Development of a didactic SPH model

Développement d'un modèle didactique basé sur la méthode numérique de « Smooth particle hydrodynamics »

Travail de fin d'études réalisé en vue de l'obtention du grade de Master Ingénieur Civil des Constructions par Louis Goffin

Jury:
Dr P. Archambeau (supervisor)
Prof. M. Pirotton (2nd supervisor)
Prof. B. Dewals
Prof. L. Duchêne
Dr T. Mouzelard

Année académique 2012-2013

Université de Liège – Faculté des Sciences Appliquées

Objectives

The objectives of this master thesis are written in French.

Le présent travail de fin d'études a pour objectif l'implémentation d'un modèle numérique de type SPH. Une attention particulière sera apportée à l'aspect didactique. Cet aspect pourra être géré par l'utilisation d'un domaine cubique permettant une définition rapide de la géométrie.

Dans un premier temps, une introduction générale devra être réalisée afin de situer la méthode SPH dans le contexte actuel des méthodes numériques. Cette introduction se concentrera principalement sur la différenciation entre méthodes Eulériennes et Lagrangiennes. Une explication à propos des méthodes *meshless* devra être apportée.

Ensuite, une revue de la littérature SPH sera menée. Cette revue se concentrera sur les concepts généraux ainsi que sur l'application au domaine de l'hydraulique. Les variantes choisies seront en accord avec la finalité didactique du projet.

L'implémentation du code sera abordée en troisième lieu. Cette partie se concentrera sur les principaux choix et variantes testés. Une discussion de quelques détails pratiques sera menée.

Un certain nombre de cas tests devront être exécutés afin de prouver le bon fonctionnement du code implémenté. Les résultats obtenus devront être discutés et justifiés. Les différentes options et caractéristiques du programme seront comparées au travers de cas tests spécifiques. Enfin, certaines simulations devront être présentées afin de montrer les possibilités d'un modèle SPH. Le nombre et la nature des différents cas tests sont laissés à l'appréciation de l'étudiant.

Pour finir, une conclusion clôturera le travail. Celle-ci mettra en avant les atouts du programme implémenté tout en n'oubliant pas de mentionner les améliorations possibles. Les possibilités actuelles ainsi que les perspectives seront discutées.

Dr ir. P. Archambeau

Prof. Dr ir. M. Pirotton

Prof. Dr ir. B. Dewals

Prof. Dr ir. L. Duchêne

Dr ir. T. Mouzelard

Abstract

Title (en)	Development of a didactic SPH model
Author	Louis Goffin, Master Ingénieur Civil des Constructions
Academic year	2012-2013

The SPH method (smoothed particle hydrodynamics) is a numerical meshless, particle and Lagrangian method. It is used in a lot of fields of engineering, such as solids mechanics, hydraulics and astrophysics. The medium is represented thanks to a set of particles that have an influence on each other.

First of all, the positioning of the method is discussed. The SPH method is compared to some existing numerical methods. Its advantages and drawbacks are introduced. Its particle, meshless and Lagrangian characteristics are compared to other existing methods.

The basics of the method are explained through the integral representation of a function and the particle approximation. Moreover, some smoothing functions are analysed.

The SPH method being Lagrangian, the Navier-Stokes equations must be written in the Lagrangian formalism. After this, the SPH method can be applied to the equations of continuity and of conservation of momentum. Some practical issues linked to the SPH method such as the neighbours search, the equations of state or the boundary conditions are discussed.

Then, the practical implementation of the method and of the chosen options is explained in details. An object-oriented programming was chosen for its advantages in comparison to a classical sequential implementation.

Finally, the implemented program is tested and commented. Its advantages are highlighted and its drawbacks are discussed and explained. There are a lot of test cases presented. They include some validation tests (dam breaks, particles fall, etc.), some test cases used to compare the options of the program and other tests present the possibilities of the code.

Résumé

Titre (fr)	Développement d'un modèle didactique basé sur la méthode numérique de « Smooth particle hydrodynamics »
Auteur	Louis Goffin, Master Ingénieur Civil des Constructions
Année académique	2012-2013

La méthode SPH (smoothed particle hydrodynamics) est une méthode numérique meshless, particulaire et Lagrangienne. Elle est utilisée dans de nombreux domaines de l'ingénierie, notamment la mécanique du solide, l'hydraulique et l'astrophysique. Le milieu est représenté par un ensemble de particules interagissant les unes avec les autres.

Le positionnement de la méthode est d'abord traité. La méthode SPH est comparée aux méthodes numériques existantes. Ses avantages et inconvénients sont introduits. Ses caractéristiques Lagrangienne, meshless et particulaire sont mises en perspective par rapport aux méthodes plus classiques.

Les fondements de la méthode sont expliqués en passant par les bases mathématiques de la représentation intégrale d'une fonction et l'approximation de particule. De plus, une analyse de plusieurs fonctions de lissage est menée.

La méthode étant Lagrangienne, les équations de Navier-Stokes doivent être établies en Lagrangien. Ensuite, le formalisme SPH est appliqué aux équations de continuité et de conservation de la quantité de mouvement. Des problèmes pratiques liés spécifiquement à la méthode SPH sont également traités. Il s'agit de la recherche des voisins, des équations d'état, des conditions frontières, etc.

Vient alors l'implémentation pratique de la méthode SPH et des options choisies. Tout cela est expliqué en détail. Une programmation orientée-object a été choisie pour les avantages qu'elle présente.

Finalement, le programme implémenté est testé et commenté. Ses avantages sont mis en avant et ses faiblesses discutées et expliquées. Les cas tests présentés sont nombreux. Ils regroupent des cas de validation (rupture de barrage, chute de particules, etc.), des cas destinés à la comparaison de différentes options et des cas tests présentant les possibilités du code.

Acknowledgement

This master thesis was a work of more than four months. I could only achieve it thanks to the help and advices of several people.

First of all, I would like to thank all the people that supervised me these last months. I think especially to Mr P. Archambeau for its availability, the technical means that I could use as well as his advices. I would like also to thank Professor M. Pirotton for the rigour and the helpful questions he brought. I thank also the other members of the jury for their attention.

I thank Quentin Duspeaux and Gianni Costantini for having read this work despite their busy schedule.

My close relatives were also very helpful for their support during all my studies as well as this master thesis. I thank especially Elsia, my parents and my brother Maxim for having supported my not always good mood during this last months.

Finally, I would like to mention the very good atmosphere in the class. It contributed to a very good working atmosphere as well as to good relaxing times. The advices from everyone were very helpful.

Remerciements

La réalisation de ce travail de fin d'études a pu être accomplie grâce au concours de plusieurs personnes.

J'aimerais adresser mes premiers remerciements à tous ceux m'ayant encadré ces derniers mois. Mes mots vont tout particulièrement à Monsieur P. Archambeau pour sa disponibilité, les moyens techniques mis à disposition ainsi que ses conseils de guidance ; mais aussi à Monsieur le Professeur M. Pirotton pour la rigueur insufflée et le questionnement suscité tout au long de ce travail. Je remercie également les autres membres du jury pour le suivi et l'attention portée à ce mémoire.

Je remercie Quentin Duspeaux et Gianni Costantini pour la relecture de ce travail, malgré leur emploi du temps chargé.

Mes proches ont également été d'une grande aide de par leur soutien tout au long de mes études mais aussi de ce travail. Je tiens à remercier tout particulièrement Elsia, mes parents et mon frère Maxim qui ont dû supporter mon humeur parfois difficile de ces derniers mois.

Je ne peux conclure ces remerciements sans mentionner l'excellente ambiance présente au sein de la classe. Cette ambiance a contribué à la fois à une atmosphère de travail saine mais aussi à de bons moments de détente. Les conseils et points de vue de chacun m'ont été d'une grande aide à de nombreuses reprises.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

For ages, man has tried to understand and master the elements on earth. In the Antiquity, the Greeks and Romans built aqueducts to carry water from a river or a lake into a city. The water was used for various purposes: public baths, fountains, sanitary use, etc. These constructions required the knowledge of a few principles.

The Romans were also among the firsts to harness the power of water with the watermills. They were used for example to grind flour or to saw wood. The principle of a watermill is still used today (but improved of course) for the production of hydroelectricity.

Archimedes stated his principle in the third century BC. He also invented the Archimedes screw which is still used today to raise water.

Another element that man always wanted to master is the air. Leonardo da Vinci was part of the firsts to imagine flying machines. Even if his inventions did not work, he tried to understand the principles linked to flying. It's only in the twentieth century (1905) that the Wright brothers achieved the first controlled and motorised airplane flight. Nowadays, planes are able to carry over 800 passengers.

Even if loads of principles have been discovered in the past centuries in the field of fluid mechanics, man still wants to understand more deeply the phenomena linked to air or water for example. Researches are actually led in various fields of fluid dynamics. To help these researches, numerical methods are essential.

1.1 Numerical methods in hydraulics

In hydraulics, a few numerical methods can be used. The *finite element method* is probably the best known method in the field of engineering. This numerical method can also be applied to fluid dynamics as illustrated in [Chung, 1978]. The finite element method was first invented in the 1950s to analyse the structural elements of an aircraft. The goal of this approximate method is to solve differential equations by dividing a continuous media into finite elements. Each element has a certain number of nodes which are used to interpolate the variable of the differential equation.

Another numerical method available to solve differential equations is the *finite volume*

method. This method divides the computational domain into volumes and then calculates the flows through the surfaces separating neighbouring volumes. The finite volume method is used in the WOLF code [Archambeau et al., 2002]. This code has been developed by the applied hydrodynamics service (HECE) at the University of Liège. This tool is used to model free surface flows. The fields of application are wide:

- Dams breaking (e.g. gradual dam failures [Archambeau et al., 2002])
- Simulations of flood (passive flood plains, hydraulic impact of a measure, etc.)
- Wave propagation
- Erosion process
- Sedimentation problems
- etc.

I can also cite two traditional methods: *finite differences* and *weighted residuals*.

Additionnaly, methods of another kind are becoming more and more popular among scientists. These methods are called *meshfree methods*. Unlike the previously cited methods, the meshfree methods do not require a grid to solve a differential equation. The *smoothed particle hydrodynamics* (SPH) method is part of the meshfree methods.

1.2 Grid-based and meshfree methods

The problem domains of traditional methods are divided into cells. These cells are used to evaluate the derivatives of the differential equations. The grid that is created may lead to a certain number of problems such as geometric problems when the domain is not regular.

Such grids are not required when using meshless methods[1]. This is very useful for problems that introduce geometric difficulties (e.g. complex boundary pattern) or discontinuities in the fluid (e.g. the whole domain is meshed but the fluid occupies only a small volume). Meshless methods are freed from problems induced by a grid.

1.2.1 Grid-based methods

A grid-based method is a method that requires a mesh to run. Methods like the finite element method (FEM) or the finite volume method are grid-based methods. The differences are well explained in [Liu and Liu, 2003].

We will distinguish two kinds of grids: Eulerian grids and Lagrangian grids.

[1] Meshfree methods are also known as meshless methods.

Eulerian grid

An Eulerian grid is a grid that is fixed in the space. The grid remains at the same position during all the simulation (if no remeshing) independently from the the material position. When the simulated object moves, it moves *through* the mesh. This leads to a flow of material through every cell's surfaces. The method is represented graphically in the figure 1.1.

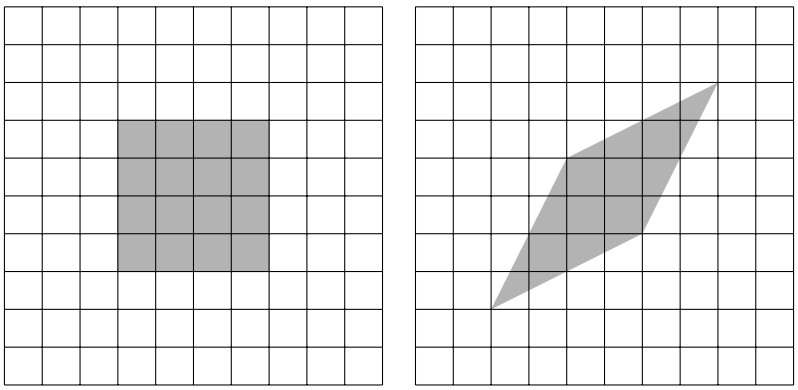


Figure 1.1: Eulerian grid

The Eulerian grid is used in the finite volume method for example. The main advantage of this method is that the grid is not disturbed by the mouvement and the deformation of the object. This is very useful in fluid dynamics. In fact, the fluid can experience large deformations (e.g. recirculation in a turbulent flow, vortex, etc.) without having a grid deformation.

To summarise, what matters in an Eulerian grid is the flow of material between cells. What happens in particular for each particle of fluid is ignored (unless a particle tracking method is implemented). The Eulerian grid only focusses on the general behaviour of the whole object. This is particularly well-suited for most applications in hydraulics: steady/unsteady flows in a river, characterisation of flood plains, wave propagation, etc.

Lagrangian grid

A Lagrangian grid is a grid fixed on the material. This means that when the object studied is deformed, the grid also deforms itself. This is represented in figure 1.2. This technique is mainly used in the finite element method.

When the material is highly deformed, the mesh is also highly distorted. When the cells are highly distorted, the accuracy is lessened. A solution is to remesh when the grid becomes too distorted. This leads to a longer calculation time.

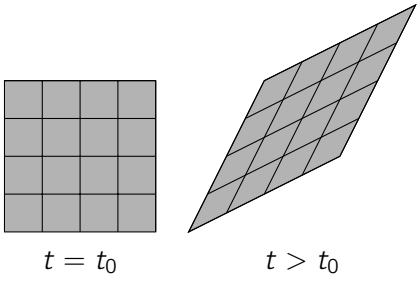


Figure 1.2: Lagrangian grid

1.2.2 Meshfree methods

The traditional methods such as the finite element method (FEM) and the finite difference method (FDM) use grids, volumes or cells according to the method used. More generally, these grids, volumes or cells can be named meshes. The meshes are necessary for FEM and FDM to solve the differential equations that rule the physical problem. However, the use of meshes leads to some issues such as:

- The creation of a mesh before the resolution of the problem takes a lot of time for the engineer/analyst. The CPU time is not a restriction anymore. Thus the manpower time is becoming a concern.
- The elements' distortion can lead to decreased accuracy.
- Remeshing is possible in 2-D but not workable in 3-D.
- Discontinuities are not well depicted in FEM.
- As written in [Liu, 2003], FDM works very well for a large number of problems, especially for fluid dynamics. However, FDM relies on regularly distributed nodes.

In order to avoid the limitations due to meshes, scientists and engineers developed what is called meshfree (or meshless) methods. Ideally, these methods do not use any mesh. The nodes are scattered over the domain in order to represent the material or the fluid. The boundaries are also represented by a set of nodes.

[Liu, 2003] gives the *minimum* requirement for a meshfree method:

> "A predefined mesh is not necessary, at least in field variable interpolation."

and the *ideal* requirement for a meshfree method:

> "No mesh is necessary at all throughout the process of solving the problem of given arbitrary geometry governed by partial differential system equations subject to all kinds of boundary conditions."

The methods developed so far are not ideal.

A number of meshfree methods are given in the table 1.1. This list has been established by [Liu, 2003]. It is possible that today new methods have been developed. It is interesting to notice that the majority of these methods have been developed in the 1990s. They are quite young compared to the finite element method which was first developed in the 1950s. Many improvements still have to be done for these new methods.

Method	**Year of 1st publication**
Element free Galerkin method	1994
Meshless local Petrov-Galerkin method	1998
Point interpolation method	1999
Point assembly method	1999
Finite point method	1996
Smooth particle hydrodynamics	1977
Diffuse element method	1992

Table 1.1: Some meshfree methods

Meshfree particle methods

A meshfree particle method (MPM) is a meshfree method that uses particles in order to carry some field variables. For example, a particle can carry information such as:

- the speed,
- the density,
- the pressure,
- the stresses,
- etc.

Thus, a system is made of a discrete number of particles used to record the state of this system. The particles can either be a discrete physical object or part of a continuum.

The MPMs can be classified into three categories according to the size of the particles [Liu and Liu, 2003]:

1. Atomistic MPMs such as:
    - the molecular dynamics method
2. Mesoscopic MPMs such as:
    - dissipative particle dynamics
    - lattice gas cellular automata
3. Macroscopic MPMs such as:

- Smoothed particle hydrodynamics (SPH)
- Particle-in-cell
- Fluid-in-cell
- Moving particle semi-implicit

Even if the methods cited above were developed for a given purpose, some are used today in different fields of study. For example, the SPH method was first developed for astrophysical problems (macroscopic scale) by [Lucy, 1977; Gingold and Monaghan, 1977]. Nowadays, it is used in hydraulics, solid mechanics but also for atomistic scale simulations.

A MPM is more likely to be a Lagrangian method. In fact, the particles carry some information and we follow those particles at every time step. Thus, we are interested in the value of a variable for a given particle (whatever position the particle occupies) and not the value of a variable at a fixed position in space. However, a few examples show that the particles can be fixed in an Eulerian frame. But this is quite uncommon.

1.3 Positioning the smoothed particle hydrodynamics (SPH) method

So far, we have seen a general background of the existing methods. I will now focus on the SPH method and define its type, its history but also its advantages.

1.3.1 Type of method

The SPH method is said to be a *meshfree*, *Lagrangian*, *particle* method. In fact, the state of a system is represented by a set of particles. Each particle carries some information related to the problem that has to be solved. For example, in hydraulics, we have mainly the position, the speed and the density as information. In solid mechanics, the stresses and strains will also be recorder for every particle.

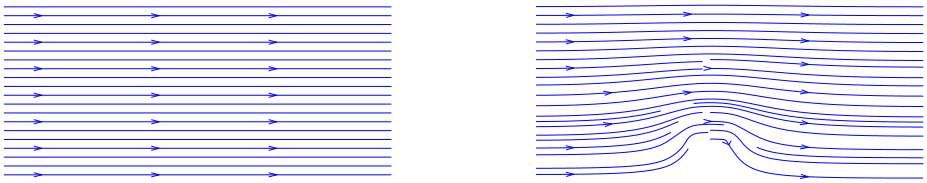
The method is

- *meshfree* because no mesh is required for field variable interpolation (minimal requirement). However, the SPH method is not an ideal meshfree method. Indeed, a mesh may be required twice: at the initialisation in order to arrange the particles and when searching for neighbours.
- *Lagrangian* because we are interested in the value of some variables for a given particle. The particles move in the domain. Thus, we follow the evolution of the variables along the path of a particle and not at a fixed position in space.
- *particle* because the matter/fluid is represented by a set of particles.

It is interesting to understand the meaning of *smoothed particle hydrodynamics*. It is composed of three words that I will define:

- a *particle* is the main entity of the system. The system is based on a set of particles which interact with each other according to the distance between them.
- *smoothed* refers to the weighting function[2] used to take into account the influence of the neighbouring particles on a given one.
- *hydrodynamics* is most probably the best domain in which the SPH method can be applied.

In other words, the SPH method uses a set of particles which interact which each other with a different intensity according to the distance between them. This can be easily visualised in an hydrodynamic problem: the movement of a particle of fluid will be more easily influenced by a near particle than by a far one. For example, let us imagine a laminar steady flow in channel. In that situation, all the fluid particles move in the same direction. Now, imagine that an obstacle appears in the channel. The particles directly in contact with this obstacle will change their direction. The particles in the neighbourhood of those ones will also change their direction and so on. But the farer particles will not be influenced as much as the closer ones to the obstacle. This is shown in the figure 1.3.



(a) (b)

Figure 1.3: Influence of an obstacle on an irrotational flow. (a) free channel, (b) channel with an obstacle.

1.3.2 History of the method

The SPH method was first invented by [Lucy, 1977; Gingold and Monaghan, 1977] to solve astrophysical problems in a 3-D open space. Since the movement of the particles in astrophysics is quite close to the one of a gas or a fluid, the classical equations of Newtonian hydrodynamics were used. In [Gingold and Monaghan, 1977], the number of particles used was of approximately 80 particles. Nowadays, the number of particles used is of approximately 100000 (see for examples [Gomez-Gesteira et al., 2012a]).

[2] I will later use different terms to refer to the weighting function: smoothing function and kernel are two examples.

Today, the SPH method is still used for astrophysical problems. But it has also extended to other domains such as fluid dynamics [Dalrymple and Knio, 2001; Gomez-Gesteira et al., 2012a,b; Monaghan, 1994; Monaghan and Kos, 1999; Vacondio et al., 2012], solid mechanics (fractures, impacts, ... [Johnson et al., 1996; Randles and Libersky, 1996]) or heat conduction [Chen et al., 1999].

Generally speaking, the SPH method can be used to solve differential equations. [Canor and Denoël, 2013] used the SPH formalism to solve the transient Fokker-Planck-Kolmogorov equation (random excitation of mechanical systems). The SPH method is very adaptive and can be used in many study areas.

During the last decades, some drawbacks inherent to the SPH method have been identified and some improvements were proposed. Those improvements concern the boundary conditions, the unity of the kernel (see chapter 3), etc.

Nowadays, the SPH method is still under development. For example, the boundary conditions are improved in order to allow an inflow or outflow [Vacondio et al., 2012].

1.3.3 Advantages of the method

I will now summarise the advantages of the SPH method over traditional methods like FDM.

- The SPH method is a *meshfree* method. This means that no mesh is necessary for the computation. Actually, a mesh may be necessary for the particles initialisation and for the searching of neighbours. However, those meshes don't deform with time. The distortion of the mesh is a big issue for grid-based methods such as FEM. Indeed, when a mesh is too distorted, the results are less accurate and a remeshing is necessary. This operation can take a lot of CPU time and may not be well executed. For FDM, the cells must be regular in order to evaluate correctly the the spatial derivative. If we have a complex geometry, then it is not easy (or even possible) to have a regular mesh. A meshless method does not have the problems associated with the use of a permanent mesh.

- The SPH method is a *Lagrangian particle* method. The fluid is represented by a set of particles. Those particles carry a number of information relative to their state. Each particle influence each other according to the distance between them. The advantage of a particle method over a traditional method is that a particle can easily detach from the fluid. It is impossible to represent this behaviour with a grid-based method. Generally speaking, the discontinuities are better represented with a MPM.

- The *Lagrangian* feature of the SPH method also allows to follow the path of the particles. This may be interesting for problems which involve the mixing of different fluids.

It is important to note that the traditional methods are still of great quality for many problems. However, in some specific cases, these methods have some limits and the SPH method can go beyond those limitations.

1.4 Goals of this master thesis

For this master thesis, I will implement the SPH method applied to hydrodynamics. The code that will be delivered will have a didactic purpose. In order to achieve this, the computation code will be implemented in `Fortran` for its high performance. Moreover, the system of input and output files will be thought in order to allow an easy and flexible use of the program for the didactic purpose.

The master thesis will be organised this way:

1. I will begin with the fundamentals of the SPH method in chapter 2. I will explain the mathematical basis as well as some general concepts like the smoothing function or the support and influence domain.

2. Then, I will go into further details in chapter 3. In this chapter, I will apply to hydraulics what was seen in chapter 2. I will also introduce some problems (and solutions) linked to the SPH method.

3. In chapter 4, I will explain how the method has been implemented.

4. Finally, in chapter 5, the code will be tested an validated thanks to some benchmark cases.

Chapter 2

Basics of the SPH method

In this chapter, I will focus on the fundamentals of the smoothed particle hydrodynamics method. The basic idea is to represent the matter (fluid or solid) with a set of particles. The particles are positioned in space without the help of any mesh. Only the coordinates $\mathbf{x} = (x, y, z)$ describe their position.

Each particle interacts with each other. The properties of a neighbouring particle will influence the properties of a given one. Of course, only close particles will influence themselves.

2.1 Integral representation of a function and its first derivative

First of all, I will begin with the elementary integral representation of a function. As explained in [Liu and Liu, 2003], a function f can be written as:

$$f(\mathbf{x}) = \int_{\Omega} f(\mathbf{x}')\delta(\mathbf{x} - \mathbf{x}') \, d\mathbf{x}' \tag{2.1}$$

with Ω a domain that contains $\mathbf{x}$ and $\delta(\mathbf{x} - \mathbf{x}')$ a Dirac function that can be defined mathematically [Kreyszig, 2006] by

$$\delta(\mathbf{x} - \mathbf{x}') = \begin{cases} \infty & \text{if } \mathbf{x} = \mathbf{x}' \\ 0 & \text{if } \mathbf{x} \neq \mathbf{x}' \end{cases} \tag{2.2}$$

and

$$\int_0^{\infty} \delta(\mathbf{x} - \mathbf{x}') \, d\mathbf{x}' = 1 \tag{2.3}$$

In one dimension, the equation (2.1) can be represented as in figure 2.1. As stated in [Liu and Liu, 2003], if $f(\mathbf{x})$ is defined and continuous in Ω and given that the Dirac function is used, the integral representation (2.1) is exact.

The next step is to replace the Dirac function by the smoothing function $W(\mathbf{x} - \mathbf{x}', h)$. The equation (2.1) becomes

$$\langle f(\mathbf{x}) \rangle = \int_{\Omega} f(\mathbf{x}')W(\mathbf{x} - \mathbf{x}', h) \, d\mathbf{x}' \tag{2.4}$$

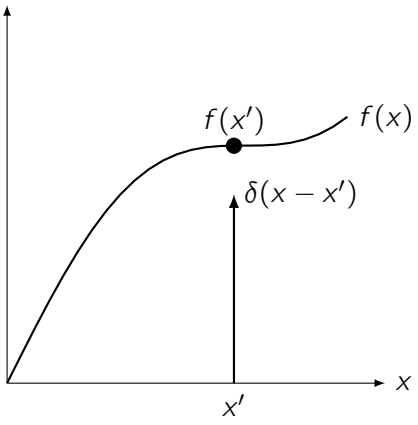


Figure 2.1: Graphical representation of equation (2.1)

This equation is not exact anymore since the Dirac function has been replaced. This is the reason why $f(\mathbf{x})$ is placed between $\langle\rangle$.

A graphical representation of equation (2.4) is given in figure 2.2. More details about the smoothing function will be given later.

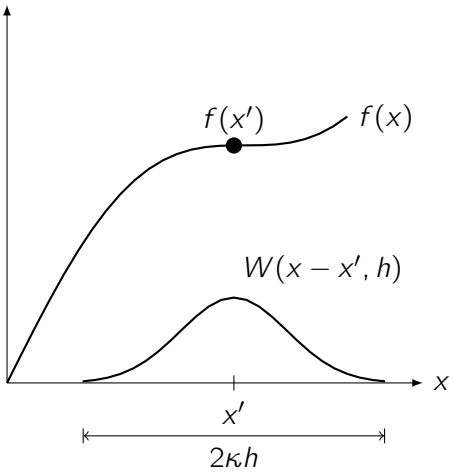


Figure 2.2: Graphical representation of equation (2.4)

In equation (2.4), h is called the smoothing length. It is used to determine the spread of the smoothing function over the domain. The initial smoothing length h_0 can be determined thanks to the initial spacing between particles (s_0). h_0 can range from 1 to 2 times s_0 depending on the author and the problem. κh determines the support domain of the smoothing function. It is represented in figure 2.2.

The first derivative of a function will appear in the particle approximation. [Liu and Liu, 2003] describes the mathematical development to obtain the first derivative of a function.

The first derivative[1] may be written as below thanks to equation (2.4).

$$\langle \nabla \cdot f(\mathbf{x}) \rangle = \int_{\Omega} \nabla \cdot \left[f(\mathbf{x}') \right] W(\mathbf{x} - \mathbf{x}', h) \, d\mathbf{x}' \tag{2.5}$$

In this equation, the part in the integral can be written

$$\nabla \cdot \left[f(\mathbf{x}') \right] W(\mathbf{x} - \mathbf{x}', h) = \nabla \cdot \left[f(\mathbf{x}') W(\mathbf{x} - \mathbf{x}', h) \right] - f(\mathbf{x}') \cdot \nabla W(\mathbf{x} - \mathbf{x}', h) \tag{2.6}$$

Combining (2.5) and (2.6), the following expression is obtained:

$$\langle \nabla \cdot f(\mathbf{x}) \rangle = \int_{\Omega} \nabla \cdot \left[f(\mathbf{x}') W(\mathbf{x} - \mathbf{x}', h) \right] \, d\mathbf{x}' - \int_{\Omega} f(\mathbf{x}') \cdot \nabla W(\mathbf{x} - \mathbf{x}', h) \, d\mathbf{x}' \tag{2.7}$$

The divergence theorem (Gauss's theorem) states that (in [Kreyszig, 2006]):

$$\int_{\Omega} \nabla \cdot f(\mathbf{x}) \, d\mathbf{x} = \int_{S} f(\mathbf{x}) \cdot \mathbf{n} \, dS \tag{2.8}$$

with

- Ω a closed region in space,
- $f(\mathbf{x})$ a vector function that is continuous, has continuous first derivative and is defined in a domain containing Ω,
- S the surface that defines the boundary of Ω and,
- $\mathbf{n}$ the unit vector normal to S.

Using (2.8), equation (2.7) can be rewritten:

$$\langle \nabla \cdot f(\mathbf{x}) \rangle = \int_{S} \left[f(\mathbf{x}') W(\mathbf{x} - \mathbf{x}', h) \right] \cdot \mathbf{n} \, dS - \int_{\Omega} f(\mathbf{x}') \cdot \nabla W(\mathbf{x} - \mathbf{x}', h) \, d\mathbf{x}' \tag{2.9}$$

Two situations have now to be discussed.

1. The support domain $2\kappa h$ is contained in the problem domain. This situation is represented in figure 2.3 (a). In this case, the first term of the right-hand side (RHS) of equation (2.9) is equal to 0. This is due to the fact that, on the problem's domain boundary S, $f(\mathbf{x}')W(\mathbf{x} - \mathbf{x}', h) = 0$ since the smoothing function is supposed to be defined on a compact.
2. The support domain $2\kappa h$ overlaps the problem domain boundary. This is represented in figure 2.3 (b). This leads to a non-zero smoothing function on the problem boundary S. The first term of the RHS of equation (2.9) is not zero anymore.

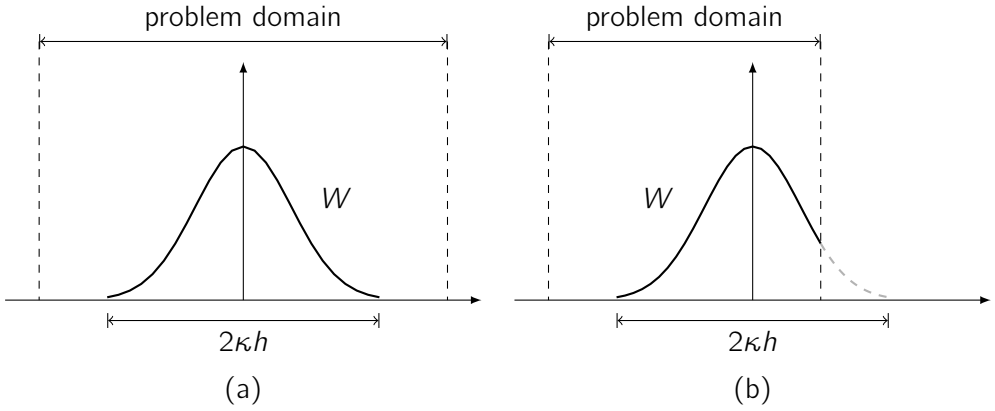


Figure 2.3: Support domain of the smoothing function contained (a) or not (b) in the problem domain

In order to simplify the problem, only the first situation will be considered for the next developments. Nevertheless, it is important to keep in mind that, when near to the boundary, errors will occur. It will be discussed later how to deal with a truncated smoothing function. When considering the first situation (figure 2.3 (a)), equation (2.9) can be simplified into

$$\langle \nabla \cdot f(\mathbf{x}) \rangle = -\int_{\Omega} f(\mathbf{x}') \cdot \nabla W(\mathbf{x} - \mathbf{x}', h)\ d\mathbf{x}' \tag{2.10}$$

Equation (2.10) shows that the approximation of the first derivative of a function is expressed with the derivative of the smoothing function instead of the first derivative of the function itself. This avoids the use of spatial finite differences.

2.2 Particle approximation

Until now, only continuous functions were used. The SPH method uses a set of discrete particles distributed in space. It is called the particle approximation. The development of the particle approximation is explained in [Liu and Liu, 2003]. Each particle carries its mass and a certain number of properties. The particle approximation is represented in figure 2.4 in 2-D. Each particle has a given mass which stays constant. It also occupies a given volume and has a given density which both can vary in time. This can be expressed as:

$$m_b = \rho_b \Delta V_b \tag{2.11}$$

In the following developments, I will use the subscripts a and b to refer respectively to a particle and its neighbours.

[1]Actually, it is the divergence that will be used. The divergence is noted $\mathrm{div} f = \nabla \cdot f$. The divergence is operated with respect to the first derivative. This is a more general way to write a derivative. In fact, functions like the velocity are vector functions but functions like the density are scalar functions.

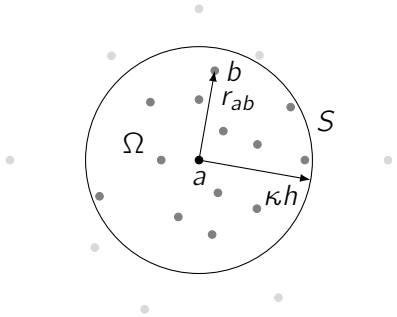


Figure 2.4: Particle approximation in 2D

The continuous integral (2.4) has to be transformed into a discrete sum in order to take account for the particle approximation. Equation (2.4) can be rewritten thanks to equation (2.11).

$$\begin{aligned}\langle f(\mathbf{x})\rangle &= \int_{\Omega} f(\mathbf{x}')W(\mathbf{x}-\mathbf{x}', h)\ d\mathbf{x}' \\ &\approx \sum_{b=1}^{N} f(\mathbf{x}_b)W(\mathbf{x}-\mathbf{x}_b, h)\Delta V_b \\ &= \sum_{b=1}^{N} \frac{m_b}{\rho_b} f(\mathbf{x}_b)W(\mathbf{x}-\mathbf{x}_b, h)\end{aligned} \tag{2.12}$$

For a given particle a, equation (2.12) can be particularised:

$$\begin{aligned}\langle f(\mathbf{x}_a)\rangle &= \sum_{b=1}^{N} \frac{m_b}{\rho_b} f(\mathbf{x}_b)W(\mathbf{x}_a-\mathbf{x}_b, h) \\ &= \sum_{b=1}^{N} \frac{m_b}{\rho_b} f(\mathbf{x}_b)W_{ab}\end{aligned} \tag{2.13}$$

with the shorten notation $W_{ab} = W(\mathbf{x}_a - \mathbf{x}_b, h)$. In the next pages, I will also use $r_{ab} = |\mathbf{x}_a - \mathbf{x}_b|$.

Equation (2.13) shows that the value of a function at the position of a particle (a) is given by a weighted sum over the values of the same function at the position of the neighbouring particles (b). The sum is weighted by the smoothing function $W(\mathbf{x}_a - \mathbf{x}_b, h)$. The mass and the density of each neighbouring particle also intervene in equation (2.13).

The same reasoning can be followed for the first derivative of a function. Using equation (2.11), equation (2.10) can be approximated as

$$\langle \nabla \cdot f(\mathbf{x}_a)\rangle = -\sum_{b=1}^{N} \frac{m_b}{\rho_b} f(\mathbf{x}_b)\ \nabla W_{ab} \tag{2.14}$$

If the gradient of the smoothing function is taken with respect to the particle a, the negative sign in equation (2.14) can be removed as the kernel first derivative is an odd function:

$$\langle \nabla \cdot f(\mathbf{x}_a) \rangle = \sum_{b=1}^{N} \frac{m_b}{\rho_b} f(\mathbf{x}_b) \ \nabla_a W_{ab} \tag{2.15}$$

where

$$\nabla_a W_{ab} = \frac{\mathbf{x}_a - \mathbf{x}_b}{r_{ab}} \left. \frac{dW}{dr} \right|_{r=r_{ab}} \tag{2.16}$$

To summarise, the particle approximation is found by approximating the continuous integral representation of a function by a sum. The sum is weighted by a smoothing function. The weighting is a function of the distance between particles r_{ab}. The closer the neighbour particle, the more weight it has.

2.3 Support and influence domain

Until now, I only mentioned the term *support domain*. At this point of the dissertation, it is important to give a definition of it. In the general context of meshfree methods, [Liu, 2003] gives the following definition for the support domain:

> "By definition, the support domain for a field point at $\mathbf{x} = (x, y, z)$ is the domain where the information for all the points inside this domain is used to determine the information at the point at $\mathbf{x}$."

The influence domain can be understood as the opposite of the support domain. In fact, the influence domain can be defined as the domain where a particle influences others.

When the smoothing length κh is equal for every particle, the action that a particle i exerts on a particle j is equal to the action that the particle j exerts on the particle i. This agrees with Newton's third law. This situation is depicted in figure 2.5 (a). It can be seen in figure 2.5 (a) that the influence domain of the particle a is only over b.

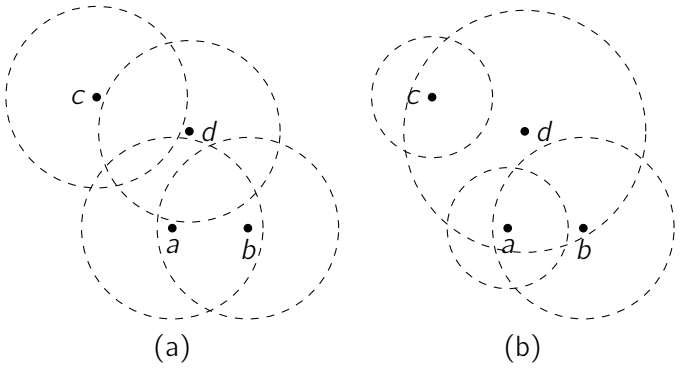


Figure 2.5: Support domains of particles having the same (a) or different (b) κh

Another situation is when the smoothing length differs between the particles. This kind of situation can occur when the smoothing length is updated differently between particles. In that case, a particle i can have an influence on a particle j while the particle j has no influence on the particle i. This violates Newton's third law which is not acceptable. The situation explained is depicted in figure 2.5 (b). Now, the influence domain of the particle a is over b and d. Inversely, the influence domains of b and d are not over a.

In order to keep a physical consistency, the problem of variable smoothing length between particles have to be studied carefully. However, this master thesis will deal only with constant smoothing length between the different particles.

2.4 Smoothing function

As explained earlier, for a given particle, only the nearest particles will influence its properties. In order to satisfy this, the SPH method needs what is called a *smoothing function*. Other denominations can be found in the literature. Here are a few ones found in [Liu and Liu, 2003]: *smoothing kernel function*, *smoothing kernel* or simply *kernel*.

The smoothing function will give a weight to each neighbouring particle.

2.4.1 Properties of a smoothing function

A number of conditions must be fulfilled by the smoothing function. [Liu and Liu, 2003] states the main properties of a smoothing function:

1. The kernel must be be *normalised*:

$$\int_{\Omega} W(\mathbf{x}-\mathbf{x}', h)\ d\mathbf{x}' = 1 \tag{2.17}$$

   This property is due to the Dirac function definition (2.3).

2. The smoothing function has to be defined on a *compact domain*. Nevertheless, this condition is not mandatory. In fact, [Gingold and Monaghan, 1977] used a Gaussian that was not compactly supported. This kernel will be discussed later.

   The property of compact support can be mathematically expressed as

$$W(\mathbf{x}-\mathbf{x}', h) = 0 \text{ if } \left|\mathbf{x}-\mathbf{x}'\right| > \kappa h \tag{2.18}$$

   In this expression, κ represents a scaling factor which determines the spread of the support domain. This will be discussed more deeply later.

   This condition is very useful for the implementation of the program as it limits the area where particles have to searched.

3. The smoothing function must be *positive* on the support domain:

$$W(\mathbf{x}-\mathbf{x}', h) \geq 0 \text{ for } \left|\mathbf{x}-\mathbf{x}'\right| \leq \kappa h \tag{2.19}$$

This property is a physical requirement rather than a mathematical one. If the smoothing function becomes negative at some points of the support domain, then it can lead to unphysical phenomena, especially in hydrodynamic problems.

4. The value of the smoothing function should be *monotonically decreasing* with the distance $|\mathbf{x} - \mathbf{x}'|$. This ensures that the nearest particles have a greater influence than the farthest ones.

5. The kernel must get close to the *Dirac function* when h decreases:

$$\lim_{h \to 0} W(\mathbf{x} - \mathbf{x}', h) = \delta(\mathbf{x} - \mathbf{x}') \tag{2.20}$$

This condition ensures that mathematical convergence occurs when h approaches 0.

6. The smoothing function should be *symmetrical*. This leads to the fact that particles at the same distance from a point have the same influence.

7. The kernel should be sufficiently *smooth*. This helps to obtain better results when the particles are disordered.

2.4.2 Some smoothing functions

Many smoothing functions have been developed by a certain number of authors. I will here give the main ones.

Gaussian kernel

[Gingold and Monaghan, 1977] proposed a smoothing function based on an exponential function. It can be written as

$$W(r, h) = \alpha_d e^{-(r/h)^2} \tag{2.21}$$

In this equation, $r = |\mathbf{x} - \mathbf{x}'|$ is the distance between two particles. The coefficient α_d is used to normalise the smoothing function as discussed in the previous section. It has different values in 1, 2 or 3 dimensions. Those values are given in the table 2.1.

The main advantage of this kernel is its smoothness. It is known as very stable even for disordered particles. Its drawback is its theoretically infinite smoothing length. This leads to more computations as the support domain is very large. However, the kernel quickly approaches 0 which allows to consider it with a compact support.

In the equations that will be used later, the first derivative of the smoothing function intervenes. I will give here the expressions of the first derivatives of the kernels in order to be as comprehensive as possible:

$$\frac{dW}{dr} = \frac{\alpha_d}{h} \left(-2 \frac{r}{h} e^{-(r/h)^2} \right) \tag{2.22}$$

The gaussian smoothing function and its first derivative are represented in one dimension in figure 2.6.

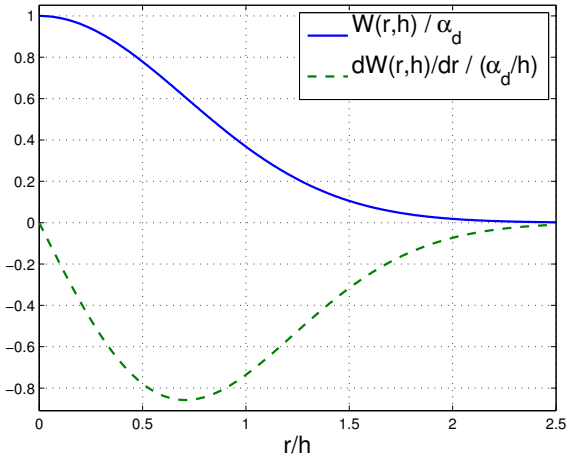


Figure 2.6: Gaussian smoothing function

Bell-shaped kernel

In 1977, the other article introducing SPH was [Lucy, 1977]. It also proposed a smoothing function. It is known as the bell-shaped kernel. Its expression is as follows:

$$W(r,h) = \alpha_d \begin{cases} \left(1+3\frac{r}{h}\right)\left(1-\frac{r}{h}\right)^3 & \frac{r}{h} \leq 1 \\ 0 & \frac{r}{h} > 1 \end{cases} \tag{2.23}$$

Its derivative can be written as:

$$\frac{dW}{dr} = \frac{\alpha_d}{h} \begin{cases} 3\left[\left(1-\frac{r}{h}\right)^3 - \left(1+3\frac{r}{h}\right)\left(1-\frac{r}{h}\right)^2\right] & \frac{r}{h} \leq 1 \\ 0 & \frac{r}{h} > 1 \end{cases} \tag{2.24}$$

The values of α_d are given in table 2.1. A graphical representation of equations (2.23) and (2.24) is given in figure 2.7.

This function is supported on a compact and it extends on a radius of $1h$. This small compact support reduces the calculations that have to be done.

Cubic spline kernel

[Monaghan and Lattanzio, 1985] defined a new kernel known as cubic spline:

$$W(r,h) = \alpha_d \begin{cases} \frac{3}{2} - \left(\frac{r}{h}\right)^2 + \frac{1}{2}\left(\frac{r}{h}\right)^3 & 0 \leq \frac{r}{h} < 1 \\ \frac{1}{6}\left(1-\frac{r}{h}\right)^3 & 1 \leq \frac{r}{h} < 2 \\ 0 & \frac{r}{h} \geq 2 \end{cases} \tag{2.25}$$

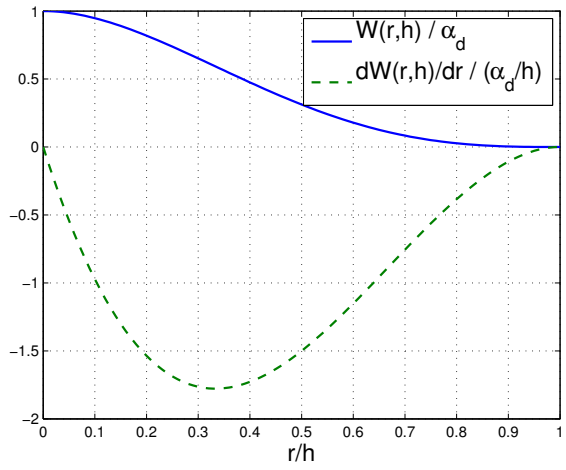


Figure 2.7: Bell-shaped smoothing function

Its first derivative is

$$\frac{dW}{dr} = \frac{\alpha_d}{h} \begin{cases} \frac{3}{2}\left(\frac{r}{h}\right)^2 - 2\frac{r}{h} & 0 \leq \frac{r}{h} < 1 \\ \frac{-1}{2}\left(2 - \frac{r}{h}\right)^2 & 1 \leq \frac{r}{h} < 2 \\ 0 & \frac{r}{h} \geq 2 \end{cases} \tag{2.26}$$

The values of α_d are given in table 2.1. The support domain extends over a radius of $2h$. A graphical representation of equations (2.25) and (2.26) is given in figure 2.8.

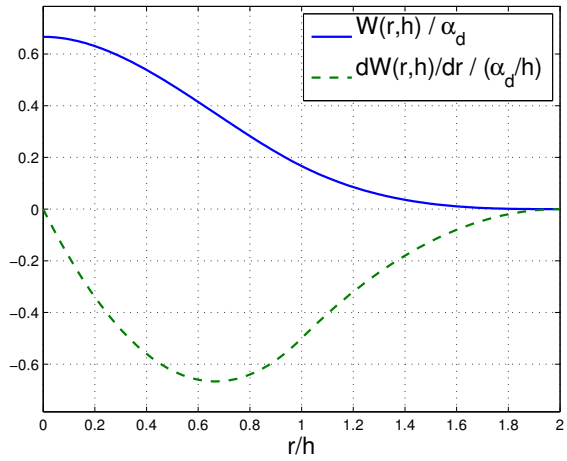


Figure 2.8: Cubic spline smoothing function

The cubic spline has been very widely used for the hydraulic problems. It has the advantage of being supported on a compact. However, it is not as smooth as other functions[2] which can lead to some instabilities according to [Liu and Liu, 2003].

Quadratic kernel

[Johnson et al., 1996] introduced a new kind of smoothing function. It is quadratic and consequently has a linear first derivative. The expression of the kernel is:

$$W(r,h) = \alpha_d \begin{cases} \left(\frac{3}{16}\left(\frac{r}{h}\right)^2 - \frac{3}{4}\frac{r}{h} + \frac{3}{4}\right) & 0 \leq \frac{r}{h} \leq 2 \\ 0 & \frac{r}{h} > 2 \end{cases} \tag{2.27}$$

and the expression of its first derivative is

$$\frac{dW}{dr} = \frac{\alpha_d}{h} \begin{cases} \left(\frac{3}{8}\frac{r}{h} - \frac{3}{4}\right) & 0 \leq \frac{r}{h} \leq 2 \\ 0 & \frac{r}{h} > 2 \end{cases} \tag{2.28}$$

A graphical representation of equations (2.27) and (2.28) is given in figure 2.9.

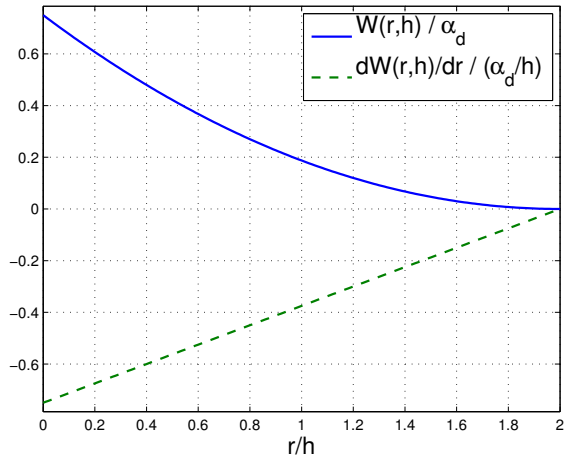


Figure 2.9: Quadratic smoothing function

In figure 2.9, one can observe the first derivative always decreasing (in absolute value) when the distance increases. The authors of this smoothing function consider this aspect as an advantage. In fact, the smoothing function appears often under its first derivative form. If we consider the derivative of the cubic spline (2.26) (figure 2.8), some very close

[2] The second derivative is piecewise linear.

particles may have less influence in comparison to particles situated around $r/h = 2/3$. The quadratic smoothing function does not have this disadvantage.

Considering its smoothness, it can be considered as good when observing figure 2.9, especially when approaching $r/h = 2$.

Quintic kernel

Some higher degree kernels have been developed. I will first show the one developed by [Wendland, 1995] and used in SPHysics[3] as mentioned in [Gomez-Gesteira et al., 2012b]. This smoothing function can be written as

$$W(r,h) = \alpha_d \begin{cases} \left(1 - \frac{1}{2}\frac{r}{h}\right)^4 \left(2\frac{r}{h} + 1\right) & 0 \leq \frac{r}{h} \leq 2 \\ 0 & \frac{r}{h} > 2 \end{cases} \tag{2.29}$$

Its derivative is

$$\frac{dW}{dr} = \frac{\alpha_d}{h} \begin{cases} -5\frac{r}{h}\left(1 - \frac{1}{2}\frac{r}{h}\right)^3 & 0 \leq \frac{r}{h} \leq 2 \\ 0 & \frac{r}{h} > 2 \end{cases} \tag{2.30}$$

This kernel is supported on a compact ($\kappa = 2$) and the values of the normalisation coefficient (α_d) are given in table 2.1. A graphical representation of equations (2.29) and (2.30) is given in figure 2.10.

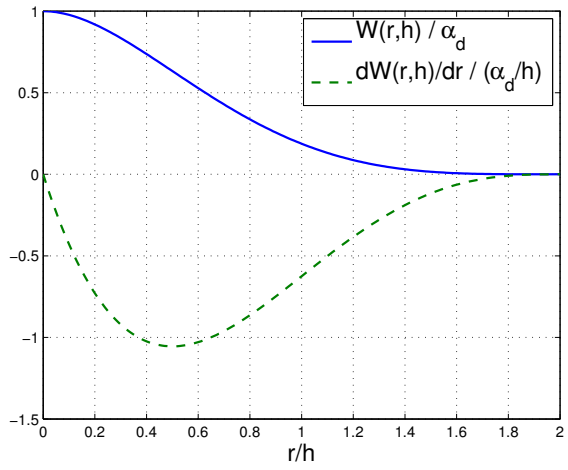


Figure 2.10: Quintic smoothing function

[3]SPHysics is an open-source SPH code developed by the University of Vigo (Spain), the University of Manchester (UK) and the John Hopkins University (USA). It is mainly used for the study of coastal processes. More information can be found on the project website: `www.sphysics.org`.

Quintic spline kernel

Another high order smoothing function is the one proposed by [Morris, 1996]:

$$W(r,h) = \alpha_d \begin{cases} \left(3-\frac{r}{h}\right)^5 - 6\left(2-\frac{r}{h}\right)^5 + 15\left(1-\frac{r}{h}\right)^5 & 0 \le \frac{r}{h} < 1 \\ \left(3-\frac{r}{h}\right)^5 - 6\left(2-\frac{r}{h}\right)^5 & 1 \le \frac{r}{h} < 2 \\ \left(3-\frac{r}{h}\right)^5 & 2 \le \frac{r}{h} < 3 \\ 0 & \frac{r}{h} \ge 3 \end{cases} \tag{2.31}$$

Its first derivative is:

$$\frac{dW}{dr} = \frac{\alpha_d}{h} \begin{cases} -5\left(3-\frac{r}{h}\right)^4 + 30\left(2-\frac{r}{h}\right)^4 - 75\left(1-\frac{r}{h}\right)^4 & 0 \le \frac{r}{h} < 1 \\ -5\left(3-\frac{r}{h}\right)^4 + 30\left(2-\frac{r}{h}\right)^4 & 1 \le \frac{r}{h} < 2 \\ -5\left(3-\frac{r}{h}\right)^4 & 2 \le \frac{r}{h} < 3 \\ 0 & \frac{r}{h} \ge 3 \end{cases} \tag{2.32}$$

A graphical representation of equations (2.31) and (2.32) is given in the figure 2.11.

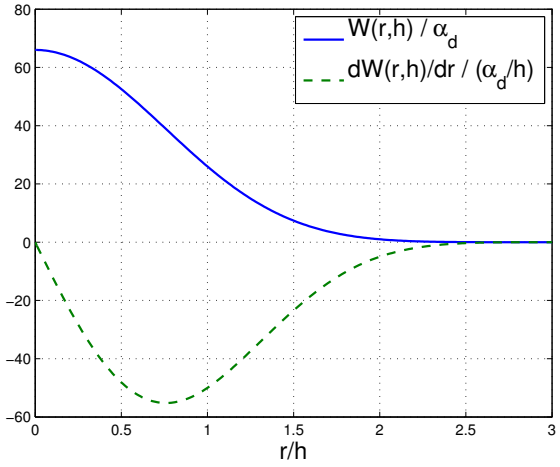


Figure 2.11: Quintic spline smoothing function

The advantage of this smoothing function is that it is very close to the Gaussian kernel. The main drawback is the larger smoothing length ($\kappa = 3$) in comparison to the other smoothing functions. Nevertheless, it is still supported on a compact which is an advantage compared to the Gaussian kernel. The values of α_d are given in table 2.1.

Comparison of the kernels

A first comparison of the smoothing lengths (κh) can be obtained by observing the values of κ for the different smoothing functions (table 2.1). A large smoothing length will tend to increase the number of particles in the support domain. This will result in a longer computation time. The Gaussian and the quintic spline are subject to larger calculation times.

Kernel	α_d **in 1D**	α_d **in 2D**	α_d **in 3D**	κ
Gaussian	$1/(\pi^{1/2}h)$	$1/(\pi h^2)$	$1/(\pi^{3/2}h^3)$	∞
Bell-shaped	$5/(4h)$	$5/(\pi h^2)$	$105/(16\pi h^3)$	1
Cubic spline	$1/h$	$15/(7\pi h^2)$	$3/(2\pi h^3)$	2
Quadratic	$1/h$	$2/(\pi h^2)$	$5/(4\pi h^3)$	2
Quintic	$3/(4h)$	$7/(4\pi h^2)$	$21/(16\pi h^3)$	2
Quintic spline	$1/(120h)$	$7/(478\pi h^2)$	$3/(359\pi h^3)$	3

Table 2.1: Normalisation coefficients (α_d) and scaling factors (κ) for some smoothing functions

Figure 2.12 compares the six smoothing functions developed above. Figure 2.13 compares the first derivatives of these functions.

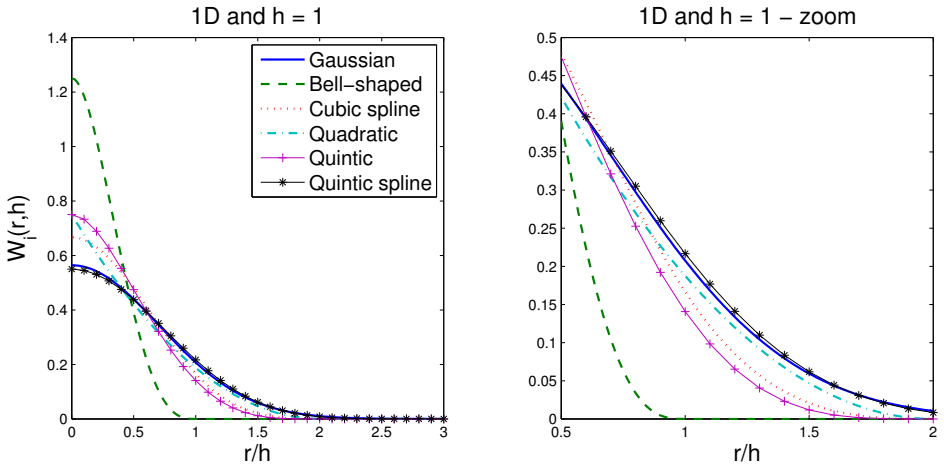


Figure 2.12: Comparison of six smoothing functions

The first observation that can be made is that the bell-shaped function is very different from the others. In fact, it is the only one to use $\kappa = 1$. Then, if we consider the Gaussian kernel to be the best suited for SPH simulation, as stated in [Monaghan, 1992], the best smoothing functions should be approaching the Gaussian kernel. The Gaussian kernel itself presents the advantage to be very stable. Unfortunately, its theoretical infinite smoothing length is a real disadvantage.

Thus, other functions which tries to approach the Gaussian kernel have been developed.

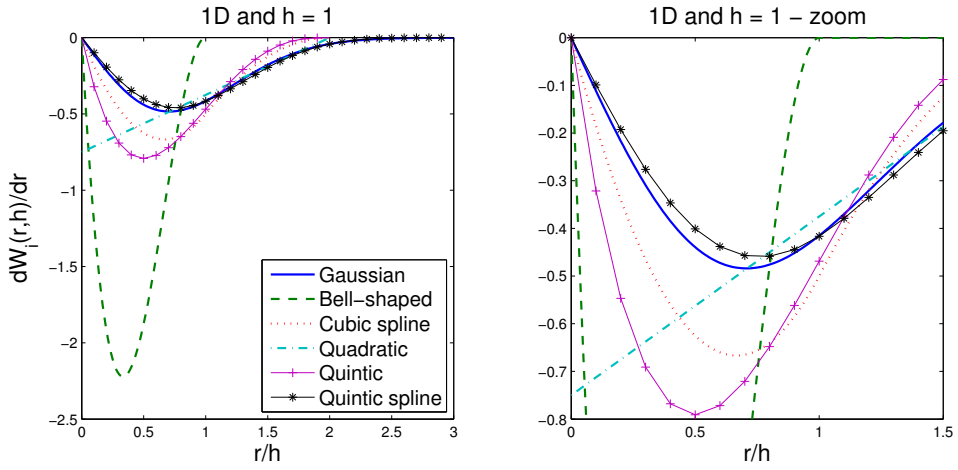


Figure 2.13: Comparison of six smoothing functions' first derivatives

The cubic spline and its first derivative are not so far from the Gaussian kernel but better results can be found. However, this function has a compact support.

The quadratic smoothing function has the advantages of being compactly supported, having a decreasing (in absolute value) first derivative and being close to the Gaussian smoothing function, at least for $r/h > 0.5$.

The quintic kernel is supported on a compact. Even if it has a high order and is used in SPHysics, it does not come very close to the Gaussian. I do not think that this kernel is the best alternative.

Finally, the quintic spline smoothing function comes very close to the Gaussian kernel and it has a compact support. I think that this function is the best alternative to the Gaussian kernel.

Given the observations made above, the smoothing functions that will be preferred and implemented in the code that will be developed later are:

- the cubic spline kernel,
- the quadratic kernel and,
- the quintic spline kernel.

Chapter 3

The SPH method applied to hydraulics

So far we have seen the basic ideas of the SPH method. Now, I will introduce some concepts relative to the SPH method applied to hydraulics.

First of all, I will develop the Navier-Stokes equation in the Lagrangian form. Then, I will apply the SPH method to those equations. Finally, I will introduce some solutions relative to the smoothing length, the boundary conditions, etc.

3.1 Navier-Stokes equations

The Navier-Stokes equations describe mainly two fundamental principles:

1. The continuity, i.e. the conservation of the mass in the system. If there is no inflow nor outflow, the mass of the system remains constant. This law is from Lavoisier:

   "Nothing is lost, nothing is created, everything is transformed."

2. The conservation of momentum. This is relative to Newton's second law which can be expressed today[1] as:

   "When viewed from an inertial reference frame, the change of momentum of a body is proportional to the forces applied on this body."

There is also a third principle described by Navier-Stokes equations:

3. The energy conservation. It states that no energy is created nor lost if the system is closed.

[1] In Newton's *Principia Mathematica* (1687) the law was expressed in Latin as:

"Mutationem motus proportionalem esse vi motrici impressae, et fieri secundum lineam rectam qua vis illa imprimitur."

However, this third principle will not be used in the following developments. This third equation is rarely used in hydrodynamic applications. For example, [Gomez-Gesteira et al., 2010; Crespo et al., 2007; Monaghan, 1994] do not use this equation. This is due to the fact that the temperature of the fluid is assumed to be constant. Thus, the thermal energy is constant. Only the kinetic and the potential energy change during the simulation time. These changes are already included in the two first principles.

As explained earlier, the SPH method is a Lagrangian method. This means that the Navier-Stokes equations must be expressed in a Lagrangian form.

3.1.1 Analytical development in the Lagrangian formalism

In this subsection, I will establish the Navier-Stokes equations in their Lagrangian form. The different steps of the development will be explained.

The Navier-Stokes equations are often established in the Eulerian formalism. This is done for example in [Ryhming, 2004]. In that situation, the control volume used to establish the equations is fixed. In the Lagrangian formalism, the observer moves with the flow. This leads to a mobile control volume. This situation is depicted in figure 3.1. The development of the Navier-Stokes equations is explained in [Liu and Liu, 2003].

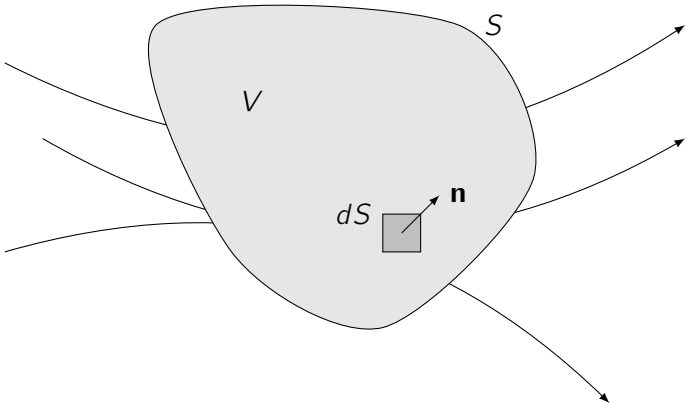


Figure 3.1: Control volume in the Lagrangian formalism

The total mass of the volume remains constant as we are working in a Lagrangian formalism. In order to keep a constant mass, the volume V and the surface S must change while the fluid contained in the volume is compressed, expands, etc. This means that to keep the same particles of fluid inside the volume, the surface S must change. The movement of the surface is induced by the movement of the fluid inside the volume V. A change of S directly leads to a change of V. This can be written mathematically as

$$\Delta V = \int_S \mathbf{u} \cdot \mathbf{n} \, \Delta t \, dS \tag{3.1}$$

with $\mathbf{n}$ the unit normal vector to the surface S. Dividing by Δt leads to

$$\frac{\Delta V}{\Delta t} = \int_S \mathbf{u} \cdot \mathbf{n} \; dS \tag{3.2}$$

Thanks to the divergence theorem (2.8), equation (3.2) becomes

$$\frac{\Delta V}{\Delta t} = \int_V \nabla \cdot \mathbf{u} \; dV \tag{3.3}$$

For a very small volume δV where the properties of the fluid can be considered constant, a particle derivative can be used:

$$\begin{aligned} \frac{D(\delta V)}{Dt} &= \int_{\delta V} \nabla \cdot \mathbf{u} \; d(\delta V) \\ &= \nabla \cdot \mathbf{u} \int_{\delta V} 1 \; d(\delta V) \\ &= \nabla \cdot \mathbf{u} \; \delta V \end{aligned} \tag{3.4}$$

Conservation of mass

The density ρ of the fluid contained in a very small volume δV can be considered constant. Thus, the mass of that volume is

$$\delta m = \rho \; \delta V \tag{3.5}$$

We know that the mass of a Lagrangian control volume doesn't change in time:

$$\frac{D(\delta m)}{Dt} = \frac{D(\rho \; \delta V)}{Dt} = 0 \tag{3.6}$$

Equation (3.6) can be rewritten:

$$\frac{D\rho}{Dt}\delta V + \rho \frac{D(\delta V)}{Dt} = 0 \tag{3.7}$$

Thanks to equation (3.4), we obtain

$$\frac{D\rho}{Dt} = -\rho \; \nabla \cdot \mathbf{u} \tag{3.8}$$

Equation (3.8) is the equation of conservation of mass written in the Lagrangian formalism.

Conservation of momentum

To express the conservation of momentum, I will use a 2-D control volume. This control volume and the forces acting on it in the x direction are represented in figure 3.2.

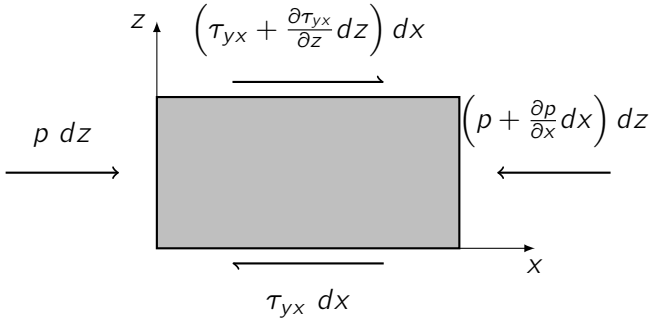


Figure 3.2: 2-D control volume and the forces acting in the x direction

The equilibrium of the control volume of figure 3.2 along the x direction can be written as:

$$\begin{aligned} \rho\ dx\ dz \frac{Du_x}{Dt} &= p\ dz - \left(p + \frac{\partial p}{\partial x} dx\right) dz + \left(\tau_{yx} + \frac{\partial \tau_{yx}}{\partial z} dz\right) dx - \tau_{yx}\ dx \\ &= -\frac{\partial p}{\partial x} dx\ dz + \frac{\partial \tau_{yx}}{\partial z} dx\ dz \end{aligned} \tag{3.9}$$

After adding the body forces in the x direction and having simplified equation (3.9), we obtain

$$\rho \frac{Du_x}{Dt} = -\frac{\partial p}{\partial x} + \frac{\partial \tau_{yx}}{\partial z} + \rho F_x \tag{3.10}$$

Equation (3.10) can be generalised in 3-D and written with a vectorial formalism:

$$\rho \frac{D\mathbf{u}}{Dt} = -\nabla p + \nabla \cdot \mathbf{T} + \rho \mathbf{F} \tag{3.11}$$

with $\mathbf{T}$ a second-order tensor which contains the stresses τ_{ij}. Equation (3.11) is the second Navier-Stokes equation.

3.1.2 Application to the SPH method

Now that we have the equations that express the conservation of mass (3.8) and the conservation of momentum (3.11), I will use the SPH formalism in order to implement them.

Conservation of mass

In order to express the conservation of the mass, there is more than one technique [Liu and Liu, 2003]. One of them is to use equation (2.13) developed earlier. When using $f(\mathbf{x}) = \rho$, equation (2.13) becomes

$$\rho_a = \sum_{b=1}^{N} m_b W_{ab} \tag{3.12}$$

with a and b which represent the current particle and the neighbouring ones respectively. Equation (3.12) is called the *summation density*. It represents the essence of the SPH method but doesn't take into account the conservation of mass (3.8) established earlier.

Another way to calculate the density of a particle is to use the conservation of mass (3.8). It can also be written as

$$\frac{D\rho}{Dt} = \mathbf{u} \cdot \nabla\rho - \nabla \cdot (\rho\mathbf{u}) \tag{3.13}$$

Using equation (2.15) for the two terms of the RHS:

$$\mathbf{u} \cdot \nabla\rho = \mathbf{u}_a \sum_{b=1}^{N} \frac{m_b}{\rho_b} \rho_b \nabla_a W_{ab} = \sum_{b=1}^{N} m_b \mathbf{u}_a \nabla_a W_{ab} \tag{3.14}$$

$$\nabla \cdot (\rho\mathbf{u}) = \sum_{b=1}^{N} \frac{m_b}{\rho_b} \rho_b \mathbf{u}_b \nabla_a W_{ab} = \sum_{b=1}^{N} m_b \mathbf{u}_b \nabla_a W_{ab} \tag{3.15}$$

We can get what is called the *continuity density* by using equations (3.13), (3.14) and (3.15):

$$\frac{D\rho_a}{Dt} = \sum_{b=1}^{N} m_b \mathbf{u}_{ab} \nabla_a W_{ab} \tag{3.16}$$

with $\mathbf{u}_{ab} = \mathbf{u}_a - \mathbf{u}_b$.

The *continuity density* (3.16) will be preferred to the *summation density* (3.12) for two main reasons:

- The continuity density equation represents the physical behaviour described by the Navier-Stokes equations.
- The density will be updated thanks to the relative velocity between two particles. This update will directly lead to a change in the pressure field. This will avoid interpenetration between particles.

It can be seen from equation (3.16) that mainly two parameters will influence the change of the density of a particle:

1. The relative velocities between the particles: if two particles are approaching each other quickly, their densities will increase. A density increase leads to a pressure increase as it will be seen in a few lines. This pressure increase creates a repulsive force between the particles which avoids interpenetration. This means that the equation of conservation of mass will directly influence the conservation of momentum.
2. The distance between the particles: the closer the particles, the bigger the gradient of the kernel. This leads to a larger change rate of ρ. However, for very close particles, this is not always true. This has already been discussed in the section about the smoothing functions.

Conservation of momentum

I will first neglect the friction term $\mathbf{T}$ and the body forces $\mathbf{F}$ of the conservation of momentum (3.11):

$$\frac{D\mathbf{u}}{Dt} = -\frac{1}{\rho}\nabla p \tag{3.17}$$

The RHS of equation (3.17) can also be written as

$$-\frac{1}{\rho}\nabla p = -\left(\nabla\left(\frac{p}{\rho}\right) + \frac{p}{\rho^2}\nabla\rho\right) \tag{3.18}$$

When relation (2.15) is applied to the terms of the RHS of equation (3.18), we have:

$$\nabla\left(\frac{p}{\rho}\right) = \sum_{b=1}^{N} m_b \frac{p_b}{\rho_b^2}\nabla_a W_{ab} \tag{3.19}$$

$$\frac{p}{\rho^2}\nabla\rho = \frac{p_a}{\rho_a^2}\sum_{b=1}^{N}\frac{m_b}{\rho_b}\rho_b\nabla_a W_{ab} = \sum_{b=1}^{N} m_b\frac{p_a}{\rho_a^2}\nabla_a W_{ab} \tag{3.20}$$

Equation (3.17) can be rewritten thanks to equations (3.18), (3.19) and (3.20):

$$\frac{D\mathbf{u}_a}{Dt} = -\sum_{b=1}^{N} m_b\left(\frac{p_b}{\rho_b^2} + \frac{p_a}{\rho_a^2}\right)\nabla_a W_{ab} \tag{3.21}$$

The body forces can be added to equation (3.21). The friction forces will be considered in this work with an artificial viscosity denoted Π_{ab} [Monaghan, 1992]. This term will be discussed in the following pages. The conservation of momentum is written in the SPH formalism as

$$\frac{D\mathbf{u}_a}{Dt} = -\sum_{b=1}^{N} m_b\left(\frac{p_b}{\rho_b^2} + \frac{p_a}{\rho_a^2} + \Pi_{ab}\right)\nabla_a W_{ab} + \mathbf{F} \tag{3.22}$$

Equation (3.22) shows that the change of the motion of a particle is due to the pressure field, the viscosity (a virtual viscosity in this case) and the body forces acting on the fluid such as gravity.

In order to have a better understanding of this equation, I will apply it to a very simple case where there is no gravity and where a constant pressure field is applied. The situation is depicted in figure 3.3. In the beginning, the particles are motionless. I will check now that $D\mathbf{u}_a/Dt = 0$. First of all, the viscosity term $\Pi_{ab} = 0$ for all the particles as they have no velocity at the beginning[2]. The terms p_a/ρ_a^2 and p_b/ρ_b^2 are not equal to zero and positive. The last term $\nabla_a W_{ab}$ is the gradient of the kernel, i.e. a vector. This term gives a direction and a part of the intensity of the force applied from the particles b_i to the particle a. Its expression is given at equation (2.16). If the masses m_b are equal and the particles arranged in a Cartesian lattice, the result $D\mathbf{u}_a/Dt$ is equal to 0. A representation of the vectors $\nabla_a W_{ab}$ is given in figure 3.3. It can be seen that the vectors of the particles b_1, b_3, b_5 and b_7 are smaller than for the other particles. This is due to the fact that those particles are further from a. Thus, the value of $\partial W/\partial r$ is smaller.

[2]This fact will be highlighted in the section that is relative to the artificial viscosity.

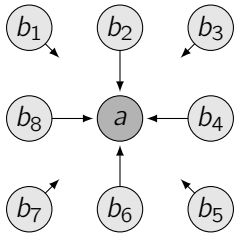


Figure 3.3: Very simple situation to understand the conservation of momentum

3.2 Numerical and implementation aspects

We have now the two main equations necessary to solve an hydrodynamic problem with the SPH method. Neverthless, a few points still have to be clarified. Those points concern the temporal integration, the equation of state, the viscosity, the boundary conditions, etc.

3.2.1 Equation of state

As seen in the continuity equation (3.16), the density is updated. In equation (3.22), the densities as well as the pressures are used to update the velocity. The pressure is not a direct variable of the problem. However, we can calculate the pressure thanks to an equation of state which links the density to the pressure. At the end of this subsection, I will also explain how we can compute the speed of the sound c in the fluid.

Different possibilities are available to link the pressure to the density. We can consider the fluid as:

1. an ideal gas or,
2. a quasi-incompressible fluid.

Ideal gas law

The ideal gas law is often expressed as

$$p = \frac{\rho\, R\, T}{M} \tag{3.23}$$

where p [N/m^2] is the pressure, ρ [kg/m^3] the density, $R = 8.314$ [J/(K mol)] the ideal gas constant, T [K] the temperature and M [kg/mol] is the molar mass. The temperature will be considered constant and will be taken at 20°C, i.e. 293.15°K. Equation (3.23) gives an absolute pressure. Indeed, if we consider a point at the free-surface its pressure is null. Then, by equation (3.23), the density ρ should be null as well. Physically it is not the case.

Equation (3.23) will be transformed in a relative form:

$$p = \frac{R\, T}{M}\left(\frac{\rho}{\rho_0} - 1\right) \tag{3.24}$$

where ρ_0 is the reference density.

From equation (3.24), we easily notice a linear evolution of the pressure with respect to the density. Equation (3.24) is represented in figure 3.4. We see that for high pressures (100 bars) the density can increase up to 75000 kg/m^3. This means that the fluid is highly compressible.

This is not very realistic for the case of water. In fact, it can be easily understood that the density of water will barely increase with an increasing pressure. This is the reason why we need another equation of state for water.

Quasi-incompressible fluid

To solve the problem introduced above, [Monaghan, 1994] used another equation of state. This one is based on the fact that water is weakly compressible. He explained that the variation of density in a fluid flow is proportional to M^2, where M is the Mach number. The equation of state proposed is

$$p = B\left(\left(\frac{\rho}{\rho_0}\right)^{\gamma} - 1\right) \tag{3.25}$$

with $\gamma \approx 7$ and B a term relative to the speed of sound. [Landau and Lifshitz, 1971] explain that the speed of sound is linked to the density by the following relation:

$$c^2 = \left(\frac{\partial p}{\partial \rho}\right)_s \tag{3.26}$$

at constant entropy. By applying this relation to the equation of state (3.25), we obtain

$$B = \frac{c_0^2 \rho_0}{\gamma} \tag{3.27}$$

More details are available in the appendix A.1.

In order to understand the origin of this equation of state, I have reviewed the literature that concerns it. Equation (3.25) proposed by Monaghan is derived from [Batchelor, 1967]. In that book, the equation is written as

$$\frac{p + B}{1 + B} = \left(\frac{\rho}{\rho_0}\right)^{\gamma} \tag{3.28}$$

If B is considered very big[3], we can find back equation (3.25). This equation is established for oceanic purposes where large pressures are measured.

Batchelor references [Cole, 1948] who references the Tait equation.

The Tait equation has been established by the Scottish mathematician and physician Peter Guthrie Tait during a scientific expedition between 1873 and 1876 [Dymond and Malhotra, 1988]. His work was to investigate the physical conditions of the deep seas.

[3] In practice, for water, $B \approx 3\ 10^8$ [N/m^2] with $c_0 = 1480$ [m/s].

He made a number of measurements of density for different depths in the ocean. After these measurements he established an empirical law considered today as one of the most satisfactory. The equation established is isothermal.

According to [Dymond and Malhotra, 1988], what is called *the true Tait equation* is written

$$\frac{V_0 - V}{p\, V_0} = \frac{A}{\pi + p} \tag{3.29}$$

where A and π are quantities to be found and V a volume. This equation can be written in a different way by replacing the compressibility coefficient by a differential operator:

$$\frac{\Delta V}{\Delta p} = \frac{A}{B + p} \tag{3.30}$$

After integrating, we obtain what is called *the modified Tait equation*:

$$V = V_0 \left(1 - A \ln\left(1 + \frac{p}{B}\right)\right) \tag{3.31}$$

[Cole, 1948] uses equation (3.30) written with a derivative:

$$-\left(\frac{\partial V}{\partial p}\right)_T = \frac{1}{\gamma\,(p + B(T))} \tag{3.32}$$

with $B(T)$ a function of the temperature. The next step is to integrate at constant entropy the following expression:

$$-\frac{1}{V}\left(\frac{\partial V}{\partial p}\right)_s = \frac{1}{\gamma\,(p + B(s))} \tag{3.33}$$

After integration (see appendix A.2 for details) we obtain equation (3.25). The value of γ is approximately 7 according to experiments.

The main features of this equation of state are:

1. Equation (3.29) is established thanks to experimental measures for sea water for large pressures. The Tait equation is empirical but widely used and recognised.

2. The integrated form (3.25) assumes that the phenomenon is isentropic.

Equation (3.25) is depicted in figure 3.4. We notice a quicker increase in density with pressure with the ideal gas law than with a quasi-incompressible fluid law.

When equation (3.25) is plotted separately, it can be seen that the evolution of $\rho(p)$ is quasi linear up to large pressures such as 100 bars. This is shown in figure 3.5. The error for 100 bars is lower than 0.01%. The two first terms of the Taylor expansion[4] of equation (3.25) are:

$$p = 0 + B\,\gamma\left(\frac{\rho}{\rho_0} - 1\right) + \mathcal{O}\left(\frac{\rho}{\rho_0} - 1\right)^2 \tag{3.34}$$

[4]The Taylor expansion is performed around $\rho/\rho_0 = 1$.

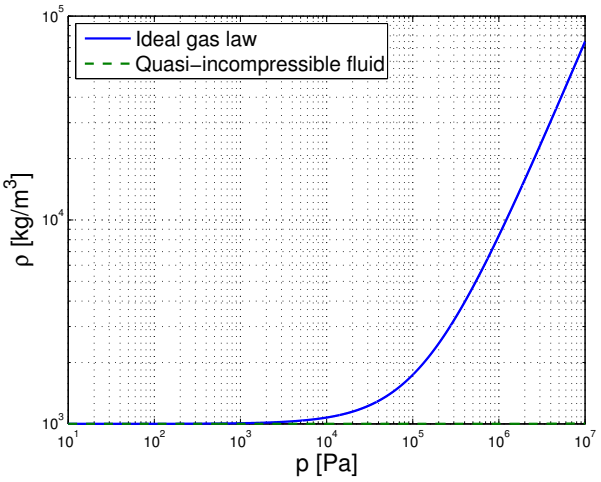


Figure 3.4: Comparison between the two equations of state

Neglecting the last term,

$$p = B\,\gamma\left(\frac{\rho}{\rho_0} - 1\right) = c_0^2\,\rho_0\left(\frac{\rho}{\rho_0} - 1\right) \tag{3.35}$$

This shows that the evolution of density in respect to pressure can be considered linear for a practical range of pressures. It also shows that γ has no influence for practical pressures.

However, equation (3.25) will be used instead of equation (3.35). As explained later, the speed of sound can be lessened in order to increase the time steps. This leads to the impossibility to use a linear expression for the equation of state. This is shown in figure 3.6 for $c_0 = 100$ m/s.

Remarks about the speed of the sound

The use of a lower speed of sound can have a significant impact on the resolution time. This is why [Monaghan and Kos, 1999] suggest to use a speed of sound $c = 10\sqrt{gH}$, with H the biggest depth. This is possible because the flows studied have a free surface.

The use of a different speed of sound has a direct impact on the compressibility of the fluid. This can be easily observed in figures 3.5 and 3.6. Indeed, for a pressure of 10 bars, a density of 1000.5 kg/m^3 is obtained for $c_0 = 1480$ m/s and a density of 1080 kg/m^3 is obtained for $c_0 = 100$ m/s. However, [Monaghan, 1994] states that the density variations stay consistent. It will be seen later that this assumption is not always relevant.

During the calculation, the speed of sound can be deduced from the density of the particles. Using equation (3.26), the speed of sound is

$$c = c_0 \tag{3.36}$$

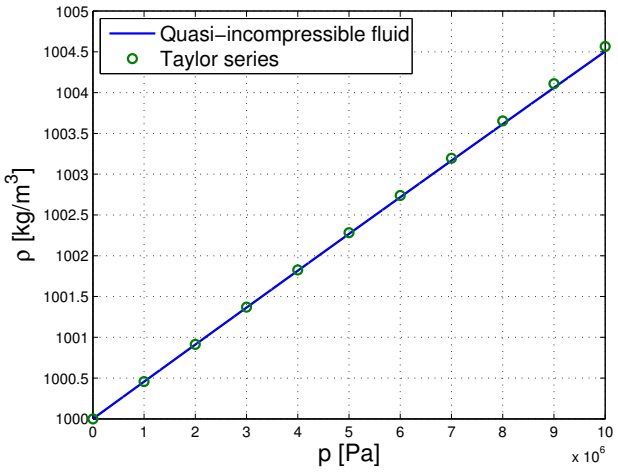


Figure 3.5: Evolution of density in respect to pressure for a quasi-incompressible fluid ($c_0 = 1480$ m/s)

for an ideal gas law and

$$c = c_0 \sqrt{\left(\frac{\rho}{\rho_0}\right)^{\gamma-1}} \tag{3.37}$$

for a quasi-incompressible fluid.

Setting of the hydrostatic pressure

For some situations, an hydrostatic pressure is present in the fluid at the beginning of a simulation. This is the case, for example, for a dam break. This means that an hydrostatic pressure must be initialised at the beginning of the simulation. However, the pressure is not a variable attached to a particle. Indeed, the pressure is deduced from the density which is a variable of a particle. Thus, the initial pressure field within the fluid is set thanks to the initial densities of the particles.

This is done thanks to the following formulations (details in the appendix A.3):

1. For an ideal gas law,
$$\rho = \rho_0 \left(1 + \frac{M}{RT}\rho_0\ g\ H\right) \tag{3.38}$$
2. For a quasi-incompressible fluid law,
$$\rho = \rho_0 \left(1 + \frac{1}{B}\rho_0\ g\ H\right)^{1/\gamma} \tag{3.39}$$

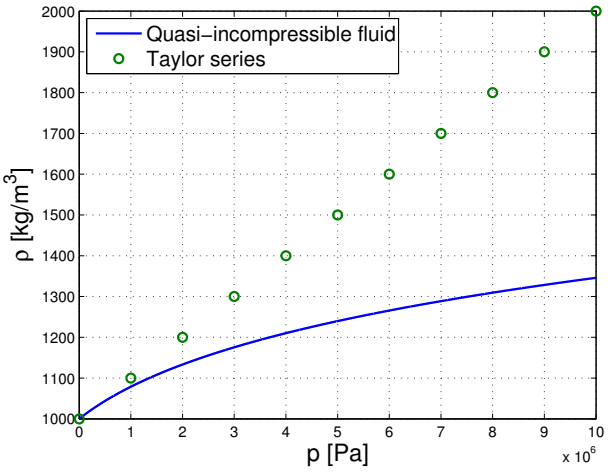


Figure 3.6: Evolution of density in respect to pressure for a quasi-incompressible fluid ($c_0 = 100$ m/s)

3.2.2 Artificial viscosity

As seen in equation (3.22), which describes the conservation of momentum, an artificial viscosity term Π_{ab} is present. This term has two main purposes:

1. to create a viscosity term in order to reproduce the friction that occurs in a real fluid and,
2. to avoid numerical instabilities when the particles are moving away from each other.

This artificial viscosity has been introduced by [Monaghan, 1992]. Its expression is

$$\Pi_{ab} = \begin{cases} \dfrac{-\alpha \bar{c}_{ab} \mu_{ab} + \beta \mu_{ab}^2}{\bar{\rho}_{ab}} & \text{for } \mathbf{u}_{ab} \cdot (\mathbf{x}_a - \mathbf{x}_b) < 0 \\ 0 & \text{for } \mathbf{u}_{ab} \cdot (\mathbf{x}_a - \mathbf{x}_b) \geq 0 \end{cases} \tag{3.40}$$

with

- $\bar{c}_{ab}$ the mean speed of sound of the particles a and b
- $\bar{\rho}_{ab}$ the mean density of the particles a and b
- μ_{ab} expressed as

$$\mu_{ab} = \frac{h\, \mathbf{u}_{ab} \cdot (\mathbf{x}_a - \mathbf{x}_b)}{(\mathbf{x}_a - \mathbf{x}_b)^2 + \eta^2} \tag{3.41}$$

with $\eta^2 = 0.01 h^2$ which is used to avoid singularities.

- α and β two coefficients that must be chosen by the user. [Monaghan, 1992] suggests the values to be around 1 for α and 2 for β. However, in practice, β is often taken at 0 and $\alpha \in [0.01; 0.5]$.

The term of equation (3.40) associated to α produces a bulk and shear viscosity. The term associated to β is useful to handle shocks.

The artificial viscosity is quite easy to implement and very flexible for the user. However, the values given to α or β have no physical meaning. They are only used to stabilise the system and produce a shear force. The lack of physical values associated to this formulation may be a difficulty when the numerical method is used to predict the behaviour of a problem for which the result is unknown. This emphasizes the need for a physical formulation of the viscosity. This kind of formulation is developed in [Violeau, 2012] from the Navier-Stokes equations and the dynamic viscosity (often noted μ) appears clearly. This formulation has not been implemented in the context of this master thesis. Nevertheless, it would be interesting to include this kind of viscosity for further works.

3.2.3 Smoothing length

The smoothing length h is an important parameter for the SPH method. Indeed, it influences directly the number of neighbours that will intervene in the summations. The first thing to do is to choose an initial smoothing length. In [Liu and Liu, 2003], $h = 1.2s$ with s the initial spacing between the particles. Other authors take the smoothing length as $1s$, $1.5s$ or $2s$. The choice made for this master thesis is $h = 1.2s$ as [Liu and Liu, 2003] or [Crespo et al., 2007] did for their work. This choice has been made because it offers a good compromise between computational efficiency and the number of particles that are part of the support domain.

The choice of h is important as it will influence the number of neighbours. Too many neighbours will lead to a greater calculation time but it may also lead to non-physical phenomenon as the properties of the closer particles will be smoothed by the further ones. On the other hand, not enough neighbours would not represent well the physics of the problem. The influence of this choice is represented in figure 3.7 for $\kappa = 2$.

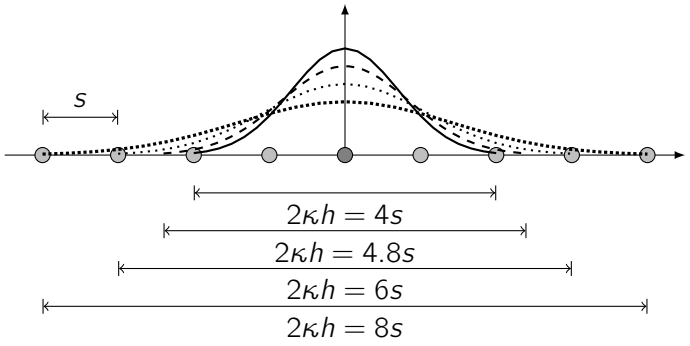


Figure 3.7: Influence of the smoothing length on the number of neighbours in 1D

During the computation, the particles are moving away from each other (the density decreases) or toward each other (the density increases). If h stays constant for a given particle, the number of neighbours will vary while the density increases or decreases. This may lead to accuracy problems. To avoid this, the smoothing length can be updated in order to keep a constant number of neighbours. Two techniques are mainly available:

1. update all the smoothing lengths together at every time step or,
2. update the smoothing lengths separately at every time step.

The second possibility is more complicated to handle. Indeed, the support and influence domain of a particle (see chapter 2) can be different. This will lead to the violation of Newton's third law. In order to avoid this violation, [Nelson and Papaloizou, 1994] explain that additional terms must be added to the equation of conservation of momentum. An easier way to make the smoothing length evolute with the change of density is to update all the smoothing lengths at once with the same value. This is done with the formulation suggested by [Monaghan and Lattanzio, 1985]:

$$h = h_0 \left(\frac{\rho_0}{\rho} \right)^{1/d} \tag{3.42}$$

with d the number of dimensions and ρ the average density. The drawback of this method is for local low or high densities. Indeed, the update of h is made for the average density and not for local densities. A lack of information is possible for local high or low densities.

For this work, equation (3.42) has been chosen for its simplicity and the good results it provides in the case of hydrodynamic problems.

3.2.4 Nearest neighbours search

The smoothing function W is supposed to be compactly supported. This means that the number of neighbours is finite. In order to calculate the value of $\nabla_a W_{ab}$ or W_{ab}, the neighbours of a particle must be known. To achieve this the particles within a given radius must be found thanks to a nearest neighbours search algorithm.

[Liu and Liu, 2003] identify three algorithms:

1. all-pair search algorithm
2. linked-list algorithm
3. tree search algorithm

The first one is the least efficient one. In fact, in order to find the neighbours of one particle, all the particles of the domain are scanned and the distances r_{ab} verified. This leads to a computing time of the order of N^2 [5], which is not very efficient.

[5] N is the number of particles in the domain.

The search of neighbours being achieved several times (N) every iteration, it must be as efficient as possible. The linked-list algorithm consists of dividing the domain in cells with sides of κh and of sorting the particles in its corresponding cell. When it is done, the neighbours of a particle are in the neighbouring cells. This limits the number of calculations of r_{ab}. This method is represented in figure 3.8 for a 2-D domain and randomly distributed particles. This algorithm is much more efficient than the previous one. According to [Liu and Liu, 2003], its efficiency is estimated to $\mathcal{O}(N)$ if the number of particles per cell is low enough.

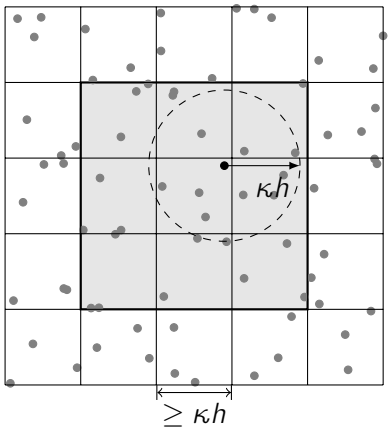


Figure 3.8: Linked-list algorithm in 2D

The only drawback of this algorithm is that it is not very efficient for variable smoothing lengths in space. However, for this master thesis, the smoothing length was chosen constant in space.

The last algorithm is called the tree search algorithm. The domain is divided into smaller sub-domains that contain only one particle. The particles are sorted along a tree. The efficiency of this algorithm is $\mathcal{O}(N \log N)$. It is slower than the previous one but it can handle more efficiently variable smoothing lengths in space. More information about this algorithm can be found in [Liu and Liu, 2003].

The linked-list algorithm described above is used for this work. It offers the most efficient way to find the neighbours of a particle when the smoothing length is constant in space. The way to implement this kind of algorithm will be explained in the next chapter.

3.2.5 Numerical precision

During the implementation of the code, I noticed that the numerical precision for the real numbers is essential. This kind of problem may not be an issue in many applications but it is important to point out that for a few situations it might lead to some troubles.

First of all, I will recall some basis about the representation of real numbers in the memory of a computer [Boigelot, 2009]. A real number is represented thanks to a set of bits. The bits are distributed between the sign, the mantissa and the exponent. The

number of bits used for each part is given in table 3.1. The mantissa is a number f that can be calculated thanks to

$$f = 1 + \sum_{i=1}^{m} b_i 2^{-i} \tag{3.43}$$

with m the number of bits used for the mantissa and b_i the value of a bit (1 or 0). Formula (3.43) can only be used for non-extreme exponents.

From equation (3.43), we can deduce the number of significant digits of a given representation. This information is given in table 3.1.

	Sign	**Mantissa**	**Exponent**	**Total**	**Significant digits**
Single precision	1	8	23	32 bits	7
Double precision	1	11	52	64 bits	16

Table 3.1: Representation of a real number in the IEEE 754 standard

Now, let me take a simple case where a column of particles is set with an hydrostatic pressure. The particles are distant of 0.01 m. The table 3.2 gives the densities of two neighbouring particles placed on a same vertical line. The densities are calculated with the equation for quasi-incompressible fluids (3.39) for two different c_0.

	Depth [m]	**Pressure** [Pa]	ρ **[kg/m^3]** with $c_0 = 1480$ m/s	ρ **[kg/m^3]** with $c_0 = 30$ m/s
Part. 1	0.50	4905.0	1000.0022393	1003.957
Part. 2	0.51	5003.1	1000.0022841	1004.035

Table 3.2: Densities of two particles when the hydrostatic pressure applies

It can be seen from table 3.2 that a double precision must be used for $c_0 = 1480$ m/s in order to have enough precision to distinguish the densities of the two particles. Indeed, it is only the ninth significant digit that makes the difference between the two particles. If a single precision was used, no pressure difference would be seen between the two particles which can lead to a non-physical behaviour.

When a lower speed of sound c_0 is used, i.e. when the fluid is considered more compressible, a single precision is enough to distinguish the pressure difference between the particles.

3.2.6 Unity of the smoothing function

One of the properties of the smoothing function is that it must be normalised. This property is expressed by equation (2.17). This property is not always respected, especially in two cases:

1. when the particle is near a boundary or,
2. when the particle is near the free surface.

Those two possibilities are represented is figure 3.9. It can be seen that there is a lack of particles in some regions of the smoothing function. This can lead to the loss of the unity of the kernel.

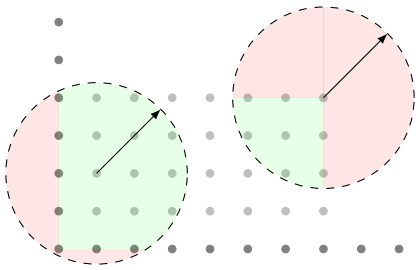


Figure 3.9: Examples of truncated smoothing functions

In order to remedy to this, the kernel or its gradient can be renormalised. In this master thesis, the formulae used to solve the Navier-Stokes equations use only the gradient. Thus, I will only focus on the kernel gradient. The normalisation of the kernel itself is quite easy and can be achieved thanks to the following expression:

$$\tilde{W}_{ab} = \frac{W_{ab}}{\sum_{b=1}^{N} \frac{m_b}{\rho_b} W_{ab}} \tag{3.44}$$

As the $\nabla_a W_{ab}$ intervenes in (3.16) and (3.22), the normalisation will be done differently. The developments are explained in [Chen et al., 1999]. First, let us focus on a 1-D case. The Taylor series expansion for a function f about x' can be written:

$$f(x) = f(x') + \frac{1}{1!}\frac{df(x')}{dx}(x - x') + \frac{1}{2!}\frac{d^2 f(x')}{dx^2}(x - x')^2 + \cdots \tag{3.45}$$

This equation can be multiplied by $W' = dW(x - x', h)/dx$ to both sides and integrated. The second order or higher derivative of f are ignored:

$$\int_\Omega f(x)W' \, dx \approx f(x') \int_\Omega W' \, dx + \frac{df(x')}{dx} \int_\Omega (x - x')W' \, dx \tag{3.46}$$

Equation (3.46) can be manipulated and written as

$$\frac{df(x')}{dx} \approx \frac{\int_\Omega (f(x) - f(x'))W' \, dx}{\int_\Omega (x - x')W' \, dx} \tag{3.47}$$

The particle approximation (2.12) allows me to write

$$\frac{df(x_a)}{dx} \approx \frac{\sum_{b=1}^{N} \frac{m_b}{\rho_b}(f(x_b) - f(x_a))\frac{dW_{ab}}{dx}}{\sum_{b=1}^{N} \frac{m_b}{\rho_b}(x_b - x_a)\frac{dW_{ab}}{dx}} \tag{3.48}$$

From equation (3.48), we can deduce an expression for the corrected kernel first derivative:

$$\frac{d\tilde{W}_{ab}}{dx} = \frac{\frac{dW_{ab}}{dx}}{\sum_{b=1}^{N} \frac{m_b}{\rho_b}(x_b - x_a)\frac{dW_{ab}}{dx}} \tag{3.49}$$

For a 3-D case, the problem can be generalised using the same reasoning. Neglecting the second order and higher derivatives, the Taylor series expansion in 3-D can be written as

$$\begin{aligned} f(\mathbf{x}) \approx f(\mathbf{x}') &+ (x_1 - x_1')\frac{\partial f(\mathbf{x}')}{\partial x_1} \\ &+ (x_2 - x_2')\frac{\partial f(\mathbf{x}')}{\partial x_2} \\ &+ (x_3 - x_3')\frac{\partial f(\mathbf{x}')}{\partial x_3} \end{aligned} \tag{3.50}$$

Equation (3.50) can be abbreviated by

$$f(\mathbf{x}) \approx f(\mathbf{x}') + (x_i - x_i')\frac{\partial f(\mathbf{x}')}{\partial x_i} \tag{3.51}$$

When both sides of equation (3.51) are multiplied by $\partial W(\mathbf{x} - \mathbf{x}', h)/\partial x_j$ and integrated, it becomes

$$\int_\Omega f(\mathbf{x})\frac{\partial W}{\partial x_j}\, d\Omega \approx f(\mathbf{x}')\int_\Omega \frac{\partial W}{\partial x_j}\, d\Omega + \frac{\partial f(\mathbf{x}')}{\partial x_i}\int_\Omega (x_i - x_i')\frac{\partial W}{\partial x_j}\, d\Omega \tag{3.52}$$

This equation can also be written as

$$\frac{\partial f(\mathbf{x}')}{\partial x_i} \approx \frac{\int_\Omega (f(\mathbf{x}) - f(\mathbf{x}'))\frac{\partial W}{\partial x_j}\, d\Omega}{\int_\Omega (x_i - x_i')\frac{\partial W}{\partial x_j}\, d\Omega} \tag{3.53}$$

After the particle approximation, equation (3.53) is written as

$$\frac{\partial f(\mathbf{x}_a)}{\partial x_i} \approx \frac{\sum_{b=1}^{N} \frac{m_b}{\rho_b}(f(\mathbf{x}_b) - f(\mathbf{x}_a))\frac{\partial W_{ab}}{\partial x_j}}{\sum_{b=1}^{N} \frac{m_b}{\rho_b}(x_{b,i} - x_{a,i})\frac{\partial W_{ab}}{\partial x_j}} \tag{3.54}$$

The corrected gradient of the kernel can be written as

$$\tilde{\nabla}_a W_{ab} = \mathbf{M}^{-1} \nabla_a W_{ab} \tag{3.55}$$

with the elements of the matrix $\mathbf{M}$ expressed by

$$M_{ij} = \sum_{b=1}^{N} \frac{m_b}{\rho_b} (x_{b,i} - x_{a,i}) \frac{\partial W_{ab}}{\partial x_j} \tag{3.56}$$

The derivative of the smoothing function is also expressed as in equation (2.16):

$$\frac{\partial W_{ab}}{\partial x_j} = \frac{x_{a,j} - x_{b,j}}{r_{ab}} \left. \frac{dW}{dr} \right|_{r=r_{ab}} \tag{3.57}$$

It can be seen from equation (3.55) that a system must be solved in order to normalise the gradient. I have noticed however that the matrix $\mathbf{M}$ is symmetric. In fact, this is due to expression (3.57) where $(x_{a,j} - x_{b,j})$ appears. This leads to a potential time saving during the resolution.

The normalisation has a big drawback. As explained in [Vaughan et al., 2008], the normalisation leads to the loss of symmetry. Indeed, the gradients of two neighbouring particles which are not at the same distance of a boundary or free surface will have their gradient corrected differently. This means that Newton's third principle will be violated: the force produced by a particle a toward a particle b will not be the same as the force produced by the particle b toward the particle a as it should be. The violation of this basic principle induces a small loss of interest for the normalisation.

If the correction of the kernel gradient is applied, it should be applied only on the conservation of momentum as stated by [Gomez-Gesteira et al., 2010]. In fact, the loss of symmetry applied to the continuity equation may lead to severe oscillations of the density.

To correct the continuity equation, other methods exist but are not implemented for this master thesis. A review of these is available in [Gomez-Gesteira et al., 2012b]. They are named *density filters*. The most straightforward mean to filter the densities is to perform every ~ 30 iterations the following filter:

$$\tilde{\rho}_a = \sum_{b=1}^{N} m_b \tilde{W}_{ab} \tag{3.58}$$

This filter is based on the summation density (3.12) and the corrected kernel equation (3.44). This technique avoids large density variations in the fluid.

3.2.7 Boundary conditions

The boundaries of a domain can be modelled in different ways. For this work, only one was chosen. This choice will be discussed later. [Crespo et al., 2007] cites three kinds of boundary conditions:

1. **Ghost particles**. This kind of boundary condition was introduced by [Randles and Libersky, 1996]. It consists of three kinds of particles:

   (a) *Interior particles* which are the fluid particles.

   (b) *Boundary particles* which represents the boundary itself.

   (c) *Ghost particles* which are used to complete the kernel.

   These particles are represented in figure 3.10. This kind of boundary condition can be implemented easily for regular surfaces but it is much more complicated when special boundaries have to be used.

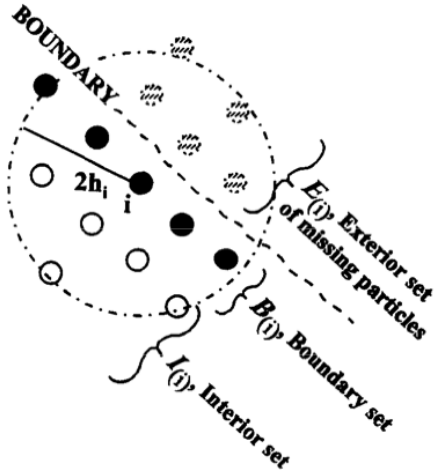


Figure 3.10: Interior, boundary and ghost particles [Randles and Libersky, 1996]

   The principle of this method can be summarised as follows: when a particle approaches the boundary, a ghost particle is created to the other side of the boundary with the same density but with an opposite velocity which creates a repulsive force. The variable number of particles is not easy to handle.

2. **Repulsive particles** were introduced by [Monaghan, 1994] in order to create boundary conditions. These particles exert a repulsive central force on a fluid particle. This repulsive force is based on the Lennard-Jones equation.

3. **Dynamic particles** are well described and analysed by [Crespo et al., 2007]. These particles have the same properties than the fluid particles except that the conservation of momentum is not solved for them. It means that $D\mathbf{u}/Dt$ is set by the user to 0 (the particles keep their initial velocity) or to an arbitrary law. This is very useful for moving boundaries. This kind of boundary condition is very flexible for the shape of the boundaries er for the motion of the boundaries. It is also easy to treat the boundary conditions as the dynamic particles can be included in the same loop as the the fluid particles.

For this master thesis, the *dynamic particles* have been chosen for their flexibility and their ease of use. These particles offer a repulsive mechanism thanks to the continuity equation (3.16). When a mobile (fluid) particle approaches a fixed (boundary) particle and when the distance between the particles is lower than κh, the terms of equation (3.16) are not null. This leads directly to an increase in density as $D\rho/Dt \neq 0$. This density increase induces a non null pressure. This non null pressure will at its turn lead to a $D\mathbf{u}/Dt \neq 0$ for the fluid particle. This can be seen in equation (3.22). From this description of the phenomenon, it can be deduced that it is the term $\mathbf{u}_{ab} = \mathbf{u}_a - \mathbf{u}_b$ together with the distance between the particles (via the gradient of the smoothing function) that play the biggest roles in the repulsive effect.

The use of dynamic particles as boundary can create a gap between the fluid particles and the boundary. This phenomenon can be observed when a thin layer of fluid flows paralleled to the boundary. This gap is approximately equal to the distance κh.

The way to initialise the boundary particles is an important factor. The most straightforward way to set the particles is to set them in a Cartesian lattice with the same spacing as the fluid particles. The density assigned to each particle is deduced from the hydrostatic pressure if the boundary touches the fluid. Only one layer of particles can be used. However, in some cases, some precautions must be taken. For example, when violent shocks between the particles and the boundary are possible, a second layer of particles can be used. This was done by [Crespo et al., 2007] to simulate the fall of a single particle on a boundary. The particles should be staggered as shown in figure 3.11 for a 2 or 3-D problem.

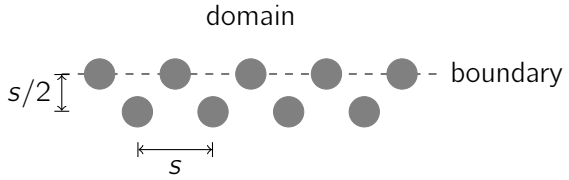


Figure 3.11: Two layers of boundary particles

3.2.8 Initialisation of the particles

The initialisation of the particles is an important step in the simulation of a problem. For this master thesis, it has been decided to initialise the position in a Cartesian grid. Some authors decide to initialise the particles in a staggered way. As it will be seen later, this decision has no impact on the implementation of the SPH program because the input files are created separately.

After having decided the position given to the particles, the other parameters such as the velocity, the density and the mass must be set. The velocity is an initial condition which will be set to 0 in many test cases of this work. The pressure and the mass are dependant from each other.

If we consider an initial hydrostatic pressure in the fluid, the density of each particle can be determined thanks to equation (3.38) or equation (3.39). Considering what is written in

[Violeau, 2012], the mass should be set according to the the volume occupied by a particle and its density:

$$m_a = V_a \rho_a \tag{3.59}$$

with a volume V_a calculated according to the spacing between the particles: $V_a = s^d$ with s the spacing between the particles and d the number of dimensions.

However, this kind of initialisation do not lead to the equilibrium of water in a tank as exposed in the section 5.1.2. In order to have a perfect equilibrium when setting the particles in a Cartesian grid with an initial hydrostatic pressure, it has been decided to analyse the problem in 1-D. This discussion is done in the appendix A.5. In this appendix, I found a solution only when the number of particles was even. This question is very interesting and further research should be done.

Finally, the technique used during this work to initialise the particles is the one proposed by [Violeau, 2012], which does not lead to an initial equilibrium. However, the SPH method is mainly used for dynamic problems. For example, for a dam break, the initial hydrostatic pressure is only used to initiate the movement of water. The pressure inside the fluid comes quickly to be more uniformly distributed. This is why the imprecision introduced at the beginning of the simulation can be considered as negligible for the range of problems that suits best to the SPH method.

3.2.9 Time integration

The time steps can be deduced from the Courant condition, the forces acting on the particles and the viscous forces. [Monaghan, 1992] gives the expression for the time step. The first expression given is relative to the body forces:

$$\Delta t_f = \min_a \left(\frac{h_a}{|\mathbf{F}_a|} \right) \tag{3.60}$$

with $a = 1, \cdots, N$, N being the total number of particles. The second expression is relative to the Courant condition as well as the viscous diffusion:

$$\Delta t_{cv} = \min_a \left(\frac{h_a}{c_a + 0.6(\alpha c_a + \beta \max_b \mu_{ab})} \right) \tag{3.61}$$

where the coefficients α and β are the same as the ones in the artificial viscosity (3.40). c_a is the speed of sound at the particle a.

The final time step in obtained by

$$\Delta t_{qi} = \min(0.25\Delta t_f; 0.4\Delta t_{cv}) \tag{3.62}$$

The coefficients 0.25 and 0.4 are given by [Monaghan, 1992] from computer experiments.

For an ideal gas, the time step can be increased as the fluid is more compressible. After a few experiments, the time step for the ideal gases was taken at $\Delta t_{ig} = 5\Delta t_{qi}$.

The time integration scheme used for this work was a Runge-Kutta second order predictor-corrector scheme, abbreviated RK22. Before developing this integration scheme, let us concentrate on the Eulerian scheme. It can be written:

$$y_{i+1} = y_i + hf(x_i, y_i) \tag{3.63}$$

with $x_{i+1} = x_i + h$ and $f(x_i, y_i) = y_i'$. This scheme is the most straightforward way to calculate y_{i+1}. This method is illustrated in figure 3.12 (a). However, the results obtained are quite bad as the error is $\mathcal{O}h$ [Kreyszig, 2006].

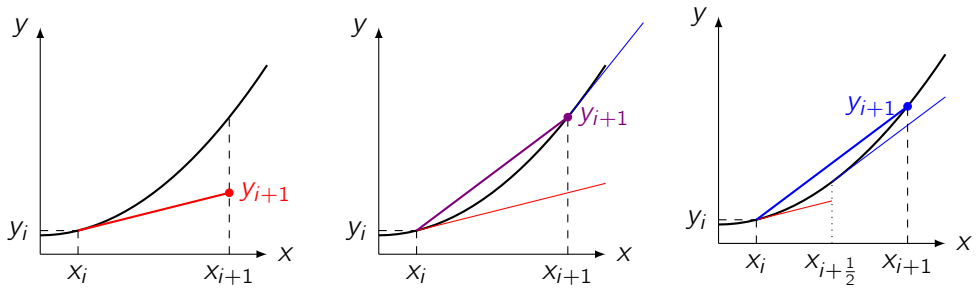


Figure 3.12: Integration schemes: (a) Euler, (b) RK22 – $\theta = 0.5$ and (c) RK22 – $\theta = 1$

The RK22 schemes are second order methods. This means that the error is $\mathcal{O}h^2$ [Kreyszig, 2006]. A first version is the Heun (or improved Euler) method. It can be obtained thanks to a development given in [Jedrzejewski, 2005]:

$$\begin{cases} y_{i+1} = & y_i + \frac{1}{2}h\,(k_1 + k_2) \\ k_1 = & f(x_i, y_i) \\ k_2 = & f(x_i + h, y_i + hk_1) \end{cases} \tag{3.64}$$

The Heun method can be visualised in figure 3.12 (b). The slope in this method is obtained thanks to a mean of the slope at the beginning of the step and at the end of the step. Equations (3.64) can be generalised:

$$\begin{cases} y_{i+1} = & y_i + h\,((1-\theta)k_1 + \theta k_2) \\ k_1 = & f(x_i, y_i) \\ k_2 = & f(x_i + \frac{1}{2\theta}h, y_i + \frac{1}{2\theta}hk_1) \end{cases} \tag{3.65}$$

The explanations concerning this generalisation can be found in [Jedrzejewski, 2005]. The Heun method is obtained when θ is replaced by 0.5.

Another version of the RK22 is obtained by substituting $\theta = 1$ in equations (3.65). This gives:

$$\begin{cases} y_{i+1} = & y_i + hk_2 \\ k_1 = & f(x_i, y_i) \\ k_2 = & f(x_i + \frac{1}{2}h, y_i + \frac{1}{2}hk_1) \end{cases} \tag{3.66}$$

This method is represented in figure 3.12 (c). It can be understood as if a slope was calculated at the middle of the step and applied at the beginning of the step. The error is the same between the two variants of RK22. It can be also noted that other values of θ can be used.

For this master thesis, the two RK22 schemes will be used and the three integration schemes will be compared.

Other integration schemes exist like the Leap-Frog scheme or higher order Runge-Kutta schemes (e.g. RK44). However, because of the high computational cost inherent to the SPH method and the didactic purpose of this work, the RK22 schemes will be chosen for its good compromise between the computational cost and the precision.

Chapter 4

Implementation of the method

This chapter will focus on the practical implementation of an SPH program. First of all, I will discuss the main scheme of the program, regardless of the implementation choices that can be made. Then, I will explain the different choices made to implement this program. I will concentrate on two implementations and I will discuss the reason why I chose one and not another.

A copy of the source code is available in the appendix B.4.

4.1 General scheme of the program

I will begin by the presentation of the general scheme of the program. A graphical representation is shown in figure 4.1.

The first step consists in initialising the system. This step includes:

1. The creation of the boundaries thanks to *dynamic particles*. These particles will be named *fixed particles* (FP) for the following pages. They are named fixed because they are mainly used as non moving boundaries. However, this type of boundary conditions can be used to model mobile boundaries. The data needed to initialise a boundary particle are:

   (a) its position $\mathbf{x}$ (3 components in 3-D),

   (b) its initial velocity $\mathbf{u}$ (3 components in 3-D)[1],

   (c) its mass and,

   (d) its density.

   This means that for one boundary particle, 8 data should be given to the program. The density and the mass are needed as an initial pressure may exist in the fluid, which induces a density $\rho \neq \rho_0$.

[1] The velocity is only useful when the boundaries are moving. In order to be as general as possible these three components are included in the initialisation.

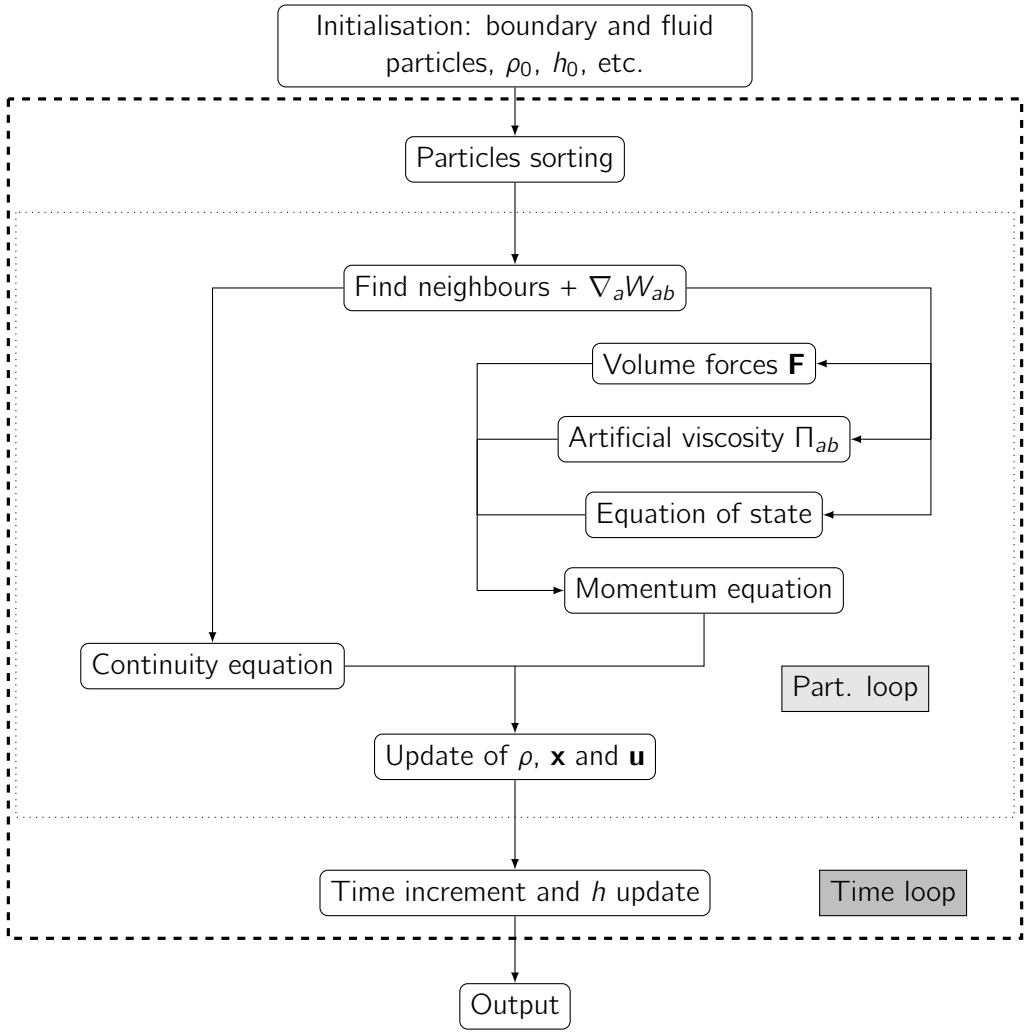


Figure 4.1: General scheme of an SPH program

2. The creation of the fluid particles is similar to the creation of the FP. The fluid particles will be named *mobile particles* (MP) for the following pages. These MP need the same data as the FP. However, the fixed and mobile particles should be distinguished as the conservation of momentum is not solved for the FP.

3. The initialisation of some parameters. These parameters include:

   - the kind of smoothing function to use,
   - the simulation time,
   - the saving interval,
   - etc.

The parameters given to the implemented program will be detailed later.

This initialisation can be done in different ways: the program itself can create the set of data that is needed, or the program reads some input files. The second option seems more flexible and was chosen for this master thesis.

After the initialisation, the program can enter into a time loop. In figure 4.1, the predictor and corrector steps are not represented for the sake of clarity. The first thing to do at the beginning of a time step is to sort the particles. Indeed, the search algorithm chosen is the linked-list search algorithm. The first step of this algorithm is to sort the particles in the different cells of the domain. Then, the neighbours can be searched. The step called *particle sorting* in figure 4.1 refers to the operation that calculates the cell in which a particle is. This operation is done only once every time step, not at every predictor or corrector step. Practically speaking, this operation requires a loop over all the particles.

After this sorting, a loop over all the particles can begin. For every particle, the neighbours ($r_{ab} \leq \kappa h$) must be found. This is done thanks to the summarised algorithm 1. The variable $Ncells$ refers to the number of cells that have to be scanned to find the neighbours. For the particular case presented in figure 3.8, $Ncells = 9$.

Algorithm 1 Algorithm used to find the neighbours of a particle

```
for i = 1 → Ncells do ▷ Ncells is the number of cells where neighbours can be found
    for j = 1 → Nparts do ▷ Nparts is the number of particles in the cell i
        calculate r_ab ▷ a refers to the current particle and b to the tested one
        if r_ab ≤ κh then
            add neighbour ▷ the particle b will be known as a neighbour of a
        end if
    end for
end for
```

When the neighbours of a particle are found, the values of the vector $\nabla_a W_{ab}$ can be calculated for every particle. This is done thanks to formula (2.16). The values of $\nabla_a W_{ab}$ can be saved in order to use them for the continuity and the conservation of momentum. If a corrected kernel is required, its calculation can take place now.

After these preliminary steps, the calculations to solve the continuity equation (3.16) and the conservation of momentum (3.22) can begin. These calculations can be performed in parallel because they are independent from each other at a given predictor or corrector step.

The continuity equation is solved for all the particles (FP and MP). In order to solve it for a given particle a, a loop over the neighbours b is performed as shown in the algorithm 2.

The conservation of momentum can be solved in a similar way for the MP: the sum over the neighbours can be done thanks to a loop over them. Moreover, some other terms must be computed: the artificial viscosity Π_{ab} and the pressures p_a and p_b. The pressures can be saved for each particle in order to save some time. The pressure is given thanks to the equation of state (3.24) or (3.25). Indeed, the pressure at a given particle will be needed

Algorithm 2 Calculation of the continuity equation

$sum \leftarrow 0$ ▷ $sum = D\rho_a/Dt$
for $b = 1 \rightarrow Nneigh$ **do**
  $\mathbf{u}_{ab} \leftarrow \mathbf{u}_a - \mathbf{u}_b$ ▷ a stands for the current part. and b for the current neigh.
  $sum \leftarrow sum + m_b \mathbf{u}_{ab} \nabla_a W_{ab}$
end for

more than once as it is part of more than one support domain. The artificial viscosity can be obtained thanks to equation (3.40). The body forces **F** can be calculated separately but can also be assumed constant with time which is the case for the gravitational force. The conservation of momentum is not solved for the fixed particles. Indeed, these particles are not part of the fluid and their movement is ruled by an arbitrary law (no movement or imposed movement).

After having calculated $D\rho/Dt$ from the continuity equation and $D\mathbf{u}/Dt$ from the conservation of momentum, the variables ρ, **u** and **x** of a particle can be updated. If the time integration scheme is an Eulerian scheme, the data are updated this way:

$$\rho_{a,\text{next}} = \rho_a + \frac{D\rho_a}{Dt}\Delta t \tag{4.1}$$

$$\mathbf{u}_{a,\text{next}} = \mathbf{u}_a + \frac{D\mathbf{u}_a}{Dt}\Delta t \tag{4.2}$$

$$\mathbf{x}_{a,\text{next}} = \mathbf{x}_a + \mathbf{u}_a \Delta t \tag{4.3}$$

The pressure and the speed of sound of each particle can be updated and saved after having obtained $\rho_{a,\text{next}}$.

After having updated the variables ρ, **u** and **x** of each particle, we can go to the next time step. The time step can be updated as well as the smoothing length. These operations are performed thanks to equations (3.62) and (3.42), respectively. The new smoothing length is the same for every particle as the choice of a constant smoothing length in space has been made.

Finally, after all the required time steps, the output can be generated. This output can include the following information:

- the positions **x**, velocities **u**, densities ρ and/or pressures p of the MP
- the same data for the FP
- the saving times
- the performance information of the program

These data can be treated in two different ways:

1. directly by the program that performs the calculation: after or during the calculation
2. by another program that will use the output files generated by the main program

For this work, the second option was chosen because it offers more flexibility. Moreover, the first option slows down the calculation. The programming language used to make the calculations may not be suitable for graphical representations. The second option allows the user to plot the information he desires while the first one is more complicated to adjust.

4.2 Practical implementation of the SPH method

The previous section explained the main scheme of an SPH program. I will now focus on the way I implemented the program that will be used. I will also discuss an alternative implementation that has not been used because of performance problems. These problems will be explained.

4.2.1 First choices

Object-oriented programming

Before any implementation, a choice of programming philosophy must be done. In many situations a *sequential* philosophy is chosen because it is straightforward and it offers a lot of advantages for many applications. However, for this SPH code I chose to design the program around an *object-oriented* philosophy. This way to implement is not usual in a classical civil engineering curriculum but I had the opportunity to attend a course of object-oriented programming some years ago. This slight knowledge of the subject made me think that it could be interesting to try this kind of implementation.

Before going further, I would like to describe the object-oriented programming [Budd, 2002]. This kind of programming exists from the 1960's. However, it became truly popular in the 1980's. Nowadays, a lot of programmers use this paradigm. Budd describes this paradigm as being a new way of viewing the world. Indeed, an object-oriented program can be seen as a community of agents. [Budd, 2002] states that:

> "An object-oriented program is structured as a *community* of interacting agents called *objects*. Each object has a role to play. Each object provides a service or preforms an action that is used by other members of the community."

The different objects of a program can communicate between them thanks to messages.

From a general point of view, an object contains two things:

1. Variables such as numbers, strings, arrays, etc.
2. Methods that are some mechanisms used to perform actions.

In the object-oriented paradigm, the variables and methods can have different scopes. But this possibility will not be used in this work.

An *object* is an *instance* of a *class*. The class is a kind of mould. It is used to create the objects. This mould (class) can also be modified in order to create other objects that

have the same kind of properties but with other features. This is called the inheritance and hierarchy in object-oriented programming.

The object-oriented paradigm includes other mechanisms that will not be discussed here.

Choice of a programming language

In order to implement a program with the object-oriented paradigm, a language must be chosen. For example, `Java` and `C++` are well known for their object-oriented features. These languages will not be used because `Java` is not directly processed by the computer. It must use a virtual machine in order to run on a device. This is an advantage for programs which must be easily portable. However, it is a big drawback for programs that require high performance. `C++` was not a solution either because I preferred to concentrate my attention on more fundamental problems linked to the SPH method instead of learning a new language.

Given these observations `Fortran` seemed to be the best solution. Indeed, this language offers some object-oriented features and is quicker to learn given my programming background. `Fortran` is also a high-level programming language that allows very good performance. It was first designed by IBM in the 1950's for scientific and engineering problems. It is still mainly used today in that field. `Fortran` may appear as an old-fashion language to many people but it has been reviewed many times in the last years. The last standard in called `Fortran 2008` and was approved in 2010 [Chivers and Sleightholme, 2012]. The last updates of `Fortran` brought some object-oriented features (since `Fortran 90` and more with `Fortran 2003` and `Fortran 2008`) and bindings with the `C` language (`Fortran 2003`).

As discussed earlier, the pre and post-processing can be handled in two different ways:

1. the particles are created and post-processed by the main program or,
2. the particles are created and post-processed by a secondary program implemented in a more convenient language.

The second solution has been chosen for its flexibility. The pre and post-processing will be done thanks to two `Matlab` programs. Three files will be needed for the input and two files will be produced for the output.

The input files are:

1. A parameter file. This is a text file with an extension `*.prm`. It contains 15 lines with the following information:
   - number of fixed particles
   - number of mobile particles
   - initial smoothing length h_0
   - reference speed of sound c_0
   - reference density ρ_0

- size of the domain. The whole domain is considered cubic. This data is the length of one side. Even if the whole domain is cubic, the boundaries can be set in order to use only a part of the domain and thus the final domain can be of any geometry.
- kind of kernel
- artificial viscosity coefficient α
- artificial viscosity coefficient β
- equation of state
- value of γ in the equation of state for quasi-incompressible fluids
- molar mass in the equation of state for ideal gases
- kernel correction
- simulation time
- saving interval

The values to give to these properties will be given later.

2. A file which contains the information of the fixed particles. It is also a text file with an extension `*.fp`. A line represents a particle. For a line, there are 8 data:
   - columns 1 to 3: initial position **x**
   - columns 4 to 6: initial velocity **u**
   - column 7: initial density of the particle
   - column 8: mass of the particle

   The columns are separated with a tabulation.
3. A file which contains the information of the mobile particles. It is also a text file with an extension `*.mp`. Its structure is the same as for the fixed particles.

In order to localise these files, the main program requires a file named `paths.txt` placed in the same directory as the main program. This file contains the paths toward the files `*.prm`, `*.fp` and `*.mp`. This technique allows to save the working files in a different directory than the main program's directory. This is more convenient than handling the files manually from one directory to another.

The output files are:

1. `results.out` which contains
   - the 3 components of the position **x**
   - the 3 components of the velocity **u**
   - the density ρ
   - the pressure p

these data are saved for every mobile particle, every time a saving is required.

2. `time.out` which contains the saving times

The two output files are binary files that contain single precision real numbers. The choice of binary files has been made in order to increase the efficiency of saving and of reading.

To sum up, it has been decided to implement an SPH model in `Fortran` in an object-oriented way. This program will use 3 input files and output 2 binary files for post-processing.

4.2.2 The implemented program

Now that the main choices are explained, I will focus on the practical implementation of the SPH model.

As stated earlier, I chose to write the program in an object-oriented way. Thus I have to decide the classes to implement. In the `Fortran` language the classes are called types. A basic diagram of the program is sketched in figure 4.2. This diagram will help to create the types needed. The central entity of the program is what is called the particle manager. This manager knows all the particles (fixed and mobile). It knows also the characteristics of the simulation. This manager can communicate with an object called `particle_sort`. This object is used by the particle manager to sort the particles in the different cells of the domain as discussed earlier. The role of this *sorting machine* is based on the explanations of figure 3.8. A link between the particles and the particle manager is necessary. This can be achieved thanks to the use of an array. Each cell of the array contains a pointer toward an object particle (MP or FP). The use of an array of pointer instead of an array of objects was done because of the limited possibilities of `Fortran` for this particular case[2].

The diagram of figure 4.2 gives a first idea of how the different objects of the program interact with each other. I will now explain what is contained in each type.

Particle manager

Let us begin with the particle manager. It contains a lot of information relative to the the problem. For example: the number of particles (total, FP and MP), the kind of kernel, the simulation time, the equation of state to use, etc. Beside all these information, it is linked to the `particle_sort` object and to a list that contains pointers toward the particle objects. This list will be explained later. The particle manager also contains four procedures:

- `initialisation` which initialises all the problem. It is the first procedure called. Its job is to load all the data from the input files and to create and prepare the different objects used by the program.

[2]As it will be seen later, the mobile and fixed particles use polymorphism which does not make the use of an array of objects possible.

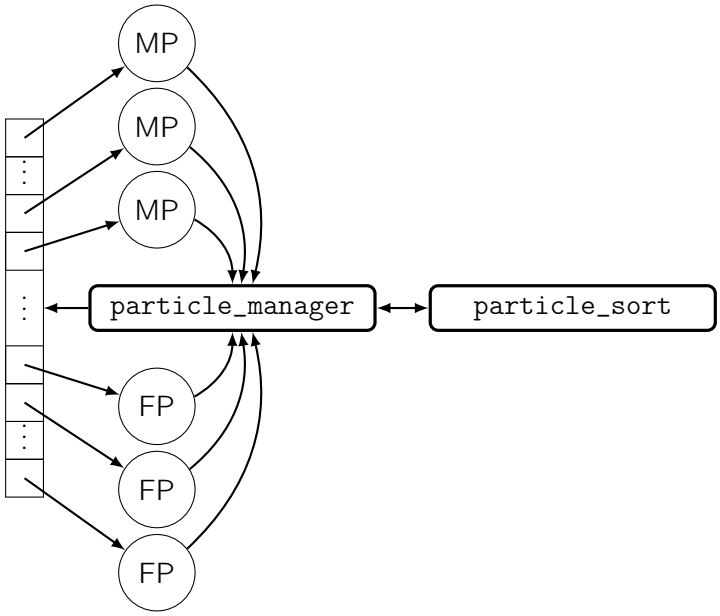


Figure 4.2: Diagram of the program

- `solver` is used to solve the problem. It is the main procedure that will call other procedures from the object `particle_manager` or from other objects. Its behaviour will be described later.
- `timeStepUpdate` is used to update the time step according to equation (3.62).
- `slUpdate` is used to update the smoothing length according to equation (3.42).

Sorting of the particles

Then, let us focus on the object `particle_sort`. This object is present only once in the program. Its role is to know in which cell a particle is. It communicates with the particles via the particle manager. Its variables are, the maximum smoothing length, the length of a side of a cell, the number of cells on a side and the total number, a pointer toward the particle manager and a 2-D list that stores the particles in each cell. A minimal example of this list is given in figure 4.3. It can be seen that an array with a size equal to the number of cells is created. This array contains a list of particles. This list has not a constant size in time. The mechanism used to handle this variable size will be explained later. The construction of this array of lists allows to know which particles are in which cell. This will be very useful for the search of neighbours of a particle. Indeed, as a particle knows in which cell it is located, it also knows the surrounding cells. This allows a particle to know its potential neighbours. Its last job will be only to check the distances between it and the potential neighbours.

In order to build the lists of particles, the object `particle_sort` needs a few procedures:

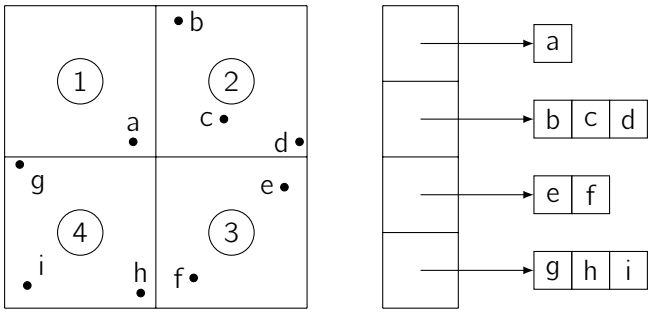


Figure 4.3: Lists of sorted particles

- `get_h_max` calculates the maximum smoothing length. This procedure is only useful if the smoothing lengths are not constant in space. This has been implemented in order to be as general as possible. After having found the maximum h, the result is multiplied by a coefficient ≥ 1 in order to keep the same number of cells during all the computation time. This coefficient is chosen according to the equation of state. If the fluid is an ideal gas, it will be more compressible and the smoothing length will tend to vary more. The calculations made to determine these coefficients are given in the appendix A.4. The results are given in table 4.1.

- `setCells` is used to set the size of the cells. The cells are cubic and the domain is cubic. This means that there is the same number of cells on every side. The procedure allocates also the array of lists of figure 4.3.

- `particlesSort` is the main procedure and its job is to sort the particles in the different cells. The lists presented in figure 4.3 are created or updated. Firstly, some modulo operations are done in order to know in which cell a particle is located. Then, it is added to the corresponding list.

	Ideal gas	**Quasi-incompressible fluid**
$\tilde{h}_{\max} =$	$1.1 h_{\max}$	$1.02 h_{\max}$

Table 4.1: Coefficients used to increase the maximum smoothing length

The sorting operation is not performed at every predictor-corrector step. It is done only at the beginning of a time step.

Particles

The last objects are the mobile and fixed particles. As these objects have many similar properties, a system of inheritance has been implemented. The main class is the `fixed_particle` type. The `mobile_particle` type extends the `fixed_particle` type because it has exactly the same properties as the fixed particles except that the conservation of momentum must be solved. The type `fixed_particle` has mainly the following

variables: position, velocity, density, pressure, speed of sound, smoothing length, list of neighbours, values of the kernel gradient for every neighbour, etc. All theses variables are common with the mobile particles. The procedures of a fixed particle are:

1. `getNeighbours` is used to find the neighbours. The procedure scans the cell in which the particle is located as well as the neighbouring cells. It calculates the distance between two particles and then add the particle as a neighbour if the distance $\leq \kappa h$.
2. `calcPressure` is a function that calculates the pressure according to the equation of state chosen.
3. `calcCelerity` is a function that calculates the speed of sound according to the equation of state chosen.
4. `gradW` is a procedure that loops over all the neighbours in order to calculate the gradient of the kernel at a considered neighbour. The values of the gradients are saved in a matrix $(3 \times Nneigh)$.
5. `kernelCorr` is the routine used to correct the kernel gradient as described in the section 3.2.6.
6. `varUpdate_fixed` is the procedure that performs the calculations required by the equation of continuity. The results are saved in the variables of the concerned object according to the current Runge-Kutta step.

As said earlier, the `mobile_particle` type is an extension of the `fixed_particle` type. The only two differences between these two types are:

1. `ArtificialViscosity` is a new procedure that calculates the Π_{ab} terms.
2. `varUpdate_mobile` is an overload procedure that solves the equations of continuity and of conservation of momentum. It updates also the variables according to the current Runge-Kutta step.

Lists in the program

Lists of particles are needed at several places in the code. Two kinds of lists can be distinguished:

1. Lists with a fixed size
2. Lists with a variable number of elements

The first kind of list can be observed in the particle manager. This list has a constant number of particles as the number of particles in the system is supposed to stay constant. This kind of list can be easily handled with an array. However, the arrays in `Fortran` do not allow to have polymorphic types as elements of the array. This has been handled thanks to the use of a pointing objects which were called `min_link` (to abraviate *minimal link*). This

type contains only a variable which can point toward an object of class `fixed_particle`. As the type `mobile_particle` extends the type `fixed_particle`, the mobile particles can be pointed by an object `min_link`. The declaration of `min_link` in `Fortran` is

```
type min_link
   class (fixed_particle) , pointer :: ptr => null()
end type min_link
```

This pointing object will be used to point toward a created particle object (MP or FP) and will be stored in the array. The creation of the objects MP or FP is done during the initialisation of the system. The particles must be declared at this time as a FP or a MP:

```
allocate(fixed_particle :: this%part(i)%ptr)
```

or

```
allocate(mobile_particle :: this%part(i)%ptr)
```

respectively. These expressions allow the minimal link to know if it is pointing a FP or a MP. Thanks to this trick, an array of polymorphic objects can be created.

The second kind of list has a variable number of elements. This could be handled thanks to linked-list for example. This implementation will be developed in the next subsection. Lists of variable length can be found for the lists of neighbours of a particle or the lists of particles contained in a cell of the domain. The choice made for this type of list relies on the arrays and the possibility to allocate and deallocate them. Let us imagine a first kind of array which has a variable length every time an element is added or removed. The array in this case must be allocated and deallocated every time. This leads to a lot of manipulations in the memory of the computer which can lead to a longer computation. Another way would be allocate an array only once at the beginning with a very large size in order to be sure to keep the same size during all the computation. Even if this solution is very efficient concerning the execution time, it has a big drawback: the memory used.

The advantages of the two solutions described can be implemented in what I will call a *dynamic list*. A dynamic list is a type which is composed of an array and some auxiliary variables. These variables are:

- the total size of the array,
- the number of elements in the array and,
- an incremental value used to increase the size of the array.

A UML diagram of this type is given in figure 4.4. In order to handle such a list, three procedures are needed:

1. `initList` is used to initialise the dynamic list

2. `addElement` is used to add an element to the list. If the array is full, the elements of the array must be saved in a temporary array. The previous array is deallocated and a new array is allocated with the previous number of elements plus the increment. This is described by the algorithm 3.

3. `resetList` is used to reset the list. The size of the array is unchanged but the number of elements in it is set to zero. When new elements will be added to the list, the previous elements (not erased) will be overwritten.

It must be noted that the elements of the implemented list are of type `link`. These elements are the same as `min_link` except that they have one more value which describes the distance between two particles. This new variable is useful for the list of neighbours

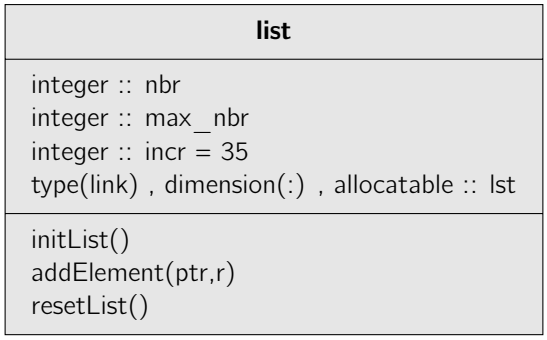


Figure 4.4: UML diagram of a dynamic list

Algorithm 3 Adding an element to a dynamic list

Require: ptr ▷ pointer toward a particle
Require: r ▷ distance between two particles
  if $maxnbr = 0$ **then** ▷ the list is empty
    call initList()
  else if $nbr = maxnbr$ **then** ▷ the list is full
    allocate(temp_lst(1:$maxnbr + incr$)) ▷ a temp. list is allocated with a bigger size
    temp_lst(1:$maxnbr$) = lst(1:$maxnbr$) ▷ the elements are transferred
    call move_alloc(temp_lst,lst)
    $maxnbr \leftarrow maxnbr + incr$
  end if
  $nbr \leftarrow nbr + 1$
  $lst(nbr).ptr \leftarrow ptr$ ▷ the pointer of an element of the list points toward the ptr
  $lst(nbr).r \leftarrow r$

In this work, the increment was set at 35. This was done because the dynamic list was used at two places: as a list of neighbours and as a list of particles in a cell. These two lists have not the same length at all. The first one is estimated to have a number of

particles around 50, but in areas of very high density, the number of neighbours can reach 65-70. This means that two increments are enough to reach this number without wasting too many space in the memory of the computer. For the lists of every cells the number can be of approximately 300 for full cells and 0 for empty cells. This means that the increment chosen will not cause too many reallocations. The increment of 35 seemed to fit well the problem.

There is a final detail I would like to mention. It concerns the elements of the dynamic list. For the lists of neighbours, the elements need to know who is the neighbour and it can also save the distance between the current particle and the neighbouring particle. This avoids to calculate this distance again when the value of the kernel gradient must be determined. This is why it is necessary to have an element `link` with two variables: a pointer and a real number for the distance. For the list of particles in a cell, the elements of the list do not need to have an information of distance. However, the type `list` could not be implemented with two kinds of elements or even polymorphic elements. This is due to `Fortran`. This is why a dynamic list is implemented with only one kind of elements: `link`. This leads to a very small loss of memory space. In fact, the unused variable r is a real in double precision (8 bytes). The number of particles of the domain can reach 150000 in ordinary situations. This means that the lists of all the cells will have in total a bit more than 150000 elements allocated in memory. This means that a little bit more than 1200000 bytes (1.14 MB) will be allocated but not used. This amount of memory is not a problem for actual computers (RAM $\geq$ 4 GB). The implementation of a single type of list is justified for this reason.

Parallelism

Current computers have more than one core. This property can be used in order to reduce the calculation time. Indeed, if two or more processors are used to compute the same calculation as on a single processor, the execution time will be decreased as the different calculations are spread over the processors. This technique cannot always be used.

For the implemented program, more than one processor can be used. The parallelism can be used when the procedures `varUpdate` are launched. In fact, at that moment, the particles are independent from each other, they do not share a same variable that could be changed during the calculation. Parallelising this part of the program can be useful.

The parallelising can be done thanks to the library *OpenMP* (Open message passing). This is done very easily in the implemented program. Indeed, only two lines have to be added around a loop that can be be parallelised. These two lines are:

```
!$OMP PARALLEL DO PRIVATE(i) SCHEDULE(DYNAMIC)
...
!$OMP END PARALLEL DO
```

These instructions are placed around the loop over the particles for updating ρ, $\mathbf{x}$ and $\mathbf{u}$.

4.2.3 An alternative program

In order to have better performances, another implementation has been tried. This new implementation concentrates mainly on linked-list and on another way to update the lists.

The linked-list implemented is composed of elements that are called `link`. The linked-list must know which element is the first one, which is the last one and the number of elements in the list. If this list has to be updated by a number of new elements, it is interesting to know which element is the first added and how many elements have been added. The procedures used by a linked list must include an initialisation procedure, a procedure to add elements to the list, another to remove elements and a last one to reset the list. The type linked-list is given in figure 4.5. The type `link` is an element of a linked list. Its structure is given in figure 4.6. This type being an element of a list, it must have two pointers in order to know which element is the previous one and which element is the next one.

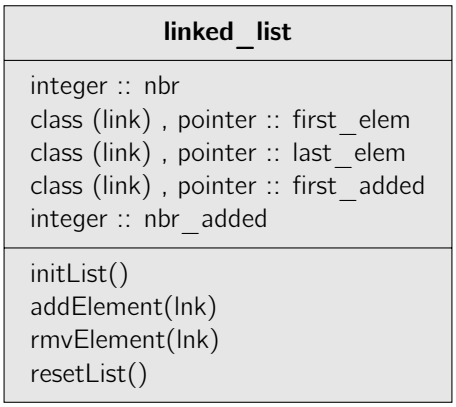


Figure 4.5: UML diagram of a linked-list

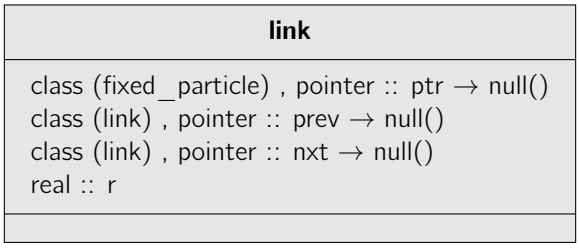


Figure 4.6: UML diagram of a linked-list element

This linked-list can be used to store the particles of a cell or to store the neighbours of a particle.

For the storage of the particles in a cell, two procedures are used: one for initialisation and another for the update. The initialisation sorts for the first time the particles in the different cells. After this first sorting, the loops over the particles will be done directly

via the linked-lists of the different cells. After a time step, the linked-list of every cell is updated, i.e. the linked-list are not reset but only updated with the particles that changed of cell. The procedure used to update the linked-list of every cell is the algorithm 4.

Algorithm 4 Update of the linked-list of the sorting cells

```
for i = 1 → Ncells do
    for j = 1 → NpartPerCell do
        if particle changed of cell then
            delete particle from old list
            add particle to new list
        end if
    end for
end for
```

In this alternative implementation, each particle has two lists: one with the neighbours and another with the potential neighbours. The potential neighbours are particles that are located in the neighbouring cells but are too far to be neighbours. This new kind of implementation is useful to update the the list of neighbours at each time step. The initialisation of these lists is done thanks to the procedure `initNeighbours`. This procedure is summarised by the algorithm 5.

Algorithm 5 Initialisation of the neighbours and potential neighbours lists

```
reset pot_neighbours and neighbours lists
for i = 1 → NsuroundingCells do
    for j = 1 → NpartPerCell do
        calculation of the distance r
        if r < κh then
            add the particle to the list of neighbours
        else
            add the particle to the list of pot_neighbours
        end if
    end for
end for
```

After a whole time step, the list of neighbours is updated. The procedure that do this is called `updateNeighbours` and an overview of its structure is the algorithm 6. This procedure scans the lists of neighbours and potential neighbours. The distance is recalculated and the lists updated according to this distance. The particles added to the neighbouring cells are also scanned and the distances calculated. According to this distance, the new particle is added to the list of neighbours or the list of potential neighbours.

This new implementation seems very efficient on an algorithmic point of view as the changes in the lists are minimised. However, for a given problem, this alternative program was approximately 25% slower than the previous implementation. This can be explained thanks to the way `Fortran` handles the pointers. A pointer is a memory address of a given

Algorithm 6 Update of the lists of neighbours and potential neighbours

```
if the particle is in the same cell then
    for i = 1 → Nneigh do
        calculation of the distance r
        if r > κh then
            delete from the list of neighbours
            add to the list of pot_neighbours
        end if
    end for
    for i = 1 → NpotNeigh do
        calculation of the distance r
        if r < κh then
            delete from the list of pot_neighbours
            add to the list of neighbours
        end if
    end for
    for i = 1 → Nadded do                ▷ loop over the part. added to neigh. cells
        calculation of the distance r
        if r < κh then
            add to the list of neighbours
        else
            add to the list of pot_neighbours
        end if
    end for
else
    call initNeighbours()
end if
```

real, integer or type for example. `Fortran` uses a lot of memory space in order to store this address. It can reach up to 72 bytes as stated by [Kumbera]:

> "At run time, there is no additional overhead used by Cray pointers. Each pointer uses only the number of bits necessary to hold a memory address (typically 4 or 8 bytes on current machines). In contrast, on the 32-bit compilers available at the Laboratory, Fortran 90 pointers use up to 72 bytes of memory each – 18 times the size of a Cray pointer!"

For comparison, the `C` language needs 4 or 8 bytes only to store a pointer. As pointers are intensively used in the new implementation, the size of the `Fortran` pointers can be the cause of a slower code.

In order to take advantage of the new implementation, it could be interesting to use *self-made* pointers which are more efficient than the one offered by `Fortran`. Another solution could be to investigate the possibilities of the C-binding offered by the `Fortran` 2003 standard.

4.3 How to run the program

The implementation of the program being explained, I will concentrate on how to run the program. The user must only focus on the input and the output files. These files have already been presented in a previous section. I will discuss now the different possible parameters.

4.3.1 The input files

There are 3 input files:

- one for the mobile particles and one or the fixed particles. The different particles must be initialised according to what was explained in the chapter 3.
- one parameter file. This file takes 15 values. These values and the possible options are given and explained in table 4.2.

Parameters	**Possible values**	**Explanations**
Number of fixed part.	D.O.F[3]	
Number of mobile part.	D.O.F	
h_0	U.C[4]	initial smoothing length
c_0	U.C.	reference speed of sound
ρ_0	U.C.	reference density
Size of the domain	U.C.	cubic domain
Kind of kernel	1	cubic spline
	2	quadratic kernel
	3	quintic spline
α	e.g. $[0.01; 0.5]$	coefficient of bulk viscosity
β	often 0	used in Π_{ab} to handle shocks
Equation of state	1	ideal gas law
	2	quasi-incompressible fluid
γ	e.g. 7	exponent of the eqn state 2
Molar mass	U.C.	used for the ideal gas law
Kernel correction	0	no correction
	1	correction of the gradient
Simulation time	U.C.	time is seconds
Saving interval	U.C	time in seconds

Table 4.2: Description of the options of the parameter file

The paths of the input files must be mentioned in an auxiliary file named `paths.txt`. After all these files created, the program can be launched. At every saving, the program will

[3] D.O.F. = depends on other files
[4] U.C. = user choice

show a message that indicates the number of iterations done, the time of the simulation when the saving was performed as well as the time step. These information allow the user to estimate the remaining time.

4.3.2 The output files

The output files are binary files. This type of file has been chosen in order to increase the efficiency to access the saved data. For the file `time.out` the saving times are saved as single precision real numbers. It is easy to extract these times as a vector.

Concerning the file `results.out` the data are saved as single precision real numbers. The order in which the MP are saved is given in table 4.3. Knowing this sequence of saving, an exterior program can extract these data easily.

$x_1(p_1, t_1)$	$\cdots$	$u_3(p_1, t_1)$	$\rho(p_1, t_1)$	$m(p_1, t_1)$	$x_1(p_2, t_1)$	$\cdots$	$x_1(p_1, t_2)$	$\cdots$

Table 4.3: Saving of the mobile particles state

The graphs that can be generated by an exterior program are varied:

- film of the movement of the particles
- evolution of the pressure
- evolution of the position of a single particle
- evolution of the velocities
- etc.

Some possibilities will be illustrated in the next chapter.

4.3.3 Compilation

The implemented program has been written in order to respect the `Fortran` standard. This makes the program compilable by any `Fortran` compiler on *any platform*. This is a considerable advantage for a didactic program.

The compilers tested were the GNU compiler for `Fortran`, gfortran, as well as the Intel compiler. After a few tests, gfortran seemed to offer better results than the Intel compiler. The calculation was, in the same conditions, approximately 40% quicker with gfortran. This may seem to be a lot of difference between a commercial and a free compiler. A reason that was found for this difference concerns the way Intel handles a pointer that points another pointer.

All the tests made in the next chapter use gfortran.

Chapter 5

Testing the program

In this chapter, the program implemented will be tested and validated thanks to some test cases. The limitations will also be highlighted. After the validation, some comparisons between different parameters will be done. Finally, the possibilities of the method will be exposed thanks to some specific situations.

All the tests were made on the same computer, fitted with an Intel i7 920 processor with a frequency of 2.66 GHz. This processor is composed of 4 cores but 8 threads. All the indicated times are real times and not CPU times. They were calculated in the same conditions of use.

5.1 Test cases to validate the program

In order to validate the program, five test cases will be done. Some limitations will be highlighted and ideas of remedy will be proposed.

5.1.1 Particles fall

The first test case consists in letting a cubic set of particles fall. The initial geometry of this problem is given in figure 5.1. The dimensions are given from particle to particle.

The goal of this test case is to check if the particles fall according to the uniformly accelerated (UA) law. In this particular case, the position is given by

$$z = z_0 - g\frac{t^2}{2} \tag{5.1}$$

with $g = 9.81$ m/s^2, z_0 the initial z position of a particle and t the elapsed time. The final aim of this test case is to verify the integration scheme used in the implementation.

The test has been done for an ideal gas as well as for a quasi-incompressible fluid. The characteristics of this test case are given in table 5.1.

A particle fall is represented in figure 5.2 (a) for an ideal gas and in figure 5.2 (b) for a quasi-incompressible fluid. The line is given for an arbitrary particle of the set. Only the

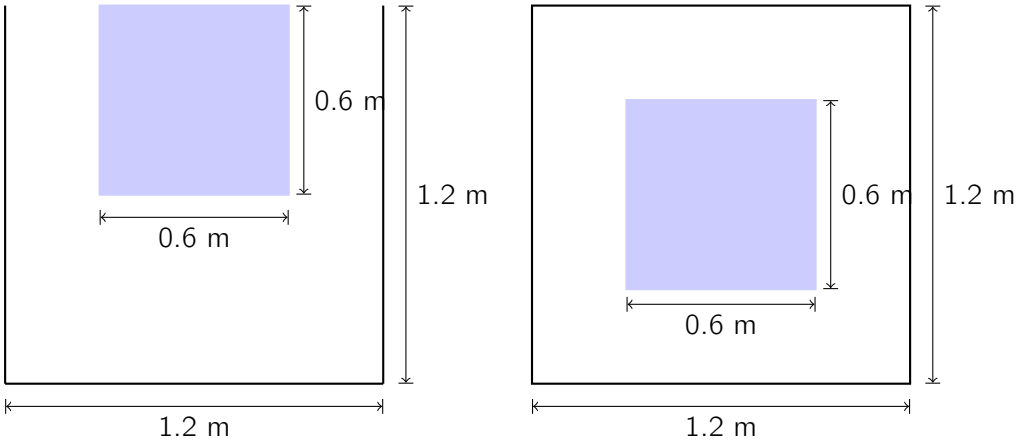


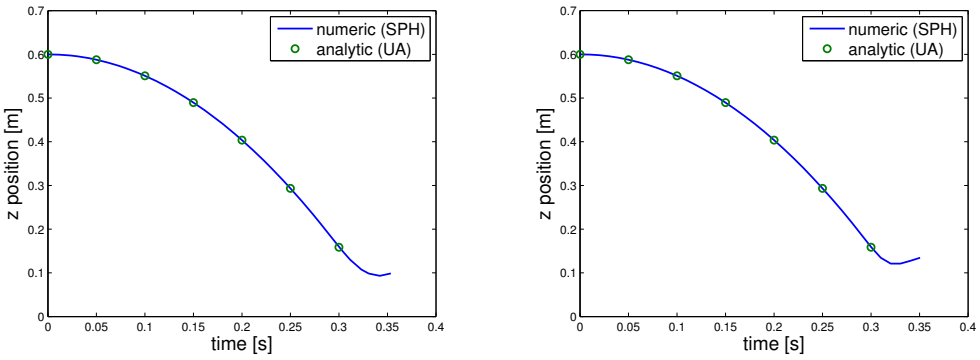
Figure 5.1: Initial geometry of the test case *particles fall*

Characteristics	**Ideal gas**	**QI fluid**
Nb FP	745	745
Nb MP	343	343
h_0 [m]	0.12	0.12
c_0 [m/s]	30	30
Smoothing function	Cubic spline	Cubic spline
α	0.5	0.5
Equation of state	Ideal gas	QI fluid
Kernel correction	no	no
Simulation time [s]	1.5	1.5
Calculation time	1 s	5 s

Table 5.1: Particles fall characteristics

interval of time during which the particles are falling is represented. It can be seen that the z position corresponds well to what is expected by the analytical solution (5.1). The analytical solution being a parabola, the numerical solution should fit perfectly as a RK22 scheme is used. This can be observed when a zoom is done. This observation validates the RK22 integration scheme used.

When the particles approach to the bottom boundary, they are repelled and a splashing occurs. This occurs differently according to the fluid used. Snapshots are taken at 4 different times for each fluid. They can be observed in figure 5.3. The color scale represents the pressures of every particle. It is different for each frame in order to see easily where there is a high pressure. A difference of behaviour can be observed between the two fluids, especially at 0.7 s. Indeed, a QI fluid being less compressible than an ideal gas, larger forces are produced when particles tend to go away from each other. This is why the particles of a QI fluid will tend to stay closer to the boundaries than the particles of an ideal gas.



(a) (b)

Figure 5.2: z position of a particle that falls: (a) ideal gas, (b) quasi-incompressible fluid

The evolution of the pressure and density of a given particle is given in figure 5.4 for an ideal gas as well as for a QI fluid. It can be observed that larger pressures are observed for a QI fluid than for an ideal gas. This is due to the equation of state used. For the densities, larger oscillations are observed for an ideal gas. This is because an ideal gas is more compressible than a QI fluid. At the end of the simulation, the fluid stabilises and the pressure, as well as the density, stabilises around a given value. It is also important to notice that there is no density fluctuation before the interaction between a fluid particle and a boundary particle. Indeed, $\mathbf{u}_{ab} = 0$ as all the particles of the set has the same velocity. This leads to have $D\rho/Dt = 0$ according to equation (3.16).

At the end of the simulation (at 1.5 s), the water in the tank is almost still. A plan view of this situation is given in figure 5.5. Four symmetry axes can be observed and are drawn in figure 5.5. The initial situation has the same symmetry axis at the beginning. Thus, it is normal to have a symmetric situation at the end of the simulation. This highlights the fact that the program solves the same equations according to x, y and z.

The elapsed times for this simulation are given in table 5.1. We can observe that these simulations are quite fast. For an ideal gas, the calculation time is even shorter than the simulation time. This can be obtained only because the number of particles is relatively low and because of a low speed of sound.

Two conclusions can be drawn after this first test case:

1. the integration scheme RK22 is correct and,

2. the symmetry of a problem is respected.

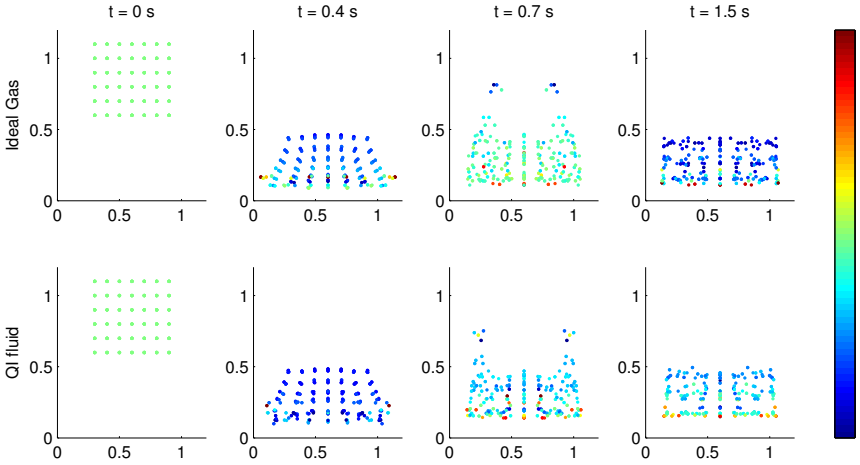


Figure 5.3: Comparison of the positions at different times

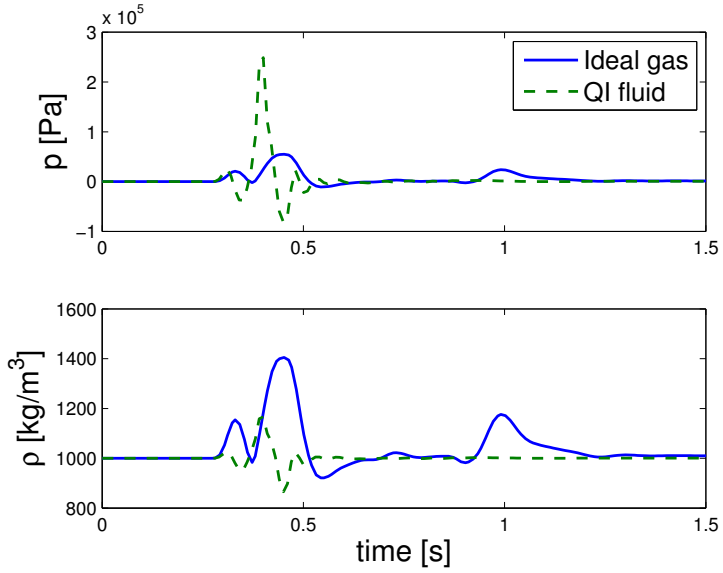


Figure 5.4: Comparison of the pressures and densities for a given particle

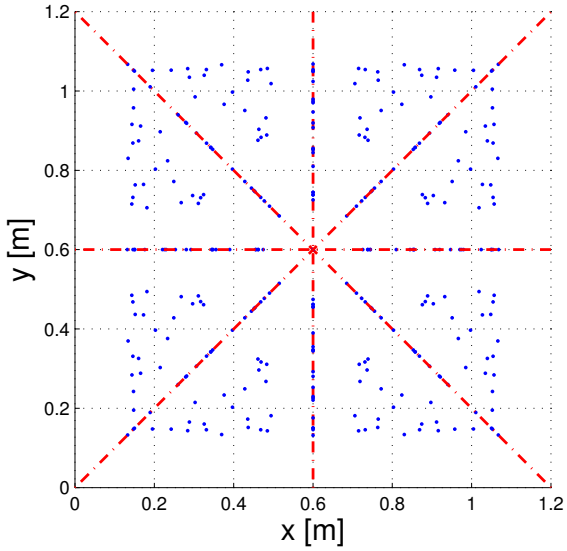


Figure 5.5: Symmetry axis at the end of the simulation

5.1.2 Water in a still tank

In this test case, an amount of water will be placed in a still tank. The aim of this test case is to check the equilibrium of the system. Multiple cases have been done but they are not all given in this section. Please refer to the appendix B.1 for further information.

In this section, I will just treat the case of an amount of water placed in a tank with a pressure equal to zero for every particle. The goal is to see if the hydrostatic pressure is the pressure that will be found for the particles in the tank. The initial geometry of the test is given in figure 5.6. The parameters used are given in table 5.2. The problem is simulated during only 5 s. After such a period of time, some conclusions can be made even if the system is not stabilised. A longer simulation would request too much calculation time.

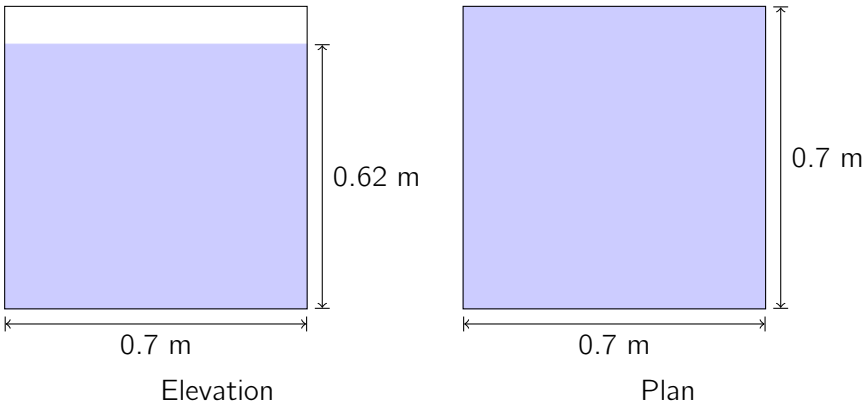


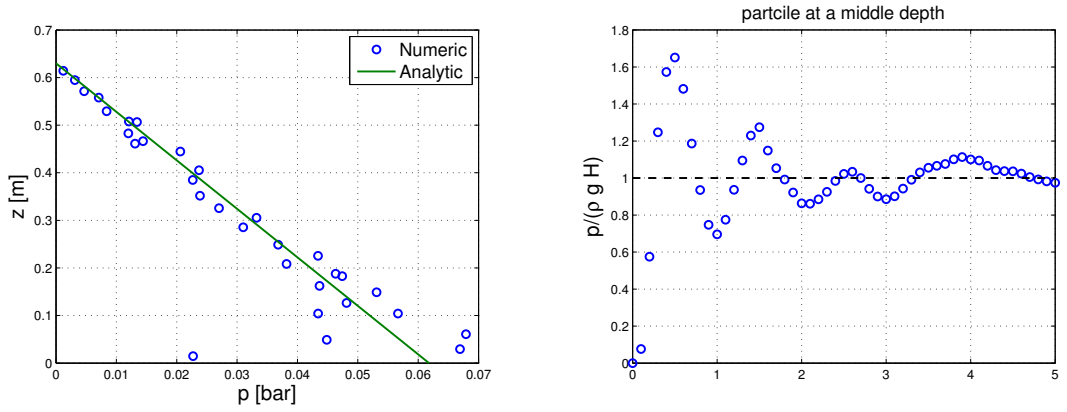
Figure 5.6: Initial geometry of the test case *water in a still tank*

Characteristics	**Test a**
Nb FP	6196
Nb MP	35836
h_0 [m]	0.024
c_0 [m/s]	30
Smoothing function	Cubic spline
α	0.05
Equation of state	QI fluid
Kernel correction	no
Simulation time [s]	5
Calculation time	75.8 min

Table 5.2: Characteristics of the test *water in a still tank*

The goal of this simulation is to show that the hydrostatic pressure is the solution of the problem. After 5 s, the distribution of the pressure is given in figure 5.7 (a). The particles concerned are the particles of a column of water at the center of the tank. It can be seen that the pressures are quite close too the analytical solution, especially for the

particles of the top. However, the results are very bad for the bottom of the tank. This is due to the fact that the kernel is not normalised at the bottom. In the appendix B.1, it is shown that adding more boundary particles has a positive effect. Figure 5.7 (b) shows that the pressure stabilises around the hydrostatic pressure $\rho g H$. After 5 s, the stabilisation is not completed but the oscillations are damped which suggests that the stabilisation will occur a few seconds after.



(a) (b)

Figure 5.7: (a) Pressure distribution ate the center of the tank after 5 s and (b) evolution of a central particle

In order to see how the pressure establishes on a slice of the tank, some snapshots were taken at different times and are shown in figure 5.8. It can be understood that the hydrostatic pressure establishes slowly (more than 2 s are needed). After this first general establishment, some oscillations occurs and need to be damped. As shown in figure 5.7 (b), more than 5 s are needed. Moreover, near the boundaries the pressure is not equal to the hydrostatic pressure. This is due to the fact that the kernel is not normalised there.

This test case shows that the hydrostatic pressure is a solution that is obtained a few seconds after the beginning of the simulation. However, some problems can be seen near the boundaries. As explained in the appendix A.5, an initial equilibrium state can be found with an initial hydrostatic pressure. However, this requires the resolution of a system of equations in order to find a particular distribution of masses. This solution was found only for an even number of particles.

Some other test cases are presented and discussed in the appendix B.1 in order to show the influence of different parameters on this very particular test case.

5.1.3 Dam break on a dry bed

This test case is performed in many articles such as [Crespo et al., 2007; Gomez-Gesteira et al., 2010, 2012a]. It is based on the experiments of [Koshizuka and Oka, 1996].The initial geometry of the problem is given in figure 5.9. It can be seen that the problem is a

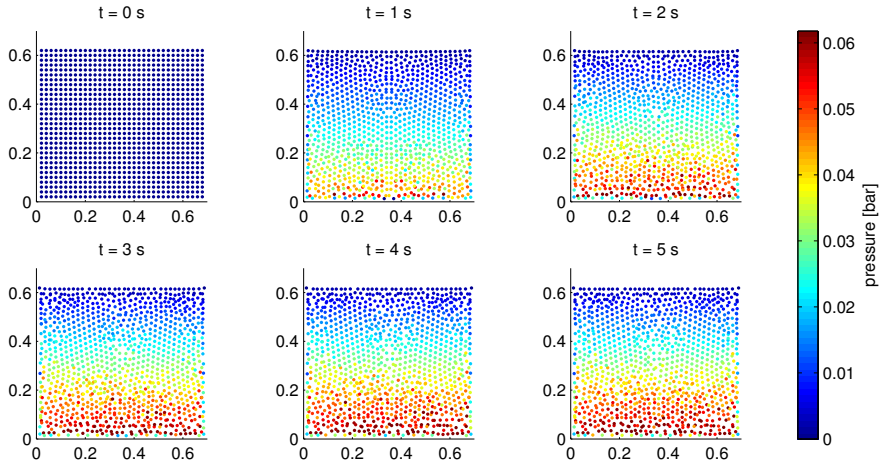


Figure 5.8: Snapshots ate different times of the establishment of the hydrostatic pressure

2-D problem. However, the code is implemented in 3-D which force me to make this test run in 3-D.

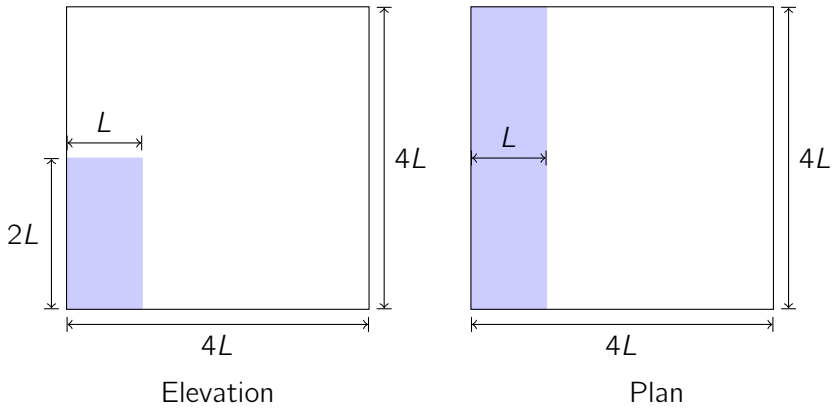


Figure 5.9: Initial geometry of the test case *dam break on a dry bed*

In this case, it has been decided to take $L = 0.5$ m. All the parameters of the test are given in table 5.3.

In order to check the good results given by the code, I will refer to [Koshizuka and Oka, 1996]. This article gives the position of the front measured experimentally[1]. My results are compared to the experiment in figure 5.10. This figure takes non dimensional values in abscissa and ordinate. Z is the position of the front compared to the left boundary. It can be seen that the numerical results follow quite well the experimental measures. The

[1]The experimental results are given in a graph in this article. No numerical values were available. The points represented in my figures are points that were measured on the article's graph. Some errors may be present.

Characteristics	Dam break on a dry bed
Nb FP	60402
Nb MP	123750
h_0 [m]	0.024
c_0 [m/s]	35
Smoothing function	Cubic spline
α	0.5
Equation of state	QI fluid
Kernel correction	no
Simulation time [s]	2
Calculation time	2.9 h

Table 5.3: Characteristics of the test *dam break on a dry bed*

artificial viscosity has been increased to 0.5 in order to take into account, as well as possible, the friction between the tank and the fluid.

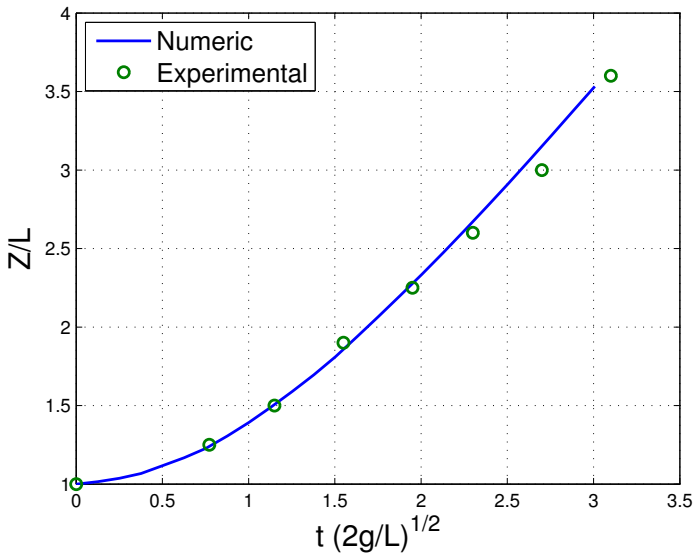


Figure 5.10: Position of the front for the dam break on a dry bed

Some snapshots taken from the numerical simulation are shown in figure 5.11. The colors represent the pressure in the fluid. The first observation that can be made concerns the pressure field. It is not smoothly distributed and oscillations around a given point can be observed. However, the general layout of the pressure field is good: high pressures are observed when the water hits the boundary and higher pressures can be seen for bigger depth. The oscillations in the fluid are due to the fact that the kernel gradient is not normalised. This can be handled thanks to a density filter which is not implemented in this

code. [Gomez-Gesteira et al., 2010] shows that the results obtained with a filter are not significantly better.

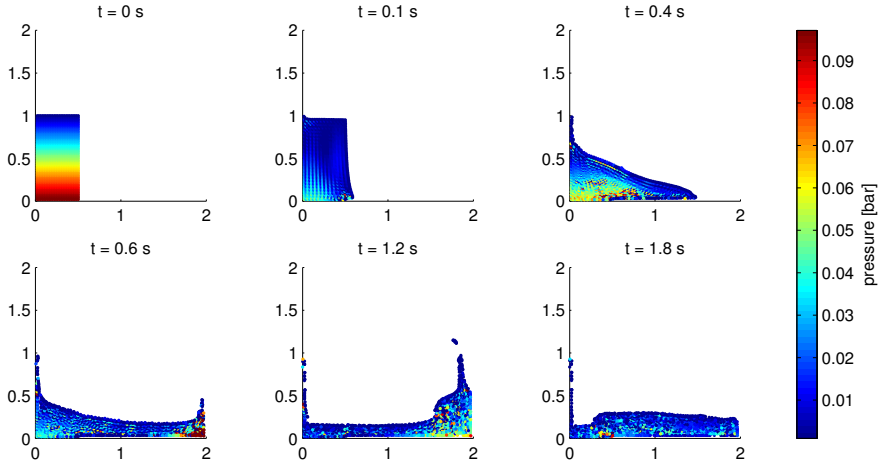


Figure 5.11: Snapshots of a dam break on a dry bed

Another observation that can be made is that some particles stay *glued* on the left boundary. This is an unwanted phenomenon. However, it can be explained quite easily. It is due the artificial viscosity introduced in the system. This viscosity creates a force that makes it more difficult for two particles to go away from each other. As the boundary particles do not move, some particles cannot overcome that attraction force with their own weight. In many articles which consider dynamic particles as boundary conditions, these particles are erased for the graphics. [Violeau, 2012] highlights also this phenomenon.

A small gap can also be observed between the fluid and the bottom boundary. This is due to the dynamic particles used as boundary conditions. This problem is documented in [Crespo et al., 2007] and explained in the chapter 3.

The last observation about these snapshots is about the shape of the fluid when it falls from the right boundary. [Koshizuka and Oka, 1996] give some snapshots of their experiment. One of these is given in figure 5.12. We can observe a similarity between the snapshot at 1.2 s of figure 5.11 and figure 5.12.

From the observations that have been made, it can be concluded that a dam break on a dry bed is well represented by the SPH program concerning the motion of the fluid. However, without a density filter, the pressures do not look so good even if the general pressure field layout is consistent. According to these observations, the results obtained by the SPH program can be concluded as satisfactory.

5.1.4 Dam break on a wet bed

Another test case that was done is a dam break on a wet bed. This test case is often used for testing many programs. An experiment was done by [Jánosi et al., 2004]. However,

Figure 5.12: Snapshot of a dam break on a dry bed taken from [Koshizuka and Oka, 1996]

the implemented program being in 3-D, the domain should also be a 3-D domain. The experiment needs a very long domain ($\sim$ 10 m) but very flat ($\sim$ 0.15 m). If this problem was implemented in the program, a lot of space would be lost as the implemented domain must be cubic. This experiment cannot be easily represented by the program of this work.

In order to simulate a dam break on a wet bed, the geometry will be adapted and the checking will be done on the celerity of the wave front. For an infinite domain, the analytical wave celerity is $U = c_1 = \sqrt{gh_1}$ for a rectangular section, h_1 being the bigger depth before the dam break. The domain is not infinite and this solution is not exact as some waves will reflect on the boundaries. However, this solution can be considered as a good start to check if the program is consistent.

The implemented domain is given in figure 5.13. The values of the parameters used are given in table 5.4. A low α was taken in order to avoid numeric instabilities and not to add too much viscosity. The execution time is not too long: 23.5 min.

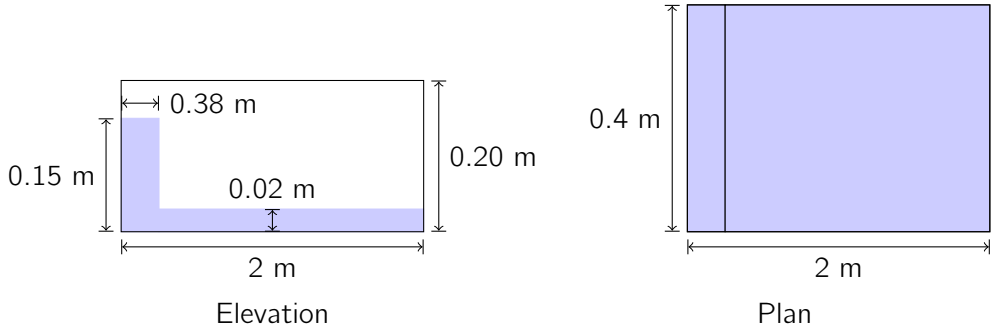


Figure 5.13: Initial geometry of the test case *dam break on a dry bed*

Some snapshots were taken at different times. They are given in figure 5.14[2]. The red dashed line represents the theoretical position of the wave front calculated thanks to the constant velocity $U = \sqrt{gh_1} = 1.21$ m/s. As said earlier, this position is not exact as:

[2]It must be noticed that the x and z scales are different.

Characteristics	Dam break on a wet bed
Nb FP	17841
Nb MP	34788
h_0 [m]	0.012
c_0 [m/s]	15
Smoothing function	Cubic spline
α	0.01
Equation of state	QI fluid
Kernel correction	no
Simulation time [s]	1.5
Calculation time	23.5 min

Table 5.4: Characteristics of the test *dam break on a wet bed*

1. waves may reflect on boundaries and,
2. h_1 is not constant because the domain is not infinite.

This red dashed line is an *approximation* of the wave position. However, it can be used as a good estimation of the position of wave front.

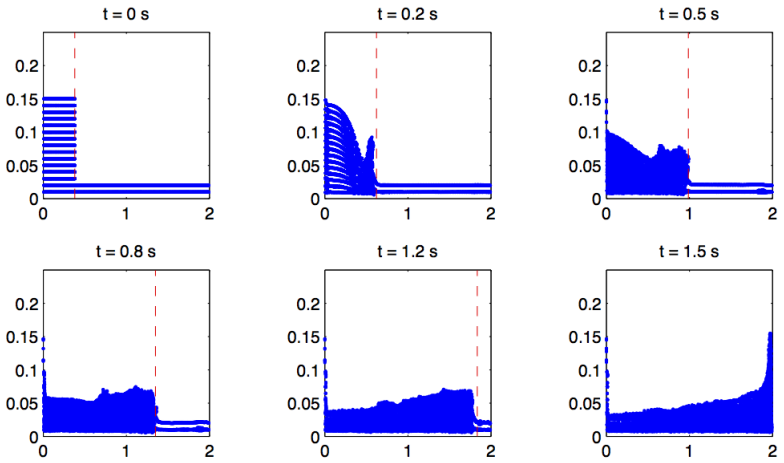


Figure 5.14: Snapshots of a dam break on a wet bed

As seen in figure 5.14, the theoretical position is very close to the wave front up to 1 s. After, the wave front calculated by the SPH method is not so close. This can be explained by the high decrease of h_1.

An interesting observation in figure 5.14 is the surface waves that can be seen especially at $t = 0.8$ and 1.2 s. These waves are due to sound waves that propagate in a fluid in open channel.

As it has been observed, the SPH program gives results quite close to what was expected by a theoretical approximation. This allows me to state that this test case is passed.

5.1.5 Spinning tank

The last test case used to validate the code is a cylindrical tank half-filled with water[3]. This tank is spinning around its axis at an angular velocity $\omega = 1.5$ rad/s. The dimensions and the initial configuration are given in figure 5.15. At the beginning of the simulation, the tank has an initial velocity which stays constant during all the time, while the water has no velocity with an horizontal free-surface.

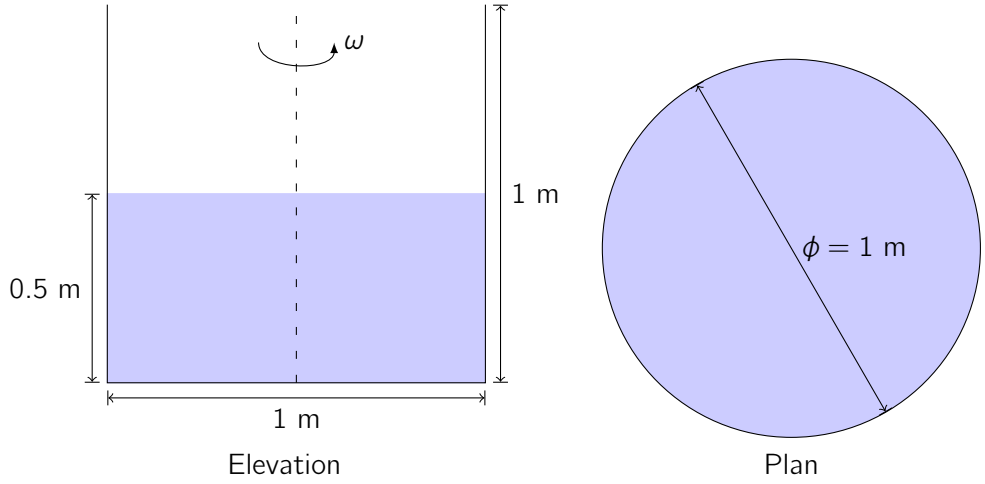


Figure 5.15: Initial geometry of the test case *spinning tank*

The parameters used for this test case are given in table 5.5. Two experiments were made: one with a low speed of sound as suggested by [Monaghan and Kos, 1999] and discussed earlier (test a) and another with the real speed of sound $c_0 = 1480$ m/s (test b). The first observation concerns the calculation time. The speed of sound is an important parameter is the calculation of the time step as it can be seen in equation (3.61). A bigger speed of sound will lead to a longer calculation time. This is obvious when table 5.5 is observed. When $c_0 = 1480$ m/s is used, more than 4.5 days are needed to simulate only 7 s.

The profile of the free-surface of a fluid in rotation is given by the following analytical solution:

$$z(r) = \frac{r^2\omega^2}{2g} + z_0 - \frac{\omega^2 R^2}{6g} \tag{5.2}$$

The development used to obtain equation (5.2) is given in the appendix A.6. The analytical profile will be compared to the numerical results. This is done in figure 5.16[4] for the two

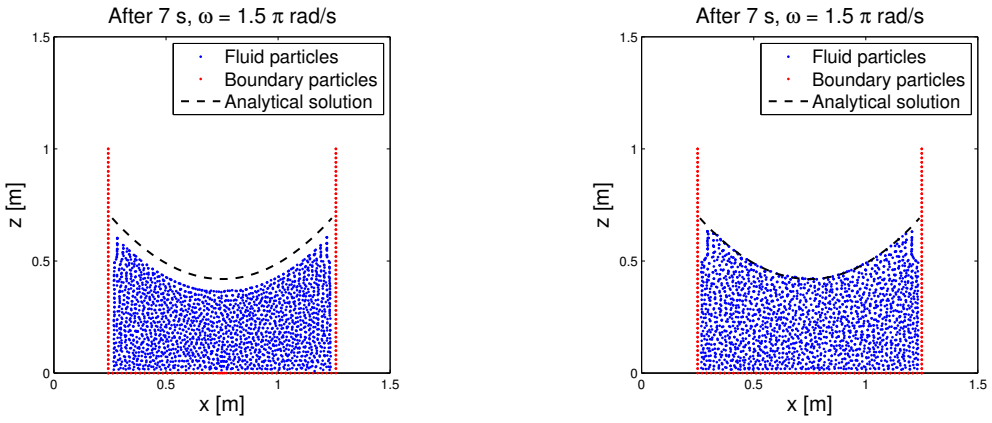
[3]Acknowledgement: this test case was suggested by Laurent Miny, a classroom mate.
[4]The figures represent a slice of the cylinder.

Characteristics	**Test a**	**Test b**
Nb FP	9894	9894
Nb MP	47175	47175
h_0 [m]	0.024	0.024
c_0 [m/s]	50	1480
Smoothing function	Cubic spline	Cubic spline
α	0.2	0.01
Equation of state	QI fluid	QI fluid
Kernel correction	no	no
Simulation time [s]	7	7
Calculation time	4.5 h	109.8 h

Table 5.5: Characteristics of the test *spinning tank*

speed of sound chosen. The most obvious observation is about the test a. Indeed, it is clear that the profile obtained numerically has a good shape but is not placed correctly. The numerical profile is under the analytical one. This is due to a too big compressibility due to the fact that the speed of sound was lowered. In comparison, the numerical solution fits perfectly the analytical one for a speed of sound that has a physical meaning (test b). This shows clearly that the assumption made by [Monaghan and Kos, 1999] is not always usable, especially when the compressibility of the fluid is an important parameter. The influence of the speed of sound on the compressibility was discussed in the chapter 3.

After this observation, it is important to recall the importance of the speed of sound on the calculation time. Some calculations would be impossible if $c_0 = 1480$ m/s was always used. This is why a lower speed of sound can be used in the situations where the compressibility is not as important as in this test case.



test a test b

Figure 5.16: Profile of the free surface for two c_0

Another observation that can be made in figure 5.16 concerns the gaps near the bound-

aries. These ones can be explained thanks to the repulsive forces created by the dynamic particles that make the boundaries. The fluid being less compressible for the test b, the forces exerted by the boundary particles are greater and the radial acceleration is not enough to overcome this repulsion force and get the fluid particles closer to the boundary. For the test a, the gap is much smaller because the repulsion forces are lower due to the greater compressibility of the fluid.

Given the previous visual observation, it is important to know if the system is stabilised. This analysis has been made in the appendix B.2. It comes that the system can be considered as stabilised.

The fluid particles are initialised without velocity and with an horizontal free surface. This does not represent an equilibrium state. The different steps of the system to stabilise are given in figure 5.17. We can observe a stabilised profile after 4 s. Form this figure, it can be understood that the fluid experience shear forces in order increase the velocity of the particles and to obtain a profile close to the analytical solution.

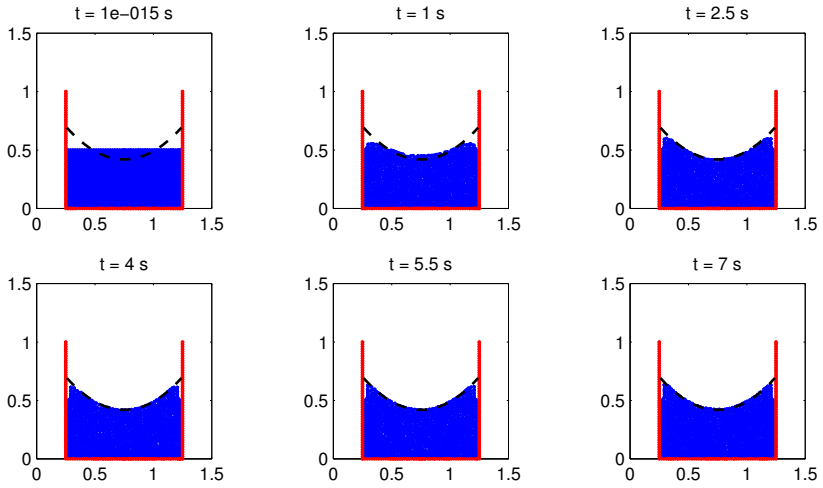


Figure 5.17: Snapshots at different times of the spinning tank

To sum up, this last test case was very interesting because:

1. It showed that the assumption that takes $c_0 = 10\sqrt{gH}$ is not always good to make. The real value of c_0 should be preferred in some cases.
2. The program is able to deal with moving boundaries, even if not natively implemented.
3. The program is able to reproduce the profile of the free surface of a fluid placed in a spinning tank. This suggests that the equation of conservation of momentum (see the appendix A.6 for the analytical solution and the governing equation) as well as the equation of continuity are solved correctly (update of the density and thus of the pressure which lead to a consistent pressure field in order to solve the equation of conservation of momentum).

5.1.6 Conclusions

The previous test cases allowed to validate the code. The following features were assessed and validated:

1. The integration scheme (RK22) allows to fit perfectly a particle fall. This validates the implementation of the integration scheme as it is a second order scheme which must fit perfectly a parabola.
2. The hydrostatic pressure is found for the particles in the middle of a tank after damping.
3. The position of the wave front of a dam break on a wet bed follows well the experimental results.
4. The propagation of a wave on a wet bed is close to the theoretical velocity of the propagation in an infinite domain.
5. The profile of the water contained in a spinning tank respects the theoretical profile.

However, a few limitations linked to the SPH method (but not the implementation of it) were highlighted. These limitations concern mainly the boundary conditions: a gap was observed for the dam break on a wet bed, some particles stay attached to the boundary particles and the fact that the kernel is not normalised in this region leads to large pressure oscillations (water in a still tank). Nevertheless, these limitations do not lead to the impossibility to use the SPH method, only to a few little errors near the boundaries.

According to the good results found in the five previous test cases, the program can be validated.

5.2 Comparison of some implemented options

In this section, I will compare the differences between some options (implemented or easily implementable). All the tests are based on the a same initial configuration which is the same dam break as the one given in figure 5.9. The differences between the results as well as the differences between the execution times will be discussed.

5.2.1 Comparison between the different smoothing functions

The first comparison concerns the 3 implemented smoothing functions: a cubic spline, a quadratic kernel and a quintic spline. Each are tested on the same test case. The parameters are given in table 5.6.

Before having a look ate the results, the first observation concerns the calculation time. A bar graph is given in figure 5.18. It can be easily understood that the quintic spline asks much more time for the calculation. This is due to the fact that its support domain is bigger ($\times 1.33$) than the support domains of the other smoothing functions. The small

Characteristics	**Test a**	**Test b**	**Test c**
Nb FP	9762	9762	9762
Nb MP	7800	7800	7800
h_0 [m]	0.06	0.06	0.06
c_0 [m/s]	35	35	35
Smoothing function	Cubic spline	Quadratic kernel	Quintic spline
α	0.5	0.5	0.5
Equation of state	QI fluid	QI fluid	QI fluid
Kernel correction	no	no	no
Simulation time [s]	1.5	1.5	1.5
Calculation time	3.4 min	3.4 min	8.5 min

Table 5.6: Characteristics of the tests used to compare the smoothing functions

difference between the cubic spline and the quadratic kernel can be explained by exterior factors such as a background task. The difference is not significant enough to draw any conclusion. A same calculation time seems legit as the number of neighbours of every particles should be the same (same support domains for every particle).

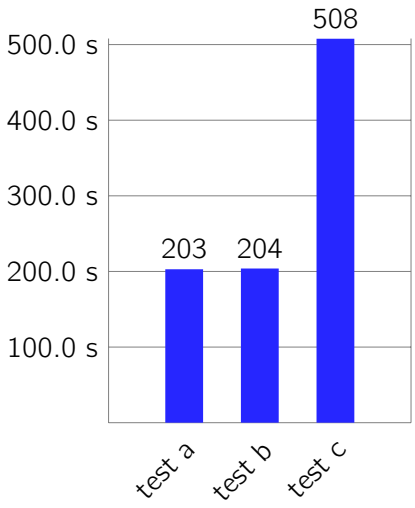


Figure 5.18: Calculation times for different smoothing functions

Let us focus now on the results given by the three chosen smoothing functions. A first comparison is given in figure 5.19. It shows the position of the wave front. This is the same graph as the one produced earlier for the dam break on a dry bed (see figure 5.10). It can be observed that there is nearly no difference between the three smoothing functions.

However, it can be noticed that the number of particles has a great impact on the solution. Indeed, when we look at figure 5.10, we see that the solution with more particles (previous test) was more accurate (closer to the experimental measures).

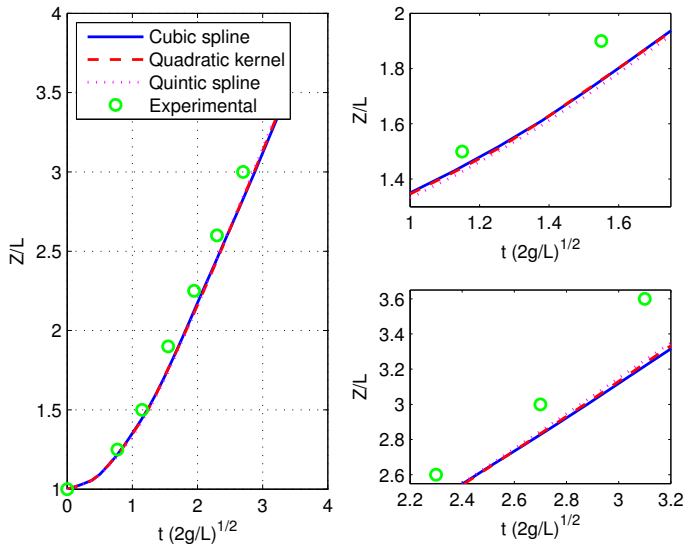


Figure 5.19: Comparison of the position of the wave front for three smoothing functions

A slice of the fluid has been captured at a given time for the three smoothing functions. These snapshots are given in figure 5.20. Nearly no differences can be observed. The only comment that can be made concerns the position of the particles. For the quintic spline, it appears that the particles in the middle of the fluid are better ordered than for the two other kernels. This is most probably due to the bigger support domain. However, this characteristic does not influence much the final result.

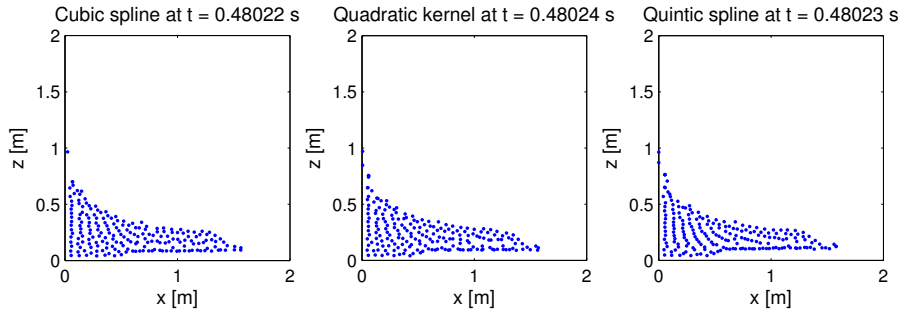


Figure 5.20: Comparison of a slice of fluid for three smoothing functions

To sum up, the three smoothing functions give very close results. Concerning the execution time, the cubic spline and the quadratic kernel are better than the quintic spline.

During my tests, I noticed that the cubic spline was more stable than the quadratic kernel. For these reasons, I recommend the use of the cubic spline.

5.2.2 Comparison between 3 integration schemes

In the chapter 3, three integration schemes have been explained. In this section, these schemes will be tested and compared thanks to a dam break on a dry bed. The geometry used is the same as the one given in figure 5.9. The parameters used for this comparison are given in table 5.7.

Characteristics	**Dam break on a dry bed**
Nb FP	9762
Nb MP	7800
h_0 [m]	0.06
c_0 [m/s]	35
Smoothing function	Cubic spline
α	0.5
Equation of state	QI fluid
Kernel correction	no
Simulation time [s]	1.5
Calculation time	see table 5.8

Table 5.7: Characteristics of the comparison between the integration schemes

Let us first begin with the execution time. The results are given in table 5.8. The code has been compiled with two compilers: gfortran (free) and Intel (commercial). First of all, the execution time is approximately 70% greater with an Intel compilation[5]. This is a huge difference in the advantage of a free compiler. This can be explained thanks to a bad handling of the pointers by the Intel compiler.

Then, it can be be noticed that the difference between an Euler and an RK22 scheme is not as big as it could be thought. Indeed, the search for neighbours is an expensive operation and it is performed only once per time step (at the beginning). For the correcting step (in the RK22 scheme), only the distances between neighbours are recalculated which is not as expensive as a complete search of neighbours. This is why there is not a factor 2 between the execution times of an Euler scheme and a RK22 scheme.

Concerning the results, the figure 5.21 shows the position of the wave front with respect to the time for the three integration schemes. It can be seen that they are very close. The Euler integration scheme is not as precise as the RK22 schemes. These two schemes offer exactly the same results.

A snapshot at a given time is given in figure 5.22 for the three schemes. No difference can be seen for the two RK22 schemes. However, there is a slight difference between the Euler and the RK22 schemes. Indeed, more particles stay attached to the left boundary for an Euler scheme.

[5]The same compiling options were chosen. OpenMP was turned on for both compilers.

	Euler	**RK22** $\theta = 1$	**RK22** $\theta = 0.5$
gfortran	168 s	204 s	203 s
Intel	299 s	335 s	-

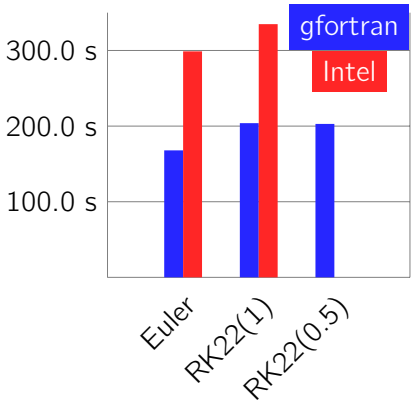


Table 5.8: Execution times for the comparison between the integration schemes

It can be concluded that a RK22 scheme is preferable to an Euler scheme. Indeed, there is not a big difference between the execution times but the precision provided by a second order scheme is better.

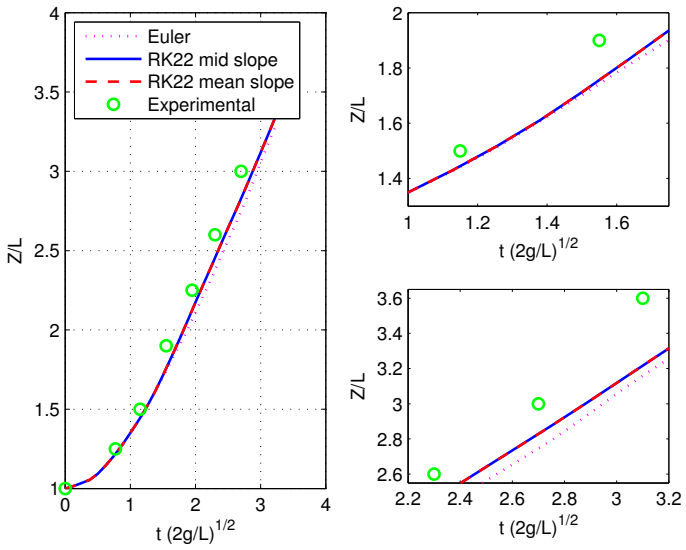


Figure 5.21: Comparison of the position of the wave front for three integration schemes

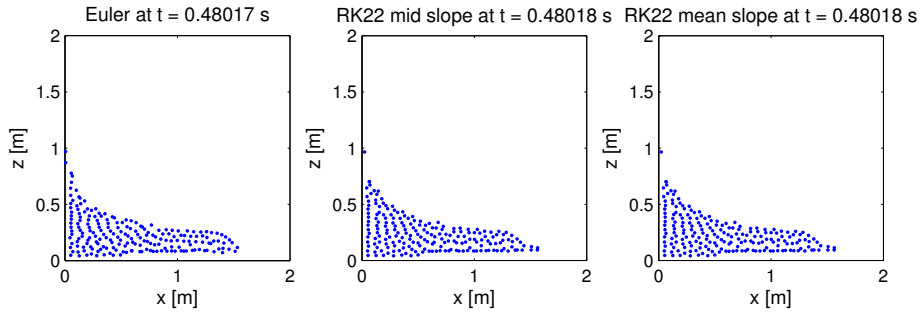


Figure 5.22: Comparison of a slice of fluid for three integration schemes

5.2.3 Using a symmetry condition in order to implement a boundary

The boundary conditions are not easy to handle in the SPH formalism. Indeed, near the boundaries, the kernel is not normalised and some undesired viscosity forces can happen. In order to avoid these problems, the symmetry properties of some specific situations can be used. In this section, the example of a dam break on a wet bed will be illustrated. Two situations will be considered. One which takes the same geometry as the one given in figure 5.9 and another which will consider a symmetry axis. This one is given in figure 5.23.

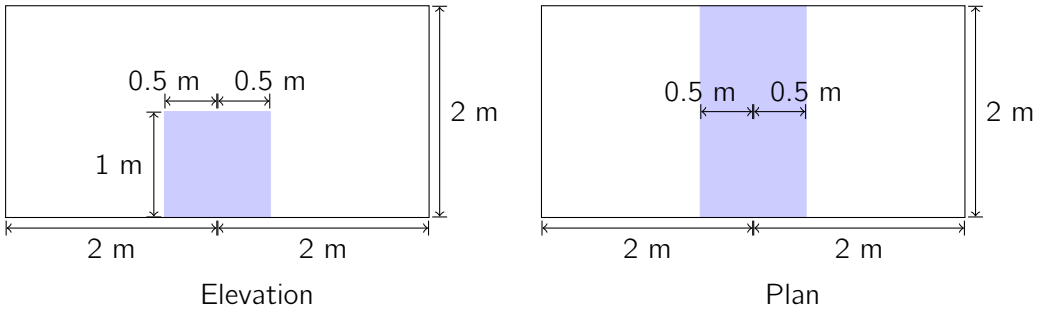


Figure 5.23: Initial geometry of test case that uses the symmetry

The parameters used for the ordinary test (test a) and the symmetric test (test b) are given in table 5.9. The execution time of the test b is nearly equal to twice the execution time of the ordinary test. This is normal as the number of particles is approximately doubled.

Characteristics	**Test a**	**Test b**
Nb FP	9762	16242
Nb MP	7800	16380
h_0 [m]	0.06	0.06
c_0 [m/s]	35	35
Smoothing function	Cubic spline	Cubic spline
α	0.5	0.5
Equation of state	QI fluid	QI fluid
Kernel correction	no	no
Simulation time [s]	1.5	1.5
Calculation time	3.4 min	6.9 min

Table 5.9: Characteristics of the tests used to compare the influence of the boundary consditions

The position of the wave front is given in figure 5.24. It can be easily seen that the use of a symmetry condition offers better results than the ordinary configuration. From figure 5.25, it can be understood that no more viscous force intervene near the boundary which is the cause of better results.

Such boundary conditions seem very interesting even if the calculation time is doubled.

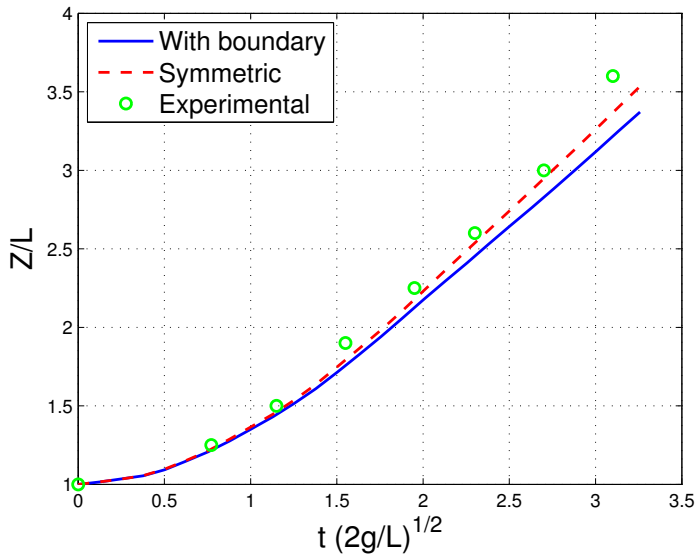


Figure 5.24: Comparison of the position of the wave front when using or not a symmetry condition

Nevertheless, the same level of precision is obtained more quickly with a symmetry than with more particles (such as in the section 5.1.3). This kind of boundary condition is not studied more in this master thesis but it could be interesting to investigate more deeply this possibility for further works.

5.2.4 Influence of the kernel gradient correction

The last comparison concerns the use of the correction of the kernel gradient. The geometry used is the one of figure 5.9. The parameters are given in table 5.10. It shows an execution time 40% greater when the kernel gradient correction is turned on.

Concerning the results, nearly no difference are observed between the two approaches. The kernel gradient correction shows a very slight advantage when the positions of wave front are compared. However, the increase in the execution time is not justified by the poor gain obtained. The figures 5.26 and 5.27 display the small differences.

It should be noticed that the test has been performed only for a dam break on a dry bed. The conclusions could be different in another situation.

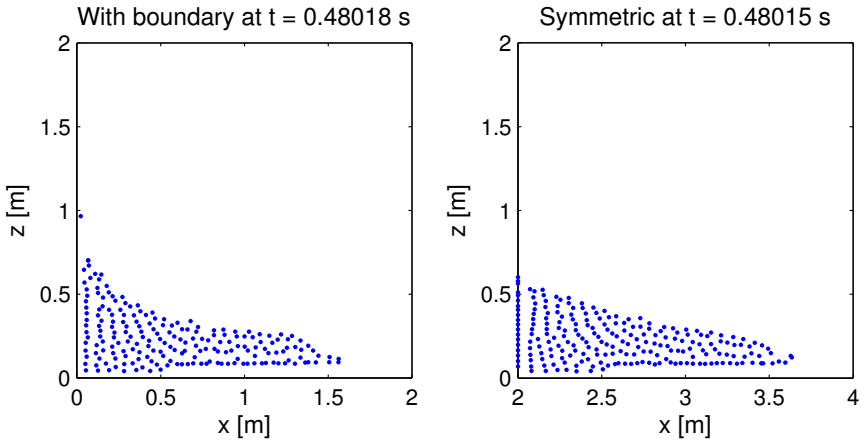


Figure 5.25: Comparison of a slice of fluid when using or not a symmetry condition

Characteristics	**Nor correction**	**With correction**
Nb FP	9762	9762
Nb MP	7800	7800
h_0 [m]	0.06	0.06
c_0 [m/s]	35	35
Smoothing function	Cubic spline	Cubic spline
α	0.5	0.5
Equation of state	QI fluid	QI fluid
Kernel correction	no	yes
Simulation time [s]	1.5	1.5
Calculation time	3.4 min	4.8 min

Table 5.10: Characteristics of the tests used to compare the influence of a kernel gradient correction

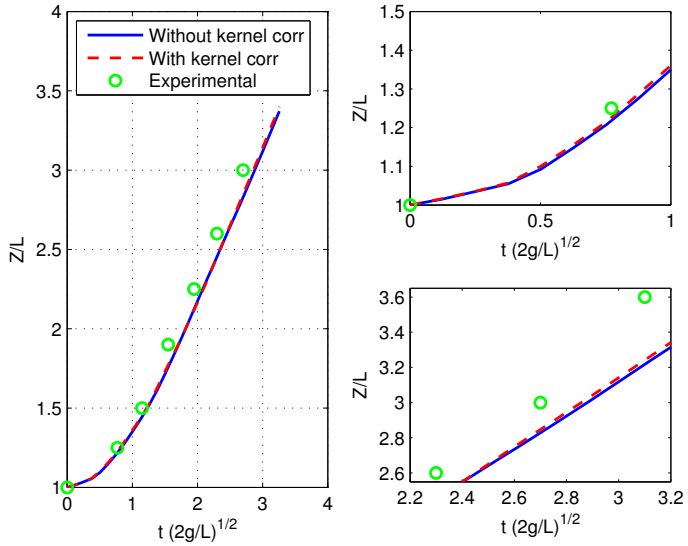


Figure 5.26: Comparison of the position of the wave front when using or not a kernel gradient correction

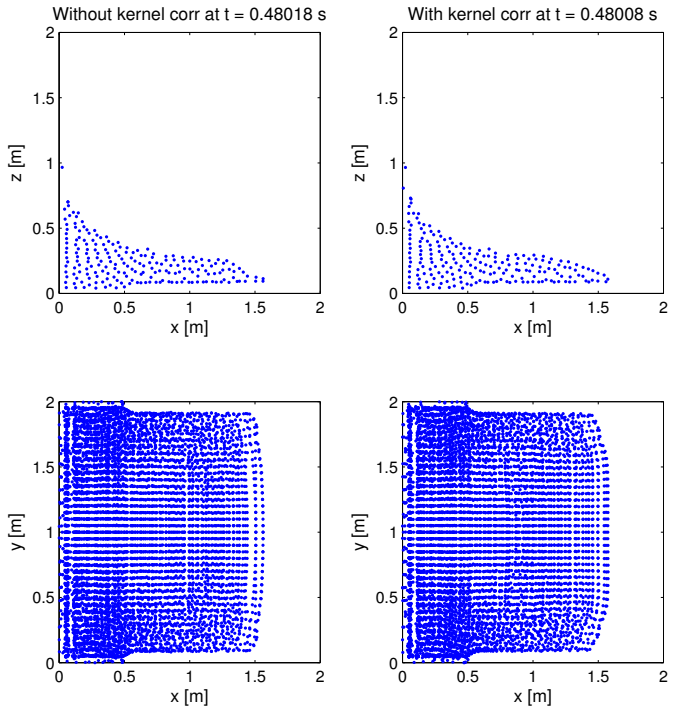


Figure 5.27: Comparison of a slice of fluid and a plan view when using or not a kernel gradient correction

5.3 Test cases to expose the possibilities

The previous test cases are quite simple. Most of them can be done thanks to an Eulerian program in 2-D. In this section, I will focus on some problems that are more suited to a 3-D SPH program. The possibilities of the code will be highlighted.

We have already seen that the program was able to:

1. reproduce some simple problems such as a dam break
2. handle moving boundaries

Here, I will show the 3-D features of the program as well as the flexibility of the boundary conditions used.

5.3.1 Dam break followed by a jump

This test case takes the same geometry as for the dam break on a dry bed except that a jump is added. The initial geometry is described in figure 5.28.

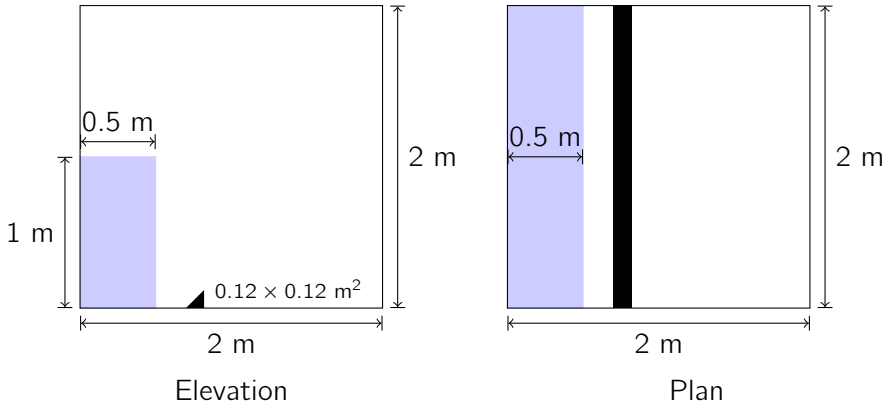


Figure 5.28: Initial geometry of the test case *dam break followed by a jump*

The parameters used for this test case are given in table 5.11. It can be seen that a higher speed of sound was taken. This was done in order to avoid problems when the water hits the right boundary. With a lower speed of sound (and thus a higher time step), the fluid particles penetrated the boundary. Even if the results presented after are in 2-D, the simulation was done in 3-D. Nothing interesting happens in the third dimension.

The results of this test case are given as snapshots in figure 5.29. The colours represent the velocity of a particle calculated as:

$$U = \sqrt{u_x^2 + u_y^2 + u_z^2} \tag{5.3}$$

We can observe a greater speed when the fluid takes the jump. When it hits the right corner, the velocity is also increased as a high repulsion force is created by the boundary. Finally, a wave is observed at the end of the simulation.

Characteristics	**Dam break with a jump**
Nb FP	52973
Nb MP	123750
h_0 [m]	0.024
c_0 [m/s]	70
Smoothing function	Cubic spline
α	0.12
Equation of state	QI fluid
Kernel correction	no
Simulation time [s]	1.8
Calculation time	4.1 h

Table 5.11: Characteristics of the test *dam break followed by a jump*

In order to have a better view of the simulation, a website has been created with some animations. Information about this website is given in the appendix B.3.

This test case shows that a SPH model can deal easily with jumps. This kind of simulation can be helpful to simulate the flow on a spillway and the jump. It can be also imagined to simulate the impact of two jets coming from two spillways (e.g. Hoover dam, NV-AZ, USA, see figure 5.30).

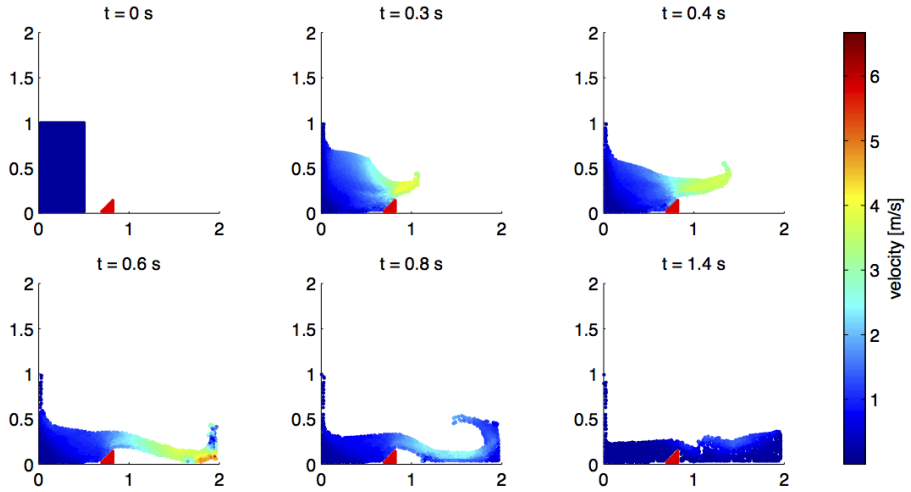


Figure 5.29: Snapshots at different times of the dam break with a jump

Figure 5.30: Impacting jets from the Hoover dam (source: United States Bureau of Reclamation)

5.3.2 Dam break through a grid

Another test case that has been implemented is a dam break through a grid. This test will show how the the dynamic particles can handle this kind of boundary condition. The initial geometry is given in figure 5.31. The grid is generated thanks to fixed particles that are placed with a greater distance than for ordinary boundaries. For example, the traditional boundary particles are spaced with 2 cm. For the grid, the spacing was 8 cm. This allows the fluid particles to go through the grid without being stuck.

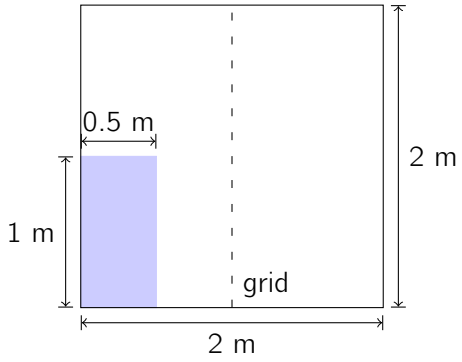


Figure 5.31: Initial geometry of the test case *dam break through a grid*

The parameters used for this simulation are given in table 5.12. The speed of sound has been lowered since there is no more impact to handle.

Characteristics	**Dam break through a grid**
Nb FP	60978
Nb MP	123750
h_0 [m]	0.024
c_0 [m/s]	35
Smoothing function	Cubic spline
α	0.25
Equation of state	QI fluid
Kernel correction	no
Simulation time [s]	1
Calculation time	1.3 h

Table 5.12: Characteristics of the test *dam break through a grid*

The results are given in figure 5.32. A slow-down in speed can be observed around the grid. This can be well seen at 0.6 s: the flow slows down upstream of the grid while a higher velocity can be observed downstream. Another observation is about the velocities just next to the grid. At 0.4 s, a stream through a mesh can be clearly observed. This means that the velocities increase locally in order to go through the grid.

This test was not verified. It could be interesting to check the physics of this phe-

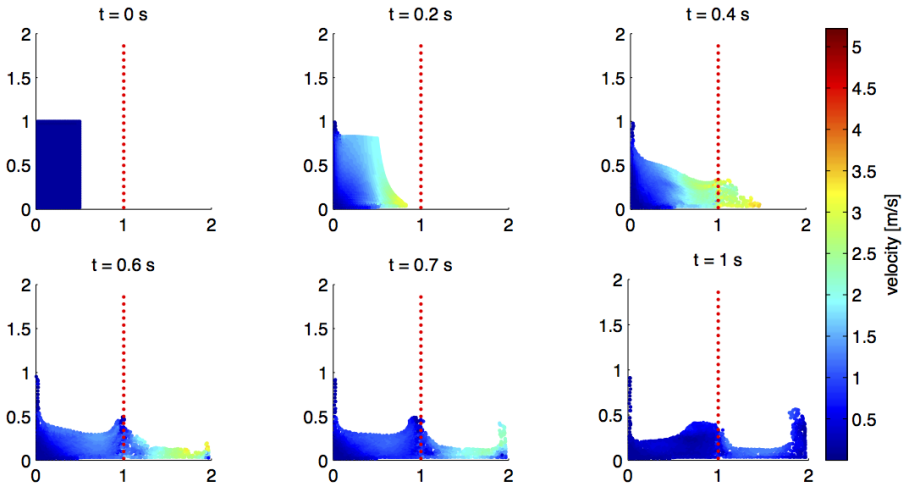


Figure 5.32: Snapshots at different times of the dam break through a grid

nomenon thanks to experimental measures. However, this task is not part of this master thesis.

5.3.3 3-D dam break

The next test case is a 3-D dam break. This is the second test case that really needs a 3-D implementation, after the spinning tank. The initial geometry is described in figure 5.33. An initial water column is placed in a corner of the domain. An initial hydrostatic pressure in set. Then, the set of particles is released.

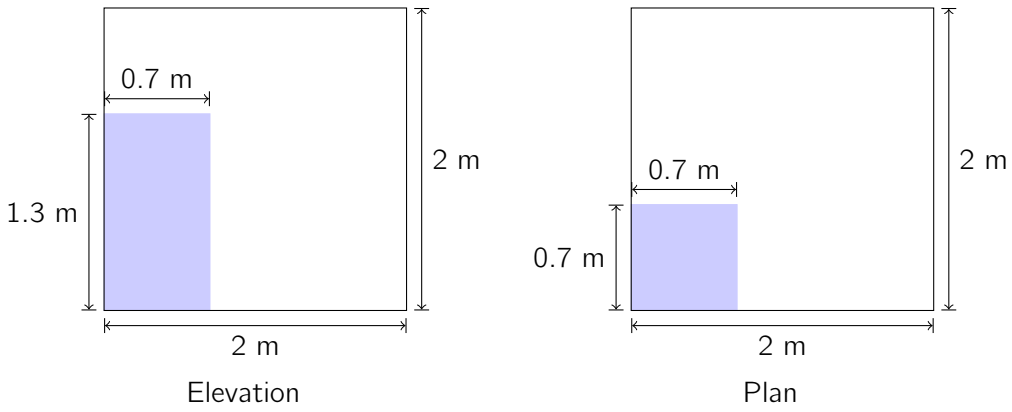


Figure 5.33: Initial geometry of the test case *3-D dam break*

The parameters used for this test case are given in table 5.13.
Some snapshots are given in figure 5.34. It is interesting to notice the splashing that

Characteristics	3-D dam break
Nb FP	60402
Nb MP	79625
h_0 [m]	0.024
c_0 [m/s]	35
Smoothing function	Cubic spline
α	0.2
Equation of state	QI fluid
Kernel correction	no
Simulation time [s]	2
Calculation time	1.8 h

Table 5.13: 3-D dam break characteristics

occurs at 1.2 s. This is a good illustration of the possibilities of an SPH program. This kind of phenomenon is not reproducible by an Eulerian code.

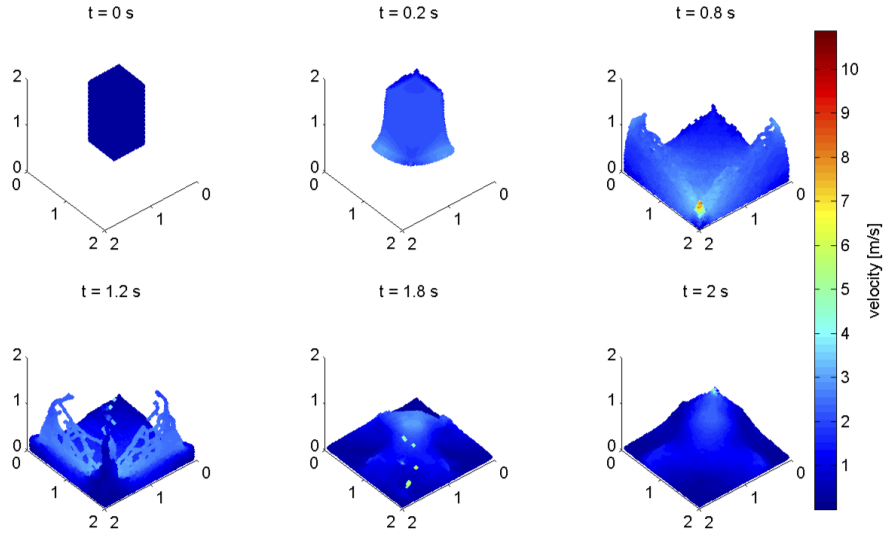


Figure 5.34: Snapshots at different times of the 3-D dam break

This simulation represents exactly the purpose of the implemented code:

1. the SPH method is especially well suited to deal with
   (a) discontinuous problems (e.g. the splashing of this case)
   (b) problems where the dynamic terms are dominant
2. the implemented code is a 3-D code made to handle problems which cannot be represented in 2-D.

5.3.4 Structure impact

The last test case concerns the impact of two structures by a breaking dam. The initial geometry is given in figure 5.35.

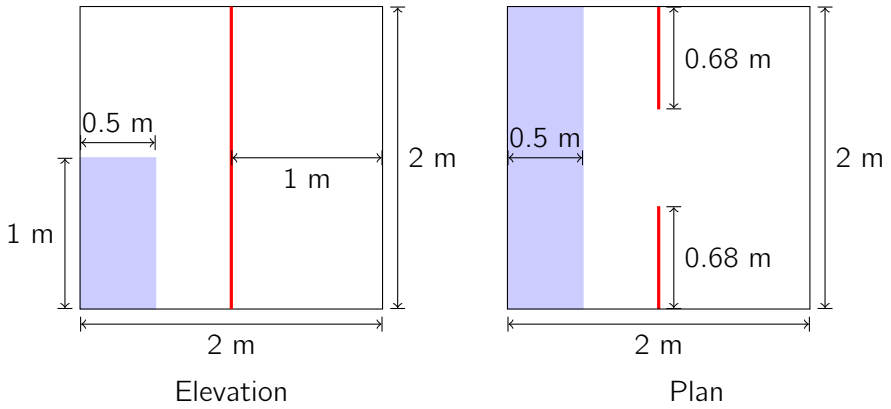


Figure 5.35: Initial geometry of the test case *structure impact*

The parameters used for this simulation are given in table 5.14. These parameters are similar to what was used earlier.

Characteristics	**Structure impact**
Nb FP	67134
Nb MP	123750
h_0 [m]	0.024
c_0 [m/s]	35
Smoothing function	Cubic spline
α	0.2
Equation of state	QI fluid
Kernel correction	no
Simulation time [s]	2
Calculation time	2.7 h

Table 5.14: Characteristics of the test *structure impact*

A snapshot at 0.6 s is given in figure 5.36. It shows a splashing on the structures as well as a jet that impacts the boundary of the domain. An animation can be viewed on the website given in the appendix B.3. This animation gives a better view of the simulation. In this animation, some interesting phenomena can be observed such as vortexes behind the structures and a better view of the free surface. It could be interesting to check the existence of these vortexes via an experimental model.

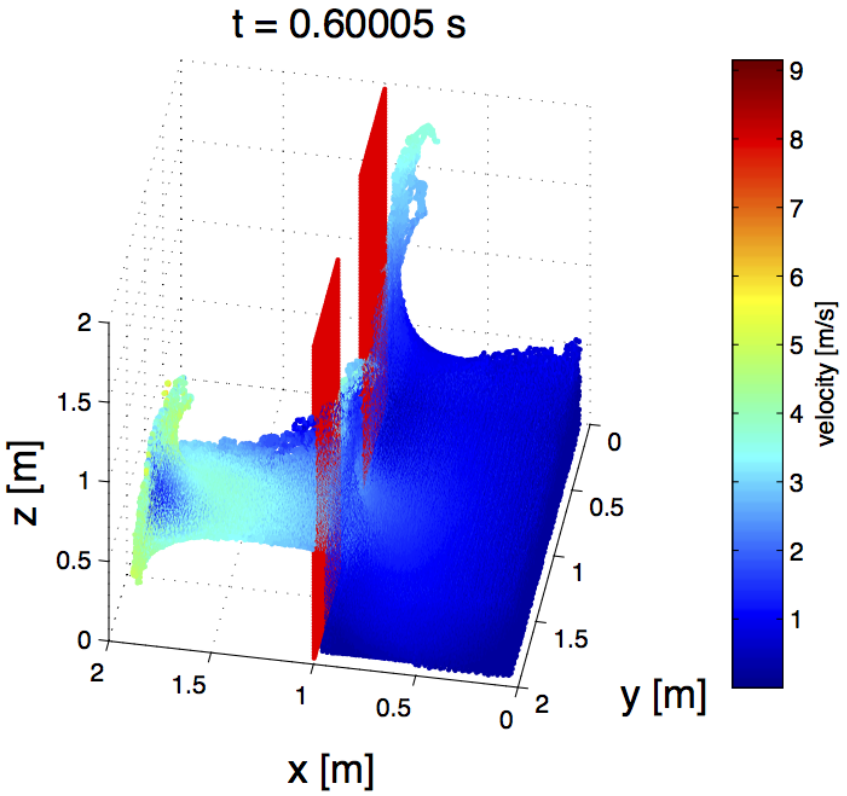


Figure 5.36: Snapshot of the a structure impact by a dam break

5.3.5 Conclusions

The previous simulations have showed that the SPH program implemented had a lot of possibilities. Indeed, it can handle 3-D problems natively, 2-D problems can be also simulated if they are generated in 3-D. Moreover, the SPH method implemented shows interesting behaviours that are difficult to reproduce with Eulerian methods. These behaviours include splashes, jumps, etc. Finally, with minimal efforts, the code is able to deal with mobile boundaries (e.g. spinning tank).

These observations make the implemented SPH code usable for a wide range of problems which are not suited for Eulerian methods.

Chapter 6

Conclusion

6.1 Summary of this master thesis

During this work, I started without any knowledge about the SPH method. My first job was to make a review of the method. I began with the positioning the method in the context of current numerical methods. The SPH method is a meshless, particle and Lagrangian method. It is quite different from the classical methods which are mainly, in hydraulics, Eulerian methods.

After the positioning, I had to understand the establishment of the SPH formalism. It is based on the integral representation of a function. Then, the Naviers-Stokes equations needed to be written in a Lagrangian form in order to be used with the SPH method. After having applied the SPH formalism to the Naviers-Stokes equations of continuity and of conservation of momentum, I discussed a few problems relative to the method. These issues concerned mainly the equations of state, the nearest neighbours search as well as the boundary conditions.

After having well described the method, I could begin with the explanation of the implementation. It was based on the object-oriented paradigm. This kind of implementation was very useful for different reasons.

Finally, the program has been tested with some test cases. These tests were made in order to validate the program, to compare different options as well as to show the different possibilities of the code.

6.1.1 Objectives fulfilment

The main objective was to develop a program based on the SPH method in order to provide didactic results to the user. I will explain in this section how my work provided such results.

The development began with the understanding of the SPH method and its different options. The choices of the different options were made in order to allow an easy and efficient way to run any desired simulation. To do so, I have decided:

1. To implement the code in 3-D. Thanks to this, the users are not limited to only 2-D problems. An infinite range of asymmetric and 3-D distributed problems (e.g.

3-D dam break) can be modelled. Moreover, it has been decided to implement cubic domains only in order to give an easy way to simulate any problem. If the physical domain is not cubic, the boundaries can be initialised in order to avoid the restriction of a cubic domain. Nevertheless, memory will be allocated for the whole cubic domain.

2. To use dynamic boundary particles. These particles provide an easy way to implement the boundaries. Indeed, many geometries can be done with these particles. For example, in the different tests, I used the particles for traditional boundaries but also for moving boundaries and for obstacles put in the middle of the domain. With such boundaries, the user is free to model a large range of boundary conditions.

3. To use two equations of state. The one used for ideal gas allows to use a lower time step which allows quicker simulations. This choice can be done for a first approximation. The second one can be used for incompressible fluids. The use of two equations of state allows the user to understand better the influence of the compressibility on a flow.

4. To use an artificial viscosity. This kind of viscosity has the disadvantage to have no physical coefficients in its expression. However, it is very flexible which allows the user to play on the viscosity parameter (thanks to the coefficients α and β) to understand better the role of the viscosity in a problem.

5. To implement the program with an object-oriented paradigm. This allows the more advanced users to change easily the code according to some specific requirements which could not be taken into account in the current code.

6. To use input and output files. These are generated and post-processed by a user's program. This gives much more possibilities to the user than with an integrated interface.

The previous items show clearly that the program has been designed in order to provide the best experience to a wide range of users: from the student to the researcher.

Indeed, the students can test the program with a limited number of particles and play with some parameters in order to see how they influence physically the problem. The execution time can be fast if the number of particles is lower than 5000. For the researchers, the implemented program can provide a first overview of the main characteristics of a problem. These reasons allows me to state that the implemented program is a didactic model.

In order to provide better results (more quantitative than qualitative) some improvements should be done. These improvements are discussed later.

6.1.2 Difficulties and solutions

During the study of the method, the implementation and the tests, a few issues were encountered. Here are the solutions I brought.

The first one concerns the equation of state for quasi-incompressible fluids. During the spinning tank test case, I saw that using a lower speed of sound as suggested by [Monaghan and Kos, 1999] leads to wrong results. Indeed, as analysed in the chapter 3, the speed of sound has a great influence on the compressibility of a fluid. The assumption introduced by [Monaghan and Kos, 1999] should be used carefully. If problems occur, it is preferable to take the physical value (1480 m/s) which is the value used in the Tait equation.

Then, during my tests, I figured out that the numerical precision was an important parameter. In fact, as shown in the chapter 3, if a very low compressibility is used, a single precision is not enough and the repulsive forces between the particles could not be taken into account as it should. A single precision is still possible for weakly compressible fluids (with a low speed of sound).

During the implementation, a few solutions had to be found in order to avoid some `Fortran`'s restrictions. These problems include some restrictions with polymorphism. This was solved by using less extendible classes. Another point concerns the execution when the code is compiled with gfortran or the Intel compiler. Huge differences between the execution times were observed. The code compiled with Intel is approximately 70% slower[1] than with gfortran. gfortran was naturally chosen for its efficiency as well as because it is free.

The Last problem encountered concerned the initialisation of the particles. Indeed, when particles are put in a still tank with an hydrostatic pressure, there is no initial equilibrium. In order to understand and solve the problem, a study of the problem was led in 1-D (see appendix A.5). It was shown that an initial equilibrium could be found by initialising the masses in a particular way. A linear system must be solved and a few iterations should also be done. However, this system can be solved only when the number of particles is even.

6.2 Future

This master thesis was a first work on the SPH method in the HECE service of the University of Liège. This section will show the possibilities of the current program as well as the improvements that can be made for further work.

6.2.1 Further work

The implemented program features already a few interesting characteristics such as two equations of state and the possibility to correct the kernel gradient. However, some other possibilities can be added in order to use the program in a more quantitative way and for other purposes.

Currently, the boundaries are considered fixed. It is possible to make theme move (e.g. spinning tank). However, this requires a modification of the code. It could be interesting to add a file that includes a law of movement for the boundaries. With this kind of improvement, a moving boundary could be used to generate waves.

[1] The same compilation options were chosen and the OpenMP was turned on in both cases.

The implemented boundaries are made to avoid any leakage. But it could be interesting to add the possibility of permeability and thus inflow and outflow. This kind of boundary condition has been discussed in [Vacondio et al., 2012].

A correction of the kernel is possible currently for the kernel gradient present in the equation of conservation of momentum. But no correction is made for the kernel gradient in the equation of continuity. This leads to some oscillations of the densities. In order to improve this, adding a density filter should be interesting. This filter smooths the densities every ~ 30 iterations which provides better results.

Currently, the viscosity is treated thanks to a term of artificial viscosity. This term is not a physical quantity. If an unknown problem must be simulated thanks to the method, it will be difficult to know is the results are correct (quantitatively). Other methods that use a physical viscosity exists. Some information can be found in [Violeau, 2012]. Using such a viscosity would be a great advantage for the code.

A final suggestion for further work would be to initialise the particles in order to have an initial equilibrium. A solution was discussed in the appendix A.5. Nevertheless, this kind of improvement can be made directly in the input file. No modification is required in the code.

Other variants of the method are given in the literature such as a smoothing of the velocities thanks to the XSPH method. However, such corrections are not considered as a priority.

6.2.2 Potential applications

As shown earlier, the implemented program is able to simulate a lot of situations. These situations include dam breaks (on dry or wet bed), impact of structures, flows through a grid, spinning tank, etc. The SPH method is well suited for the problems where the fluid impact structures. This is what was shown for the dam breaks. These test cases are more theoretical test cases than practical cases.

Considering the tests made earlier and the properties of the SPH method, it is well suited for a wide range of practical problems such as the impact of two jets (e.g. Hoover dam's spillways), the impact of a flow on a structure (e.g. opening of a valve and sudden impact of a bridge pier), impact of a structure with openings (e.g. waves that impact a building with open windows, storm and offshore structures (see figure 6.1[2]),...), etc.

Moreover, with some small modifications, the program could be able to handle the mixing of:

- two liquids, e.g.: water and oil, design of barriers against pollution, etc.
- a gas and a liquid, e.g.: mixed flows, inclusion of air when a wave breaks, etc.
- a liquid and a solid, e.g.: transport of sediments when the properties of the flow quickly change.

[2]Source: `http://www.theartofdredging.com/jonesactii.htm`

Figure 6.1: Waves impacting an oil platform

These examples show how the SPH method can be used in hydraulics. The possibilities are very wide and some improvements still must be done in order to increase its accuracy. The SPH method is a very promising method in many fields.

Appendix A

Mathematical developments

A.1 B in the equation of state

The value of B in the equation of state (3.25)

$$p = B\left(\left(\frac{\rho}{\rho_0}\right)^{\gamma} - 1\right)$$

can be determined thanks to the relation between the density and the speed of sound (3.26)

$$c^2 = \left(\frac{\partial p}{\partial \rho}\right)_s$$

When applying this relation to the equation of state, we obtain

$$c^2 = \left(\frac{\partial p}{\partial \rho}\right)_s = \frac{B\gamma}{\rho_0}\left(\frac{\rho}{\rho_0}\right)^{\gamma-1} \tag{A.1}$$

Knowing that at the reference density (ρ_0), the speed of sound is equal to the the speed of sound at the reference density ($c = c_0$), equation (A.1) can be rewritten as

$$c_0^2 = \frac{B\gamma}{\rho_0}\left(\frac{\rho_0}{\rho_0}\right)^{\gamma-1} = \frac{B\gamma}{\rho_0} \tag{A.2}$$

Equation (A.2) gives the expression of B:

$$B = \frac{c_0^2\rho_0}{\gamma} \tag{A.3}$$

A.2 Integration of the modified Tait equation

Equation (3.33)

$$-\frac{1}{V}\left(\frac{\partial V}{\partial p}\right)_s = \frac{1}{\gamma\left(p + B(s)\right)} \tag{A.4}$$

can be integrated in order to obtain a relation between the pressure and the the density:

$$\int_{V_0}^{V} \frac{-1}{V}\, dV = \int_{p_0=0}^{p} \frac{1}{\gamma} \frac{1}{p+B}\, dp \tag{A.5}$$

Once integrated, we obtain

$$\ln\left(\frac{V_0}{V}\right) = \frac{1}{\gamma}\left(\ln\left(p+B\right) - \ln\left(B\right)\right) \tag{A.6}$$

which can be simplified into

$$\frac{V_0}{V} = \left(1+\frac{p}{B}\right)^{1/\gamma} \tag{A.7}$$

Equation (A.7) can also be written as

$$p = B\left(\left(\frac{V_0}{V}\right)^{\gamma} - 1\right) \tag{A.8}$$

For an arbitrary volume of fluid, we have

$$\begin{aligned} m &= V\ \rho \\ &= V_0\ \rho_0 \end{aligned}$$

thanks to the conservation of mass. This expression leads to

$$\frac{V_0}{V} = \frac{\rho}{\rho_0} \tag{A.9}$$

Equation (A.9) allows to write equation (A.8) in respect to densities ρ and ρ_0:

$$p = B\left(\left(\frac{\rho}{\rho_0}\right)^{\gamma} - 1\right) \tag{A.10}$$

Equation (A.10) is the same as equation (3.25).

A.3 Setting of an initial pressure

The setting of the pressure within a fluid is done thanks to the initialisation of the densities. The hydrostatic pressure is expressed by

$$p = \rho_0\ g\ H \tag{A.11}$$

where H is the depth of a particle and ρ_0 the reference density. The situation is represented in figure A.1.

For an ideal gas, we can write

$$p = \rho_0\ g\ H = \frac{R\ T}{M}\left(\frac{\rho}{\rho_0} - 1\right) \tag{A.12}$$

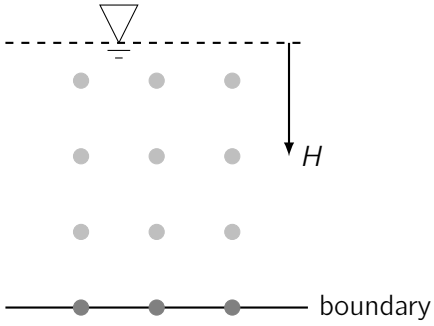


Figure A.1: Initialisation of densities

thanks to equations (A.11) and (3.24). After a trivial manipulation, we obtain

$$\rho = \rho_0 \left(1 + \frac{M}{RT}\rho_0 \; g \; H\right) \tag{A.13}$$

For a quasi-incompressible fluid, equations (A.11) and (3.25) can be combined:

$$p = \rho_0 \; g \; H = B\left(\left(\frac{\rho}{\rho_0}\right)^{\gamma} - 1\right) \tag{A.14}$$

This leads to the following equation:

$$\rho = \rho_0 \left(1 + \frac{1}{B}\rho_0 \; g \; H\right)^{1/\gamma} \tag{A.15}$$

A.4 Coefficients used to increase the maximum smoothing length

The maximum smoothing length is useful for the setting of the cells' size used to sort the particles. In order to avoid a modification in the number of these cells, the maximum smoothing length h_{max} is increased. The coefficient used to increase this smoothing length is based on formula (3.42):

$$\frac{h}{h_0} = \left(\frac{\rho_0}{\rho}\right)^{1/d}$$

It is estimated that, if water is considered as an ideal gas, the density ρ can decrease to 800 kg/m^3. This seemed to be a good order of magnitude when observing the results of some tests. This leads to a coefficient of

$$\left(\frac{h}{h_0}\right)_{\text{IG}} = \left(\frac{1000}{800}\right)^{1/3} = 1.077 \approx 1.1 \tag{A.16}$$

For a quasi-incompressible fluid, the minimum density was estimated at 950 kg/m^3. This leads to the following coefficient:

$$\left(\frac{h}{h_0}\right)_{\text{QI}} = \left(\frac{1000}{950}\right)^{1/3} = 1.017 \approx 1.02 \tag{A.17}$$

This development is based on estimations that were verified during the test cases. However, it is possible that the coefficients do not cover all the possible density fluctuations. This is why a warning message is implemented in the program. Nevertheless, this warning message never appeared during the tests of the program. The reasons are:

- The smoothing length is updated according to the mean of the densities. This means that the coefficients are taken with a sufficient safety factor.
- The cells are initialised in order to have an integer number of cells on a side and a size of cell $\leq \kappa h_{\max}$. In many situations, this allows to have a new safety margin.

A.5 Initial equilibrium of a water column

A.5.1 First approach

The initialisation of the particles described in section 3.2.8 do not lead to the equilibrium of an amount of water placed in a tank. This is highlighted in section 5.1.2.

In order to have an initial equilibrium, it has been decided to analyse the problem in 1-D. The system used for this is shown in figure A.2. There are 4 mobile particles and a fixed particle used to represent the boundary. They are spaced with $s = 0.01$ m.

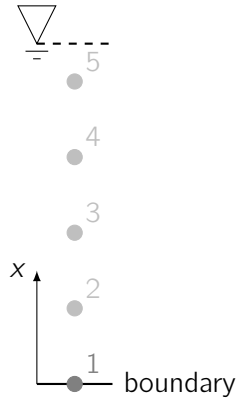


Figure A.2: Easy system to analyse the initial equilibrium

In order to have the equilibrium at the beginning of the simulation, Du/Dt must be equal to 0 for every particle. The expression of Du/Dt is given in equation (3.22). The artificial viscosity is equal to 0 as initially $u_{ab} = 0$. It remains in 1-D:

$$\frac{Du_a}{Dt} = -\sum_{b=1}^{N} m_b \left(\frac{p_b}{\rho_b^2} + \frac{p_a}{\rho_a^2} \right) \nabla_a W_{ab} - g \tag{A.18}$$

Let us consider an initial hydrostatic pressure. Thus, the pressures and the densities can be calculated. A quasi-incompressible law is used with $c_0 = 5$ m/s. The last variable on which

we can play in order to have the equilibrium is the mass, or more especially the volume occupied by a particle. A quadratic kernel is used with $h = 1.2s$.

Equation (A.18) can be written for each particle with the masses m_b as unknowns. This gives a linear system of equations to solve, $Am = b$:

$$\begin{pmatrix} 0 & -2.3134 & -0.5795 & 0 & 0 \\ 2.3134 & 0 & -1.7473 & -0.4166 & 0 \\ 0.5795 & 1.7473 & 0 & -1.1731 & -0.2513 \\ 0 & 0.4166 & 1.1731 & 0 & -0.5904 \\ 0 & 0 & 0.2513 & 0.5904 & 0 \end{pmatrix} \begin{pmatrix} m_1 \\ m_2 \\ m_3 \\ m_4 \\ m_5 \end{pmatrix} = \begin{pmatrix} 9.81 \\ 9.81 \\ 9.81 \\ 9.81 \\ 9.81 \end{pmatrix} \tag{A.19}$$

where the elements of the matrix are equal to

$$A_{ab} = -\left(\frac{p_b}{\rho_b^2} + \frac{p_a}{\rho_a^2}\right) \nabla_a W_{ab} \tag{A.20}$$

The particle 1 is a boundary particle. This means it cannot move freely. In order to take it into account, a virtual force must be added to equilibrate the gravity and the pressure forces coming from the neighbours. This repulsive force is introduced in equation (A.18) which becomes for the particle 1:

$$\frac{Du_1}{Dt} = -\sum_{b=1}^{N} m_b \left(\frac{p_b}{\rho_b^2} + \frac{p_1}{\rho_1^2}\right) \nabla_1 W_{1b} - g + R \tag{A.21}$$

with $R = g + \sum_{b=1}^{N} m_b \left(\frac{p_b}{\rho_b^2} + \frac{p_1}{\rho_1^2}\right) \nabla_1 W_{1b}$. The term R is added to the vector b:

$$b = \begin{pmatrix} 9.81 - R \\ 9.81 \\ 9.81 \\ 9.81 \\ 9.81 \end{pmatrix} \tag{A.22}$$

Adding R in the vector b asks an iterative solving.

When we want to solve the system (A.19), we get $\det A = 0$. This means that the system cannot be solved. In order to solve it, I have tried to give an initial mass to a particle by modifying the matrix A and the vector b. I have decided to fix the mass of the top particle (5) by assigning it to $m_5 = s\rho_5$. The system (A.19) becomes

$$\begin{pmatrix} 0 & -2.3134 & -0.5795 & 0 & 0 \\ 2.3134 & 0 & -1.7473 & -0.4166 & 0 \\ 0.5795 & 1.7473 & 0 & -1.1731 & -0.2513 \\ 0 & 0.4166 & 1.1731 & 0 & -0.5904 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} m_1 \\ m_2 \\ m_3 \\ m_4 \\ m_5 \end{pmatrix} = \begin{pmatrix} 9.81 - R \\ 9.81 \\ 9.81 \\ 9.81 \\ 10.02 \end{pmatrix} \tag{A.23}$$

Now, the matrix A can be inverted. In order to begin the iteration, the masses m_2 and m_3 should be given initial values. To converge as quickly as possible, the values assigned

must be around $m_i = s\rho_i$. When a first estimation of the masses is obtained, the masses m_2 and m_3 should be used to modify the artificial repulsive force R. After approximately 5 iterations, the system has converged. The mass vector is equal to:

$$m = \begin{pmatrix} m_1 \\ m_2 \\ m_3 \\ m_4 \\ m_5 \end{pmatrix} = \begin{pmatrix} 13.69 \\ 10.13 \\ 9.81 \\ 11.35 \\ 10.02 \end{pmatrix} \tag{A.24}$$

When the results are verified by using equations (A.18) and (A.21), all the $Du_i/Dt = 0$ except for the particle 5: $Du_5/Dt = 0.21$. The equilibrium is not found and it seems impossible to find an equilibrium using this technique. Indeed, having a singular matrix in (A.18) leads to the impossibility to find a unique solution.

Having a singular matrix is due to the fact that the matrix is anti-symmetric with an odd number of equations. With an even number of equations, the system can be solved. This condition is very annoying for a random problem. Indeed, nothing guaranty to have an even number of particles.

A.5.2 Second approach

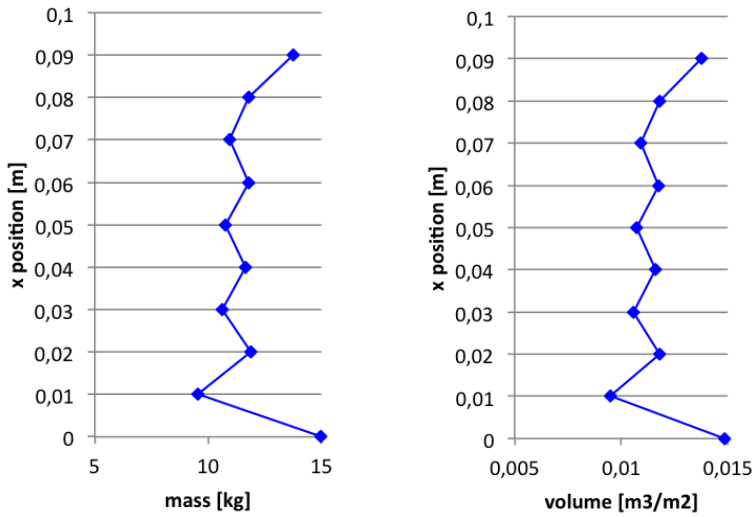
For the second approach, an even number of particles is considered. It is equal to 10. The first particle is still a boundary particle. After having written equation (A.18) for every particles, the matrix A is obtained:

$$\begin{pmatrix} 0.00 & -5.27 & -1.42 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 5.27 & 0.00 & -4.70 & -1.26 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 1.42 & 4.70 & 0.00 & -4.12 & -1.09 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 1.26 & 4.12 & 0.00 & -3.54 & -0.93 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 1.09 & 3.54 & 0.00 & -2.95 & -0.76 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.93 & 2.95 & 0.00 & -2.37 & -0.59 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.76 & 2.37 & 0.00 & -1.78 & -0.42 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.59 & 1.78 & 0.00 & -1.19 & -0.25 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.42 & 1.19 & 0.00 & -0.60 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.25 & 0.60 & 0.00 \end{pmatrix} \tag{A.25}$$

with $\det A = 2063$. After having inverted the matrix A (A.25) and solved the system in the same way as in the previous subsection[1], the masses can be found. These masses are represented in figure A.3 (a). The volumes that correspond to every particles are given in figure A.3 (b). This volume is calculated thanks to $V_a = m_a/\rho_a$.

In figure A.3, the masses are close to a mean value ≈ 10 kg and the volumes are close to a mean value ≈ 0.01 m^3/m^2. These mean values are coherent with the characteristics of the problem. Indeed, the initial spacing is equal to 0.01 m which leads to a mass that should be around $\rho_0 s = 10$ kg/m^2.

[1] With the iterations to find the good R.



(a) (b)

Figure A.3: Distribution of the masses (a) and equivalent volumes (b) in order to have an initial equilibrium

A.5.3 Conclusion

The previous developments have shown that an initial equilibrium can only be found for an even number of particles. When this condition is fulfilled, the masses are distributed around a mean value close to what is expected. However, some small oscillations are observed and the masses near the boundaries (with a kernel not normalised) are not so close to the mean value. Meanwhile, the masses found allow to have an initial equilibrium.

For further works, a more general solution or a trick should be found in order to have an initial equilibrium for an odd number of particles.

A.6 Free-surface of a fluid in a spinning tank

For a static fluid, the conservation of momentum can be written as

$$\rho \mathbf{g} = \nabla p + \mathbf{F} \tag{A.26}$$

Thus, the pressure gradient is written as

$$\nabla p = \rho \left(\mathbf{g} - \mathbf{a} \right) \tag{A.27}$$

For a rotating cylindrical tank (represented in figure A.4), the acceleration terms are equal to

$$\mathbf{g} - \mathbf{a} = \begin{pmatrix} r\omega^2 \\ 0 \\ -g \end{pmatrix} \tag{A.28}$$

Using equations (A.27) and (A.28), we get

$$\nabla p = \begin{pmatrix} \partial p/\partial r \\ \partial p/\partial \theta \\ \partial p/\partial z \end{pmatrix} = \begin{pmatrix} \rho r \omega^2 \\ 0 \\ -\rho g \end{pmatrix} \tag{A.29}$$

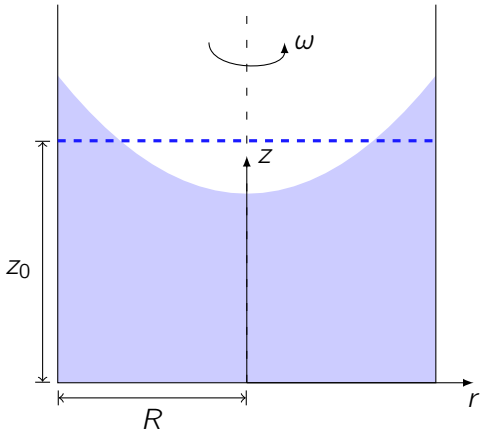


Figure A.4: Diagram of a rotating cylindrical tank

An increment of pressure is equal to

$$\begin{aligned} dp &= \frac{\partial p}{\partial r} dr + \frac{\partial p}{\partial \theta} d\theta + \frac{\partial p}{\partial z} dz \\ &= \rho r \omega^2 dr - \rho g dz \end{aligned} \tag{A.30}$$

At the free surface, $dp = 0$. From (A.30), we have $\rho r \omega^2 dr = \rho g dz$. This gives the derivative of z in respect to r:

$$\frac{dz}{dr} = \frac{r\omega^2}{g} \tag{A.31}$$

After having integrated (A.31), we have

$$z(r) = \frac{r^2 \omega^2}{2g} + C \tag{A.32}$$

with C an integration constant that can be defined thanks to the conservation of mass. If the fluid is incompressible (the case of water), the initial projected surface $A_0 = 2Rz_0$ is equal to the final projected surface A_f which can be calculated by

$$A_f = 2 \int_0^R \left(\frac{r^2 \omega^2}{2g} + C \right) dr \tag{A.33}$$

With $A_0 = A_f$, the constant C can be determined:

$$C = z_0 - \frac{\omega^2 R^2}{6g} \tag{A.34}$$

Finally, the profile of the free surface is obtained thanks to equations (A.32) and (A.34):

$$z(r) = \frac{r^2\omega^2}{2g} + z_0 - \frac{\omega^2 R^2}{6g} \tag{A.35}$$

Appendix B

Additional contents

This appendix shows some additional contents that could not be presented in the previous chapters. The source code is also available at the end of this appendix.

B.1 Still tank

It was shown in the chapter 5 that the hydrostatic pressure was found after a few seconds for an amount of water set without initial pressure in still tank. The next step is naturally to set the particles with an initial hydrostatic pressure. It has been shown earlier that it does not lead to an initial equilibrium. However, let see what happens.

Three tests have been made: the first one is the same as the one in the chapter 5 except that an initial pressure is set (test b), the second one is made with the kernel gradient correction (test c) and for the last one, a second layer of fixed particles has been added (test d). The parameters used for these are given in table B.1.

Characteristics	**Test b**	**Test c**	**Test d**
Nb FP	6196	6196	11840
Nb MP	35836	35836	30720
h_0 [m]	0.024	0.024	0.024
c_0 [m/s]	30	30	30
Smoothing function	Cubic spline	Cubic spline	Cubic spline
α	0.05	0.05	0.05
Equation of state	QI fluid	QI fluid	QI fluid
Kernel correction	no	yes	no
Simulation time [s]	5	5	5
Calculation time	75.2 min	94.8 min	73.2 min

Table B.1: Characteristics of additional tests for *water in a still tank*

The results for the test b are given in figure B.1 (a) and figure B.2 (a). The oscillations are limited in comparison to the test case without initial pressure. However, the pressure

does not settle around the hydrostatic pressure. This is due to some factors such as a non normalised kernel. Some snapshots are given in figure B.3. It can be seen that the particles are no longer ordered. and the pressure does not vary linearly. Near the boundaries, pressure oscillations can be observed.

The results obtained are not very satisfying. In order to improve these, the tests c and d will be observed and discussed.

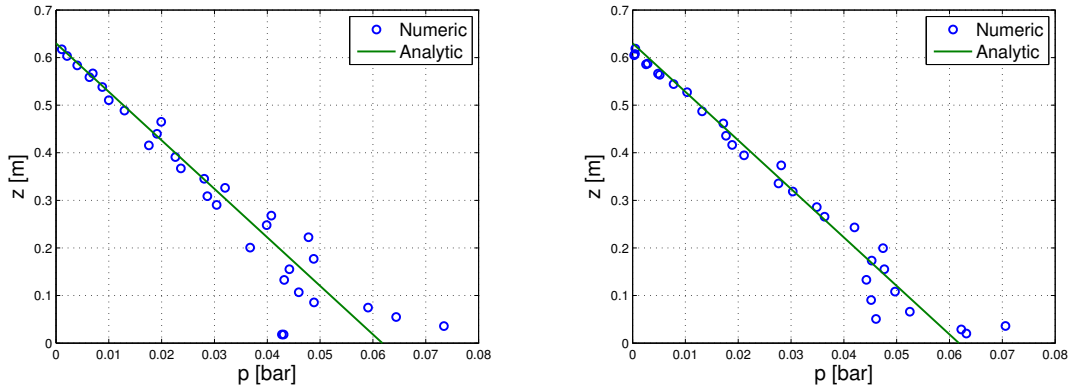


Figure B.1: Distribution of the pressure without the kernel gradient correction (a) and with it (b)

The results of the test c are provided in figures B.1 (b) and B.2 (b). The snapshots are not given because they are very similar to figure B.3. It can be observed that the pressures are closer to the analytical solution with a kernel gradient correction. It can be also seen that the pressures stabilise around the hydrostatic pressure. For these reasons, the correction of the kernel gradient can be considered as positive. However, the initial equilibrium is not obtained.

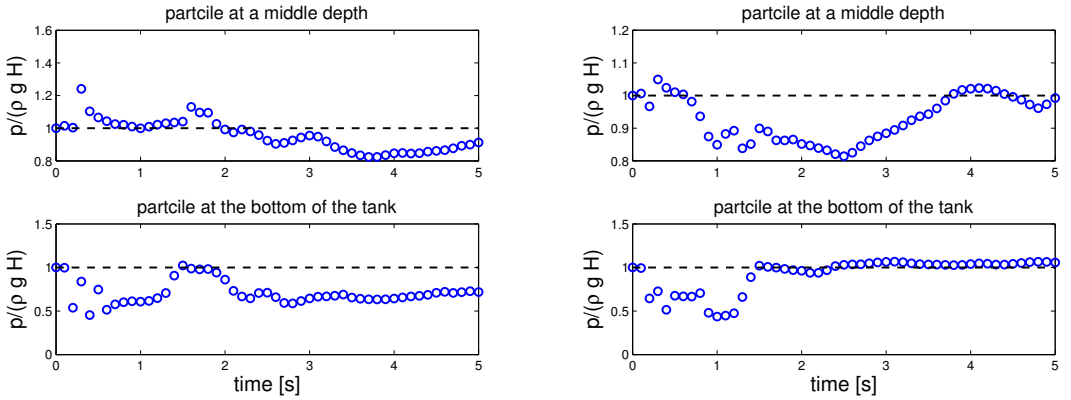


Figure B.2: Evolution of the pressure without the kernel gradient correction (a) and with it (b)

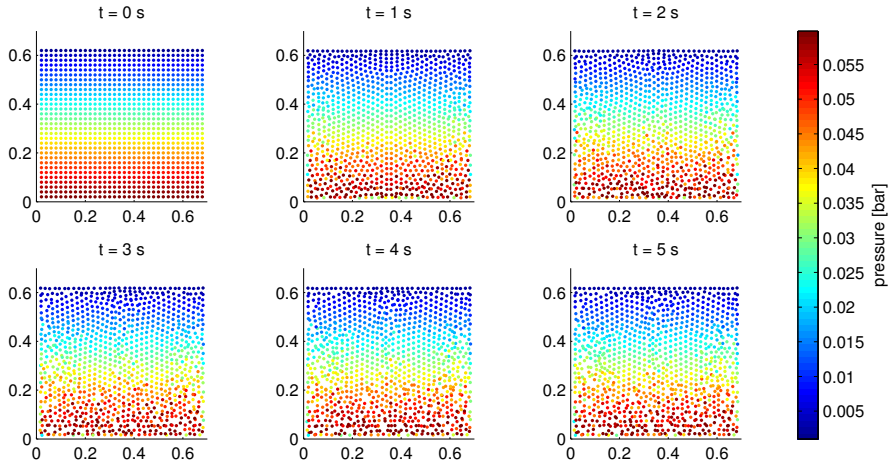


Figure B.3: Snapshots without kernel gradient correction

In the previous tests, the boundary conditions seemed to play an important role in the results' quality. In order to assess this, it has been decided to had a second layer of boundary particles. This is the test d. Some results are given in figure B.4. Some snapshots are given in figure B.5. An improvement of the results is observed. Indeed, the pressures are closer to the analytical solution and converge quicker toward it. Moreover, the results are greatly improved next to the boundaries as it can be seen in figures B.4 (a) and B.5. Adding a second layer of boundary particle, and thus having a more normalised kernel, helps to have better results even if there is no equilibrium at the beginning.

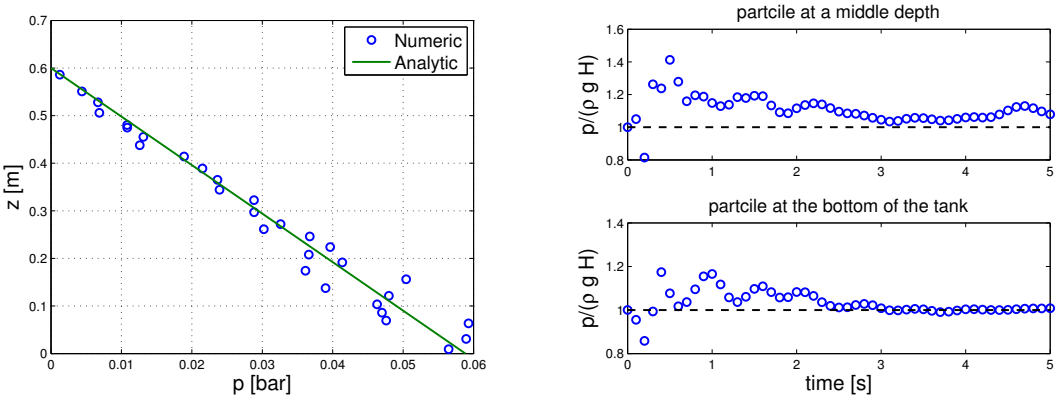


Figure B.4: Evolution of the pressure for a column of water (a) and for two given particles (b)

The previous observations show that the hydrostatic pressure is the equilibrium pressure of the fluid. However, setting the particles in a Cartesian grid with an hydrostatic pressure does not lead to an initial equilibrium. This is also highlighted in the chapter 3 and the

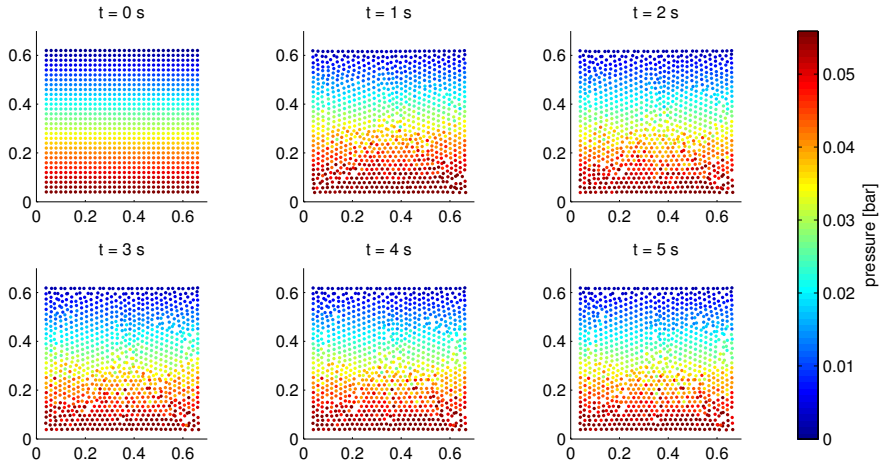


Figure B.5: Snapshots with two layers of boundary particles

appendix A.5. In order to have this initial equilibrium, a work should be done on the masses and/or on their initial position. A further work should deal with it.

However, a few techniques were found in order to improve the results. These techniques concern the normalisation of the kernel. It was done thanks to the adding of boundary particles or the correction of the kernel gradient. Adding a density filter should also help.

It has been observed that the particles go from a regular ordering to a staggered one. Positioning the particles in a staggered way should also help.

B.2 Spinning tank

The test case that concerns the rotating tank presents some gaps near the boundaries (see figure 5.16). In the chapter 5, I gave a reason that concerned the repulsive forces produced by the the boundary particles and the incompressible nature of the fluid. However, in order to validate this theory, I must show that the fluid is in equilibrium.

To do so, I have plotted some parameters in figure B.6. This figure concerns an arbitrary particle close to the boundary. r is the distance between the particle and the center of the tank, v is the tangential velocity of the particle and the up right graph gives the x and y position of the particle at different times.

The first thing to notice is that r is quickly stabilised. The z position is almost stabilised after 7 s. Moreover, when the snapshots of figure 5.17 are observed, a global stabilisation is noticed after 4 s. These observations allow to conclude that the system is stabilised when the profiles are plotted in figure 5.16, i.e. after 7 s.

We can also look at the oscillations of the tangential velocity. A bigger plot with more information is available in figure B.7. It shows that the velocity of a particle oscillates around the velocity of the boundary. The mean tangential velocity is even lower than the

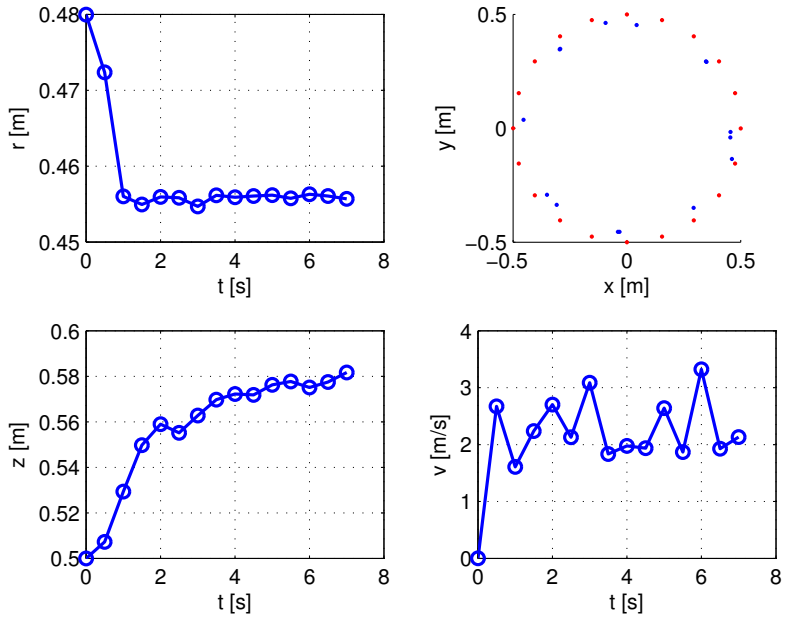


Figure B.6: Evolution of some parameters in test case that concerns the spinning tank

tangential velocity of the boundary which is consistent as the fluid particle is closer to the axis of rotation. However, the velocities calculated are not always equal to the theoretical velocity (according to the distance between the axis of rotation and the particle). These oscillations are mostly due to the changes in the neighbourhood of the particle.

B.3 Animations

Animations have been created for some test cases. In order to visualise them, a temporary website has been created. It can be visited at the following URL: `http://www.sphprogram.p.ht`.

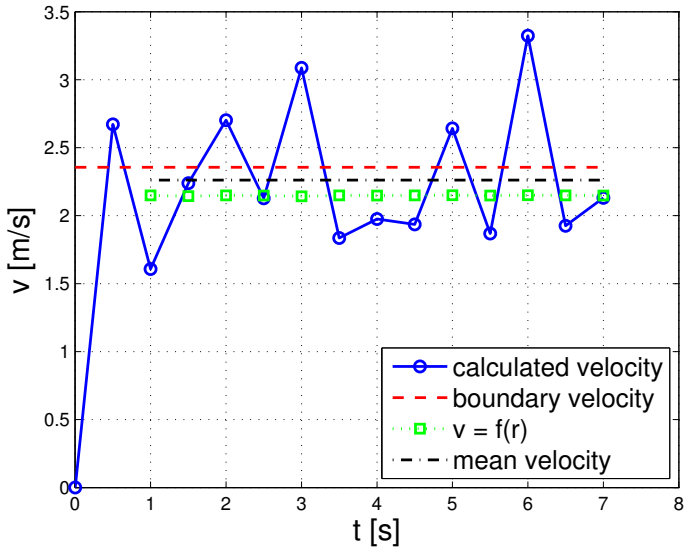


Figure B.7: Tangential velocity of a particle near the boundary

B.4 Source code

B.4.1 Main file

```
!> SPH simulation
!! @n This program is used to solve the Navier-Stokes
   equations
!! using the SPH method. A number of files must be given:
!! paths.txt, *.prm, *.fp and *.mp.
!! @warining The domain must be cubic!
!! @brief Main program to launch an SPH simulation
!! @author Louis Goffin
!! @date 2013-05-26
!! @version 1.0.0
program SPH_simulation
    use SPH_module
    implicit none
    integer :: t1, t2, clock_rate, clock_max
    type (particle_manager) :: manager

    call system_clock ( t1, clock_rate, clock_max )

    call manager%initialisation
    call manager%solver

    call system_clock ( t2, clock_rate, clock_max )
    print *,'Elapsed real time = ', (t2 - t1) / clock_rate
    write(24,*) 'Elapsed real time (s) = ', (t2 - t1) /
       clock_rate

    close(unit=24)

end program SPH_simulation
```

B.4.2 Module file

```
!> SPH module
!! @n Contains all the classes necessary to run a SPH
   simulation.
!! @brief Group of classes (types) and procedures
   definitions.
!! @author Louis Goffin
!! @date 2013-05-26
!! @version 1.0.0
```

```
module SPH_module

    implicit none
    integer , parameter :: prec = 8 !> @param prec sets the
       real precision.
                                    !! 4 = single precision
                                    !! 8 = double precision
    real (kind=prec) , parameter :: pi = 3.14159265359d0

    !> Minimal link class
    !! @n This class is a minimal pointer, i.e. the object
       is composed of only a pointer.
    !! No other information is included. The pointers point
       toward particles
    type min_link
        class (fixed_particle) , pointer :: ptr => null()
                        !< pointer toward a particle
    end type min_link

    !> Link class
    !! @n This class contains a pointer that points toward
       an object
    !! and the distance between 2 particles. This class is
       used to
    !! build vectors of pointers toward objects.
    type link
        class (fixed_particle) , pointer :: ptr => null()
           !< pointer toward a particle
        real (kind=prec) :: r = 0
           !< distance between neighbours
    end type link

    !> List class
    !! @n This class is a list that contains pointers toward
        objects (+distance).
    !! The only problem of this list class is that only link
        objects (ptr+r) can be added
    type list
        integer :: nbr = 0 !< number of elements in the list
        integer :: max_nbr = 0 !< max number of elements in
           the list (size of the array)
```

```
        integer :: incr = 35 !< increment: number of spaces
           to add when the list is full
        type (link) , dimension(:) , allocatable :: lst
                        !< list containing elements

        contains
            !> initialises the list
            procedure initList
            !> adds an element to the list
            procedure addElement
            !> reset the list
            procedure resetList
    end type list


    !> Fixed particle class
    !! @n This class contains a certain number of parameters
        describing
    !! the state of a fixed particle (boundary particle). It
        also
    !! includes the needed procedures to calculate the
       continuity
    !! and some other equations.
    type fixed_particle
        real (kind=prec) , dimension (1:3,1:3)  :: coord
                        !< 3x3 array containing the
                           coordinates of a particle.
                        !! column 1 = currentTime
                        !! column 2 = RK step
                        !! column 3 = nextTime
        real (kind=prec) , dimension (1:3,1:3)  :: speed
                        !< 3x3 array containing the velocity
                            of a particle.
                        !! column 1 = currentTime
                        !! column 2 = RK step
                        !! column 3 = nextTime
        real (kind=prec) , dimension (1:3)      :: rho
                        !< 3x1 array containing the density
                           of a particle.
                        !! element 1 = currentTime
                        !! element 2 = RK step
                        !! element 3 = nextTime
        real (kind=prec) :: m !< mass of the particle
```

```
        real (kind=prec) , dimension (1:3) :: p
                        !< pressure of the particle
                        !! element 1 = currentTime
                        !! element 2 = RK step
                        !! element 3 = nextTime
        real (kind=prec) , dimension (1:3) :: c
                        !< speed of sound of a particle
                        !! element 1 = currentTime
                        !! element 2 = RK step
                        !! element 3 = nextTime
        real (kind=prec) :: h !< smoothing length
        type (list) :: neighbours !< list of neighbours
        integer :: numOfNeighbours !< number of neighbours
        real (kind=prec) , dimension(1:3,1:150) :: vec_gradW
                        !< array that contains the gradient
                           for every
                        !! neighbours; initially set to 150
                           elements to
                        !! increase the computational
                           efficiency
        real (kind=prec) , dimension(1:3,1:150) ::
           vec_gradW_mod
                        !< corrected vec_gradW if asked
        class (particle_manager) , pointer :: manager
                        !< pointer toward the object
                           particle_manager
        real (kind=prec) :: max_mu_ab
                        !< maximum mu_ab of a particle (used
                            for the time step calculation)

        contains
            !> creates the vector neighbours (pointers and
               distance)
            procedure getNeighbours
            !> calculates the pressure p
            procedure calcPressure
            !> calculates the celerity c
            procedure calcCelerity
            !> creates a vector with the values of the
               gradient of the smoothing function:
            !! @f[ \vec{\nabla}_{a} W_{ab} @f]
            procedure gradW
```

```
            !> correction of the gradient of the kernel if
               asked
            procedure kernel_corr
            !> update of rho
            procedure :: varUpdate => varUpdate_fixed
    end type fixed_particle

    !> Mobile particle class
    !! @n This is an extension of the fixed_particle class.
    !! The procedure varUpdtate is overwritten to include
       the update of u and x.
    type , extends (fixed_particle) :: mobile_particle

        contains
            !> update of rho, u and x
            procedure :: varUpdate => varUpdate_mobile
            !> function that calculates the artificial
               viscosity \f$ \Pi_{ab} \f$
            procedure ArtificialViscosity
    end type mobile_particle

    !> Particle sort class
    !! @n This class is able to sort the particles. A grid
       is generated
    !! and the particles are sorted in each cell.
    type particle_sort
        real (kind=prec)     :: h_max !< maximum smoothing
           length
        real (kind=prec)     :: cellSize !< length of a side
           of a cube
        integer              :: nCells=0 !< number of cells
           in the domain
        integer              :: nCellsSide !< number of cells
            on a row
        logical              :: init !< true if the cells
           must be initialised
        type (list) , dimension (:) , allocatable :: storage
                        !< vector of lists that contain
                        !! the particles in a cell
        class (particle_manager) , pointer :: manager
                        !< pointer toward the object
                           particle_manager
```

```
        contains
            !> calculates the maximum smoothing length to
               build a grid
            procedure get_h_max
            !> sets the sizes of the cells, the number of
               cells and it also allocates "storage"
            procedure setCells
            !> sorts the particles in their cell
            procedure particlesSort
    end type particle_sort

    !> Particle manager class
    !! @n This class is used to manage all the particles,
    !! i.e. it contains a reference to every particles,
    !! it contains a number of parameters useful for the
       problem
    !! (variable smoothing length or not, ...), it has a
       solver, etc.
    type particle_manager
        type (particle_sort) :: sorting !< sorting machine
        type (min_link), dimension(:), allocatable :: part
                        !< array of pointers toward
                           particles
        integer :: numFP !< number of fixed particles
        integer :: numMP !< number of mobile particles
        integer :: numPart !< number of particles (FP+MP)
        integer :: kernelKind !< kind of kernel
                              !! 1 = cubic spline kernel
                              !! 2 = quadratic kernel
                              !! 3 = quintic spline kernel
        integer :: kappa !< kappa linked to the eqnstate
        real (kind=prec) :: alpha !< weighting factor in the
            artificial viscosity formulation
        real (kind=prec) :: beta !< weighting factor in the
           artificial viscosity formulation
        integer :: eqnState !< equation of state
                            !! 1 = ideal gas law
                            !! 2 = quasi-incompressible
                               fluid
        integer :: state_gamma !< power in eqnState2.
                               !! often taken around 7
        real (kind=prec) :: molMass !< Molar mass of the
           fluid for the prefect gas law
```

```
        integer :: kernelCorrection !< correction of the
           kernel
                                    !! 0 = no correction
                                    !! 1 = correction
                                       enabled
        real (kind=prec) :: maxTime !< simulation time in
           seconds
        real (kind=prec) :: saveInt !< saving interval
        real (kind=prec) :: h_0 !< initial smoothing length
        real (kind=prec) :: rho_0 !< density of the fluid at
            free surface
        real (kind=prec) :: c_0   !< speed of sound in
           normal conditions
        real (kind=prec) :: timeStep !< time step (not
           constant)
        real (kind=prec) :: currentTime !< current time
        integer :: RKstep !< used to know in which RK
           iteration we are
        real (kind=prec) :: dom_dim
                        !< length of a side of the domain (
                           exterior particle to exterior
                           particle).
                        !! the domain is assumed to be cubic

        contains
            !> initialises the particle manager and the
               particles
            procedure initialisation
            !> solves the problem
            procedure solver
            !> updates the time step
            procedure timeStepUpdate
            !> updates the smoothing length
            procedure slUpdate
    end type particle_manager

    contains

        !> eval_r is a function that calculates the distance
            between two particles
        function eval_r(xyz,xyz2)
            real(kind=prec) , dimension(3) :: xyz
            real(kind=prec) , dimension(3) :: xyz2
```

```
        real(kind=prec) :: eval_r

        eval_r = sqrt(sum((xyz(:) - xyz2(:))*(xyz(:) -
            xyz2(:))))
    end function eval_r

    !> initList is a routine used to initialise a list
       when it is first created
    !! @param this : calling object
    subroutine initList(this)
        class(list) this

        if (this%incr < 1) then
            this%incr = 1
        end if
        this%max_nbr = this%incr
        allocate(this%lst(1:this%max_nbr))
    end subroutine initList

    !> addElement is a routine that adds an element to a
        list. If the list is full, then
    !! it increases the size of the list.
    !! @param this : calling object
    !! @param ptr : pointer of type link to be added in
       the list
    !! @param r : distance between two particles
    subroutine addElement(this,ptr,r)
        class(list) this
        class (fixed_particle), pointer :: ptr
        real (kind=prec) :: r
        type (link), dimension(:), allocatable ::
           temp_lst
                    !< temporary list used when it is
                       necessary
                    !! to increase the size of the
                       existing list

        !< if the list is empty, it must be initialised
        if (this%max_nbr == 0) then
            call this%initList
        !< if the list is full it must be resized
        else if (this%nbr == this%max_nbr) then
            allocate(temp_lst(1:this%max_nbr+this%incr))
```

```
                temp_lst(1:this%max_nbr) = this%lst(1:this%
                    max_nbr)
                call move_alloc(temp_lst,this%lst)
                this%max_nbr = this%max_nbr+this%incr
            end if
            this%nbr = this%nbr + 1
            this%lst(this%nbr)%ptr => ptr
            this%lst(this%nbr)%r = r
        end subroutine addElement

        !> resetList is a routine that reset the list. It
            only sets the number
        !! of elements to 0 but it keeps the maximal size to
            max_nbr
        !! @param this : calling object
        subroutine resetList(this)
            class(list) this

            this%nbr = 0
        end subroutine resetList

        !> getNeighbours is a routine that questions the
            particle_sort object
        !! to get the particles in the neighbouring cells.
            The distance is
        !! calculated and the neighbours selected.
        !! @param this : calling object
        subroutine getNeighbours(this)
            class(fixed_particle), target :: this
            real (kind=prec), dimension(:), pointer,
                contiguous :: xyz
                        !< position of the particle
            integer :: xCell,yCell,zCell !< number of the
                cell according to x, y and z
            integer :: nCellsSide !< number of cells on a
                row of the domain
            integer :: i,j,k !< loop counter
            integer, dimension(1:27) :: cellsToCheck
                        !< number of the cells to check for
                            the neighbours
            real (kind=prec), dimension(:), pointer,
                contiguous :: neighXYZ
                        !< coordinates of a neighbour
```

```
            real (kind=prec) :: r !< distance between two
               particles
            class (fixed_particle) , pointer :: cur_ptr
                        !< current pointer toward a particle
280         type (link) , pointer :: cur_neigh
                        !< current list of neighbours
            type (particle_sort) , pointer :: srt
                        !< pointer toward the sorting
                           machine
            type (list) , pointer :: storage
285                     !< pointer toward the storage
            integer , pointer :: cur_RKstep
                        !< pointer toward the RKstep

            !> pointer initialisation
290         srt => this%manager%sorting
            cur_RKstep => this%manager%RKstep

            !> coordinates of the particle
            xyz => this%coord(:,cur_RKstep)
295
            if (cur_RKstep == 1) then
                !> calculates the number of the cell in
                   which the particle is
                nCellsSide = srt%nCellsSide
                if (xyz(1) == this%manager%dom_dim) then
300                 xCell = nCellsSide
                else
                    xCell = nint((xyz(1)-mod(xyz(1),srt%
                       cellSize))/srt%cellSize) + 1
                end if
                if (xyz(2) == this%manager%dom_dim) then
305                 yCell = nCellsSide
                else
                    yCell = nint((xyz(2)-mod(xyz(2),srt%
                       cellSize))/srt%cellSize) + 1
                end if
                if (xyz(3) == this%manager%dom_dim) then
310                 zCell = nCellsSide
                else
                    zCell = nint((xyz(3)-mod(xyz(3),srt%
                       cellSize))/srt%cellSize) + 1
                end if
```

```
315                 !> calculates the number of the neighbouring
                        cells
                    do i=-1,1
                        do j=-1,1
                            do k=-1,1
                                if ((xCell+i > 0) .and. (yCell+j
                                    > 0) .and. (zCell + k > 0) .
                                   and. (xCell+i <= nCellsSide)
                                   .and. &
320                                     (yCell+j <= nCellsSide)
                                        .and. (zCell+k <=
                                        nCellsSide)) then
                                    cellsToCheck((i+1)*9+(j+1)
                                       *3+(k+2)) =&
                                        (xCell+i-1)*nCellsSide
                                        **2 + (yCell+j-1)*
                                        nCellsSide + (zCell+k
                                        )
                                else
                                    cellsToCheck((i+1)*9+(j+1)
                                       *3+(k+2)) = 0
325                             end if
                            end do
                        end do
                    end do
                end if
330
                !> stores the neighbours of the particle in the
                   neighbours list.
                !! First, the list is reseted, then the
                   neighbouring cells are scanned.
                !! In each cell, the distance between the two
                   particles is calculated.
                !! If it is lower than the support domain and it
                    is not the particle we
335             !! are working with (r > 0 but here r > 1E-12
                   for numerical errors), an element
                !! link (ptr + r) is added to the neighbours
                   list.
                !! For the second RK step, only the distances r
                   are recalculated. It is assumed that
```

```
            !! the neighbours remain the same between 2 RK
               step.
            if (cur_RKstep == 1) then
340             call this%neighbours%resetList
                do i=1,27
                    if (cellsToCheck(i) > 0) then
                        storage => srt%storage(cellsToCheck(
                           i))
                        do j=1,srt%storage(cellsToCheck(i))%
                           nbr
345                         cur_ptr => storage%lst(j)%ptr
                            neighXYZ => cur_ptr%coord(:,
                               cur_RKstep)
                            r = eval_r(xyz,neighXYZ)
                            if ((r <= this%manager%kappa*
                               this%h) .and. (r > 1E-12))
                               then
                                call this%neighbours%
                                   addElement(cur_ptr,r)
350                         end if
                        end do
                    end if
                end do
                this%numOfNeighbours = this%neighbours%nbr
355         else
                do i=1,this%numOfNeighbours
                    cur_neigh => this%neighbours%lst(i)
                    neighXYZ => cur_neigh%ptr%coord(:,
                       cur_RKstep)
                    cur_neigh%r = eval_r(xyz,neighXYZ)
360             end do
            end if
        end subroutine getNeighbours

        !> calcPressure is a function that calculates the
           pressure according
365     !! to the equation of state chosen.
        !! @param this : calling object
        !! @param rho : actual density
        function calcPressure(this,rho)
            class(fixed_particle) this
370         real (kind=prec) :: rho
            real (kind=prec) :: calcPressure
```

```
            real (kind=prec) , parameter :: idealGasCst =
               8.3144621d0
            real (kind=prec) :: B

375         if (this%manager%eqnState == 1) then !< 1 =
               ideal gas law at 20 degrees C
                calcPressure = (rho/this%manager%rho_0-1)*
                   idealGasCst*293.15d0/this%manager%molMass
            else if (this%manager%eqnState == 2) then !< 2 =
                quasi-incompressible fluid
                B = this%manager%c_0**2.d0*this%manager%
                   rho_0/this%manager%state_gamma
                calcPressure = B*((rho/this%manager%rho_0)**
                   this%manager%state_gamma-1)
380         end if
        end function calcPressure

        !> calcCelerity is a function that calculates the
           celerity according
        !! to the equation of state chosen. The equation
           used is
385     !! @f[ c = \sqrt{\frac{dp}{d\rho}} @f]
        !! @param this : calling object
        !! @param rho : actual density
        function calcCelerity(this,rho)
            class(fixed_particle) this
390         real (kind=prec) :: rho
            real (kind=prec) :: calcCelerity

            if (this%manager%eqnState == 1) then !< 1 =
               considering the ideal gas law at 20 degrees C
                calcCelerity = this%manager%c_0
395         else if (this%manager%eqnState == 2) then !< 2 =
                considering a quasi-incompressible fluid
                calcCelerity = this%manager%c_0*((rho/this%
                   manager%rho_0)**(this%manager%state_gamma
                   -1))**0.5d0
            end if
        end function calcCelerity

400     !> gradW is a routine used to create a vector that
           contains the values
        !! of the gradient for each neighbour.
```

```
        !! @param this : calling object
        subroutine gradW(this)
            class(fixed_particle) , target :: this
            integer :: i
            real (kind=prec) :: alpha_d !< normalisation
               coefficient
            real (kind=prec) , pointer :: r !< distance
               between a particle and a neighbour
            class (fixed_particle) , pointer :: cur_neigh !<
                pointer toward a neighbour
            real (kind=prec) :: cur_h !< value of h
            integer , pointer :: cur_RKstep !< pointer
               toward the current RK step

            if (this%numOfNeighbours > 150) then
                print *,'Error: Number of neighbours greater
                   than expected (max 150 for vec_gradW): '
                   , this%numOfNeighbours
                write(24,*) 'Error: Number of neighbours
                   greater than expected (max 150 for
                   vec_gradW): ', this%numOfNeighbours
                stop
            end if

            cur_h = this%h
            cur_RKstep => this%manager%RKstep

            if (this%manager%kernelKind == 1) then !< cubic
               spline
                alpha_d = 3.d0/(2.d0*pi*cur_h**3.d0)
                do i=1,this%numOfNeighbours
                    r => this%neighbours%lst(i)%r
                    cur_neigh => this%neighbours%lst(i)%ptr
                    if ((r/cur_h >= 0.d0).and.(r/cur_h < 1.
                       d0)) then
                        this%vec_gradW(:,i) = alpha_d/cur_h
                           * (3.d0/2.d0*(r/cur_h)**2.d0 - 2.
                           d0*(r/cur_h)) * &
                        (this%coord(:,cur_RKstep)-cur_neigh%
                           coord(:,cur_RKstep))/r
                    else if ((r/cur_h >= 1.d0).and.(r/cur_h
                       < 2.d0)) then
```

```
430                         this%vec_gradW(:,i) = alpha_d/cur_h
                               * (-0.5d0 * (2.d0 - r/cur_h)**2.
                               d0) * &
                            (this%coord(:,cur_RKstep)-cur_neigh%
                               coord(:,cur_RKstep))/r
                        else
                            this%vec_gradW(:,i) = 0.d0
                        end if
435                 end do
                elseif (this%manager%kernelKind == 2) then !<
                    quadratic
                    alpha_d = 5.d0/(4.d0*pi*cur_h**3.d0)
                    do i=1,this%numOfNeighbours
                        r => this%neighbours%lst(i)%r
440                     cur_neigh => this%neighbours%lst(i)%ptr
                        if ((r/cur_h >= 0.d0).and.(r/cur_h <= 2.
                           d0)) then
                            this%vec_gradW(:,i) = alpha_d/cur_h
                               * (3.d0/8.d0*r/cur_h - 3.d0/4.d0)
                                * &
                            (this%coord(:,cur_RKstep)-cur_neigh%
                               coord(:,cur_RKstep))/r
                        else
445                         this%vec_gradW(:,i) = 0.d0
                        end if
                    end do
                elseif (this%manager%kernelKind == 3) then !<
                    quintic spline
                    alpha_d = 3.d0/(359.d0*pi*cur_h**3.d0)
450                 do i=1,this%numOfNeighbours
                        r => this%neighbours%lst(i)%r
                        cur_neigh => this%neighbours%lst(i)%ptr
                        if ((r/cur_h >= 0.d0).and.(r/cur_h < 1.
                           d0)) then
                            this%vec_gradW(:,i) = alpha_d/cur_h
                               * (-5.d0*(3.d0-r/cur_h)**4.d0+30.
                               d0*(2.d0-r/cur_h)**4.d0-75.d0*&
455                            (1.d0-r/cur_h)**4.d0)*(this%coord(:,
                               cur_RKstep)-cur_neigh%coord(:,
                               cur_RKstep))/r
                        else if ((r/cur_h >= 1.d0).and.(r/cur_h
                           < 2.d0)) then
```

```
                        this%vec_gradW(:,i) = alpha_d/cur_h
                           * (-5.d0*(3.d0-r/cur_h)**4.d0+30.
                           d0*(2.d0-r/cur_h)**4.d0) * &
                        (this%coord(:,cur_RKstep)-cur_neigh%
                           coord(:,cur_RKstep))/r
                    else if ((r/cur_h >= 2.d0).and.(r/cur_h
                       < 3.d0)) then
460                     this%vec_gradW(:,i) = alpha_d/cur_h
                           * (-5.d0*(3.d0-r/cur_h)**4.d0) *
                           &
                        (this%coord(:,cur_RKstep)-cur_neigh%
                           coord(:,cur_RKstep))/r
                    else
                        this%vec_gradW(:,i) = 0.d0
                    end if
465             end do
            end if
            if (this%manager%kernelCorrection == 1) then
                this%vec_gradW_mod = this%vec_gradW
            end if
470     end subroutine gradW

        !> kernel_corr is a routine that takes into account
           the fact that the kernel may be truncated.
        !! It corrects the gradient of the kernel
        !! @param this : calling object
475     subroutine kernel_corr(this)
            class(fixed_particle) this
            real (kind=prec) , dimension(3,3) :: M !< matrix
                used to correct the kernel gradient
            real (kind=prec) , dimension(3,3) :: L !<
               inverse of the matrix used to correct the
               kernel gradient
            real (kind=prec) :: detM !< determinant of M
480         integer :: i !< loop counter
            real (kind=prec) :: MDivRho !< m_b/rho_b
            class(fixed_particle) , pointer :: cur_neigh !<
               pointer toward the current neighbour
            integer , pointer :: cur_RKstep !< current RK
               step

485         cur_RKstep => this%manager%RKstep
            M(1,1) = 0.d0
```

```
                M(2,2) = 0.d0
                M(3,3) = 0.d0
                M(1,2) = 0.d0
490             M(1,3) = 0.d0
                M(2,3) = 0.d0
                do i=1,this%numOfNeighbours
                    cur_neigh => this%neighbours%lst(i)%ptr
                    MDivRho = cur_neigh%m/cur_neigh%rho(
                       cur_RKstep)
495                 M(1,1) = M(1,1) + MDivRho*(cur_neigh%coord
                       (1,cur_RKstep)-this%coord(1,cur_RKstep))*
                       this%vec_gradW(1,i)
                    M(2,2) = M(2,2) + MDivRho*(cur_neigh%coord
                       (2,cur_RKstep)-this%coord(2,cur_RKstep))*
                       this%vec_gradW(2,i)
                    M(3,3) = M(3,3) + MDivRho*(cur_neigh%coord
                       (3,cur_RKstep)-this%coord(3,cur_RKstep))*
                       this%vec_gradW(3,i)
                    M(1,2) = M(1,2) + MDivRho*(cur_neigh%coord
                       (1,cur_RKstep)-this%coord(1,cur_RKstep))*
                       this%vec_gradW(2,i)
                    M(1,3) = M(1,3) + MDivRho*(cur_neigh%coord
                       (1,cur_RKstep)-this%coord(1,cur_RKstep))*
                       this%vec_gradW(3,i)
500                 M(2,3) = M(2,3) + MDivRho*(cur_neigh%coord
                       (2,cur_RKstep)-this%coord(2,cur_RKstep))*
                       this%vec_gradW(3,i)
                end do
                M(2,1) = M(1,2) !< M is symmetric
                M(3,1) = M(1,3) !< M is symmetric
                M(3,2) = M(2,3) !< M is symmetric
505
                detM = M(1,1)*(M(2,2)*M(3,3)-M(3,2)*M(2,3))-M
                   (1,2)*(M(2,1)*M(3,3)-M(3,1)*M(2,3))+ &
                M(1,3)*(M(2,1)*M(3,2)-M(3,1)*M(2,2))
                L(1,1) = M(2,2)*M(3,3)-M(3,2)*M(2,3)
                L(2,2) = M(1,1)*M(3,3)-M(3,1)*M(1,3)
510             L(3,3) = M(1,1)*M(2,2)-M(2,1)*M(1,2)
                L(1,2) = M(3,1)*M(2,3)-M(2,1)*M(3,3)
                L(2,1) = L(1,2) !< the inverse of a symmetric
                   matrix is symmetric
                L(1,3) = M(2,1)*M(3,2)-M(3,1)*M(2,2)
```

```
            L(3,1) = L(1,3) !< the inverse of a symmetric
                matrix is symmetric
            L(2,3) = M(3,1)*M(1,2)-M(1,1)*M(3,2)
            L(3,2) = L(2,3) !< the inverse of a symmetric
                matrix is symmetric
            L = (1.d0/detM)*L

            do i=1,this%numOfNeighbours
                this%vec_gradW_mod(1,i) = L(1,1)*this%
                    vec_gradW(1,i) + L(1,2)*this%vec_gradW(1,
                    i) + L(1,3)*this%vec_gradW(3,i)
                this%vec_gradW_mod(2,i) = L(2,1)*this%
                    vec_gradW(1,i) + L(2,2)*this%vec_gradW(2,
                    i) + L(2,3)*this%vec_gradW(3,i)
                this%vec_gradW_mod(3,i) = L(3,1)*this%
                    vec_gradW(1,i) + L(3,2)*this%vec_gradW(2,
                    i) + L(3,3)*this%vec_gradW(3,i)
            end do
        end subroutine kernel_corr

        !> varUpdate_fixed is a routine used to update the
            density of a fixed particle.
        !! The update of the velocity et the position are
            not performed.
        !! The integration scheme is a RK22 scheme.
        !! @param this : calling object
        subroutine varUpdate_fixed(this)
            class(fixed_particle) this
            real (kind=prec) :: Delta_rho !< \f$ d\rho/dt \
                f$
            real (kind=prec) , dimension(1:3) :: u_ab !<
                relative velocity between the particle and a
                neighbour
            integer :: i !< loop counter
            integer , pointer :: cur_RKstep !< pointer
                toward the value of the current RK step
            class (fixed_particle) , pointer :: cur_neigh !<
                 current neighbour
            Delta_rho = 0.d0
            cur_RKstep => this%manager%RKstep

            call this%getNeighbours
            call this%gradW
```

```
            do i=1,this%numOfNeighbours
                cur_neigh => this%neighbours%lst(i)%ptr
                u_ab(:) = this%speed(:,cur_RKstep) -
                    cur_neigh%speed(:,cur_RKstep)
                Delta_rho = Delta_rho + this%m*dot_product(
                    u_ab,this%vec_gradW(:,i))
            end do

            if (cur_RKstep == 1) then !< 1st RK step
                this%rho(2) = this%rho(1) + Delta_rho*this%
                    manager%timeStep
                this%rho(3) = this%rho(1) + Delta_rho*this%
                    manager%timeStep/2.d0
                this%speed(:,2) = this%speed(:,1)
                this%coord(:,2) = this%coord(:,1)
                this%p(2) = this%calcPressure(this%rho(2))
                this%c(2) = this%calcCelerity(this%rho(2))
            else !< 2nd RK step
                this%rho(3) = this%rho(3) + Delta_rho*this%
                    manager%timeStep/2.d0
                this%speed(:,3) = this%speed(:,2)
                this%coord(:,3) = this%coord(:,2)
                this%p(3) = this%calcPressure(this%rho(3))
                this%c(3) = this%calcCelerity(this%rho(3))
            end if
        end subroutine varUpdate_fixed

        !> varUpdate_mobile is a routine that updates the
            density, the velocity
        !! and the position of a mobile particle.
        !! The integration scheme is a RK22 scheme.
        !! @param this : calling object
        subroutine varUpdate_mobile(this)
            class(mobile_particle) this
            real (kind=prec) :: Delta_rho !< \f$ d\rho/dt \
                f$
            real (kind=prec) , dimension(1:3) :: Delta_u !<
                \f$ du/dt \f$
            real (kind=prec) , dimension(1:3) :: Delta_x !<
                \f$ dx/dt \f$
            real (kind=prec) , dimension(1:3) :: u_ab !<
                relative velocity between the particle and a
                neighbour
```

```
                real (kind=prec) , dimension(1:3) :: F !< Volume
                    forces
575             real (kind=prec) :: pi_ab !< Artificial
                   viscosity
                integer :: i !< loop counter
                integer , pointer :: cur_RKstep !< pointer
                   toward the value of the current RK step
                class (fixed_particle) , pointer :: cur_neigh !<
                    current neighbour
                Delta_rho = 0.d0
580             Delta_u = (/0.d0,0.d0,0.d0/)
                Delta_x = (/0.d0,0.d0,0.d0/)
                F = (/real(0,prec),real(0,prec),real(-9.81,prec)
                   /)
                cur_RKstep => this%manager%RKstep

585             call this%getNeighbours
                call this%gradW
                if (this%manager%kernelCorrection == 1) then
                    call this%kernel_corr
                end if
590             !> reset max_mu_ab
                if (cur_RKstep == 1) then
                    this%max_mu_ab = 0.d0
                end if
                if (this%manager%kernelCorrection == 1) then
595                 do i=1,this%numOfNeighbours
                        cur_neigh => this%neighbours%lst(i)%ptr
                        u_ab(:) = this%speed(:,cur_RKstep) -
                           cur_neigh%speed(:,cur_RKstep)
                        pi_ab = this%ArtificialViscosity(
                           cur_neigh,this%manager%alpha,this%
                           manager%beta)
                        Delta_rho = Delta_rho + this%m*
                           dot_product(u_ab,this%vec_gradW(:,i))
600                     Delta_u = Delta_u + this%m*(cur_neigh%p(
                           cur_RKstep)/&
                        cur_neigh%rho(cur_RKstep)**2.d0+this%p(
                           cur_RKstep)/&
                        this%rho(cur_RKstep)**2.d0+pi_ab)*this%
                           vec_gradW_mod(:,i)
                    end do
                else
```

```
605                 do i=1,this%numOfNeighbours
                        cur_neigh => this%neighbours%lst(i)%ptr
                        u_ab(:) = this%speed(:,cur_RKstep) -
                            cur_neigh%speed(:,cur_RKstep)
                        pi_ab = this%ArtificialViscosity(
                            cur_neigh,this%manager%alpha,this%
                            manager%beta)
                        Delta_rho = Delta_rho + this%m*
                            dot_product(u_ab,this%vec_gradW(:,i))
610                     Delta_u = Delta_u + this%m*(cur_neigh%p(
                            cur_RKstep)/&
                        cur_neigh%rho(cur_RKstep)**2.d0+this%p(
                            cur_RKstep)/&
                        this%rho(cur_RKstep)**2.d0+pi_ab)*this%
                            vec_gradW(:,i)
                    end do
                end if
615             Delta_u = -Delta_u+F

                if (cur_RKstep == 1) then !< 1st RK step
                    this%rho(2) = this%rho(1) + Delta_rho*this%
                        manager%timeStep
                    this%rho(3) = this%rho(1) + Delta_rho*this%
                        manager%timeStep/2.d0
620                 this%speed(:,2) = this%speed(:,1) + Delta_u*
                        this%manager%timeStep
                    this%speed(:,3) = this%speed(:,1) + Delta_u*
                        this%manager%timeStep/2.d0
                    this%coord(:,2) = this%coord(:,1) + this%
                        speed(:,1)*this%manager%timeStep
                    this%coord(:,3) = this%coord(:,1) + this%
                        speed(:,1)*this%manager%timeStep/2.d0
                    this%p(2) = this%calcPressure(this%rho(2))
625                 this%c(2) = this%calcCelerity(this%rho(2))
                else !< 2nd RK step
                    this%rho(3) = this%rho(3) + Delta_rho*this%
                        manager%timeStep/2.d0
                    this%speed(:,3) = this%speed(:,3) + Delta_u*
                        this%manager%timeStep/2.d0
                    this%coord(:,3) = this%coord(:,3) + this%
                        speed(:,2)*this%manager%timeStep/2.d0
630                 this%p(3) = this%calcPressure(this%rho(3))
                    this%c(3) = this%calcCelerity(this%rho(3))
```

```
        end if
    end subroutine varUpdate_mobile

!> The function ArtificialViscosity calculates the
   viscosity term
!! in the momentum equation.
!! @param this : calling object
!! @param neighObj : neighbouring object
!! @param alpha : coefficicent in the artificial
   viscosity formulation
!! @param beta : coefficicent in the artificial
   viscosity formulation
    function ArtificialViscosity(this,neighObj,alpha,
       beta)
        class(mobile_particle) this
        class(fixed_particle) neighObj
        real (kind=prec) :: alpha
        real (kind=prec) :: beta
        real (kind=prec) :: ArtificialViscosity

        real (kind=prec) :: mu_ab = 0.d0 !< this term
           represents a kind of viscosity
        real (kind=prec) , dimension(1:3) :: u_ab !<
           relative velocity of a in comparison with b
        real (kind=prec) , dimension(1:3) :: x_ab !< the
            distance between a and b
        real (kind=prec) :: c_ab !< mean speed of sound
        real (kind=prec) :: rho_ab !< mean density

        u_ab = this%speed(:,this%manager%RKstep)-
           neighObj%speed(:,this%manager%RKstep)
        x_ab = this%coord(:,this%manager%RKstep)-
           neighObj%coord(:,this%manager%RKstep)
        if (dot_product(u_ab,x_ab)<0) then
            !> mu_ab is calculated using \f[ \mu_{ab} =
               \frac{h \vec{u}_{ab} \cdot \vec{x}_{ab
               }}{\vec{x}_{ab}^2+\eta^2} \f]
            mu_ab = this%h*dot_product(u_ab,x_ab)/(
               dot_product(x_ab,x_ab)+0.01d0*this%h**2.
               d0)
            c_ab = 0.5d0*(this%c(this%manager%RKstep)+
               neighObj%c(this%manager%RKstep))
```

```
660                 rho_ab = 0.5d0*(this%rho(this%manager%RKstep
                        )+neighObj%rho(this%manager%RKstep))
                    ArtificialViscosity = (-alpha*c_ab*mu_ab+
                        beta*mu_ab**2.d0)/rho_ab
                else
                    ArtificialViscosity = 0.d0
                end if
665
                !> update of max_mu_ab for the calculation of
                    the time step
                if ((this%manager%RKstep == 1) .and. (mu_ab >
                    this%max_mu_ab)) then
                    this%max_mu_ab = mu_ab
                end if
670         end function ArtificialViscosity

            !> initialisation is a subroutine that initialises
                the particle manager
            !! and all the particles. The routine takes the
                datas from external files.
            !! The files paths are given in a file saved in the
                same directory as the program.
675         !! The external files contains: the parameters, the
                fixed particles properties
            !! and the mobile particles.
            !! @param this : calling object
            subroutine initialisation(this)
                class(particle_manager),target :: this
680             character (250) :: param_path   !< path of the
                    parameters file
                character (250) :: fp_path      !< path of the
                    fixed particle (fp) file
                character (250) :: mp_path      !< path of the
                    mobile particle (mp) file
                real (kind=prec) :: x,y,z       !< coordinates
                    of a particle
                real (kind=prec) :: u_x,u_y,u_z !< velocity of a
                     particle
685             real (kind=prec) :: rho,m       !< density and
                    mass of a particle
                integer :: i                    !< loop counter
                class(fixed_particle), pointer :: cur_ptr
```

```
                this%timeStep = 1E-15    !< initial time step
690             this%currentTime = 0.d0 !< current time
                   initialisation
                this%RKstep = 1          !< RK step counter
                   initialisation


                !> Reading of the paths of the input files
695             open(unit=1,file='paths.txt')
                read(1,*) param_path
                read(1,*) fp_path
                read(1,*) mp_path

700             !> Reading and storing of the datas in the
                   parameter files
                open(unit=2,file=trim(param_path))
                read(2,*) this%numFP
                read(2,*) this%numMP
                read(2,*) this%h_0
705             read(2,*) this%c_0
                read(2,*) this%rho_0
                read(2,*) this%dom_dim
                read(2,*) this%kernelKind
                read(2,*) this%alpha
710             read(2,*) this%beta
                read(2,*) this%eqnState
                read(2,*) this%state_gamma
                read(2,*) this%molMass
                read(2,*) this%kernelCorrection
715             read(2,*) this%maxTime
                read(2,*) this%saveInt
                this%numPart = this%numFP + this%numMP
                allocate(this%part(1:this%numPart)) !<
                   allocation of the particles array
                select case (this%kernelKind)
720                 case (1)
                        this%kappa = 2
                    case (2)
                        this%kappa = 2
                    case (3)
725                     this%kappa = 3
                end select
```

```
            !> Reading and storing of the data for the fixed
                particles
            open(unit=3,file=trim(fp_path))
            do i=1,this%numFP
                read(3,*) x,y,z,u_x,u_y,u_z,rho,m
                allocate(fixed_particle :: this%part(i)%ptr)
                cur_ptr => this%part(i)%ptr
                cur_ptr%coord = 0.d0
                cur_ptr%coord(1,1) = x
                cur_ptr%coord(2,1) = y
                cur_ptr%coord(3,1) = z
                cur_ptr%speed = 0.d0
                cur_ptr%speed(1,1) = u_x
                cur_ptr%speed(2,1) = u_y
                cur_ptr%speed(3,1) = u_z
                cur_ptr%rho = 0.d0
                cur_ptr%rho(1) = rho
                cur_ptr%m = m
                cur_ptr%h = this%h_0
                cur_ptr%manager => this
                cur_ptr%p = 0.d0
                cur_ptr%p(1) = cur_ptr%calcPressure(cur_ptr%
                   rho(1))
                cur_ptr%c = 0.d0
                cur_ptr%c(1) = cur_ptr%calcCelerity(cur_ptr%
                   rho(1))
                call cur_ptr%neighbours%initList
            end do

            !> Reading and storing of the data for the
               mobile particles
            open(unit=4,file=trim(mp_path))
            do i=1,this%numMP
                read(4,*) x,y,z,u_x,u_y,u_z,rho,m
                allocate(mobile_particle :: this%part(this%
                   numFP+i)%ptr)
                cur_ptr => this%part(this%numFP+i)%ptr
                cur_ptr%coord = 0.d0
                cur_ptr%coord(1,1) = x
                cur_ptr%coord(2,1) = y
                cur_ptr%coord(3,1) = z
                cur_ptr%speed = 0.d0
                cur_ptr%speed(1,1) = u_x
```

```
                cur_ptr%speed(2,1) = u_y
                cur_ptr%speed(3,1) = u_z
                cur_ptr%rho = 0.d0
                cur_ptr%rho(1) = rho
                cur_ptr%m = m
                cur_ptr%h = this%h_0
                cur_ptr%manager => this
                cur_ptr%p = 0.d0
                cur_ptr%p(1) = cur_ptr%calcPressure(cur_ptr%
                    rho(1))
                cur_ptr%c = 0.d0
                cur_ptr%c(1) = cur_ptr%calcCelerity(cur_ptr%
                    rho(1))
                call cur_ptr%neighbours%initList
            end do

            !> Files closing
            close(unit=1)
            close(unit=2)
            close(unit=3)
            close(unit=4)

            !> Particle sort
            this%sorting%manager => this
            this%sorting%init = .true.

            !> creation of the log file
            open(unit=24,file='log.txt',form='formatted',
                access='stream',status='replace')
            write(24,*) 'Initialisation finished.'
            print *,'Initialisation finished.'
        end subroutine initialisation

        !> The solver routine is used to solve the problem.
            It loops over time and uses
        !! a RK22 time integration scheme.
        !! @param this : calling object
        subroutine solver(this)
            class(particle_manager) this
            integer :: i,j
            integer :: ite      !< iteration counter
            logical :: to_save  !< saving flag. If true a
                saving is done
```

```
805             ite = 0

                open(unit=22,file='results.out', form='
                   unformatted',access='stream',status='replace'
                   )
                open(unit=23,file='time.out', form='unformatted'
                   ,access='stream',status='replace')

810             do while (this%currentTime <= this%maxTime)
                    !> Time increment and saving status
                    if ((floor(this%currentTime/this%saveInt) /=
                        &
                    floor((this%currentTime+this%timeStep)/this%
                       saveInt)) .or. ite == 0) then
                        to_save = .true.
815                 end if
                    this%currentTime = this%currentTime + this%
                       timeStep
                    !> Runge Kutta loop
                    do j=1,2
                        this%RKstep=j
820                     call this%sorting%particlesSort
                        !> Loop over the particles
                        !$OMP PARALLEL DO PRIVATE(i) SCHEDULE(
                           DYNAMIC)
                        do i=1,this%numPart
                            call this%part(i)%ptr%varUpdate
825                     end do
                        !$OMP END PARALLEL DO
                    end do
                    !> Update of the current time variables (
                       currentTime = nextTime)
                    do i=1,this%numPart
830                     this%part(i)%ptr%rho(1) = this%part(i)%
                           ptr%rho(3)
                        this%part(i)%ptr%p(1) = this%part(i)%ptr
                           %p(3)
                        this%part(i)%ptr%c(1) = this%part(i)%ptr
                           %c(3)
                        this%part(i)%ptr%speed(:,1) = this%part(
                           i)%ptr%speed(:,3)
```

```
                this%part(i)%ptr%coord(:,1) = this%part(
                    i)%ptr%coord(:,3)
835         end do

            !> Test for the data saving
            if (to_save) then
                do i=1,this%numMP
840                 write(unit=22) real(this%part(this%
                        numFP+i)%ptr%coord(1,1),4)
                                    !> x saving
                    write(unit=22) real(this%part(this%
                        numFP+i)%ptr%coord(2,1),4)
                                    !> y saving
                    write(unit=22) real(this%part(this%
                        numFP+i)%ptr%coord(3,1),4)
845                                 !> z saving
                    write(unit=22) real(this%part(this%
                        numFP+i)%ptr%speed(1,1),4)
                                    !> u_x saving
                    write(unit=22) real(this%part(this%
                        numFP+i)%ptr%speed(2,1),4)
                                    !> u_y saving
850                 write(unit=22) real(this%part(this%
                        numFP+i)%ptr%speed(3,1),4)
                                    !> u_z saving
                    write(unit=22) real(this%part(this%
                        numFP+i)%ptr%rho(1),4)
                                    !> rho saving
                    write(unit=22) real(this%part(this%
                        numFP+i)%ptr%p(1),4)
855                                 !> pressure saving
                end do
                write(unit=23) real(this%currentTime,4)
                            !> current time saving
                print *,'Iteration␣nb␣',ite
860             print *,'␣␣␣Time␣(s)␣=␣',this%
                    currentTime
                print *,'␣␣␣Time␣step␣(s)␣=␣',this%
                    timeStep
                write(24,*) 'Iteration␣nb␣',ite
                write(24,*) '␣␣␣Time␣(s)␣=␣',this%
                    currentTime
```

```
                    write(24,*) '␣␣␣Time␣step␣(s)␣=␣',this%
                        timeStep
865                 to_save = .false.
                end if
                call this%timeStepUpdate
                call this%slUpdate
                ite = ite + 1
870         end do
            close(unit=22)
            close(unit=23)
        end subroutine solver

875     !> timeStepUpdae is a routine that computes the next
            time step using
        !! the properties of the particles.
        !! @param this : calling object
        subroutine timeStepUpdate(this)
            class(particle_manager) this
880         real (kind=prec) :: dTf , dTftemp   !< time step
                relative to the body forces
            real (kind=prec) :: dTcv , dTcvtemp !< time step
                relative to the viscous forces and Courrant
                number
            integer :: i
            class (fixed_particle) , pointer :: cur_ptr

885         !> computes the time step relative to the body
                forces
            dTf = sqrt(this%part(this%numFP+1)%ptr%h/9.81d0)
            do i=this%numFP+2,this%numPart
                cur_ptr => this%part(i)%ptr
                dTftemp = sqrt(cur_ptr%h/9.81d0)
890             if (dTftemp < dTf) then
                    dTf = dTftemp
                end if
            end do

895         !> computes the time step relative to the CN and
                the viscous forces
            dTcv = this%part(this%numFP+1)%ptr%h/(this%part(
                this%numFP+1)%ptr%c(1) + 0.6d0* &
            (this%alpha*this%part(this%numFP+1)%ptr%c(1) +
                this%beta*this%part(this%numFP+1)%ptr%
```

```
                max_mu_ab))
            do i=this%numFP+2,this%numPart
                cur_ptr => this%part(i)%ptr
900             dTcvtemp = cur_ptr%h/(cur_ptr%c(1) + 0.6d0*(
                    this%alpha*cur_ptr%c(1) + this%beta*
                    cur_ptr%max_mu_ab))
                if (dTcvtemp < dTcv) then
                    dTcv = dTcvtemp
                end if
            end do
905
            !> computes the final time step
            if (0.4d0*dTf > 0.25d0*dTcv) then
                this%timeStep = 0.25d0*dTcv
            else
910             this%timeStep = 0.4d0*dTf
            end if

            !> possibility to change the time step if we use
                the ideal gas law
            if (this%eqnState == 1) then
915             this%timeStep = 5.d0*this%timeStep
            end if
        end subroutine timeStepUpdate

        !> slUpdate is a routine that updates the smoothing
            length at each time step.
920     !! It is written to provide the same smoothing
            length for every particle.
        !! @param this : calling object
        subroutine slUpdate(this)
            class(particle_manager) this
            real (kind=prec) :: mean_rho !< mean value of
                the densities of the mobile particles
925         real (kind=prec) :: new_h    !< new smoothing
                length
            integer :: i

            mean_rho = 0.d0
            !> calculation of the average density
930         do i=1,this%numPart
                mean_rho = mean_rho + this%part(i)%ptr%rho
                    (1)
```

```
            end do
            mean_rho = mean_rho/this%numPart
            !> calculation of the new smoothing length
            new_h = this%h_0*(this%rho_0/mean_rho)**(1.d0/3.
               d0)
            !> if the smoothing length is greater than 0.5
               the size of a cell, h is limited
            if (new_h > 0.5d0*this%sorting%cellSize) then
                new_h = 0.5d0*this%sorting%cellSize
                print*,'Warning: the smoothing has been
                   limited'
                write(24,*) 'Warning: the smoothing has been
                   limited'
            end if
            !> update of the smoothing length
            do i=1,this%numPart
                this%part(i)%ptr%h = new_h
            end do
        end subroutine slUpdate

        !> get_h_max is a routine used to find the largest
           smoothing length of the particles.
        !! This is useful when h is not constant over the
           particles.
        !! @param this : calling object
        subroutine get_h_max(this)
            class(particle_sort) this
            integer :: i

            this%h_max = 0
            do i=1,this%manager%numPart
                if (this%manager%part(i)%ptr%h > this%h_max)
                    then
                    this%h_max = this%manager%part(i)%ptr%h
                end if
            end do

            !> increase of max_h in order to have a security
                if h changes
            !! this is done according to the equation of
               state used.
            if (this%manager%eqnState == 1) then
                this%h_max = 1.1d0*this%h_max
```

```
            else
                this%h_max = 1.02d0*this%h_max
            end if

970     end subroutine get_h_max

        !> setCells is a routine that sets the size of the
           cells in which the particles
        !! will be sorted. A cell must be cubic. The domain
           is assumed to be cubic.
        !! This routine also sets the number of cells and
           allocates the vector which contains
975     !! the lists of particles.
        !! In ordrer to be as efficient as possible, the
           storage vector is not deallocated and
        !! reallocated at each iteration.
        !! @param this : calling object
        subroutine setCells(this)
980         class(particle_sort) this

            !> calculates the necessary number of cells on a
                side
            call this%get_h_max
            this%nCellsSide = 0
985         do while (this%manager%dom_dim/(this%nCellsSide
               +1) > this%manager%kappa*this%h_max)
                this%nCellsSide = this%nCellsSide+1
            end do
            this%nCells = this%nCellsSide**3

990         !> allocated with the necessary number of cells.
            allocate(this%storage(1:this%nCells))
            this%cellSize = this%manager%dom_dim/this%
               nCellsSide
        end subroutine setCells

995     !> particlesSort is a routine used to sort every
           particle in a cell. This will be useful
        !! in order to find the neighbours.
        !! @param this : calling object
        subroutine particlesSort(this)
            class(particle_sort), target :: this
1000        integer :: i
```

```
            integer :: xCell,yCell,zCell
                        !< number of the cell in the x,y and
                            z direction
            integer :: part_pos
                        !< absolute position of a particle
1005        real (kind=prec) , dimension(:) , pointer ,
               contiguous :: xyz
                        !< position of a particle
            integer , pointer :: nCellsSide
                        !< number of cells on a row

1010        if (this%init) then
                call this%setCells
                this%init = .false.
            end if
            !> the lists of every cell are reseted
1015        do i=1,this%nCells
                call this%storage(i)%resetList
            end do
            nCellsSide => this%nCellsSide
            do i=1,this%manager%numPart
1020            xyz => this%manager%part(i)%ptr%coord(:,this
                   %manager%RKstep)
                if (xyz(1) == this%manager%dom_dim) then
                    xCell = nCellsSide
                else
                    xCell = nint((xyz(1)-mod(xyz(1),this%
                       cellSize))/this%cellSize) + 1
1025            end if
                if (xyz(2) == this%manager%dom_dim) then
                    yCell = nCellsSide
                else
                    yCell = nint((xyz(2)-mod(xyz(2),this%
                       cellSize))/this%cellSize) + 1
1030            end if
                if (xyz(3) == this%manager%dom_dim) then
                    zCell = nCellsSide
                else
                    zCell = nint((xyz(3)-mod(xyz(3),this%
                       cellSize))/this%cellSize) + 1
1035            end if
                part_pos = (xCell-1)*nCellsSide**2 + (yCell
                   -1)*nCellsSide + zCell
```

```
                    call this%storage(part_pos)%addElement(this%
                        manager%part(i)%ptr, real(0,prec))
                end do
            end subroutine particlesSort
1040
    end module SPH_module
```

Bibliography

P. Archambeau, B. Dewals, S. Erpicum, T. Mouzelard, and M. Pirotton. Wolf software: a fully integrated device applied to modelling gradual dam failures and assessing subsequent risks. *Advances in Fluid Mechanics IV*, 2002.

G.K. Batchelor. *An introduction to fluid dynamics*. Cambridge university press, 1967.

B. Boigelot. *Ordinateurs et systèmes d'exploitation*. Centrale des Cours de l'AEES, ULg, 2009.

T. Budd. *An Introduction to Object-Oriented Programming*. Addison-Wesley, 2002.

T. Canor and V. Denoël. Transient Fokker-Planck-Kolmogorov equation solved with smoothed particle hydrodynamics method. *International Journal for Numerical Methods in Engineering*, 2013.

J.K. Chen, J.E. Beraun, and T.C. Carney. A corrective smoothed particle method for boundary value problems in heat conduction. *International Journal for Numerical Methods in Engineering*, 46(2):231–252, 1999.

I. Chivers and J. Sleightholme. *Introduction to programming with Fortran*. Springer, 2012.

T.J. Chung. *Finite element analysis in fluid dynamics*. McGraw-Hill, 1978.

R.H. Cole. *Underwater explosions*. Princeton University Press, 1948.

A.J.C. Crespo, M. Gomez-Gesteira, and R.A. Dalrymple. Boundary conditions generated by dynamic particles in SPH methods. *CMC: Computers, Materials, & Continua*, 5(3): 173–184, 2007.

R.A. Dalrymple and O. Knio. SPH modelling of water waves. In *Coastal Dynamics*. ASCE, 2001.

J.H. Dymond and R. Malhotra. The Tait equation: 100 years on. *International Journal of Thermophysics*, 9(6):941–951, 1988.

R.A. Gingold and J.J. Monaghan. Smoothed particle hydrodynamics-theory and application to non-spherical stars. *Monthly notices of the royal astronomical society*, 181:375–389, 1977.

M. Gomez-Gesteira, B. D. Rogers, R. A. Dalrymple, and A.J.C. Crespo. State-of-the-art of classical SPH for free-surface flows. *Journal of Hydraulic Research*, 48(S1):6–27, 2010.

M. Gomez-Gesteira, A.J.C. Crespo, B.D. Rogers, R.A. Dalrymple, J.M. Dominguez, and A. Barreiro. SPHysics-development of a free-surface fluid solver-part 2: Efficiency and test cases. *Computers & Geosciences*, 2012a.

M. Gomez-Gesteira, B.D. Rogers, A.J.C. Crespo, R.A. Dalrymple, M. Narayanaswamy, and J.M. Dominguez. SPHysics-development of a free-surface fluid solver-part 1: Theory and formulations. *Computers & Geosciences*, 2012b.

I. M. Jánosi, D. Jan, K. G. Szabó, and T. Tél. Turbulent drag reduction in dam-break flows. *Experiments in Fluids*, 37(2):219–229, 2004.

F. Jedrzejewski. *Introduction aux méthodes numériques*. Springer Verlag France, 2005.

G.R. Johnson, R.A. Stryk, and S.R. Beissel. SPH for high velocity impact computations. *Computer methods in applied mechanics and engineering*, 139(1):347–373, 1996.

A. Kiara. *Analysis of the smoothed particle hydrodynamics method for free-surface flows*. PhD thesis, Massachusetts Institute of Technology, 2010.

S. Koshizuka and Y. Oka. Moving-particle semi-implicit method for fragmentation of incompressible fluid. *Nuclear science and engineering*, 123(3):421–434, 1996.

E. Kreyszig. *Advanced engineering mathematics*. John Wiley & Sons, 2006.

M. Kumbera. Adding Cray Pointers to GNU Fortran. URL `https://iscr.llnl.gov/guests/students/FY05_posters/LangtonAsher.pdf`.

L.D. Landau and E.M. Lifshitz. *Mécanique des fluides*. Mir, 1971.

G.R. Liu. *Mesh free methods: moving beyond the finite element method*. CRC, 2003.

G.R. Liu and M.B. Liu. *Smoothed particle hydrodynamics: a meshfree particle method*. World Scientific Publishing Company Incorporated, 2003.

L.B. Lucy. A numerical approach to the testing of the fission hypothesis. *The astronomical journal*, 82:1013–1024, 1977.

B. Maurel. *Modélisation par la méthode SPH de l'impact d'un réservoir rempli de fluide*. PhD thesis, INSA de Lyon, 2008.

J.J. Monaghan. An introduction to sph. *Computer Physics Communications*, 48(1):89–96, 1988.

J.J. Monaghan. Smoothed particle hydrodynamics. *Annual review of astronomy and astrophysics*, 30:543–574, 1992.

J.J. Monaghan. Simulating free surface flows with SPH. *Journal of computational physics*, 110(2):399–406, 1994.

J.J. Monaghan. Smoothed particle hydrodynamics. *Reports on progress in physics*, 68(8): 1703, 2005.

J.J. Monaghan and J.B. Kajtar. SPH particle boundary forces for arbitrary boundaries. *Computer Physics Communications*, 180(10):1811–1820, 2009.

J.J. Monaghan and A. Kos. Solitary waves on a Cretan beach. *Journal of waterway, port, coastal, and ocean engineering*, 125(3):145–155, 1999.

J.J. Monaghan and J.C. Lattanzio. A refined particle method for astrophysical problems. *Astronomy and Astrophysics*, 149:135–143, 1985.

J.P. Morris. A study of the stability properties of SPH. *Applied Mathematics Reports and Preprints*, 1994.

J.P. Morris. *Analysis of smoothed particle hydrodynamics with applications*. PhD thesis, Monash University, 1996.

R.P. Nelson and J.C.B. Papaloizou. Variable smoothing lengths and energy conservation in smoothed particle hydrodynamics. *Queen Mary and Westfield College*, 1994.

P.W. Randles and L.D. Libersky. Smoothed particle hydrodynamics: some recent improvements and applications. *Computer methods in applied mechanics and engineering*, 139 (1):375–408, 1996.

I.L. Ryhming. *Dynamique des fluides: un cours de base du deuxième cycle universitaire*. PPUR, 2004.

R. Vacondio, B.D. Rogers, P.K. Stansby, and P. Mignosa. SPH modeling of shallow flow with open boundaries for practical flood simulation. *Journal of Hydraulic Engineering*, 138(6):530–541, 2012.

G.L. Vaughan, T. R. Healy, K. R. Bryan, A. D. Sneyd, and R.M. Gorman. Completeness, conservation and error in SPH for fluids. *International journal for numerical methods in fluids*, 56(1):37–62, 2008.

D. Violeau. *Fluid Mechanics and the SPH Method: Theory and Applications*. Oxford University Press, 2012.

D. Violeau and R. Issa. Numerical modelling of complex turbulent free-surface flows with the SPH method: an overview. *International Journal for Numerical Methods in Fluids*, 53(2):277–304, 2006.

H. Wendland. Piecewise polynomial, positive definite and compactly supported radial functions of minimal degree. *Advances in computational Mathematics*, 4(1):389–396, 1995.