does* affect the *total* energy of the gas: the kinetic energy $e_{\rm kin} = u^2/2$ changes due to a change in u. The source term in the energy equation takes care of this.

An example of an application of this set of equations is that of a perturbation on an otherwise hydrostatic atmosphere on the surface of a planet (such as Earth). Let us define x to be the height above the surface of the planet and the gravity potential $\Phi(x) = g\ x$. A static solution must be such that the ∂_t terms of the above equations will be identically zero. Also we must have, by definition, $u = 0$ everywhere. The continuity equation is then automatically solved. The momentum equation, on the other hand, becomes:

$$\frac{\partial P}{\partial x} = -\rho \frac{\partial \Phi}{\partial x} \tag{5.36}$$

In discrete form this becomes:

$$\frac{P_{i+1} - P_{i-1}}{x_{i+1} - x_{i-1}} = -\rho_i \frac{\Phi_{i+1} - \Phi_{i-1}}{x_{i+1} - x_{i-1}} \tag{5.37}$$

So if we can construct a static atmosphere which obeys this equation, then the numerical hydrodynamics algorithm will produce perfectly zero time derivatives. In other words: it has 'recognized a static solution'. We can then add a small perturbation on this solution and see how it moves.

5.10 Alternative methods for the energy equation

5.10.1 Including the gravitational potential into the total energy

If gravity forces are included, the global energy conservation is not guaranteed anymore. Some codes therefore include the potential Φ into the total energy: $e_{\rm tot} = e_{\rm th} + u^2/2 + \Phi$. In that case one can derive that the energy equation becomes

$$\partial_t[\rho(e_{\rm th} + u^2/2 + \Phi)] + \partial_x[\rho(e_{\rm th} + u^2/2 + \Phi + P/\rho)u] = \rho\partial_t\Phi \tag{5.38}$$

where the only remaining source term on the rhs is the time derivative of the potential. In a static potential this would therefore be zero, and perfect conservation of total energy is again guaranteed, even in the presence of gravity.

5.10.2 Using the thermal energy or entropy as the advected variable

A drawback of using the total energy $\rho e_{\rm tot} = \rho(e_{\rm th} + u^2/2)$ (or $\rho e_{\rm tot} = \rho(e_{\rm th} + u^2/2 + \Phi)$ in case of the inclusion of gravity) as the to-be-advected quantity is that this could lead to negative pressures in the simulation, which could crash the simulation. To find the pressure at any time in the algorithm we must first compute the $u^2/2$ from the velocity, and then subtract this from the total energy. In case of gravity, one should also subtract Φ. So we have:

$$e_{\rm th} = e_{\rm tot} - u^2/2 - \Phi \tag{5.39}$$

Now suppose we have a situation in which $e_{\rm th} \ll u^2/2$ and/or $e_{\rm th} \ll \Phi$, then the value of $e_{\rm th}$ is the result of the difference between two large, and nearly equal numbers. Now, the momentum equation (which determines u) and the total energy equation (which determines $e_{\rm tot}$) may always have some small numerical errors. So these errors could then easily lead to an unwanted flip of sign of $e_{\rm tot} - u^2/2 - \Phi$, leading to a negative thermal energy, and hence a negative pressure. This would mean that the program will have to do an emergency stop. Problems of this kind can occur when the flow is extremely supersonic. The total energy is then nearly completely dominated by the kinetic energy. A tiny error in either u or $e_{\rm tot}$ could then lead to negative energies.

For this reason some codes prefer to let go of strict energy conservation and use another form of the energy equation:

$$D_t e_{\rm th} \equiv \partial_t e_{\rm th} + u\partial_x e_{\rm th} = -\frac{P}{\rho}\partial_x u \tag{5.40}$$

which is the Lagrange form of the energy conservation equation (see Chapter 1). The ZEUS code uses this kind of energy equation. The numerical integration of this equation then proceeds with the algorithms of Chapter 3, and the rhs of the equation is added in the usual way. This form of the energy equation has the clear advantage that it is easier to control, and negative energies do not occur unless the time step was taken too big. The drawback is, of course, that energy is now not anymore strictly conserved, and one must always check a-posteriori if the energy error has not increased to unacceptable levels.

An alternative, but rather similar approach is to advect the *entropy* instead of the internal energy:

$$TD_t s \equiv T(\partial_t s + u\partial_x s) = Q \tag{5.41}$$

where Q is the entropy generation source term. The main advantage is that entropy generation can be strictly controlled. This can be of great advantage, for instance in simulating atmospheres where the entropy gradients decide about convective stability or instability of an atmosphere.

This method of using the entropy equation is used in, for instance, the `PENCIL` code, which is a code for modeling accretion disks in astrophysics.

If the thermal energy equation or the entropy equation is used instead of the total energy, then the importance of the use of the Von Neumann-Richtmyer artificial viscosity is amplified enormously. This can be easily understood by looking at Eq. (5.41). Suppose we start with an isentropic state everywhere, but we have conditions such that a shock wave forms. A shock wave generates entropy. But if we have $Q = 0$, then the entropy equation will not generate entropy. The solution of the state behind the shock will therefore clearly be wrong, and in practice it will generate strong oscillations or other unwanted behavior. Now we *must* include an entropy-generating source term in order to produce the right amount of entropy.

5.11 2-D/3-D Hydrodynamics

So far we have focused primarily on 1-D hydrodynamics. In many cases our final interest lies in multi-dimensional gas flow, as this is a more realistic and interesting projection of reality. Fortunately the methods we have covered so far can be relatively easily used for 2-D and 3-D gas flow. This is done with the method of *operator splitting*, in which we apply our algorithms of hydrodynamic integration alternatively in x, y and z direction. We have applied operator splitting already before in other contexts. Here we use this technique to split the 2-D or 3-D problem into its separate directions (Strang, 1968, SIAM, J. Num. Anal, 5, 506).

There are also computer codes that do not use directional operator splitting. They solve the multi-dimensional problem in one go. Such codes are, however, a small minority. One of the advantages of such non-splitting algorithms is that they tend to be less prone to numerical artifacts. But the disadvantage of non-splitting algorithms is that they have to be developed from scratch, i.e. they cannot build on algorithms developed for 1-D. They are therefore usually harder to develop. In this lecture we will therefore exclusively focus on the use of Strang's directional operator splitting.

Let us write the equations of hydrodynamics in 2-D:

$$\partial_t \rho + \partial_x(\rho u) + \partial_y(\rho v) = 0 \tag{5.42}$$
$$\partial_t(\rho u) + \partial_x(\rho u^2 + P) + \partial_y(\rho uv) = 0 \tag{5.43}$$
$$\partial_t(\rho v) + \partial_x(\rho uv) + \partial_y(\rho v^2 + P) = 0 \tag{5.44}$$
$$\partial_t(\rho e_{\rm tot}) + \partial_x[(\rho e_{\rm tot} + P)u] + \partial_y[(\rho e_{\rm tot} + P)v] = 0 \tag{5.45}$$

The splitting is now that we first solve

$$\partial_t \rho + \partial_x(\rho u) = 0 \tag{5.46}$$
$$\partial_t(\rho u) + \partial_x(\rho u^2 + P) = 0 \tag{5.47}$$
$$\partial_t(\rho v) + \partial_x(\rho uv) = 0 \tag{5.48}$$
$$\partial_t(\rho e_{\rm tot}) + \partial_x[(\rho e_{\rm tot} + P)u] = 0 \tag{5.49}$$

for one time step, and then solve

$$\partial_t \rho + \partial_y(\rho v) = 0 \tag{5.50}$$
$$\partial_t(\rho u) + \partial_y(\rho uv) = 0 \tag{5.51}$$
$$\partial_t(\rho v) + \partial_y(\rho v^2 + P) = 0 \tag{5.52}$$
$$\partial_t(\rho e_{\rm tot}) + \partial_y[(\rho e_{\rm tot} + P)v] = 0 \tag{5.53}$$

for the same time step. Some codes do first 1/2 a time step in x-direction, then 1 time step in y-direction and finally another 1/2 time step in x-direction. This has a slightly higher accuracy.

If all the quantities live at the grid cell centers (like the algorithms shown in this chapter, i.e. not the staggered grid ones) then one can simplify the system even more, because then the full 2-D problem of $N_x \times N_y$ grid cells can be split into a set of N_y 1-D problems in x-direction plus a set of N_x 1-D problems in y-direction. We can then simply use the 1-D hydro solver N_x times in x-direction and N_x times in y-direction, provided that we include the perpendicular momentum components also into the equation. So for the N_y 1-D problems in x-direction we must include ρv as another conserved quantity and likewise for the N_x 1-D problems in y-direction we must include ρu as a conserved quantity. Since these perpendicular momentum components act as passive tracers (i.e. have no influence on the flow pattern), the inclusion of these components is nearly trivial. Note that this method of splitting the 2-D problem into sets of 1-D problems is only possible for non-staggered grids. The reason is that if, like the ZEUS code, one uses staggered grids, then the x-momentum lives on $(i + 1/2, j)$ interfaces while the y-momentum lives on $(i, j + 1/2)$ interfaces. The two momentum components therefore do not live on gridpoints that are located on the same 1-D line. Therefore, the schemes used in ZEUS, while they use operator splitting, they cannot go this extra step of reducing the problem to separate 1-D problems. For codes where the momenta are at the cell centers this problem is not there and the reduction to 1-D problems is possible.

With cell-centered algorithms the 2-D or 3-D problem is therefore only marginally more complicated (technically) than the 1-D problem. All our effort in creating a 1-D hydro solver therefore pays off: with only a little effort a fully functional multi-dimensional program can be created, with the 1-D hydro subroutine at its basis.

5.12 Practical matters: input and output of data

Once we wish to do serious modeling we cannot afford ad-hoc management of time steps and output of data anymore. We must think carefully about how to input and output our data, in particular if we wish to write a work horse application in a fast programming language such as Fortran (either F77, F90 or F95), or C/C++. There are a couple of thoughts which might be of use, but some thoughts apply only to codes that go beyond the simple IDL programming done in this lecture:

1. In case of workhorse codes, it often is useful if the initial density, energy and velocity distribution is read into the code in a file format that is similar to the output file format. In this way an output of the hydro code can later be used as the starting point of a continued simulation.

2. In true applications one rarely wants to store the results of each time step. That is too storage-intensive. One typically sets pre-defined model times at which the code should output a frame. The time stepping is then done using the usual CFL time step limit, but if the time is about to jump over one of the pre-defined save-time, then one shortens the time step such that the precise save time is reached. Then a dump of the current state is made, and the simulation is continued.

3. Especially for 2-D and 3-D simulations a compact file format is useful. A simple and reasonably compact way is to use fortran-style unformatted output. However, one should keep in mind that on some platforms the byte-order is swapped. If a frame written on one

computer looks unintelligible on another computer, then one may want to look into “endian swapping”. In IDL there is a keyword in the file management routines to automatically swap endian. Another, quite popular method of storage is the HDF format. It is very portable to other platforms and is particularly suited for large data volumes.

4. For safety, in particular for large multi-dimensional applications, one might want to set a maximum number of time steps, so that if a simulation gets stuck, it won’t continue to eat up all CPU time. Also, for similar safety concerns one might wish to set a maximum nr of steps the code can do without writing a safety dump of its variables. And finally, it is useful to make the code dump the entire state as it was before the time step if an unrecoverable error occurs.