An initiation to SPH

Lucas Braune, Thomas Lewiner
Department of Mathematics, PUC–Rio – Rio de Janeiro, Brazil

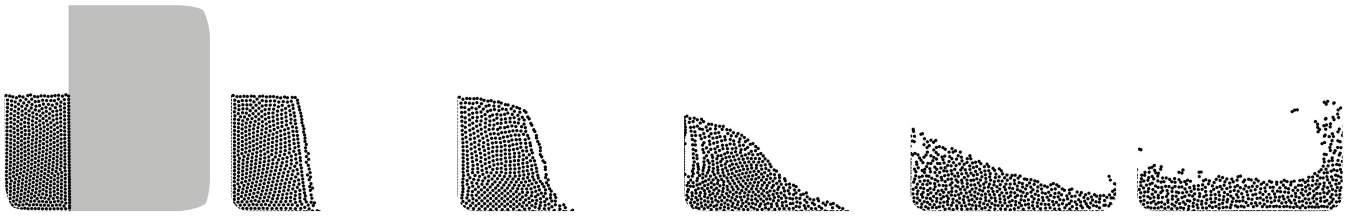

Figure 1. An SPH simulation of a dam break.

***Abstract*—The recent expansion of particle-based methods in physical simulations has introduced a lot of diversity and power aside existing numerical methods. In particular for fluid simulation, Smoothed Particle Hydrodynamics (SPH) have become a very popular technique. Even with the large available SPH literature, such methods are delicate to implement. This work proposes a short and simple SPH initiation that intends to be accessible for undergraduate students.**

***Keywords*-Smoothed Particles Hydrodynamics; SPH;**

I. Introduction

Simulation of natural phenomena is a very active and delicate field of research. On one side, it has innumerous applications, from scientific experiments and industrial validation to virtual environments and computer games. On the other side, it combines difficulties in many domains, such as applied mathematics, numerical analysis, physics and computer science. The typical example is fluid simulation, which received a lot of attention in recent years due to increased computational power and experimental instruments, which allows to generate and process each time more precise data.

In particular, the increasing manipulation of point-based data in computer science has boosted particle-based models for simulation, among which Smoothed Particle Hydrodynamics (SPH). This technique started in the 1970's for astrophysical simulations [1], [2] and has received a lot of attention since then, generating a large literature.

However, during the preparation of this scientific initiation, we found very little reference that would be both simple, short and accessible to undergraduate students. The main objective of this work is to propose a simple but effective introduction to SPH.

II. Description of the SPH method

In this work we will discuss the simulation of ideal fluids without dissipation whose motion can be described by the Euler equation [4, Sec. 6.1]:

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

where $\mathbf{v}$ is the velocity, ρ is the density, P is the pressure and $\mathbf{b}$ is the external force per unit mass applied to the fluid (body force). The canonical example of a body force is the gravity. In general P is a function of ρ and the thermal energy, but in the case where there is no dissipation the pressure can be taken as a function of ρ alone. In this equation d/dt is the material derivative, i.e. following the motion [4, Sec. 6.1].

The idea behind SPH is to replace the fluid by a set of particles whose individual motion is approximated by $d\mathbf{r}/dt = \mathbf{v}$, where $\mathbf{r}$ is the particle position. This implies that each particle's velocity changes by the rule of Eq. (1). One thus needs spatial derivatives to calculate the accelerations of each particle. The description of the SPH approximation for these follows.

A. The SPH approximation of any fluid quantity

Consider the identity

$$A(\mathbf{r}) = \int A(\mathbf{r}')\delta(\mathbf{r}-\mathbf{r}')\mathbf{dr}' \tag{2}$$

where $\delta(\mathbf{r}-\mathbf{r}')\mathbf{dr}'$ is the Dirac delta measure at $\mathbf{r}$. The integral is taken over all space. Let $W(\mathbf{r}, h)$ be a smooth function such that $\int W(\mathbf{r}', h)\mathbf{dr}' = 1$ and

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

These properties leads to the approximation of Eq. (2), for small values of h, by:

$$A(\mathbf{r}) \approx \int A(\mathbf{r}')W(\mathbf{r}-\mathbf{r}', h)\mathbf{dr}'. \tag{4}$$

One example of such a function W is the cubic spline. Let $q = |\mathbf{r}|/h$.

$$W(\mathbf{r},h) = C_h \begin{cases} (2-q)^3 - 4(1-q)^3 & \text{if} \quad 0 \le q < 1; \\ (2-q)^3 & \text{if} \quad 1 \le q < 2; \\ 0 & \text{if} \quad q \ge 2; \end{cases} \quad (5)$$

The constant C_h ensures normalization and is $1/(6h)$, $15/(14\pi h^2)$ and $1/(4\pi h^3)$ in one, two and three dimensions respectively. This is the kernel used in our simulations.

Divide the volume of fluid into a set of small volume elements (particles). The element a has mass m_a, density ρ_a and position $\mathbf{r}_a$. The value of A at the position of particle a is written A_a. Multiplying and dividing the integrand in the right hand side (RHS) of Eq. (4) by $\rho(\mathbf{r}')$, approximating the integral by a summation and noting that $\rho_a \Delta V_a = m_a$ yields

$$A(\mathbf{r}) \approx \sum_b \frac{A_b}{\rho_b} W(\mathbf{r} - \mathbf{r}_b, h) m_b. \quad (6)$$

This summation is the SPH approximation of the quantity A at position $\mathbf{r}$. It takes place over all particles but, in practice, only particles near $\mathbf{r}$ contribute since W looks like Dirac's delta.

Spatial derivatives can be approximated (see [3], [4] for argument) as

$$\frac{\partial A}{\partial x} \approx \sum_b m_b \frac{A_b}{\rho_b} \frac{\partial W}{\partial x}, \quad (7)$$

i.e., one can approximate the derivative by the derivative of the approximation of Eq. (6).

B. Approximation of the density and Euler equations

The density at particle a is in our simulations calculated via Eq. (6) by simple substitution of A by ρ

$$\rho_a = \sum_b m_b W(\mathbf{r}_a - \mathbf{r}_b, h). \quad (8)$$

Such a straightforward substitution of A by P in Eq. (7) does not yield a satisfactory approximation of ∇P for simulations. Considering a pair of particles a and b, the pressure force of a on b would in general be different of the force of b on a should this approximation be made. To write an acceleration equation which conserves total linear and angular momentum, one notes that

$$\frac{\nabla P}{\rho} = \nabla \left(\frac{P}{\rho} \right) + \frac{P}{\rho^2} \nabla \rho. \quad (9)$$

Calculation of the spatial derivatives in this expression via Eq. (7) yields the SPH approximation for each particle's acceleration of Eq. (1)

$$\frac{d\mathbf{v}_a}{dt} = -\sum_b m_b \left(\frac{P_a}{\rho_a^2} + \frac{P_b}{\rho_b^2} \right) \nabla W_{ab}, \quad (10)$$

where $\nabla W_{ab} = \nabla W(\mathbf{r}_a - \mathbf{r}_b, h)$. The body force term $+\mathbf{b}_a$ outside the summation was suppressed for clarity.

III. Numerical integration of the equations of motion

We used the Leap Frog method in our simulations for calculating the trajectories of each particle. This method will be discussed in Subsection III-B. In the first we will justify the addition of a fictitious force term in Eq. (10) in terms of the simpler Euler's method.

A. The need for motion damping

Euler's method for solving a differential equation of the form $x'(t) = f(x(t), t)$ is defined as follows

$$x(t + \Delta t) \quad = \quad x(t) + f(t)\Delta t. \quad (11)$$

Although it is not necessary for the understanding of the current text, we refer the reader interested in the numerical solution of differential equations to [5].

Consider a two dimensional vector field $\mathbf{f} : \mathbf{R}^2 \to \mathbf{R}^2$ such that its integral curves are concentric circles. The solution of the differential equation $\mathbf{x}'(t) = \mathbf{f}(\mathbf{x}(t))$ will always be on the same circle as the initial condition (IC) $\mathbf{x}(0) = \mathbf{x}_0$. Starting from this IC, the Euler solution will follow a straight line tangent to the circle over which the IC lies and fall on some other circle with a larger radius. This will happen every time step and the Euler solution will spiral outward.

A similar phenomenon occurs when solving the differential equations of a particle in a harmonic oscillator potential. In this case, the outward spiraling is replaced by a steady increase in the energy of the system: the particle's speed and movement amplitude increase with time. Such effects are diminished but not eliminated by using more sophisticated time integration methods such as Runge-Kutta and Leap Frog.

One approach for dealing with the problem above is to remove the energy introduced by the integrator with some fictitious force. We adopted this strategy in our simulations: each particle had a term proportional to its velocity deduced from its final acceleration. We refer to this strategy as motion damping.

The final equations of motion to be integrated for each particle are

$$\begin{aligned} \frac{d\mathbf{x}_a}{dt} &= \mathbf{v}_a \\ \frac{d\mathbf{v}_a}{dt} &= -\nu \mathbf{v}_a - \sum_b m_b \left(\frac{P_a}{\rho_a^2} + \frac{P_b}{\rho_b^2} \right) \nabla W_{ab} + \mathbf{b}_a \end{aligned} \quad (12)$$

where ν is the motion damping constant and $\mathbf{b}_a$ is the acceleration of particle a due to the body force (which was gravity in both our simulations).

B. Leap Frog time integration

The Leap Frog method was used in our simulations for solving the differential equations of Eq. (12). It is defined

by

$$\mathbf{v}(t+\Delta t/2) = \mathbf{v}(t-\Delta t/2)+\mathbf{a}(t)\Delta t \quad (13)$$
$$\mathbf{x}(t+\Delta t) = \mathbf{x}(t)+\mathbf{v}(t+\Delta t/2)(t)\Delta t. \quad (14)$$

This method is more accurate than Euler's (it is of second order) and requires only one acceleration calculation per time step (these are expensive in the SPH case). Moreover, the energy introduction errors are small with it (in comparison with other methods of similar computational cost).

In general, the accelerations of the particles at an instant depend on their velocities at that instant, i.e., $\mathbf{a}(t)$ may depend on $\mathbf{v}(t)$. Suppose this is the case and calculate $\mathbf{v}(t)$ as $\mathbf{v}(t) = 0.5(\mathbf{v}(t-\Delta t/2)+\mathbf{v}(t+\Delta t/2))$. The dependency of $\mathbf{a}(t)$ on $\mathbf{v}(t)$ which in turn depends on $\mathbf{v}(t+\Delta t/2)$ makes the relationship of Eq. (13) implicit. One could in principle find good approximations for $\mathbf{v}(t+\Delta t/2)$ by standard methods, but this would involve the solution of linear systems (of dimensions proportional to the number of particles) in every time step. Considering that the velocity at the current time step plays a minor role in the acceleration computation of Eq. (12), one can make the crude approximation

$$\mathbf{v}(t+\Delta t) = \frac{1}{2}(\mathbf{v}(t-\Delta t/2)+\mathbf{v}(t+\Delta t/2)). \quad (15)$$

This has been tested with satisfactory results and is used in our simulations.

Note that with this method advancing from time t to $t+\Delta t$ requires the knowledge of the quantities $\mathbf{x}(t)$, $\mathbf{v}(t)$ and $\mathbf{v}(t-\Delta t/2)$. These quantities must be updated in every time step. Suppose the initial conditions of the problem considered are only given at time $t=0$, i.e., the information $\mathbf{v}(-\Delta t/2)$ is not available. In this case, it is possible to approximate it by using Euler's method to go half a time step back in time.

IV. Implementation of a general particle system

We will discuss the implementation of the above described method in terms of a real programming language, C++. For the readers who do not know it, there is plenty of material about this language on the web. A particularly helpful resource is the tutorial available in http://www.cplusplus.com. In this section we will discuss the classes used for representing sets of particles, the function which actually does the time stepping and how boundaries can be dealt with in SPH.

A. Classes for representing particles

The first thing one needs when programming a particle system is a class for representing single particles. The declaration of such a class could look like the following piece of code.

```
class Particle{
public:
    Particle();
    float x[2], v[2], a[2];
    float pressure, density;
    float v_prev[2];
    unsigned short n;
    Particle* next;
};
```

The constructor just initializes the other members to some set of values. The members position, velocity, pressure, etc. in the next two lines of code store physical properties of this fluid element. The member v_prev[2] stores the velocity at current time minus half a time step. This is used in Leap Frog time integration. The integer n stores the position of this particle in the vector it is stored. The last member is a pointer to some other particle which will be explained later.

The particle system itself can be represented in C++ as a class with a vector of Particles. It should contain two public functions: one for displaying the particles on the screen and one for advancing the system in time. This class' declaration in our code follows.

```
class ParticleSystem{
public:
    ParticleSystem();
    void Step();
    void Draw();
    float Parameters[N_PARAMS];
protected:
    void Boundaries(Particle* a);
    void AddGravity();
    void AddMotionDamping();
    virtual void ComputeAccelerations();
    virtual void InitialPositions();
    vector<Particle> Particles;
};
```

The two members which are not functions are the vectors Parameters and Particles. The number parameters N_PARAMS should be defined earlier in the code. Examples of elements of this vector are simulation parameters such as the number of particles, the motion damping constant and the time step size. The vector of particles is implemented using the vector class which is defined in the Standard Template Library.

The following subsections will explain this class' member functions. Although the discussion will be based on our particular implementation, it is quite general.

B. ParticleSystem() and Draw()

The first thing to be done in the constructor is to fill the Parameters vector with values. After this is done, the

Particles vector should be resized to the correct values. The function InitialPositions() is then called to initialize each particle's properties. Finally, half an euler step is made backwards so as to estimate each particle's $\mathbf{v}(-\Delta t/2)$.

```
for(i=0; i<Particles.size(); i++){
    pcurr = &(Particles[i]);
    pcurr->v_prev[0] = pcurr->v[0] - 0.5*dt*pcurr->a[0];
    pcurr->v_prev[1] = pcurr->v[1] - 0.5*dt*pcurr->a[1];
}
```

The Draw() function should display on some previously set up window the fluid represented by the particles in its current configuration. In our software, we display a black disk in each particle's position. The background is white. This was done using GLUT and OpenGL. The reader who does not know how to do this is once again encouraged to ask the first author via email for the commented code.

C. Step() and ComputeAccelerations()

The stepping function initially calls the ComputeAccelerations() function and then begins a loop through all particles. In this loop, the boundaries are applied to the current particle (Boundaries(pcurr) is called) and then the Leap Frog step of Eq. (13),(14),(15) is performed. The code for this is similar to the above displayed code for euler stepping.

Our code for the ComputeAccelerations() function is displayed below.

```
void ParticleSystem::ComputeAccelerations(){
    for(int i=0; i<Particles.size(); i++){
        Particles[i].a[0] = 0.0;
        Particles[i].a[1] = 0.0;
    }
    AddGravity();
    AddMotionDamping();
}
```

The idea is to set the accelerations of all particles initially to zero and than add the contributions of different forces. The AddGravity() function goes through all particles and deduces some constant term from the vertical component of the acceleration. This term could be set to 9.8, but we rather leave it as a simulation parameter to see how the system responds to different values of external force. We make this function which computes accelerations virtual so that we can change it (add spring forces, or forces arising from pressure) when we derive from the particle system class.

D. Dealing with boundaries

The boundaries we simulated were rigid plane walls (actually, in 2D they were lines). The question is what to do when collision is detected, i.e., when some particle is found to be inside some wall.

A common approach is placing the particle back in the legal space and mirroring its velocity with respect to the wall so that it moves away from it. This approach did not yield satisfactory results in our simulations. We believe it did not work because it involves the arbitrary displacement of particles ("placing the particle back in legal space").

What we did worked very well for particle systems with up to a few thousand particles. The reason for this not working for larger systems is under investigation. We added a force proportional to how deep a particle was inside the wall and perpendicular to the boundary so as to push the particle out of it. The proportionality constant is a simulation parameter. This approach is generally discouraged in literature on physical simulation because it requires small time steps. In the SPH case this is not an issue because integrating the pressure term in the acceleration requires time steps which are just as small.

The equations arising with this collision model are the ones of a harmonic oscillator. As we mentioned before, Leap Frog introduces energy in the system when integrating these equations. For this reason, further motion damping is added to particles inside walls. How much damping is necessary is another simulation parameter.

V. Implementing SPH

Continuing the discussion of our implementation, we now derive from the above described particle system class one specialized for SPH. Its sole purpose is calculating the particles' acceleration quickly, i.e., calculating Eq. (12) quickly.

A. Computational cost

Suppose the particle system has N particles. The naïve calculation of each acceleration involves going through all $N-1$ particles other than the one whose acceleration is being calculated and doing the summation of Eq. (12). The task of computing all accelerations needs a number of operations asymptotically proportional to N^2. This can be reduced to proportional to N if the following method is employed.

Many kernels used in SPH have the property of being nonzero only within finite distance of kh of the origin. For the kernel used in our simulations of Eq. (5), $k = 2$. This implies that, although in principle the summation in the acceleration Eq. (12) takes place over all particles, only each particle's near neighbors contribute to its acceleration. This summation effectively takes place only for near neighbors of each particle. Suppose each particle has about 10 neighbors which contribute. If each particle's summation involve only these 10 neighbors, the number of operations falls from proportional to N^2 to proportional to $10N$, i.e., proportional to N.

B. Finding neighbors

To find neighbors, we subdivide space in square cells (we simulated two dimensions) of side kh. The whole set of cells

is called a grid. One need only look for the neighbors of a given particle in the cell this particle is and the ones next to it.

We now describe our particular implementation. The declaration of the particle system class specialized for SPH is below.

```
class ParticleSystemSPH: public ParticleSystem{
public:
    ParticleSystemSPH();
    float ParametersSPH[N_SPH_PARAMS];
protected:
    void ComputePressures();
    void AddSpatialInteraction();
    void ComputeAccelerations();
    void InitialPositions();

    vector<Particle*> Grid;
    void PrepareGrid();
    void FindCells(float *r, int *cells);
    unsigned short NeighborCount[16000];
    unsigned short Neighbor[16000][40];
    unsigned short NeighborDist[16000][40];
};
```

The grid is represented by a matrix. The element of the i-th row and j-th column of this matrix stores pointers to the particles which are on the cell in the i-th row and j-th column of the grid. This matrix is implemented as a vector. The pointers are stored in a list. Each element of the grid vector is a pointer to some particle in the corresponding grid cell (or NULL should the cell be empty). This particle in turn points to some other particle also on the same grid cell. This goes on until all particles are on the list.

These lists must be prepared before the pressure terms of the acceleration can be computed. The function which does this is PrepareGrid(). It first sets all entries of the grid vector to NULL. It then goes through each particle and does as follows. First, a simple calculation yields the position of the particle on the grid. Then this particle's pointer is set to the value of the grid on this position. Finally, this element of the grid vector is set to point to this particle.

C. Doing neighbor loops

Note that before one can even begin to calculate the acceleration of a single particle, one needs its density. The calculation of this quantity via Eq. (8) involves going through all the other particles. For the same reason as in the acceleration computation, only the near neighbors contribute. So in each time step one must do two neighbor loops. The first one for calculating all the pressures and densities (all this is in ComputePressures()). The second for actually adding the pressure terms to each particle's acceleration (AddSpatialInteraction(); we call it this way because other forces such those arising from viscosity can be added here).

The two neighbor loops don't have to be the same. It is clear that the grid needs only be prepared once per time step. This is to be done in the ComputePressures() function. In the first neighbor loop it is checked which particles of the surrounding grid cells are actually neighbors of a given particle. This needs not be done a second time and we store on an array of integers (Neighbors[16000][40]) the numbers of each particle's neighbors. For example, the number of the zeroth neighbor of particle 10 is stored in Neighbors[10][0]. We needed to keep track of the number of neighbors each particle had and a similar array was created for storing this information. Calculating the distance of each particle and its neighbors is expensive and the creation of a third array for storing this sped up our simulation.

The neighbor looping in the function for computing pressures begins with the preparation of the grid. Then the following is done for each particle. First the positions in the grid vector of the cells around the iterated particle are found (FindCells()). The number of neighbors (stored in the above discussed array) of this particle is set to zero. Then one goes through each particle in each cell around the iterated particle and checks if it is a neighbor. In case it is, the number of the neighbor and its distance to the iterated particle are stored and the number of neighbors is increased. This neighbor is then used in the calculation of the density. Once this is done for every particle in every neighboring grid cell, the pressure of the iterated particle is calculated.

The structure of neighbor looping in the AddSpatialInteraction() similar to the following loop.

```
for(i=0; i<Particles.size(); i++){
    a = &(Particles[i]);
    for(j=0; j<NeighborCount[i]; j++){
        //Add term due to b to a's acceleration
        b = &(Particles[ Neighbor[i][j] ]);
    }
}
```

VI. Simulation of a toy star

We will now consider a finite region of gas held together by a simple force. This system is a model of a star with the gravitational force replaced by a force which is easy to compute. Such a system is called a Toy Star. The force we consider is such that for any two elements of mass the force between them proportional to their separation and along the line of their centers.

A. The gravity force

Suppose that we have a group of N particles so that the force on particle j due to particle k is $\eta m_j m_k (\mathbf{x}_j - \mathbf{x}_k)^2$.

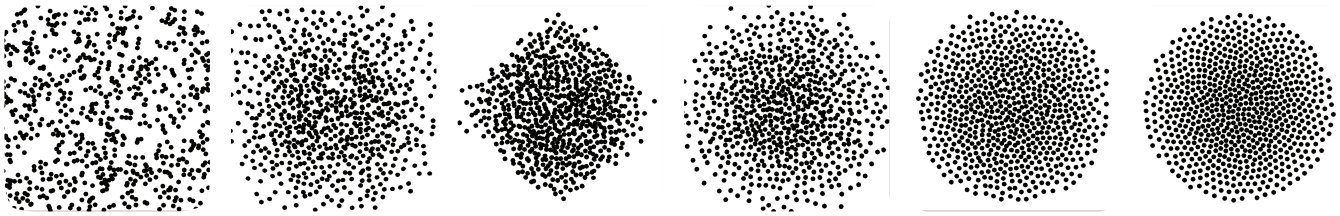

Figure 2. A few steps of the progressive ordering of the particles in the simulation of a toy star

The potential energy is

$$\mathbf{\Phi} = \frac{1}{4}\eta \sum_{j=1}^{N}\sum_{k=1}^{N} m_j m_k |\mathbf{x}_j - \mathbf{x}_k|^2. \tag{16}$$

The equation of motion of the j-th particle is then

$$m_j \frac{d^2\mathbf{x}_j}{dt^2} = -\eta m_j \sum_k m_k(\mathbf{x}_j - \mathbf{x}_k). \tag{17}$$

Choosing the center of mass

$$\frac{\sum_k m_k \mathbf{x}_k}{\sum_k m_k} \tag{18}$$

as the origin, the motion of Eq. (17) becomes

$$\frac{d^2\mathbf{x}_j}{dt^2} = -\eta M \mathbf{x}_j, \tag{19}$$

where M is the total mass of the system.

This says that even though particles are interacting, each one of them moves as if it were on a harmonic oscillator potential.

B. Analytic solution of the Euler equations

In this problem, we will use the pressure equation of state

$$P = k\rho^2. \tag{20}$$

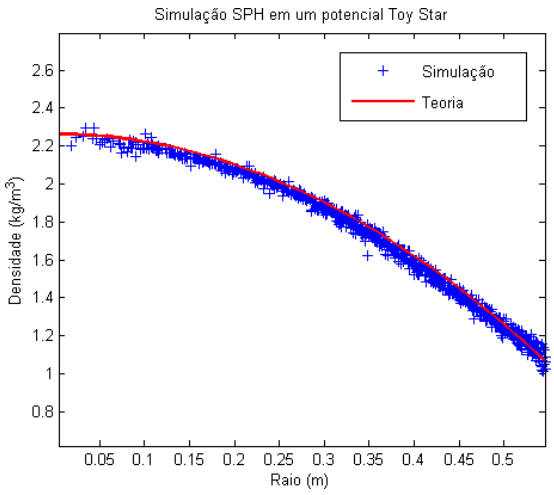

Figure 3. Comparison of the analytical solution of the toy star model with the proposed SPH simulation.

With this equation of state and a body force given by $-\lambda\mathbf{x}$, the Euler equations (1) become

$$\frac{d\mathbf{v}}{dt} = -2k\nabla\rho - \lambda\mathbf{x}. \tag{21}$$

The static model ($\mathbf{v} \equiv 0$) has a density given by

$$\rho(r) = \frac{\nu}{4k}(r_e^2 - r^2) \tag{22}$$

where r_e is determines where the density is to be zero. We compare the analytical solution of the above differential equation (integrated using Matlab solvers) with our SPH simulation in Figure 3 with the parameters described in the next subsection.

C. SPH simulation of the toy star

We simulated a toy star of mass $M = 2.0\ kg$, a radius of $R = 0.75\ m$. The simulation area was the region $[-1, 1] \times [-1, 1]$. Walls were placed at the positions $x = -0.95$, $x = 0.95$, $y = -0.95$ and $y = 0.95$ (Table I). For N particles the smoothing length was set to

$$\frac{0.04}{\sqrt{N/1000}}.$$

All particles were assigned a mass equal to M/N.

The time step was set to $0.004\ s$. With these configurations, up to $4,000$ can be used. Up to $10,000$ can be used if the time step is made half. The resulting simulation is depicted in Figure 2.

The equation of motion was Eq. (12) with $\mathbf{b}_a = -\lambda\mathbf{x}_a$. The particles's positions were initialized to random points within the circle of radius $0.75\ m$ centered in the origin.

Table I
PARAMETERS FOR THE TOY STAR SIMULATION.

Parameter	Value
Number of particles	N=1,000
Smoothing Length	0.04
Time Step	0.004
Acceleration of Gravity	$8Mk/\pi R^4$
Motion Damping	0.5
Pressure Constant	0.1
Particle Mass	0.002
Boundary Repulsive Force	20,000
Boundary Motion Damping	256

The mean square error between of the SPH solution with $3,000$ particles was 0.04. In the outer layers of the star, the computed density differed from the theoretical because it was calculated via Eq. (8), a sum of positive terms, and the theoretical value was negative.

VII. Simulation of a breaking dam

The breaking dam simulation is common challenge for students getting to know SPH. It is difficult to achieve this a simulation because tuning the parameters based on physical principles generally gives spurious results (unless this is done very carefully; the first author did not manage to do this).

We here give the parameters which resulted in the simulation depicted in Figure 1. A working simulation is useful for studying what is each parameter's effect. Indeed, we started from the working simulation parameters for the toy star simulation (i.e. read didn't blow up) and then adapted them for the breaking dam simulation (Table II).

The equation of state used was

$$P = k\left[\left(\frac{\rho}{\rho_0}\right)^7 - 1\right], \tag{23}$$

where ρ_0 is a reference density (read simulation parameter).

The constants relative to the boundary forces we used here were those of the toy star simulation. The boundaries were plane walls at $x = -0.95$, $x = -0.35$, $y = -0.95$ and $y = 0.95$.

Table II
Parameters for the breaking dam simulation.

Parameter	Value
Number of particles	600
Smoothing Length	0.068
Time Step	0.004
Acceleration of Gravity	9.8
Motion Damping	1 (8)
Pressure Constant	0.5
Particle Mass	0.0033
Reference Density	2.861

The particles' initial positions were randomly chosen from the bottom half of the initially legal area. This initial condition is in general not the equilibrium position of the simulated fluid. For this reason, particles start to move very quickly at the beginning of the simulation so as to achieve equilibrium. The very fast change in fluid properties can cause the simulation to explode, so initially the system is damped to a high degree (motion damping constant initially set to 8). When equilibrium is reached, motion damping is set to a normal value ($\nu = 1$). Then is the dam broken: the rightmost wall is moved from $x = -0.35$ to $x = 0.95$.

VIII. Conclusion

We described here a simple implementation of an SPH method for fluid simulations with more detail than it is usual in the literature. We hope to help other students, providing a short mathematical introduction and the main steps of the code. In particular, we provided all the parameters for the experiments, which is usually delicate to obtain from the literature.

Acknowledgements

The authors would like to thank CNPq (PIBIC) and the PUC-Rio for their support during the preparation of this work.

References

[1] R. Gingold and J. Monaghan, "Smoothed particle hydrodynamics - theory and application to non-spherical stars," *Monthly Notices of Royal Astronomical Society*, vol. 181, pp. 375–389, 1977.

[2] L. Lucy, "A numerical approach to the testing of the fission hypothesis," *Astronomical Journal*, vol. 82, pp. 1013–1024, 1977.

[3] A. Paiva, "Uma abordagem lagrangeana para simulação de fluidos viscoplásticos e multifásicos," Ph.D. dissertation, PUC-Rio, 2007.

[4] F. Petronetto, "A equação de Poisson e a decomposição de Helmholtz-Hodge com operadores SPH," Ph.D. dissertation, PUC-Rio, 2008.

[5] G. Golub and J. Ortega, *Scientific computing and differential equations: an introduction to numerical methods*. Academic Press, 1991.