APC524 Final Project:
Final Report

Anthony DeGennaro, Kevin Nowland, Scott Dawson and Imène Goumiri

18 January 2013

1 Project overview and goals

Broadly speaking, the aim of this project was to design a Smoothed Particle Hydrodynamics (SPH) solver, which would be capable of evolving a variety of types of fluid flows. This document will discuss the scope of the project, the principles and governing equations of SPH, and the specific features that were implemented, and the results from testing, profiling and running the code. Specifically, the goals of this project were to create a physically realistic, accurate, and efficient SPH solver, which is capable of simulating a variety of test cases, including:

- Compressible or (nearly) incompressible fluids (i.e. gases or liquids)
- Inviscid or viscous fluids of specified viscosity
- Flows governed by gravitational body forces
- The interaction/mixing between two fluids of different density

All of these aims were met, with the exception of simulating compressible flows (which would simply require the creation of another physics class in the current code).

2 Governing Equations

Before discussing the specifics of our implementation, we provide a brief derivation and discussion of the typical governing equations that our code will be solving. More information about such methods is can be found in [1] and [3], for example. To begin with, we consider the Euler equations

$$\frac{d\rho}{dt} + \rho\nabla\cdot(\boldsymbol{u}) = 0 \tag{1}$$

$$\frac{d\boldsymbol{u}}{dt} = -\frac{1}{\rho}\nabla P, \tag{2}$$

which respectively satisfy conservation of mass and momentum for a fluid, where ρ is density, P is pressure, and $\boldsymbol{u}$ is velocity. Here we use the total (Lagrangian) derivative $\frac{d}{dt} = \frac{\partial}{\partial t} + \boldsymbol{u}\cdot\nabla$, and for simplicity we are neglecting the influence of body forces (such as gravity) and dissipative forces (i.e. viscosity), though we note that these will be added to our code. Any computational fluids solver must approximate these equations in order to give a form which is amenable to numerical solution. To describe SPH, we begin by considering some domain Σ, which we parametrize by $\boldsymbol{r}$, and looking at some field $A(\boldsymbol{r})$ on this domain (which typically could be a scalar field such as ρ, or vector field such as $\boldsymbol{u}$. We can first make the trivial observation that

$$A(\boldsymbol{r}) = \int_{\Sigma(\boldsymbol{s})} A(\boldsymbol{s})\delta(\boldsymbol{r}-\boldsymbol{s})dV. \tag{3}$$

The main idea behind SPH is that we can approximate the Dirac-delta function by some smooth function W, which we refer to as a kernal, satisfying the conditions

$$\int_{\Sigma} W(\boldsymbol{r},h)dV = 1, \quad \text{and} \quad \lim_{h\to 0} W(\boldsymbol{r},h) = \delta(\boldsymbol{r}).$$

Here h can be thought of as smoothing length, which governs the resolution of the simulation. With this approximation, 3 becomes

$$A(\boldsymbol{r}) = \int_{\Sigma(\boldsymbol{s})} A(\boldsymbol{s})W(\boldsymbol{r}-\boldsymbol{s},h)dV. \tag{4}$$

We proceed by making the further approximation of replacing this integral by a finite sum, where we discretise the domain into volume elements of size $(\Delta V)_i$ at location $\boldsymbol{r}_i$, and sum over these elements rather than integrating. Unlike fixed grid based solvers, in SPH we let these volume elements move with the fluid itself, so we may think of them as discrete particles or 'clumps' of fluid. We now have

$$A(\boldsymbol{r}) = \sum_i A(\boldsymbol{r}_i)W(\boldsymbol{r}-\boldsymbol{r}_i,h)(\Delta V)_i = \sum_i A(\boldsymbol{r}_i)W(\boldsymbol{r}-\boldsymbol{r}_i,h)\frac{m_i}{\rho_i}, \tag{5}$$

where $m_i = \rho_i(\Delta V)_i$ is the mass of fluid particle i. In order to approximate the spatial gradient of the field A at the location of a given fluid particle a, we have

$$\nabla A(\boldsymbol{r})|_{\boldsymbol{r}_a} = \frac{\partial}{\partial \boldsymbol{r}} \sum_i A(\boldsymbol{r}_i) W(\boldsymbol{r} - \boldsymbol{r}_i, h) \frac{m_i}{\rho_i} = \sum_i A(\boldsymbol{r}_i) \frac{m_i}{\rho_i} \nabla_a W_{ai}, \tag{6}$$

where we use the shorthand notation $W_{ai} = W(\boldsymbol{r}_a - \boldsymbol{r}_i, h)$, and ∇_a indicates that we are taking the gradient with respect tot the coordinates of a.

The computation of the relevant fields over time (such as ρ and $\boldsymbol{u}$) then involves computing the quantities in equations 1 and 2 by these approximations. For conservation of mass, we have:

$$\begin{aligned}
\frac{d\rho}{dt} &= -\rho \nabla \cdot \boldsymbol{u} \\
&= -\nabla \cdot \rho \boldsymbol{u} + \boldsymbol{u} \cdot \nabla \rho, \quad \text{thus for particle } a, \\
\frac{d\rho_a}{dt} &= -\sum_i \frac{m_i}{\rho_i} (\rho_i \boldsymbol{u}_i) \cdot \nabla_a W_{ai} + \boldsymbol{u}_a \cdot \sum_i \frac{m_i}{\rho_i} \rho_i \nabla_a W_{ai} \\
&= -\sum_i m_i (\boldsymbol{u}_i - \boldsymbol{u}_a) \cdot \nabla_a W_{ai}. \qquad (7)
\end{aligned}$$

Similarly for conservation of momentum:

$$\begin{aligned}
\frac{d\boldsymbol{u}}{dt} &= -\frac{1}{\rho} \nabla P \\
&= -\frac{P}{\rho^2} \nabla \rho - \nabla \left(\frac{P}{\rho} \right), \quad \text{and so} \\
\frac{d\boldsymbol{u}_a}{dt} &= -\frac{P_a}{\rho_a^2} \sum_i \frac{m_i}{\rho_i} \rho_i \nabla_a W_{ai} - \sum_i \frac{m_i}{\rho_i} \frac{P_i}{\rho_i} \nabla_a W_{ai} \\
&= -\sum_i m_i \left(\frac{P_a}{\rho_a^2} + \frac{P_i}{\rho_i^2} \right) \nabla_a W_{ai}. \qquad (8)
\end{aligned}$$

We note that changing the form of 1 and 2 before applying our approximations gives a resulting formulation with smaller error than if we had applied them directly. To add viscous damping to the simulation, equation 8 becomes

$$\frac{d\boldsymbol{u}_a}{dt} = -\sum_i m_i \left(\frac{P_a}{\rho_a^2} + \frac{P_i}{\rho_i^2} + \Pi_{ai} \right) \nabla_a W_{ai}, \tag{9}$$

where the viscosity Π_{ai} takes the form [2]:

$$\Pi_{ai} = \begin{cases} \frac{-\alpha\mu_{ai} + \beta\mu_{ai}^2}{\bar{\rho_{ai}}} & \text{if } \boldsymbol{v_{ai}} \cdot \boldsymbol{r_{ai}} < 0 \\ 0, & \text{otherwise,} \end{cases} \tag{10}$$

where $\mu_{ai} = \frac{h\mathbf{v}_{ai}\cdot\mathbf{r}_{ai}}{\mathbf{r}_{ai}^2+\eta^2}$, and α, β and η are parameters which depend on the fluid being simulated (typically $\beta = 0$ for an incompressible fluid, for example). It can similarly be shown that the SPH equivalent of the energy equation (again for an inviscid fluid) is

$$\frac{d\boldsymbol{e}_a}{dt} = -\frac{P_a}{\rho_a}\sum_i \frac{m_i}{\rho_i}(\boldsymbol{u}_i - \boldsymbol{u}_a)\cdot\nabla_a W_{ai}, \tag{11}$$

where $\boldsymbol{e}_a$ is the internal energy of fluid particle a, though when the fluid considered is incompressible, the energy is decoupled from the momentum equation (as pressure is independent of temperature), so is not required. It should be noted here that the most relevant and accurate formulation and implementation of these equation will be dependent on the type of simulation being performed. For example, in the case of simulating a liquid, which typically has negligible density variations, and equation of state of the form

$$P = B\left(\left(\frac{\rho}{\rho_0}\right)^\gamma - 1\right), \tag{12}$$

can typically be used, where ρ_0 is a reference density, $\gamma \sim 7$, and B is chosen such that density fluctuations are sufficiently small [3]. For simulations that involve rigid boundaries, the boundary conditions can be enforced through the creation of fixed particles along the boundary, which exert a prescribed force on the fluid particles. In our implementation, a Lennard-Jones potential force is imposed between the boundary and fluid particles, which is of the form

$$f(r) = D\left(\left(\frac{r_0}{r}\right)^{p_1} - \left(\frac{r_0}{r}\right)^{p_2}\right)\frac{\boldsymbol{r}}{r}, \tag{13}$$

where r_0 is a typical particle spacing (such as the smoothing length), and p_1 and p_2 are conventionally chosen to be 12 and 6 respectively [2]. One downside of this implementation is that the boundary is not entirely smooth. This could be corrected for through modifying the code so that each boundary particle is also assigned a normal direction vector [3], but this is beyond the scope of the present code.

In our proposed code, the equations 7, 8, and 11 (or a subset thereof) will be stepped forward in time from prescribed initial conditions in order to simulate the motion of the fluid with time. In practice, and in very broad terms, this involves summing up the contributions from other particles in order to determine the changes in properties of a given particle for each timestep.

3 Program Structure

The basic program structure is shared by many integrators, and involves an initial set up process followed by a loop which at each iteration advances the physical properties and writes the current state to an output file.

The initialization process involves the set up of the initial configuration of the fluid by reading in fluid particles and their properties from one file, and if provided, the boundary particle positions from a separate file.

The timestep loop first involves determining which particles are within the range of influence of other particles. The physical properties are then advanced by the procedure outlined above. The integrations can be performed by any desired method of integration. The methods provided are Euler's method, a modified Euler method, and a predictor-corrector method. Particle positions are outputted at regular intervals (which can be specified in the driver program `sph`) to `fluid.dat`.

3.1 User Interface: Input

The parameters needed to specify a particle type in the SPH method include fluid properties (such as mass, density, pressure, and energy), kinematic properties (such as position and velocity), and particle discretization properties (i.e. a particle discretization kernel function). Our main goal in designing the user interface was to develop an input protocol which would be easy to use while still allowing the user to specify many fluid properties directly. The solution we developed is quite simple: the user creates an input text file in which each of the fluid particles (along with their initial conditions) is directly specified (boundary particle locations are specified in the same way via an additional file). The proper file syntax is shown below:

"INPUT.DAT"

```
    [Total Number of Particles]

    [Number] x y u v [Mass] [Density] [Pressure] [Energy]
                                 . . . . . . .
                                 . . . . . . .
```

Here, the particle numbering should begin at zero, and x, y, u, v specify the initial position and velocity components in a 2-D plane. The user may obviously generate the initial conditions in these input files by means of whatever pre-processing routines they desire (for example, a simple MatLab script to generate specific initial/boundary conditions often is a quick and nice solution).

Once made, the driver program `sph` can be called with the syntax

```
sph <initFile> <boundaryFile(OPTIONAL)> <tfinal> <timestep>
               <integratorType> <kernelType>
```

The data files `initFile` and `boundaryFile` should be in the format shown above. The options for `<intergratorType>` are `euler`, `eulermod`, and `pc`, while the options for `<kernelType>` are `spline` and `gaussian`. The specifics of these various implementations are discussed in section 3.3.

3.2 User Interface: Output

Output is handled in a manner which closely resembles the input protocol. Specifically, at each timestep, the SPH code writes the full state information of each fluid particle to a pre-specified output file. The output syntax is as follows:

"OUTPUT.DAT"

```
[Timestep 1]

[Number] x y u v [Mass] [Density] [Pressure] [Energy]
                      . . . . . . .
                      . . . . . . .

[Timestep 2]

[Number] x y u v [Mass] [Density] [Pressure] [Energy]
                      . . . . . . .
                      . . . . . . .
```

3.3 Classes

Perhaps the must fundamental class is `Particle`, which holds the state of a current particle and functions to get and change that state. Each particle is given a unique tag, and the particles have an array which contains the list of tags for neighboring particles. The included properties are x and y position, the velocities, u and v respectively, in those directions, the mass of the particle, associated viscosity, as well as density and pressure at the particle's position.

A vector of particles is included in the `Fluid` class, which contains the overall state of the fluid. The `Fluid` also has a vector for boundary particles. As this is smoothed particle hydrodynamics, the particles are smoothed out with the influence described by the kernel function. This function, a cubic spline or Gaussian in the provided code, is passed to `Fluid`. Aside from the usual get and set functions, an important function is `Fluid::findNeighbors()`, which determines for each particle which particles are close enough to be of influence based on the given smoothing length.

This `findNeighbors()` function is important for scaling, as the naïve comparison of each particle to every other is an $O(n^2)$ process. To overcome this, a mesh is temporarily imposed based on the extreme x and y positions of the particles with each cell having height and width of the smoothing length. Each particle is then placed in a grid cell. Particles are then neighbors if they share a grid cell or lie in adjacent cells. As long as the smoothing length is small enough, uniform, and the particles are not densely packed, this is an $O(n)$ process. If all particles lie in two adjacent cells, then this can be $O(n^2)$, but for incompressible fluids and suitably small smoothing lengths, this should only occur in unphysical locations.

The `Fluid::findNeighbors()` function is the first step of the time step loop. The `Integrator` class and its step function is then invoked. The integrator advances the fluid properties.

The `Integrator` is passed an instance of the `Physics` class, which contains the relevant processes to advance the fluid properties each timestep. In the case of an incompressible fluid, these are the velocity, position and density of each fluid particle. The only physics type contained within the present version of the code is `IncompVisc`, which can be used for incompressible viscous flows. The user specifies the relevant physical paraters of the fluid - namely the coefficients in the equation of state and viscosity calculation. These have been set in `sph.cc` to typical values for water. Also inputted is the desired smoothing length, which should be about twice the typical particle spacing, and the appropriate gravitational force. Originally we also had a `IncompInvisc` physics implementation, but this has been removed as an inviscid fluid may be simulated simply by setting the viscosity parameter to zero in `IncompVisc`.

Three types of integrator are provided:

- `euler` A standard forward stepping Euler routine
- `eulermod` Using the Euler method, but updating particle positions using the already updated velocity values.
- `predictorcorrector` A predictor corrector algorithm, based on that outlined in [4].

When chosing the appropriate timestep, the user must keep in mind that simulations will not necessarily 'break' if the timestep is too large to be accurate, so it is important to verify that the timestep chosen is sufficiently small. This can be attained either by ad-hoc convergence testing, or with reference to the Courant condition, which states that the maximum timestep to retain accuracy is [4]

$$dt_c = C_{cour} \min\left(\frac{h}{v}\right),$$

where C_{cour} is a constant that is often taken to be 0.4, and h and v are typical relative particle spacings and velocities, respectively. In reality, effects such as interactions with boundaries can interfere with the validity of this condition, so we reccomend that the user verifies the accuracy of their chosen timestep. As a rule of thumb, a timestep of 0.001 when using the `euler` method is more than sufficient for all of the testcases provided (which can be increased for the `predictorcorrector` method).

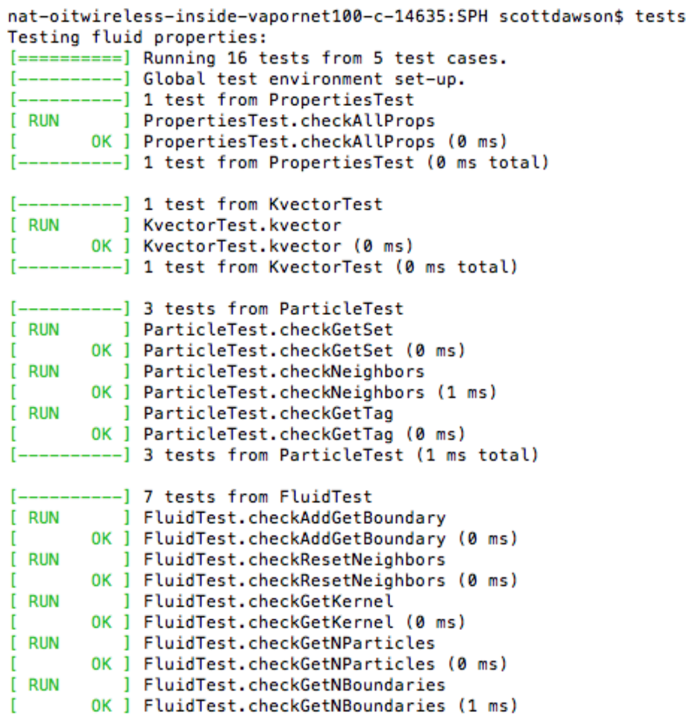
Also included is an `Output` class which is solely used to output the state of the fluid to an appropriate file.

4 Testing

Gtest was used to write tests designed to ensure that functions of the code operate as expected. Tests.cc is a driver program which runs all of the tests (which is built along with the rest of the code by the Makefile. Figure fig:tests shows a snippet of the output from running the tests driver program.

The tests focused on the classes `Particle`, `Fluid`, and `Kernel`. These classes were easiest to test as the member functions were simple relative to those in `Physics` and `Integrator`. The integrators and and types of physics had more complicated member functions which were most easily tested through simulation once the other classes were deemed operational.

Once the class structure and interfaces were determined, the tests were written and did catch several bugs. The tests proved to be more useful when



```
nat-oitwireless-inside-vapornet100-c-14635:SPH scottdawson$ tests
Testing fluid properties:
[==========] Running 16 tests from 5 test cases.
[----------] Global test environment set-up.
[----------] 1 test from PropertiesTest
[ RUN      ] PropertiesTest.checkAllProps
[       OK ] PropertiesTest.checkAllProps (0 ms)
[----------] 1 test from PropertiesTest (0 ms total)

[----------] 1 test from KvectorTest
[ RUN      ] KvectorTest.kvector
[       OK ] KvectorTest.kvector (0 ms)
[----------] 1 test from KvectorTest (0 ms total)

[----------] 3 tests from ParticleTest
[ RUN      ] ParticleTest.checkGetSet
[       OK ] ParticleTest.checkGetSet (0 ms)
[ RUN      ] ParticleTest.checkNeighbors
[       OK ] ParticleTest.checkNeighbors (1 ms)
[ RUN      ] ParticleTest.checkGetTag
[       OK ] ParticleTest.checkGetTag (0 ms)
[----------] 3 tests from ParticleTest (1 ms total)

[----------] 7 tests from FluidTest
[ RUN      ] FluidTest.checkAddGetBoundary
[       OK ] FluidTest.checkAddGetBoundary (0 ms)
[ RUN      ] FluidTest.checkResetNeighbors
[       OK ] FluidTest.checkResetNeighbors (0 ms)
[ RUN      ] FluidTest.checkGetKernel
[       OK ] FluidTest.checkGetKernel (0 ms)
[ RUN      ] FluidTest.checkGetNParticles
[       OK ] FluidTest.checkGetNParticles (0 ms)
[ RUN      ] FluidTest.checkGetNBoundaries
[       OK ] FluidTest.checkGetNBoundaries (1 ms)
```

Figure 1: Screenshot of the output of running the gtest tests.

changing implementations of features to make sure the code would still run properly. This was particularly important when rewriting `Fluid::findNeighbors()`.

5 Profiling and Optimization

Profiling was done using `gprof`. Table 1 shows the routines which took up more than 1% of runtime when running the program with 650 input particles and a box boundary. The routines which dominate the program's runtime are the calculations for the physics and various gets and sets. We were encouraged by this profile, as we found the gets and sets to largely be necessary in object-oriented code. This does suggest that a different repackaging of the properties to avoid these could significantly reduce runtime.

The other main use of profiling was to determine the scaling properties of `Fluid::findNeighbors()`. Initially, each particle's position was compared

% time	cum. seconds	self seconds	name
36.61	33.61	33.61	IncompVisc::rhs
25.23	56.77	23.16	Fluid::getParticles
18.88	74.10	17.33	Fluid::findNeighbors
9.26	82.60	8.50	Fluid::getBoundaries
5.36	87.52	4.92	Particle::getOldProperties
2.06	89.41	1.89	SplineKernel::gradW
1.38	90.68	1.27	Particle::deleteNeighbors

Table 1: Table containing sample `gprof -b sph` output.

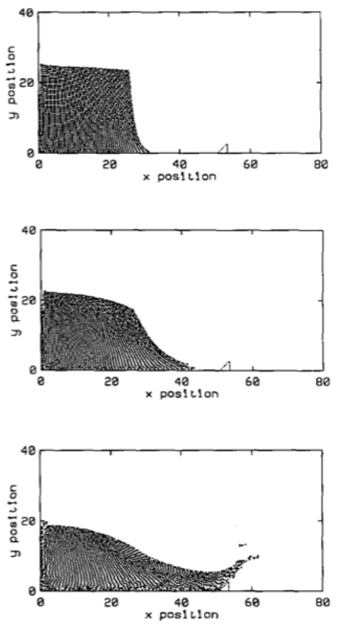
to the position of every other particle to determine possible influence. This is an $O(n^2)$ algorithm. In profiling, it was found that this function for a large number of particles took up the plurality (greater than 30%) of runtime. After switching to the mesh method described above, find neighbors scaled linearly, as expected.

Initially, we planned to reduce runtime by parallelizing many of the calculations. In fact, the inducement of the mesh in the `findNeighbors` method suggests that one way to do this would be to partition the domain based on the temporary grid. This would allow not only for the easy OpenMP parallelization of for loops, but to use the more versatile MPI.

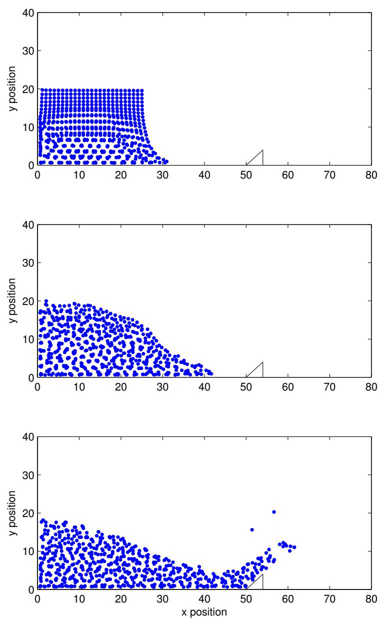
6 Results

Here some typical results from running the code in a variety of scenarios are presented. We focus on applications particularly suited to particle-based codes, such as flows with highly deforming free surfaces and mixing of fluids with different densities. We begin with a comparison with an example from the literature - a dam breaking and flowing over a triangular obstacle [2]. Figure 2 shows the comparison between the results from both the previous investigation and our own code.

Figures 3 and 4 respectively show the results of simulating a sphere of fluid falling onto an inclined plane, and two fluids of different densities mixing in a box.



(a)



(b)

Figure 2: Comparison between SPH results for a breaking dam flowing over a triangular obstacle from (a) the results of [2], and (b) our code.

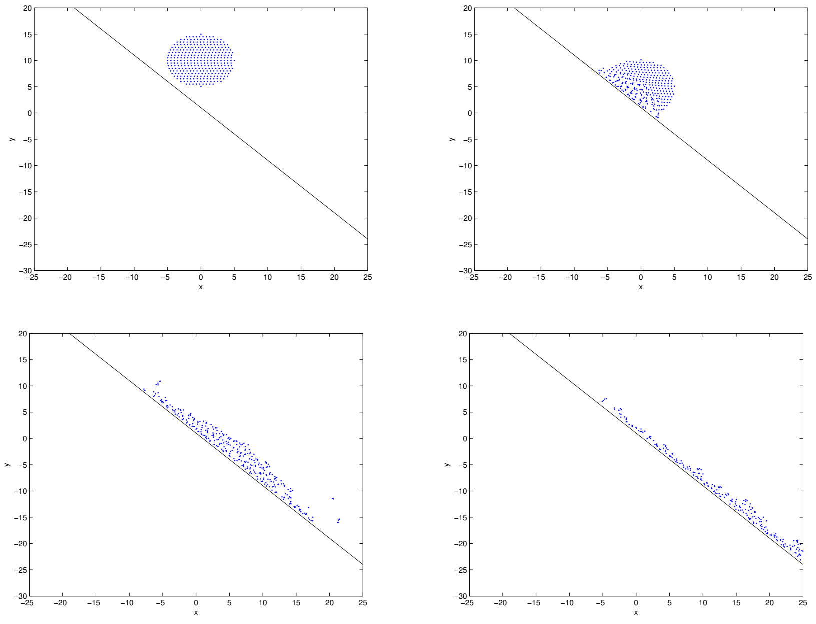


Figure 3: Simulation of a sphere impinging on a 45 degree inclined plane

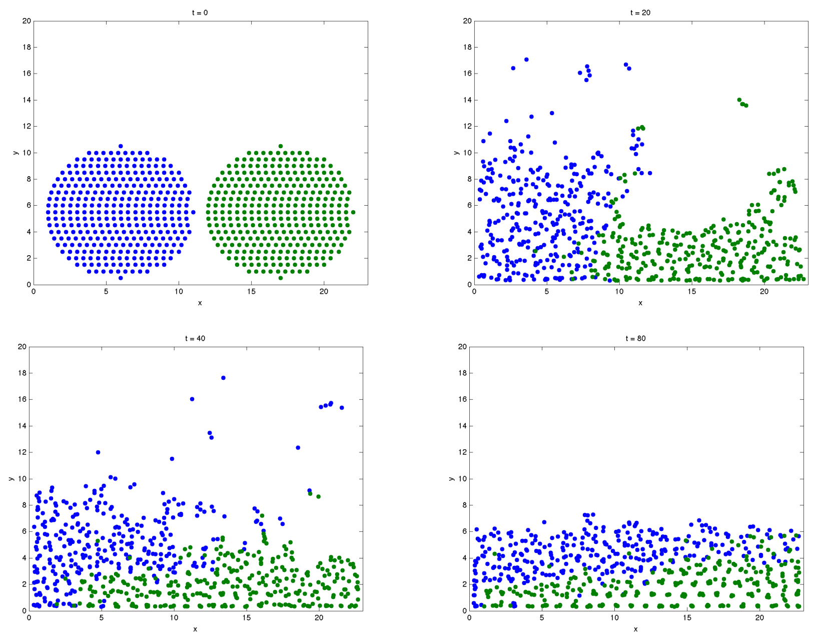


Figure 4: Simulation of two fluids of different densities mixing. The blue fluid is half the density of the green.

7 Design Process and Lessons learned

This section documents the process that went through in designing and writing our code. We began by talking through the main ideas of the project, in terms of the scope, the physics involved and the program architecture that we planned to use. We divided up the initial workload so that each person would have a class to begin coding. In addition to this, a single file, proof of concept code was written to quickly put together the main features and variables that our code would use. Upon writing each of the individual classes, we worked together to ensure that all could interface correctly with each other. This often involved rewriting parts of the code, or adding useful features that were originally overlooked. As the code became more complete, this phase transitioned into debugging and testing of the code as a whole. As perhaps is to be expected, many errors and limitations within our code emerged. Some of these were simple to fix, while others were significantly more elusive.

The act of writing tests was useful not only to uncover previously undetected errors and bugs, but also to solidify our understanding of the intended functionality of each aspect of the code

Pointers tend to proliferate in object-oriented code, and memory management, keeping track of what existed where and detecting memory leaks, became quite difficult after a time. This led to a wholesale switch to using the `boost` library's `shared_ptr`. This allowed us to remove all calls to `delete` and prevented memory leaks. The main downside for this is that `boost` is large and may not be installed everywhere, somewhat hurting the portability of the code. When `C++11` becomes fully standard and supported, similar functionality will be available without having to install a separate library.

8 Further Work

Possible extensions of the present code could include:

- Creation of alternative physics classes (such as to allow for compressible flows)
- Paralization of the code, which could involve either using OpenMP to parallalize the for loop of the timestepper, for example, or using MPI with an extension of the domain decomposition technique that was used for the findneighbors algorithm

- Modify the code so that some functions (e.g. updating neigbors) are not performed every timestep
- Extension to three dimensional flows (which would require significantly more work, and would probably want to happen after the previous step)

References

[1] J.J. Monaghan. Smoothed particle hydrodynamics. *Ann. Rev. Asntron. Astrophys.*, 30:543–574, 1992.

[2] J.J. Monaghan. Simulating free surface flows with sph. *J. Comput. Phys.*, 110:399–406, 1994.

[3] J.J. Monaghan. Smoothed particle hydrodynamics. *Rep. Prog. Phys.*, 68:1703–1759, 2005.

[4] D.J. Price. Magnetic fields in astrophysics, phd thesis. *University of Cambridge*, 2004.