4th Workshop in Virtual Reality Interactions and Physical Simulation "VRIPHYS" (2007)
J. Dingliana, F. Ganovelli (Editors)

A Fast and Compact Solver for the Shallow Water Equations

Richard Lee and Carol O'Sullivan

Graphics, Vision and Visualisation Group, Trinity College Dublin, Ireland

Abstract

This paper presents a fast and simple method for solving the shallow water equations. The water velocity and height variables are collocated on a uniform grid and a novel, unified scheme is used to advect all quantities together. Furthermore, we treat the fluid as weakly compressible to avoid solving a pressure Poisson equation. We sacrifice accuracy and unconditional stability for speed, but we show that our algorithm is sufficiently stable and fast enough for real-time applications.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

1. Introduction

The shallow water equations are a simplified version of the 3D Navier-Stokes equations, suitable for animating heightfield-based bodies of liquid such as ponds, lakes or oceans. There are two equations. The first describes conservation of mass and the second describes conservation of momentum:

$$\frac{Dh}{Dt} = -h\nabla \cdot \vec{u} \tag{1}$$

$$\frac{D\vec{u}}{Dt} = -g\nabla h \tag{2}$$

Here, h is the height of the water above ground level, $\vec{u}$ is velocity and g is gravity. Note that, for simplicity, we have assumed that the ground has flat topography in these equations. Our method easily handles variable height ground however.

The form of these equations is nearly identical to the inviscid, 2D Navier-Stokes equations with no body forces:

$$\frac{D\rho}{Dt} = -\frac{1}{\rho}\nabla \cdot \vec{u} \tag{3}$$

$$\frac{D\vec{u}}{Dt} = -\frac{1}{\rho}\nabla p \tag{4}$$

where ρ is density and p is pressure. By considering the fluid to be slightly compressible we can introduce an artificial equation of state, $p = c^2\rho$, where c is the speed at which pressure waves propagate through the liquid. The pressure gradient in equation (4) now becomes a density gradient. In this form, all of the derivatives in both pairs of equations match and solvers developed for one pair of equations can generally be applied to the other pair will little modification.

In our solver, we further simplify equations (1) and (3) by assuming that the fluid is nearly incompressible such that the divergence term, $\nabla \cdot \vec{u}$, can be treated as negligible. Hence, we set the right hand side of equations (1) and (3) to zero. An additional benefit of this is that, provided a conservative scheme is used for advection, the total height/mass will be constant. In practice, we find it difficult to tell the difference between results which include and exclude this term. In this form, h corresponds to ρ and g corresponds to $c^2 \big/ \rho$.

Our method solves equations (3) and (4) and then computes h by linearly scaling ρ. This corresponds to the assumption of hydrostatic pressure, which the shallow water equations are built on.

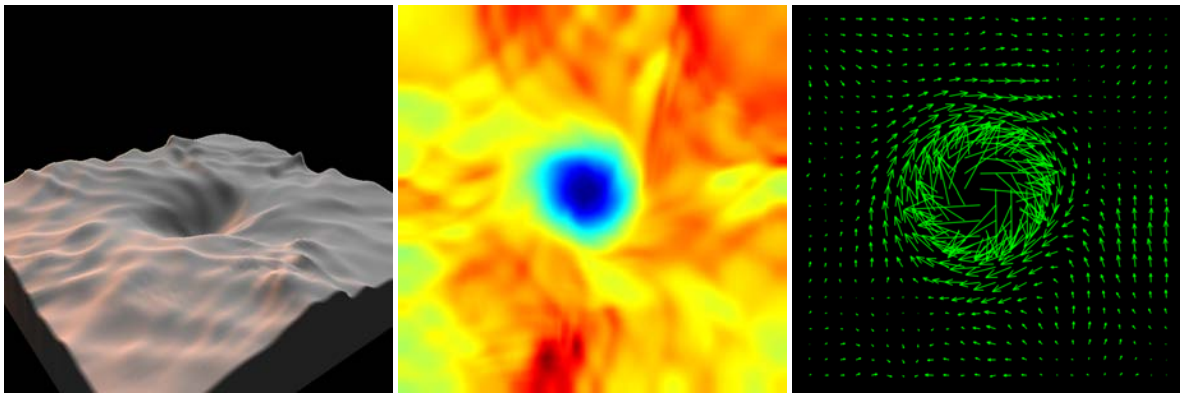

Figure 1: *Shallow water simulation on a* 100×100 *grid. On the left, the user interactively applies forces to the fluid to form a whirlpool. The corresponding density and velocity fields are shown on the right.*

2. Previous Work

There are two popular approaches for animating bodies of water in modern interactive applications. Both are based on the manipulation of heightfields, since computing solutions to the 3D Navier-Stokes equations is generally too slow.

The first approach uses procedural animation techniques to perturb the heights of vertices in a grid. These perturbations correspond to the propagation of water waves which are superimposed on each other, usually using Fast Fourier Transforms [Tes04, HNC02, FR86, Pea86, TB87, Sch80]. In particular, the technique of Tessendorf [Tes04] is very popular since it uses a statistical model from oceanography to describe the wave properties and models a dispersion relation so that different waves can travel at different speeds. Furthermore, Tessendorf presents a way to displace vertices in the horizontal plane to make the water appear choppy instead of calm. While procedural techniques can produce highly realistic results, they suffer from the major disadvantage of not being able to respond to outside influence, such as a disturbance from an object interacting with the water.

The second approach uses simulation to compute the height perturbations in the water plane. These methods are usually based on either the shallow water equations [LvdP02, TRS06], which can be derived from the Navier-Stokes equations, or the simpler wave equations [BMFGF06, OH95, KM90, Tes04], which are themselves a simplification of the shallow water equations. All of these methods have the big advantage that they allow surface disturbances to affect the behavior of the water. Furthermore, the shallow water equations have many important advantages over the wave equations: they account for the advection of fluid which allows it to flow due to its horizontal momentum; they take ground topography into account; and they can simulate the flooding of previously dry areas. However, since they do not model the velocity field within the water volume, they cannot model the dispersion relation which hinders realism. Still, for real-time applications, these methods are the most frequently used solution for animating water.

3. Fluid Solver

3.1. Overview

We consider an algorithm to solve the coupled equations, (3) and (4). Traditionally, this requires computing a mass-conserving pressure field by solving a Poisson equation. This is a matrix equation that relates pressure to divergence and is generally the bottleneck in most fluid simulators, accounting for as much as 50% of the total time in our experience. The use of an artifical equation of state relating fluid density to pressure alleviates us from having to solve the Poisson equation. Even though we are no longer solving for an incompressible fluid, we believe that compressibility in the context of shallow water simulation does not adversely affect visual quality, especially considering our method still achieves global mass and volume conservation. The main computational burden now rests on evaluating the material derivatives on the left hand sides of equations (3) and (4).

The overall structure of our fluid simulator is very simple. At the beginning of each iteration of the time-stepping loop, the velocities and densities of the fluid can be modified, for example, to simulate the effect of wind or rain on the water surface. Next, the advection step transports $\vec{u}$ and ρ along the velocity field from time t^n to time t^{n+1}. Finally, in the pressure step, the pressure gradient is calculated by scaling the density gradient, and this is then subtracted from the velocity field to give $\vec{u}$ at time t^{n+1}. At the end of the update, the new heightfield can be calculated by simply scaling the density field. Appendix A contains a listing of the C++ code which implements the entire update routine for the 1D equations.

The next section discusses the grid structure we use to arrange state variables. This is followed by a description of the

advection and pressure steps which are invoked each iteration to update the fluid's state.

3.2. Grid Configuration

The state of our fluid is defined by its 2D velocity and density fields (e.g. see Figure 1). These fields need to be sampled at a finite set of points in space. Usually a uniform grid is used for this purpose, where horizontal/vertical components of velocity are located at the centre of vertical/horizontal cell faces respectively, and density/pressure variables are located at cell centres. This staggered configuration avoids the classic "checkerboard" problem [Min96], that arises from decoupling between the pressure and velocity fields, since the location of pressure gradients naturally coincide with velocity variables, and the location of velocity gradients coincide with pressure variables.

In our solver, we abandon the staggered grid in favor of a collocated arrangement where all variables are located at the centre of grid cells (see Figure 2). This configuration has been shown to suffer from instabilities and spurious oscillations in the pressure field. Specifically, central difference approximations to pressure and velocity gradients will not use adjacent nodes in the grid and, as a consequence, pressure fields can be calculated which satisfy the discrete equations but *do not* satisfy the continuous equations. In other words, non-physical pressure values can result.

However, we expect that large computational savings can be made if the collocated scheme can be stabilised to avoid decoupling, even if this means compromising physical accuracy. Such a tradeoff may be unacceptable in other scientific fields, but our goal is to produce plausibly realistic animation as fast as possible. Furthermore, the collocated configuration is less geometrically complex and easier to understand, implement and debug, which makes it attractive from a practical standpoint.

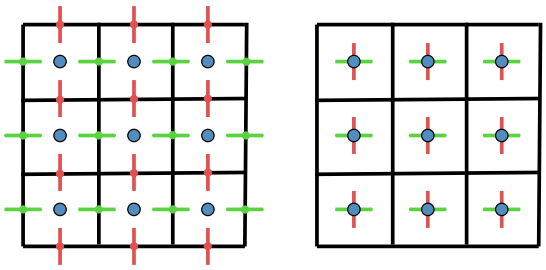

Figure 2: *Location of horizontal velocities (green), vertical velocities (red) and densities (blue) for a staggered grid configuration (left), and a collocated configuration (right).*

3.3. Advection Step

Advecting $\vec{u}$ and ρ is, by far, the most expensive part of the simulation algorithm, accounting for at least 90%

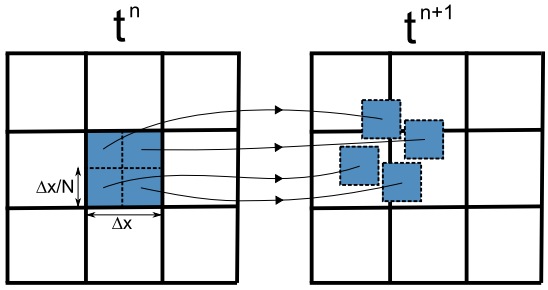

Figure 3: *All cell variables are advected together by forward-tracing $N \times N$ packets, where $N = 2$ in this example.*

of the run-time. Normally the semi-Lagrangian advection scheme [Sta99] is used in fluid animation because it is fast and remains stable irrespective of the time-step size. The downside of this scheme is that it introduces dissipative error due to interpolating the new velocity field from the old one. This tends to smear the fine details in the flow, which requires an additional technique such as vorticity confinement [FSJ01] to counteract. In our experience, the semi-Lagrangian method performs very well for velocity advection. However, the numerical diffusion incurred by interpolation performs poorly for advecting density, and we found that the divergence term we omitted from equation (3) was necessary to get this to work. A bigger problem is that the semi-Lagrangian method is not conservative, meaning that mass loss will occur when th is scheme is used to advect density.

To overcome this, we propose a new conservative advection scheme which is sufficiently stable for our requirements and, similar to the semi-Lagrangian method, avoids a time-step restriction due to cell size. While the semi-Lagrangian method works by tracing the trajectories of particles upstream (against the flow), our scheme works by tracing particles downstream (with the flow). More specifically, let us consider the advection of density from a grid cell of area $(\Delta x)^2$, whose density is ρ_{ij}. We partition the cell into an $N \times N$ subgrid (N determines the accuracy of the advection scheme) and, for each subcell, we trace a packet of area $\left(\Delta x/N\right)^2$ from the centre of the subcell, forward along the velocity field, and deposit this packet at its final location. This is illustrated in Figure 3. We use either Euler's method or the midpoint method for tracing packets and we "bounce" packets off of colliding obstacles to prevent mass from clumping in regions adjacent to boundaries. To deposit the packet, we overlay it on top of the grid and compute the area of overlap with each of the cells it intersects. Each of these overlapping cells is then updated by adding an area-weighted fraction of ρ_{ij} to that cell's density accumulator variable. After all density packets from all cells have been traced, the accumulator variables correspond to the new density field. It is clear that this scheme conserves mass – the to-

tal density before advection will equal the total density after advection.

Our approach uses this advection scheme, not only for density, but also for velocity. This is very convenient in the context of collocation since all variables (density and the two components of velocity) can be traced together using the same packets. This saves considerable computation as compared to performing semi-Lagrangian advection separately for velocity. Note that, unlike mass, our advection scheme does not conserve momentum. This is because, when we perform accumulation of velocities, positive and negative velocities will be summed which will cancel each other out, resulting in momentum loss. Momentum loss occurs when using the semi-Lagrangian method for the same reason – interpolation between velocities causes cancellation between positive and negative values.

3.4. Pressure Step

In this stage of the algorithm, we subtract the scaled density gradient from the velocity, as in equation (4). The natural way to compute this gradient is using central differences. However, as discussed in section 3.2, it is generally unstable to use central differences in conjunction with a collocated configuration due to the checkerboard problem.

We tried a number of different solutions to fix this. One approach is to average the cell-centred velocities to cell faces, then subtract the pressure gradient from the face velocities and finally average these velocities back to the cell centres. This was successful in avoiding the stability issues. However, averaging the velocities back and forth causes significant smoothing of the velocity field (even more than the semi-Lagrangian method) which produces unsatisfactory results. Note that, to reduce dissipation, it may be tempting to avoid averaging the entire velocity from cell faces to cell centres, and instead only average the *change* in velocity [GSLF05]. However, this change in velocity is equal to the pressure gradient and averaging this gradient from the faces to the centre of a cell is equivalent to performing a central difference approximation of the gradient in the first place.

There is also a technique called Rhie-Chow interpolation [RC83] which defines a way to interpolate velocities to cell faces such that the checkerboard instability is avoided. We did not consider implementing this technique since the interpolation scheme is considerably more expensive than linear interpolation. However, it can be shown that this approach is equivalent to adding a pressure smoothing term, the effect of which is to eliminate pressure oscillations. Motivated by this, we tried simply to smooth the pressure field (while using the central difference pressure gradient approximation) and found that this was sufficient to avoid the checkerboard problem. Conveniently, this smoothing can be performed as part of the advection step (described in Section 3.3) at negligible additional cost. This is done by simply scaling the size of each packet by a small amount so that it covers a slightly larger area. So, instead of an area of $(\Delta x/N)^2$, the packet now has an area of $(s\Delta x/N)^2$, where values of s greater than 1 result in smoothing. To compensate for this increase in packet size, the quantities carried by the packet need to be scaled down by $1/s^2$. This smoothing does introduce noticable artificial dissipation to the velocity field which is undesirable. However, given the performance benefits of using a collocated grid versus a staggered grid, we consider this tradeoff to be worthwhile.

Figure 4: *1D adaptive shallow water simulation. Notice that the height changes smoothly over the boundary between cells of different size.*

4. Adaptive Simulation

One advantage of our compressible fluid solver is that the same methods map conveniently to adaptive and unstructured grids. It is traditionally very complex to implement incompressible solvers on adaptive grids [LGF04, Pop03] due to difficulties in making convergence of the Poisson solver competitive with uniform grids. However, our scheme can be adapted easily to new grid types. All that is required in the ability to compute the overlap between packets and grid cells, and a consistent way to evaluate density gradients at the centres of cells.

To verify this, we have implemented an adaptive 1D version of our solver (see Figure 4). The procedure to redistribute density packets across the boundary between cells of different size is trivial. However, computing the density gradients is slightly more complex. Apart from the usual central difference case, there are two cases to consider: evaluating the gradient in a cell of size Δx when the two neighboring cells have sizes Δx and $2\Delta x$; and evaluating the gradient in a cell of size $2\Delta x$ with neighbors of sizes $2\Delta x$ and Δx. We compute approximations to these gradients by fitting a parabola $y = ax^2 + bx + c$, through the centres of the three cells, and evaluating its gradient, $dy/dx = 2ax + b$, at the centre cell. Note that, while it may seem like this approximation is unnecessary, it is not much more expensive than

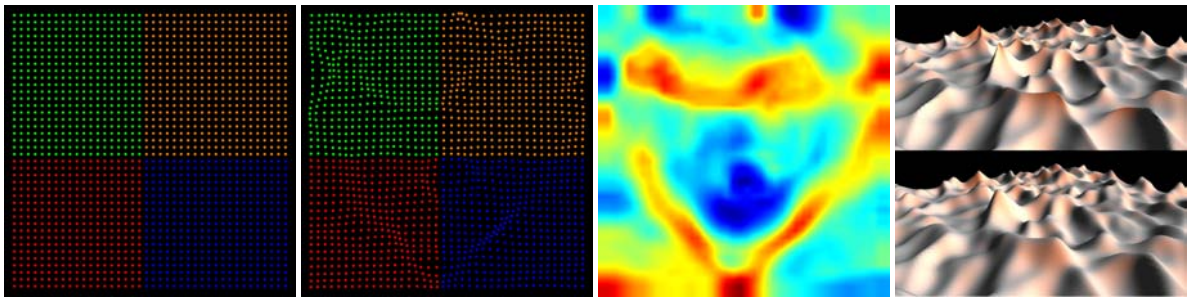

Figure 5: *From left to right: normal in-plane grid vertices; in-plane grid vertices deformed by pressure gradient; corresponding pressure field; 3D heightfield with (top) and without (bottom) choppy wave modification.*

central differencing and wa s found to be the only solution which gave smooth transitions across the boundary between different sized cells. We also tried the gradient approximations proposed in [LGF04] but found that these approximations produced small, spurious waves which reflect off the adaptive boundary. These artifacts did not appear when the parabolic approximation was used.

We are currently in the process of implementing an adaptive solver for the 2D case. Our strategy is to tile uniform grids of varying resolution, instead of using a quadtree as is more common. In our experience, implementing fluid solvers on quadtrees/octrees, particularly when using a staggered configuration of variables, is very complex and difficult to make efficient. Uniform grids in comparison have a number of advantages. They perform better, they are more memory and cache efficient, they are much quicker to query and update, and they are easier to parallelize. For performance critical applications, we suspect that adaptive quadtree refinement is overkill compared to using tiled uniform grids and we will investigate this claim in future work.

5. Results

The results of our fluid simulation are converted to a heightfield as described in Section 1 and visualized in OpenGL. Although not physically based, we displace the grid vertices in the horizontal plane to make the water appear choppy. We experimented with using velocity and pressure gradient information to compute the displacements but we achieved the best results by simply displacing the vertex coordinates from (x,z) to $\left(x+cdp/dx, z+cdp/dz\right)$, where p is the pressure computed in Section 3.4, and c is a scale factor which controls the degree of choppiness. It is also useful to clamp the gradient to avoid excessively large displacements which would result in mesh tangling. We achieved good results with this simple approach, but note that more advanced filters for the gradient terms could be designed to achieve nicer-looking effects. The results of this modification are shown in Figure 5.

We evaluated the speed of our serial solver for different grid sizes on a Xeon 3.6GHz PC using 2×2 packets per cell in all cases. The timings are as follows:

No. cells	25^2	50^2	100^2	200^2	400^2
Updates/sec	3185.7	736.0	177.5	43.7	10.2

We also note that, compared to incompressible fluid solvers, our method is very easy to parallize. Our current approach is to subdivide the uniform grid into subgrids. Each subgrid is operated on in parallel. When packets need to be transferred across subgrid boundaries they can either be added to a list which is later processed when safe to do so, or locks can be used to protect cells near subgrid boundaries. So far, we have only been able to test the former algorithm on a dual-core machine. For large grid sizes (e.g. 250×250), we achieve speedups of about 1.8x. Although we have parallelized the entire simulation update routine, the parallel efficiency drops significantly for small grid sizes. We have not yet investigated the reason for this loss of efficiency but we believe that excellent parallel scaling can be extracted from the algorithm.

6. Conclusions and Future Work

In this paper we have discussed how stability and efficiency concerns affect the choice of grid configuration, advection scheme and pressure gradient computation for a compressible fluid solver. We have presented an efficient technique for advecting velocity and density together on a collocated grid and we discussed how the pressure instabilities can be avoided by smoothing the density field during advection. We also discussed how the solver can be parallelized easily, and modified to work on multiresolution grids. Finally, we presented a cheap modification to augment the visual quality of heightfield water.

Much remains to be done as future work. Of primary importance is a proper evaluation of the accuracy, stability and efficiency of the presented advection scheme and how overall performance compares to using the semi-Lagrangian method on staggered and collocated grids. Further investigation into the effect of smoothing the density field should

also be performed to see if this can be improved using more advanced filters to "blur" the packets.

Finally, it would be interesting to map this algorithm to graphics hardware and let the GPU perform the overlap tests and accumulative advection step using rasterisation and blending. This could be achieved by using textures which are larger than the grid dimensions to perform the advection step and downsampling the result back to a texture whose dimensions are the same as the grid.

References

[BMFGF06] BRIDSON R., MÜLLER-FISCHER M., GUENDELMAN E., FEDKIW R.: Fluid simulation course notes. In *SIGGRAPH 2006* (2006).

[FR86] FOURNIER A., REEVES W. T.: A simple model of ocean waves. *SIGGRAPH Comput. Graph. 20*, 4 (1986), 75–84.

[FSJ01] FEDKIW R., STAM J., JENSEN H. W.: Visual simulation of smoke. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2001), ACM Press, pp. 15–22.

[GSLF05] GUENDELMAN E., SELLE A., LOSASSO F., FEDKIW R.: Coupling water and smoke to thin deformable and rigid shells. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers* (New York, NY, USA, 2005), ACM Press, pp. 973–981.

[HNC02] HINSINGER D., NEYRET F., CANI M.-P.: Interactive animation of ocean waves. In *SCA '02: Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation* (New York, NY, USA, 2002), ACM Press, pp. 161–166.

[KM90] KASS M., MILLER G.: Rapid, stable fluid dynamics for computer graphics. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1990), ACM Press, pp. 49–57.

[LGF04] LOSASSO F., GIBOU F., FEDKIW R.: Simulating water and smoke with an octree data structure. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers* (New York, NY, USA, 2004), ACM Press, pp. 457–462.

[LvdP02] LAYTON A. T., VAN DE PANNE M.: A numerically efficient and stable algorithm for animating water waves. *The Visual Computer 18*, 1 (2002), 41–53.

[Min96] MINION M. L.: A projection method for locally refined grids. *J. Comput. Phys. 127* (1996), 158–177.

[OH95] O'BRIEN J. F., HODGINS J. K.: Dynamic simulation of splashing fluids. In *Computer Animation '95* (1995), pp. 198–205.

[Pea86] PEACHEY D. R.: Modeling waves and surf. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1986), ACM Press, pp. 65–74.

[Pop03] POPINET S.: Gerris: a tree-based adaptive solver for the incompressible euler equations in complex geometries. *J. Comput. Phys. 190*, 2 (2003), 572–600.

[RC83] RHIE C., CHOW W.: Numerical study of the turbulent flow past an airfoil with trailing edge separation. In *AIAA Journal, 21* (1983), pp. 1525–1532.

[Sch80] SCHACHTER B. J.: Long crested wave models. 187–201.

[Sta99] STAM J.: Stable fluids. In *Siggraph 1999, Computer Graphics Proceedings* (Los Angeles, 1999), Rockwood A., (Ed.), Addison Wesley Longman, pp. 121–128.

[TB87] TS'O P. Y., BARSKY B. A.: Modeling and rendering waves: wave-tracing using beta-splines and reflective and refractive texture mapping. *ACM Trans. Graph. 6*, 3 (1987), 191–214.

[Tes04] TESSENDORF J.: Simulating nature course notes. In *SIGGRAPH 2004* (2004).

[TRS06] THÜREY N., RÜDE U., STAMMINGER M.: Animation of open water phenomena with coupled shallow water and free surface simulations. In *SCA '06: Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation* (Aire-la-Ville, Switzerland, Switzerland, 2006), Eurographics Association, pp. 157–164.

Appendix A

The C++ listing below is code to update the velocity (*vels*) and density (*rho*) arrays for a 1D fluid solver. *N* is the number of grid cells. The first and last cell are assumed to be boundary cells and are not considered. *DX* is the cell size. *DT* is the time-step size. *DIV* is the number of packets per cell. *SMOOTHING* is a value in $[1,2]$ that is used to scale packet dimensions. *CSQ* is a constant which is used to control compressibility.

```
const int DIV = 3;
const double SMOOTHING = 1.05;
const double HALFPKT = 0.5*(SMOOTHING/DIV);
const double LFTWALL = (1+HALFPKT)*DX+1e-12;
const double RGTWALL = N*DX - LFTWALL;

memset(newVels, 0, N*sizeof(double));
memset(newRho, 0, N*sizeof(double));
for(int i = 1; i < N-1; ++i)
  for(int j = 0; j < DIV; ++j)
  {
    x = (i + (j+0.5)/DIV) * DX;
    gx = x/DX - 0.5;
    idx = int(gx);
    fx = gx - idx;
    u = (1-fx)*vels[idx] + fx*vels[idx+1];
```

```
    nx = Clamp(x+DT*u, LFTWALL, RGTWALL);
    left = nx/DX - HALFPKT;
    right = nx/DX + HALFPKT;
    li = int(left);
    ri = int(right);
    f1 = (ri - left) / SMOOTHING;
    f2 = (right - ri) / SMOOTHING;
    newRho[li] += rho[i] * f1;
    newRho[ri] += rho[i] * f2;
    newVels[li] += vels[i] * f1;
    newVels[ri] += vels[i] * f2;
  }
memcpy(vels, newVels, N*sizeof(double));
memcpy(rho, newRho, N*sizeof(double));

double s = CSQ * DT / DX * 0.5;
vels[1] -= (rho[2] - rho[1]) * s;
for(int i = 2; i < N-2; ++i)
  vels[i] -= (rho[i+1] - rho[i-1]) * s;
vels[N-2] -= (rho[N-2] - rho[N-3]) * s;
```