ETH Zürich
Computational Statistical Physics
Dr. Andreas Adelmann

Smoothed Particle Hydrodynamics
Programming report

Radovan Dabetić, Andrés Tamargo, Vincent Boulard, Marc de Miguel

Abstract

In this project we attempt to implement a smoothed particle hydrodynamics code in `C++` using IPPL. To do this, we implement a code which assumes a constant smoothing length and do a nearest-neighbor search using a mesh-based approach. The performance is demonstrated using test cases like the shock tube and a Kelvin-Helmoltz instability. We explain the drawbacks of using a constant smoothing length and elucidate further improvements on the algorithms which were implemented.

1 Introduction

Marc de Miguel

Fluid dynamics simulations are key for many areas of science, including aerodynamic or thermal engineering, but also astrophysics, in order to studying phenomena that may take thousands of years in reality. In this context, there is a need for boundary-free methods, with also the possibility of obtaining overall properties such as density or pressure. A wide range of algorithms are available to tackle these problems. In this project we implement **Smoothed Particle Hydrodynamics** (SPH), a meshless Lagrangian method that discretizes the fluid as particles. We will see that these particles can be interpolated to obtain the smoothed properties of the fluid, hence the name.

In order to implement the SPH scheme, we will use the C++ library IPPL [5] (Independent Parallel Particle Layer), which provides a set of tools for portable particle simulations. Its main advantage is that it includes platform-independent parallelization of the simulations, which becomes a requirement when scaling these kind of algorithms.

Below, we detail the main theoretical aspects of SPH and also describe our implementation. One of the challenges apart from implementing the classical SPH, was to find a method of obtaining the nearest neighbours of particles with low computational complexity, which we also explain in Section . We showcase the usefulness of our code with some classical examples, such as the Sod Shock Tube or visualizing the Kelvin-Helmholtz instability.

All the codes used can be found at `https://github.com/Mipoza/CSP10-SPH/tree/main`.

2 Theory description

Marc de Miguel, Andrés Tamargo

2.1 Kernel Interpolation

At the core of SPH, we find what are called Kernel interpolants or Kernel functions. One can understand such objects by considering the usual Dirac delta integral relation:

$$f(\mathbf{r}) = \int \delta(\mathbf{r}-\mathbf{r}')f(\mathbf{r}')d^N\mathbf{r}' \tag{1}$$

and taking the following approximation:

$$f(\mathbf{r}) \simeq \int W(\mathbf{r}-\mathbf{r}', h)f(\mathbf{r}')d^N\mathbf{r}' \tag{2}$$

such that $\lim_{h\to 0} W(\mathbf{r}-\mathbf{r}', h) = \delta(\mathbf{r}-\mathbf{r}')$, where the parameter h is called the *smoothing length.* In Section 3.3 we describe how a variable smoothing length can also be implemented. These interpolants are used to compute physical quantities which are smoothed over the neighboring variables. The kernel can be chosen from a wide variety of radial functions (eg. Gaussian), which must be at least of class C^2, and often a function with finite support is chosen. In our project, we will use the following kernel (normalized for 2D):

$$W(\mathbf{r},h) = \frac{10}{7\pi h^2}\begin{cases}\frac{1}{4}(2-q)^3, & 1 \leqslant q \leqslant 2\\ 1-\frac{3}{2}q^2(1-\frac{q}{2}), & 0 \leqslant q \leqslant 1\\ 0, & q > 1\end{cases}, \text{ with } q = \frac{|\mathbf{r}|}{h} \tag{3}$$

The next step in this formalism is to discretize the integral in Equation 2 and turn it into a summation through all the particles positions. To do this, we can take the volume element as $d^N\mathbf{r}_j \rightarrow \frac{m_j}{\rho(\mathbf{r}_j)}$ for each particle j. With this in mind, the obtain the following formulation of the kernel function:

$$f(\mathbf{r}) \simeq \sum_j \frac{m_j}{\rho(\mathbf{r}_j)} f(\mathbf{r}_j) W(\mathbf{r}-\mathbf{r}_j, h) \tag{4}$$

From this expression, we can get the first important expression in terms of the kernel function:

$$\rho(\mathbf{r}) = \sum_j m_j W(\mathbf{r}-\mathbf{r}_j, h) \tag{5}$$

which is actually an intuitive expression. Equation 5 just shows that in the kernel interpolant formalism, the density of the system of particles is understood as the sum of the particles masses weighted by the kernel function centered at the particle j of the summation. Note here that we can compute the density for any point in space, and we will denote as ρ_i when referring to the density at the position of particle i.

Starting from the Euler equations for an inviscid gas, a Lagrangian formulation can be used to derive the discrete equations of motion for each particle, that take the following form [6]:

$$\frac{d\mathbf{v}_i}{dt} = -\sum_j m_j \left(f_i \frac{P_i}{\rho_i^2} \nabla_i W_{ij}(h_i) + f_j \frac{P_j}{\rho_j^2} \nabla_i W_{ij}(h_j)\right) \tag{6}$$

with $f_i = \left(1 + \frac{h_i}{3\rho_i}\frac{\partial \rho_i}{\partial h_i}\right)^{-1}$, which will be equal to 1 if we are considering a constant smoothing parameter.

If we want to compute the pressure, either the entropy or the energy density need to be calculated, for the moment we have a model with constant entropy. Now we can compute the pressure by using an adiabatic index, which can be set depending on the fluid to be modeled:

$$P_i = A_i \rho_i^{\gamma} \tag{7}$$

This set of equations (5-7) will determine the evolution of the system and they fulfill the conservation of energy, momentum and angular momentum.

One other thing to mention is the choice of the constant smoothing parameter. Throughout our implementation we consider it constant, making also the $f_i = 1$ by not having the $\frac{\partial \rho}{\partial h}$ terms, which will be something that would be interesting to change in the future. As we will explain later, the smoothing length is one of the crucial parameters in SPH and it has been proved that having an approximately constant number of neighbours improves greatly the convergence.

2.2 Artificial viscosity

In some cases, the Euler equations can produce discontinuities as in shock waves, where the properties of the fluid change instantly. To avoid these problems, we introduce a small artificial viscosity term that will allow dissipation and an increase in entropy if shock waves occur. The viscous term added is:

$$\left.\frac{d\mathbf{v}_i}{dt}\right|_{\text{visc}} = -\sum_{j=1}^{N} m_j \Pi_{ij} \nabla_i \overline{W}_{ij} \tag{8}$$

with $\overline{W}_{ij} = \frac{1}{2}(W_{ij}(h_i) + W_{ij}(h_j))$ and $\mathbf{v}_{ij} = \mathbf{v}_i - \mathbf{v}_j$.

Now we also need to take into account the change in entropy of the particles that now is not conserved, according to:

$$\left.\frac{dA_i}{dt}\right|_{\mathrm{visc}} = \frac{1}{2}\frac{\gamma-1}{\rho_i^{\gamma-1}}\sum_{j=1}^{N} m_j \Pi_{ij}\mathbf{v}_{ij}\nabla_i\overline{W}_{ij} \tag{9}$$

The viscosity term may take different forms, in this project we used the version introduced in [4].

After the addition of viscosity, the equation of motion that we will use is the following (considering a constant smoothing length):

$$\frac{d\mathbf{v}_i}{dt} = -\sum_j m_j\left(\frac{P_i}{\rho_i^2}\nabla_i W_{ij}(h_i) + \frac{P_j}{\rho_j^2}\nabla_i W_{ij}(h_j)\right) - \sum_{j=1}^{N} m_j \Pi_{ij}\nabla_i\overline{W}_{ij} \tag{10}$$

3 Numerical implementation

3.1 Nearest-Neighbor Problem

Radovan Dabetić

In order to reduce the computational complexity of $\mathcal{O}(N^2)$, the compact support of the smoothing kernel needs to be exploited. The idea is to only sum over the particles which have a non-zero contribution, meaning that one needs to find all particles within a certain radius of the chosen particle. In our case, we can make a few key simplifying assumptions about this problem:

- The smoothing kernel parameter is constant.
- The particles are located within the same bounded domain of space at all times.

In the following paragraphs, an algorithm will be described which finds an approximate list of all particles within a radius of any given point. It is based on the chaining mesh in [2] and also on [1].
Let P be the particle of which the neighbors are to be found and M a mesh with the mesh-width equal to the smoothing kernel length, covering the computational domain. Then it is sufficient to find all the particles which are located in and around the cell of M within which the particle P is located.
For finding the cell in M which contains the particle, we round the coordinates. More specifically, given the position $\mathbf{x} = (x_1, \ldots, x_d)$, the origin (i.e. lower left corner) $\mathbf{x}_0$, the number of mesh cells N_i in each direction as well as the extent L_i of the mesh in each direction, define the "coordinates" in the mesh $\mathbf{c}_P \in \mathbb{N}^d$ as

$$\mathbf{c}_P^i = \texttt{cell\_idx}(\mathbf{x})_i := \left\lfloor N_i \frac{x_i - x_{i,0}}{L_i} \right\rfloor.$$

This gives a d-tuple of indices, which can be mapped to a d-dimensional array.
Now one can go through all the particles and put them in their corresponding cells, which takes $\mathcal{O}(N)$ time. Then one can find the neighbors of P by simply retrieving the particles in the adjacent cells to P, which might take a variable amount of time depending on the distribution of particles, but that varies regardless of the method we choose (assuming the smoothing kernel parameter is constant).
This method sounds simple in principle, however, it is not entirely trivial to implement and there are many implementation details which were skipped, as we will see in the section below about code design.

3.2 Numerical integration

Vincent Boulard

In this section, we explore the numerical methods applied to solve the equations of motion (EOM) derived from the smoothed-particle hydrodynamics framework. It is important to understand that the core idea behind SPH is to transform a partial differential equations problem into a set of ordinary differential equations that we then integrate over time.

3.2.1 Framework

We have seen that the density for each particle is computed using the equation:

$$\rho_i = \sum_j m_j W_{ij}, \tag{11}$$

where $W_{ij} = W(r_{ij})$, and r_{ij} is the distance between the i-th and j-th particles. Then we know that the acceleration of each particle is derived from the gradients of pressure and density as:

$$\frac{d\mathbf{v}_i}{dt} = -\sum_j m_j \left(\frac{P_i}{\rho_i^2} + \frac{P_j}{\rho_j^2} \right) \nabla W_{ij}, \tag{12}$$

This set of ordinary differential equations is what we need to integrate over time to simulate fluid motion, having the trajectories of each individual particle.

3.2.2 Numerical Scheme

The integration of these ODEs is carried out using a time-stepping method where the global time step, Δt, is selected based on the problem's specific parameters to ensure stability and accuracy. In cases where the problem features strong inhomogeneities in density, it could be beneficial to clusterize the particles and assign different time steps to each cluster, corresponding to their different characteristic times of evolution. However, for simplicity and due to time constraints, we are employing a single global time step in this work.
The Leapfrog integration scheme, noted for its symplectic properties and effectiveness in long-term numerical integration of dynamical systems, is utilized:

$$\mathbf{v}\left(t + \frac{\Delta t}{2}\right) = \mathbf{v}(t - \frac{\Delta t}{2}) + \mathbf{a}(t)\Delta t, \tag{13}$$

where $\mathbf{v}(t)$ and $\mathbf{a}(t)$ denote the velocity and acceleration of the particle at time t, respectively. This scheme updates velocities at half-step intervals relative to the position updates. Positions are then updated based on the new velocity:

$$\mathbf{r}(t + \Delta t) = \mathbf{r}(t) + \mathbf{v}\left(t + \frac{\Delta t}{2}\right)\Delta t. \tag{14}$$

This Leapfrog scheme effectively separates the updates of velocity and position, thus preserving the Hamiltonian structure of the equations over time. Symplectic schemes are often used for the integration of Hamiltonian systems because they prioritize the preservation of physical properties, such as the conservation of phase space volume, over numerical accuracy.

3.3 Smoothing Kernel Parameters

Vincent Boulard

In the realm of SPH, the smoothing kernel size plays a crucial role. The ideal scenario would involve each particle maintaining a constant mass within its kernel, which is achievable, for instance, in a homogeneous fluid.

To achieve this generally, one might consider individualizing the smoothing kernel size for each particle rather than using a global size. To maintain a constant number of particles within each kernel, we require that:

$$\rho_i h_i^d = \text{const}, \tag{15}$$

where h_i is the kernel size for particle i, ρ_i is the density at particle i, and d is the dimension of the space. However, computing the densities requires the h_i values because

$$\rho_i = \sum_j m_j W_{ij}(h_i). \tag{16}$$

Thus, we now face a system of equations that we need to solve to determine the appropriate kernel smoothing sizes and densities. Typically, numerical methods such as Newton-Raphson or the secant method are employed to solve these equations.

Despite the potential benefits of having a variable smoothing kernel size for each particle, for simplicity and due to time constraints, we opted to use a constant smoothing kernel size for all particles.

3.4 Gridding Problem

Vincent Boulard

A significant drawback of using a constant smoothing kernel size is the tendency for particles to align themselves into a grid pattern, as shown in Figure 1. This occurs because particles stick together when they come close, leading to unnatural clustering.

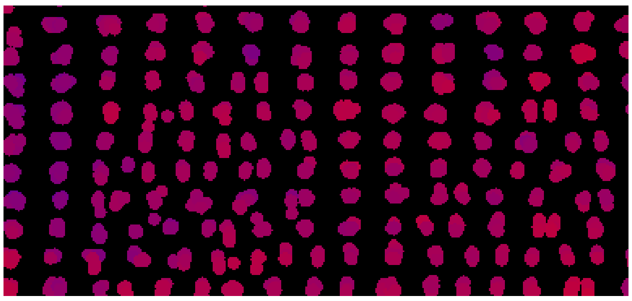


Figure 1: Example of the gridding behavior.

This gridding effect is likely due to the constant smoothing kernel size. When particles cluster too closely, their overlapping kernels result in exaggerated density estimates and artificial clumping. We think that using a variable smoothing kernel size for each particle, as presented in the previous section, could mitigate this issue.

4 Code design

4.1 Nearest-Neighbor Problem

Radovan Dabetić

4.1.1 First Version of the Algorithm in `C++`

This was the first version of the code which was used in our project. The code was designed with simplicity of use in mind, as several people are working on different parts of the program. Another important thing to mention is that since IPPL stores the particles in a struct of vectors and not a vector of structs, it is beneficial to work with indices rather than the particles themselves. From here on, the particles are identified with their indices.
For storing the particles in a cell of the mesh, a singly linked list (`std::forward_list<std::size_t>` using the standard library) is used and a vector of `OpenMP` locks for each cell is there to ensure that we get no race conditions during the parallel insertion of the particles.

For the iteration through the neighbors, a class with an iterator was implemented to deal with accessing the correct cells of the mesh given a particle position. This class is called `SizeListCollection` in our code and a vector of them is set up upon a constructor call of the mesh. In theory we do not need to have a vector of them, but it simplifies some things (at the expense of some memory). It stores the underlying pointers of the cells in a neighborhood and handles iterating through them. Since this code is supposed to work in different dimensions, the neighbors are found by recursively going through the dimensions: the "0-dimensional neighbor" is just the cell itself and the d-dimensional neighbors are the union of the $d-1$ dimensional neighbors with offsets $-1, 1$ ("left" and "right") and 0 (center) of the cell in the mesh in the d-th direction. This recursion was done using templates at compile-time.

Another important thing to mention are the periodic boundary conditions, which change the neighbor cells at the boundaries, because the distance between two points then becomes non-euclidean. Take for example two points very near to the edge of the domain, then the "shortest line" connecting them goes through the boundary to the other particle (topologically we are working with a torus). An example of this using an improved algorithm be found in Figure 2.
At the end, the only thing the user needs to do is iterate through the neighbors using an iterator, which gives an index.

4.1.2 Improvements

There is one significant problem with the above approach: cache. When iterating through the neighbors of a particle, the parameters of each neighbor (e.g. pressure, velocity, density etc.) are stored in an array and are accessed in a (possibly) unordered fashion, which can and probably will lead to many cache misses. Hence an improvement to this algorithm is to re-order the arrays used in the computation such that the parameters of the particles are stored in a contiguous chunk of memory. This could be done e.g. by making a counter of how many particles are in each cell and then "reserving" that space in one contiguous vector (the memory is already allocated, but the vector is partitioned in chunks of these sizes; each cell gets a starting index) and

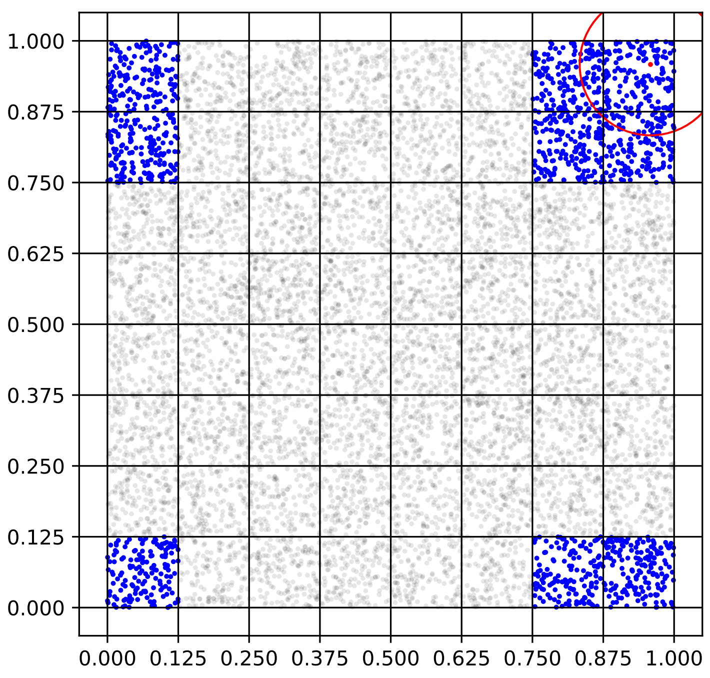


Figure 2: Example nearest-neighbor problem with periodic boundary conditions. The grid corresponds to the mesh used.

placing the values in there, which can easily be done in parallel.

The idea remains the same in principle, but now the start index of a particle in a cell is stored, together with the number of particles therein. The iteration through the neighbors will not be as easy as just one iterator, but it is still quite simple. This approach would mean that each attribute of the particles needs to be copied an re-ordered, so it might be a bit more memory-intensive.
Of course, there is also the question of scaling to bigger systems (using `MPI` and/or GPUs), but this is beyond this project.
A newer version for shared memory systems using CPUs was written with Kokkos, inspired by the work of Timo Schwab[1]. This newer version uses the methods described above to mitigate the aforementioned problem with cache. In fact, Figure 2 was generated using this algorithm. The algorithm still needs to be tested rigorously to ensure correct behavior (and interoperability with different data types), so we are using the old version.
Extending this to variable-size smoothing kernels would not change much in the iteration, however finding the neighbor indices in the mesh might get a bit more complicated as a variable amount of neighbors needs to be added.

4.2 SPHParticle and Manager classes

Andrés Tamargo

After describing the Nearest-Neighbor code design, we can now explain the other two important classes of our code structure.

4.2.1 SPHParticle class

Andrés Tamargo

This is a simple and straightforward class to store all the "local" properties of each particle in the simulation. As expected when working with IPPL, this particle class is a inherited from the ippl::ParticleBase class. For the needs of the SPH simulations, this particle class will have several physical attributes, namely, mass, velocity, density, pressure, entropy and acceleration, which can be taken as ippl particle attribute data type. Moreover, we included ChainingMeshHelper object as an attribute of our particle class which is necessary for the nearest-neighbor search of each particle. As such, we add an updateneighbor member function which

[1]E-Mail: `tischwab@student.ethz.ch`, GitHub url: `https://github.com/tischwab0911/ippl/tree/master`

allows to look for all the nearest neighbor of each particles when called at each integration step.

4.2.2 Manager class

Andrés Tamargo

To run the simulations, we decided to write a manager class that would allow to have all the simulation data (inputs and outputs) in a compact manner within an object. The idea was to store all the "global" aspects of the simulation in this class. By doing that, everything that not only depends on single particles but depend also on the nearest-neighbors of each particle are calculated inside this class.

With this idea in mind, the Manager class has several member functions for each stage in the simulation. Once again, as we are working with IPPL, the Manager class has been inherited from the IPPL BaseManager template class and override some of its member functions. As such, we use the Manager class constructor to initialize some basic parameters like the kernel smoothing length, the time step and adiabatic index. Also, in the constructor we add the particle spatial layout and the extend of the domain that is going to be used in the simulation. Some of these are then used to initialize a SPHParticle object as an attribute of the manager class, which is going to store all the particles information during the simulation.

```
template<unsigned int Dim>
class Manager: public BaseManager {
public:

    //Attributes are defined here...

    //Constructor
    Manager(ParticleSpatialLayout<double,Dim>& L, ippl::Vector<double,Dim
       >& low,
    ippl::Vector<double,Dim>& extent, double dt_, double h_, double
       Adiabatic_index_) :
    dt(dt_), h(h_), particles(L, low, extent, h_), Adiabatic_index(
       Adiabatic_index_), L_(extent), low_(low) {}
```

The `pre_run()` member function is overridden to introduce all the particles initial positions, velocities, specific entropies and masses. Within this member function the process of allocating this data to the particles object happens. The `pre_step()` is overridden and used to calculate the acceleration and entropy time derivative summing along the nearest-neighbors of each particle using Equation 10 and Equation 9. Here is where the ChainingMesh object of the Manager class is used to obtain the nearest neighbor for each particle. The `advance()` function was also taken from the template to write the time integration, such that the acceleration and entropy time derivative calculated previously in `pre_step()` is used. This is how the simulations "advance" one time step every time this function is called.

4.3 Particle Drawing

Vincent Boulard

In our simulation, we sought a visual method to understand the dynamics of the fluids. To achieve this, we implemented a technique to draw each particle's position and color it based on its density. An example is shown below.

The following code snippet, is integrated into the main loop of our program :

```
for (int j = 0; j < positions.size(); j++) {
    sf::CircleShape circle(4.0f);  // Create a circle representing a
       particle

    // Calculate the normalized density value for color mapping
    float t = (manager.particles.density(j) - minRho) / (maxRho - minRho +
        eps);
    sf::Color color(static_cast<sf::Uint8>(255 * t), 0, static_cast<sf::
       Uint8>(255 * (1-t)));
    circle.setFillColor(color);  // Set the color based on density

    // Set the position of the circle based on particle coordinates
    circle.setPosition(positions(j)[0] * width, positions(j)[1] * height);
    window.draw(circle);  // Draw the circle to the window
}
```

Each particle is represented as a circle with its color indicating the particle's density. The density is normalized between 'minRho' and 'maxRho', where colors transition from blue to red as the density increases.

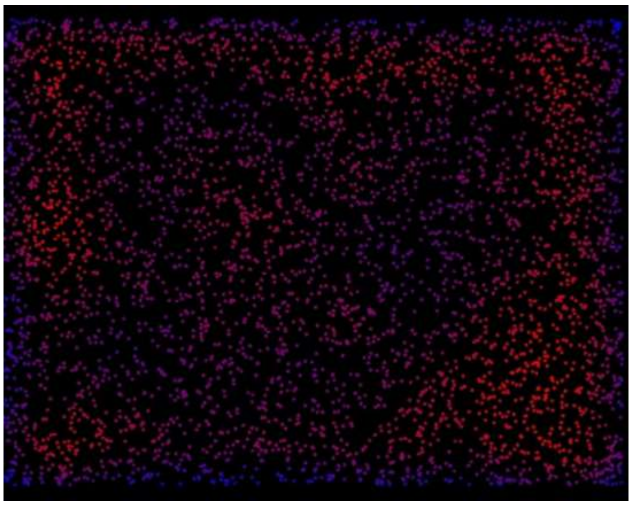


Figure 3: Screenshot of a simulation showing a gas at equilibrium with a Maxwell speed distribution.

5 Results

Andrés Tamargo, Vincent Boulard

After the explanation of the code design, the objective was to use the ChainingMesh class, Manager class and SPHParticle class to try simulating fluid dynamics problems for which the results are well known. We have tried the 1D shock tube problem and the Kelvin–Helmholtz instability, and we expect to implement more test cases in the near future.

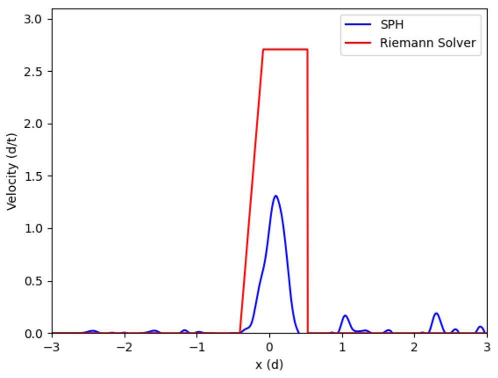
For the following results, we will take the following general dimensions: we will call d, τ as arbitrary length and time dimensions, respectively, while we will normalize the mass unit of all particles involved, i.e. $m_i = 1$.

5.1 1D Shock tube

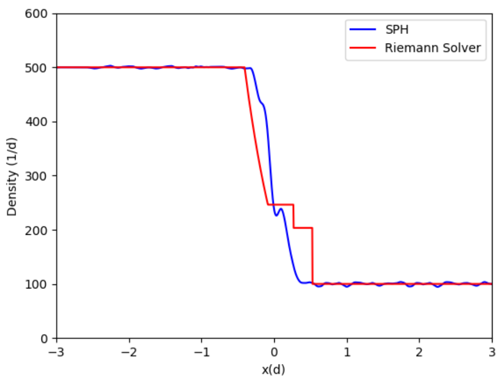
Andrés Tamargo

For this test case, we initialize a total of 6000 particles in a one dimensional domain of size $20d$ that goes from -10 to 10, such that we put 5000 particles in the $[-10, 0]$ region and only 1000 particles in the $[0, 10]$ region all with null initial velocity. This way we ensure a density discontinuity at 0 and see the shock effects that should be reproducible considering the use of artificial viscosity in our simulation. Lastly, we will take our smoothing length as $h = 0.05d$ and time steps of $dt = 10^{-3}\tau$ integrating from $t = 0$ to $t = 0.099\tau$ As a comparison we will use a Riemann solver method to solve 1D shock tube problems sod-shocktube. By doing this we can analyze the performance of our SPH simulations taking this Riemann solver as an exact solution.

In Fig. 4, we can see the results for the density and velocity profiles close to the discontinuity of the shock tube. In the left panel the velocity profile is shown and we see the increase of velocity near the border related to the pressure of the right side being higher than the pressure on the left, resulting in a particle transfer from the left to the right. However, we observe a significant disagreement between the Riemann solver and our simulation results. We blame this disagreement to the fact that we are using a constant smoothing length in our simulations, which results in the unnatural "clustering behaviour" explained in section 3.4. On the right panel of Fig. 4 the density profile is plotted showing the expected jump from the right high density region to the left low density region. We still see disagreements between the Rienmann solver and our SPH simulation



(a) Velocity Profile



(b) Density Profile

Figure 4: Density and velocity profiles close to the density discontinuity in the 1D shock tube at $t = 0.099\tau$

5.2 Kelvin-Helmholtz instability

Vincent Boulard

Here, we initialize $N = 10000$ particles of the same mass uniformly in the $[0, 1] \times [0, 1]$ square. The particle speed in the upper half of the rectangle is $v_1 = 5.0\frac{d}{\tau}$, and in the lower half, it is $v_2 = -5.0\frac{d}{\tau}$. We took $h = 0.01d$. For the boundary conditions, the top and bottom edges of the square are reflective, and the left and right edges are periodic, so topologically, fluids are flowing on a cylinder. We want to observe what we call a Kelvin-Helmholtz instability, which happens when two viscous fluids flow at different speeds at the interface. Figure 5 illustrates this phenomenon well.

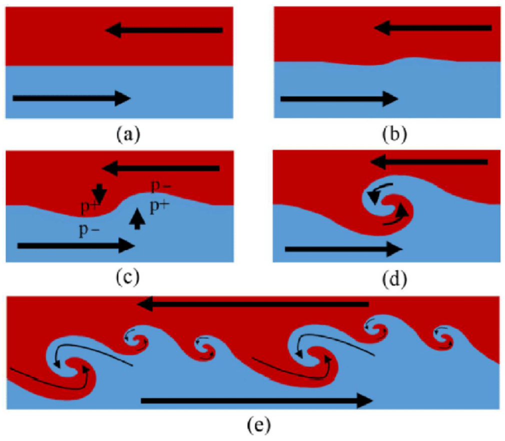
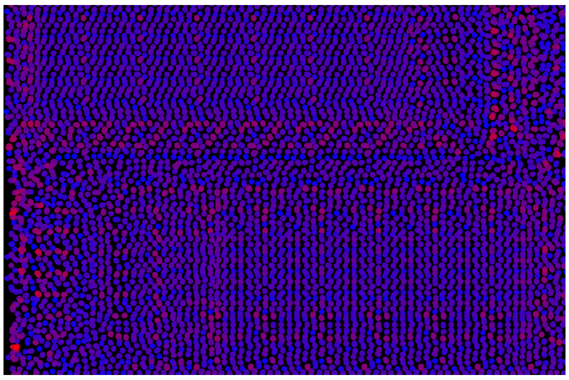


Figure 5: Illustration of the formation of Kelvin-Helmholtz instability in a two-layer fluid system [3].

Now, in Figure 6, we observe screenshots taken at different time during the simulation. Initially, although it is difficult to discern, a velocity shear is visible in the center of the first screenshot, while the interface remains smooth. In the second screenshot, a wavy pattern begins to form at the interface, indicating the onset of the instability. This observation demonstrates that our viscosity implementation is effective.


(a) Screenshot at time $t_1 > 0$

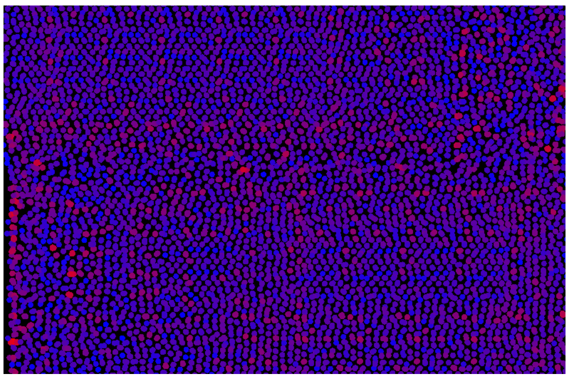

(b) Screenshot at time $t_2 > t_1$

Figure 6: Screenshots from our simulation illustrating the formation of Kelvin-Helmholtz instability.

We note the unusual distribution of density in these images, which we believe is due to the necessity of using a very small smoothing length h to observe the effect. This issue arises because we have not implemented a dynamic smoothing kernel size for each particle, highlighting that the gridding problem remains a significant challenge for us.

6 Conclusion

Vincent Boulard

The most significant challenge of this project was time management, as we only had a few weeks to build everything from scratch. In particular, comprehending IPPL in such a short time was very difficult. Nevertheless, we decided to use it because we were initially very ambitious, aiming to implement additional use cases, particularly N-body problem simulations with gravitational interactions. We believed that IPPL would be beneficial for this.

However, the learning curve and complexity of the library cost us valuable time, leading us to scale down our ambitions. Despite this, we are pleased with our progress, having implemented a significant portion of our planned use cases. We are particularly happy with our C++ implementation and the way we addressed the nearest-neighbor problem through a chaining mesh approach. We are also proud of our ability to simulate the Kelvin-Helmholtz instability and the 1D shock tube problem.

Overall, our experience has been rewarding, and the insights gained from overcoming these challenges have provided us with a solid introduction to SPH.

References

[1] Juan Pablo Cruz Pérez and José Antonio González Cervera. "Efficient Neighborhood Search in SPH". In: *Environmental Science and Engineering*. Springer Berlin Heidelberg, Oct. 2012, pp. 185–199. ISBN: 9783642277238. DOI: 10.1007/978-3-642-27723-8_12.

[2] R.W Hockney and J.W Eastwood. *Computer Simulation Using Particles*. CRC Press, Mar. 2021. ISBN: 9780367806934. DOI: 10.1201/9780367806934. URL: http://dx.doi.org/10.1201/9780367806934.

[3] Jeffery A. Horwood et al. "The formation of Kelvin-Helmholtz instability". In: *ResearchGate* (2019). URL: https://www.researchgate.net/figure/The-formation-of-Kelvin-Helmholtz-instability-Horwood-et-al-2019_fig2_351272332.

[4] J. J. Monaghan and R. A. Gingold. "Shock simulation by the particle method SPH". In: *Journal of computational physics* 52.2 (Nov. 1983), pp. 374–389. DOI: 10.1016/0021-9991(83)90036-0. URL: https://doi.org/10.1016/0021-9991(83)90036-0.

[5] Sriramkrishnan Muralikrishnan et al. "Scaling and performance portability of the particle-in-cell scheme for plasma physics applications through mini-apps targeting exascale architectures". In: *Proceedings of the 2024 SIAM Conference on Parallel Processing for Scientific Computing (PP)*. SIAM. 2024, pp. 26–38.

[6] Volker Springel. "Smoothed Particle Hydrodynamics in Astrophysics". In: *Annual Review of Astronomy and Astrophysics* 48.1 (Aug. 2010), pp. 391–430. ISSN: 1545-4282. DOI: 10.1146/annurev-astro-081309-130914. URL: http://dx.doi.org/10.1146/annurev-astro-081309-130914.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

SMOOTHED PARTICLE HYDRODYNAMICS PROGRAMMING REPORT

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
DABETIC	RADOVAN
BOULARD	VINCENT

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
ZÜRICH, ZÜRICH; 07.05.2024	Radovan Dabetić
	[illegible]
	[illegible]
	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.