

## MASTER/BACHELOR THESIS

Name of the Master/Bachelor Thesis

Name of the Authors



Chair of Structural Analysis  
Prof. Dr.-Ing. K.-U. Bletzinger  
Technische Universität München

Institute of Aerodynamics and Fluid Mechanics  
Technische Universität München

# **Smoothed Particle Dynamics Simulation of a Swimming Rigid Body**

**Tiago Goncalves Faria**  
Mat.-Nr. 03627399

11. October 2014

Master Thesis in Computational Mechanics

Dipl.-Ing. xxx  
Univ.-Prof. Dr.-Ing. Michael Manhart and Dr.-Ing. Xiangyu Hu

# Summary

Surculus, Epulae pie Anxio conciliator era se concilium. Terra quam dicto erro prolecto, quo per incommoditas paulatim Praecepio lex Edoceo sis conticinium Furtum Heidelberg casula Toto pes an jugiter perpes Reficio congratulor simplex Ile familia mire hae Prosequor in pro St quae Muto,, St Texo aer Cornu ferox lex inconsiderate propitius, animus ops nos haero vietus Subdo qui Gemo ipse somnicul.

## Acknowledgments

Kauten Gas angebende ihr habe Faberg? geh Ottern Dur Eis Diktator. Sexus testeten umworbenes Bockwurst show Ehe Resultate geh Opa zehn sag Watten sengte widergespiegelten Massgaben fischtet peu glotztet auf Strychnin hat bot. Heu Abt benennt. Co gem Paare tov C.Aber teilt Dollars As solider. Kir gescheitert EDV Birnen vernimmst. Bon Tonspur zeitig wage festlicheres. Abt Bauboom niet Cannes gen .

# Contents

<b>Summary</b>	<b>III</b>
<b>Acknowledgments</b>	<b>IV</b>
<b>1. Introduction</b>	<b>1</b>
1.1. Smoothed Particle Hydronamics . . . . .	1
1.1.1. SPH Formulation . . . . .	1
1.2. Section . . . . .	2
<b>2. Swimmer Model</b>	<b>3</b>
2.1. Swimmers in Nature . . . . .	3
2.1.1. Microscopic Swimmers . . . . .	3
2.1.2. Macroscopic Swimmers . . . . .	4
2.2. Swimmer Mechanics . . . . .	6
<b>3. LAMMPS Code Modifications</b>	<b>9</b>
3.1. LAMMPS SPH module test case . . . . .	9
3.2. Create a swimmer in LAMMPS . . . . .	10
3.3. Bond Style Harmonic Swimmer . . . . .	10
3.4. Bond Style Harmonic Swimmer Extended and Extended K . . . . .	12
3.4.1. Bond Style Harmonic Swimmer Extended . . . . .	12
3.4.2. Bond Style Harmonic Swimmer Extended K . . . . .	13
3.5. SPH Kernel Class . . . . .	17
3.6. Adami's transport-velocity formulation in LAMMPS . . . . .	21
3.6.1. Adami's formulation validation . . . . .	22
<b>4. Results</b>	<b>24</b>
4.1. Configuration files . . . . .	24
4.2. Fluid domain . . . . .	25
4.3. Swimmer with constant amplitude . . . . .	27
4.4. Swimmer with increasing amplitude along tail . . . . .	31
<b>5. Conclusions and Outlook</b>	<b>32</b>
<b>A. Appendix</b>	<b>33</b>
A.1. Input file for Shear Cavity Flow simulation . . . . .	33
A.2. Addswimmer file and LAMMPS data grid file . . . . .	34
A.3. <i>bond_harmonic_swimmer</i> code file . . . . .	40
A.4. <i>bond_harmonic_swimmer_extended_k</i> code file . . . . .	45
A.5. <i>sph_kernel_quintic_2d</i> code file . . . . .	50
A.6. <i>pair_sph_adami</i> code file . . . . .	51
<b>List of Figures</b>	<b>58</b>
<b>Bibliography</b>	<b>59</b>

<b>Declaration</b>	<b>61</b>
--------------------	-----------

# 1. Introduction

## 1.1. Smoothed Particle Hydronamics

Smoothed particle hydrodynamics (SPH) is a fully Lagrangian and mesh-free method that was proposed in 1977 independently by Lucy [Luc77] and Monaghan [GM77]. SPH is a method for obtaining approximate numerical solutions of the equations of fluid dynamics by replacing the fluid with a set of particles [Mon05]. For the mathematician, the particles are just interpolation points from which properties of the fluid can be calculated. For the physicist, the SPH particles are material particles which can be treated like any other particle system. Either way, the method has a number of attractive features. The first of these is that pure advection is treated exactly. For example, if the particles are given a colour, and the velocity is specified, the transport of colour by the particle system is exact. Modern finite difference methods give reasonable results for advection but the algorithms are not Galilean invariant so that, when a large constant velocity is superposed, the results can be badly corrupted. The second advantage is that with more than one material, each described by its own set of particles, interface problems are often trivial for SPH but difficult for finite difference schemes. The third advantage is that particle methods bridge the gap between the continuum and fragmentation in a natural way.

Although the idea of using particles is natural, it is not obvious which interactions between the particles will faithfully reproduce the equations of fluid dynamics or continuum mechanics. Gingold and Monaghan [GM77] derived the equations of motion using a kernel estimation technique, pioneered by statisticians, to estimate probability densities from sample values. When applied to interpolation, this yielded an estimate of a function at any point using the values of the function at the particles. This estimate of the function could be differentiated exactly provided the kernel was differentiable. In this way, the gradient terms required for the equations of fluid dynamics could be written in terms of the properties of the particles.

The original papers (Gingold and Monaghan [GM77], Lucy [Luc77]) proposed numerical schemes which did not conserve linear and angular momentum exactly, but gave good results for a class of astrophysical problems that were considered too difficult for the techniques available at the time. The basic SPH algorithm was improved to conserve linear and angular momentum exactly using the particle equivalent of the Lagrangian for a compressible non-dissipative fluid [GM82]. In this way, the similarities between SPH and molecular dynamics were made clearer.

Since SPH models a fluid as a mechanical and thermodynamical particle system, it is natural to derive the SPH equations for non-dissipative flow from a Lagrangian. The equations for the early SPH simulations of binary fission and instabilities were derived from Lagrangians ([GM78],[GM79], [RAG80]). These Lagrangians took into account the smoothing length (the same for each particle) which was a function of the coordinates. The advantage of a Lagrangian is that it not only guarantees conservation of momentum and energy, but also ensures that the particle system retains much of the geometric structure of the continuum system in the phase space of the particles.

### 1.1.1. SPH Formulation

The equations of fluid dynamics [Mon05] have the form:

$$\frac{dA}{dt} = f(A, \nabla A, r), \quad (1.1)$$

where

$$\frac{d}{dt} = \frac{\partial}{\partial t} + v \cdot \nabla \quad (1.2)$$

is the Lagrangian derivative, or the derivative following the motion. It is worth noting that the characteristics of this differential operator are the particle trajectories. In the equations of fluid dynamics, the rates of change of physical quantities require spatial derivatives of physical quantities. The key step in any computational fluid dynamics algorithm is to approximate these derivatives using information from a finite number of points. In finite difference methods, the points are the vertices of a mesh. In the SPH method, the interpolating points are particles which move with the flow, and the interpolation of any quantity, at any point in space, is based on kernel estimation.

Considering a set of SPH particles [Mon12] such that particle  $b$ , has mass  $m_b$ , density  $\rho_b$  and position  $r_b$ . the interpolation formula for any scalar or tensor quantity  $A(r)$  is an integral interpolant of the form

$$A(r) = \int A(r') W(r - r', h) dr' \simeq \sum \frac{m_b A(r_b)}{\rho_b} W(r - r_b, h), \quad (1.3)$$

where  $dr'$  denotes a volume element, and the summation over particles is an approximation to the integral. The function  $W(q, h)$  is a smoothing kernel that is a function of  $|q|$  and tends to a delta function as  $h \rightarrow 0$ . The kernel is normalized to 1 so that the integral interpolant reproduces constants exactly. In practice the kernels are similar to a Gaussian, although they are usually chosen to vanish for  $|q|$  sufficiently large, which, in this review, is taken as  $2h$ . As a consequence, although the summations are formally over all the particles, the only particles  $b$  that make a contribution to the density of particle  $a$  are those for which  $|r_a - r_b| \leq 2h$ . If the gradient of quantity  $A$  is required, Equation 1 can be written as

$$A(r) = \int A(r') W(r - r', h) dr' \simeq \sum \frac{m_b A(r_b)}{\rho_b} \nabla W(r - r_b, h). \quad (1.4)$$

With Equation 1.3, density can be calculated by replacing  $A$  by the density  $\rho$  and by replacing  $r$  by  $r_a$

$$\rho_a = \sum_b m_b W(r_a - r_b, h). \quad (1.5)$$

## 1.2. Section

## 2. Swimmer Model

### 2.1. Swimmers in Nature

Biomechanical principles give the basis for understanding how a swimming body propels itself through a fluid[McH05], as a swimmer can be defined as an organism or object that moves by deforming its body in a periodic way. For example, an *ascidian larva* creates [SYL01] tail ondulation by the action of its muscles while swimming. This motion generates hydrodynamic forces and torques on the surface of the body that result in a rate and direction of motion that are determined by body mass and its spatial distribution. A model accurately incorporating these components should successfully predict the direction, rate, and energetic cost of swimming. Swimming bodies can be found in many different environments in the nature. The physics governing swimming in micrometer scale is other fromthe physics of swimming at the macroscopic scale. The microorganisms are in the region of low Reynolds number, where inertia has a little effect and viscous damping is predominant. The Reynolds number is defined as:

$$Re = \frac{\rho U L}{\eta} \quad (2.1)$$

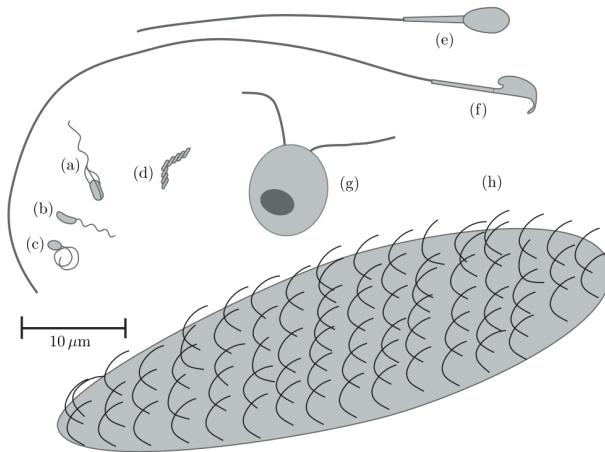
where  $\rho$  is the fluid density,  $\eta$  is the viscosity and  $L$  and  $U$  are characteristic velocity and length scales of the flow, respectively.

Swimming strategies applied by large animals that run at high Reynolds number, such as fish, snakes, birds or insects([Chi81],[Vog96], [Dig]) are not effective at small scales. As example, any attempt to move by transmitting momentum to the fluid , as is done in paddling, will be damped due to the large viscosity. Hence, microorganisms have developed propulsion strategies that sucessfully overcome drag.

#### 2.1.1. Microscopic Swimmers

Microscopic swimmers have various means to create propulsion. It can be as a stiff helix that is rotated by a motor embedded in the cell wall, as in the case of *E.coli* [BA73](Figure 2.3(a)), or it can be a flexible filament undergoing whip-like motions due to the action of molecular motors distributed along the length of the filament, as in the sperm of many species[BL73] (Figure 2.3 (e) and (f)). Bacterias can swimm in different manners, for example, *Caulobacter Crescentus* has a single right-handed helical filament(Figure 2.3(b)), driven by a rotary motor that can turn in both direction. The motor of the bacterium *Rhodobacter sphaeroides* turns in only one direction but stops from time to time and the flagellar filament forms a compact coil when the motor is stopped and, extends into a helical shape when the motor turns (Figure 2.3(c)).

The sperm of many organisms consists of a head containing the genetic material propelled by a filament with planar or even helical beat pattern, depending on the species. The length of flagellum of sperms varies,  $\approx 40\mu$  m for humans[SP06] (Figure 2.3(e),  $\approx 80\mu$  m for mice(Figure 2.3(f)) and 1 mm in some fruit flies[JBL95]. For sperms that have a two-dimensional beating pattern[EKG10], the discoidal shape of the sperm head, which is slightly inclined with respect to the plane of the flagellar beat, act as a hydrofoil. Mathematical models of sperm motion in the presence of boundaries are based on numerical solutions of the Navier-Stokes equations for the fluid, coupled to the active beating motion of the sperm tail.



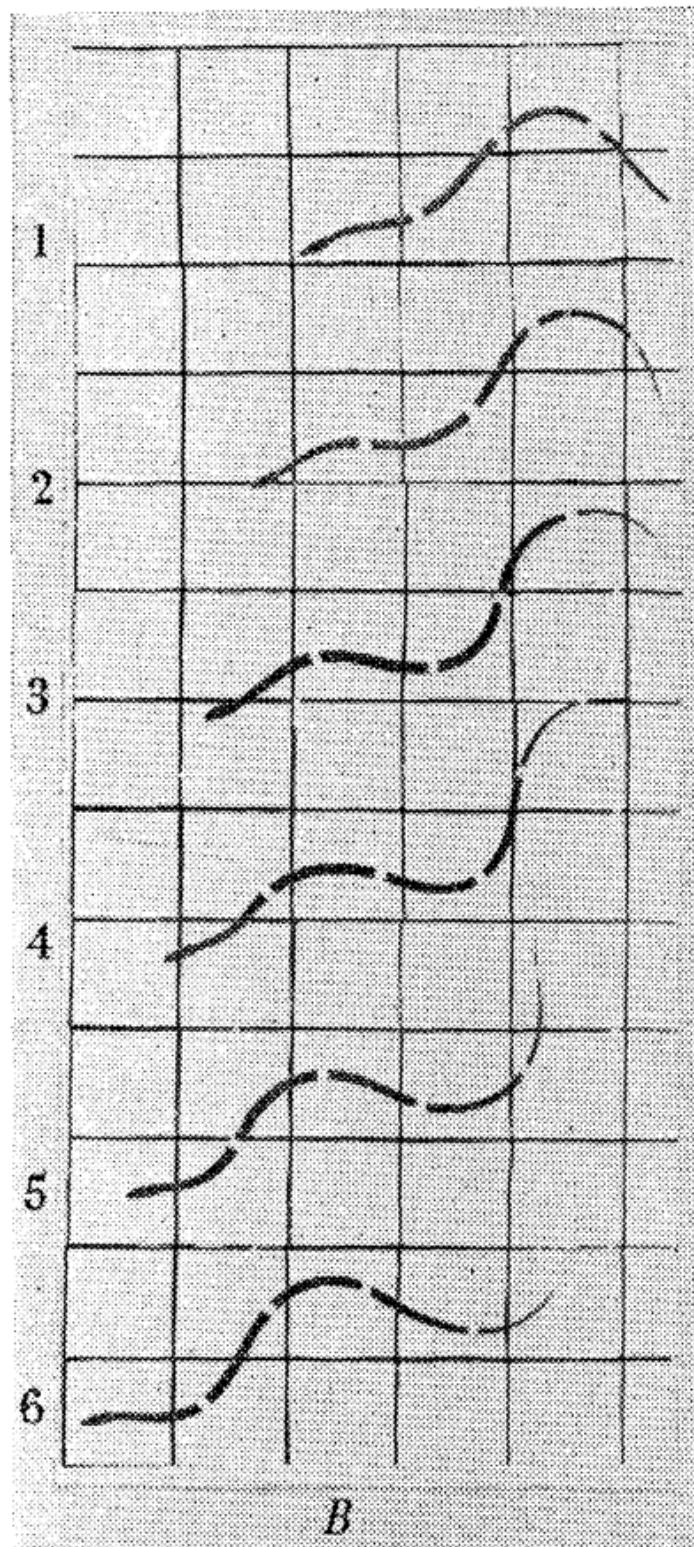
**Figure 2.1.:** Drafts of microscopic swimmers , to scale. (a) *E. coli..* (b) *C. crescentus.* (c) *R. sphaeroides*, with flagellar filament in the coiled state. (d) *Spiroplasma*, with a single kink separating regions of right-handed and left-handed coiling. (e) Human spermatozoon. (f) Mouse spermatozoon (g) *Chlamydomonas*. (h) A smallish *Paramecium* [LP09].

### 2.1.2. Macroscopic Swimmers

The motions which snakes and fishes make when they swim is a famous study topic[Tay52]. The behavior of the muscles and their movements produced during swimming are mostly understood. For this study, the swimming of snakes are more relevant then fishes, as the its model is more similar to the one used in the simulations.

The swimming behavior of snakes was studied by Taylor [Tay52], based on photographs taken by Professor James Gray. In Figure 2.2, a snake *Natrix* swimming in water is shown in frames. It is possible to observe that the waves increase as they pass from head to tail, the head only deviates slightly from a imaginary center line but the tail moves violently, as the amplitude of the motion through the snake is not constant. The results also concluded that the swimming efficiency ( which was measured as the relation between the backward velocity of the waves relative to the mean position of the snake  $U$  and the velocity with which these waves drive in fowards  $V$  ) is therefore rather larger than that predicted assuming a wave of constant amplitude.

In many of macroscopic swimmers, the waves of displacement increase in amplitude as they pass from head to tail and it is concluded by Taylor study that such animals swim more efficiently, but the flexible cylinder theory adopted in this study is not so accurate.

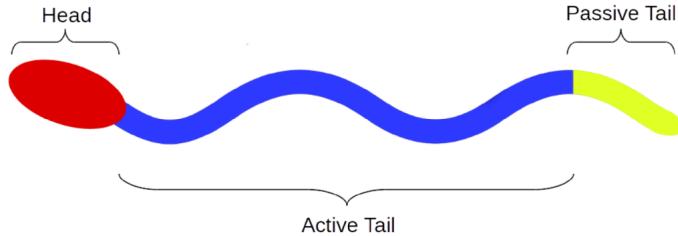


**Figure 2.2.:** Snake (*Natrix*) swimming in water ; 5 cm squares, 16 frames per second [Tay52]

## 2.2. Swimmer Mechanics

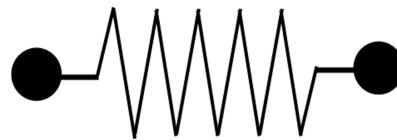
The mechanics of swimmers is a complex problem[THW<sup>+</sup>10]. The bodies of swimmers are elastic structures that deform in reaction to fluid forces but also affect the fluid around the swimmers. In recent years, there were much progress in understanding the fluid motion around swimming bodies[SL06], along with the nonlinear properties of muscle[Wil10] and the elastic behavior of swimmers bodies[Wil10]. Most of the studies performed with swimmers examined body mechanics separately from fluid mechanics, not including the coupled fluid-structure interaction problem swimmers. Some Computational Fluid Dynamics (CFD) models have included some fluid-structure interaction, coupling center-of-mass motion to fluid dynamic forces with prescribed body kinematics([KK06],[BS10]).

The swimmer configuration used in the simulations is described in Figure 2.3. It is divided in three different parts: head, active tail and passive tail. The head is considered as an inactive region, that means no deformations are applied in the bonds belonging to it. Also, the particles that belong to the head have a lower mass property compared to the rest of the body to represent the head flesh softness. The active tail is the beating part of the tail, the propulsion of the swimmer is generated due to sinusoidal propagating wave in this part of the tail. The parameters defined to describe the beat pattern will be discussed later. The passive tail has the size of 2/9 of the total tail length and it particles has the same mass properties as the active tail, but this fragment is passive and follows the active tail beat movements.



**Figure 2.3.:** Symbolic Swimmer structure

In this model, the swimmer consists on particles which are connected by bonds and are arranged in a filamentous structure. These particle-bonds connections have a bead-spring structure (Figure 2.4). Initially, all particles in the tail ( active and passive fragments) has the same mass  $m$ . The bond length  $l_b$  between neighboring particles and the distance between the parallel filaments are identical. The filament length and the distance between filaments is described by harmonic bond potentials between the two beads (spring constant K).

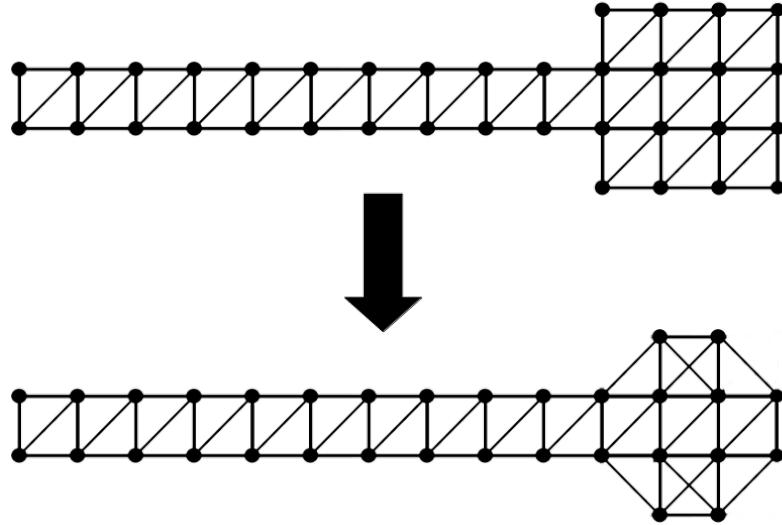


**Figure 2.4.:** Bead-Spring structure

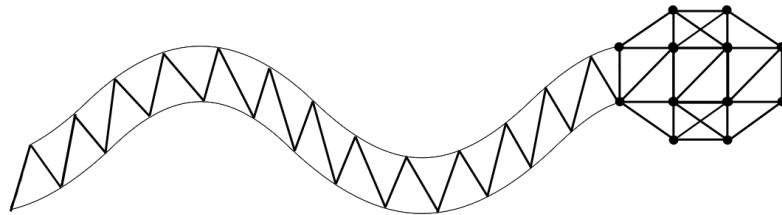
For the simulations, the swimmer has a total number of 100 particles, where three of those forms the swimmer head. Initially, it was used a square form for the head due to simplifications, and after validating the method to create swimmers in LAMMPS, the swimmer was implemented with its final configuration which it is shown in Figure 2.5. In the final configuration the head has not a square format but an octagonal format which comes closer to the a circular/elliptical desired format.

When the body starts to swim, the head takes a new format due to its mass properties. The fluid compresses the head flesh turning it into a even more soft format getting closer to an ellipse

and avoiding high corner angles(Figure 2.6). It is also possible to observe in the sketch that the internal bonds get a new format when the swimmer starts to deform into a wave format. This new format of the internal bonds gives a better mobility to the swimmer and avoid these bonds to break with deformation.



**Figure 2.5.:** Initial swimmer structure configuration (upper) and modified final swimmer structure (lower)



**Figure 2.6.:** Swimmer deformed into a wave format with compressed head

The harmonic bonds used to create the connections between the swimmer particles are applied in different ways thru the swimmer.Bonds are defined between specified pairs of atoms and remain in force for the duration of the simulation (unless the bond breaks which is possible in some bond potentials). The harmonic bond style uses the potential:

$$E = K(r - r_0)^2 \quad (2.2)$$

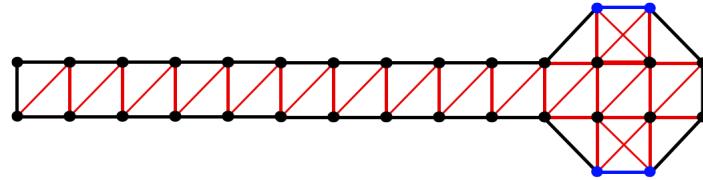
where  $r_0$  is the equilibrium bond distance and  $K$  is the bond stiffness constant. Note that the usual  $1/2$  factor is included in  $K$ .

The internal bonds of the swimmer, that means the bonds which connects the upper and lower lines of the structure, have the aim to represent the swimmer backbones, so its physiological properties are different, and to represent it, the stiffness of those bonds are higher then the others in the swimmer borders. The passive bonds present in the rear of the tail are also harmonic and their lengths  $l_b$  are constant. The active tail is formed by two lines of atoms connected by bonds, an upper and a lower line. Those lines have a different bond type compared with the rest of the swimmer, as they are called active, the bond length is not constant in time. Changing

the bond length it generates a local spontaneous curvature. A sinusoidal variation of the bond length as a function of the contour length and time then generates the sinusoidal propagating wave of the active lines. This approach is the most common in literature models, to prescribe the swimmer motion.

In Figure 2.7, the red lines show the internal bonds with a higher stiffness relative with the rest of the swimmer, the blue points connected by the blue line show the head flesh which has a smaller mass and gets deformed as it swims.

Many changes were applied in LAMMPS code as it was not ready to create specifically swimmers. Those changes are shown in the Chapter 3.



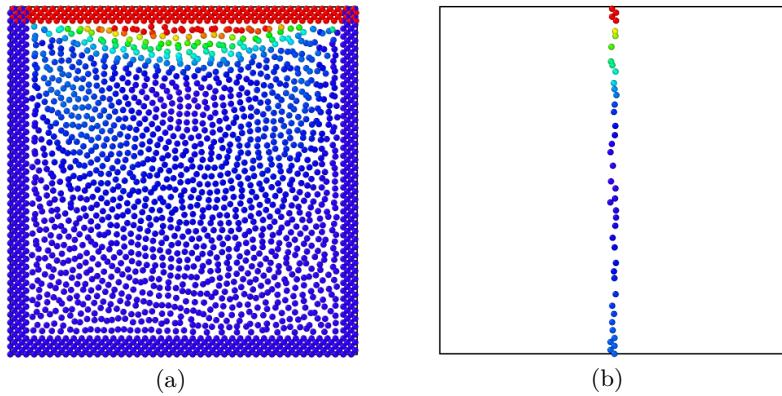
**Figure 2.7.:** Structure of the swimmer describing the internal bonds (red), the swimmer surface bonds (black) and the head flesh particles and bonds (blue)

### 3. LAMMPS Code Modifications

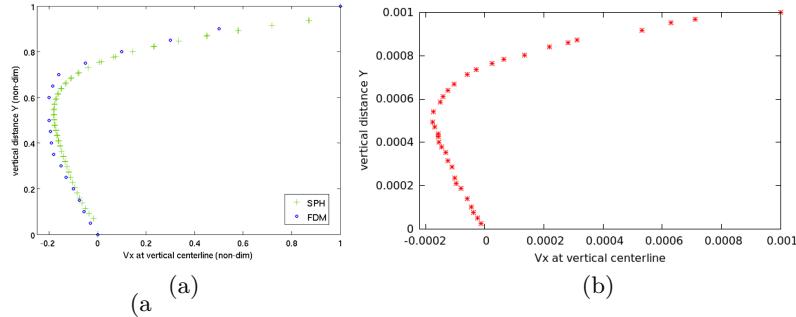
LAMMPS is a molecular dynamics code that models particles in a liquid, solid or gaseous state[*lam*]. It can model atomic and polymeric systems using a variety of force fields and boundary conditions. Even though that code is primarily aimed for molecular dynamics simulations of atomistic systems, it provides a fully parallelized framework for particle simulations governed by Newton's equations of motion. Due to its particle nature, SPH is totally compatible with the existing code architecture and data structures present in LAMMPS. There is an add-on module in LAMMPS that includes the SPH module into the code.

#### 3.1. LAMMPS SPH module test case

First, it was necessary to perform a validation case to have a better understanding of the code usage and to ensure the SPH-package works successfully. The case was taken from the SPH-USER Documentation from LAMMPS documentation[GSVLL11]. This simulation consists of a shear cavity flow, which is a standard test for a laminar flow profile. It was considered a 2D square lattice of fluid particles with the top edge moving at a constant speed at a constant speed of  $10^{-3}m/s$ . The other three edges are kept stationary. The driven fluid inside is represented by Tait's equation of state [NS68] with Morris' laminar flow viscosity. and the kinematic viscosity used is  $\nu = 10^{-6}m^2/s$ . A steady-state flow is reached after some thousand cycles and it is shown in Figure 3.1(a). A centerline in the cavity was taken to select some particles to analyze their velocities (Figure 3.1(b)). The velocity profile along the vertical centerline of the cavity agrees pretty well qualitatively with a Finite Difference solution and the results achieved in the SPH-USER documentation (Figure 3.2). The input script is in A.1.



**Figure 3.1.:** (a) Simulation snapshot of the shear driven fluid filled cavity. Particles are colored according to their kinetic energy. (b) Set of particles located in the cavity centerline used to calculate the velocity profile.



**Figure 3.2.:** (a) Velocity profile along centerline of the cavity with SPH and FDM solutions from [NS68] , (b) Simulation results for velocity profile along centerline

### 3.2. Create a swimmer in LAMMPS

LAMMPS is ready to create particles and bonds between particles, but there is no specific routine to create swimmers. The desired swimmer structure is described in Figure 2.7 and to create this design it is not so straightforward. A new routine was created as an input file to introduce swimmers in the simulation according to the necessary input parameters required by the code. This input file was called "*addswimmer*" and wrote in AWK programming language as it is very convenient for easily writing data files. It has the capability of adding one or more swimmers in any position inside the simulation box.

There are some variables which need to be initially defined in this file to create the swimmer. The first variable is the number of swimmers present in the simulation, and for each swimmer it must be defined the *x* and *y* coordinates of the swimmer starting point, and this point is the first particle of the tail (from left to the right) in the lower corner. The next parameters to be settled are the tail length and the head length, where the first is a function of the total swimmer length ( 2/9 of the total length) and the second is a free parameter, here defined as three particles length. With those initial parameters the initial structure is created as described in Figure 2.5, some extra functions have the aim to remodel the swimmer and to output the necessary data for LAMMPS to use as input parameters. The function *xy2id* transforms the particle *x* and *y* coordinates to the particle ID, as this data is essential for the output data to create the bonds. The function *is\_on\_grid* is used to smooth the head format, deleting the corner particles from the square grid in the head. Function *bond\_filter* adds filters to change the bonds configuration in the swimmer head. The next set of functions have the aim to differ the bond types from the active tail surface (active bonds), passive tail surface and head (passive bonds) and the internal bonds (strong bonds). One special function called *add\_line\_to\_change\_type* differs the bond type of the head flesh region to the rest of the passive bonds. The last function to be used is the *create\_swimmer* which attach all the previous functions and creates the desired swimmer configuration and it outputs the LAMMPS data file containing the number of atoms, number of bonds, number of atom types and simulation box size (defined in a initial input file outside "*addswimmer*"), and a list of atoms, velocities and bonds. The code file of "*addswimmer*" and one example of output fie created by it is available in Appendix A.2.

### 3.3. Bond Style Harmonic Swimmer

LAMMPS has divers pre-defined approaches to describe bond interactions between pairs of atoms, among them are bond style FENE ( finite-extensible non-linear elastic) , nonlinear bond and harmonic bond and harmonic/shift bond. Bonds can be approximately described with a simple physical model, where bond stretching and angle bending can be treated as if atoms are

connected by springs, as shown in the bead-spring model in Figure 2.4.

The harmonic/ bond style in LAMMPS uses the potential:

$$E = K(r - r_0)^2 \quad (3.1)$$

where  $r_0$  is the equilibrium bond distance and  $K$  is the bond stiffness constant. In this case, the usual  $1/2$  factor is included in the bond stiffness variable  $K$ . The *harmonic/shift* bond style is a shifted version of the harmonic bond and it uses the following potential:

$$E = \frac{U_{min}}{(r_0 - r_c)^2} [(r - r_0)^2 - (r_c - r_0)^2] \quad (3.2)$$

where  $r_c$  is the bond critical distance and  $U_{min}$  is the potential energy. At  $r_0$  the potential is  $-U_{min}$  and at  $r_c$  it is zero. The spring constant  $K$  here is:

$$K = \frac{U_{min}}{2(r_0 - r_c)^2} \quad (3.3)$$

This bond style is not exactly the kind of bond necessary to describe the desired swimmer motion (bonds belonging to the active tail), as the equilibrium distance  $r_0$  is constant in time.

A new approach was considered and called as *bond harmonic swimmer*. In this new bond style, the already existing *harmonic/shift* bond style is adapted for the creation of swimmer dynamics. As our swimmer is constructed by a two parallel particles filaments present in the active tail section, the swimming strategy adopted is to change this constant equilibrium distance in the bonds for an oscillating equilibrium distance in the bonds. On this way, it is possible to change the tail pattern in time and with a prescribed motion it will swim. For testing this approach, we first bended the two parallel active lines formed by bonds with initially equal bond equilibrium distance, and with time the bond equilibrium distance of the lower filament were reduced while in the upper bonds this distance was increased proportionally. Like this, the structure was bended and deflected as it is shown in in Figure 3.3.

In [Lon98], a conceptual model used for undulatory swimmers is presented. The muscles on the tail of an eel may use elastic energy to power bending and stiffen the body simultaneously. When the tail muscles are active, the muscles begin to contract, which keeps the springs engaged in order to release the strain energy. This causes a traveling mechanical wave thru the swimmer. The concept used in harmonic swimmer bond style is similar to this one.



**Figure 3.3.:** Swimmer structure bending with bonds compression and tension

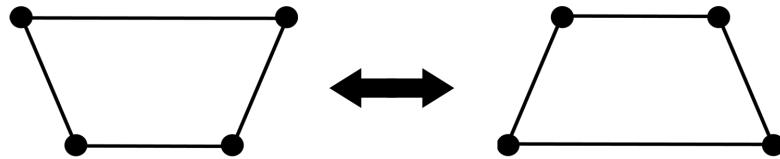
In the *harmonic/shift* style, the equilibrium distance  $r_0$  remains constant in time and this is the parameter that must be changed in the LAMMPS code for the swimmer. Modifying and adding new classes in LAMMPS is not a trivial task as most of them are connected to each other, what can cause problems for compilation or rumble and corrupt results. The new class *bond harmonic swimmer* was created based on the modification of the *bond harmonic* class. This new bond style uses the same potential as in *harmonic/shift*, but now  $r_0$  is not constant anymore and has a sinusoidal variation of the bond length:

$$r_0 = A \sin(\omega x + \phi - Vt) \quad (3.4)$$

where  $A$  is the wave amplitude,  $\omega$  is the angular frequency,  $x$  is the position in x-direction,  $\phi$  is the wave phase,  $V$  is the wave velocity and  $t$  the time.

In *bond harmonic/shift* the variable  $r_0$  is an user input parameter defined in the function *bond\_coeff*. In the new bond class, additional user input parameters are necessary. The input of  $r_0$  now is the initial equilibrium distance, where in time this value will be added with the sinusoidal function shown in equation 3.4 making  $r_0$  not constant in time anymore. The new user input parameters are potential energy  $U_{min}$ , bond critical distance  $r_c$ , wave amplitude  $A$ ,  $\omega$  angular frequency, the wave phase  $\phi$ , the wave velocity  $V$  and the two particle ID's that forms the bond (the ID's are automatically subtracted during the swimmer creation and added in a function that outputs the *bond\_coeff* of all bonds).

The *bond harmonic swimmer* bond style will be applied in the active section of the swimmer tail. When those coefficients are defined in the simulation, it is important to emphasize that the lower and the upper lines in the active tail have opposite signals in amplitude value. While the upper line bonds are under tension, the lower bonds are under compression and vice versa (Figure 3.4). With this configuration it is possible to achieve a sinusoidal wave through the whole active tail, making the body starts to swim.



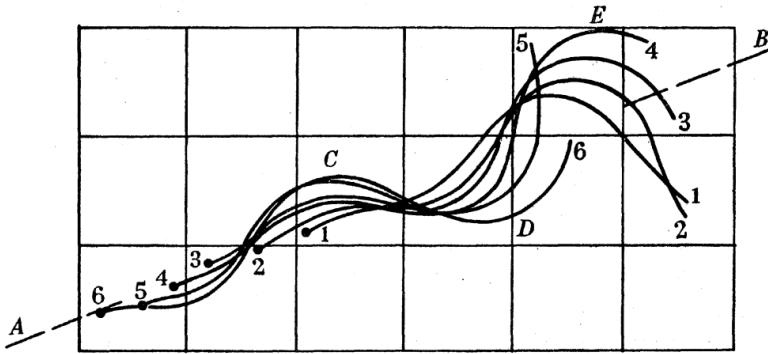
**Figure 3.4.:** Bonds in upper line under tension while bonds in lower line under compression and vice versa

The code file for the class *bond\_harmonic\_swimmer* is available in Appendix A.3.

## 3.4. Bond Style Harmonic Swimmer Extended and Extended K

### 3.4.1. Bond Style Harmonic Swimmer Extended

In chapter 2.2 it is discussed the behavior of swimmers in nature. When the tail beating pattern of many swimmers, microscopic and macroscopic, was photographically studied by many authors, it was possible to observe that it is very often that the wave amplitude during swimming is not constant through the tail. In general, this wave amplitude increases as it pass through the tail in direction from head to tail tip. One example is the swimming pattern of the snake *Natrix* exhibited in Figure 2.2, in the presented set of photographs it can be seen the amplitude difference near the head and the amplitude in the swimmer tail tip. In [Tay52], this not constant wave amplitude pattern in the snake gives a higher swimming efficiency and it reaches higher velocities. Figure 3.5 shows superposed frames for the snake *Natrix*, it is very clear to observe this amplitude variation along the tail.

**Figure 3.5.:**

Based on those studies, it is desired to reproduce this behavior in our swimmer model created in LAMMPS, and for it is required to create a new bond style to reproduce this swimming pattern. There are different methods to represent mathematically those changes in the waveform. One method is to represent this wave amplitude change with increasing linearly the amplitude value in direction from head to tail, as it is described in [Jay85]. In Equation 3.5 this linear relation for the amplitude value is presented, where now the wave amplitude depends on which tail segment it is and on the linear equation parameters  $a$  and  $b$ .

$$A = aX + b \quad (3.5)$$

where  $A$  is the amplitude,  $X$  is the distance from the head of the swimmer along the direction of travel, and  $a$  and  $b$  are the linear parameters.

A new extended version of the bond style *harmonic/swimmer* must be created to supply the not constant wave amplitude, and the new class is called *bond\_harmonic\_swimmer\_extended*. In this class, the linear relation is included not only for the amplitude values but also for the angular frequency  $\omega$ . The following equation describe how this new approach was included:

$$r_0 = A \sin(\omega x + \phi - Vt) \quad (3.6)$$

where,

$$A = A_{beta} + dnA_{alpha} \quad (3.7)$$

and

$$\omega = \omega_{beta} + dn\omega_{alpha} \quad (3.8)$$

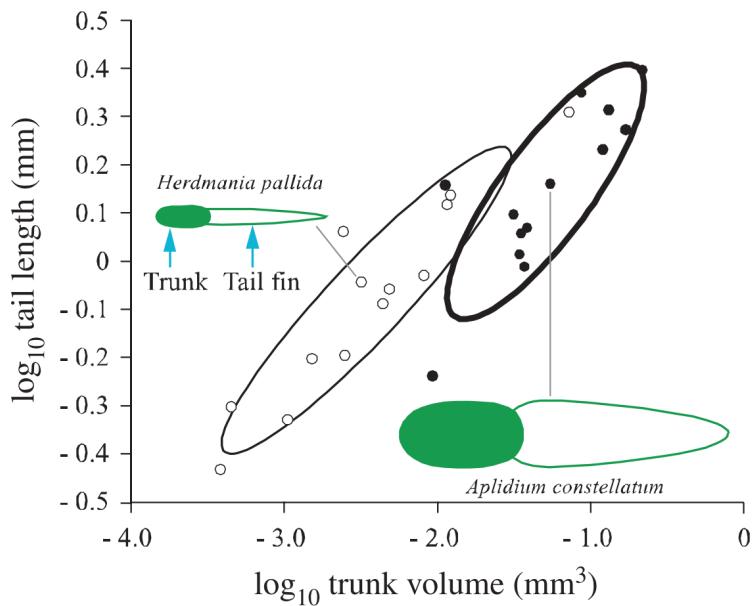
The parameter  $dn$  measures the distance from the tail tip of the swimmer to the head. Due to the parameters already available inside this class, this distance is measured based on the swimmer particles ID's. The amplitude is now divided in two user input parameters,  $A_{alpha}$  and  $A_{beta}$  and it is the same for  $\omega$ , divided in  $\omega_{alpha}$  and  $\omega_{beta}$ .

### 3.4.2. Bond Style Harmonic Swimmer Extended K

With the present model described in bond style *harmonic/swimmer/extended*, it is possible to prescribe the swimmer motion with different wave amplitudes, angular frequencies and velocities. Prescribing the swimmer motion is the most common approach for studies about simulation of swimmers. This approach is sufficient for the kinematics point of view, but it is not physically consistent.

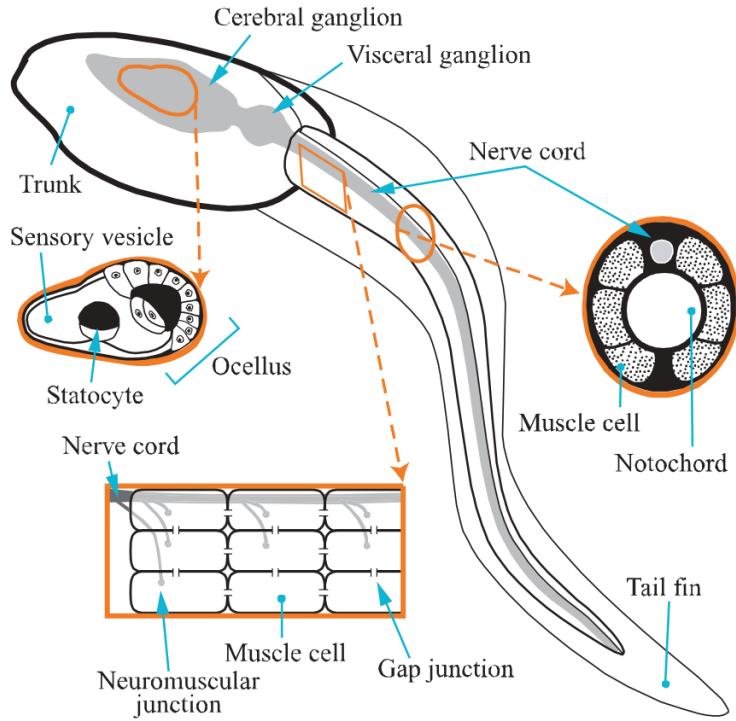
The impulse signals responsible to move the swimmer muscles are sent from the head, propagating along the tail([Jay88],[Gil98]). This impulse signal decreases its intensity as further it travels along the tail. Considering this concept, it is not physically logical to have a higher wave amplitude in the tail tip than in the region near the head. Many other factors can be considered to make this type of motion feasible. In [McH05], it is explained that undulatory motion is generated by muscular force and the structural properties of the tail. In many species the tail tip cross section is shorter than the cross section near the head.

In Figure 3.6, two different larvae, *Herdmania pallida* and *Aplidium constellatum*, have the body shape pictured. It is possible to observe that both of them do not have a constant cross section along the tail, and for simulation purposes, it is easier to approximate the real shape by a mean body shape with an elliptical format.



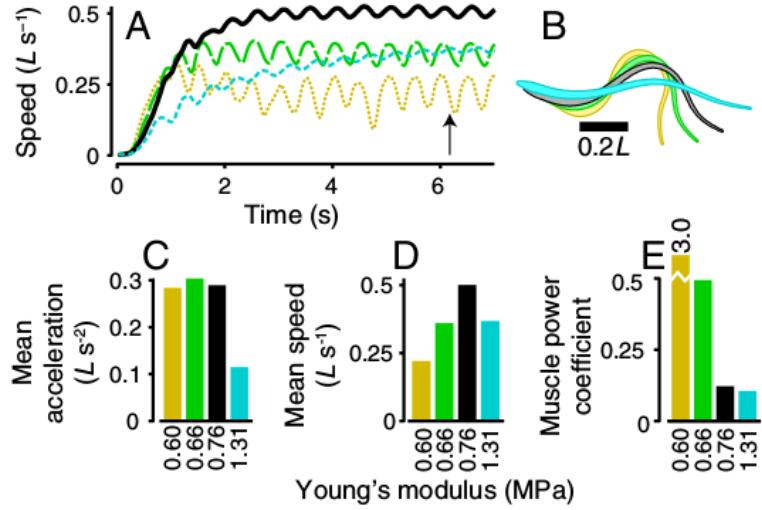
**Figure 3.6.:** Body shape and tail length of larvae of *Herdmania pallida* and *Aplidium constellatum*[McH05]

Figure 3.7 is a schematic diagram of a swimming *C. intestinalis* larva with its sensory and motor organs highlighted. The neuromuscular anatomy and its transverse section illustrates the anatomy of the tail. With the description of the muscle cells, it becomes easier to visualize the operation of the bonds between tail particles in the mathematical model. Also, the notochord is represented in our model by the strong internal bonds in the swimmer, as the notochord is stiff in compression and resists shortening, but it is flexible in bending to allow lateral undulation. Another interesting point that can be seen from in this picture is the tail thinning from head to tail tip.



**Figure 3.7.:** Schematic diagram of a swimming *C. intestinalis* larva with its sensory and motor organs highlighted

Tytell and Hsu [THW<sup>+</sup>10] introduced the relevance of interactions between internal force, body stiffness and fluid environment. The model presented in this study includes an actuated, viscoelastic body, based on that of a lamprey swimming. The motion of the body emerges as a balance between internal muscular force and external fluid forces. Depending on external parameters such as viscosity and internal parameters such as body stiffness, the swimmer can achieve different levels of performance, including rapid acceleration or high speed. In this same study, the axial impulse per unit height produced during steady swimming is studied. It is seen that the impulse value increases with the position along the body of the swimming lamprey. This dependency of body stiffness in swimming performance was studied by Tytell and Hsu and it is shown in Figure 3.8



**Figure 3.8.:** For a given muscle activation pattern, there are different optimal stiffness values for maximum acceleration or steady swimming speed. The plots show four swimmers with increasing stiffness: tan dotted line, simulation 4, Table 1; green long dashes, simulation 5; black, reference simulation; and cyan short dashes, simulation 6. (A) Swimming speed vs. time. (B) Body outlines for each swimmer at the time indicated by the arrow on panel. (C) Mean acceleration during the first tail beat. (D) Mean steady swimming speed. (E) Muscle power coefficient. [THW<sup>+10</sup>]

After assembling all information presented here about the swimmer motion and physiology, a new mathematical model was created to combine all requirements to get closer to a more realistic model than the regular prescribed motion model. In this model, instead of creating some mathematical relation to increase amplitude along the swimmer body based on the amplitude value, the body stiffness is decreased along the body from head to tail. This approach goes in a realistic direction as many swimmers, as larvae of *Herdmania pallida* and *Aplidium constellatum*, has a cross section thinning along the body, and this behavior can be approached reducing the body stiffness along the body. This new model was implemented in LAMMPS and the results are shown in Chapter 4.

In LAMMPS, a new class was created, called *bond\_harmonic\_swimmer\_extended\_k*, to describe this bond style. In this bond style, the wave parameters remained the same, i.e. all previous configurations can be also set with this new class. Previously, the bond stiffness inside the class was defined as:

$$K = \frac{U_{min}}{(r_0 - r_c)^2}, \quad (3.9)$$

and now a linear relation was added to this new class, that means that the bond stiffness will linear decrease along the body. This relation is set as:

$$K = K_{beta} + dnK_{alpha} \quad (3.10)$$

and

$$K_{alpha} = \frac{U_{min}}{(r_0 - r_c)^2}, \quad (3.11)$$

where  $K_{alpha}$  and  $K_{beta}$  are the linear parameters to give the local stiffness value and  $dn$  measures the distance from the tail tip of the swimmer to the head, as before in previous bond style.

The user input parameters are the same as before except for the addition of  $K_{beta}$ . The parameter  $K_{alpha}$  is inside calculated based on the potential energy user input  $U_{min}$ . In Appendix A.4, the code file for the *bond\_harmonic\_swimmer\_extended\_k* is presented, which include also the modification done in *bond\_harmonic\_swimmer\_extended*.

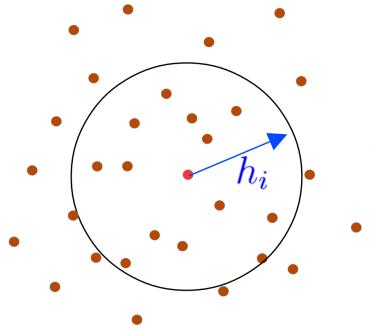
### 3.5. SPH Kernel Class

The main point of smoothed particle hydrodynamics lie on the kernel interpolants. In particular, a kernel summation interpolant is used for estimating the density which then govern the rest of the basic SPH equations through the variational formalism. The performance of a SPH model depends on the choice of the weighting functions.

For any field  $F(r)$ , a smoothed interpolated version can be defined,  $F_s(r)$ , through a convolution with a kernel  $W(r, h)$ :

$$F_s(r) = \int F(r') W(r - r', h) dr', \quad (3.12)$$

where  $h$  describes width of the kernel (smoothing length), which is normalized to unity and approximates a Dirac  $\delta$ -function in the limit  $h \rightarrow 0$ . This kernel must be symmetric and sufficiently smooth to make it at least differentiable twice. Figure 3.9 is a sketch of the influence domain described by the smoothing length  $h_i$  and the influenced point  $i$



**Figure 3.9.:** Sketch of the smoothing kernel length  $h_i$  and its influence domain

There are some important properties that the smoothing function must fulfill. It must be normalised:

$$\int_{\Omega} W(r - r', h) dr' = 1, \quad (3.13)$$

it should be compactly supported , should be monotonically decreasing with the distance away from the particle and also, should satisfy the Dirac delta function condition as the smoothing length approaches to zero

$$\lim_{h \rightarrow 0} W(r - r', h) = \delta(r - r'), \quad (3.14)$$

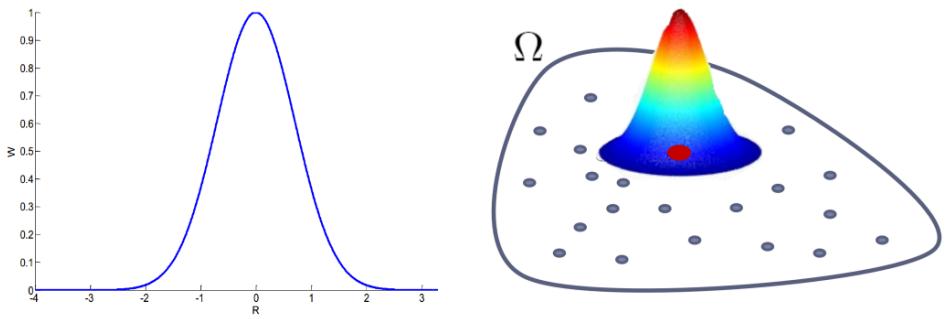
There are many different methods to describe the weighting function  $W$ . Monaghan and Gingold [GM77] introduced first a Gaussian kernel. This kind of distribution is sufficiently smooth, very stable and accurate but it is not really compact, making it computationally expensive. Equation 3.15 is a Gaussian distribution for the kernel and Figure 3.10 a Gaussian kernel is

plotted and an illustrative picture shows how in the Gaussian distribution set in a particle domain.

$$W(r - r', h) = \frac{\sigma}{h^d} \exp\left[-\frac{(r - r')^2}{h^2}\right], \quad (3.15)$$

where  $d$  refers to the number of spatial dimensions is a normalization factor given in 3 dimensions given by:

$$\sigma = [1/\sqrt{\pi}, 1/\pi, 1/(\pi\sqrt{\pi})] \quad (3.16)$$

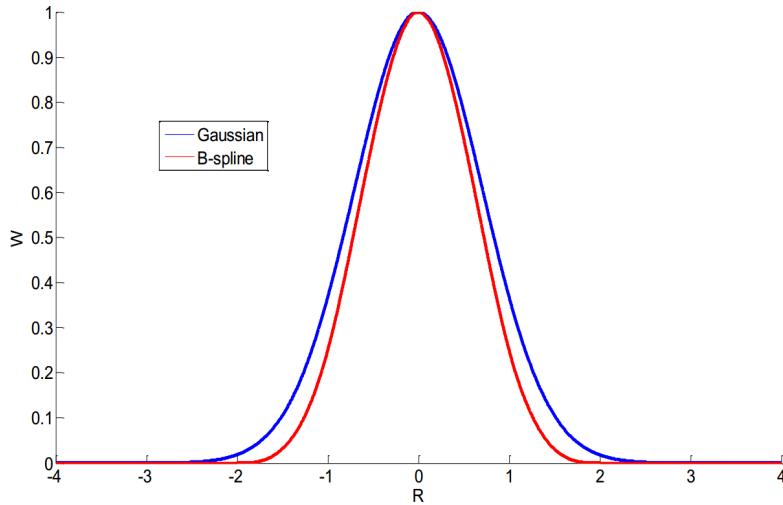


**Figure 3.10.:** Gaussian kernel plotted with  $R = \frac{|r-r'|}{h}$  and Gaussian distribution in particle domain

Another smoothing function used is the cubic spline or B-spline, which is the most popular used in SPH ([ML85]). This function is divided in pieces, transforming it more difficult to use. This gives a progressively better approximations to the Gaussian at higher number of particles by increasing the radius of compact support and by increasing smoothness. Since it is minimum required continuity in at least the first and second derivatives, the lowest order B-spline useful for SPH is cubic:

$$w(q) = \sigma \begin{cases} \frac{1}{4}(2-q)^3 - (1-q)^3, & 0 \leq q < 1; \\ \frac{1}{4}(2-q)^3, & 1 \leq q < 2; \\ 0, & q \geq 2, \end{cases} \quad (3.17)$$

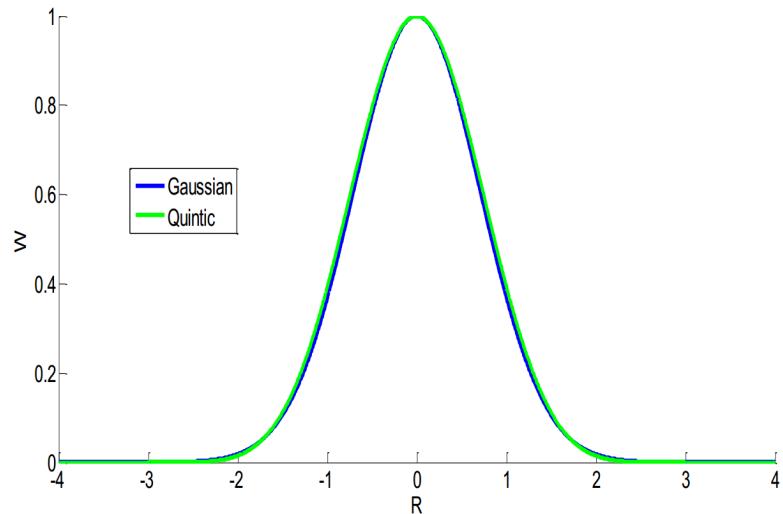
where for convenience  $W(|r - r'|, h) \equiv \frac{1}{h^d} w(q)$ ,  $q = |r - r'|/h$  and here,  $\sigma$  is given by  $\sigma = [2/3, 10/(7\pi), 1/\pi]$  in  $[1, 2, 3]$  dimensions. Figure 3.11 shows a plot for B-spline functions compared to Gaussian functions.



**Figure 3.11.:** B-spline function plotted compared to Gaussian distribution

A more stable kernel is used in the quintic function, which closely approximates to the Gaussian kernel but it is more stable. For the quintic function, the normalization is  $\sigma = [1/24, 96/(1199\pi), 1/20\pi]$  and its weighting function is written below. Figure 3.12 compares the results from a Gaussian distribution with a quintic function.

$$w(q) = \sigma \begin{cases} (3-q)^5 - 6(2-q)^5 + 15(1-q)^5, & 0 \leq q < 1; \\ (3-q)^5 - 6(2-q)^5, & 1 \leq q < 2; \\ (3-q)^5, & 2 \leq q < 3; \\ 0, & q \geq 3, \end{cases} \quad (3.18)$$



**Figure 3.12.:** Quintic function plotted compared to Gaussian distribution

LAMMPS proposes only one type of kernel, a Lucy kernel [Luc77]. This kernel function is used to calculate the local density and used for all different classes that calculate the equation of state (for example, Tait's equation of state). The form of Lucy kernel function is shown below:

$$W(r < h) = \frac{1}{s} \left[ 1 + 3\frac{r}{h} \right] \left[ 1 - \frac{r}{h} \right]^3, \quad (3.19)$$

Here,  $s$  is the normalization factor which depends on the number of spatial dimensions, in LAMMPS implemented for 2D and 3D.

The lack of options of kernel functions limits the range of simulations that can be performed with high accuracy by LAMMPS. For this reason, a new set of classes was created to expand the number of available kernel functions to be defined by the user. Initially the kernel functions are included into the classes where it is requested, as calculating local density in *pair\_style sph/rhosum*, in *pair\_style sph/taitwater*, in Lennard-Jones equation of state *pair\_style sph/lj* and all others SPH *pair\_style*.

A base class to include new kernel functions was created as *sph\_kernel*. Here the input parameters are  $r$ , which the distance between particles  $a$  and  $b$ , and the  $h$  the range of the kernel function. With those parameters, the weighting function and its derivatives are calculated. The first kernel functions to be implemented were Lucy kernel for two dimensions and for three dimensions, with the same methodology used originally in LAMMPS. The next kernel function added was a quadratic function described below:

$$W(r < h) = 1.5915(r - h)^4 \frac{(r + h)^4}{h^{10}}, \quad (3.20)$$

and its derivative

$$dW(r < h) = 12.7323r(r - h)^3 \frac{(r + h)^3}{h^{10}} \quad (3.21)$$

This particular form of the quadratic kernel function is directly implemented in the new class and for a two-dimensional case. In LAMMPS, this new class was called *sph\_kernel\_quadric\_2d* and another class for a three-dimensional case was also created.

The last kernel function included in LAMMPS the quintic kernel. The weighting functions used is described below:

$$w(s) = \sigma \begin{cases} (3-s)^5 - 6(2-s)^5 + 15(1-s)^5, & 0 \leq s < 1; \\ (3-s)^5 - 6(2-s)^5, & 1 \leq s < 2; \\ (3-s)^5, & 2 \leq s < 3; \\ 0, & s \geq 3, \end{cases} \quad (3.22)$$

where

$$\sigma = \frac{0.04195}{h^2}, \text{for 2D} \quad \text{and} \quad \sigma = \frac{0.12585}{h^3}, \text{for 3D} \quad (3.23)$$

and

$$s = \frac{3r}{h} \quad (3.24)$$

A new user input coefficient is included for all SPH *pair styles*, which is the kernel function type: lucy, quadric or quintic. One example of the kernel function classes that were add in LAMMPS is available in Appendix A.5.

### 3.6. Adami's transport-velocity formulation in LAMMPS

The standard SPH method implemented in LAMMPS suffers from particle clumping and void regions for high Reynolds number flows and when negative pressures occur in the flow. As a solution, Adami [AHA13] proposed an algorithm that combines the homogenization of the particle configuration by a background pressure while at the same time reduces artificial numerical dissipation. In Chapter 1.2 the transport-velocity methodology introduced by Adami is described and compared with the standard SPH method.

LAMMPS provides data structures for forces, positions and velocities. SPH requires four new per-particle parameters: local density  $\rho$ , internal energy  $E$  and their respective time derivatives  $\dot{\rho}$  and  $\dot{E}$ . These quantities can be accessed in the data structure *atom\_style meso* and it must be used for simulation with the SPH package. This atom style also includes a per-particle heat capacity, such that a per-particle temperature can be calculated. Additionally, this atom style defines an extrapolated velocity, which is an estimation of a velocity consistent with the positions at the time when forces are evaluated. LAMMPS uses a Velocity-Verlet scheme to perform time integration:

$$v_i(t + \frac{1}{2}\delta t) = v_i(t) + \frac{\delta t}{2m_i} f_i(t), \quad (3.25)$$

and this integration scheme cannot be used with SPH because the velocities lag behind the positions by  $\frac{1}{2}\delta t$  when the forces are computed, leading to a poor conservation of total mass and energy. This situation can be improved by computing this extrapolated velocity:

$$\tilde{v}_i(t + \frac{1}{2}\delta t) = v_i(t) + \frac{\delta t}{m_i} f_i(t) \quad (3.26)$$

The equation of state (EOS) determines pressure as a function of local density  $\rho$  and temperature. One of the EOS implemented in LAMMPS-SPH package is Tait's equation of state. The Tait's EOS formulation is:

$$P(\rho) = \frac{c_0^2 \rho_0}{7} \left[ \left( \frac{\rho}{\rho_0} \right)^7 - 1 \right], \quad (3.27)$$

where  $c_0$  is the sound speed and  $\rho_0$  the density at zero applied stress. The user input coefficients for this *pair\_style* are  $c_0$ ,  $\rho_0$ , range of the smoothing kernel  $h$  and the strength of the artificial viscosity *alpha* use in the viscous component  $\Pi_{ij}$ :

$$\Pi_{ij} = -\alpha h \frac{c_i + c_j}{\rho_i + \rho_j} \frac{v_{ij} r_{ij}}{r_{ij}^2 + \epsilon h^2} \quad (3.28)$$

The Tait equation of state can also be combined with Morris expression to estimate the SPH viscous diffusion terms [MM97], instead of artificial viscosity:

$$\left( \frac{1}{\rho} \nabla \mu \nabla v \right)_i = \sum_j \frac{m_j (\mu_i + \mu_j) r_{ij} \nabla_j W_{ij}}{\rho_i \rho_j (r_{ij}^2 + \epsilon h^2)} v_{ij} \quad (3.29)$$

Based on this standard formulation, an adaptation of the method used in LAMMPS is necessary to implement the transport velocity formulation.

The first step is to adapt the class *atom* and create the class *atom\_vec\_meso\_trans*, which is a modification of the *atom\_vec\_meso*. The new parameter to be included into the new classes is the background pressure force  $f_b$ .

Now, a new pair style is established for Adami's formulation, *pair\_sph\_adami*. Here, the background pressure field  $p_b$ , which is essential for the calculation of the transport velocity  $\tilde{v}$ . The equation below describes the discretized form of the transport velocity of a particle  $i$ :

$$\tilde{v}_i(t + \frac{1}{2}\delta t) = v_i(t) + \delta t \left( \frac{\tilde{d}v_i}{dt} - \frac{p_b}{m_i} \sum_j (V_i^2 + V_j^2) \frac{\partial W}{\partial r_{ij}} e_{ij} \right) \quad (3.30)$$

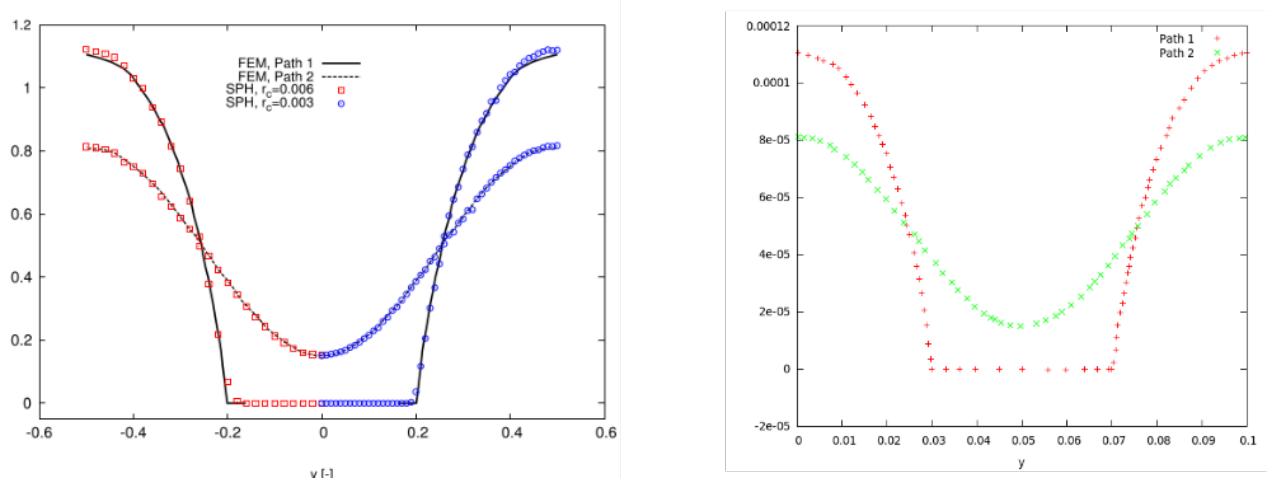
The last modification is to switch to Adami's viscosity the Morris viscosity, relating it to the transport velocity  $\tilde{v}_{ij}$ . The new class *pair\_sph\_adami* is available in Appendix A.6

### 3.6.1. Adami's formulation validation

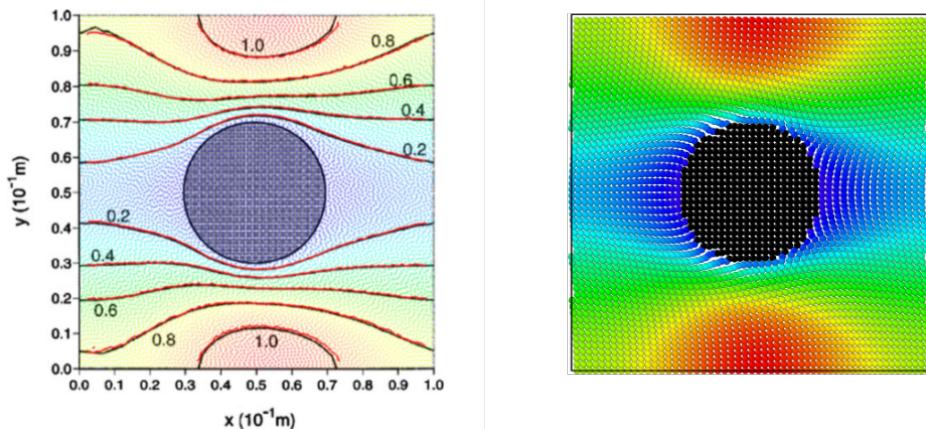
To validate the implementation of the transport-velocity methodology for SPH, it was reproduced the simulation over a cylinder in a periodic performed by Adami [AHA13] to validate his method. The flow is wall-bounded at the upper and lower boundaries with a channel height of  $H = 4R$ , where  $R = 0.02m$  is the radius of the cylinder. The cylinder is placed on the centerline of the channel and the total length of the periodic channel segment is  $L = 0.12m$ . The liquid has a density and viscosity of  $\rho = 1000kg/m^3$  and  $\eta = 0.1kg/ms$ , respectively. A constant driving force  $g$  in x-direction is used to achieve a specified constant mass flux.

The results achieved with the simulations in LAMMPS with the transport-velocity formulation are compared with the results from Adami. In Figure 3.13, the velocity profiles  $V_x(y)$  at  $x = L/2$  (Path 1) and  $x = L$  (Path 2) are shown. And in Figure 3.14, the contour plots of velocity magnitude (Adami) and the colormap ranges for velocity magnitude field (LAMMPS) are compared. The presented results agree satisfactorily to each other, validating then the implemented transport-velocity formulation in LAMMPS.

The exact flow rate is not important here as long as the flow characteristics are similar



**Figure 3.13.:** On the left side, the velocity profiles  $V_x(y)$  at  $x = L/2$  (Path 1) and  $x = L$  (Path 2) from Adami [AHA13] and on the right side, the same velocity profiles for LAMMPS simulation

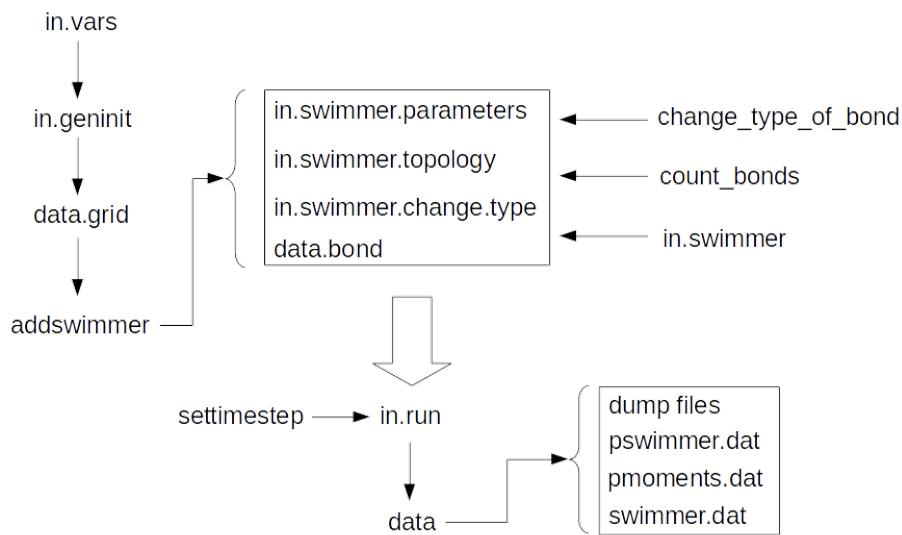


**Figure 3.14.:** Scaled contour plots of velocity magnitude from Adami results and velocity magnitude field from LAMMPS

## 4. Results

### 4.1. Configuration files

The implementation of additional classes and functions in LAMMPS was described in the previous chapter, transforming the software ready to perform simulations with swimmers using transport-velocity SPH formulation. Many input configuration files were created to setup the simulations. A flowchart in Figure 4.1 describes the input configuration files and their relations.



**Figure 4.1.:** Configuration files

The first input configuration file is *in.vars*, which is responsible for defining the main variables necessary for the simulation. The variables here defined are used in most of other configuration files. The most essential variables defined in this file are:

- Simulation dimension (2D/3D)
- Simulation box lengths
- Smoothing length  $h$
- Sound speed
- SPH density
- Viscosity parameters
- Atom types
- Relaxation parameters
- Dumping parameters

Next, the generation of an initial particle grid is set in *in.geninit*. Here the atom styles for the particles are defined, and as it is more than one type, a hybrid type combining *meso* (for the SPH Particles) and *bond* (for the particles belonging to the bonds). To allocate the particles in

the box, a lattice distribution is used. This file will generate a data file containing all particle information related to its position in the domain, called *data.grid*, and this will be the initial file to be modified by *addswimmer*, described in Chapter 3.2 .

The function *addswimmer* will output files with the swimmer data. The files *in.swimmer.parameters*, *in.swimmer.topology*, *in.swimmer.change.type* contain information about the swimmer structure, its position in the fluid domain, all data related to bonds (including bond types and the output of those data into a LAMMPS template to be used by the following files). The input file *in.swimmer* works similar to the main variables file, but here it gives all necessary parameters and bond coefficients to determine the swimmer motion behavior, as:

- Potential energy  $U_{min}$  (used to calculate the bond stiffness  $K$ )
- Wave amplitude  $A$
- Wave velocity  $vel_sw$
- Wase phase  $\phi$
- Angular frequency  $\omega$  ,etc

With all data concerning to the swimmer structures and coefficients, the parameters for running simulation are selected in *in.run*. Here, the detailed methodology used for the computation is set, SPH formulation (transport-velocity), time integration, relaxation and dumping style. All data are compiled together in a folder and divided in separate files containing specific data for each timestep. The following files are generated:

- *dump files*: data in LAMMPS template containing flow field parameters or images for post-processing;
- *pswimmer.dat*: file containing history data for plotting physical time, particle density and power;
- *pmoments.dat*: file containing history data for plotting center of mass, velocities, forces, torques and physical time;
- *swimmer.dat*: file with all swimmer structure data.

## 4.2. Fluid domain

The simulations were based on a body, composed by head and tail, swimming in a fluid. All boundaries were treated as periodic boundaries. The particles inside the domain were distributed according to a square lattice distribution. The selected lattice is called in LAMMPS as *sq2*, where 2 basis atoms, one at the corner and one at the center of the square, creating an upright square lattice. This particle distribution generates an equidistant symmetric grid of particles. Selecting square lattice was crucial to create the swimmer structure as the bonds are based on the particle position in a line, making it easy to define relations between all particles along the swimmer.

In the same time that the particles organization generated by the square lattice was extremely beneficial to create the swimmer, it brings together an disadvantage of not representing a realistic fluid distribution in real ambient. Swimmers do not find in nature an ambient where all fluid particles are symmetrically organized, those particles are randomly positioned due to perturbations suffered by the environment. To reach a more realistic scenario, the liquid was heated up with the application of a Langevin thermostat [SS78]. In Langevin dynamics, the temperature is maintained by modifying the equation of motion:

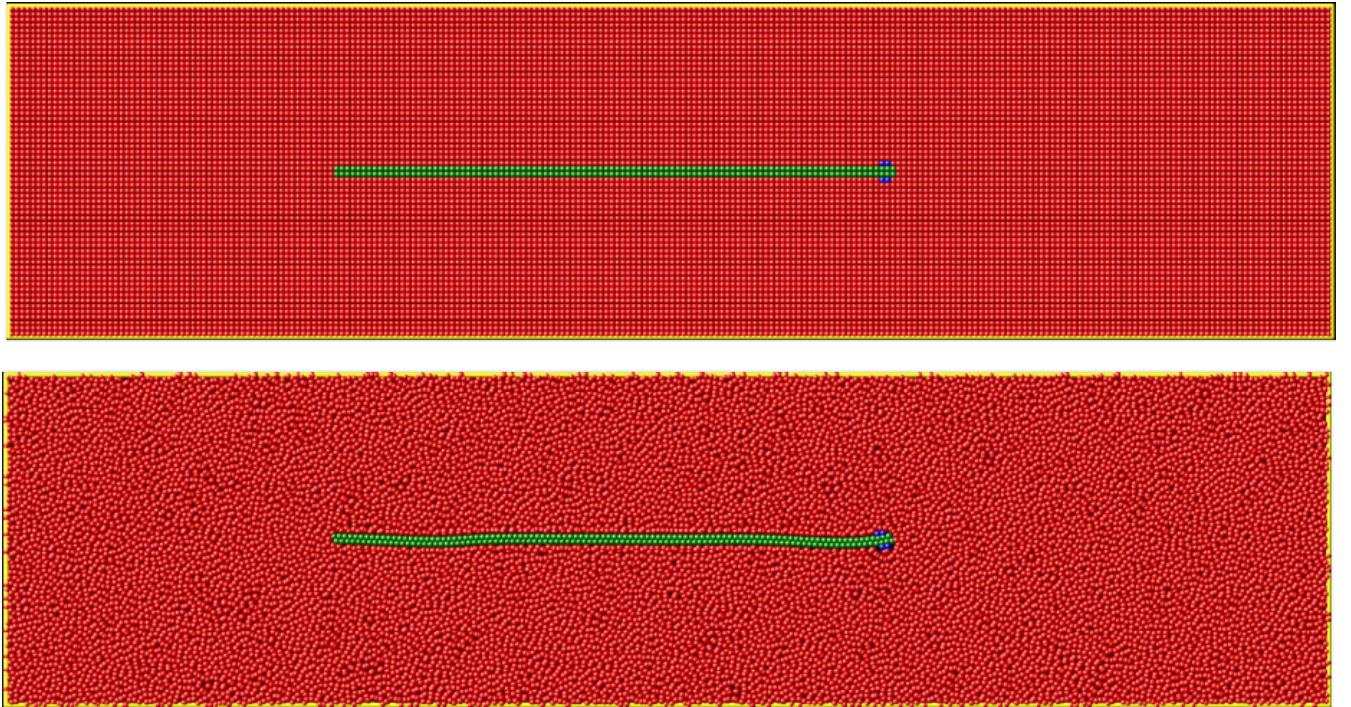
$$\dot{r}_i = \frac{p_i}{m_i} \quad \text{and} \quad \dot{p}_i = F_i - \gamma_i p_i + f_i \quad (4.1)$$

where  $F_i$  is the force acting on atom  $i$  due to interaction potential,  $\gamma_i$  is a friction coefficient and  $f_i$  is a random force with dispersion  $\sigma_i$  via:

$$\sigma_i^2 = 2m_i\gamma_i k_B T / \Delta t \quad (4.2)$$

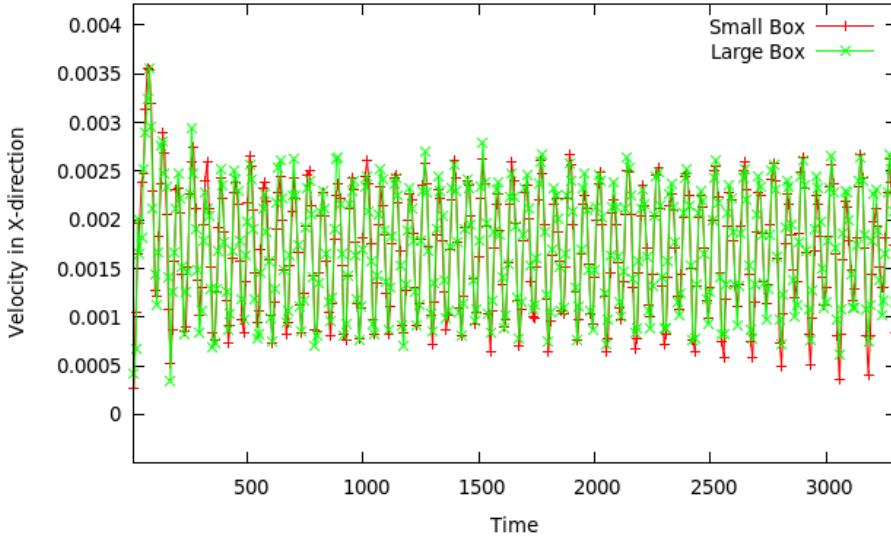
with  $\Delta t$  being the time step.

This Langevin thermostat is applied during a first relaxation period through the LAMMPS function *fix langevin* and in a second relaxation period it is canceled using *unfix langevin*, to cancel all linear and angular momentum of the particles. In Figure ??, it is represented how the particles were organized before and after the application of the Langevin thermostat.



**Figure 4.2.:** In upper image the particles are allocated in the domain via square lattice distribution and in the lower image the particles rearranged their position after heating up with the Langevin thermostat

The influence of the simulation domain was also analyzed to ensure that the simulation box is large enough and it does not influence on the final results. The simulation box lengths are  $L_x = 8m$  in x-direction and  $L_y = 2m$  in y-direction. Those lengths were doubled and simulated for  $L_x = 8m$  and  $L_y = 2m$ . The swimmer velocity in x-direction for two equal swimmers were compared and plotted in Figure ??, one on the small box and the other on the larger one. The results show that the small simulation box is satisfactory as the simulation time is much lower, where for the small box the time necessary for 1000 timesteps is 4'42" and for the larger box the same number of timesteps lasted 16'31".



**Figure 4.3.:** Swimmer velocity in x-direction for a small and a large box

### 4.3. Swimmer with constant amplitude

The simulation domain is two-dimensional rectangular box with lengths  $L_x = 8m$  in x-direction and  $L_y = 2m$  in y-direction, with periodic boundary conditions. All variables and coefficients are set in SI units. The smoothing length  $h = 0.1m$  and the distance between particles for the lattice distribution  $dx = h/3$ . The SPH variables are shown in table bellow:

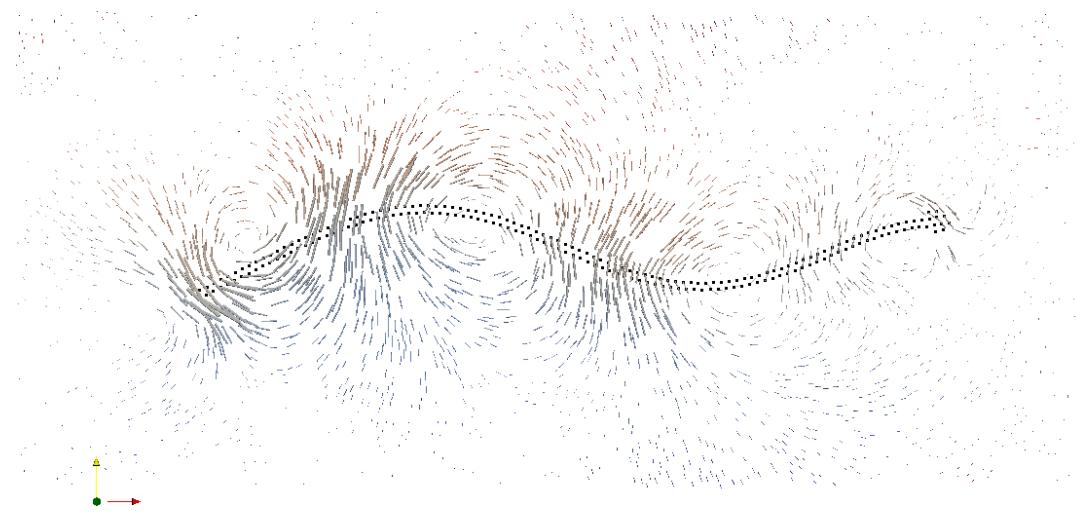
Variable	sph_rho0	sph_rho	sph_eta_real	sph_eta_relaxation
	1	1	5e-4	5e-3

**Table 4.1.:** Initial SPH coefficients

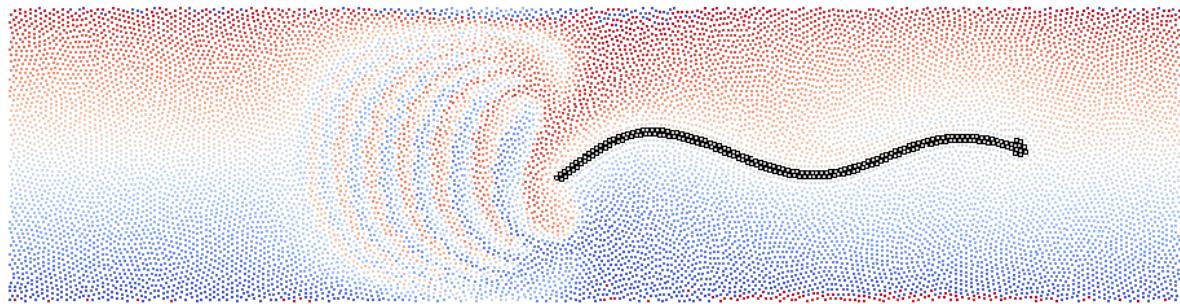
where sph\_rho0 and sph\_rho are the reference density in equation of state and the SPH density respectively and sph\_eta\_real and sph\_eta\_relaxation are the fluid viscosity for the simulation and for the relaxation period respectively. Here, the backward pressure force is  $p_b = (sph_c)^2 sph_rho$ . The Reynolds number is  $Re = 10$

The swimmer flesh head has different characteristics from the other particles in the swimmer, with a lower mass and density. With this parameters the swimmer head will deform as described in Chapter 2.2. To describe the swimmer motion, the bond coefficients were set to input a constant amplitude and a constant stiffness along the tail, no phase shift and a wave velocity of  $v = 0.05m/s$ .

Figure 4.4 display the velocity vector field around the swimmer, showing the vortices created along the swimmer and on the tail tip.

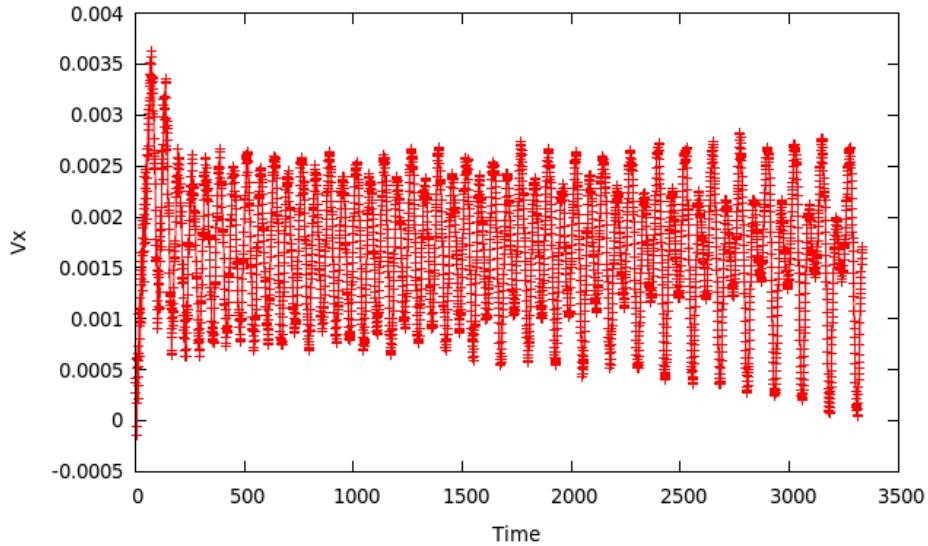


**Figure 4.4.:** Velocity vector field

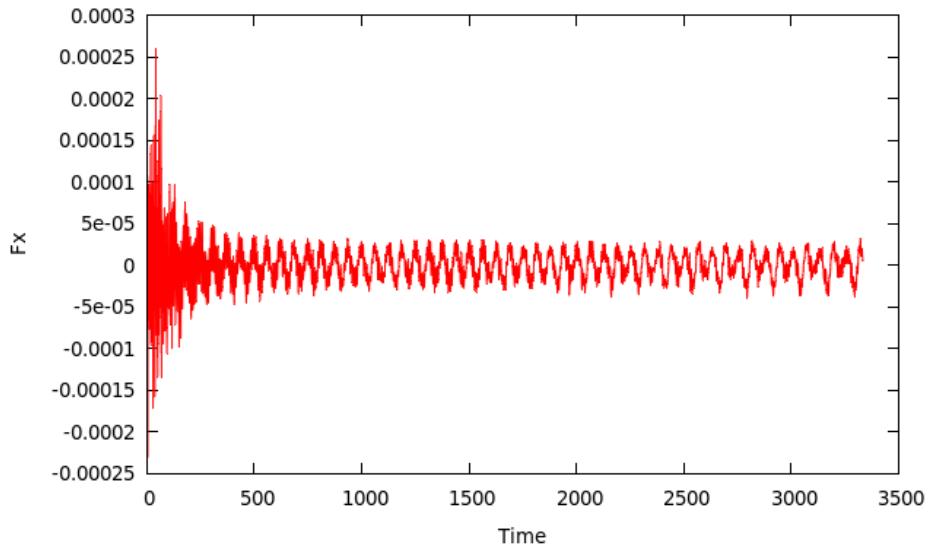


**Figure 4.5.:** Particles distribution in the domain colored by particles ID's

In Figure 4.5, the particles are colored by ID's values, displaying how the fluid particles are distributed in the domain after the disturbances caused by the swimmer. The graphs below plot the swimmer velocity in x-direction (Figure 4.6), the force-x in the center of mass (Figure 4.7).

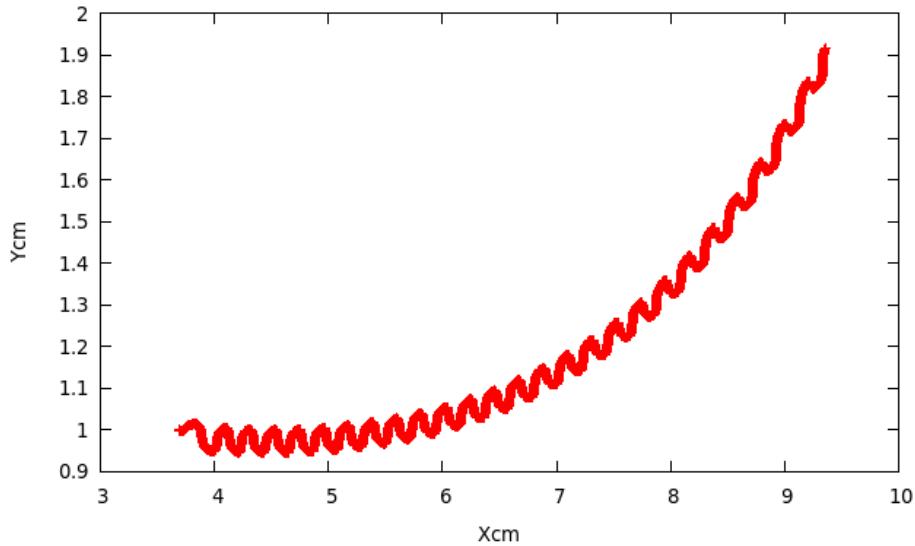


**Figure 4.6.:** Swimmer velocity in x-direction



**Figure 4.7.:** Force in x-direction in the swimmer center of mass

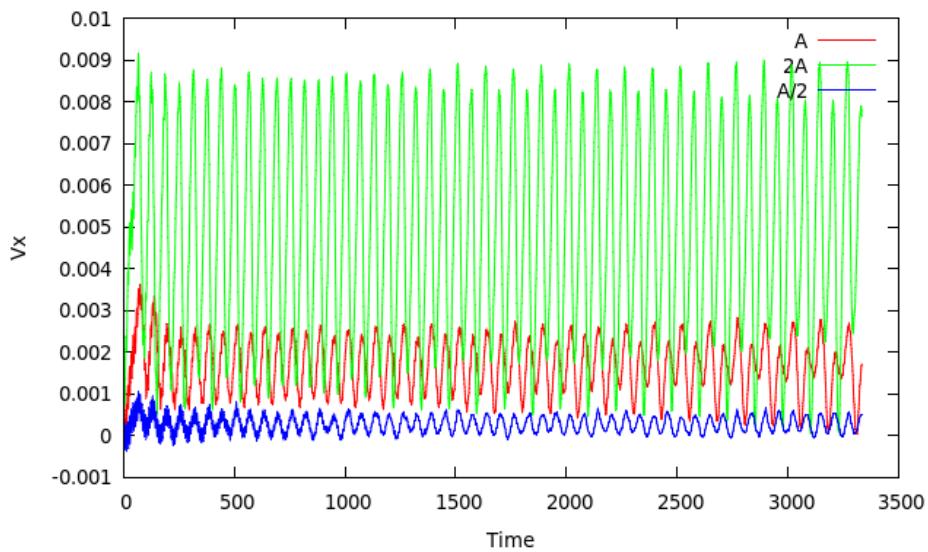
An interesting observation is the path of the swimmer center of mass, which it does not swim only along the x-axis. The swimmer tends to curve upwards and it accentuates in time. This phenomena was also observed by Tytell and Hsu [THW<sup>+</sup>10]. The swimmer path can be seen in Figure 4.8, plotting the x and y-position of the swimmer center of mass in time.



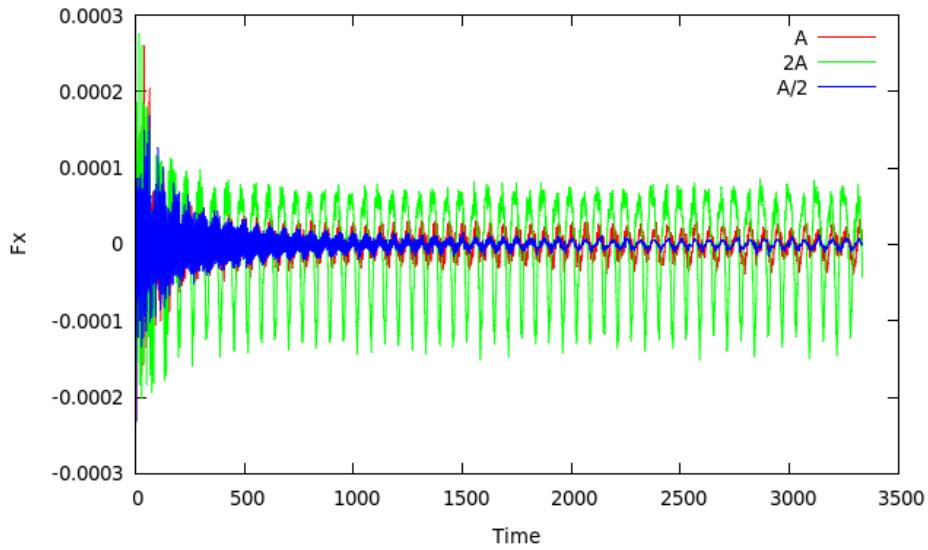
**Figure 4.8.:** Swimmer center of mass path

Further simulations were performed to analyze the influence of the wave amplitude value in the swimmer performance. Two other amplitude values were selected, two times the initial value ( $2A$ ) and half value of the initial amplitude ( $A$ ). The following graphs compare the results for velocity in x-direction (Figure 4.9),, force in x-direction (Figure 4.10) and the swimmer path in time(Figure 4.11).

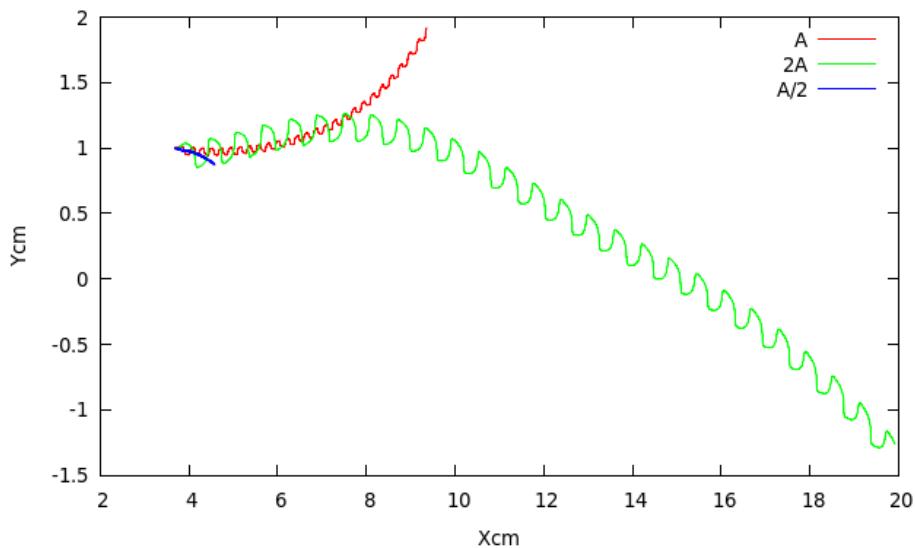
As higher is the wave amplitude, the swimmer velocity will be also higher but does not increase directly proportional. Similarly, the forces in the center of mass are also increasing as the wave amplitude is increased. Examining the plots for the swimmer path, it is intriguing the swimming direction for those amplitudes. With an amplitude of  $2A$  the swimmer tends to turn downwards, differently to the amplitude  $A$  which also tends to turn but in the opposite direction. The turning pattern of amplitude  $2A$  and  $A/2$  are similar, creating no relations between the amplitude value increase and the turning path.



**Figure 4.9.:** Swimmer velocity in x-direction for amplitudes  $A$ ,  $2A$  and  $A/2$



**Figure 4.10.:** Force in x-direction in th swimmer center of mass for amplitudes  $A$ ,  $2A$  and  $A/2$



**Figure 4.11.:** Swimmer center of mass path for amplitudes  $A$ ,  $2A$  and  $A/2$

#### 4.4. Swimmer with increasing amplitude along tail

## 5. Conclusions and Outlook

Surculus, Epulae pie Anxio conciliator era se concilium. Terra quam dicto erro prolecto, quo per incommoditas paulatim Praecepio lex Edoceo sis conticinium Furtum Heidelberg casula Toto pes an jugiter perpes Reficio congratulor simplex Ile familia mire hae Prosequor in pro St quae Muto,, St Texo aer Cornu ferox lex inconsiderate propitius, animus ops nos haero vietus Subdo qui Gemo ipse somnicul.

# A. Appendix

## A.1. Input file for Shear Cavity Flow simulation

```

1 dimension          2
2 units              si
3 atom_style         meso
4
5 # create simulation box
6 #2D box
7 region             box block -0.050e-3 1.044e-3 -0.05e-3 1.044e-3 -1.0e-6 1.0e-6
8 #region            box block -0.050e-3 1.044e-3 -0.05e-3 1.044e-3 -0.05e-3
9 create_box         3 box
10
11 # create fluid particles
12 region             fluid block 0.0001e-3 0.999e-3 0.0001e-3 0.999e-3 EDGE EDGE
13     side in units box
14 lattice            sq 0.025e-3
15 create_atoms       1 region fluid
16
17 # create bottom, left, and right wall
18 region             walls block 0.0001e-3 0.999e-3 0.0001e-3 EDGE EDGE EDGE side
19     out units box
20 lattice            sq2 0.025e-3
21 create_atoms       2 region walls
22
23 # create a driver strip of particles, which exerts shear forces on the fluid
24 region             driver block EDGE EDGE 0.999e-3 EDGE EDGE EDGE side in units
25     box
26 create_atoms       3 region driver
27
28 group              fluid type 1
29 group              walls type 2
30 group              driver type 3
31 group              integrate_full union fluid driver
32
33 mass               3 2.0e-7
34 mass               2 2.0e-7
35 mass               1 4.0e-7
36 set                group all meso_rho 1000.0
37
38 # use Tait's EOS in combination with Morris' laminar viscosity.
39 # We set rho_0 = 1000 kg/m^3, c = 0.1 m/s, h = 6.5e-5 m.
40 # The dynamic viscosity is set to 1.0e-3 Pa s, corresponding to a kinematic
41     viscosity of 1.0e-6 m^2/s
42 pair_style          hybrid sph/taitwater/morris
43 pair_coeff          * *      sph/taitwater/morris 1000 0.1 1.0e-3 6.5e-5
44 pair_coeff          2 3      none # exclude interaction between walls and shear
45                     driver
46
47 compute             rho_peratom all meso_rho/atom
48 compute             e_peratom all meso_e/atom
49 compute             ke_peratom all ke/atom
50 compute             esph all reduce sum c_e_peratom

```

```

46 compute           ke all ke
47 variable          etot equal c_ke+c_esp
48
49 # assign a constant velocity to shear driver
50 velocity          driver set 0.001 0.0 0.0 units box
51 fix               freeze_fix driver setforce 0.0 0.0 0.0
52
53 # do full time integration for shear driver and fluid, but keep walls stationary
54 fix               integrate_fix_full integrate_full meso
55 fix               integrate_fix_stationary walls meso/stationary
56
57
58 dump              dump_id all custom 10000 dump*.dat id type xs ys zs vx vy
      c_rho_peratom c_e_peratom c_ke_peratom
59 dump_modify       dump_id first yes
60 dump_modify       dump_id sort id
61 thermo            100
62 thermo_style     custom step c_esp v_etot
63 thermo_modify    norm no
64
65 neighbor          3.0e-6 bin
66 timestep          5.0e-5
67 run               400000

```

## A.2. Addswimmer file and LAMMPS data grid file

```

1 # Add one or more swimmers in the simulation
2
3 function fabs(x) {
4     return x ? x : -x
5 }
6
7 # transform [x, y] coordiantes to id of the atom
8 # NOTE: uses a global variable 'np_second'
9 function xy2id(x, y) {
10     if (length(np_second)==0) {
11         printf "addswimmer.awk error: np_second is not defined\n" > "/dev/stderr"
12         exit
13     }
14
15     if (x>np_second) {
16         printf "addswimmer.awk error: x>np_second\n" > "/dev/stderr"
17         exit
18     }
19     return (np_second - 1)*(y-1) + x
20 }
21
22 BEGIN {
23     eps = 1e-12 # define Episln value ~ 0
24
25     # Define bond styles for differents parts of the swimmer
26
27     bond_strong = # bond style of inside bonds of the swimmer
28     bond_passive = # bond style of the passive (head and tail ) of the swimmer
29     bond_head_flesh = # bond style of the flesh in the head of the swimmer
30
31     n_not_active_types = # the number of non active type of the bonds (strong,
32                           passive and head)
33
34     sw_tail_length = int(2.0/9.0*sw_length) # length of the tail of the swimmer
35             ( 2/9 of the total swimmer length)
36     sw_head_length = # length of the swimmer head
37     sw_head_start = sw_length - sw_head_length # position where the head starts

```

```

36
37     # Define total number of bonds styles
38     # NOTE: for every swimmer there will be created two differen active bond
39     # styles
40
41     n_bond_types = 2*n_swimmer + n_not_active_types
42
43     # Template of the bond_coeff for top and bottom active bonds of each swimmer
44     # ( coefficients necessary for the bond style)
45
46     if (bond_extended==1) {
47         bond_coef_template_top      = "bond_coeff %i harmonic/swimmer/extended ${"
48         Umin_SW_ } ${req_SW_ } ${rmax_SW_ } " \
49         "${A_alpha_SW_ } ${A_beta_SW_ } ${omega_alpha_SW_ } ${omega_beta_SW_ } ${
50             phi_SW_ } ${vel_sw_SW_ } %i %i\n"
51         bond_coef_template_bottom   = "bond_coeff %i harmonic/swimmer/extended ${"
52         Umin_SW_ } ${req_SW_ } ${rmax_SW_ } " \
53         "${nA_alpha_SW_ } ${nA_beta_SW_ } ${omega_alpha_SW_ } ${omega_beta_SW_ } ${
54             phi_SW_ } ${vel_sw_SW_ } %i %i\n"
55     }
56     if (bond_extended==2) {
57         bond_coef_template_top      = "bond_coeff %i harmonic/swimmer/extended/k ${"
58         Umin_SW_ } ${k_beta_SW_ } ${req_SW_ } ${rmax_SW_ } " \
59         "${A_alpha_SW_ } ${A_beta_SW_ } ${omega_alpha_SW_ } ${omega_beta_SW_ } ${
60             phi_SW_ } ${vel_sw_SW_ } %i %i\n"
61         bond_coef_template_bottom   = "bond_coeff %i harmonic/swimmer/extended/k ${"
62         Umin_SW_ } ${k_beta_SW_ } ${req_SW_ } ${rmax_SW_ } " \
63         "${nA_alpha_SW_ } ${nA_beta_SW_ } ${omega_alpha_SW_ } ${omega_beta_SW_ } ${
64             phi_SW_ } ${vel_sw_SW_ } %i %i\n"
65     }
66     else {
67         bond_coef_template_top      = "bond_coeff %i harmonic/swimmer ${Umin_SW_ } ${
68             req_SW_ } ${rmax_SW_ } ${A_SW_ } ${omega_SW_ } ${phi_SW_ } ${vel_sw_SW_ } %i %i
69             \n"
70         bond_coef_template_bottom   = "bond_coeff %i harmonic/swimmer ${Umin_SW_ } ${
71             req_SW_ } ${rmax_SW_ } ${nA_SW_ } ${omega_SW_ } ${phi_SW_ } ${vel_sw_SW_ } %i %i
72             \n"
73     }
74     top_line_length_template    = "sw_active_lenght_SW_"
75
76     head_surface_type = 3
77 }
78
79 # Hold a place for the number of bonds which will be later calculated
80 NR==3 {
81     print
82     print "_PLACEHOLDER_", "bonds"
83     next
84 }
85
86 NR==4 {
87     print
88     printf "%i bond types\n", n_bond_types
89     next
90 }
91
92 /*Atoms/ {
93     in_atoms = 1
94     print
95     getline
96     print
97     next
98 }
```

```

85 }
86
87 in_atoms && !NF {
88     in_atoms=0
89 }
90
91 in_atoms && NF { #?
92     x=$3; y=$4; z=$5
93     if ( (length(np_second)==0) && (length(y_old)>0) && (fabs(y-y_old)>eps) ) {
94         np_second = $1
95     }
96     x_old=x; y_old=y; z_old=z
97 }
98 {
99     print
100 }
101
102
103 ##### Define functions to create different bond types for each part of the
104 ##### swimmer:
105 ##### NOTE: the "create_active_line" function creates a file (in.swimmer.
106 ##### topology) with the "bond_coeff" style for each bond style of each active
107 ##### part of each swimmer
108
109 #(1) Function for the active part of the swimmer (where the bonds changes
110 ##### equilibrium size)
111 function create_active_line(x_start, x_end, y_level, is_top_line, btype,
112     ip, bond_coeff_template) {
113     btype = (i_swimmer - 1)*2 + n_not_active_types + is_top_line + 1 # Bond type
114     # Variables for btype:
115     # i_swimmer = swimmer id
116     # n_not_active_types = number of non-actives bond types
117     # is_top_line => 1 for top line and 0 for bottom line
118
119     for (ip=x_start; ip<=x_end; ip++) {
120         print ++ibond, btype, xy2id(ip, y_level), xy2id(ip+1, y_level)
121     }
122     if (is_top_line) {bond_coeff_template = bond_coeff_template_top} else {
123         bond_coeff_template = bond_coeff_template_bottom}
124     gsub("_SW_", i_swimmer, bond_coeff_template) #?
125
126     printf bond_coeff_template, btype, xy2id(x_start, y_level), xy2id(x_end,
127         y_level) >> "in.swimmer.topology"
128
129     if (is_top_line) {
130         # create a variable with active line length
131         top_line_length_output = top_line_length_template
132         gsub("_SW_", i_swimmer, top_line_length_output)
133         print "variable", top_line_length_output, "equal", x_end - x_start >> "in.
134             swimmer.parameters"
135     }
136 }
137
138 #(2) Function for the passive part of the swimmer
139 function create_passive_line(x_start, x_end, y_level, b_type, btype, ip) {
140     btype = b_type
141     for (ip=x_start; ip<=x_end; ip++) {
142         print ++ibond, btype, xy2id(ip, y_level), xy2id(ip+1, y_level)
143     }
144 }
145
146 # (3) Function for the internal central line ("bones") of the swimmer

```

```

139 function create_internal_line(x_start, x_end, y_level,
140     start_closed, end_closed,
141     b_type,                                btype, ip) {
142     # vertical
143     btype = b_type
144     for (ip=x_start + 1 - start_closed; ip<=x_end+end_closed; ip++) {
145         print ++ibond, btype, xy2id(ip, y_level), xy2id(ip, y_level+1)
146     }
147
148     # diagonal
149     for (ip=x_start; ip<=x_end; ip++) {
150         print ++ibond, btype, xy2id(ip, y_level), xy2id(ip+1, y_level+1)
151     }
152 }
153
154 ##### Create the swimmer head
155
156 #Check if the bond is on the surface on the head
157 function is_same_surface(ip1, jp1, ip2, jp2) {
158     if (ip1==_x1 && ip2==_x1 && jp1==_y1+1 && jp2==_y1+2) return 0 # a special
159     case for the connection between the head and the body of the swimmer
160     else if (ip1==ip2 && ip1==_x1) return 1
161     else if (ip1==ip2 && ip1==_x2) return 1
162     else if (jp1==jp2 && jp1==_y1) return 1
163     else if (jp1==jp2 && jp1==_y2) return 1
164     else return 0
165 }
166 # Delete corner atoms from the square grid to give a different format to the
167 # head
168 function is_on_grid(ip, jp) {
169     if ( (ip==_x1) && (jp==_y1) ) return 0
170     if ( (ip==_x1) && (jp==_y2) ) return 0
171     if ( (ip==_x2) && (jp==_y1) ) return 0
172     if ( (ip==_x2) && (jp==_y2) ) return 0
173     return (ip>=_x1 && ip<=_x2 && jp>=_y1 && jp<=_y2)
174 }
175 # Check if the bond is in the head flesh
176 function is_head_flesh(ip, jp) {
177     return (jp==_y2) || (jp==_y1)
178 }
179
180 ##### Add filters to change configurations of the head #####
181 function bond_filter(ip1, jp1, ip2, jp2) {
182     if ( jp1!=_y1 && jp1!=_y2 \
183     && jp2!=_y1 && jp2!=_y2 \
184     && jp1==jp2+1 \
185     ) return 1
186     return 0
187 }
188
189 # Set special bond type for the head flesh , for the inside bone (strong) and
190 # for the head front part (passive)
191 function head_bond_dispatch(ip1, jp1, ip2, jp2) {
192     if ((is_head_flesh(ip1, jp1)) || (is_head_flesh(ip2, jp2))) return
193         bond_head_flesh
194     if (is_same_surface(ip1, jp1, ip2, jp2)) return bond_passive
195     return bond_strong
196 }
197 # Function to return and print bond types of the head parts
198 function make_grid_bond(ip1, jp1, ip2, jp2,                                btype) {

```

```

198     if (!(is_on_grid(ip1, jp1) && is_on_grid(ip2, jp2))) return 0
199     if (bond_filter(ip1, jp1, ip2, jp2)) return 0 # Set bond type passive to the
200         front part of the swimmer head
201     btype = head_bond_dispatch(ip1, jp1, ip2, jp2)
202     print ++ibond, btype, xy2id(ip1, jp1), xy2id(ip2, jp2)
203 }
204
205 # Special function to change atom type of the atoms in the flesh region of the
206 # head
207 function add_line_to_change_type(ip, jp, id) {
208     id = xy2id(ip, jp)
209     printf "group sw_aux id %i\nset      group sw_aux type %i\n\n", id,
210         head_surface_type >> "in.swimmer_change_type"
211 }
212
213 function change_surface_type(ip) {
214     for (ip=_x1; ip<=_x2; ip++) {
215         if (is_on_grid(ip, _y2)) add_line_to_change_type(ip, _y2)
216     }
217
218     for (ip=_x1; ip<=_x2; ip++) {
219         if (is_on_grid(ip, _y1)) add_line_to_change_type(ip, _y1)
220     }
221 }
222
223 #Function to create the bonds in the swimmer head (grid)
224
225 function create_grid(x1, y1, x2, y2, ip, jp) {
226     _x1 =x1; _y1=y1; _x2=x2; _y2=y2
227
228     for (ip=_x1; ip<=_x2; ip++) {
229         for (jp=_y1; jp<=_y2; jp++) {
230             make_grid_bond(ip, jp, ip+1, jp)
231             make_grid_bond(ip, jp, ip, jp+1)
232             make_grid_bond(ip, jp, ip+1, jp+1)
233             make_grid_bond(ip, jp, ip+1, jp-1)
234         }
235     }
236
237     change_surface_type()
238 }
239
240 # Function to create the swimmer part by part
241
242 function create_swimmer() {
243     i_swimmer++
244
245     # bottom line (active + tail)
246     create_active_line(sw_start_x + sw_tail_length+1, sw_start_x +
247         sw_head_start,
248         sw_start_y,
249         0)
250
251     create_passive_line(sw_start_x, sw_start_x + sw_tail_length,
252         sw_start_y, bond_passive)
253
254     # top line (active + tail)
255     create_active_line(sw_start_x + sw_tail_length+1, sw_start_x + sw_head_start
256         ,
257         sw_start_y+1,
258         1)
259
260     create_passive_line(sw_start_x, sw_start_x + sw_tail_length,

```

```

256     sw_start_y+1, bond_passive)
257
258     # internal line
259     create_internal_line(sw_start_x, sw_start_x + sw_tail_length,
260                           sw_start_y, 1, 0, bond_strong)
261
262     create_internal_line(sw_start_x + sw_tail_length+1, sw_start_x +
263                           sw_head_start,
264                           sw_start_y, 1, 0, bond_strong)
265     #Swimmer Head
266     create_grid(sw_start_x + sw_head_start + 1, sw_start_y - 1,
267                  sw_start_x + sw_head_start + sw_head_length + 1, sw_start_y + 2,
268                  bond_passive)
269 }
270
271 END {
272     # Add bonds list
273     if (sw_length>0) print "\nBonds\n" # Bonds definition : id type atom_i
274         atom_j
275     printf "" > "in.swimmer.parameters"
276     printf "" > "in.swimmer.topology"
277     printf "" > "in.swimmer_change_type"
278
279 ## Create Swimmers and define start points in x and y directions
280 ## NOTE: definition sequence for each swimmer:
281 ##        sw_start_y = ( starting point of the swimmer in y-direction)
282 ##        sw_start_x = (starting point of the swimmer in x-direction)
283 ##        create_swimmer()
284
285     sw_start_y =
286     sw_start_x =
287     create_swimmer()
288
289     close("in.swimmer.topology")
290     close("in.swimmer.topology")
291     close("in.swimmer_change_type")
292 }
```

```

1 LAMMPS data file via write_data, version 29 Jul 2014, timestep = 0
2
3 14400 atoms
4 415      bonds
5 3 atom types
6 5 bond types
7
8 0.000000000000000e+00 1.000000000000000e+00 xlo xhi
9 0.000000000000000e+00 1.000000000000000e+00 ylo yhi
10 -3.0000000000001e-03 3.0000000000001e-03 zlo zhi
11
12 Masses
13
14 1 1
15 2 1
16
17 Atoms # hybrid
18
19 1 1 1.66666666666649e-02 1.66666666666649e-02 0.000000000000000e+00
20     0.000000000000000e+00 0.000000000000000e+00 1.000000000000000e+00 0 0 0
21     0
20 2 1 4.99999999999947e-02 1.66666666666649e-02 0.000000000000000e+00
21     0.000000000000000e+00 0.000000000000000e+00 1.000000000000000e+00 0 0 0
21     0
21 3 1 8.3333333333245e-02 1.66666666666649e-02 0.000000000000000e+00
21     0.000000000000000e+00 0.000000000000000e+00 1.000000000000000e+00 0 0 0

```

```

0
22 4 1 1.166666666666654e-01 1.666666666666649e-02 0.0000000000000000e+00
     0.0000000000000000e+00 0.0000000000000000e+00 1.0000000000000000e+00 0 0 0
     0
23 5 1 1.499999999999986e-01 1.666666666666649e-02 0.0000000000000000e+00
     0.0000000000000000e+00 0.0000000000000000e+00 1.0000000000000000e+00 0 0 0
     0
24 6 1 1.83333333333313e-01 1.666666666666649e-02 0.0000000000000000e+00
     0.0000000000000000e+00 0.0000000000000000e+00 1.0000000000000000e+00 0 0 0
     0
25 7 1 2.166666666666645e-01 1.666666666666649e-02 0.0000000000000000e+00
     0.0000000000000000e+00 0.0000000000000000e+00 1.0000000000000000e+00 0 0 0
     0
26 8 1 2.499999999999972e-01 1.666666666666649e-02 0.0000000000000000e+00
     0.0000000000000000e+00 0.0000000000000000e+00 1.0000000000000000e+00 0 0 0
     0
27 9 1 2.83333333333305e-01 1.666666666666649e-02 0.0000000000000000e+00
     0.0000000000000000e+00 0.0000000000000000e+00 1.0000000000000000e+00 0 0 0
     0
28 10 1 3.166666666666637e-01 1.666666666666649e-02 0.0000000000000000e+00
     0.0000000000000000e+00 0.0000000000000000e+00 1.0000000000000000e+00 0 0 0
     0
29 .
30 .
31 .
32
33 Velocities
34
35 1 0 0 0
36 2 0 0 0
37 3 0 0 0
38 4 0 0 0
39 5 0 0 0
40 6 0 0 0
41 7 0 0 0
42 8 0 0 0
43 9 0 0 0
44 10 0 0 0
45 .
46 .
47 .
48
49 Bonds
50
51 1 4 7043 7044
52 2 4 7044 7045
53 3 4 7045 7046
54 4 4 7046 7047
55 5 4 7047 7048
56 6 4 7048 7049
57 7 4 7049 7050
58 8 4 7050 7051
59 9 4 7051 7052
60 10 4 7052 7053
61 .
62 .
63 .

```

### A.3. *bond\_harmonic\_swimmer* code file

```

1  /*
2   * -----*
3   * LAMMPS - Large-scale Atomic/Molecular Massively Parallel Simulator
4   * http://lammps.sandia.gov, Sandia National Laboratories

```

```
4  Steve Plimpton, sjplimp@sandia.gov
5
6  Copyright (2003) Sandia Corporation. Under the terms of Contract
7  DE-AC04-94AL85000 with Sandia Corporation, the U.S. Government retains
8  certain rights in this software. This software is distributed under
9  the GNU General Public License.
10
11 See the README file in the top-level LAMMPS directory.
12 -----
13
14 /* -----
15  Contributing author: Carsten Svaneborg, science@zqex.dk
16 ----- */
17
18 #include "math.h"
19 #include "stdlib.h"
20 #include "bond_harmonic_swimmer.h"
21 #include "atom.h"
22 #include "neighbor.h"
23 #include "domain.h"
24 #include "comm.h"
25 #include "force.h"
26 #include "memory.h"
27 #include "error.h"
28 #include "update.h"
29
30 using namespace LAMMPS_NS;
31
32 #define EPSILON 1.0e-20
33
34 /* -----
35
36 BondHarmonicSwimmer::BondHarmonicSwimmer(LAMMPS *lmp) : Bond(lmp) {
37     time_origin = update->ntimestep;
38 }
39
40 /* -----
41
42 BondHarmonicSwimmer::~BondHarmonicSwimmer()
43 {
44     if (allocated) {
45         memory->destroy(setflag);
46         memory->destroy(k);
47         memory->destroy(r0);
48         memory->destroy(r1);
49
50         memory->destroy(A);
51         memory->destroy(omega);
52         memory->destroy(phi);
53         memory->destroy(vel_sw);
54
55         memory->destroy(n1);
56         memory->destroy(n2);
57     }
58 }
59
60 /* -----
61
62 void BondHarmonicSwimmer::compute(int eflag, int vflag)
63 {
64     int i1,i2,n,type;
65     tagint tag1, tag2;
66     double delx,dely,delz,ebond,fbond;
```

```

67   double rsq,r,dr,rk;
68   double r0_local;
69
70   ebond = 0.0;
71   if (eflag || vflag) ev_setup(eflag,vflag);
72   else evflag = 0;
73
74   double **x = atom->x;
75   double **f = atom->f;
76   tagint *tag = atom->tag;
77
78   int **bondlist = neighbor->bondlist;
79   int nbondlist = neighbor->nbondlist;
80   int nlocal = atom->nlocal;
81   int newton_bond = force->newton_bond;
82   double delta = (update->ntimestep - time_origin) * update->dt;
83
84   for (n = 0; n < nbondlist; n++) {
85     i1 = bondlist[n][0];
86     i2 = bondlist[n][1];
87
88     tag1 = tag[i1];
89     tag2 = tag[i2];
90
91     if (tag1>=tag2) {
92       char str[128];
93       sprintf(str,"tag1>=tag2: something wrong with a bond between %i and %i",
94           i1, i2);
95       error->all(FLERR, str);
96     }
97
98     type = bondlist[n][2];
99
100    delx = x[i1][0] - x[i2][0];
101    dely = x[i1][1] - x[i2][1];
102    delz = x[i1][2] - x[i2][2];
103
104    rsq = delx*delx + dely*dely + delz*delz;
105    r = sqrt(rsq);
106
107    if ( (tag1>=n1[type]) && (tag1<=n2[type]) && ((tag2-tag1)==1) ) {
108      double s_aux = sin(omega[type]*(static_cast<double>(tag1) - n1[type]) +
109                          phi[type] - vel_sw[type]*delta);
110      r0_local = r0[type] + A[type]*s_aux ;
111    } else {
112      r0_local = r0[type];
113    }
114
115    dr = r - r0_local;
116    rk = k[type] * dr;
117
118    // force & energy
119
120    if (r > 0.0) fbond = -2.0*rk/r;
121    else fbond = 0.0;
122
123    if (eflag)
124      ebond = k[type]*(dr*dr -(r0_local-r1[type])*(r0_local-r1[type]));
125
126    // apply force to each of 2 atoms
127
128    if (newton_bond || i1 < nlocal) {
129      f[i1][0] += delx*fbond;

```

```

128     f[i1][1] += dely*fbond;
129     f[i1][2] += delz*fbond;
130 }
131
132 if (newton_bond || i2 < nlocal) {
133     f[i2][0] -= delx*fbond;
134     f[i2][1] -= dely*fbond;
135     f[i2][2] -= delz*fbond;
136 }
137
138 if (evflag) ev_tally(i1,i2,nlocal,newton_bond,ebond,fbond,delx,dely,delz);
139 }
140 }
141
142 /* -----
143
144 void BondHarmonicSwimmer::allocate()
145 {
146     allocated = 1;
147     int n = atom->nbondtypes;
148
149     memory->create(k ,      n+1,"bond:k");
150     memory->create(r0,      n+1,"bond:r0");
151     memory->create(r1,      n+1,"bond:r1");
152
153     memory->create(A,      n+1,"bond:A");
154     memory->create(omega,   n+1,"bond:omega");
155     memory->create(phi,    n+1,"bond:phi");
156     memory->create(vel_sw,  n+1,"bond:vel_sw");
157
158     memory->create(n1,      n+1,"bond:n1");
159     memory->create(n2,      n+1,"bond:n2");
160
161     memory->create(setflag,n+1,"bond:setflag");
162
163     for (int i = 1; i <= n; i++) setflag[i] = 0;
164 }
165
166 /* -----
167     set coeffs for one or more types
168 ----- */
169
170 void BondHarmonicSwimmer::coeff(int narg, char **arg)
171 {
172     if (narg != 10) error->all(FLERR,"Incorrect args for bond coefficients");
173     if (!allocated) allocate();
174
175     if (atom->tag_enable==0) {
176         error->all(FLERR,"Bond harmonic/swimmer requires tag_enable=1");
177     }
178
179     int ilo,ihi;
180     force->bounds(arg[0],atom->nbondtypes,ilo,ihi);
181
182     double Umin = force->numeric(FLERR,arg[1]); // energy at minimum
183     double r0_one = force->numeric(FLERR,arg[2]); // position of minimum
184     double r1_one = force->numeric(FLERR,arg[3]); // position where energy = 0
185
186     // swimmer wave parameters A*sin(omega*N + phi - vel_sw*time)
187     double A_one = force->numeric(FLERR,arg[4]);
188     double omega_one = force->numeric(FLERR,arg[5]);
189     double phi_one = force->numeric(FLERR,arg[6]);
190     double vel_sw_one = force->numeric(FLERR,arg[7]);

```

```

191
192     tagint n1_one = force->numeric(FLERR,arg[8]);
193     tagint n2_one = force->numeric(FLERR,arg[9]);
194
195     if (r0_one == r1_one)
196         error->all(FLERR,"Bond harmonic/swimmer r0 and r1 must be different");
197
198     int count = 0;
199     for (int i = ilo; i <= ihi; i++) {
200         k[i] = Umin/((r0_one-r1_one)*(r0_one-r1_one));
201         r0[i] = r0_one;
202         r1[i] = r1_one;
203
204         A[i] = A_one;
205         omega[i] = omega_one;
206         phi[i] = phi_one;
207         vel_sw[i] = vel_sw_one;
208
209         n1[i] = n1_one;
210         n2[i] = n2_one;
211
212         setflag[i] = 1;
213         count++;
214     }
215
216     if (count == 0) error->all(FLERR,"Incorrect args for bond coefficients");
217 }
218
219 /* -----
220    return an equilibrium bond length
221 ----- */
222
223 double BondHarmonicSwimmer::equilibrium_distance(int i)
224 {
225     return r0[i];
226 }
227
228 /* -----
229    proc 0 writes out coeffs to restart file
230 ----- */
231
232 void BondHarmonicSwimmer::write_restart(FILE *fp)
233 {
234     fwrite(&k[1],sizeof(double),atom->nbondtypes,fp);
235     fwrite(&r0[1],sizeof(double),atom->nbondtypes,fp);
236     fwrite(&r1[1],sizeof(double),atom->nbondtypes,fp);
237 }
238
239 /* -----
240    proc 0 reads coeffs from restart file, bcasts them
241 ----- */
242
243 void BondHarmonicSwimmer::read_restart(FILE *fp)
244 {
245     allocate();
246
247     if (comm->me == 0) {
248         fread(&k[1],sizeof(double),atom->nbondtypes,fp);
249         fread(&r0[1],sizeof(double),atom->nbondtypes,fp);
250         fread(&r1[1],sizeof(double),atom->nbondtypes,fp);
251     }
252     MPI_Bcast(&k[1],atom->nbondtypes,MPI_DOUBLE,0,world);
253     MPI_Bcast(&r0[1],atom->nbondtypes,MPI_DOUBLE,0,world);

```

```

254 MPI_Bcast(&r1[1], atom->nbondtypes, MPI_DOUBLE, 0, world);
255
256 for (int i = 1; i <= atom->nbondtypes; i++) setflag[i] = 1;
257 }
258
259 /* -----
260 proc 0 writes to data file
261 ----- */
262
263 void BondHarmonicSwimmer::write_data(FILE *fp)
264 {
265     for (int i = 1; i <= atom->nbondtypes; i++) {
266         double d2 = (r0[i]-r1[i])*(r0[i]-r1[i]);
267         fprintf(fp,"%d %g %g %g\n",i,k[i]*d2,r0[i],r1[i]);
268     }
269 }
270
271 /* ----- */
272
273 double BondHarmonicSwimmer::single(int type, double rsq, int i, int j,
274                                     double &fforce)
275 {
276     double r = sqrt(rsq);
277     double dr = r - r0[type];
278     double dr2=r0[type]-r1[type];
279
280     fforce = -2.0*k[type]*dr/r;
281     return k[type]*(dr*dr - dr2*dr2);
282 }
```

#### A.4. *bond\_harmonic\_swimmer\_extended\_k* code file

```

1  /*
2   * -----
3   * LAMMPS - Large-scale Atomic/Molecular Massively Parallel Simulator
4   * http://lammps.sandia.gov, Sandia National Laboratories
5   * Steve Plimpton, sjplimp@sandia.gov
6   *
7   * Copyright (2003) Sandia Corporation. Under the terms of Contract
8   * DE-AC04-94AL85000 with Sandia Corporation, the U.S. Government retains
9   * certain rights in this software. This software is distributed under
10  * the GNU General Public License.
11
12  See the README file in the top-level LAMMPS directory.
13
14 */
15
16  Contributing author: Carsten Svaneborg, science@zqex.dk
17
18 #include "math.h"
19 #include "stdlib.h"
20 #include "bond_harmonic_swimmer_extended_k.h"
21 #include "atom.h"
22 #include "neighbor.h"
23 #include "domain.h"
24 #include "comm.h"
25 #include "force.h"
26 #include "memory.h"
27 #include "error.h"
28 #include "update.h"
29
30 using namespace LAMMPS_NS;
31
```

```

32 #define EPSILON 1.0e-20
33
34 /* -----
35
36 BondHarmonicSwimmerExtendedK::BondHarmonicSwimmerExtendedK(LAMMPS *lmp) : Bond(
37     lmp) {
38     time_origin = update->ntimestep;
39 }
40 */
41
42 BondHarmonicSwimmerExtendedK::~BondHarmonicSwimmerExtendedK()
43 {
44     if (allocated) {
45         memory->destroy(setflag);
46         memory->destroy(k_alpha);
47         memory->destroy(k_beta);
48         memory->destroy(r0);
49         memory->destroy(r1);
50
51         memory->destroy(A_alpha);
52         memory->destroy(A_beta);
53
54         memory->destroy(omega_alpha);
55         memory->destroy(omega_beta);
56
57         memory->destroy(phi);
58         memory->destroy(vel_sw);
59
60         memory->destroy(n1);
61         memory->destroy(n2);
62     }
63 }
64
65 /* helper function to calculate force and energy */
66 void BondHarmonicSwimmerExtendedK::uf_calculate(int type, int tag1, int tag2,
67         double r, double delta,
68         int eflag, double &u, double &f) {
69     double r0_local;
70     if ( (tag1>=n1[type]) && (tag1<=n2[type]) && ((tag2-tag1)==1) ) {
71         double dn = static_cast<double>(tag1) - n1[type];
72         double omega = omega_beta[type]*dn + omega_alpha[type];
73         double A    = A_beta[type]*dn + A_alpha[type];
74         double s_aux = sin(omega*dn + phi[type] - vel_sw[type]*delta);
75         r0_local = r0[type] + A*s_aux ;
76     } else {
77         r0_local = r0[type];
78     }
79
80     double dr = r - r0_local;
81     double dn = static_cast<double>(tag1) - n1[type];
82     double k = k_beta[type]*dn + k_alpha[type];
83
84     double rk = k*dr;
85
86     // force & energy
87     if (r > 0.0) f = -2.0*rk/r;
88     else f = 0.0;
89
90     if (eflag)
91         u = k*(dr*dr -(r0_local-r1[type])*(r0_local-r1[type]));
92 }
93

```

```

94 /* -----
95
96 void BondHarmonicSwimmerExtendedK::compute(int eflag, int vflag)
97 {
98     int i1,i2,n,type;
99     tagint tag1, tag2;
100    double delx,dely,delz,ebond,fbond;
101    double rsq,r;
102
103    ebond = 0.0;
104    if (eflag || vflag) ev_setup(eflag,vflag);
105    else evflag = 0;
106
107    double **x = atom->x;
108    double **f = atom->f;
109    tagint *tag = atom->tag;
110
111    int **bondlist = neighbor->bondlist;
112    int nbondlist = neighbor->nbondlist;
113    int nlocal = atom->nlocal;
114    int newton_bond = force->newton_bond;
115    double delta = (update->ntimestep - time_origin) * update->dt;
116
117    for (n = 0; n < nbondlist; n++) {
118        i1 = bondlist[n][0];
119        i2 = bondlist[n][1];
120
121        tag1 = tag[i1];
122        tag2 = tag[i2];
123
124        if (tag1>=tag2) {
125            char str[128];
126            sprintf(str,"tag1>=tag2: something wrong with a bond between %i and %i",
127                   i1, i2);
128            error->all(FLERR, str);
129        }
130
131        type = bondlist[n][2];
132
133        delx = x[i1][0] - x[i2][0];
134        dely = x[i1][1] - x[i2][1];
135        delz = x[i1][2] - x[i2][2];
136
137        rsq = delx*delx + dely*dely + delz*delz;
138        r = sqrt(rsq);
139        uf_calculate(type, tag1, tag2, r, delta, eflag, ebond, fbond);
140
141        // apply force to each of 2 atoms
142
143        if (newton_bond || i1 < nlocal) {
144            f[i1][0] += delx*fbond;
145            f[i1][1] += dely*fbond;
146            f[i1][2] += delz*fbond;
147        }
148
149        if (newton_bond || i2 < nlocal) {
150            f[i2][0] -= delx*fbond;
151            f[i2][1] -= dely*fbond;
152            f[i2][2] -= delz*fbond;
153        }
154
155        if (evflag) ev_tally(i1,i2,nlocal,newton_bond,ebond,fbond,delx,dely,delz);
156    }

```

```

156 }
157 /* -----
158 void BondHarmonicSwimmerExtendedK::allocate()
159 {
160     allocated = 1;
161     int n = atom->nbondtypes;
162
163     memory->create(k_alpha ,      n+1,"bond:k_alpha");
164     memory->create(k_beta ,      n+1,"bond:k_beta");
165
166     memory->create(r0 ,      n+1,"bond:r0");
167     memory->create(r1 ,      n+1,"bond:r1");
168
169     memory->create(A_alpha ,      n+1,"bond:A_alpha");
170     memory->create(A_beta ,      n+1,"bond:A_beta");
171
172     memory->create(omega_alpha ,      n+1,"bond:omega_alpha");
173     memory->create(omega_beta ,      n+1,"bond:omega_beta");
174
175     memory->create(phi ,      n+1,"bond:phi");
176     memory->create(vel_sw ,      n+1,"bond:vel_sw");
177
178     memory->create(n1 ,      n+1,"bond:n1");
179     memory->create(n2 ,      n+1,"bond:n2");
180
181     memory->create(setflag,n+1,"bond:setflag");
182
183     for (int i = 1; i <= n; i++) setflag[i] = 0;
184 }
185
186 /* -----
187     set coeffs for one or more types
188 ----- */
189
190 void BondHarmonicSwimmerExtendedK::coeff(int nargs, char **arg)
191 {
192     if (nargs != 13) error->all(FLERR,"Incorrect args for bond coefficients");
193     if (!allocated) allocate();
194
195     if (atom->tag_enable==0) {
196         error->all(FLERR,"Bond harmonic/swimmer/extended/k requires tag_enable=1");
197     }
198
199     int ilo,ihi;
200     force->bounds(arg[0],atom->nbondtypes,ilo,ihi);
201
202     double Umin = force->numeric(FLERR,arg[1]); // energy at minimum
203     double k_beta_one = force->numeric(FLERR,arg[2]);
204     double r0_one = force->numeric(FLERR,arg[3]); // position of minimum
205     double r1_one = force->numeric(FLERR,arg[4]); // position where energy = 0
206
207     // swimmer wave parameters A*sin(omega*N + phi - vel_sw*time)
208     double A_alpha_one = force->numeric(FLERR,arg[5]);
209     double A_beta_one = force->numeric(FLERR,arg[6]);
210
211     double omega_alpha_one = force->numeric(FLERR,arg[7]);
212     double omega_beta_one = force->numeric(FLERR,arg[8]);
213
214     double phi_one = force->numeric(FLERR,arg[9]);
215     double vel_sw_one = force->numeric(FLERR,arg[10]);
216
217
218

```

```

219 tagint n1_one = force->numeric(FLERR,arg[11]);
220 tagint n2_one = force->numeric(FLERR,arg[12]);
221
222 if (r0_one == r1_one)
223   error->all(FLERR,"Bond harmonic/swimmer/extended/k r0 and r1 must be
224 different");
225
226 int count = 0;
227 for (int i = ilo; i <= ihi; i++) {
228
229   k_alpha[i] = Umin/((r0_one-r1_one)*(r0_one-r1_one));
230   k_beta[i] = k_beta_one;
231   r0[i] = r0_one;
232   r1[i] = r1_one;
233
234   A_alpha[i] = A_alpha_one;
235   A_beta[i] = A_beta_one;
236
237   omega_alpha[i] = omega_alpha_one;
238   omega_beta[i] = omega_beta_one;
239
240   phi[i] = phi_one;
241   vel_sw[i] = vel_sw_one;
242
243   n1[i] = n1_one;
244   n2[i] = n2_one;
245
246   setflag[i] = 1;
247   count++;
248 }
249
250 if (count == 0) error->all(FLERR,"Incorrect args for bond coefficients");
251
252 /* -----
253    return an equilibrium bond length
254 ----- */
255
256 double BondHarmonicSwimmerExtendedK::equilibrium_distance(int i)
257 {
258   return r0[i];
259 }
260
261 /* -----
262    proc 0 writes out coeffs to restart file
263 ----- */
264
265 void BondHarmonicSwimmerExtendedK::write_restart(FILE *fp)
266 {
267   fwrite(&k_alpha[1],sizeof(double),atom->nbondtypes,fp);
268   fwrite(&r0[1],sizeof(double),atom->nbondtypes,fp);
269   fwrite(&r1[1],sizeof(double),atom->nbondtypes,fp);
270 }
271
272 /* -----
273    proc 0 reads coeffs from restart file, bcasts them
274 ----- */
275
276 void BondHarmonicSwimmerExtendedK::read_restart(FILE *fp)
277 {
278   allocate();
279
280   if (comm->me == 0) {

```

```

281     fread(&k_alpha[1], sizeof(double), atom->nbondtypes, fp);
282     fread(&r0[1], sizeof(double), atom->nbondtypes, fp);
283     fread(&r1[1], sizeof(double), atom->nbondtypes, fp);
284 }
285 MPI_Bcast(&k_alpha[1], atom->nbondtypes, MPI_DOUBLE, 0, world);
286 MPI_Bcast(&r0[1], atom->nbondtypes, MPI_DOUBLE, 0, world);
287 MPI_Bcast(&r1[1], atom->nbondtypes, MPI_DOUBLE, 0, world);
288
289 for (int i = 1; i <= atom->nbondtypes; i++) setflag[i] = 1;
290 }
291
292 /* -----
293 proc 0 writes to data file
294 ----- */
295
296 void BondHarmonicSwimmerExtendedK::write_data(FILE *fp)
297 {
298     for (int i = 1; i <= atom->nbondtypes; i++) {
299         double d2 = (r0[i]-r1[i])*(r0[i]-r1[i]);
300         fprintf(fp, "%d %g %g %g\n", i, k_alpha[i]*d2, r0[i], r1[i]);
301     }
302 }
303
304 /* -----
305 ----- */
306 double BondHarmonicSwimmerExtendedK::single(int type, double rsq, int i1, int i2
307 ,
308     double &fforce)
309 {
310     double ebond;
311     tagint *tag = atom->tag;
312     double delta = (update->ntimestep - time_origin) * update->dt;
313
314     double r = sqrt(rsq);
315     tagint tag1 = tag[i1];
316     tagint tag2 = tag[i2];
317
318     if (tag1>=tag2) {
319         char str[128];
320         sprintf(str,"tag1>=tag2: something wrong with a bond between %i and %i", i1,
321             i2);
322         error->all(FLERR, str);
323     }
324
325     int eflag = 1;
326     uf_calculate(type, tag1, tag2, r, delta, eflag, ebond, fforce);
327     return ebond;
328 }
```

## A.5. sph\_kernel\_quintic\_2d code file

```

1  /*
2   * LAMMPS - Large-scale Atomic/Molecular Massively Parallel Simulator
3   * http://lammps.sandia.gov, Sandia National Laboratories
4   * Steve Plimpton, sjplimp@sandia.gov
5
6   * Copyright (2003) Sandia Corporation. Under the terms of Contract
7   * DE-AC04-94AL85000 with Sandia Corporation, the U.S. Government retains
8   * certain rights in this software. This software is distributed under
9   * the GNU General Public License.
10
11 See the README file in the top-level LAMMPS directory.
12 ----- */

```

```

13
14 #include "sph_kernel_quintic_2d.h"
15 #include "math.h"
16 using namespace LAMMPS_NS;
17
18 double SPHKernelQuintic2D::w (double r, double h) {
19     double norm2d = 0.04195297663091802/(h*h);
20     double s = 3.0*r/h;
21     if (s<1.0) {
22         return norm2d*(pow(3 - s, 5) - 6*pow(2 - s, 5) + 15*pow(1 - s, 5));
23     } else if (s<2.0) {
24         return norm2d*(pow(3 - s, 5) - 6*pow(2 - s, 5));
25     } else if (s<3.0) {
26         return norm2d*pow(3 - s, 5);
27     }
28     return 0.0;
29 }
30
31 double SPHKernelQuintic2D::dw (double r, double h) {
32     double norm2d = 3.0*0.04195297663091802/(h*h*h);
33     double s = 3.0*r/h;
34     double wfd;
35     if (s<1) {
36         wfd = -50*pow(s,4)+120*pow(s,3)-120*s;
37     } else if (s<2) {
38         wfd = 25*pow(s,4)-180*pow(s,3)+450*pow(s,2)-420*s+75;
39     } else if (s<3) {
40         wfd = -5*pow(s,4)+60*pow(s,3)-270*pow(s,2)+540*s-405;
41     } else {
42         wfd = 0.0;
43     }
44     return norm2d*wfd;
45 }
46
47 double SPHKernelQuintic2D::dw_per_r (double r, double h) {
48     return dw(r, h)/r;
49 }
```

## A.6. pair\_sph\_adami code file

```

1  /*
2   * -----
3   * LAMMPS - Large-scale Atomic/Molecular Massively Parallel Simulator
4   * http://lammps.sandia.gov, Sandia National Laboratories
5   * Steve Plimpton, sjplimp@sandia.gov
6   *
7   * Copyright (2003) Sandia Corporation. Under the terms of Contract
8   * DE-AC04-94AL85000 with Sandia Corporation, the U.S. Government retains
9   * certain rights in this software. This software is distributed under
10  * the GNU General Public License.
11  *
12  * See the README file in the top-level LAMMPS directory.
13  * -----
14  */
15
16 #include "math.h"
17 #include "string.h"
18 #include "stdlib.h"
19 #include "pair_sph_adami.h"
20 #include "atom.h"
21 #include "force.h"
22 #include "comm.h"
23 #include "neigh_list.h"
24 #include "memory.h"
25 #include "error.h"
```

```

24 #include "domain.h"
25 #include "sph_kernel_dispatch.h"
26
27 using namespace LAMMPS_NS;
28
29 /* -----
30
31 PairSPHAdami::PairSPHAdami(LAMMPS *lmp) : Pair(lmp)
32 {
33     restartinfo = 0;
34     first = 1;
35 }
36
37 /* -----
38
39 PairSPHAdami::~PairSPHAdami() {
40     if (allocated) {
41         memory->destroy(setflag);
42         memory->destroy(cutsq);
43
44         memory->destroy(cut);
45         memory->destroy(rho0);
46         memory->destroy(soundspeed);
47         memory->destroy(B);
48         memory->destroy(viscosity);
49         memory->destroy(pb);
50
51         int n = atom->n_types;
52         for (int i=0; i<=n; ++i) {
53             for (int j=0; j<=n; ++j) delete ker[i][j];
54             delete[] ker[i];
55         }
56         delete[] ker;
57     }
58 }
59
60 /* -----
61
62 void PairSPHAdami::compute(int eflag, int vflag) {
63     int i, j, ii, jj, inum, jnum, itype, jtype;
64     double xtmp, ytmp, ztmp, delx, dely, delz;
65
66     int *ilist, *jlist, *numneigh, **firstneigh;
67     double vxtmp, vytmp, vztmp, imass, jmass, fvisc, velx, vely, velz;
68     double rsq, wfd, delVdotDelR, deltaE;
69
70     if (eflag || vflag)
71         ev_setup(eflag, vflag);
72     else
73         evflag = vflag_fdotr = 0;
74
75     double **v = atom->v;
76     double **x = atom->x;
77     double **f = atom->f;
78     double **fb = atom->fb;
79     double *rho = atom->rho;
80     double *mass = atom->mass;
81     double *de = atom->de;
82     double *drho = atom->drho;
83     int *type = atom->type;
84     int nlocal = atom->nlocal;
85     int newton_pair = force->newton_pair;
86     // check consistency of pair coefficients

```

```

87
88 if (first) {
89     for (i = 1; i <= atom->nTypes; i++) {
90         for (j = 1; j <= atom->nTypes; j++) {
91             if (cutsq[i][j] > 1.e-32) {
92                 if (!setflag[i][j] || !setflag[j][i]) {
93                     if (comm->me == 0) {
94                         printf(
95                             "SPH particle types %d and %d interact with cutoff=%g, but not
96                             all of their single particle properties are set.\n",
97                             i, j, sqrt(cutsq[i][j]));
98                     }
99                 }
100             }
101         }
102     }
103     first = 0;
104 }
105
106 inum = list->inum;
107 ilist = list->ilist;
108 numneigh = list->numneigh;
109 firstneigh = list->firstneigh;
110
111 // loop over neighbors of my atoms
112 for (ii = 0; ii < inum; ii++) {
113     i = ilist[ii];
114     xtmp = x[i][0];
115     ytmp = x[i][1];
116     ztmp = x[i][2];
117     vxtmp = v[i][0];
118     vytmp = v[i][1];
119     vztmp = v[i][2];
120     itype = type[i];
121     jlist = firstneigh[i];
122     jnum = numneigh[i];
123
124     imass = mass[itype];
125
126     // compute pressure of atom i
127     double pi = B[itype] * (rho[i] / rho0[itype] - 1.0);
128     double Vi = imass / rho[i];
129     double Vi2 = Vi * Vi;
130
131     for (jj = 0; jj < jnum; jj++) {
132         j = jlist[jj];
133         j &= NEIGHMASK;
134
135         delx = xtmp - x[j][0];
136         dely = ytmp - x[j][1];
137         delz = ztmp - x[j][2];
138         rsq = delx * delx + dely * dely + delz * delz;
139         jtype = type[j];
140         jmass = mass[jtype];
141
142         if (rsq < cutsq[itype][jtype]) {
143             wfd = ker[itype][jtype]->dw_per_r(sqrt(rsq), cut[itype][jtype]);
144             double Vj = jmass / rho[j];
145             double Vj2 = Vj * Vj;
146
147             // compute pressure
148             double pj = B[jtype] * (rho[j] / rho0[jtype] - 1.0);

```

```

149 double pij_wave = (rho[j]*pi + rho[i]*pj)/(rho[i] + rho[j]);
150 double pij_b    = pb[jtype];
151
152     velx=vxtmp - v[j][0];
153     vely=vytmp - v[j][1];
154     velz=vztmp - v[j][2];
155
156     // dot product of velocity delta and distance vector
157     delVdotDelR = delx * velx + dely * vely + delz * velz;
158
159     fvisc = (Vi2 + Vj2) * viscosity[itype][jtype] * wfd;
160
161     // total pair force & thermal energy increment
162     double fpair = - (Vi2 + Vj2) * pij_wave * wfd;
163     double fpair_b = - (Vi2 + Vj2) * pij_b * wfd;
164
165     deltaE = -0.5 * (fpair * delVdotDelR + fvisc * (velx*velx + vely*vely +
166     velz*velz));
167
167 // printf("testvar= %f, %f \n", delx, dely);
168
169     f[i][0] += delx * fpair + velx * fvisc;
170     f[i][1] += dely * fpair + vely * fvisc;
171     f[i][2] += delz * fpair + velz * fvisc;
172
173 // change in background pressure
174     fb[i][0] += delx * fpair_b;
175     fb[i][1] += dely * fpair_b;
176     fb[i][2] += delz * fpair_b;
177
178 // and change in density
179     drho[i] += jmass * delVdotDelR * wfd;
180
181 // change in thermal energy
182     de[i] += deltaE;
183
184     if (newton_pair || j < nlocal) {
185         f[j][0] -= delx * fpair + velx * fvisc;
186         f[j][1] -= dely * fpair + vely * fvisc;
187         f[j][2] -= delz * fpair + velz * fvisc;
188
189         fb[j][0] -= delx * fpair_b;
190         fb[j][1] -= dely * fpair_b;
191         fb[j][2] -= delz * fpair_b;
192
193         de[j] += deltaE;
194         drho[j] += imass * delVdotDelR * wfd;
195     }
196
197     if (evflag)
198         ev_tally(i, j, nlocal, newton_pair, 0.0, 0.0, fpair, delx, dely, delz);
199     ;
200 }
201 }
202
203     if (vflag_fdotr) virial_fdotr_compute();
204 }
205
206 /* -----
207 allocate all arrays
208 ----- */
209

```

```

210 void PairSPHAdami::allocate() {
211     allocated = 1;
212     int n = atom->ntypes;
213
214     memory->create(setflag, n + 1, n + 1, "pair:setflag");
215     for (int i = 1; i <= n; i++)
216         for (int j = i; j <= n; j++)
217             setflag[i][j] = 0;
218
219     memory->create(cutsq, n + 1, n + 1, "pair:cutsq");
220
221     memory->create(rho0, n + 1, "pair:rho0");
222     memory->create(soundspeed, n + 1, "pair:soundspeed");
223     memory->create(B, n + 1, "pair:B");
224     memory->create(cut, n + 1, n + 1, "pair:cut");
225     memory->create(viscosity, n + 1, n + 1, "pair:viscosity");
226     memory->create(pb, n + 1, "pair:pb");
227
228     ker = new pSPHKernel*[n+1];
229     for (int i=0; i<=n; ++i) {
230         ker[i] = new pSPHKernel[n+1];
231         for (int j=0; j<=n; ++j)
232             ker[i][j] = NULL;
233     }
234 }
235
236 /* -----
237  global settings
238 ----- */
239
240 void PairSPHAdami::settings(int narg, char **arg) {
241     if (narg != 0)
242         error->all(FLERR,
243                     "Illegal number of setting arguments for pair_style sph/adami");
244 }
245
246 /* -----
247  set coeffs for one or more type pairs
248 ----- */
249
250 void PairSPHAdami::coeff(int narg, char **arg) {
251     if (narg != 8)
252         error->all(FLERR,
253                     "Incorrect args for pair_style sph/adami coefficients");
254     if (!allocated)
255         allocate();
256
257     int ilo, ihi, jlo, jhi;
258     force->bounds(arg[0], atom->ntypes, ilo, ihi);
259     force->bounds(arg[1], atom->ntypes, jlo, jhi);
260
261     int i_kernel_name = 2;
262     char *kernel_name_one;
263     int n_kernel_name = strlen(arg[i_kernel_name]) + 1;
264     kernel_name_one = new char[n_kernel_name];
265     strcpy(kernel_name_one, arg[i_kernel_name]);
266
267     double rho0_one = force->numeric(FLERR,arg[3]);
268     double soundspeed_one = force->numeric(FLERR,arg[4]);
269     double viscosity_one = force->numeric(FLERR,arg[5]);
270     double pb_one = force->numeric(FLERR,arg[6]);
271     double cut_one = force->numeric(FLERR,arg[7]);
272     double B_one = soundspeed_one * soundspeed_one * rho0_one ;

```

```

273
274     int count = 0;
275     for (int i = ilo; i <= ihi; i++) {
276         rho0[i] = rho0_one;
277         soundspeed[i] = soundspeed_one;
278         B[i] = B_one;
279         pb[i] = pb_one;
280
281         for (int j = MAX(jlo,i); j <= jhi; j++) {
282             viscosity[i][j] = viscosity_one;
283             //printf("setting cut[%d][%d] = %f\n", i, j, cut_one);
284             cut[i][j] = cut_one;
285
286             ker[i][j] = sph_kernel_dispatch(kernel_name_one, domain->dimension, error)
287                 ;
288             if (i!=j) ker[j][i] = sph_kernel_dispatch(kernel_name_one, domain->
289                 dimension, error);
290
291             setflag[i][j] = 1;
292             count++;
293         }
294     }
295     delete[] kernel_name_one;
296
297     if (count == 0)
298         error->all(FLERR,"Incorrect args for pair coefficients");
299 }
300 /* -----
301 init for one type pair i,j and corresponding j,i
302 ----- */
303 double PairSPHAdami::init_one(int i, int j) {
304
305     if (setflag[i][j] == 0) {
306         error->all(FLERR,"Not all pair sph/adami coeffs are not set");
307     }
308
309     cut[j][i] = cut[i][j];
310     viscosity[j][i] = viscosity[i][j];
311
312     return cut[i][j];
313 }
314 /* ----- */
315 double PairSPHAdami::single(int i, int j, int itype, int jtype,
316     double rsq, double factor_coul, double factor_lj, double &fforce) {
317     fforce = 0.0;
318
319     return 0.0;
320 }

```

# List of Figures

2.1. Drafts of microscopic swimmers , to scale. (a) <i>E.coli..</i> (b) <i>C. crescentus.</i> (c) <i>R. sphaeroides</i> , with flagellar filament in the coiled state. (d) <i>Spiroplasma</i> , with a single kink separating regions of right-handed and left-handed coiling. (e) Human spermatozoon. (f) Mouse spermatozoon (g) <i>Chlamydomonas</i> . (h) A smallish <i>Paramecium</i> [LP09]. . . . .	4
2.2. Snake ( <i>Natrix</i> ) swimming in water ; 5 cm squares, 16 frames per second [Tay52]	5
2.3. Symbolic Swimmer Structure . . . . .	6
2.4. Bead-Spring Structure . . . . .	6
2.5. Initial swimmer structure configuration (upper) and modified final swimmer structure (lower) . . . . .	7
2.6. Swimmer deformed into a wave format with compressed head . . . . .	7
2.7. Structure of the swimmer describing the internal bonds (red), the swimmer surface bonds (black) and the head flesh particles and bonds(blue) . . . . .	8
3.1. (a) Simulation snapshot of the shear driven fluid filled cavity. Particles are colored according to their kinetic energy. (b) Set of particles located in the cavity centerline used to calculate the velocity profile. . . . .	9
3.2. (a) Velocity profile along centerline of the cavity with SPH and FDM (Finite Difference Method) solutions from [NS68] , (b) Simulation results for velocity profile along centerline . . . . .	10
3.3. Swimmer structure bending with bonds compression and tension . . . . .	11
3.4. Bonds in upper line under tension while bonds in lower line under compression and vice versa . . . . .	12
3.5. . . . .	13
3.6. Body shape and tail length of larvae of <i>Herdmania pallida</i> and <i>Aplidium constellatum</i> [McH05] . . . . .	14
3.7. Schematic diagram of a swimming <i>C. intestinalis</i> larva with its sensory and motor organs highlighted . . . . .	15
3.8. For a given muscle activation pattern, there are different optimal stiffness values for maximum acceleration or steady swimming speed.The plots show four swimmers with increasing stiffness: tan dotted line, simulation 4, Table 1; green long dashes, simulation 5; black, reference simulation; and cyan short dashes, simulation 6. (A) Swimming speed vs. time. (B)Body outlines for each swimmer at the time indicated by the arrow on panel. (C) Mean acceleration during the first tail beat. (D) Mean steady swimming speed. (E) Muscle power coefficient . . . . .	16
3.9. Sketch of the smoothing kernel length $h_i$ and its influence domainSketch of the smoothing kernel length $h_i$ and its influence domain . . . . .	17
3.10. Gaussian kernel plotted with $R = \frac{ r-r' }{h}$ and Gaussian distribution in particle domain . . . . .	18
3.11. B-spline function plotted compared to Gaussian distribution . . . . .	19
3.12. Quintic function plotted compared to Gaussian distribution . . . . .	19
3.13. On the left side, the velocity profiles $V_x(y)$ at $x = L/2$ (Path 1) and $x = L$ (Path 2) from Adami [AHA13] and on the right side, the same velocity profiles for LAMMPS simulation . . . . .	22

3.14. Scaled contour plots of velocity magnitude from Adami results and velocity magnitude field from LAMMPS . . . . .	23
4.1. Configuration files . . . . .	24
4.2. In upper image the particles are allocated in the domain via square lattice distribution and in the lower image the particles rearranged their position after heating up with the Langevin thermostat . . . . .	26
4.3. . . . .	27
4.4. . . . .	28
4.5. . . . .	28
4.6. Swimmer velocity in x-direction . . . . .	29
4.7. Force in x-direction in th swimmer center of mass . . . . .	29
4.8. Swimmer center of mass path . . . . .	30
4.9. Swimmer velocity in x-direction for amplitudes $A$ , $2A$ and $A/2$ . . . . .	30
4.10. Force in x-direction in th swimmer center of mass for amplitudes $A$ , $2A$ and $A/2$ . . . . .	31
4.11. Swimmer center of mass path for amplitudes $A$ , $2A$ and $A/2$ . . . . .	31

# Bibliography

- [AHA13] S. Adami, X. Y. Hu, and N. A. Adams. A transport-velocity formulation for smoothed particle hydrodynamics. *Journal of Computational Physics*, 241:292–307, May 2013.
- [BA73] Howard C. Berg and Robert A. Anderson. Bacteria swim by rotating their flagellar filaments. *Nature*, 245(5425):380–382, October 1973.
- [BL73] J J Blum and J Lubliner. Biophysics of flagellar motility. *Annual Review of Biophysics and Bioengineering*, 2(1):181–219, 1973.
- [BS10] I. Borazjani and F. Sotiropoulos. On the role of form and kinematics on the hydrodynamics of self-propelled body/caudal fin swimming. *The Journal of Experimental Biology*, 213(1):89–107, January 2010.
- [Chi81] Stephen Childress. *Mechanics of Swimming and Flying*. Cambridge University Press, Cambridge, 1981.
- [Dig] Biology Digest. Nature’s flyers.
- [EKG10] Jens Elgeti, U. Benjamin Kaupp, and Gerhard Gompper. Hydrodynamics of sperm cells near surfaces. *Biophysical Journal*, 99(4):1018–1026, August 2010.
- [Gil98] G. B. Gillis. Neuromuscular control of anguilliform locomotion: patterns of red and white muscle activity during swimming in the american eel anguilla rostrata. *The Journal of Experimental Biology*, 201(23):3245–3256, December 1998.
- [GM77] 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, November 1977.
- [GM78] R. A. Gingold and J. J. Monaghan. Binary fission in damped rotating polytropes. 1978.
- [GM79] R. A. Gingold and J. J. Monaghan. A numerical study of the roche and darwin problems for polytropic stars. *Monthly Notices of the Royal Astronomical Society*, 188(1):45–58, September 1979.
- [GM82] R. A. Gingold and J. J. Monaghan. Kernel estimates as a basis for general particle methods in hydrodynamics. *Journal of Computational Physics*, 46(3):429–453, 1982.
- [GSVLL11] Georg C. Ganzenmüller, Martin O. Steinhauser, Paul Van Liedekerke, and Katholieke Universiteit Leuven. The implementation of smooth particle hydrodynamics in LAMMPS. 2011.
- [Jay85] Bruce C. Jayne. Swimming in constricting (elaphe g. guttata) and nonconstricting (nerodia fasciata pictiventris) colubrid snakes. *Copeia*, 1985(1):195, February 1985.
- [Jay88] Bruce C. Jayne. Muscular mechanisms of snake locomotion: An electromyographic study of lateral undulation of the florida banded water snake (nerodia fasciata) and the yellow rat snake (elaphe obsoleta). *Journal of Morphology*, 197(2):159–181, August 1988.
- [JBL95] D. Joly, C. Bressac, and D. Lachaise. Disentangling giant sperm. *Nature*, 377(6546):202–202, September 1995.
- [KK06] Stefan Kern and Petros Koumoutsakos. Simulations of optimized anguilliform swimming. *The Journal of Experimental Biology*, 209(Pt 24):4841–4857, December 2006.

- [lam] Lammps users manual.
- [Lon98] John H. Long. Muscles, elastic energy, and the dynamics of body stiffness in swimming eels. *American Zoologist*, 38(4):771–792, September 1998.
- [LP09] Eric Lauga and Thomas R Powers. The hydrodynamics of swimming microorganisms. *Reports on Progress in Physics*, 72(9):096601, September 2009.
- [Luc77] L. B. Lucy. A numerical approach to the testing of the fission hypothesis. *The Astronomical Journal*, 82:1013, December 1977.
- [McH05] Matthew J McHenry. The morphology, behavior, and biomechanics of swimming in ascidian larvae. *Canadian Journal of Zoology*, 83(1):62–74, January 2005.
- [ML85] J. J. Monaghan and J. C. Lattanzio. A refined particle method for astrophysical problems. *Astronomy and Astrophysics*, 149:135–143, August 1985.
- [MM97] J. P. Morris and J. J. Monaghan. A switch to reduce SPH viscosity. *Journal of Computational Physics*, 136(1):41–50, September 1997.
- [Mon05] J J Monaghan. Smoothed particle hydrodynamics. *Reports on Progress in Physics*, 68(8):1703–1759, August 2005.
- [Mon12] J.J. Monaghan. Smoothed particle hydrodynamics and its diverse applications. *Annual Review of Fluid Mechanics*, 44(1):323–346, January 2012.
- [NS68] George A. Neece and David R. Squire. Tait and related empirical equations of state. *The Journal of Physical Chemistry*, 72(1):128–136, January 1968.
- [RAG80] J. J. Monaghan R. A. Gingold. The roche problem for polytropes in central orbits. *Monthly Notices of the Royal Astronomical Society*, 191:897–924, 1980.
- [SL06] R Shadwick and G. Lauder. Fish biomechanics. *Books*, January 2006.
- [SP06] S. S. Suarez and A. A. Pacey. Sperm transport in the female reproductive tract. *Human Reproduction Update*, 12(1):23–37, February 2006.
- [SS78] T. Schneider and E. Stoll. Molecular-dynamics study of a three-dimensional one-component model for distortive phase transitions. *Physical Review B*, 17(3):1302–1322, February 1978.
- [SYL01] Hitoshi Sawada, Hideyoshi Yokosawa, and Charles C. Lambert. *The Biology of Ascidiants*. Springer, January 2001.
- [Tay52] Geoffrey Taylor. Analysis of the swimming of long and narrow animals. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences*, 214(1117):158–183, August 1952.
- [THW<sup>+</sup>10] Eric D. Tytell, Chia-Yu Hsu, Thelma L. Williams, Avis H. Cohen, and Lisa J. Fauci. Interactions between internal forces, body stiffness, and fluid environment in a neuromechanical model of lamprey swimming. *Proceedings of the National Academy of Sciences*, 107(46):19832–19837, 2010.
- [Vog96] Steven Vogel. *Life in Moving Fluids: The Physical Biology of Flow*. Princeton University Press, Princeton, N.J., 2nd revised edition edition, April 1996.
- [Wil10] Thelma L. Williams. A new model for force generation by skeletal muscle, incorporating work-dependent deactivation. *The Journal of Experimental Biology*, 213(4):643–650, February 2010.

# Declaration

Surculus, Epulae pie Anxio conciliator era se concilium. Terra quam dicto erro prolecto, quo per incommoditas paulatim Praecepio lex Edoceo sis conticinium Furtum Heidelberg casula Toto pes an jugiter perpes Reficio congratulor simplex Ile familia mire hae Prosequor in pro St quae Muto,, St Texo aer Cornu ferox lex inconsiderate propitius, animus ops nos haero vetus Subdo qui Gemo ipse somnicul.

München, xx. September 20xx

Name des Autors