A Pedagogical Tool Using Smoothed Particle Hydrodynamics to Model Fluid Flow Past a System of Cylinders.

Brian Schlatter

June 15, 1999

Contents

Part I
Introduction

The study of fluid flow problems is extremely important since almost everything on this planet is either a fluid or moves with respect to a fluid. Some common engineering examples are pumps, fans, turbines, airplanes, ships, rivers, windmills and pipes. To aid in understanding fluid flow problems, engineers often use either experimental simulations or computational simulations along with a mathematical theory. Two methods used to describe fluid flow are the Eulerian and the Lagrangian approach. [White 1994]

One way to think of the Eulerian method is to think of the observer as standing on the sidelines watching the fluid move past the area of interest. The Lagrangian method, on the other hand, has the observer moving along with the flow. The Eulerian approach is also called a grid method, or a finite difference method, while the Lagrangian approach is called a particle method.

The model we use in this project is called Smoothed Particle Hydrodynamics (SPH) and is considered a pure Lagrangian method [Libersky 1993]. SPH uses interpolating nodes in order to determine field values within the fluid. A field value is any physical characteristic of the fluid (*e.g* density, velocity, pressure.). Each node has a mass associated with it and then can be considered a physical fluid particle and hence its field values will vary in accordance to the conservation laws of continuum fluid mechanics. Each node represents an average of its immediate area. It determines this value by performing a weighted sum over its neighboring nodes' values.

One reason to use SPH over other numerical methods such as Particle in Cell (PIC), or Finite Difference Methods (FDM) methods is that SPH does not depend on the boundary conditions. The FDM is based on breaking up the simulation region into a grid. The reason for creating a grid is that in order to calculate the spatial derivatives of the field variables, one must divide the difference of the field variables at two points by the distance between these two points. Thus, to improve accuracy, the distance between the points in which the field variables are known is decreased. If this distance is decreased for the entire region and the field variables change drastically only over a small region, then computational time is wasted. A solution to breaking the entire region into grid with a small mesh size is to make the grid finer only in the regions in which field variables will change the most. The problem is that we must know where these regions are *a priori*. Thus, if we change an obstacle's position in the flow from one place to another, then we must change our grid to correspond.

In contrast, the SPH method does not deal with a grid at all. The way in which it finds spatial derivatives is by moving this differentiation from the field variables onto a function called the "kernel". The kernel does not depend on where the obstacles are in the simulation. The details of this are explained in the Theory section.

Another reason to use SPH is that it easily extends to three dimensions and is robust in complicated and unexpected situations. The goal of this project is to produce a usable, robust, pedagogical tool for the fluid mechanics students and faculty at Oregon State University. The tool will allow its users to modify physical values pertinent to the fluid system as well as add certain two dimensional objects in the flow itself. The tool will produce an animation of how the fluid interacts with the chosen objects. In addition to providing the following powerful features, we wanted it to be easily extendible to more complicated systems.

1. Adding N objects to the scene
2. Varying the viscosity of the fluid
3. Varying the flow rate (*i.e.* velocity, external pressure gradient)
4. Displaying density contour plots and velocity fields
5. Modifying the number of particles in the simulation

The next section discusses the theory of Smoothed Particle Hydrodynamics. This includes deriving the equations of motion in SPH form, discussing the kernel, smoothing length, artificial viscosity and the method of incorporating boundary conditions. Section 3 mentions other work related to this project and discusses what we have taken from each. Section 4, the Analysis and Design section, talks about each of our design properties and how we decided to design in order to satisfy them. Section 5 covers the architecture of our program, including a discussion of how the Server and Client are broken up into their class structure. Section 6 describes each of our versions of this program with an emphasis on data structures, algorithms and bugs. Section 7 contains a description of the test and its results. Section 8 restates our goals and discusses which ones were met and which ones were not met. It also describes what still needs to be done in order to fulfill all of the design properties. Section 9 explains where to find a working version of the program and how to run it.

Part II
Theory

1 Introduction

The Smoothed Particle Hydrodynamics (SPH) technique created by Lucy [Lucy 1977] and Monaghan & Gingold [Monaghan & Gingold 1977], was originally developed to simulate astrophysical processes. Since then it has been used in various fluid dynamical systems including viscous flows [Takeda *et al.* 1994] and free surface flows [Monaghan 1994], as well as low Reynolds number, incompressible flow [Morris 1997].

As we noted in the introductory, a method is termed Lagrangian if the fluid is described with particles. The units in the SPH method represent a region of space and not a physical particle. Thus, we term these pseudo-particles. Field variables at a point in space are found by averaging, or smoothing, the field variables of pseudo-particles over the region of interest. Spatial derivatives of a field variable are not calculated using a grid such as in Finite Difference Methods. Instead, the interpolating formula themselves are analytically differentiated. SPH is often described as a rugged, robust technique that is easily extended to N dimensions.

Most of its advantages are directly related to its Lagrangian nature. There are no constraints imposed either on the geometry of the system or in how far it may evolve from the initial conditions. The number density of the particles maps directly to the mass density of the fluid, leading directly to increased resolution in high density regions.

2 Fundamentals

The foundation of the Smoothed Particle Hydrodynamics method is interpolation theory. The conservation laws of continuum fluid dynamics are transformed into integral equations through the use of an interpolation function that gives the "kernel estimate" of the field variable at a point [Libersky 1993]. The term "kernel" refers to a weighting function and defines how much of each field variable contributes to the field variable at the point $\mathbf{r}$. The kernel estimate of $A(\mathbf{r})$ is defined by

$$\langle A(\mathbf{r}) \rangle = \int_{Space} A(\mathbf{r}')W(\mathbf{r}-\mathbf{r}', h)\, d\mathbf{r}' \tag{1}$$

where the function $W(\mathbf{r}-\mathbf{r}', h)$ is a kernel with the following two properties:

$$\int_{Space} W(\mathbf{r}-\mathbf{r}', h)\, d\mathbf{r}' = 1 \tag{2}$$

and

$$\lim_{h\to 0} W(\mathbf{r}-\mathbf{r}', h) = \delta(\mathbf{r}-\mathbf{r}') \tag{3}$$

If $A(\mathbf{r}')$ is known only at a discrete set of N points $\mathbf{r}_1, \mathbf{r}_2, ..., \mathbf{r}_N$, then we approximate $A(\mathbf{r}')$ as,

$$A(\mathbf{r}') = \sum_{j=1}^{N} \delta(\mathbf{r}'-\mathbf{r}_j)A(\mathbf{r}_j)(dV)_j \tag{4}$$

where $(dV)_j$ is the differential volume element around the point $\mathbf{r}_j$. When we insert equation (4) into equation (1), we get,

$$\langle A(\mathbf{r}) \rangle = \sum_{j=1}^{N} \int \delta(\mathbf{r}'-\mathbf{r}_j)A(\mathbf{r}_j)(dV)_j W(\mathbf{r}-\mathbf{r}', h)\, d\mathbf{r}' \tag{5}$$

and upon integrating, we have:

$$\langle A(\mathbf{r}) \rangle = \sum_{j=1}^{N} A(\mathbf{r}_j)W(\mathbf{r}-\mathbf{r}_j, h)(dV)_j \tag{6}$$

The final step is to express the differential volume element $(dV)_j$ as $\frac{m_j}{\rho(\mathbf{r}_j)}$. Thus, our approximation for $A(\mathbf{r})$ can be written as:

$$\langle A(\mathbf{r}) \rangle = \sum_{j=1}^{N} \frac{m_j}{\rho(\mathbf{r}_j)} A(\mathbf{r}_j)W(\mathbf{r}-\mathbf{r}_j, h) \tag{7}$$

The error of equation (7) in approximating equation (1) depends on how disordered the points are. This is discussed further in the section covering kernels. We move each particle with the acceleration an element of fluid experiences at the position of the particle.

We now show an example of how one finds the smoothed version of the density:

$$
\begin{aligned}
\langle \rho(\mathbf{r}) \rangle &= \sum_{j=1}^{N} \frac{m_j}{\rho(\mathbf{r}_j)} \rho(\mathbf{r}_j) W(\mathbf{r} - \mathbf{r}_j, h) \\
&= \sum_{j=1}^{N} m_j W(\mathbf{r} - \mathbf{r}_j, h) \qquad (8)
\end{aligned}
$$

Next, we discuss how one would find the smoothed gradient of a field variable. We proceed in a similar fashion in going from equation (1) to equation (7):

$$\langle \nabla_{\mathbf{r}} A(\mathbf{r}) \rangle = \int \nabla_{\mathbf{r}'} A(\mathbf{r}') W(\mathbf{r} - \mathbf{r}', h)\, d\mathbf{r}' \qquad (9)$$

where $\nabla_{\mathbf{r}'}$ implies integration with respect to the primed coordinates. If we integrate by parts with,

$$
\begin{aligned}
u &= W(\mathbf{r} - \mathbf{r}', h) \Rightarrow du = \nabla_{\mathbf{r}'} W(\mathbf{r} - \mathbf{r}', h) d\mathbf{r}' \\
dv &= \nabla_{\mathbf{r}'} A(\mathbf{r}') d\mathbf{r}' \Rightarrow v = A(\mathbf{r}')
\end{aligned}
\qquad (10)
$$

we obtain:

$$\langle \nabla_{\mathbf{r}} A(\mathbf{r}) \rangle = A(\mathbf{r}') W(\mathbf{r} - \mathbf{r}', h)|_{Surface} - \int A(\mathbf{r}') \nabla_{\mathbf{r}'} W(\mathbf{r} - \mathbf{r}', h)\, d\mathbf{r}' \qquad (11)$$

Equation (11) has the gradient with respect to primed coordinates. All papers have the gradient with respect to unprimed coordinates. In making this conversion we get a -1 out. We will briefly go through the steps that verify this change of variable for the gradient operator. One of the properties of our kernel is that it depends only on the distance between the points **r** and **r**',

$$W(\mathbf{r} - \mathbf{r}', h) = N f\left(\frac{|\mathbf{r} - \mathbf{r}'|}{h}\right) \qquad (12)$$

where N is a normalization constant. If we take the gradient of f with respect to primed coordinates, we get

$$\nabla_{\mathbf{r}'} f\left(\frac{|\mathbf{r} - \mathbf{r}'|}{h}\right) = \hat{i} \frac{\partial}{\partial x'} f\left(\frac{|\mathbf{r} - \mathbf{r}'|}{h}\right) + \hat{j} \frac{\partial}{\partial y'} f\left(\frac{|\mathbf{r} - \mathbf{r}'|}{h}\right) \qquad (13)$$

$$\nabla_{\mathbf{r}'} f\left(\frac{|\mathbf{r}-\mathbf{r}'|}{h}\right) = \hat{\imath}\left[\frac{(x-x')(-1)}{h\,|\mathbf{r}-\mathbf{r}'|} f'\left(\frac{|\mathbf{r}-\mathbf{r}'|}{h}\right)\right] + \hat{\jmath}\left[\frac{(y-y')(-1)}{h\,|\mathbf{r}-\mathbf{r}'|} f'\left(\frac{|\mathbf{r}-\mathbf{r}'|}{h}\right)\right] \tag{14}$$

where f' implies differentiation with respect to $|\mathbf{r}-\mathbf{r}'|$. The next step is then,

$$\nabla_{\mathbf{r}'} f\left(\frac{|\mathbf{r}-\mathbf{r}'|}{h}\right) = \frac{-1}{|\mathbf{r}-\mathbf{r}'|} f'\left(\frac{|\mathbf{r}-\mathbf{r}'|}{h}\right)\left(\hat{\imath}(x-x') + \hat{\jmath}(y-y')\right) \tag{15}$$

and if we take the gradient with respect to the unprimed coordinates, we get:

$$\nabla_{\mathbf{r}} f\left(\frac{|\mathbf{r}-\mathbf{r}'|}{h}\right) = \hat{\imath}\frac{\partial}{\partial x} f'\left(\frac{|\mathbf{r}-\mathbf{r}'|}{h}\right) + \hat{\jmath}\frac{\partial}{\partial y} f\left(\frac{|\mathbf{r}-\mathbf{r}'|}{h}\right) \tag{16}$$

$$\nabla_{\mathbf{r}'} f\left(\frac{|\mathbf{r}-\mathbf{r}'|}{h}\right) = \hat{\imath}\left[\frac{(x-x')(+1)}{h\,|\mathbf{r}-\mathbf{r}'|} f'\left(\frac{|\mathbf{r}-\mathbf{r}'|}{h}\right)\right] + \hat{\jmath}\left[\frac{(y-y')(+1)}{h\,|\mathbf{r}-\mathbf{r}'|} f'\left(\frac{|\mathbf{r}-\mathbf{r}'|}{h}\right)\right] \tag{17}$$

$$\nabla_{\mathbf{r}'} f\left(\frac{|\mathbf{r}-\mathbf{r}'|}{h}\right) = \frac{+1}{|\mathbf{r}-\mathbf{r}'|} f'\left(\frac{|\mathbf{r}-\mathbf{r}'|}{h}\right)\left(\hat{\imath}(x-x') + \hat{\jmath}(y-y')\right) \tag{18}$$

Thus, we see that equations (15) and (18) only differ by the constant -1. With this relation and the assumption that the kernel, the variable, or both, go to zero at infinity, we can write equation (11) as:

$$\langle \nabla A(\mathbf{r}) \rangle = \int A(\mathbf{r}')\nabla_{\mathbf{r}} W(\mathbf{r}-\mathbf{r}', h)\, d\mathbf{r}' \tag{19}$$

The surface term does not necessarily vanish for finite systems. For these cases we must deal with the boundary explicitly. Many approaches have been explored by other authors. The approach we use in this project was modeled after the one by Morris [Morris 1997], in which smoothed particles make up the boundary and are included in the interaction with normal particles.

We are now ready to write down our smoothed gradient as a summation. Similar to equation (7), we write:

$$\langle \nabla_{\mathbf{r}} A(\mathbf{r}) \rangle = \sum_{j=1}^{N} \frac{m_j}{\rho(\mathbf{r}_j)} A(\mathbf{r}_j)\nabla_{\mathbf{r}} W(\mathbf{r}-\mathbf{r}_j, h) \tag{20}$$

This equation is one of the main reasons SPH is so popular. It removes the need for a mesh to calculate spatial derivatives. With equations (7) and (20), we are ready to write down our numerical equations for the smoothed values of each point. From here on, we will remove the brackets and place a subscript i on the value. This implies that the value being calculated is the smoothed value for particle i. The rewritten version of our two SPH equations are then:

$$\langle A(\mathbf{r})\rangle \equiv A_i = \sum_{j=1}^{N} \frac{m_j}{\rho_j} A_j W_{ij} \tag{21}$$

and

$$\langle \nabla_{\mathbf{r}} A(\mathbf{r})\rangle \equiv \nabla_i A_i = \sum_{j=1}^{N} \frac{m_j}{\rho_j} A_j \nabla_i W_{ij} \tag{22}$$

where $W_{ij} = W(\mathbf{r}_i - \mathbf{r}_j, h)$ and ∇_i implies the spatial derivative with respect to particle i's coordinates.

We now give the general approach to rewriting any equation in smoothed form:

1. Each of the exact equations is multiplied by a kernel and then integrated over the solution space.
2. We then integrate by parts, possibly with further approximations to deal with nonlinear terms, then produce equations for the kernel estimates.

Although equations (7) and (20) are correct, empirically it is more accurate to use:

$$\nabla_{\mathbf{r}} A = \frac{1}{\rho}\left[\nabla_{\mathbf{r}}(\rho A) - A\nabla_{\mathbf{r}}\rho\right] \tag{23}$$

If we now apply steps 1 and 2 to each term we get:

$$\langle \nabla_{\mathbf{r}}(\rho(\mathbf{r})A(\mathbf{r}))\rangle = \int \nabla_{\mathbf{r}'}(\rho(\mathbf{r}')A(\mathbf{r}'))W(\mathbf{r}-\mathbf{r}', h)\, d\mathbf{r}' \tag{24}$$

Using the exact same idea as in equation (9), we get:

$$\begin{aligned}\nabla_i(\rho_i A_i) &= \sum_{j=1}^{N} \frac{m_j \rho_j A_j}{\rho_j} \nabla_i W_{ij} \\ &= \sum_j m_j A_j \nabla_i W_{ij}\end{aligned} \tag{25}$$

Similarly, the second term becomes:

$$\langle \nabla_{\mathbf{r}} \rho(\mathbf{r}) \rangle = \int \nabla_{\mathbf{r}'} \rho(\mathbf{r}') W(\mathbf{r} - \mathbf{r}', h)\, d\mathbf{r}' \tag{26}$$

Upon integrating by parts, we get:

$$\begin{aligned} \nabla_i \rho_i &= \sum_{j=1}^{N} \frac{m_j \rho_j}{\rho_j} \nabla_i W_{ij} \\ &= \sum_{j=1}^{N} m_j \nabla_i W_{ij} \end{aligned} \tag{27}$$

Combining the results from equations (25) and (27) yields:

$$\begin{aligned} \nabla_i A &= \frac{1}{\rho_i} \left[\sum_{j=1}^{N} m_j A_j \nabla_i W_{ij} - A_i \sum_{j=1}^{N} m_j \nabla_i W_{ij} \right] \\ &= \frac{1}{\rho_i} \sum_{j=1}^{N} (A_j - A_i) \nabla_i W_{ij} \end{aligned} \tag{28}$$

Morris [1995] states that this approach is less susceptible to particle disorder. Most of the equations that follow will be derived in similar fashion.

3 Smoothing Length

The smoothing length h represents the effective width of the kernel, or the distance at which the particle in question interacts with other particles (See Figure (1)). It is equivalent to the width of a grid-cell in finite difference methods. SPH works best when the number of neighbors is between 20 - 35 in two dimensions (the range varies with the number of dimensions). At certain times throughout a simulation more or less than this range will be in a particle's neighborhood, which is problematic. One way to maintain this number is to have each particle dynamically adjust its smoothing length. It is important when calculating the forces on particles that the same smoothing length be used in the calculation of the force on particle i from particle j, as in the calculation of the force on particle j from particle i. If this is not done, then momentum is not conserved. The method widely used in SPH is to average the smoothing lengths between two particles, such as:

$$h_{ij} = \frac{h_i + h_j}{2} \tag{29}$$

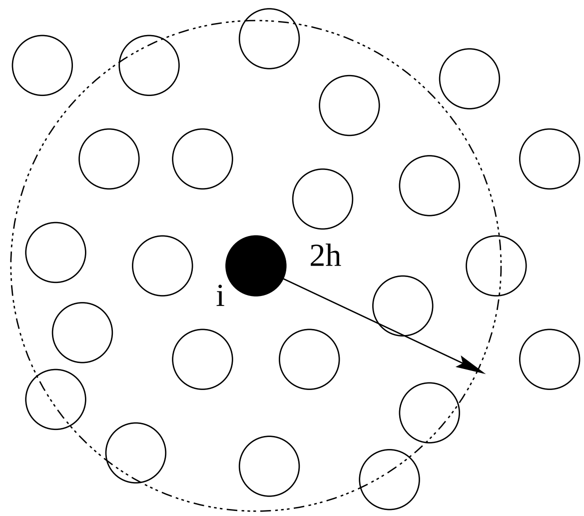

Figure 1: Particle i interacts with other particles only out to a distance of $2h$.

Another reason for having a dynamical smoothing length is that if h is too small, then that particle may have too few neighbors, and the statistical errors in the calculations of smoothed quantities are high. On the other hand, if h is too large and thus has too many neighbors, then computer time is being wasted. The equation one uses to calculate the new smoothing length that will keep the number of neighbors approximately constant from one time step to the next is given by

$$h_i = h_o \left(\frac{\rho_o}{\rho_i}\right)^{\frac{1}{\nu}} \tag{30}$$

where ρ_o and h_o are the original density and smoothing length, respectively, and ν is the number of dimensions.

The problem with equation (30) is that h_i is needed to calculate ρ_i, but we cannot get ρ_i until we have h_i. The solution to this is to find an equation that does not depend on ρ_i at all. Benz [Benz 1990] gives an equation for the rate of change of h_i:

$$\frac{dh_i}{dt} = \frac{1}{3} h \nabla \cdot \mathbf{v_i} \tag{31}$$

In general, equation (31) works quite well, but it eventually begins to approach the maximum or minimum number of neighbors. When this happens, we use a different equation, namely:

$$\frac{dh_i}{dt} = \frac{e^x \frac{(h\nabla\cdot\mathbf{v}_i)}{3} \pm e^{-x} 2h_i}{e^x + e^{-x}} \tag{32}$$

where "+" is chosen if h is to increase or "$-$" if it is to decrease. The value for x is given by:

$$x = \begin{cases} \frac{N_{upper} - N_{cur}}{.3n_{range}}, & \text{if } (N_{upper} - N_{cur}) < n_{range} \\ \frac{N_{cur} - N_{lower}}{.3n_{range}}, & \text{if } (N_{cur} - N_{lower}) < n_{range} \\ \infty, & \text{otherwise} \end{cases} \tag{33}$$

where N_{upper} and N_{lower} are the upper and lower bounds on the number of neighbors a particle should have for the dimensionality. n_{range} is 1, 5, or 12 in one, two, or three dimensions, respectively.

In our program, we only use a constant smoothing length. The main reason for this is because we use a grid of constant sized cells in order to find neighbors quickly. The use of some sort of neighbor list reduces the time complexity of the algorithm from $O(N^2)$ to $O(N)$.

4 Kernels

Along with the constraints that the kernel approximate a delta function in the $h \to 0$ limit and be normalized, it should also be differentiable at least once, which is required by equation (20). A delta function, or more correctly termed, a Dirac delta function, is defined by its assigned properties:

$$\delta(0) = 0, r \neq 0, \tag{34}$$

$$\int f(r)\delta(r)d\tau = f(0) \tag{35}$$

where $f(r)$ is any well-behaved function and volume integration includes the origin [Arfken 1985]. In addition to the kernel approximating a delta function, it must have a continuous second derivative so as to keep the integral slowly varying. This also ensures that the error in approximating the integral interpolants by summation interpolants are small provided the particle disorder is not too large.

The error in the integral interpolant can be determined by Taylor expansion of $A(\mathbf{r}')$ about $\mathbf{r}$ in equation (7). We will do this in one dimension for simplicity:

$$\langle A(x)\rangle = \int W(u,h)\left\{A(x) - uA'(x) + \frac{u^2}{2!}A''(x) - ...\right\} du \tag{36}$$

The error terms involving odd powers of u vanish if $W(u,h)$ is an even function of u. The dominant error term in the bracket is then given by $\frac{u^2}{2!}A''(x)$ and is $O(h^2)$. More accurate kernels can be constructed by requiring them to be normalized, to have zero first moments and vanishing second moments:

$$\int r^2 W(\mathbf{r},h)d\mathbf{r} = 0 \tag{37}$$

These kernels can be constructed by starting with any of the standard kernels and determining the coefficients c_1 and c_2 according to:

$$\widetilde{W} \equiv c_1 W(\mathbf{r},h)(1 - c_2 r^2) \tag{38}$$

so that the kernel $\widetilde{W}$ is normalized and the second moment vanishes. The error associated with these kernels is $O(h^4)$.

One of the standard kernels used is the cubic spline and it is given by:

$$W(\mathbf{r},h) = \frac{\sigma}{h^\nu}\begin{cases} (1 - \frac{3}{2}s^2 + \frac{3}{4}s^3), & \text{if } 0 \le s \le 1, \\ \frac{1}{4}(2-s)^3, & \text{if } 1 \le s \le 2, \\ 0, & \text{otherwise} \end{cases} \tag{39}$$

where $s = r/h$, ν is the number of dimensions and σ is the normalization constant with values $\frac{2}{3}$, $\frac{10}{7\pi}$, or $\frac{1}{\pi}$ in one, two, or three dimensions, respectively. One of the key advantages of using equation (39) as opposed to, say a Gaussian, is that it has compact support, that is, its interactions are exactly zero at distances $r > 2h$. Higher order splines can be used, but they interact at farther distances and thus require more computational time. Thus, the spline used depends on what error is acceptable and how long one is willing to wait to get those results. Since our goal is to have a near real time application, we chose the cubic spline.

5 Fluid Equations

5.1 Equation of State

The equation of state ties together all of the conservation equations for continuum fluid dynamics. So the equation of state must be chosen to model the appropriate fluid (*i.e.* compressible, incompressible). In SPH, the fluid pressure is an explicit function of density and thus particles are driven by local density gradients. For incompressible fluids such as water, a prohibitively small time step must be taken if we used the actual equation of state (*i.e.* Courant-Friedrich-Levy condition). Because of this, artificial equations of state have been defined for situations where the speed of sound is low enough to be practical yet high enough to limit density fluctuations to within 3% [Morris 1997]. Some authors have used an equation of state suggested by Batchelor [Batchelor 1967]:

$$P = P_o \left[\left(\frac{\rho}{\rho_o} \right)^{\gamma} - 1 \right] \tag{40}$$

where $\gamma = 7$, and P_o and ρ_o are the initial values. We can see that small variations in ρ produce large variations in P. This is what we would expect to happen for incompressible fluids, but it also produces large error values in pressure for smaller error values in density [Morris 1997]. The -1 in this equation was introduced to allow free surfaces to be modeled more accurately. The subtraction of one acts as though there is an external pressure of P_o pushing back on the surface.

Following Morris [1997], we decided not to use this equation in our final version; mostly because we are not planning on modeling free surface flows. The equation we do use is given by:

$$P = c^2 \rho \tag{41}$$

where c is the speed of sound in this fluid. We see that if c is large, then the pressure reacts strongly to slight changes in density, but not overly strong. As mentioned earlier, we do not want c to be too large or the pressure will be too large, producing large forces on the particles and thus causing us to use smaller time steps. We can determine our speed of sound in one of two ways. The first is by using the equation for the Mach number:

$$M = \frac{v}{c} \tag{42}$$

and noting that a fluid is considered incompressible if its Mach number is much less than 1.0 [White 1994]. Monaghan states that for density to vary by 1.0%, the Mach number should be 0.1. If we can determine what our maximum velocity will be, then we can determine the value of c from:

$$M = \frac{v_{max}}{c} \Rightarrow c = \frac{v_{max}}{M} = \frac{v_{max}}{0.1} = 10 v_{max} \tag{43}$$

The other way to find c is to balance the forces in the Navier-Stokes momentum equation to come up with several relations that the speed of sound is proportional to [Morris 1997]. The Navier-Stokes momentum equation is given by,

$$\frac{d\mathbf{v}}{dt} = -\frac{1}{\rho}\nabla P + \nu\nabla^2\mathbf{v} + \mathbf{F} \tag{44}$$

where ν is the kinematic viscosity and $\mathbf{F}$ is a body force per unit mass. The speed of sound should be comparable with the largest of:

$$c^2 \sim \frac{V_o^2}{\delta}, \frac{\nu V_o}{L_o\delta}, \frac{FL_o}{\delta} \tag{45}$$

where

$$\delta = \frac{\triangle\rho}{\rho_o} \tag{46}$$

and V_o is a velocity scale and L_o is a length scale. Thus, if we want a variation in density of only 3%, we set $\delta = 0.03$ and solve each with the appropriate length and velocity scales. The correct speed of sound for that simulation is then proportional to the largest value from the relations in equation (45).

5.2 The Continuity Equation

The density of each particle can be found from equation (8):

$$\rho_i = \sum_{j=1}^{N} m_j W_{ij} \tag{47}$$

but this equation requires an extra loop over all particles before it can be used in the calculations. A better formula is found from finding the smoothed version of the continuity equation:

$$\frac{d\rho}{dt} = -\rho\nabla\cdot\mathbf{v} \tag{48}$$

We can calculate the right hand side of this equation in either a straightforward fashion, to get:

$$-\rho\nabla\cdot\mathbf{v} = \sum_{j=1}^{N}\frac{m_j}{\rho_j}\mathbf{v}_j\cdot\nabla_i W_{ij} \tag{49}$$

or we can use the relation:

$$\nabla \cdot \mathbf{v} = \frac{\nabla \cdot (\rho \mathbf{v}) - \mathbf{v} \cdot \nabla \rho}{\rho} \tag{50}$$

which yields a more accurate result [Morris 1995]

With this relation, the continuity equation becomes:

$$\begin{aligned} \frac{d\rho_i}{dt} &= \sum_{j=1}^{N} m_j (\mathbf{v}_i - \mathbf{v}_j) W_{ij} \\ &= \sum_{j=1}^{N} m_j \mathbf{v}_{ij} W_{ij} \end{aligned} \tag{51}$$

where $\mathbf{v}_{ij} \equiv \mathbf{v}_i - \mathbf{v}_j$.

5.3 The Momentum Equation

The momentum conservation equation for a non-viscous fluid in the absence of gravity is given by Euler's equation:

$$\frac{d\mathbf{v}}{dt} = -\frac{1}{\rho} \nabla P \tag{52}$$

We see that the right hand side of equation (52) has a familiar form and we have already calculated two SPH relations for it; namely equations (20) and (28). We already stated that equation (20) did not produce as accurate results as equation (28) in practice, so we know we do not want to use that version. It turns out that equation (28) does not conserve linear or angular momentum and hence we do not want to use this version either.

We can find a formalism in which linear and angular momentum are conserved by rewriting the pressure gradient as:

$$\frac{\nabla P}{\rho} = \nabla \left(\frac{P}{\rho} \right) + \frac{P}{\rho^2} \nabla \rho \tag{53}$$

With the use of equation (22), the first term can be written as:

$$\begin{aligned} \nabla_i \left(\frac{P_i}{\rho_i} \right) &= \sum_{j=1}^{N} \frac{m_j}{\rho_j} \frac{P_j}{\rho_j} \nabla_i W_{ij} \\ &= \sum_{j=1}^{N} m_j \frac{P_j}{\rho_j^2} \nabla_i W_{ij} \end{aligned} \tag{54}$$

and the second term is written:

$$\begin{aligned}\nabla_i \rho_i &= \sum_{j=1}^{N} \frac{m_j}{\rho_j} \rho_j \nabla_i W_{ij} \\ &= \sum_{j=1}^{N} m_j \nabla_i W_{ij}\end{aligned} \tag{55}$$

Combining terms yields:

$$\begin{aligned}\frac{\nabla_i P}{\rho_i} &= \sum_{j=1}^{N} m_j \frac{P_j}{\rho_j^2} \nabla_i W_{ij} + \frac{P_i}{\rho_i^2} \sum_j m_j \nabla_i W_{ij} \\ &= \sum_{j=1}^{N} m_j \left(\frac{P_j}{\rho_j^2} + \frac{P_i}{\rho_i^2} \right) \nabla_i W_{ij}\end{aligned} \tag{56}$$

Thus, the acceleration that particle i feels due to the pressure force is given by:

$$\frac{d\mathbf{v}_i}{dt} = -\sum_{j=0}^{N} m_j \left(\frac{P_j}{\rho_j^2} + \frac{P_i}{\rho_i^2} \right) \nabla_i W_{ij} \tag{57}$$

5.4 Particle Motion

In general, particles can be moved by:

$$\frac{d\mathbf{r}_i}{dt} = \mathbf{v}_i \tag{58}$$

or the XSPH variant velocity:

$$\frac{d\mathbf{r}_i}{dt} = \hat{\mathbf{v}}_i \equiv \mathbf{v}_i + \epsilon \sum_{j=0}^{N} m_j \frac{\mathbf{v}_i}{\overline{\rho}_{ij}} W_{ij} \tag{59}$$

where $\overline{\rho}_{ij} = (\rho_i + \rho_j)/2$ and $(0 \leq \epsilon \leq 1)$ is a constant. The XSPH variant velocity was developed by Monaghan for free surface flows [Monaghan 1994]. The extra term tends to keep the particles more ordered and in high speed flows, it prevents the penetration of one fluid by another. If one uses the XSPH variant and equation (51), then the velocity in equation (51) should contain the right hand side of equation (59). Since we are not modeling free surface nor high speed flows, we will use equation (58).

6 Artificial Viscosity

The equations so far have been derived for non-viscous fluids. We have not implemented a mechanism in which the kinetic energy is converted into heat. This conversion process is the essence of viscosity and must be included if we wish to accurately simulate real fluids.

There are many forms of artificial viscosity that have been proposed for the SPH method ([Lucy 1977], [Wood 1981], [Monaghan & Gingold 1983], [Loewenstein & Mathews 1984], and [Evrad 1988]) but the most common is to add the following viscous pressure term into the momentum equation:

$$\frac{d\mathbf{v_i}}{dt} = -\sum_{j=1}^{N} m_j \left(\frac{P_i}{\rho_i^2} + \frac{P_j}{\rho_j^2} + \prod_{ij} \right) \nabla_i W(\mathbf{r_{ij}}, h) \tag{60}$$

where $\prod_{ij}$ is defined to be the viscous pressure and is given by:

$$\prod_{ij} = \begin{cases} \frac{-\alpha c_{ij}\mu_{ij} + \beta \mu_{ij}^2}{\rho_{ij}}, & \text{if } \mathbf{v}_{ij} \cdot \mathbf{r}_{ij} < 0, \\ 0, & \text{otherwise} \end{cases} \tag{61}$$

with:

$$\mu_{ij} = \frac{h\mathbf{v}_{ij} \cdot \mathbf{r}_{ij}}{\mathbf{r}_{ij}^2 + \eta^2}$$

where h is the smoothing length, $\eta^2 = 0.01h^2$, α and β are constants dependent on the particular type of simulation being run. Common values for α and β for astrophysical simulations are approximately 1 and 2, respectively. For versions $1.x$ and $2.x$ of this project we used $\alpha = 0.1$ and $\beta = 0.0$ as Monaghan mentions in his paper on free surface flows [Monaghan 1994]. The linear term in equation (61) produces a shear and bulk viscosity [Monaghan 1985] while the quadratic term handles high Mach number shocks.

The main purpose of this artificial viscosity equation is to model strong shocks in astrophysical processes ([Benz 1989], [Monaghan 1992]). Morris [Morris 1997] states that although this form has been used to model real viscosity, it produced inaccurate velocity profiles in his simulations. He presents a different formula and is the one in which we have chosen to use as well. Morris states that their formula is based on an expression used to model heat conduction, but cites an unpublished paper. We will proceed anyway since his tests show that his model works well for low Reynolds number, incompressible flows, such as the flow in our model. Morris writes down the following relation:

$$\left\{\left(\frac{1}{\rho}\nabla\cdot\mu\nabla\right)\mathbf{v}\right\}_i = \sum_{j=1}^{N}\frac{m_j\left(\mu_i+\mu_j\right)\mathbf{r}_{ij}\cdot\nabla_i W_{ij}}{\rho_i\rho_j\left(\mathbf{r}_{ij}^2+0.01h^2\right)}\mathbf{v}_{ij} \tag{62}$$

where $\mu = \rho\nu$ is the dynamic viscosity. He then states that this expression can be shown to be appropriate by taking a Taylor expansion about particle i. The paper he cites is the same unpublished one as before. Equation (62) conserves linear momentum exactly, but only approximately conserves angular momentum [Morris 1997].

If the kernel only depends on the distance between interacting particles, then we can write:

$$\left\{\left(\frac{1}{\rho}\nabla\cdot\mu\nabla\right)\mathbf{v}\right\}_i = \sum_{j=0}^{N}\frac{m_j\left(\mu_i+\mu_j\right)}{\rho_i\rho_j}\left(\frac{1}{|\mathbf{r}_{ij}|}\frac{\partial W_{ij}}{\partial r_i}\right)\mathbf{v}_{ij} \tag{63}$$

Thus, the final momentum equation can be written as:

$$\frac{d\mathbf{v}_i}{dt} = -\sum_{j=1}^{N}m_j\left(\frac{P_i}{\rho_i^2}+\frac{P_j}{\rho_j^2}\right)\nabla_i W(\mathbf{r}_{ij},h)+\sum_{j=1}^{N}\frac{m_j\left(\mu_i+\mu_j\right)}{\rho_i\rho_j}\left(\frac{1}{|\mathbf{r}_{ij}|}\frac{\partial W_{ij}}{\partial r_i}\right)\mathbf{v}_{ij}+\mathbf{F}_i \tag{64}$$

where $\mathbf{F}_i$ is the body force per unit mass evaluated at particle i.

7 Boundary Conditions

As stated in the assumption in going from equation (11) to equation (19), if the surface term does not go to zero within our domain, then we must apply explicit boundary conditions. This is most often done by replacing the boundary as smoothed particles with special characteristics. In Monaghan's paper on free surface flows [Monaghan 1994], he modeled the boundary with "boundary particles" that lined the intersection between the boundary area and the free area. (See Figure (2)). The boundary particles are just like the smooth particles except that they cannot move and they interact with the free particles according to the following force:

$$f(\mathbf{r}) = \begin{cases} D\frac{\mathbf{r}}{r^2}\left(\left(\frac{r_o}{r}\right)^{p_1}-\left(\frac{r_o}{r}\right)^{p_2}\right), & \text{for } r<r_o \\ 0, & \text{otherwise}\end{cases} \tag{65}$$

where D is some constant, r_o is the initial free particle spacing and p_1 and p_2 are taken to be 12 and 6 respectively (although some authors have used

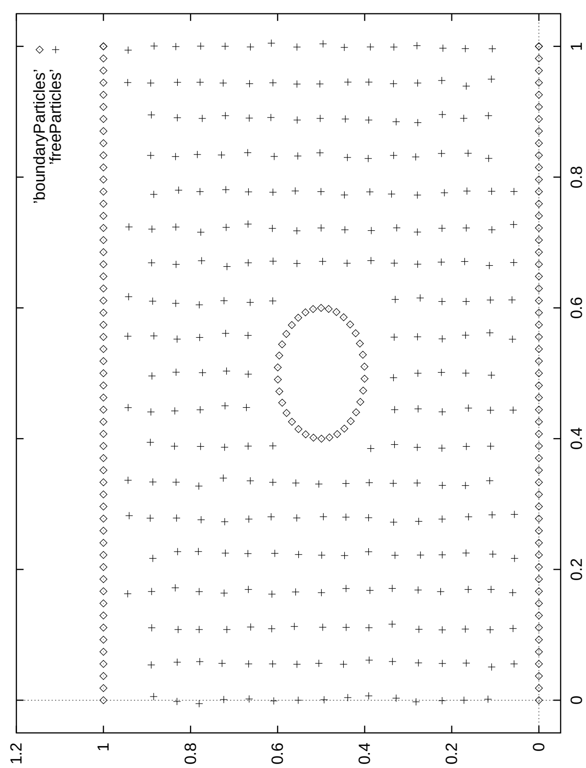

Figure 2: The “diamonds” represent boundary particles and the “pluses” represent the free particles. Notice that the boundary particles are placed only along the edge of the shape and not inside of it.

4 and 2). The effect of this function is to produce a strongly peaked, purely repulsive barrier. If we wish to incorporate no-slip conditions, then we just include the boundary particles in the interaction loops. For versions 1.x and 2.x, we used this form of boundary conditions but decided to use Morris's [Morris 1997] in version 3.x.

Morris models the boundaries using special smoothed particles as well, except that these boundary particles do not necessarily have to lie on the intersection of the border. Instead they fill the boundary region uniformly; see Figure (3). The boundary particles contribute to the density such that the pressure decreases when fluid and boundary particles diverge. The boundary particles interact with free particles by including an extra summation in the conservation equations. The equations for free particles then become:

$$\begin{aligned} P_i &= c^2 \rho_i, && (66) \\ \frac{d\rho_i}{dt} &= \sum_{j \in \mathbf{F}} m_j \mathbf{v}_{ij} W_{ij} + \sum_{j \in \mathbf{B}} m_j \mathbf{v}_{ij} W_{ij} && (67) \end{aligned}$$

$$\begin{aligned} \frac{d\mathbf{v}_i}{dt} = & - \sum_{j \in \mathbf{F}} m_j \left(\frac{P_i}{\rho_i^2} + \frac{P_j}{\rho_j^2} \right) \nabla_i W_{ij} \\ & + \sum_{j \in \mathbf{F}} \frac{m_j (\mu_i + \mu_j)}{\rho_i \rho_j} \left(\frac{1}{|\mathbf{r}_{ij}|} \frac{\partial W_{ij}}{\partial r_i} \right) \mathbf{v}_{ij} \\ & - \sum_{j \in \mathbf{B}} m_j \left(\frac{P_i}{\rho_i^2} + \frac{P_j}{\rho_j^2} \right) \nabla_i W_{ij} \\ & + \sum_{j \in \mathbf{B}} \frac{m_j (\mu_i + \mu_j)}{\rho_i \rho_j} \left(\frac{1}{|\mathbf{r}_{ij}|} \frac{\partial W_{ij}}{\partial r_i} \right) \mathbf{v}_{ij} \\ & + \mathbf{F}_i \qquad (68) \end{aligned}$$

where $j \in \mathbf{F}$ implies that j must belong to the set of free particles and $j \in \mathbf{B}$ implies that j must belong to the set of boundary particles. The boundary particles do not have their position or velocity updated but they do evolve their density and hence their pressure. Their equations are:

$$P_i = c^2 \rho_i \qquad (69)$$

$$\frac{d\rho_i}{dt} = \sum_{j \in \mathbf{F}} m_j \mathbf{v}_{ij} W_{ij} \qquad (70)$$

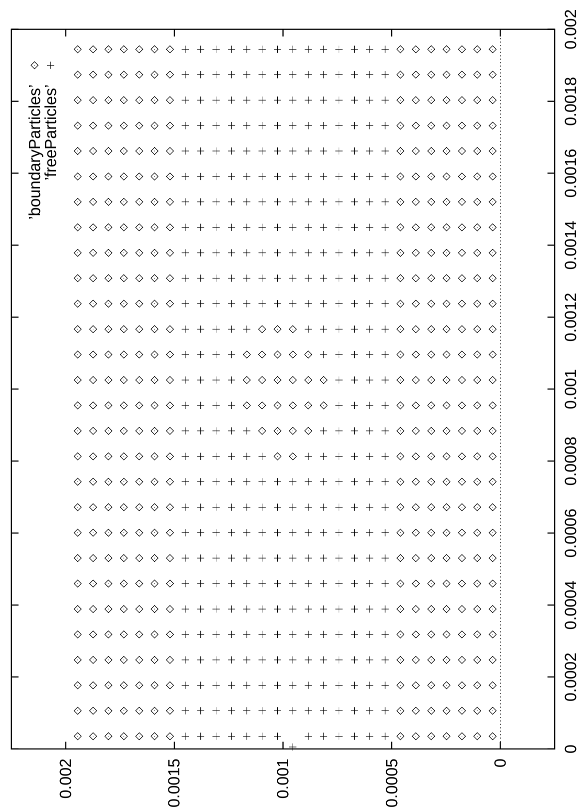

Figure 3: The “diamonds” represent boundary particles and the “pluses” represent the free particles. The boundary particles are placed throughout the shape. This allows the free particles to interact with the boundary particles in a consistent manner.

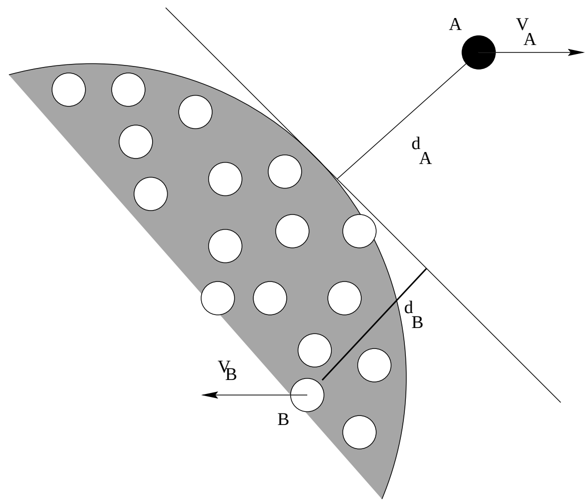

Figure 4: Particle **B** gets its artificial velocity from the relative distances of it and the free particle **A** to a tangent line.

The boundary particles do not move, thus their true velocity is always zero. The problem is that if their velocities are always zero, then they cannot interact correctly with the free particles through the SPH equations. Thus, Morris has come up with a way to attribute an artificial velocity to a boundary particle due to the interaction with a free particle. (See Figure (4)). The method of calculating the artificial velocity is as follows:

For each free particle A
    For each boundary particle B
        Calculate the normal distance to the boundary from particle A
        Calculate the tangent line in accordance with the normal of A
        Calculate the normal distance to the tangent line from particle B
        Extrapolate the velocity of free particle A across the tangent line,
        assuming zero velocity at tangent line, to particle B

Thus, each boundary particle has an artificial velocity that is dependent on who it is interacting with and is given by:

$$\mathbf{v}_B = -\left(\frac{d_B}{d_A}\right)\mathbf{v}_A \tag{71}$$

The relative velocity is then:

$$\begin{aligned}
\mathbf{v}_{AB} &= \mathbf{v}_A - \left(-\left(\frac{d_B}{d_A}\right)\mathbf{v}_A\right) \\
&= \left(1 + \left(\frac{d_B}{d_A}\right)\right)\mathbf{v}_A \\
&= \beta\mathbf{v}_A
\end{aligned} \tag{72}$$

To keep the boundary particles' values a finite number in circumstances in which particle A approaches the boundary and thus $d_A \to 0$, we use the following conditional for β:

$$\beta = min\left(\beta_{max}, 1 + \frac{d_B}{d_A}\right) \tag{73}$$

Morris states that a good value for β_{max} is 1.5.

8 Time Integration

We use the *Leap-Frog* technique as our numerical integration method. See [Press *et al.* 1997] for an in-depth discussion of the *Leap-Frog* method. For this, or any numerical integration routine, we must be careful with our time step; Morris gives several criteria which must be satisfied [Morris 97],

$$\triangle t \leq 0.25\frac{h}{c} \tag{74}$$

$$\triangle t \leq 0.25 min_i \left(\frac{h}{|f_i|}\right)^{\frac{1}{2}} \tag{75}$$

$$\triangle t \leq 0.125\frac{h^2}{\nu} \tag{76}$$

where equation (74) comes from the Courant-Friedrich-Levy condition, equation (75) is from the forces on each particle, and equation (76) comes from viscous diffusion. In our program, equations (74) and (76) are calculated during the initialization to come up with a ball park figure for the time step. Then during the simulation, we check our time step versus equation (75) in order to ensure accuracy of results. If our time-step is too large, then we display a message and continue computing. It is up to the user to stop the program or not.

Part III
Previous Work

SPH has been around for just over 20 years, but most of its use has been focused on astrophysics. In the past few years, several papers have been written showing when SPH could be applied to fluid flow problems and how much error was incurred.

Monaghan *et al.* [Monaghan 1994] wrote a paper about free surface flows, and in this paper they showed how SPH and their method of boundary conditions could be used to acquire reasonable results. The tests they ran were for motion of an elliptical drop, motion between horizontal plates, jets of fluid exiting a nozzle, and a bursting dam. The authors used stationary SPH particles as boundary points and they repelled normal SPH particles through a purely repulsive force that only acts to a finite distance (see equation (65)). In versions $1.x$ and $2.x$ of our project, we used several of the equations from that paper, as well as their method of modeling boundaries.

Morris *et al.* [Morris 1997] wrote a paper describing low Reynolds number, incompressible fluid flows which use a different method for modeling boundaries, as well as a different formula for the viscosity than Monaghan. Version $3.x$ of our project incorporates their work from that paper. The authors test their method on Couette flow, Poiseuille flow, and flow through a periodic lattice of cylinders. The tests in section 6 reproduce their results for these cases. The final version of our project will use the same method as was used by Morris *et al.*.

Takeda *et al.* [Takeda 1994] wrote a paper about the simulation of viscous flows and here they also showed that SPH could be used to model fluid flow problems. Their test problems were 2D Poiseuille flow, 3D Hagen-Poiseuille flow and 2D isothermal flow around a cylinder. They used a boundary method based on "imaginary" particles just on the other side of the boundary.

A group from George Mason University [Wallin URL] have a web-site that goes over the most important aspects of SPH and gives a high-level algorithm for writing SPH code. This algorithm is the basis for our project. They also have many links to other groups around the world using SPH.

There are other papers that test the SPH method, but most are in the astrophysical realm and not fluid flow. It should be noted that there has been much growth in the last few years in modeling fluid flows using SPH, but there are far more papers discussing astrophysics. The only SPH software that was available to the public was astrophysical SPH software and it was difficult to decipher due to the data structures that improved performance but destroyed readability. Thus, we believe that there is not any readily available SPH fluid flow software available to the general public, much less object oriented software that is written for both a serial and parallel architecture. In addition, we are fairly confident that no one has an architecture independent user interface that communicates to the server and also displays the results.

Part IV
Analysis and Design

9 Analysis

We will briefly describe what properties we wish our program to have, explain what design decisions we made in order to satisfy them and finally evaluate if we succeeded or not. This tool is to be used by undergraduate and graduate engineering students taking Intermediate Fluid Mechanics, ME 460/560, at Oregon State University. They will use this tool to better understand simple two dimensional problems in fluid mechanics. The professor of the fluid mechanics course, Dr. Jim Liburdy, was happy to talk with us about what information is important and how he would like to see it displayed. Some of the general ideas that came from that conversation and in discussions with the other graduate student working on this project are:

1. Usefulness - The program should be simple enough to learn quickly, yet powerful enough to simulate and visualize complex systems.
2. Portability - The program should be able to run on multiple architectures, not just the engineering HP's and SWARM.
3. Robust - The program should not just crash, or return incorrect results without sending a message about the problem.
4. Performance - The program should run approximately in real time.
5. Extendibility - The code should be written in such a way that new physics could be incorporated easily. (*e.g.* gaseous flow, adding new

shapes, *etc*).

The overall goal is to make the SPH code available to anyone, while at the same time leaving behind a base that has the potential to become a very powerful tool. Thus, we felt that extendibility was the key feature to this project. If the program is designed with extendibility in mind, then eventually all of the other properties can be realized.

10 Design

In this section, we will go through each of our requirements and explain what high-level decisions were made to try and meet that requirement.

The first property is "usefulness", or building a tool that is easy to learn and easy to get results quickly. We felt that in order for the program to be useful, we had to incorporate a user interface. This user interface will be in charge of receiving the initial conditions and then displaying the results in a variety of different formats. Some of the visualizations we considered are:

1. Streamlines - Click in the flow field and watch that particle move about.

2. Vorticity - Contour plot; color variations.

3. Density - Contour plot; color variations.

4. Pressure - Contour plot; color variations.

5. Velocity Field - Arrows representing direction of flow and length of arrow representing relative speed.

We decided that we would only have time to do the simplest of these. We chose to implement contour plots for the density, pressure, and speed of the fluid. The reason these were chosen is because the contour plots show how the fluid interact with the shapes. We felt that breaking the program up into a Client-Server architecture would separate the chores of the application. The server has the job of running the simulation of the fluid flow, while the Client receives initial conditions from the user, as well as, display the results.

Along with the property of usefulness, we had to consider portability, which is discussed next, so we chose Java as the language for the Client. The Server is not written in Java since we must also consider performance.

Thus, the Server is written in C++ and in the case of the parallel version, uses Message Passing Interface (MPI) routines. It should be noted that the other visualization ideas could be implemented very easily since all of the data will be sent to the Client for each frame (*i.e.* position, velocity, density, *etc.*). In the Future Work section, it is discussed how one could implement additional visualizations.

Another way to increase the usefulness of this project is to include mouse interactions to the user interface. Some of our ideas include:

1. Click and drag to create, resize, and move shapes around the display.

2. Click on the viewing window and a streamline appears.

3. Zoom into certain regions of the viewing window.

As it stands, none of these ideas will be implemented for this version of the project. It seems that usefulness leans more heavily upon graphical user interface design. That, in itself, could be another Masters of Science project. We will briefly describe how each of these features could be implemented in the Future Work section.

The next section is "portability", which includes developing the software in such a way that someone could down-load it and run it with minor modifications. As we discussed in the previous section, Java fulfills this requirement since it is architecture independent. For the Server, we would ideally have two identical versions, one utilizing parallel architectures and Message Passing Interface (MPI) and the other utilizing serial architectures. The parallel code is written in ANSI C++ and MPI, both of which are very common to scientists and should not pose a problem in portability. The serial version is also written in C++, but it does not need the MPI libraries to run. Since the Client and Server must communicate, we must make sure it is through a common interface. We chose to use socket connections, which everyone should have as long as they are connected to the Internet. The only problem occurs if we try to run both the serial version of the program and Client on the same PC.

One problem is that most home computers do not have their own domain names. Another problem is that we are not sure it is a "good" way to communicate between the Client and Server when they are both on the same machine. In the future work section, we will explain how the serial version Server and Client could communicate on the same machine.

Another design property we sought was "robustness". We wanted the code to either gracefully crash in expected ways or prompt for redirection.

This is probably the most important property for a scientist, since they want their results to be "correct" **all** of the time. Development time did not allow for adequate enforcement of this feature. Since the code is not expected to be used to simulate any "new" physics and since it will only be used by students, we can assume that a crash or two is acceptable, as long as it is a predictable and reproducible crash. Our attempt at robustness is to check the number of particles each particle interacts with, to calculate the correct time step and compare with the current time step, and to calculate the density variation.

In order to incorporate "performance" as a design property, we have written the code so that it can be run on parallel architectures. The idea is to break the problem up into smaller pieces, each of which a processor can work on. This should increase the performance and thus decrease the execution time. An ideal speedup would be linear in the number of processors, but it is only a theoretical limit. There are several languages available that allow one to write a parallel program, but MPI is the most common and standardized, so we chose it. Another way we tried to speed up the program was to delegate the work required for running the simulation and the work required to display the output. Thus, the Client does the parsing and displaying of the images, while the Server runs the simulation. Another option would have been to allocate one more processor which is in charge of parsing data sets, generating images and then sending them to the Client. We have chosen the former method in the current version of this application.

Finally, the last design property is "extendibility". Again, this is the most important to fully implement. We knew the program was going to get pretty large in size, and we knew that it had to be capable of crunching numbers quickly. These two realizations lead us to an object oriented language and some sort of scientific language. What we came up with was either C++ or Fortran 90. We chose C++ for several reasons; the first is that I had more experience with C++ than with Fortran. The second is that if the program is to be extended by someone else, then we felt that C++ would allow for easier modification of the existing code. We utilized object oriented design as much as possible, and a concerted effort was made to keep programming styles (naming conventions) similar between the two people who programmed this application. Also, variables and methods have names that describe what they are and what they are doing.

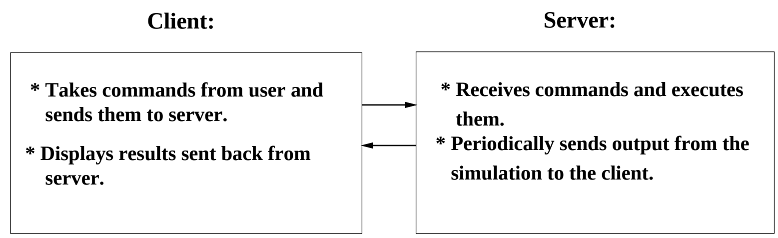

Figure 5: Overall architecture of the program.

Part V
Architecture

11 Server

The overall form of our program is a Client-Sever model, in which the Client is written in Java and the Server is written in C++ and MPI. The Client sends commands and initial conditions to the Server and the Server executes those commands. The Server then periodically returns results to the Client, in which it then displays as an animation. (See Figure (5)). The periodicity corresponds to predefined time increments with respect to the simulation (*i.e.* simulation time as opposed to wall clock time).

We now look in more detail at what happens once the Server has been sent a command and/or initial conditions. This is shown in Figure (6). This level is broken up into two classes, the Master and the Slave. As their names imply, the Master requests that the Slave perform an action and the Slave then does this and waits for a new command. The most common request is to run the simulation and periodically send data back, which the Master then bundles up and sends on to the Client.

For the parallel versions (*e.g.* Version $2.x$ and $3.x$), we must send these requests via the Message Passing Interface, or MPI. Thus, we have two more classes between the Master and the Slave. (See Figure (7)). The primary task for the classes, SlaveMPI and MasterMPI, is to communicate the commands and data between the Master and the Slave. SlaveMPI has one more important job and that is to perform calculations on the simulation. Figure (7) is a little misleading because it only represents one SlaveMPI object,

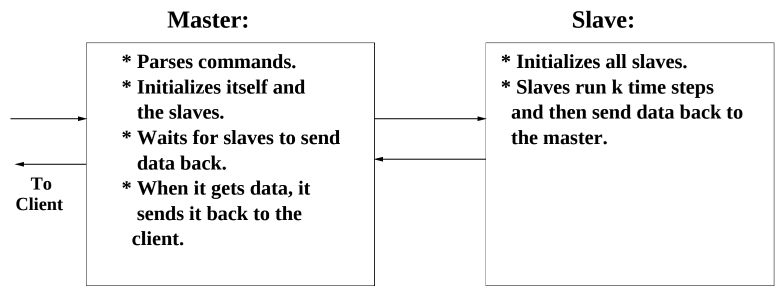

Figure 6: The Master is the interface between the Client and the algorithm which computes the simulation.

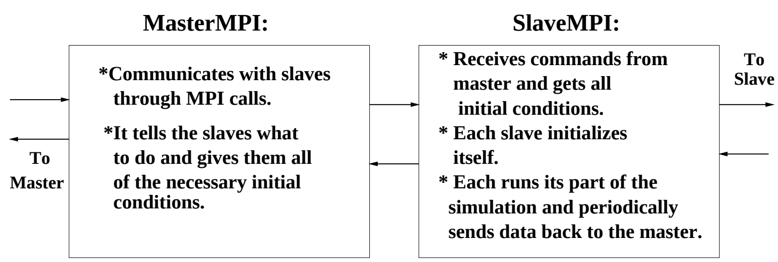

Figure 7: This shows how the Master interface uses MPI routines to to communicate with the rest of the program.

when in fact there are "p" of them. Each of these SlaveMPI objects is actually a separate process running on its own processor and in the case of our parallel system (SWARM), it has its own main memory and cache as well.

Figure (8) represents what goes on within each SlaveMPI object. Each receives the initial conditions (*i.e.* what part of the simulation it is to work on), initializes itself and runs the simulation. The Slaves are not completely free to do their own work since the SPH model requires that each particle interacts with neighboring particles; it is possible that a neighboring particle is in another process's region. For this reason, the program remains synchronized at each time step and thus the program is only as fast as the slowest processor. Vertanen [Vertanen 1999] discusses alternative solutions to this problem, but errors in the results are produced.

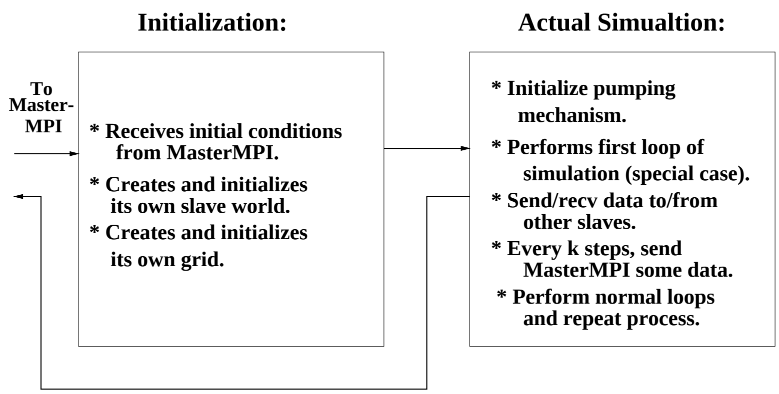

Figure 8: This is the code that initializes and runs the actual simulation.

The lowest level we can get to without actually implementing the program is given in Figure (9). This shows the actual simulation algorithm and was taken from the web site of a group that works on SPH [Wallin]. The algorithm uses a *Leap-Frog* integration routine, so we must first predict our values at a half step forward and then use these to calculate our pressure and the changes in all variables due to interacting with neighbors. Then we use all of this information to calculate our values a full time step ahead. After each time step, we reload our buffer rows. These buffer rows include the particles that border us but belong to another process (*i.e.* Slave). See Vertanen's paper for a full discussion of this topic [Vertanen 1999].

12 Client

The Client consists of the user interface, a connection to the Server, and a method to create images for display. The user interface is shown in Figure (10) and consists of only the core functionality. This includes loading and saving previous work (*i.e.* results and initial conditions), opening a connection to the Server, sending and receiving of data to the Server, and viewing various plots of the data.

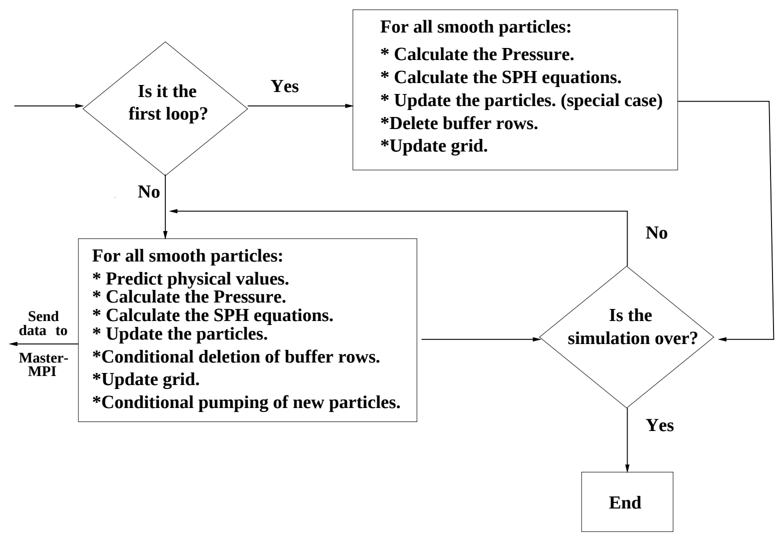

Figure 9: This shows the flow of the algorithm that calculates the field values for all particles.

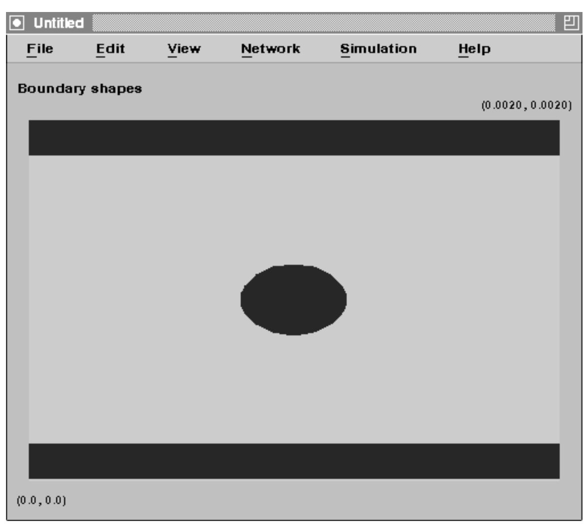

Figure 10: Client interface with top and bottom boundaries, as well as a cylinder in the middle.

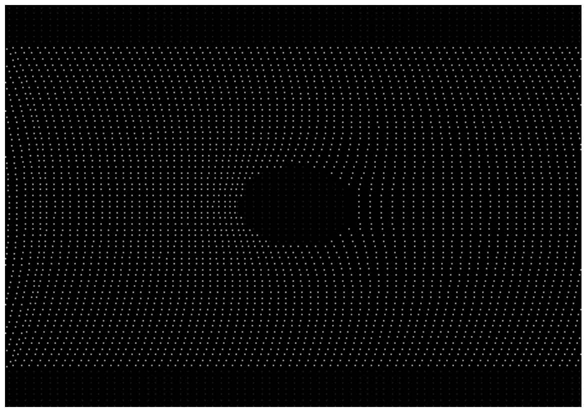

Figure 11: Positional plot of particles after t time steps with a cylinder in the middle of the flow. The direction of flow is left to right. Also, the fluid is exhibiting the no-slip interaction with the boundaries.

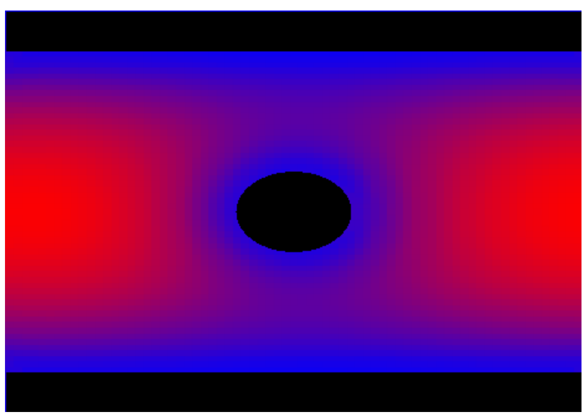

Figure 12: “speed” plot after same t time steps as in figure (11). The speed gives an energy representation of the flow, so that redder areas are moving faster than the bluer areas. (Unless this figure is being viewed in color, the color variation is difficult to detect.)

Part VI
Implementation

In the Analysis and Design section, we discussed the properties we wanted from our program and how we decided to design in order to achieve them. In this section, we will discuss the algorithms in more detail with a focus on data structures, time complexity and bugs. Overall, there were three major versions of this program; the first was the serial version based on Monaghan's work with free surface flows [Monaghan 1994], the second was identical to the first except that it was written for parallel architectures, and the third version has both a serial and parallel version with equations based on Morris's work with low Reynolds number, incompressible flows [Morris 1997].

13 Version $1.x$

We wrote the program around the algorithm used to update physical values of each particle in the simulation (see Figure (9)). The first draft consisted of hard coded arrays that contained particle objects and several miscellaneous variables used for bookkeeping. The initial conditions were also hard coded and the output was raw data containing the positions of each particle. The main reason for such a crude first draft was to minimize the complexity of the algorithm while debugging the numerical integration routine. In using the *Leap-Frog* numerical integration routine, we are required to we keep a "history" of each particle; we need to know the values of the particle at the old time step, at its mid step and then we calculate the new values at a full time step ahead. The source of most bugs here was keeping track of which time step values to use and where to use them.

At this point, we were using a naive algorithm in which every particle had its interaction calculated with every other particle. Thus, it was $O(N^2)$, where N is the number of particles. This was a huge waste of CPU time since each particle gets non-zero values from only a small fraction of the total N. After we were confident the general algorithm worked we wanted to make it run faster (See Figures (13) - (16)).

Our approach to speed up the program was to maintain a neighbor list for each particle. The exact data structure used to keep track of neighbors was a 2D array of "cells" called a Grid. (See Figure (17)). Each Cell in the

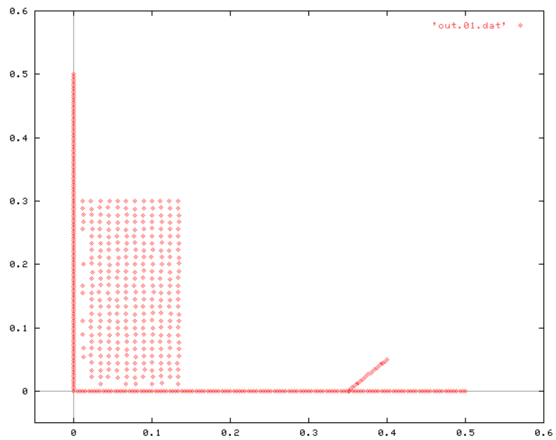

Figure 13: Breaking dam at t=0 with an obstacle “down stream”.

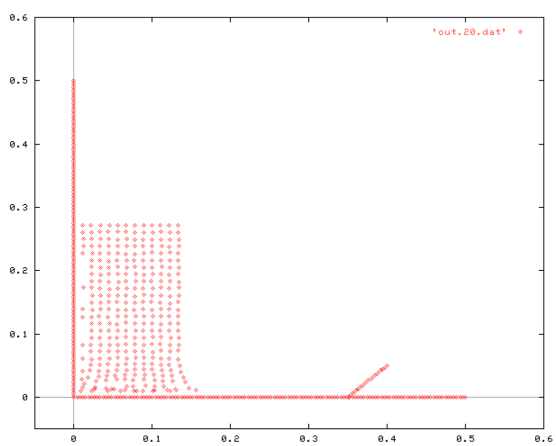

Figure 14: Same as in figure (13) but at a later time.

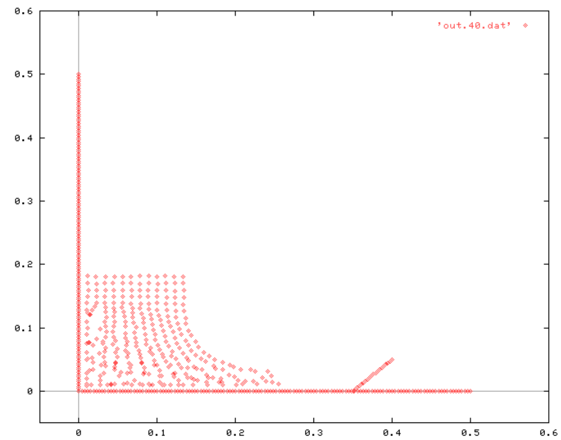

Figure 15: An even later time.

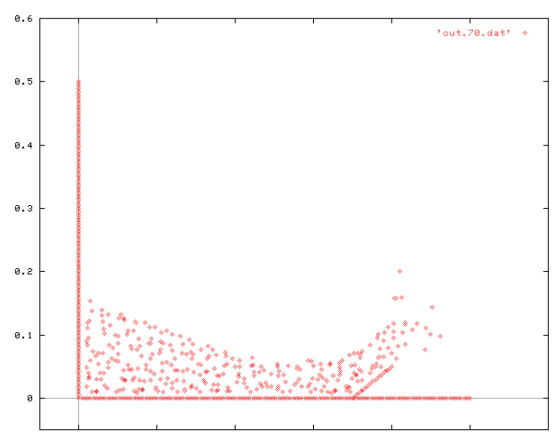

Figure 16: An even later time

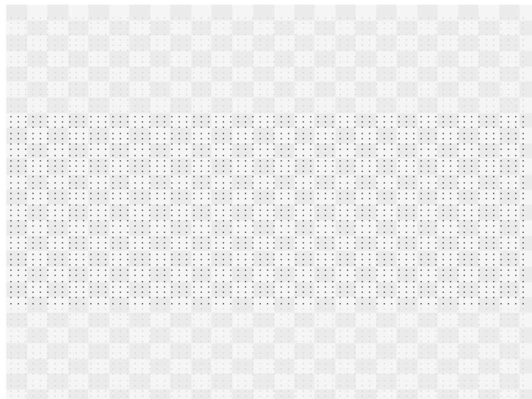

Figure 17: The faint checkerboard is our Grid and the dots represent the position of our free particles within this Grid.

Grid contains two linked lists, one for the free particles in that cell and one for the boundary particles. Each cell in the Grid was a square with sides equal to $2h$, or two times the smoothing length. The reason the cells are $2h$ in size is because the particles only interact up to $r < 2h$ and thus we only need to check the current cell and neighboring cells to find all of the possible particles that interact with the current one. The algorithm we used to interact particles within cells is as follows:

```
loop over cells in column I
loop over cells in column J
    cells[I][J] interacts with cells[I-1][J+1]
    cells[I][J] interacts with cells[I+1][J+1]
    cells[I][J] interacts with cells[I+1][J-1]
    cells[I][J] interacts with cells[I-1][J-1]
    cells[I][J] interacts with cells[I-1][J]
    cells[I][J] interacts with cells[I][J+1]
    cells[I][J] interacts with cells[I+1][J]
    cells[I][J] interacts with cells[I][J-1]
    cells[I][J] interacts with cells[I][J]
```

When we write, “cells[I][J] interacts with cells[I’][J’]”, we are implying that the interaction is over all particles in the current cell [I][J] with all other particles in the other cell [I’][J’].

In later versions, we realized that the order of interaction has an impact on the solution, albeit a small one. If one looks at the curve of how much each particle interacts as a function of distance, one sees that almost all of a particle’s interaction value comes from neighboring particles less than h away. The other key point to realize is that computers can only store finite representations of real numbers. Thus, summing a sequence of numbers from largest to smallest is NOT the same as summing the same sequence from smallest to largest. The most reliable way to sum numbers within a computer is from from smallest to largest. Thus, we only have to rearrange our interaction calculation so that we look at corner cells first, then top, bottom, right, left, in any order, and finally the same cell. To maintain the correct neighbor lists, we loop through all particles and hash them according to their position in relation to the cell size. The use of the Grid object made our time complexity an astounding $O(N)$. This is optimal since the program must at least loop through all N particles in order to update their position. The next step in the evolution of the program was to make it run on parallel architectures.

14 Version $2.x$

This version was written almost exclusively by the other graduate student working on this project, Keith Vertanen. His task was to take the serial version and make it run on a parallel architecture; and this turned out to be a formidable task.

We decided to break up our problem into strips that run parallel to the flow in the absence of any shapes. (See Figure (18)). Each strip represents a region of the flow field and includes all free flowing and boundary particles there-in. Each processor contains its own strip and is in charge of calculating the interaction of each of its free particle with that particle’s neighbors. The first problem to overcome is how to interact with particles in another strip. The solution we came up with is to allocate two “buffer rows” of particles that neighbor each strip: One buffer above the strip and one below (see Figure (19)). The buffer rows are used as read only, since the real particles

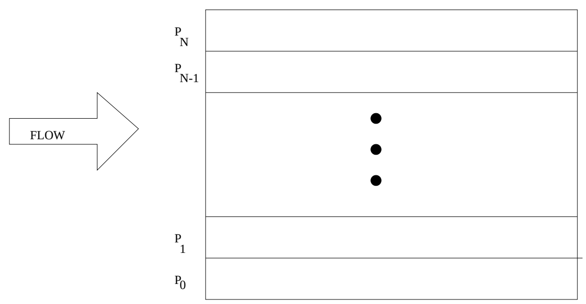

Figure 18: Each processor is allocated a strip of cells. Each strip can contain multiple rows of cells.

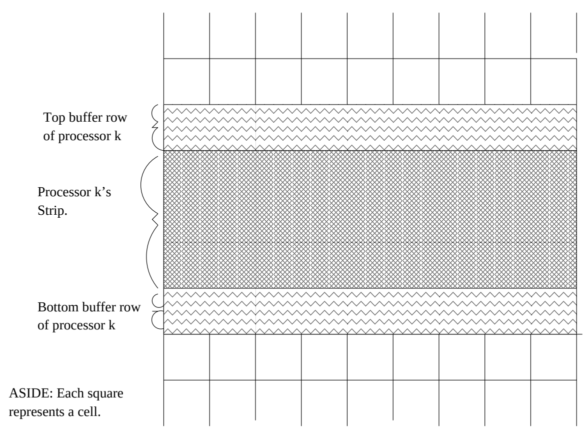

Figure 19: Each processor needs to be able to interact with a row of cells above and below its strip in order to calculate the SPH equations.

are contained within another process's strip and thus has its interactions taken care of by that process.

Since we need to calculate the interactions of each particle with its neighbors at each time step, we need to update the buffer rows at each time step. This implies communication between processors, and communication is very expensive compared to computation. One way to overcome this bottleneck is to either add additional buffers and push the problem of neighboring cells further out, or just assume that the values of each particle do not change much from one time step to another.

If the latter statement is true, then skipping updates of the buffer rows is far simpler than implementing multiple buffers and performance is increased. Keith Vertanen's MS Project has to do with implementing and testing the parallelization of this program. Thus, I will only describe the overall results and refer the interested reader to Vertanen's paper [Vertanen 1999]. The results of execution time versus number of time steps per update of buffer rows, which has been named the lazy update method, are exactly what one would expect. The program runs faster when it doesn't have to communicate as much. Vertanen goes on to point out that this difference in performance tapers off when the number of particles becomes too large. This is because the computation is what is taking the longest. Also, the accuracy tends to to be reduced as one increases the number of steps until the buffer rows are updated. The results for this should not be taken blindly, since the method in which they were calculated leaves room for question. The actual method was to calculate the average position of all particles for each simulation after k time steps. One case in which this technique will fail is when half of the particles have positions slightly greater than the correct positions and the other half of the particles have positions slightly less than the correct positions. The result is that the average value is identical to the correct value for the average position. Under the circumstances, (*i.e.* code not working correctly), this was the best method to use. Once the code is running correctly, a hydrodynamic value such as the drag force on a shape should be tested.

There were two major bugs in this version. One had to do with general bookkeeping (*i.e.* buffer updates, moving particles from one strip to the next, *etc.*) and the other bug had to do with the SPH method itself. The second bug took so much of our time, we ended up naming it: we called it "cancer". The first bug was boring in the sense that it was easy to find the cause. The idea is that, within each strip, every particle is kept track of as a SmoothParticle object. The problem occurs when we have to send a particle

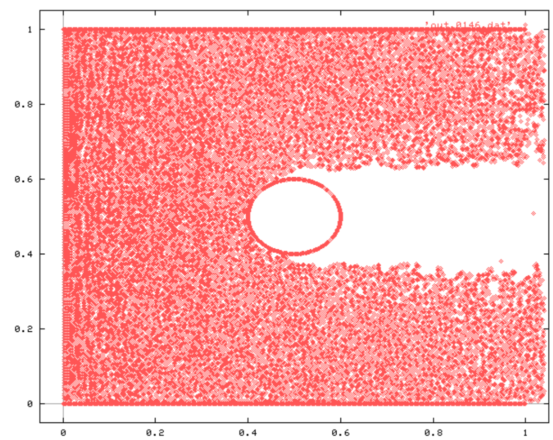

Figure 20: Simulation of flow past a cylinder before “cancer” strikes.

from one process to another because of buffer updates or because the particle has moved into its region. MPI does not allow the sending and receiving of objects such as SmoothParticles, so we had to create methods that picked out variables unique to that particle, pack it up in an array and use MPI to send these arrays to another process. The receiving process would then unpack it and initialize a SmoothParticle object according to these unpacked attributes. The problem was that we were not sending enough information, and those particles became lost in the shuffle of trying to place them. The result was particles disappearing and then never reappearing without any warning.

The second bug had to do with particles somehow getting large results and then spreading these results to nearby particles. The effect is a radial spread of bad values and hence cancerous behavior. Within only a couple frames, every particle would become infected and the simulation would crash. (See Figures (20) through (22)). Our first solution was ad hoc, and ultimately inadequate. Basically, we irradiated any particle that showed erratic behavior by removing that particle from the simulation (*e.g.* particles with speeds greater than some acceptable value). This cauterized the problem, but it also introduced extra unphysical behavior to our simulation by producing large voids in the simulation within one time step.

The second solution was not much better. We decided to identify the

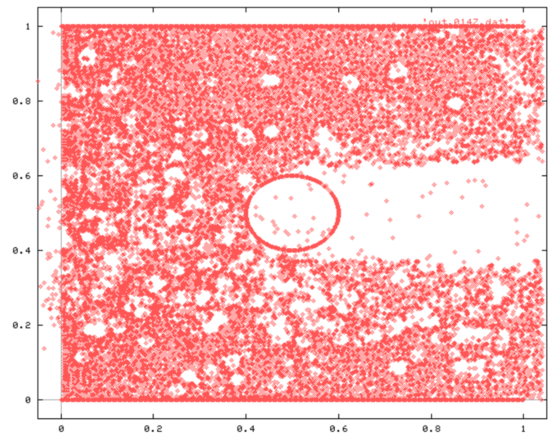

Figure 21: “Cancer” has just set in.

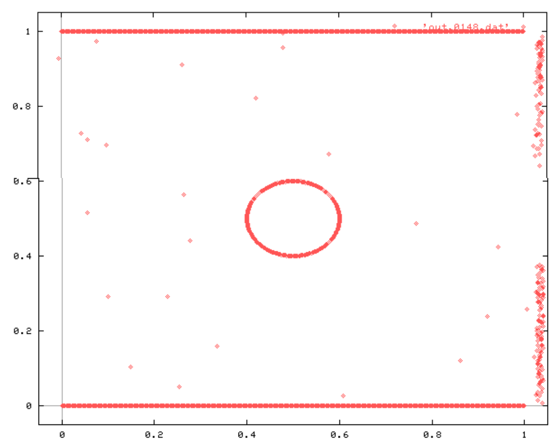

Figure 22: Same simulation as in figures (20) and (20), but at a few time steps later.

particle that caused the cancer from the beginning and watch its values throughout the simulation until they became erratic. It turned out that the particle went hay-wire once its position went negative or greater than 1.0. These numbers corresponded to the beginning and ending position of the Grid. Our solution was to delete particles as soon as they moved off of the Grid. The ones that moved off to the right, or $x > 1.0$ caused the program to crash because their position caused them to hash to a cell that did not exist. The ones with position $x < 0.0$ were a little more tricky. Consider the following scenario:

if $x = 0.1$ and $cellSize = 0.25$,

then $hash = (int)\left((\frac{x}{cellSize})\right) = (int)(\frac{0.1}{0.25}) = (int)(0.4) = 0$

Now assume that the particle has moved just to the left of the y-axis:

if $x = -0.1$ and $cellSize = 0.25$,

then $hash = (int)(\frac{x}{cellSize}) = (int)(\frac{-0.1}{0.25}) = (int)(-0.4) = 0$

So we see that the program would not crash when particles went less than zero, they would just be placed in cell with index 0. Actually, the fact that the cells thought that that particle belonged to them was not the critical problem. The reason is that the interaction depends on distance, and when it came time to interact, it would calculate the correct distance and thus only interact if it truly was a neighbor.

The real problem came down to the outer cells of the Grid. There are two main steps when finding a new field value for a particle. The first is to calculate all interactions with neighboring particles, and the second is to use this interaction value to determine what the new value will be. When calculating the interactions of particles, we loop over all cells except the outer ones. The reason was because they did not have as many neighbors, and we did not want edges to affect our solution. But when it came time to update each particle's values, we looped over all particles, even the ones in the outer cells. What this means is that a particle just inside the outer cells has its own interaction and update calculated correctly, but as soon as it goes into the outer cells, its interaction remains the same value as before and it continuously gets updated as if it is interacting the same for every time step. This caused its values to update without bound and thus behave un-physically. The final solution, and we feel the correct one, is to loop over all cells, even the outer ones, and calculate its interaction with all 8 surrounding cells. If a current cell does not have a neighbor, then it does not do anything for the interaction with that surrounding cell.

One other subtle problem is in initializing the flow. We attempted two methods in this version, but have settled on a different method in the next version. The two choices we came up with we called "pumping" and "pushing". In "pumping", all particles start off at rest and we initialize the far left column to have some initial velocity. We then let them go and interact with particles to the right. After k time steps, we introduce a new column of particles on the far left with the same initial velocity as before and let them go. We continue this for the entire simulation in the hopes that eventually all will end up with the same speed as the particles being pumped in. The problem is that the transient state is far too long and things seem unphysical for a fluid simulation.

The next choice of initialization was to "push" the particles in from the left. Basically, several columns to the left were identified as "pushing columns" and when particles belonged to these columns, their velocity remained constant, so that it would have the effect of slowly pushing the rest of the particles into motion. Once a particle left the pushing columns, it was free to interact with other particles and its velocity could change accordingly. Particles were introduced to the pushing columns in an identical manner as in the "pumping" technique just described.

Both methods are removed in the third version. The type of initialization of flow in version $3.x$ is based on work done by [Morris 1997], and this, along with new boundary conditions and a new viscosity, are the main reasons for a version $3.x$.

15 Version $3.x$

The physics of version $3.x$ follow that of [Morris 1997]. We introduced a new form for the viscosity and the way in which we represent and interact with boundary particles. The flow is initialized by applying a body force to each particle. We do not "pump" or "push" particles in this method, but instead use a body force and periodic boundary conditions, in which the particles see an infinitely long flow field and thus all of the objects seem to repeat themselves periodically.

This version is also fully functional with the Client. The Client can send commands to the server and the server executes those commands and sends back data to be displayed. The serial version has not been updated from its original version $1.x$ form. How to update the serial version is explained in the Future Work section.

Part VII
Tests

16 Poiseuille Flow

This first test is used to verify the correctness of our no-slip conditions. The simulation consists of two infinite plates at $y = 0$ and $y = L$. We model the plates being infinite by "wrapping" the particles around the ends. One way to think of this is to consider the fluid moving within a donut. Upon wrapping the particles, we keep all of their fluid properties the same except for the x position. The system starts off at rest and for times $t \geq 0$, we apply a body force in the x-direction. This has the effect of producing a pressure gradient. The exact values we used are $\nu = 10^{-6}\frac{m^2}{s}, L = 10^{-3}m, \rho = 10^3\frac{kg}{m^3}$, and $F_x = 10^{-4}\frac{m}{s^2}$. This corresponds to a peak fluid velocity of 1.25 x 10^{-5} and a Reynolds number of 0.0125.

We based this test on the one done in Morris's paper [Morris 1997]. He gives an analytical solution of the x velocity as a function of position along the width of the flow region and time:

$$v_x\left(y,t\right) = \frac{F}{2\nu}y\left(y-L\right) + \sum_{n=0}^{\infty}\frac{4FL^2}{\nu\pi^3\left(2n+1\right)^3} \sin\left(\frac{\pi y}{L}\left(2n+1\right)\right)exp\left(-\frac{\left(2n+1\right)^2\pi^2\nu}{L^2}t\right) \tag{77}$$

Figure (23) shows how our values matched up with the analytical values. Each pair of curves (analytical and numerical) represent a specific time in the simulation and all values come from the same group of particles. Figure (24) shows how the positions of this same column of particles change in time. Each curve in this figure relates back to a pair of curves in Figure (23).

Our initial results are not correct for several reasons. The first reason is that we are attempting to model a one dimensional problem (field values only change as a function of y) by using a 2D kernel. To fix this, we need

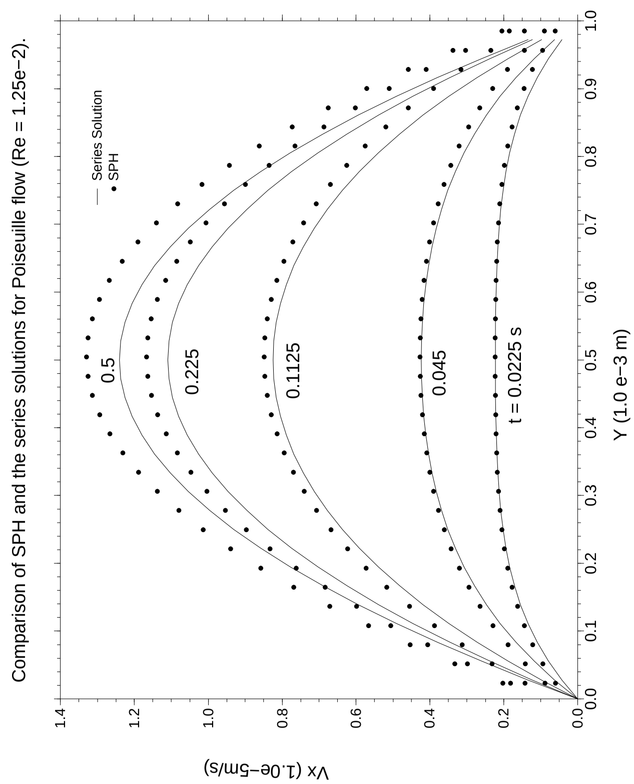

Figure 23: Poiseuille Flow: Y vs Vx

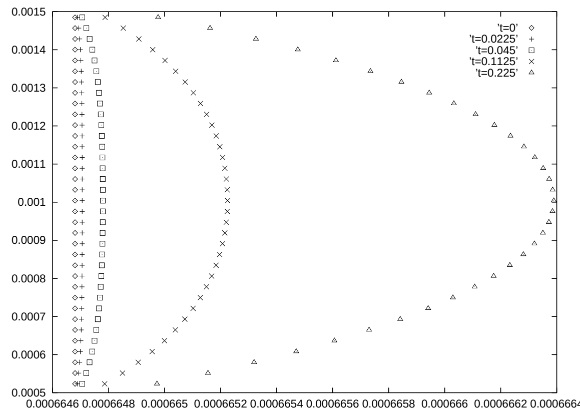

Figure 24: Poiseuille Flow: X vs Y

to re-normalize it and have each cell interact with other cells only to the top and bottom. Then the interacting distance should be a function of y position only. The second reason for our incorrect results is that our density is varying by more than 3%. This can be fixed by solving the relations in equation (45), testing our density variation and then adjusting the value for the speed of sound accordingly. We are currently making these changes and will insert our results as soon as they become available.

Part VIII
Conclusion and Future Work

17 Did We Meet Our Objectives?

The goals of this project were to produce a usable, robust, pedagogical tool for the fluid mechanics students and faculty at Oregon State University. The tool was to allow for the modification of physical parameters and distribution of cylinders as initial conditions and then to produce animations of the flow in the form of contour plots. In addition, this tool was to be written for serial, as well as parallel architectures supporting MPI.

The tool is written for parallel architectures, it does display contour plot animations, and it does allow for the modification of initial conditions.

So we accomplished some of our goals. The tool still needs a lot of work on visualization, robustness and testing of accuracy. More importantly, we feel that the current version is well designed and capable of being easily extended to model more complex systems, such as flow past objects other than cylinders.

18 Future Work

We will describe some of the things that should be implemented in the future.

1. Quintic Splines - Since the type of kernel affects how particles interact with each other, the introduction of the quintic spline kernel could offer more accurate results than the cubic spline kernel. The changes needed to include the quintic kernel in the program are to implement its equation into the SmoothParticle class and to modify which neighboring cells are interacted with in the particle interaction method of the SlaveGrid class. The buffering of rows would also have to be changed from one buffer on each side of a processor to two. The reason is that the interaction distance of a particle increase when the kernel is quintic from one cell away to two cells away.

2. Serial Version - The serial version needs to be updated so that it simulates the same fluid flow problems as the parallel version. In doing this, we would increase the portability of our Server program, since it could then be run on any machine with C++ compiler. The communication to the Client would have to be changed from the current socket connection to something else. This is needed only if both the Client and Server are running on the same machine.

3. Shapes in the Flow - Right now the program is pretty boring in the sense that the flow only interacts with the walls and displays the no-slip condition. Adding a cylinder will increase the usability of this program and should not cause too many problems in doing so. The main reason it is not used now is because there has not been time to fully test the program with cylinders.

4. Mouse Interactions - The addition of mouse interaction would also increase the usability. This interaction should consist of the initialization of shapes in the flow (*i.e.* position, size) by using a click-and-drag

mechanism. Once the mouse interaction is implemented, we could then use it to select streamlines and also to zoom into the display.

5. Visualization - The velocity field should be implemented in which each particle will be represented by an arrow with direction and relative magnitude dependent on the velocity at that point. Also, streamlines should be implemented. We do not have a thorough solution for this but it should be feasible since the Client will contain the raw data from the simulation. One solution would be to take the initial position of the streamline and map that to our velocity field. Then we can extend the streamline according to the direction and magnitude of each subsequent arrow. Finally, the ability to zoom in to the flow field should be implemented. This could, however, prove to be misleading since our data is finite and must be mapped to each pixel. This mapping involves weighting the value from a data point dependent on its distance from that pixel.

Part IX
Usage

The Client was tested on a Windows NT running JDK1.2Beta4. The minimum system requirement is JDK1.2 and Swing libraries. The Server was been tested on SWARM. The parallel version of the program (Client and Server) can be found at:
http://www.cs.orst.edu/*tilde* vertanen/project/
where *tilde* represents the symbol "tilde".

These files must then be untar'd, compiled, and executed. In order to run a simulation, one must compile the Client using the command:
javac sphclient.java
and then run it with the command:
java sphclient

The Client should pop up a window on your monitor. The Server must be compiled and run on its local machine. As of right now, the Server has only been tested on SWARM, thus we give the usage for that system. First log on to one of the "bees", we'll call it beeXY.cs.orst.edu. Now go into the directory with the Server program files and type:

make
mpirun -np "p" sph "portNumber"

where "**sph**" is the executable that comes from the "**make**" command. "**p**" is the number of processors you wish to run the simulation on. "**portNumber**" is the port the Server is listening on.At this point, we need to tell the Client where the Server is located by opening the Edit menu and then choosing Network. Here you can either load a predefined network connection or you can manually enter one. For our case of manually entering the network information, the Client must be informed that the Server is on beeXY.cs.orst.edu. It is important that both the Client and Server are listening on the same socket connection. Thus, type in the same value that was inserted in the **mpirun ...** command. Once the text fields are filled in, choose OK.

Now, the variables need to be initialized. This is done from the Edit menu again, but this time choose the Parameter option. A window comes up where you can enter each of the initial conditions manually, or load a file that contains all of these in it. When the initial conditions are entered, choose OK. If shapes are implemented, then go back into the Edit menu and choose World. There are buttons at the bottom to either Insert a shape, or again, one can load a file with the shapes already in it. Choose OK when done entering shapes.

Once the simulation is initialized, go to the Network Menu item and choose Connect. Once the connection is established, go to the Simulation menu item and choose Run New Simulation. The initial conditions have been sent to the Server and periodically, the Server will send back data to be displayed. To view the various visualizations, go to the View menu item and choose the type of data display you want. The data files will be displayed as they come and you can run through them by using the VCR commands at the bottom of the Client window.

If the program crashes while running, then the Client and Server must be restarted and the ports must be changed to some other value.

References

[Arken 1985] Arfken, G. *Mathematical Methods for Physicists. Third Edition* Academic Press, Inc.

[Batchelor 1967] Batchelor, G.K., *An Introduction to Fluid Dynamics*, (Cambridge Univ. Press, Cambridge, UK, 1967)

[Benz 90] Benz, W., Bowers, R.L., Cameron, A.G.W., Press, W.H., *Ap. J.*, vol. 348, (1990), p. 647.

[Evrad 1988] Evrad, A.E., MNRAS, vol. 235, (1988), p. 911.

[Gingold 77] Gingold, R.A., Monaghan, J.J., MNRAS, vol. 181 (1977), p. 375.

[Lattanzio 1985] Lattanzio, J.C., Monaghan, J.J., Pongracic, H., Schwarz, M.P., MNRAS, vol. 215, (1985), p. 125.

[Libersky 1993] Libersky, L.D., Petschek, A.G., Carney, C.C., Hipp, J.R., Allahdadi, F.A., *J. Comput. Phys.*, vol. 109, (1993), p. 67.

[Loewenstein & Mathews 1984] Loewenstein, M., Mathews, W.G., *J. Comput. Phys.*, vol. 62, (1984), p. 414.

[Lucy 77] Lucy, L. B., "A Numerical Approach to the Testing of the Fission Hypothesis.", *Astro. J.*, vol. 82 (1977), p. 1013.

[Monaghan 1982] Monaghan, J.J., SIAM *J. Sci. Stat. Comput.*, vol. 3, (1982), p. 42.

[Monaghan 1985] Monaghan, J.J., *Comput. Phys. Rep.*, vol. 3(2), (1985), p. 71.

[Monaghan 92] J.J. Monaghan, "Smoothed Particle Hydrodynamics", *Annu. Rev. Astron. Astrophys.*, vol. 30 (1992), pp. 543-574.

[Monaghan 94] J.J. Monaghan et. al., "Simulation of Free Surface Flows with SPH", ASME *Symposium on Computational Methods in Fluid Dynamics, Lake Tahoe, June 19-23, 1994*, http:///www-personal.monash.edu.au/ mecheng/mct/mct.html

[Monaghan & Gingold 83] Monaghan, J.J., Gingold, R.A., MNRAS, vol. 204 (1983), p. 715.

[Morris 1997] Morris, Joseph., *Journal of Computational Physics*, vol. 136, (1997), p. 214.

[Natanson 1960] Natanson, I.P., *Theory of Functions of a Real Variable.*, NewYork: Ungar

[Press *et al.* 1997] ress, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P., *Numerical recipes in C. The Art of Scientific Computing. Second Edition. Cambridge University Press.*

[Takeda 94] Hidenori Takeda *et al.*, "Numerical Simulation of Viscous Flow by Smoothed Particle Hydrodynamics", *Progress of Theoretical Physics*, vol. 92, No. 5, November (1994), pp. 939-960.

[Wallin URL] Wallin, John, Antunes, Sandy, Brugioni, John, Haque, Aamer, Milder, Seth, http://www.science.gmu.edu/ ahaque/math/sph.html

[White 94] White, F.M., *Fluid Mechanics*, McGraw-Hill:, Third Ed. p. 19.

[Wood 1981] Wood, D., MNRAS, vol. 194, (1981), p. 201.

[Vertanen 1999] Vertanen, Keith Master of Science Project June 1999,