# Documentation

# Table of contents

# Introduction

Smoothed Particle Hydrodynamics is a Lagrangian meshless method that has been used in an expanding range of applications within the field of Computation Fluid Dynamics (CFD) [Gómez-Gesteira et al., 2010] where particles represent the flow, interact with structures, and exhibit large deformation with moving boundaries. The SPH model is approaching a mature stage for CFD with continuing improvements and modifications such that the accuracy, stability and reliability of the model are reaching an acceptable level for practical engineering applications.

The DualSPHysics code originates from SPHysics, which is an open-source SPH model developed by researchers at the Johns Hopkins University (US), the University of Vigo (Spain), the University of Manchester (UK) and the University of Rome, La Sapienza. The software is available to free download at www.sphysics.org. A complete guide of the FORTRAN code is found in [Gómez-Gesteira et al., 2012a; 2012b].

The SPHysics FORTRAN code was validated for different problems of wave breaking [Dalrymple and Rogers, 2006], dam-break behaviour [Crespo et al., 2008], interaction with coastal structures [Gómez-Gesteira and Dalrymple, 2004] or with a moving breakwater [Rogers et al., 2010].

Although SPHysics allows problems to be simulated using high resolution and a wide range of formulations, the main problem for its application to real engineering problems is the excessively long computational runtimes, meaning that SPHysics is rarely applied to large domains. Hardware acceleration and parallel computing are required to make SPHysics more useful and versatile for engineering application.

Originating from the computer games industry, Graphics Processing Units (GPUs) have now established themselves as a cheap alternative to High Performance Computing (HPC) for scientific computing and numerical modelling. GPUs are designed to manage huge amounts of data and their computing power has developed in recent years much faster than conventional central processing units (CPUs). Compute Unified Device Architecture (CUDA) is a parallel programming framework and language for GPU computing using some extensions to the C/C++ language. Researchers and engineers of different fields are achieving high speedups implementing their codes with the CUDA language. Thus, the parallel power computing of GPUs can be also applied for SPH methods where the same loops for each particle during the simulation can be parallelised.

The FORTRAN SPHysics code is robust and reliable but is not properly optimised for huge simulations. DualSPHysics is implemented in C++ and CUDA language to carry out simulations on either the CPU or GPU respectively. The new CPU code presents some advantages, such as more optimised use of the memory. The object-oriented programming paradigm provides a code that is easy to understand, maintain and modify with a sophisticated control of errors available. Furthermore, better optimisations are implemented, for example particles are reordered to give faster access to memory, and the best approach to create the neighbour list is implemented [Domínguez et al., 2011]. The CUDA language manages the parallel execution of threads on the GPUs. The best approaches were considered to be implemented as an extension of the C++ code, so the most appropriate optimizations to parallelise particle interaction on GPU were implemented [Domínguez et al., 2013a; 2013b]. The first rigorous validations were presented in [Crespo et al., 2011]. The version 3.0 of the code is fully documented in [Crespo et al., 2015].

Version 4 of the code has been developed to include the latest developments including coupling with the Discrete Element Method (DEM) and multi-phase developments as detailed in Section 3.

In the following sections we will describe the SPH formulation available in DualSPHysics, the implementation and

optimization techniques, how to compile and run the different codes of the DualSPHysics package and future developments.

# Developers and institutions

Different countries and institutions collaborate in the development of DualSPHysics. The project is mainly led by the Environmental Physics Laboratory (EPHYSLAB) from Universidade de Vigo (Spain) and the School of Mechanical, Aerospace and Civil Engineering (MACE) from The University of Manchester (UK).

The following list includes the researchers that have collaborated in the current version of the code or are working in functionalities to be updated in future releases.

## Developers:

### Universidade de Vigo, Spain

- Dr José M. Domínguez
- Dr Alejandro J.C. Crespo
- Dr Anxo Barreiro
- Professor Moncho Gómez Gesteira

### EPHYTECH SL, Spain

- Orlando G. Feal

### The University of Manchester, UK

- Dr Benedict D. Rogers
- Dr Georgios Fourtakas
- Dr Athanasios Mokos

### Science & Technology Facilities Council, UK

- Dr Stephen Longshaw

### Instituto Superior Tecnico, Lisbon, Portugal

- Dr Ricardo Canelas

### Università degli studi di Parma, Italy

- Dr Renato Vacondio

### Universiteit Gent - Flanders Hydraulics Research, Belgium

- Dr Corrado Altomare

### Contributors:

- Imperial College London, UK. Mashy D Green

- Universidad Politécnica de Madrid, Spain. Jose Luis Cercós Pita.
- Universidad de Guanajuato, Mexico. Carlos Enrique Alvarado Rodríguez.

# SPH formulation

## SPH formulation

Smoothed Particle Hydrodynamics (SPH) is a Lagrangian meshless method. The technique discretises a continuum using a set of material points or particles. When used for the simulation of fluid dynamics, the discretised Navier-Stokes equations are locally integrated at the location of each of these particles, according to the physical properties of surrounding particles. The set of neighbouring particles is determined by a distance based function, either circular (two-dimensional) or spherical (three-dimensional), with an associated characteristic length or smoothing length often denoted as h. At each timestep new physical quantities are calculated for each particle, and they then move according to the updated values.

The conservation laws of continuum fluid dynamics are transformed from their partial differential form to a form suitable for particle based simulation using integral equations based on an interpolation function, which gives an estimate of values at a specific point.

Typically this interpolation or weighting function is referred to as the kernel function (W) and can take different forms, with the most common being cubic or quintic. In all cases however, it is designed to represent a function F(r) defined in r' by the integral approximation

$$F(\mathbf{r}) = \int F(\mathbf{r}')W(\mathbf{r}-\mathbf{r}',h)d\mathbf{r}' \quad (1)$$

The smoothing kernel must fulfil several properties [Monaghan, 1992; Liu, 2003], such as positivity inside a defined zone of interaction, compact support, normalization and monotonically decreasing value with distance and differentiability. For a more complete description of SPH, the reader is referred to [Monaghan, 2005; Violeau, 2013].

The function F in Eq. (1) can be approximated in a non-continuous, discrete form based on the set of particles. In this case the function is interpolated at a particle (a) where a summation is performed over all the particles that fall within its region of compact support, as defined by the smoothing length h

$$F(\mathbf{r}_a) \approx \sum_b F(\mathbf{r}_b)W(\mathbf{r}_a-\mathbf{r}_b,h)\Delta v_b \quad (2)$$

where the subscript denotes an individual particle, $\Delta v_b$ is the volume of a neighbouring particle (b). If $\Delta v_b = m_b/\rho_b$, with m and ρ being the mass and the density of particle b respectively then Eq. (2) becomes

$$F(\mathbf{r}_a) \approx \sum_b F(\mathbf{r}_b)\frac{m_b}{\rho_b}W(\mathbf{r}_a-\mathbf{r}_b,h) \quad (3)$$

## The Smoothing Kernel

The performance of an SPH model depends heavily on the choice of the smoothing kernel. Kernels are expressed as a function of the non-dimensional distance between particles (q), given by $q = r/h$, where r is the distance between any two given particles a and b and the parameter h (the smoothing length) controls the size of the area around particle a in which neighbouring particles are considered. Within DualSPHysics, the user is able to choose from one of the following kernel definitions:

1. Cubic Spline

$$W(r,h)=\alpha_D\begin{cases}1-\frac{3}{2}q^2+\frac{3}{4}q^3 & 0\le q\le 1\\ \frac{1}{4}(2-q)^3 & 1\le q\le 2\\ 0 & q\ge 2\end{cases} \quad (4)$$

where αD is equal to 10/7πh2 in 2-D and 1/πh3 in 3-D.

The tensile correction method, proposed by [Monaghan, 2000], is only actively used in the cases of a kernel whose first derivative goes to zero with the particle distance q.

1. Quintic

$$W(r,h)=\alpha_D\left(1-\frac{q}{2}\right)^4(2q+1) \qquad 0\le q\le 2 \quad (5)$$

   where αD is equal to 7/4πh2 in 2-D and 21/16πh3 in 3-D.

In the text that follows, only kernels with an influence domain of 2h (q≤2) are considered

## Momentum Equation

The momentum conservation equation in a continuum is

$$\frac{d\boldsymbol{v}}{dt}=-\frac{1}{\rho}\nabla P+\boldsymbol{g}+\boldsymbol{\Gamma} \quad (6)$$

where Γ refers to dissipative terms and g is gravitational acceleration. DualSPHysics offers different options for including the effects of dissipation.

## Artificial Viscosity

The artificial viscosity scheme, proposed by [Monaghan, 1992], is a common method within fluid simulation using SPH due primarily to its simplicity. In SPH notation, Eq. 6 can be written as

$$\frac{d\boldsymbol{v}_a}{dt}=-\sum_b m_b\left(\frac{P_b+P_a}{\rho_b\cdot\rho_a}+\Pi_{ab}\right)\nabla_a W_{ab}+\boldsymbol{g} \quad (7)$$

where Pk and ρk are the pressure and density that correspond to particle k (as evaluated at a or b). The viscosity term $\Pi_{ab}$ is given by

$$\Pi_{ab} = \begin{cases} \dfrac{-\alpha \overline{c_{ab}} \mu_{ab}}{\overline{\rho_{ab}}} & \boldsymbol{v}_{ab} \cdot \boldsymbol{r}_{ab} < 0 \\ 0 & \boldsymbol{v}_{ab} \cdot \boldsymbol{r}_{ab} > 0 \end{cases} \quad (8)$$

where ab $\boldsymbol{r}_{ab} = \boldsymbol{r}_a - \boldsymbol{r}_b$ and $\boldsymbol{r}_{ab} = \boldsymbol{r}_a - \boldsymbol{r}_b$ with rk and vk being the particle position and velocity respectively. $\mu_{ab} = h\boldsymbol{v}_{ab} \cdot \boldsymbol{r}_{ab} / (r_{ab}^2 + \eta^2)$, $\overline{c_{ab}} = 0.5(c_a + c_b)$ is the mean speed of sound, $\eta^2 = 0.01h^2$ and α is a coefficient that needs to be tuned in order to introduce the proper dissipation. The value of α=0.01 has proven to give the best results in the validation of wave flumes to study wave propagation and wave loadings exerted onto coastal structures [Altomare et al., 2015a; 2015c].

## Laminar viscosity and Sub-Particle Scale (SPS) Turbulence

Laminar viscous stresses in the momentum equation can be expressed as [Lo and Shao, 2002]

$$\left(\upsilon_0 \nabla^2 \boldsymbol{v}\right)_a = \sum_b m_b \left( \frac{4\upsilon_0 \, \boldsymbol{r}_{ab} \cdot \nabla_a W_{ab}}{(\rho_a + \rho_b)(r_{ab}^2 + \eta^2)} \right) \boldsymbol{v}_{ab} \quad (9)$$

where uo is kinematic viscosity (typically 10-6 m2s for water). In SPH discrete notation this can be expressed as

$$\frac{d\boldsymbol{v}_a}{dt} = -\sum_b m_b \left( \frac{P_b + P_a}{\rho_b \cdot \rho_a} \right) \nabla_a W_{ab} + \boldsymbol{g} + \sum_b m_b \left( \frac{4\upsilon_0 \boldsymbol{r}_{ab} \cdot \nabla_a W_{ab}}{(\rho_a + \rho_b)(r_{ab}^2 + \eta^2)} \right) \boldsymbol{v}_{ab} \quad (10)$$

The concept of the Sub-Particle Scale (SPS) was first described by [Gotoh et al., 2001] to represent the effects of turbulence in their Moving Particle Semi-implicit (MPS) model. The momentum conservation equation is defined as

$$\frac{d\boldsymbol{v}}{dt} = -\frac{1}{\rho}\nabla P + \boldsymbol{g} + \upsilon_0 \nabla^2 \boldsymbol{v} + \frac{1}{\rho} \nabla \cdot \bar{\tau} \quad (11)$$

where the laminar term is treated as per Eq. 9 and $\bar{\tau}$ represents the SPS stress tensor. Favre-averaging is needed to account for compressibility in weakly compressible SPH [Dalrymple and Rogers, 2006] where eddy viscosity assumption is used to model the SPS stress tensor with Einstein notation for the shear stress component in coordinate directions i and $j$ $\dfrac{\bar{\tau}_{ij}}{\rho} = \nu_t \left( 2S_{ij} - \dfrac{2}{3} k \delta_{ij} \right) - \dfrac{2}{3} C_I \Delta^2 \delta_{ij} \left| S_{ij} \right|^2$ where $\bar{\tau}_{ij}$ is the sub-particle stress tensor, $\nu_t = \left[ (C_S \, \Delta l) \right]^2 |S|$ the turbulent eddy viscosity, k the SPS turbulence kinetic energy, Cs the Smagorinsky constant (0.12), CI=0.0066, $\Delta l$ the particle to particle spacing and $|S| = 0.5(2S_{ij}S_{ij})$ where $S_{ij}$ is an element of the SPS strain tensor. [Dalrymple and Rogers, 2006] introduced SPS into weakly compressible SPH using Favre averaging, Eq.11 can be re-written as

$$\frac{d\boldsymbol{v}_a}{dt} = -\sum_b m_b \left( \frac{P_b + P_a}{\rho_b \cdot \rho_a} \right) \nabla_a W_{ab} + \boldsymbol{g}$$
$$+ \sum_b m_b \left( \frac{4 v_0 r_{ab} \cdot \nabla_a W_{ab}}{(\rho_a + \rho_b)(r_{ab}^2 + \eta^2)} \right) \boldsymbol{v}_{ab} + \quad (12)$$
$$+ \sum_b m_b \left( \frac{\bar{\tau}_{ij}^b}{\rho_b^2} + \frac{\bar{\tau}_{ij}^a}{\rho_a^2} \right) \nabla_a W_{ab}$$

## Continuity Equation

Throughout the duration of a weakly-compressible SPH simulation (as presented herein) the mass of each particle remains constant and only their associated density fluctuates. These density changes are computed by solving the conservation of mass, or continuity equation, in SPH form:

$$\frac{d\rho_a}{dt} = \sum_b m_b \mathbf{v}_{ab} \cdot \nabla_a W_{ab} \quad (13)$$

## Equation of State

Following the work of [Monaghan, 1994], the fluid in the SPH formalism defined in DualSPHysics is treated as weakly compressible and an equation of state is used to determine fluid pressure based on particle density. The compressibility is adjusted so that the speed of sound can be artificially lowered; this means that the size of time step taken at any one moment (which is determined according to a Courant condition, based on the currently calculated speed of sound for all particles) can be maintained at a reasonable value. Such adjustment however, restricts the sound speed to be at least ten times faster than the maximum fluid velocity, keeping density variations to within less than 1%, and therefore not introducing major deviations from an incompressible approach. Following [Monaghan et al., 1999] and [Batchelor, 1974], the relationship between pressure and density follows the expression

$$P = b\left[ \left( \frac{\rho}{\rho_0} \right)^{\gamma} - 1 \right] \quad (14)$$

where $\gamma = 7, \quad b = c_0^2 \rho_0 / \gamma$ where $\rho_0 = 1000\,\mathrm{kg/m^3}$ is the reference density and $c_o = c(\rho_o) = \sqrt{(\partial P / \partial \rho)\big|_{\rho_o}}$ which is the speed of sound at the reference density.

## DeltaSPH

Within DualSPHysics it is also possible to apply a delta-SPH formulation, that introduces a diffusive term to reduce density fluctuations. The state equation describes a very stiff density field, and together with the natural disordering of the Lagrangian particles, high-frequency low amplitude oscillations are found to populate the density scalar field [Molteni and Colagrossi, 2009]. DualSPHysics uses a diffusive term in the continuity equation,

now written as

$$\frac{d\rho_a}{dt} = \sum_b m_b \mathbf{v}_{ab} \cdot \nabla_a W_{ab} + 2\delta_\Phi hc_0 \sum_b (\rho_b - \rho_a) \frac{\mathbf{r}_{ab} \cdot \nabla_a W_{ab}}{\mathbf{r}_{ab}^2} \frac{m_b}{\rho_b} \quad (15)$$

This represents the original delta-SPH formulation by [Molteni and Colagrossi, 2009], with the free parameter δΦ that needs to be attributed a suitable value. This modification can be explained as the addition of the Laplacian of the density field to the continuity equation. [Antuono et al., 2012] has presented a careful analysis of the influence of this term in the system, by decomposing the Laplacian operator, observing the converge of the operators and performing linear stability analysis to inspect the influence of the diffusive coefficient. This equation represents exactly a diffusive term in the domain bulk. The behaviour changes close to open boundaries such as free-surface. Due to truncation of the kernel (there are no particles being sampled outside of an open boundary), the first-order contributions are not null [Antuono et al., 2010], resulting in a net force applied to the particles. This effect is not considered relevant for nonhydrostatic situations, where this force is many orders of magnitude inferior to any other force involved. Corrections to this effect were proposed by [Antuono et al., 2010], but involve the solution of a renormalization problem for the density gradient, with considerable computational cost. A delta-SPH (δΦ) coefficient of 0.1 is recommended for most applications.

## Shifting algorithm

Anisotropic particle spacing is an important stability issue in SPH as, especially in violent flows, particles cannot maintain a uniform distribution. The result is the introduction of noise in the velocity and pressure field, as well as the creation of voids within the water flow for certain cases.

To counter the anisotropic particle spacing, [Xu et al., 2009] proposed a particle shifting algorithm to prevent the instabilities. The algorithm was first created for incompressible SPH, but can be extended to the weakly compressible SPH model used in DualSPHysics [Vacondio et al., 2013]. With the shifting algorithm, the particles are moved (“shifted”) towards areas with fewer particles (lower particle concentration) allowing the domain to maintain a uniform particle distribution and eliminating any voids that may occur due to the noise.

An improvement on the initial shifting algorithm was proposed by [Lind et al., 2012] who used Fick’s first law of diffusion to control the shifting magnitude and direction. Fick’s first law connects the diffusion flux to the concentration gradient:

$$\boldsymbol{J} = -D_F \nabla C \quad (16)$$

where J is the flux, C the particle concentration, and DF the Fickian diffusion coefficient.

Assuming that the flux, i.e. the number of particles passing through a unit surface in unit time, is proportional to the velocity of the particles, a particle shifting velocity and subsequently a particle shifting distance can be found. Using the particle concentration, the particle shifting distance δrs is given by:

$$\delta \boldsymbol{r}_s = -D \nabla C_i \quad (17)$$

where D is a new diffusion coefficient that controls the shifting magnitude and absorbs the constants of proportionality. The gradient of the particle concentration can be found through an SPH gradient operator:

$$\nabla C_i = \sum_j \frac{m_j}{\rho_j} \nabla W_{ij} \quad (18)$$

The proportionality coefficient D is computed through a form proposed by [Skillen et al., 2013]. It is set to be large enough to provide effective particle shifting, while not introducing significant errors or instabilities. This is achieved by performing a Von Neumann stability analysis of the advection-diffusion equation:

$$D \leq \frac{1}{2} \frac{h^2}{\Delta t_{\max}} \quad (19)$$

where Δtmax is the maximum local time step that is permitted by the CFL condition for a given local velocity and particle spacing. The CFL condition states that:

$$\Delta t_{\max} \leq \frac{\mathrm{h}}{\left\| \boldsymbol{u} \right\|_i} \quad (20)$$

Combining Eq. 19 and 20 we can derive an equation to find the shifting coefficient D:

$$D = Ah \left\| \mathbf{u} \right\|_i dt \quad (21)$$

where A is a dimensionless constant that is independent of the problem setup and discretization and dt is the current time step. Values in the range of [1,6] are proposed with 2 used as default.

The shifting algorithm is heavily dependent on a full kernel support. However, particles at and adjacent to the free surface cannot obtain the full kernel support, which will introduce errors in the free-surface prediction, potentially causing non-physical instabilities. Applying Fick's law directly would result in the rapid diffusion of fluid particles from the fluid bulk, due to the large concentration gradients at the free surface.

To counter this effect, [Lind et al., 2012] proposed a free-surface correction that limits diffusion to the surface normal but allow shifting on the tangent to the free surface. Therefore, this correction is only used near the free surface, identified by the value of the particle divergence, which is computed through the following equation, first proposed by [Lee et al., 2008]:

$$\nabla \cdot \mathbf{r} = \sum_j \frac{m_j}{\rho_j} \mathbf{r}_{ij} \cdot \nabla_i W_{ij} \quad (22)$$

This idea is applied to the DualSPHysics code by multiplying the shifting distance of Equation (17) with a free-surface correction coefficient AFSC.

$$A_{FSC} = \frac{\nabla \cdot \boldsymbol{r} - A_{FST}}{A_{FSM} - A_{FST}} \quad (23)$$

where AFST is the free-surface threshold and AFSM is the maximum value of the particle divergence. The latter depends on the domain dimensions:

$$A_{FSM} = \begin{cases} 2 \text{ for 2D} \\ 3 \text{ for 3D} \end{cases}$$

while the free surface threshold is selected for DualSPHysics as:

$$A_{FST} = \begin{cases} 1.5 \text{ for 2D} \\ 2.75 \text{ for 3D} \end{cases}$$

To identify the position of the particle relative to the free surface, the difference of the particle divergence to AFST is used. Therefore, the full shifting equation (Eq. 17) with the free surface correction is:

$$\delta \mathbf{r}_s = \begin{cases} -A_{FSC} Ah \|\mathbf{u}\|_i dt \cdot \nabla C_i & \text{if } (\nabla \cdot r - A_{FST}) < 0 \\ -Ah \|\mathbf{u}\|_i dt \cdot \nabla C_i & \text{if } (\nabla \cdot r - A_{FST}) = 0 \end{cases} \quad (24)$$

More information about the shifting implementation can be found in [Mokos, 2013].

## Time stepping

DualSPHysics includes a choice of numerical integration schemes, if the momentum ( a v ), density ( aρ ) and position ( ar ) equations are first written in the form

$$\frac{d\boldsymbol{v}_a}{dt} = \boldsymbol{F}_a \quad (25a)$$

$$\frac{d\rho_a}{dt} = D_a \quad (25b)$$

$$\frac{d\boldsymbol{r}_a}{dt} = \boldsymbol{v}_a \quad (25c)$$

These equations are integrated in time using a computationally simple Verlet based scheme or a more numerically stable but computationally intensive two-stage Symplectic method.

## Verlet Scheme

This algorithm, which is based on the common Verlet method [Verlet, 1967] is split into two parts and benefits from providing a low computational overhead compared to some other integration techniques, primarily as it does not require multiple (i.e. predictor and corrector) calculations for each step. The predictor step calculates the variables according to

$$\boldsymbol{v}_a^{n+1} = \boldsymbol{v}_a^{n-1} + 2\Delta t \boldsymbol{F}_a^n \; ; \; \boldsymbol{r}_a^{n+1} = \boldsymbol{r}_a^n + \Delta t \boldsymbol{V}_a^n + 0.5\Delta t^2 \boldsymbol{F}_a^n \; ; \; \rho_a^{n+1} = \rho_a^{n-1} + 2\Delta t D_a^n \quad (26)$$

where the superscript n denotes the time step, $\boldsymbol{F}_a^n$ and $D_a^n$ are calculated using Eq. 7 (or. 12) and Eq. 13 (or Eq. 14) respectively. However, once every Ns time steps (where $\mathrm{N_s} \approx 50$ is suggested), variables are calculated according to

$$\boldsymbol{v}_a^{n+1} = \boldsymbol{v}_a^n + \Delta t \boldsymbol{F}_a^n ; \; \boldsymbol{r}_a^{n+1} = \boldsymbol{r}_a^n + \Delta t \boldsymbol{V}_a^n + 0.5\Delta t^2 \boldsymbol{F}_a^n ; \rho_a^{n+1} = \rho_a^n + \Delta t D_a^n \quad (27)$$

This second part is designed to stop divergence of integrated values through time as the equations are no longer coupled. In cases where the Verlet scheme is used but it is found that numerical stability is an issue, it may be sensible to increase the frequency at which the second part of this scheme is applied, however if it should be necessary to increase this frequency beyond Ns = 10 then this could indicate that the scheme is not able to capture the dynamics of the case in hand suitably and the Symplectic scheme should be used instead.

## Symplectic Scheme

Symplectic integration algorithms are time reversible in the absence of friction or viscous effects [Leimkuhler, 1996]. They can also preserve geometric features, such as the energy time-reversal symmetry present in the equations of motion, leading to improved resolution of long term solution behaviour. The scheme used here is an explicit second-order Symplectic scheme with an accuracy in time of O(Δt2) and involves a predictor and corrector stage.

During the predictor stage the values of acceleration and density are estimated at the middle of the time step according to

$$\boldsymbol{r}_a^{n+\frac{1}{2}} = \boldsymbol{r}_a^n + \frac{\Delta t}{2}\boldsymbol{v}_a^n \quad ; \quad \rho_a^{n+\frac{1}{2}} = \rho_a^n + \frac{\Delta t}{2} D_a^n . \quad (28)$$

During the corrector stage $\mathrm{d}\boldsymbol{v}_a^{n+\frac{1}{2}} / dt$ is used to calculate the corrected velocity, and therefore position, of the particles at the end of the time step according to

$$\begin{aligned} \boldsymbol{v}_a^{n+1} &= \boldsymbol{v}_a^{n+\frac{1}{2}} + \frac{\Delta t}{2}\boldsymbol{F}_a^{n+\frac{1}{2}}, \\ \boldsymbol{r}_a^{n+1} &= \boldsymbol{r}_a^{n+\frac{1}{2}} + \frac{\Delta t}{2}\boldsymbol{v}_a^{n+1} . \end{aligned} \quad (29)$$

and finally the corrected value of density $d\rho_a^{n+1} / dt = D_a^{n+1}$ is calculated using the updated values of $\boldsymbol{v}_a^{n+1}$ and $\boldsymbol{r}_a^{n+1}$ [Monaghan, 2005].

## Variable Time Step

With explicit time integration schemes the timestep is dependent on the Courant- Friedrichs-Lewy (CFL) condition, the forcing terms and the viscous diffusion term. A variable time step Δt is calculated according to [Monaghan et al., 1999] using

$$\Delta t = CFL \cdot \min\left(\Delta t_f, \Delta t_{cv}\right)$$
$$\Delta t_f = \min_a\left(\sqrt{h/|\boldsymbol{f}_a|}\right)$$
$$\Delta t_{cv} = \min_a \frac{h}{c_s + \max_b \left|\frac{h\boldsymbol{v}_{ab}\cdot\boldsymbol{r}_{ab}}{(r_{ab}^2+\eta^2)}\right|} \quad (30)$$

where Δtf is based on the force per unit mass (|fal), and Δtcv combines the Courant and the viscous time step controls.

## Boundary Conditions

In DualSPHysics, the boundary is described by a set of particles that are considered as a separate set to the fluid particles. The software currently provides functionality for solid impermeable and periodic open boundaries. Methods to allow boundary particles to be moved according to fixed forcing functions are also present.

Dynamic Boundary Condition

The Dynamic Boundary Condition (DBC) is the default method provided by DualSPHysics [Crespo et al., 2007]. This method sees boundary particles that satisfy the same equations as fluid particles, however they do not move according to the forces exerted on them. Instead, they remain either fixed in position or move according to an imposed/assigned motion function (i.e. moving objects such as gates, wave-makers or floating objects).

When a fluid particle approaches a boundary and the distance between its particles and the fluid particle becomes smaller than twice the smoothing length (h), the density of the affected boundary particles increases, resulting in a pressure increase. In turn this results in a repulsive force being exerted on the fluid particle due to the pressure term in the momentum equation.

Stability of this method relies on the length of time step taken being suitably short in order to handle the highest present velocity of any fluid particles currently interacting with boundary particles and is therefore an important point when considering how the variable time step is calculated.

Different boundary conditions have been tested in DualSPHysics in the work of [Domínguez et al., 2015]: Dynamic Boundary Condition (DBC), Local Uniform STencil (LUST) and Boundary Integral (INTEGRAL). Validations with dam-break flows and sloshing tanks highlighted the advantages and drawbacks of each method.

Periodic Open Boundary Condition

DualSPHysics provides support for open boundaries in the form of a periodic boundary condition. This is achieved by allowing particles that are near an open lateral boundary to interact with the fluid particles near the complimentary open lateral boundary on the other side of the domain.

In effect, the compact support kernel of a particle is clipped by the nearest open boundary and the remainder of its clipped support applied at the complimentary open boundary [Gómez-Gesteira et al., 2012a].

Pre-imposed Boundary Motion

Within DualSPHysics it is possible to define a pre-imposed movement for a set of boundary particles. Various predefined movement functions are available as well as the ability to assign a time-dependant input file containing kinematic detail. These boundary particles behave as a DBC described in Section 3.8.1, however rather than being fixed, they move independently of the forces currently acting upon them. This provides the ability to define complex simulation scenarios (i.e. a wave-making paddle) as the boundaries influence the fluid particles appropriately as they move.

Fluid-driven Objects

It is also possible to derive the movement of an object by considering its interaction with fluid particles and using these forces to drive its motion. This can be achieved by summing the force contributions for an entire body. By assuming that the body is rigid, the net force on each boundary particle is computed according to the sum of the contributions of all surrounding fluid particles according to the designated kernel function and smoothing length. Each boundary particle k therefore experiences a force per unit mass given by

$$\boldsymbol{f}_k = \sum_{a \in WPs} \boldsymbol{f}_{ka} \quad (31)$$

where fka is the force per unit mass exerted by the fluid particle a on the boundary particle k, which is given by

$$m_k \boldsymbol{f}_{ka} = -m_a \boldsymbol{f}_{ak} \quad (32)$$

For the motion of the moving body, the basic equations of rigid body dynamics can then be used

$$M \frac{d\boldsymbol{V}}{dt} = \sum_{k \in BPs} m_k \boldsymbol{f}_k \quad (33a)$$

$$I \frac{d\boldsymbol{\Omega}}{dt} = \sum_{k \in BPs} m_k \left(\boldsymbol{r}_k - \boldsymbol{R}_0\right) \times \boldsymbol{f}_k \quad (33b)$$

where M is the mass of the object, I the moment of inertia, V the velocity, Ω the rotational velocity and R0 the centre of mass. Equations 33a and 33b are integrated in time in order to predict the values of V and Ω for the beginning of the next time step. Each boundary particle within the body then has a velocity given by

$$\boldsymbol{u}_k = \boldsymbol{V} + \boldsymbol{\Omega} \times \left(\boldsymbol{r}_k - \boldsymbol{R}_0\right) \quad (34)$$

Finally, the boundary particles within the rigid body are moved by integrating Eq. 34 in time. The works of [Monaghan et al., 2003] and [Monaghan, 2005] show that this technique conserves both linear and angular momentum. [Bouscasse et al., 2013] presented successful validations of nonlinear water wave interaction with floating bodies in SPH comparing with experimental data from [Hadzić et al., 2005] that includes deformations in the free-surface due to the presence of floating boxes and the movement of those objects during the experiment (heave, surge and roll displacements). Several validations using DualSPHysics are performed in [Canelas et al., 2015] that analyse the buoyancy-driven motion with solid objects larger than the smallest flow scales and with various densities. They compared SPH numerical results with analytical solutions, with other numerical methods [Fekken, 2004] and with experimental measurements.

## Wave Generation

Wave generation is included in this version of DualSPHysics, for long-crested waves only. In this way, the numerical model can be used to simulate a physical wave flume. Both regular and random waves can be generated. The following sections refer only to the piston-type wavemaker.

First order wave generation

The Biesel transfer functions express the relation between wave amplitude and wavemaker displacement [Biesel and Suquet, 1951], under the assumption of irrotational and incompressible fluid and constant pressure at the free surface. The transfer function links the displacement of the piston-type wavemaker to the water surface elevation, under the hypothesis of monochromatic sinusoidal waves in one dimension in the x-direction:

$$\eta(x,t) = \frac{H}{2}\cos(\omega t - kx + \delta) \quad (35)$$

where H is the wave height, d the water depth, x is distance and δ is the initial phase. The quantity ω=2π/T is the angular frequency and k=2π/L is the wave number with T equal to the wave period and L the wave length. The initial phase δ is given by a random number between 0 and 2π.

Eq. 35 expresses the surface elevation at infinity that Biesel defined as the far-field solution. The Biesel function can be derived for the far-field solution and for a pistontype wavemaker as:

$$\frac{H}{S_0} = \frac{2\sinh^2(kd)}{\sinh(kd)\cosh(kd) + kd} \quad (36)$$

where S0 is the piston stroke. Once the piston stroke is defined, the time series of the piston movement is given by:

$$e_1(t) = \frac{S_0}{2}\sin(\omega t + \delta) \quad (37)$$

Second order wave generation

The implementation of a second order wavemaker theory will prevent the generation of spurious secondary waves. The second order wave generation theory implemented in DualSPHysics is based on [Madsen, 1971] who developed a simple second-order wavemaker theory to generate long second order Stokes waves that would not change shape as they propagated. The theory proposed by [Madsen, 1971] is straightforward, controllable, computationally inexpensive with efficient results, and is accurate for waves of first and second order.

The piston stroke S0 can be redefined from Eq. 36 as S0=H/m1 where:

$$m_1 = \frac{2\sinh^2(kd)}{\sinh(kd)\cosh(kd) + kd} \quad (38)$$

Following [Madsen, 1971], to generate a wave of second order, an extra term must be added to Eq. 37. This

term, for piston-type wavemaker, is equal to:

$$e_2(t) = \left[\left(\frac{H^2}{32d}\right)\cdot\left(\frac{3\cosh(kd)}{\sinh^3(kd)}\right) - \frac{2}{m_1}\right]\sin(2\omega t + 2\delta) \quad (39)$$

Therefore, the piston displacement, for regular waves, is the summation of Eq. 37 and Eq. 39:

$$e(t) = \frac{S_0}{2}\sin(\omega t + \delta) + \left[\left(\frac{H^2}{32d}\right)\cdot\left(\frac{3\cosh(kd)}{\sinh^3(kd)}\right) - \frac{2}{m_1}\right]\sin(2\omega t + 2\delta) \quad (40)$$

Madsen limited the application of this approximate second order wavemaker theory to waves that complied with the condition given by HL2/d3 < 8π2/3. A specific warning is implemented in DualSPHysics to inform the user whether or not this condition is fulfilled.

First order wave generation of irregular waves

Monochromatic waves are not representative of sea states that characterise real wave storm conditions. Sea waves are mostly random or irregular in nature. Irregular wave generation is performed in DualSPHysics based on [Liu and Frigaard, 2001]. Starting from an assigned wave spectra, the Biesel transfer function (Eq. 36) is applied to each component in which the spectrum is discretised. The procedure for the generation of irregular waves is summarised as follows:

1. Defining the wave spectrum through its characteristic parameters (peak frequency, spectrum shape, etc.).
2. Dividing the spectrum into N parts (N>50) in the interval (fstart, fstop), where generally the values assumed by the spectrum (Sη) at the extremes of this interval are smaller than the value assumed for the peak frequency, fp: Sη(fstart)≤0.01·Sη(fp) and Sη(fstop) ≤ 0.01 · Sη(fp).
3. The frequency band width is so-defined as Δf=(fstop-fstart)/N. The irregular wave is decomposed into N linear waves.
4. Determining the angular frequency ωi, amplitude ai and initial phase δi (random number between 0 and 2π) of each i-th linear wave. The angular frequency ωi and amplitude ai can therefore be expressed as follows:

$$\omega_i = 2\pi f_i \quad (41)$$

$$a_i = \sqrt{2S_\eta(f_i)\Delta f} = \frac{H_i}{2} \quad (42)$$

1. Converting the time series of surface elevation into the time series of piston movement with the help of Biesel transfer function:

$$\frac{H_i}{S_{0,i}} = \frac{2\sinh^2(k_i d)}{\sinh(k_i d)\cosh(k_i d) + k_i d} \quad (43)$$

1. Composing all the i-th components derived from the previous equation into the time series of the piston displacement as:

$$e(t) = \sum_{i=1}^{N} \frac{S_{0,i}}{2} \sin(\omega_i t + \delta_i) \quad (44)$$

In DualSPHysics two standard wave spectra have been implemented and used to generate irregular waves: JONSWAP and Pierson-Moskowitz spectra. The characteristic parameters of each spectrum can be assigned by the user together with the value of N (number of parts in which the spectrum is divided).

The user can choose among four different ways to define the angular frequency. It can be determined assuming an equidistant discretization of the wave spectrum (fi=fstart+iΔf-Δf/2), or chosen as unevenly distributed between (i-0.5)Δf and (i+0.5)Δf. An unevenly distributed band width should be preferred: in fact, depending on N, an equidistant splitting can lead to the repetition of the same wave group in the time series that can be easily avoided using an unevenly distributed band width. The last two ways to determine the angular frequency of each component and its band width consist of the application of a stretched or cosine stretched function. The use of a stretched or cosine stretched function has been proven to lead the most accurate results in terms of wave height distribution and groupiness, even when the number of spectrum components N, is relatively low. If there is a certain wave group that is repeating, finally the full range of wave heights and wave periods is not reproduced and statistically the wave train is not representing a real sea state of random waves.

A phase seed is also used and can be changed in DualSPHysics to obtain different random series of δi. Changing the phase seed allows generating different irregular wave time series both with the same significant wave height (Hm0) and peak period (Tp).

## Coupling with Discrete Element Method (DEM)

The discrete element method (DEM) allows for the computation of rigid particle dynamics, by considering contact laws to account for interaction forces. The coupled numerical solution, based on SPH and DEM discretisations, resolves solid-solid and solid-fluid interactions over a broad range of scales.

Forces arise whenever a particle of a solid object interacts with another. In the particular case of a solid-solid collision, the contact force is decomposed into Fn and Ft, normal and tangential components respectively. Both of these forces include viscous dissipation effects. This is because two colliding bodies undergo a deformation which will be somewhere between perfectly inelastic and perfectly elastic, usually quantified by the normal restitution coefficient

$$e_n = -\frac{v_n\big|_{t=t^n}}{v_n\big|_{t=0}}, e \in [0,1] \quad (45)$$

The total forces are decomposed into a repulsion force, Fr, arising from the elastic

deformation of the material, and a damping force, Fd, for the viscous dissipation of energy during the deformation.

Figure 3-1 generally illustrates the proposed viscoelastic DEM mechanism between two interacting particles.

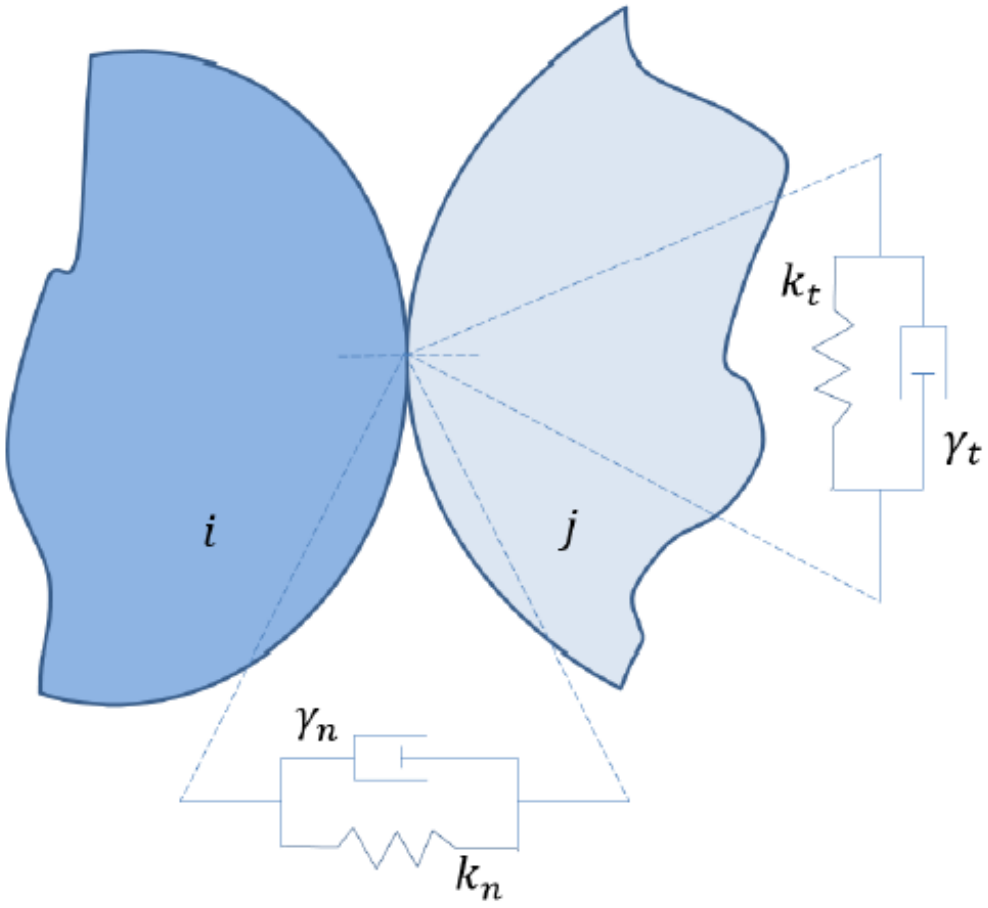



**Figure 3-1.** Schematic interaction between particles with viscoelastic DEM mechanism.

The normal force is given by

$$\mathbf{F}_{n,ij} = \mathbf{F}_n^r + \mathbf{F}_n^d = k_{n,ij}\delta_{ij}^{3/2}\mathbf{e}_{ij}^n - \gamma_{n,ij}\delta_{ij}^{1/2}\dot{\delta}_{ij}\,\mathbf{e}_{ij}^n \quad (46)$$

where the stiffness is given by

$$k_{n,ij} = \frac{4}{3}E^*\sqrt{R^*} \quad (47)$$

and the damping coefficient is

$$\gamma_{n,ij} = -\frac{\log e_{ij}}{\sqrt{\pi^2 + \log^2 e_{ij}}} \quad (48)$$

where $\mathbf{e}_{ij}^n$ is the unit vector between the centers of particles i and j.

The restitution coefficient ij e is taken as the average of the two materials coefficients, in what is the only calibration parameter of the model. It is not a physical parameter, but since the current method does not account for internal deformations and other energy losses during contact, the user is given a choice to change this parameter freely in order to control the dissipation of each contact. The reduced radius and reduced elasticity are given by

$$R^* = \left( \frac{1}{R_1} + \frac{1}{R_2} \right)^{-1} ; E^* = \left( \frac{1-\nu_{p^1}^2}{E_1} + \frac{1-\nu_{p^2}^2}{E_2} \right)^{-1} \quad (49)$$

where Ri is simply the particle radius, Ei is the Young modulus and vp is the Poisson coefficient of material i, as specified in the Floating_Materials.xml.

This results in another restriction to the time-step, adding

$$\Delta t_{c,ij} = \frac{3.21}{50} \left( -\frac{M^*}{k_{n,if}} \right)^{2/5} v_{n,ij}^{-1/5} \quad (50)$$

to the existing CFL restrictions (Eq. 30), where vn is the normal relative velocity and M* is the reduced mass of the system where there is a contact.

Regarding tangential contacts, friction is modelled using the same model:

$$\mathbf{F}_{t,ij} = \mathbf{F}_t^r + \mathbf{F}_t^d = k_{t,ij} \delta_{ij}^t \mathbf{e}_{ij}^t - \gamma_{t,ij} \delta_{ij}^t \dot{\delta}_{ij} \mathbf{e}_{ij}^t \quad (51)$$

where the stiffness and damping constants are derived to be

$$k_{t,ij} = 2/7 k_{n,ij} ; \gamma_{t,ij} = 2/7 \gamma_{n,ij} \quad (52)$$

as to insure internal consistency of the time scales between normal and tangential components. This mechanism models the static and dynamic friction mechanisms by a penalty method. The body does not statically stick at the point of contact, but is constrained by the spring-damper system. This force must be bounded above by the Coulomb friction law, modified with a sigmoidal function in order to make it continuous around the origin regarding the tangential velocity:

$$\mathbf{F}_{t,ij} = \min \left( \mu_{IJ} \mathbf{F}_{n,ij} \tanh(8\delta_{ij}^t) \mathbf{e}_{ij}^t, \mathbf{F}_{t,ij} \right) \quad (53)$$

where μIJ is the friction coefficient at the contact of object I and object J and is simply taken as the average of the two friction coefficients of the distinct materials, indicated in the Floating_Materials.xml.

More information about DEM implementation can be found in [Canelas, 2015; Canelas et al., 2016].

## Multi-phase: Two-phase liquid-sediment implementation in DualSPHysics

This guide provides a concise depiction of the multi-phase liquid-sediment model implemented in DualSPHysics solver. The model is capable of simulating problems involving liquid-sediment phases with the addition of highly non-linear deformations and free-surface flows which are frequently encountered in applied hydrodynamics. More specifically, the two-phase liquid-solid model is aimed at flow-induced erosion of fully saturated sediment. Applications include scouring in industrial tanks, port hydrodynamics, wave breaking in coastal applications and scour around structures in civil and environmental engineering flows among others.

Description of the physical problem

A typical saturated sediment scour induced by rapid liquid flow at the interface undergoes a number of different behavioural regime changes mostly govern by the characteristics of the sediment and liquid phase rheology at the interface. These sediment regimes are distinguishable as an un-yielded region of sediment, a yielded non-Newtonian region and a pseudo Newtonian sediment suspension region where the sediment is entrained by the liquid flow. These physical processes can be described by the Coulomb shear stress τmc, the cohesive yield strength τc which accounts for the cohesive nature of fine sediment, the viscous shear stress τv which accounts for the fluid particle viscosity, the turbulent shear stress of the sediment particle τt and the dispersive stress τd which accounts for the collision of larger fraction granulate. The total shear stress can be expressed as

$$\tau^{\alpha\beta} = \tau_{mc} + \tau_c + \tau_v^{\alpha\beta} + \tau_t^{\alpha\beta} + \tau_d^{\alpha\beta} \quad (54)$$

The first two parameters on the right-hand side of the equation define the yield strength of the material and thus can be used to differentiate the un-yielded or yielded region of the sediment state according to the induced stress by the liquid phase in the interface. The model implemented in DualSPHysics uses the Drucker-Prager yield criterion to evaluate yield strength of the sediment phase and the sediment failure surface.

When the material yields the sediment behaves as a non-Newtonian rate dependent Bingham fluid that accounts for the viscous and turbulent effects of the total shear stress of Eq. 54. Typically, sediment behaves as a shear thinning material with a low and high shear stress state of a pseudo-Newtonian and plastic viscous regime respectively. Herein, the Herschel-Buckley-Papanastasiou model is employed as a power law Bingham model. This combines the yielded and un-yielded region using an exponential stress growth parameter and a power law Bingham model for the shear thinning or thickening plastic region.

Finally, the characteristics of the low concentration suspended sediment that has been entrained by the liquid are modelled using a volumetric concentration based viscosity in a pseudo-Newtonian approach by employing the Vand equation.

Sediment phase

The yield surface prediction is modelled using the Drucker-Prager (DP) model. The DP can be written in a general form as [Fourtakas and Rogers, 2016]

$$f(I_1, J_2) = \sqrt{J_2} + (ap - \kappa) = 0 \quad (55)$$

The parameters a and κ can be determined by projecting the Drucker-Prager onto the Mohr-Coulomb yield criterion in a deviatoric plane

$$a = -\frac{2\sqrt{3}\sin(\phi)}{3-\sin(\phi)} \qquad \kappa = \frac{2\sqrt{3}\cos(\phi)}{3-\sin(\phi)} \quad (56)$$

where ϕ is the internal friction and c the cohesion of the material. Finally, yielding will occur when the following equation is satisfied

$$-ap + \kappa < 2\mu_d \sqrt{II_D} \quad (57)$$

The multi-phase model uses the Herschel-Bulkley-Papanastasiou (HBP) [Papanastasiou, 1987] rheological characteristics to model the yielded region. The HBP model reads

$$\phi_1 = \frac{|\tau_y|}{\sqrt{\Pi_D}}\left[1 - e^{-m\sqrt{\Pi_D}}\right] + 2\mu\left|4II_D\right|^{\frac{n-1}{2}} \quad (58)$$

where m controls the exponential growth of stress, n is the power law index and μ is the apparent dynamic viscosity (or consistency index for sediment flows). Figure 3-2(a) shows the initial rapid growth of stress by varying m whereas Figure 3-2(b) shows the effect of the power law index n.

Note that as m → ∞ the HBP model reduces to the original Herschel-Bulkley model and when n=1 the model reduces to a simple Bingham model. Consequently, when n=1 and m=0 the model reduces to a Newtonian constitutive equation. Therefore, both phases can be modelled using the same constitutive equation. Most importantly, since the HBP parameters can be adjusted independently for each phase the current model is not restricted to Newtonian/Non-Newtonian formulation but can simulate a variety of combinations of flows (i.e. Newtonian/Newtonian, Non-Newtonian/Non-Newtonian with or without a yield strength, etc.).

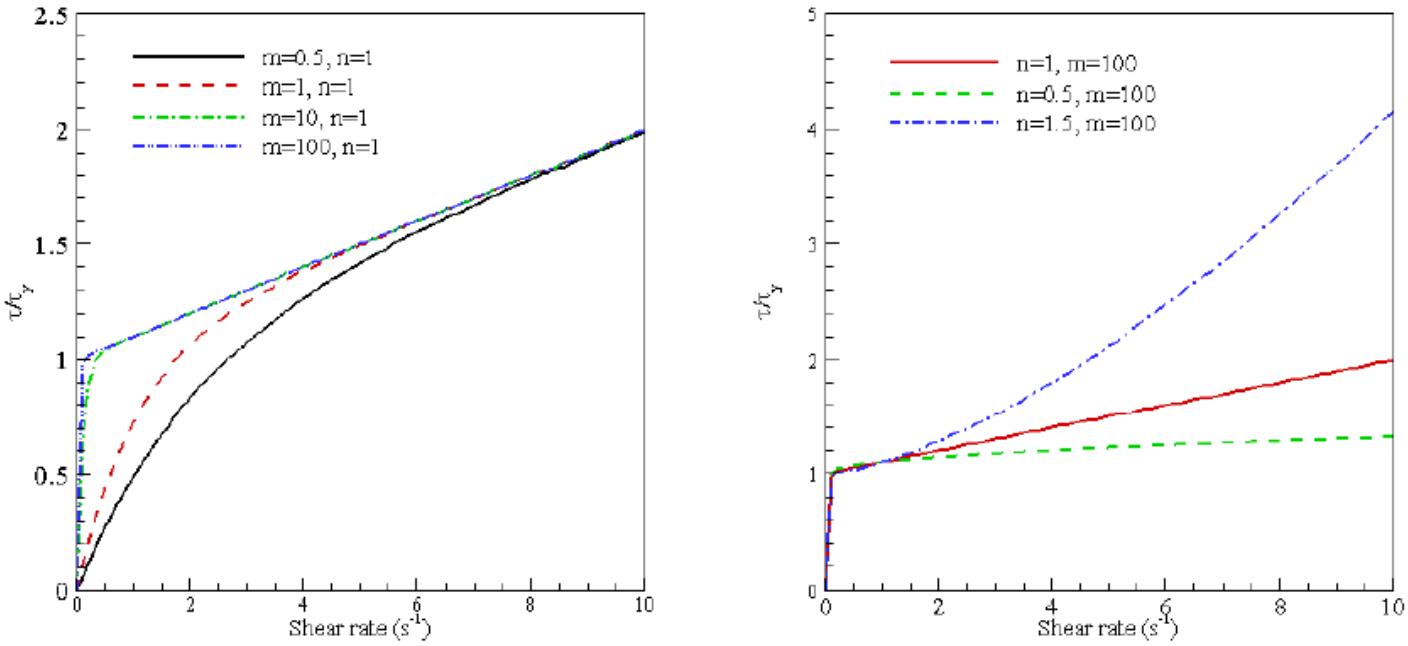



**Figure 3-2.** Initial rapid growth of stress by varying m and effect of the power law index n for the HBP model.

The rheological characteristics of the sediment entrainment by the fluid can be controlled through the volume fraction of the mixture by using a concentration volume fraction in the form of

$$c_{v,i} = \frac{\sum_{j_{sat}\in 2h}^{N} \frac{m_j}{\rho_j}}{\sum_{j\in 2h}^{N} \frac{m_j}{\rho_j}} \quad (59)$$

where the summation is defined within the support of the kernel and jsat refers to the yielded saturated sediment particles only.

We use a suspension viscosity based on the Vand experimental colloidal suspension equation [Vand, 1948] of sediment in a fluid by

$$\mu_{susp} = \mu e^{\frac{2.5c_v}{1-\frac{39}{64}c_v}} \qquad c_v \leq 0.3 \tag{60}$$

assuming an isotropic material with spherically shaped sediment particles. Eq. 60 is applied only when the volumetric concentration of the saturated sediment particle within the SPH kernel is lower than 0.3, which is the upper validity limit of Eq. 60.

More information about this multi-phase implementation can be also found in [Fourtakas, 2014; Fourtakas and Rogers, 2016].

# CPU and GPU implementation

## General Information

Detailed information about the CPU and GPU implementation can be found in the papers:

Crespo AJC, Domínguez JM, Rogers BD, Gómez-Gesteira M, Longshaw S, Canelas R, Vacondio R, Barreiro A, García-Feal O. 2015. DualSPHysics: open-source parallel CFD solver on Smoothed Particle Hydrodynamics (SPH). Computer Physics Communications, 187: 204-216. doi: 10.1016/j.cpc.2014.10.004

Domínguez JM, Crespo AJC, Valdez-Balderas D, Rogers BD. and Gómez-Gesteira M. 2013. New multi-GPU implementation for Smoothed Particle Hydrodynamics on heterogeneous clusters. Computer Physics Communications, 184: 1848-1860. doi: 10.1016/j.cpc.2013.03.008

Domínguez JM, Crespo AJC and Gómez-Gesteira M. 2013. Optimization strategies for CPU and GPU implementations of a smoothed particle hydrodynamics method. Computer Physics Communications, 184(3): 617-627. doi:10.1016/j.cpc.2012.10.015

Valdez-Balderas D, Domínguez JM, Rogers BD, Crespo AJC. 2012. Towards accelerating smoothed particle hydrodynamics simulations for free-surface flows on multi-GPU clusters. Journal of Parallel and Distributed Computing. doi:10.1016/j.jpdc.2012.07.010

Crespo AJC, Domínguez JM, Barreiro A, Gómez-Gesteira M and Rogers BD. 2011. GPUs, a new tool of acceleration in CFD: Efficiency and reliability on Smoothed Particle Hydrodynamics methods. PLoS ONE, 6(6), e20685. doi:10.1371/journal.pone.0020685

Domínguez JM, Crespo AJC, Gómez-Gesteira M, Marongiu, JC. 2011. Neighbour lists in Smoothed Particle Hydrodynamics. International Journal For Numerical Methods in Fluids, 67(12): 2026-2042. doi: 10.1002/fld.2481

The DualSPHysics code is the result of an optimised implementation using the best approaches for CPU and GPU with the accuracy, robustness and reliability shown by the SPHysics code. SPH simulations such as those in the SPHysics and DualSPHysics codes can be split in three main steps; (i) generation of the neighbour list, (ii) computation of the forces between particles (solving momentum and continuity equations) and (iii) the update of the physical quantities at the next time step. Thus, running a simulation means executing these steps in an iterative manner:

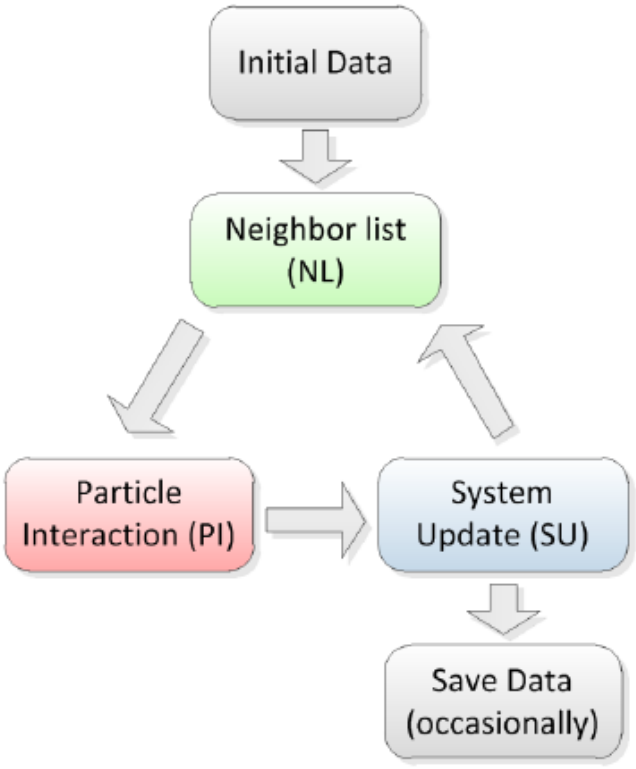
1. Fist step: Neighbour list (Cell-linked list described in [Domínguez et al., 2011]):
2. Domain is divided into square cells of side 2h (or the size of the kernel domain).
3. A list of particles, ordered according to the cell to which they belong, is generated.
4. All the arrays with the physical variables belonging the particles are reordered according the list of particles.
5. Second step: Force computation:
6. Particles of the same cell and adjacent cells are candidates to be neighbours.
7. Each particle interacts with all its neighbouring particles (at a distance < 2h).
8. Third step: System Update:

9. New time step is computed.
10. Physical quantities for the next step are updated starting from the values of physical variables at the present or previous time steps using the particle interactions.
11. Particle information (velocity and density) are saved on local storage (the hard drive) at defined times.

The GPU implementation is focused on the force computation since following [Domínguez et al., 2011] this is the most consuming part in terms of runtime. However the most efficient technique consists of minimising the communications between the CPU and GPU for the data transfers. If neighbour list and system update are also implemented on the GPU the CPU-GPU memory transfer is needed at the beginning of the simulation while relevant data will be transferred to the CPU when saving output data is required (usually infrequently). [Crespo et al., 2011] used an execution of DualSPHysics performed entirely on the GPU to run a numerical experiment where the results are in close agreement with the experimental results.

The GPU implementation presents some key differences in comparison to the CPU version. The main difference is the parallel execution of all tasks that can be parallelised such as all loops regarding particles. One GPU execution thread computes the resulting force of one particle performing all the interactions with its neighbours. Different to previous versions, in version 4.0 onwards, the symmetry of the particle interaction is not employed on the CPU, the same as in the GPU implementation. On a GPU it is not efficient due to memory coalescence issues. Now, the new CPU structure mimics the GPU threads, which ensures continuity of coding and structure (hence ease of debugging, etc.) – see Section 6.

DualSPHysics is unique where the same application can be run using either the CPU or GPU implementation; this facilitates the use of the code not only on workstations with an Nvidia GPU but also on machines without a CUDA-enabled GPU. The CPU version is parallelised using the OpenMP API. The main code has a common core for both the CPU and GPU implementations with only minor source code differences implemented for the two devices applying the specific optimizations for CPU and GPU. Thus, debugging or maintenance is easier and comparisons of results and computational time are more direct.



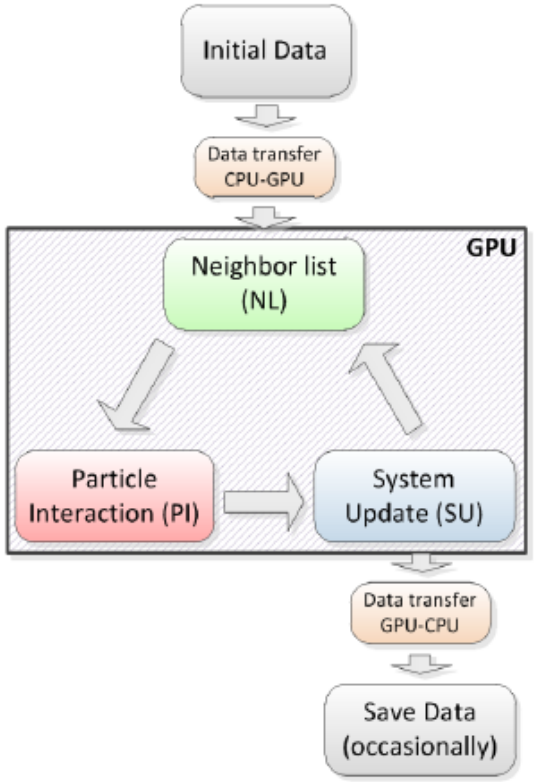



**Figure 4-1.** Flow diagram of the CPU (left) and total GPU implementation (right).

## Double Precision

The parallel computing power of Graphics Processing Units (GPUs) has led to an important increase in the size of the simulations but problems of precision can appear when simulating large domains with high resolution (specially in 2-D simulations). Hence, there are numerous simulations that require a small particle size “dp” relative to the computational domain [Domínguez et al., 2013c], namely fine resolution or long domains. DualSPHysics v4.0 now includes an implementation with double precision where necessary. For example, arrays of position now use double precision and updating state of particles is also implemented with double precision.

## OpenMP for multi-core executions.

In the new version, the CPU implementation aims to achieve a higher performance in the current machines that contains many cores. Now, the interactions of a given particles with all its neighbours are carried out by the same execution thread. Symmetry in the force computation is not applied in order to increase the parallelization level of the algorithms. Previous versions of DualSPHysics were fast on CPUs with 4-8 cores but efficiency decreases significantly with number of cores. Furthermore, memory consumption increases with more cores. The new CPU code achieves an efficiency of 86.2% simulating 150,000 particles with 32 cores while the same execution achieved an efficiency of only 59.7% with previous version 3.0 and without memory increase. The current OpenMP implementation is not only fast and efficient with many cores, but also offers more advantages for users that want to modify the code; implementation is now easier since symmetry is removed during force computation such that the CPU code is more similar to the GPU code which facilitates its comprehension and editing.

## Optimisation of the size of blocks for execution of CUDA kernels.

The size of the blocks is a key parameter in the execution of CUDA kernels since it can lead to performance differences of 50%. This variation becomes more significant in the kernels that compute particle interactions since take more than 90% of the total execution time.

The version 4.0 includes a new automatic estimation of the optimum block size of CUDA kernels (particle interactions on GPU). This optimum block size depends on: (i) features of the kernel (registers and shared memory), (ii) compilation parameters and the CUDA version, (iii) hardware to be used and GPU specifications and (iv) input data to be processed by the kernel (divergence, memory coalescent access). The CUDA Occupancy Calculator is available from CUDA version 6.5.

# DualSPHysics open source code

This section provides a brief description of the source files of DualSPHysics v4.0. The source code is freely redistributable under the terms of the GNU General Public License (GPL) as published by the Free Software Foundation (www.gnu.org/licenses/).

The documentation has been created using the documentation system Doxygen (www.doxygen.org). The user can open the HTML file index.html (Figure 6-1) located in the directory mentioned above to navigate through the full documentation.

## Common Files

### Functions.h & Functions.cpp

Declares/implements basic/general functions for the entire application.

### JBinaryData.h & JBinaryData.cpp

Declares/implements the class that defines any binary format of a file.

### JException.h & JException.cpp

Declares/implements the class that defines exceptions with the information of the class and method.

### JLog2.h & JLog2.cpp

Declares/implements the class that manages the output of information in the file Run.out and on screen.

### JMatrix4.h

Declares the template for a matrix 4x4 used for geometric transformation of points in space.

### JMeanValues.h & JMeanValues.cpp

Declares/implements the class that calculates the average value of a sequence of values.

### JObject.h & JObject.cpp

Declares/implements the class that defines objects with methods that throw exceptions.

### JObjectGpu.h & JObjectGpu.cpp

Declares/implements the class that defines objects with methods that throw exceptions for tasks on the GPU.

### JPartDataBi4.h & JPartDataBi4.cpp

Declares/implements the class that allows reading files with data of particles in format bi4.

### JPartFloatBi4.h & JPartFloatBi4.cpp

Declares/implements the class that allows reading information of floating objects saved during simulation.

## JPartOutBi4Save.h & JPartOutBi4Save.cpp

Declares/implements the class that allows writing information of excluded particles during simulation.

## JRadixSort.h & JRadixSort.cpp

Declares/implements the class that implements the algorithm RadixSort.

## JRangeFilter.h & JRangeFilter.cpp

Declares/implements the class that facilitates filtering values within a list.

## JReadDatafile.h & JReadDatafile.cpp

Declares/implements the class that allows reading data in ASCII files.

## JSpaceCtes.h & JSpaceCtes.cpp

Declares/implements the class that manages the info of constants from the input XML file.

## JSpaceEParms.h & JSpaceEParms.cpp

Declares/implements the class that manages the info of execution parameters from the input XML file.

## JSpaceParts.h & JSpaceParts.cpp

Declares/implements the class that manages the info of particles from the input XML file.

## JSpaceProperties.h & JSpaceProperties.cpp

Declares/implements the class that manages the properties assigned to the particles in the XML file.

## JTimer.h

Declares the class that defines a class to measure short time intervals.

## JTimerCuda.h

Declares the class that defines a class to measure short time intervals on the GPU using cudaEvent.

## TypesDef.h

Declares general types and functions for the entire application.

## JFormatFiles2.h

Declares the class that provides functions to store particle data in formats VTK, CSV, ASCII.

## JFormatFiles2.lib (libjformatfiles2.a)

Precompiled library that provides functions to store particle data in formats VTK, CSV, ASCII.

## JSphMotion.h

Declares the class that provides the displacement of moving objects during a time interval

### JSphMotion.lib (libjsphmotion.a)

Precompiled library that provides the displacement of moving objects during a time interval.

### JXml.h

Declares the class that helps to manage the XML document using library TinyXML

### JXml.lib (libjxml.a)

Precompiled library that helps to manage the XML document using library TinyXML.

### JWaveGen.h

Declares the class that implements wave generation for regular and irregular waves.

### JWaveGen.lib (libjwavegen.a)

Precompiled library that implements wave generation for regular and irregular waves.

## SPH Solver

### main.cpp

Main file of the project that executes the code on CPU or GPU.

### JCfgRun.h & JCfgRun.cpp

Declares/implements the class that defines the class responsible for collecting the execution parameters by command line.

### JPartsLoad4.h & JPartsLoad4.cpp

Declares/implements the class that manages the initial load of particle data.

### JPartsOut.h & JPartsOut.cpp

Declares/implements the class that stores excluded particles at each instant until writing the output file.

### JSaveDt.h & JSaveDt.cpp

Declares/implements the class that manages the use of prefixed values of DT loaded from an input file.

### JSph.h & JSph.cpp

Declares/implements the class that defines all the attributes and functions that CPU and GPU simulations share.

### JSphAccInput.h & JSphAccInput .cpp

Declares/implements the class that manages the application of external forces to different blocks of particles (with the same MK).

### JSphDtFixed.h & JSphDtFixed.cpp

Declares/implements the class that manages the info of dt.

### JSphVisco.h & JSphVisco.cpp

Declares/implements the class that manages the use of viscosity values from an input file.

### JTimerClock.h

Defines a class to measure time intervals with precision of clock().

### JTimeOut.h & JTimeOut.cpp

Declares/implements the class that manages the use of variable output time to save PARTs.

### Types.h

Defines specific types for the SPH application.

## SPH Solver only for CPU executions

### JSphCpu.h & JSphCpu.cpp

Declares/implements the class that defines the attributes and functions used only in CPU simulations.

### JSphCpuSingle.h & JSphCpuSingle.cpp

Declares/implements the class that defines the attributes and functions used only in Single-CPU.

### JSphTimersCpu.h

Measures time intervals during CPU execution.

### JCellDivCpu.h & JCellDivCpu.cpp

Declares/implements the class responsible of generating the Neighbour List in CPU.

### JCellDivCpuSingle.h & JCellDivCpuSingle.cpp

Declares/implements the class responsible of generating the Neighbour List in Single-CPU.

### JArraysCpu.h & JArraysCpu.cpp

Declares/implements the class that manages arrays with memory allocated in CPU.

## SPH Solver only for GPU executions

### JSphGpu.h & JSphGpu.cpp

Declares/implements the class that defines the attributes and functions used only in GPU simulations.

### JSphGpu_ker.h & JSphGpu_ker.cu

Declares/implements functions and CUDA kernels for the Particle Interaction (PI) and System Update (SU).

### JSphGpuSingle.h & JSphGpuSingle.cpp

Declares/implements the class that defines the attributes and functions used only in single-GPU.

### JSphTimersGpu.h

Measures time intervals during GPU execution.

### JCellDivGpu.h & JCellDivGpu.cpp

Declares/implements the class responsible of generating the Neighbour List in GPU.

### JCellDivGpu_ker.h & JCellDivGpu_ker.cu

Declares/implements functions and CUDA kernels to generate the Neighbour List in GPU.

### JCellDivGpuSingle.h & JCellDivGpuSingle.cpp

Declares/implements the class that defines the class responsible of generating the Neighbour List in Single-GPU.

### JCellDivGpuSingle_ker.h & JCellDivGpuSingle_ker.cu

Declares/implements functions and CUDA kernels to compute operations of the Neighbour List.

### JArraysGpu.h & JArraysGpu.cpp

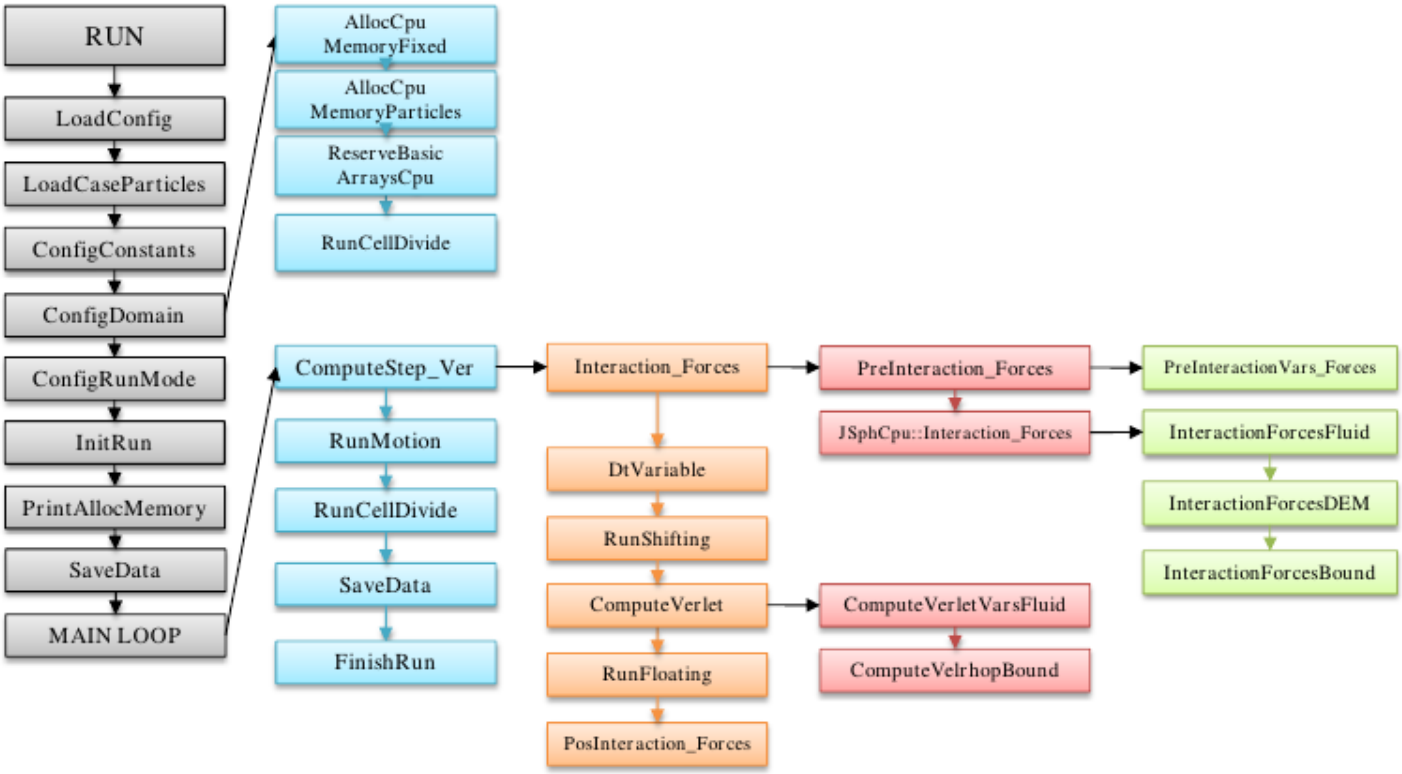
Declares/implements the class that manages arrays with memory allocated in GPU.

### JBlockSizeAuto.h & JBlockSizeAuto.cpp

Declares/implements the class that manages the automatic computation of optimum Blocksize in kernel interactions.

## CPU source files

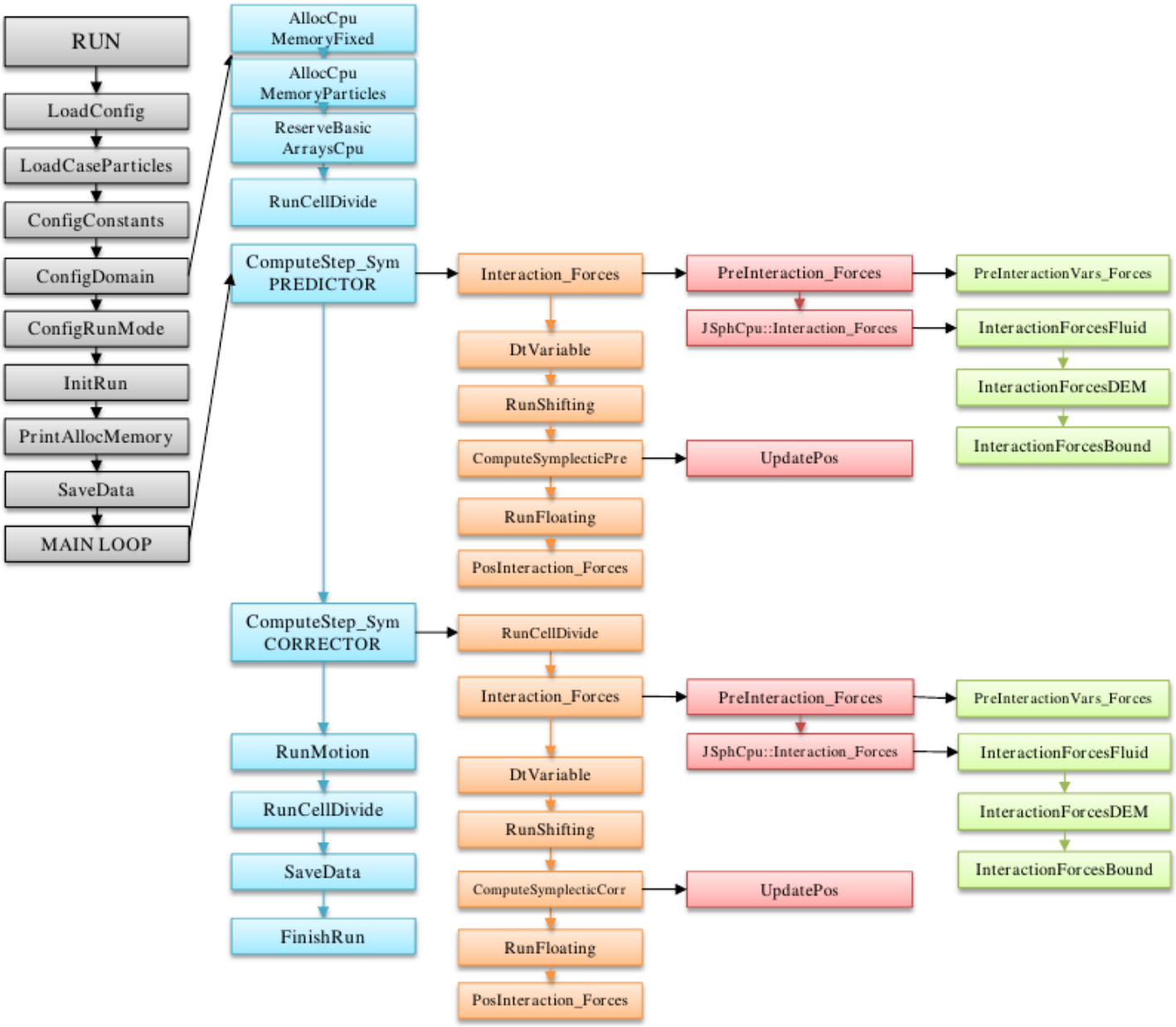
The source file JSphCpuSingle.cpp can be better understood with the help of the diagram of calls represented in Figure 6-2. Note that now particle interactions are no longer performed in terms of cells as used in previous versions.




| | |
|---|---|
| RUN | Starts simulation. |
| LoadConfig | Loads the configuration of the execution. |
| LoadCaseParticles | Loads particles of the case to be processed. |
| ConfigConstants | Configures value of constants. |
| ConfigDomain | Configuration of the current domain. |
| AllocCpuMemory | Allocates memory of main data in CPU. |
| ReserveBasicArraysCpu | Arrays for basic particle data in CPU. |
| RunCellDivide | Generates Neighbour List. |
| ConfigRunMode | Configures execution mode in CPU. |
| InitRun | Initialisation of arrays and variables for the execution. |
| PrintAllocMemory | Visualizes the reserved memory. |
| SaveData | Generates file with particle data of the initial instant. |
| MAIN LOOP | Main loop of the simulation. |
| ComputeStep_Ver | Computes Particle Interaction and System Update using Verlet. |
| Interaction_Forces | Call for Particle Interaction (PI). |
| PreInteraction_Forces | Prepares variables for Particle Interaction. |
| JSphCpu::Interaction_Forces | Computes Particle Interaction. |
| DtVariable | Computes the value of the new variable time step. |
| RunShifting | Applies Shifting algorithm to particles' position. |
| ComputeVerlet | Computes System Update using Verlet. |
| ComputeVerletVarsFluid | Calculates new values of position, velocity & density for fluid particles. |
| ComputeVelrhopBound | Calculates new values of density for boundary particles. |
| RunFloating | Processes movement of particles of floating objects. |
| PosInteraction_forces | Memory release of arrays in CPU. |
| RunMotion | Processes movement of moving boundary particles. |
| RunCellDivide | Generates Neighbour List. |
| SaveData | Generates files with output data. |
| FinishRun | Shows and stores final overview of execution. |

**Figure 6-2.** Workflow of JSphCpuSingle.cpp when using Verlet time algorithm.

When the Symplectic timestepping integration scheme is used the step is split in predictor and corrector steps. Thus, Figure 6-3 shows the workflow and calls of the CPU code using this time scheme:



| | |
|---|---|
| ComputeStep_Sym | Computes Particle Interaction and System Update using Symplectic. |
| PREDICTOR | Predictor step. |
| Interaction_Forces | Call for Particle Interaction (PI). |
| PreInteraction_Forces | Prepares variables for Particle Interaction. |
| Interaction_Forces | Computes Particle Interaction. |
| DtVariable | Computes the value of the new variable time step. |
| RunShifting | Applies Shifting algorithm to particles' position. |
| ComputeSymplecticPre | Computes System Update using Symplectic-Predictor. |
| UpdatePos | Computes new positions of the particles. |
| RunFloating | Processes movement of particles of floating objects. |
| PosInteraction_Forces | Memory release of arrays in CPU. |
| CORRECTOR | Corrector step. |
| Interaction_Forces | Call for Particle Interaction (PI). |
| PreInteraction_Forces | Prepares variables for Particle Interaction. |
| Interaction_Forces | Computes Particle Interaction. |
| DtVariable | Computes the value of the new variable time step. |
| RunShifting | Applies Shifting algorithm to particles' position. |
| ComputeSymplecticCorr | Computes System Update using Symplectic-Corrector. |
| UpdatePos | Computes new positions of the particles. |
| RunFloating | Processes movement of particles of floating objects. |
| PosInteraction_Forces | Memory release of arrays in CPU. |

**Figure 6-3.** Workflow of JSphCpuSingle.cpp when using Symplectic timestepping algorithm

Note that JSphCpu::Interaction_Forces performs the particle interaction in CPU using the template InteractionForcesT. Thus, the interaction between particles is carried out considering different parameters and considering the type of particles involved in the interaction as it can be seen in Figure 6-4 and Table 6-2:

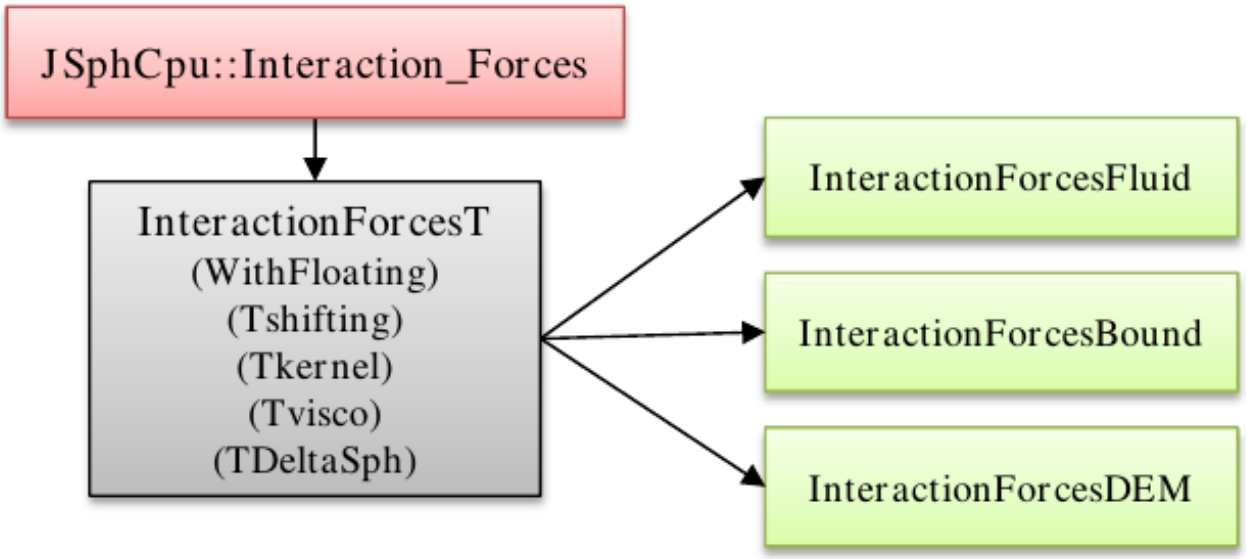



**Figure 6-4.** Call graph for the template InteractionForcesT.

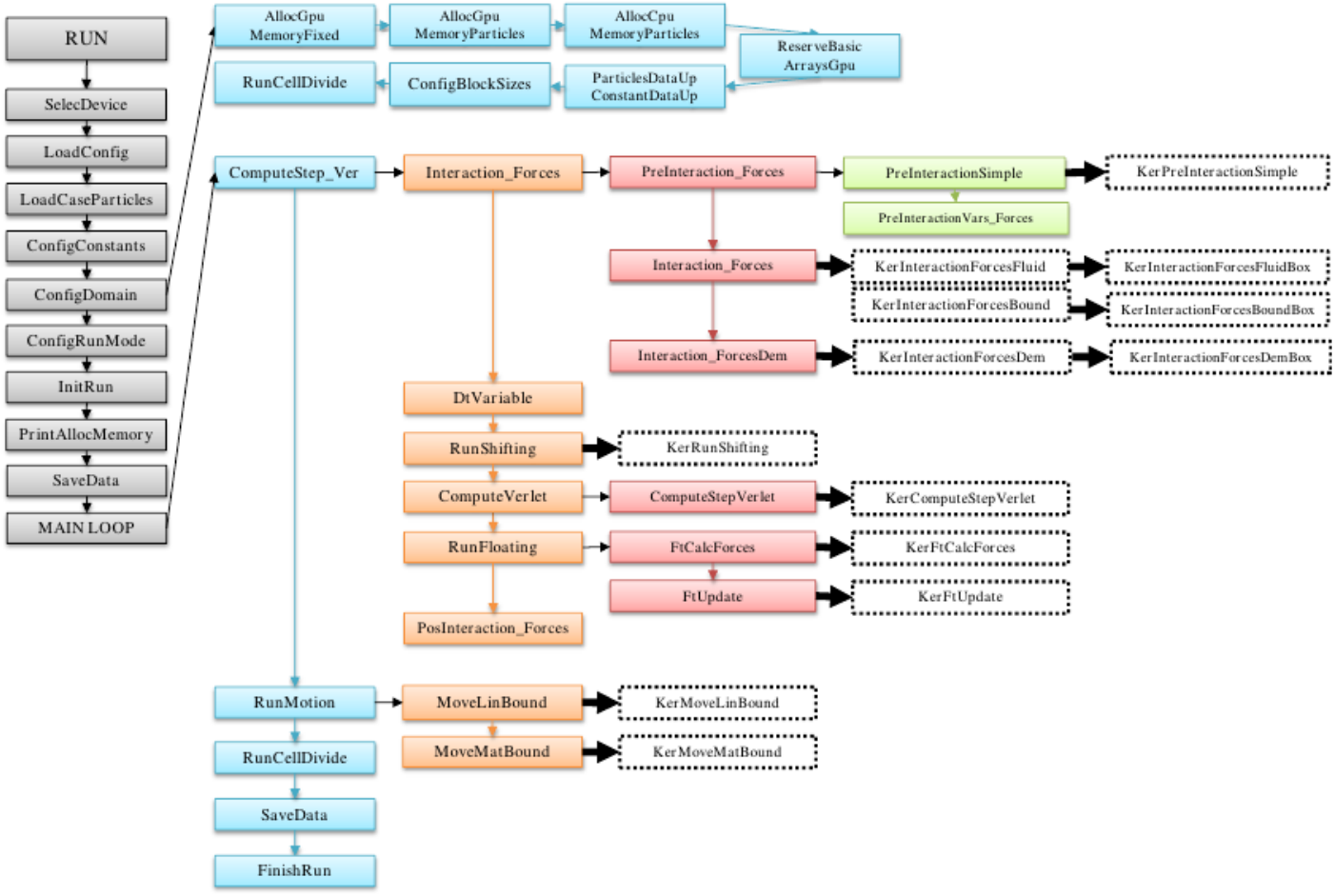
| | |
|---|---|
| InteractionForcesFluid | SPH interaction between particles<br>Fluid/Floating-Fluid/Floating<br>Fluid/Floating-Bound |
| InteractionForcesBound | SPH interaction between particles<br>Bound-Fluid/Floating |
| InteractionForcesDEM | DEM interaction between particles<br>Floating-Bound<br>Floating-Floating |

**Table 6-2.** Different particle interactions can be performed depending on the type of particles.

As mentioned before, a more complete documentation has been generated using Doxygen.

# GPU source files

The source file JSphGpuSingle.cpp can be better understood with the workflow represented in Figure 6-5 that includes the functions implemented in the GPU files. The dashed boxes indicates the CUDA kernels implemented in the CUDA files (JSphGpu_ker.cu).



| | |
|---|---|
| RUN | Starts simulation. |
| SelecDevice | Initialises CUDA device |
| LoadConfig | Loads the configuration of the execution. |
| LoadCaseParticles | Loads particles of the case to be processed. |
| ConfigConstants | Configures value of constants. |
| ConfigDomain | Configuration of the current domain. |
| AllocGpuMemoryFixed | Allocates memory in GPU of for arrays with fixed size. |
| AllocGpuMemoryParticles | Allocates memory in GPU of main data of particles. |
| AllocCpuMemoryParticles | Allocates memory in CPU of main data of particles. |
| ReserveBasicArraysGpu | Arrays for basic particle data in GPU. |
| ParticlesDataUp | Uploads particle data to the GPU. |
| ConstantDataUp | Uploads constants to the GPU. |
| ConfigBlockSizes | Calculates optimum BlockSize. |
| RunCellDivide | Generates Neighbour List. |
| ConfigRunMode | Configures execution mode in GPU. |
| InitRun | Initialisation of arrays and variables for the execution. |
| PrintAllocMemory | Visualizes the reserved memory in CPU and GPU. |
| SaveData | Generates file with particle data of the initial instant. |
| MAIN LOOP | Main loop of the simulation. |
| ComputeStep_Ver | Computes Particle Interaction and System Update using Verlet. |
| Interaction_Forces | Call for Particle Interaction (PI). |
| PreInteraction_Forces | Prepares variables for Particle Interaction. |
| Interaction_Forces | Computes Particle Interaction. |
| Interaction_ForcesDEM | Computes Particle Interaction with DEM. |
| DtVariable | Computes the value of the new variable time step. |
| RunShifting | Applies Shifting algorithm to particles' position. |
| ComputeVerlet | Call for System Update using Verlet. |
| ComputeStepVerlet | Computes System Update using Verlet. |
| RunFloating | Processes movement of particles of floating objects. |
| FtCalcForces | Computes forces on floatings. |
| FtUpdate | Updates information and particles of floating bodies. |
| PosInteractionForces | Memory release of arrays in GPU. |
| RunMotion | Processes movement of moving boundary particles. |
| MoveLinBound | Applies a linear movement to a set of particles. |
| MoveMatBound | Applies a matrix movement to a set of particles. |
| RunCellDivide | Generates Neighbour List. |
| SaveData | Generates files with output data. |
| FinishRun | Shows and stores final overview of execution. |

**Figure 6-5.** Workflow of JSphGpuSingle.cpp when using Verlet time algorithm.

# Compiling DualSPHysics

The code can be compiled for either CPU or CPU&GPU. Please note that both the C++ and CUDA version of the code contain the same features and options. Most of the source code is common to CPU and GPU, which allows the code to be run on workstations without a CUDA-enabled GPU, using only the CPU implementation.

To run DualSPHysics on a GPU using an executable, only an Nvidia CUDA-enabled GPU card is needed and the latest version of the GPU driver must be installed. However, to compile the source code, the GPU programming language CUDA and nvcc compiler must be installed on your computer. CUDA Toolkit X.X can be downloaded from Nvidia website http://developer.nvidia.com/cuda-toolkit-XX. CUDA versions from 4.0 till 7.5 have been tested.

Once the C++ compiler (for example gcc) and the CUDA compiler (nvcc) have been installed in your machine, you can download the relevant files from the repository:

## Windows compilation

The project file DualSPHysics4_vs2010.sln is provided to be opened with Visual Studio 2010 and DualSPHysics4_vs2013.sln to be opened with Visual Studio 2013. Also different configurations can be chosen for compilation:

1. Release - For CPU and GPU
2. ReleaseCPU - Only for CPU

The result of the compilation is the dualsphysics executable.

The Visual Studio project is created including the libraries for OpenMP in the executable. To not include them, user can modify Props config -> C/C++ -> Language -> OpenMp and compile again

The use of OpenMP can be also deactivated by commenting the code line in Types.h: `#define _WITHOMP ///<Enables/Disables OpenMP.`

## Linux compilation

You can build the project in GNU/Linux using the Makefile included in the source folder. Follow this steps (for the GPU version):

1. Clone this repository into your system
2. Ensure you have GCC version 4.X installed. Usually there are packages in your distro like `gcc49` that provides the `g++-4.9` executable.
3. In a terminal, go to the folder `DualSPHysics/SOURCE/DualSPHysics/Source/`
4. Execute `make clean` to make sure the environment is clean and ready to compile
5. Execute `make CC=g++-4.9 CPP=g++-4.9 CXX=g++-4.9 LD=g++-4.9 -f ./Makefile`. Be sure to replace `g++-4.9` for the executable name you have in your system (previously installed in step 2)

After compiling you should see a message like `--- Compiled Release GPU/CPU version ---`. Go to

`DualSPHysics/EXECS_LINUX/` to check that `DualSPHyiscs_linux64` or `DualSPHyiscsCPU_linux64` is there and build correctly.

To exclude the use of OpenMP you have to remove the flags `-fopenmp` and `-lgomp` in the Makefile and comment line `#define _WITHOMP` in Types.h.

The user can modify the compilation options, the path of the CUDA toolkit directory, the GPU architecture. The GPU code is already compiled (EXECS) for compute capabilities sm20, sm30, sm35, sm37, sm50, sm52 and with CUDA v7.5.

# Alternative building method via CMAKE

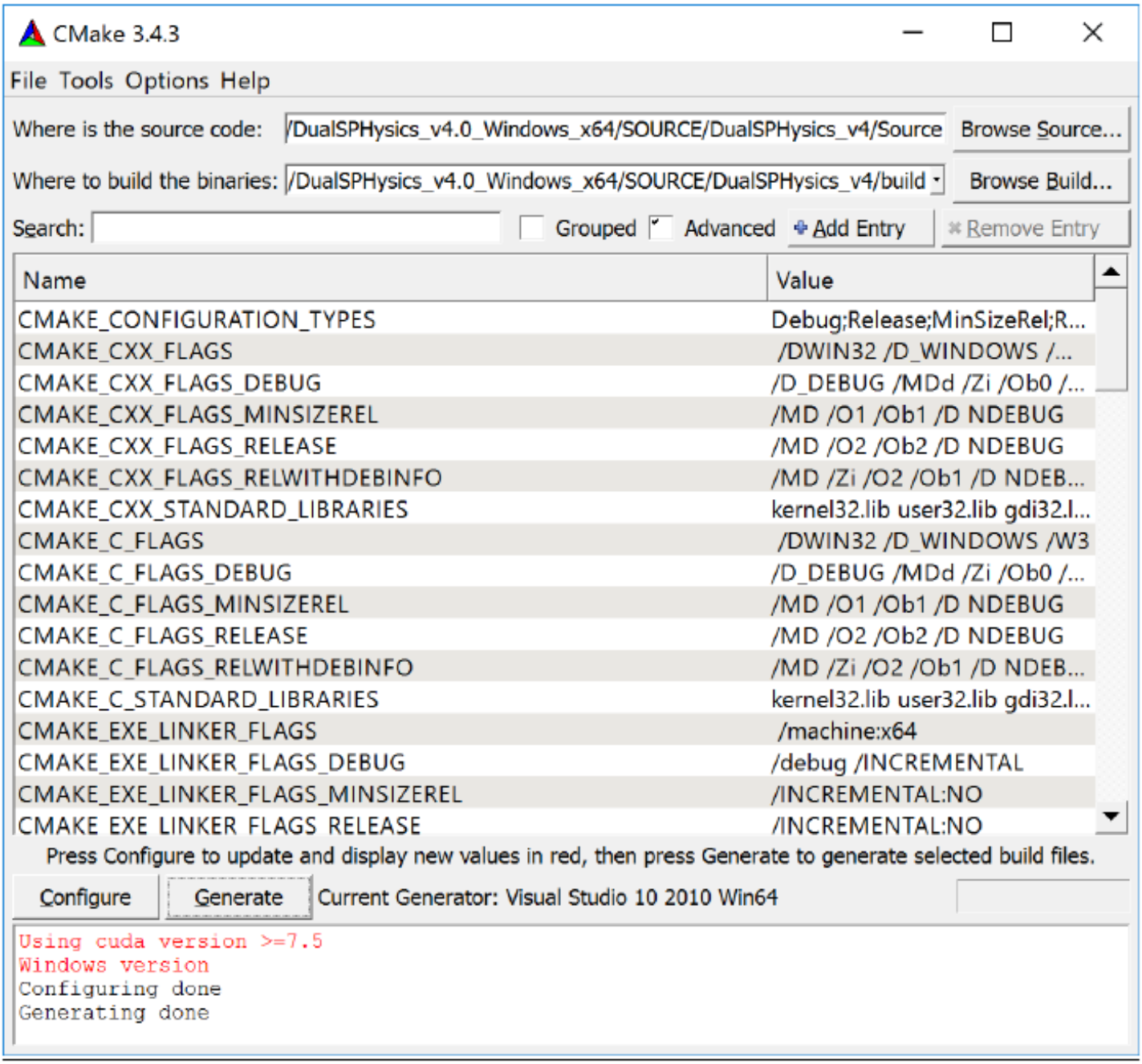
A new building method is supported in the new version 4.0 of DualSPHysics using CMAKE (https://cmake.org/). CMAKE is a cross-platform and an independent building system for compilation. This software generates native building files (like makefiles or Visual Studio projects) for any platform. The location of dependencies and the needed flags are automatically determined. Note that this method is on trial for version 4.

## Compile instructions for Microsoft Windows with Cmake

The building system needs the following dependencies:

- CMake version 2.8.10 or greater
- Nvidia CUDA Toolkit version 4.0 or greater.
- Visual Studio 2010 or 2013 version.
- File `CMakeLists.txt`

The folder build will be created in DUALSPHYSICS/SOURCE/DualSPHysics_4. This folder will contain the building files so it can be safely removed in case of rebuilding, it can actually be placed anywhere where the user has writing permissions. Afterwards, open the Cmake application, a new window will appear:




Paste the Source folder path in the textbox labeled as Where is the source code, and paste the build folder path into the Where to build the binaries textbox. Once the paths are introduced, the Configure button should be pressed. A new dialog will appear asking for the compiler to be used in the project. Please, remember that only Visual Studio 2010 and Visual Studio 2013 for 64bit are supported.

If the configuration succeeds, now press the Generate button. This will generate a Visual Studio project file into the build directory.

In order to compile both CPU and GPU versions, just change configuration to Release and compile. If the user only wants to compile one version, one can choose one of the solutions dualsphysics4cpu or dualsphysics4gpu for CPU or GPU versions respectively, and compile it.

The user can freely customize the Source/CMakeLists.txt file to add new source files or any other modifications. For more information about how to edit this file, please, refer to official Cmake documentation (https://cmake.org/documentation/).

## Compile instructions for Linux with Cmake

The building system needs the following dependencies:

- Cmake version 2.8.10 or greater.
- Nvidia CUDA Toolkit version 4.0 or greater.
- GNU G++ compiler 4.4 version or greater.
- File `CMakeLists.txt`

The folder build will be created in DUALSPHYSICS/SOURCE/DualSPHysics_4. This folder will contain the building files so it can be safely removed in case of rebuilding, it can actually be placed anywhere where the user has writing permissions. To create this folder and run Cmake just type:

```
> cd DUALSPHYSICS/SOURCE/DualSPHysics_4
> mkdir build
> cd build
> cmake ../Source
-- The C compiler identification is GNU 4.4.7
-- The CXX compiler identification is GNU 4.4.7
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
Using cuda version <7.5
Using libraries for gcc version <5.0
-- Try OpenMP C flag = [-fopenmp]
-- Performing Test OpenMP_FLAG_DETECTED
-- Performing Test OpenMP_FLAG_DETECTED - Success
-- Try OpenMP CXX flag = [-fopenmp]
-- Performing Test OpenMP_FLAG_DETECTED
-- Performing Test OpenMP_FLAG_DETECTED - Success
-- Found OpenMP: -fopenmp
-- Configuring done
-- Generating done
-- Build files have been written to: /home/user/DUALSPHYSICS/SOURCE/DualSPHysics_v4/build
```

The command cmake ../Source will search for a Cmake file (CMakeLists.txt) in the specified folder. As mentioned before, the user can freely customize this file to add new source files or any other modifications. For more information about how to edit this file, please, refer to Cmake official documentation (https://cmake.org/documentation/).

Once the cmake command runs without error, a Makefile can be found in the build folder. To build both CPU and GPU versions of DualSPHysics just type make. If the user only needs to build one of the executable files he can use the commands make dualsphysics4cpu or make dualsphysics4gpu for CPU and GPU versions respectively. In order to install the compiled binaries into the EXECS folder, the user can either copy the executable files or type the command make install.

# Format Files

The codes provided within the DualSPHysics package present some important improvements in comparison to the codes available within SPHysics. One of them is related to the format of the files that are used as input and output data throughout the execution of DualSPHysics and the pre-processing and post-processing codes. Different format files for the input and the output data are involved in the DualSPHysics execution: XML, binary and VTK-binary.

## XML File

The XML (EXtensible Markup Language) is a textual data format that can easily be read or written using any platform and operating system. It is based on a set of labels (tags) that organise the information and can be loaded or written easily using any standard text or dedicated XML editor. This format is used for input files for the code.

## BINARY File

The output data in the SPHysics code is written in text files, so ASCII format is used. ASCII files present some interesting advantages such as visibility and portability, however they also present important disadvantages particularly with simulations with large numbers of particles: data stored in text format consumes at least six times more memory than the same data stored in binary format, precision is reduced when values are converted from real numbers to text while reading and writing data in ASCII is more expensive (two orders of magnitude). Since DualSPHysics allows performing simulations with a high number of particles, a binary file format is necessary to avoid these problems. Binary format reduces the volume of the files and the time dedicated to generate them. These files contain the meaningful information of particle properties. In this way, some variables can be removed, e.g., the pressure is not stored since it can be calculated starting from the density using the equation of state. The mass values are constant for fluid particles and for boundaries so only two values are used instead of an array. Data for particles that leave the limits of the domain are stored in an independent file (PartOut_000.obi4) which leads to an additional saving. Hence, the advantages can be summarised as: (i) memory storage reduction, (ii) fast access, (iii) no precision lost and (iv) portability (i.e. to different architectures or different operating systems).

The file format used now in DualSPHysics v4.0 is named BINX4 (.bi4) which is the new binary format and can save particle position in single or double precision. This format file is a container so the user can add new metadata and new arrays can be processed in an automatic way using the current post-processing tools of the package.

## VTK File

VTK (Visualization ToolKit) files are used for final visualization of the results and can either be generated as a pre-processing step or output directly by DualSPHysics instead of the standard BINX format (albeit at the expense of computational overhead). VTK not only supports the particle positions, but also physical quantities that are obtained numerically for the particles involved in the simulations. VTK supports many data types, such as scalar, vector, tensor, texture, and also supports different algorithms such as polygon reduction, mesh smoothing, cutting, contouring and Delaunay triangulation. The VTK file format consists of a header that describes the data and includes any other useful information, the dataset structure with the geometry and topology of the dataset and its attributes. Here VTK files of POLYDATA type with legacy-binary format is used. This format is also easy for input-output (IO) or read-write operations.

# Preprocessing

A program named GenCase is included to define the initial configuration of the simulation, movement description of moving objects and the parameters of the execution in DualSPHysics. All this information is contained in a definition input file in XML format; Case_Def.xml. Two output files are created after running GenCase: Case.xml and Case.bi4 (the input files for DualSPHysics code). These input (red boxes) and output files (blue boxes) can be observed in Figure 9-1. Case.xml contains all the parameters of the system configuration and its execution such as key variables (smoothing length, reference density, gravity, coefficient to calculate pressure, speed of sound...), the number of particles in the system, movement definition of moving boundaries and properties of moving bodies. Case.bi4 contains the initial state of the particles (number of particles, position, velocity and density) in BINX4 (.bi4) format.

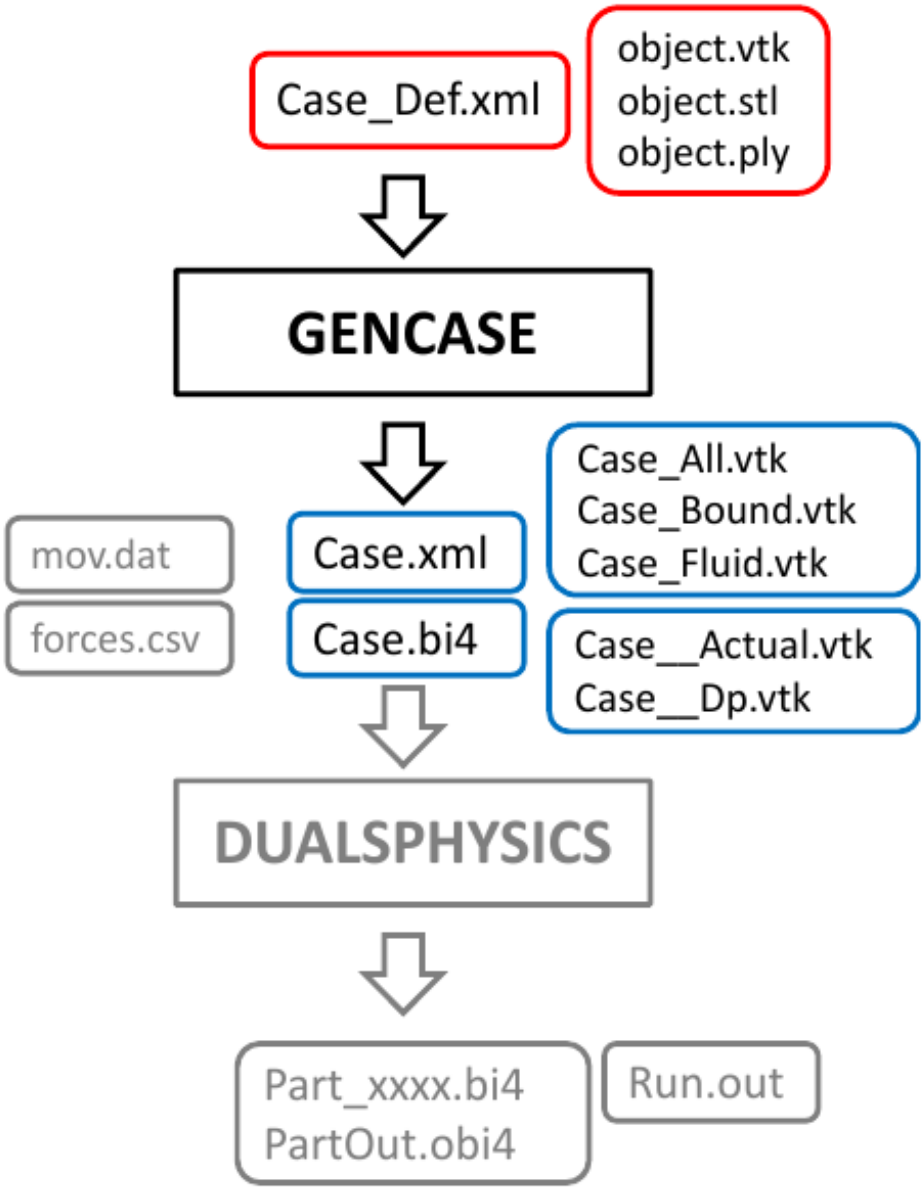



**Figure 9-1.** Input (red) and output (blue) files of GenCase code.

Particle geometries created with GenCase can be initially checked by visualising in Paraview the files Case_All.vtk, Case_Bound.vtk and Case_Fluid.vtk.

GenCase employs a 3-D Cartesian mesh to locate particles. The idea is to build any object using particles. These particles are created at the nodes of the 3-D Cartesian mesh. Firstly, the mesh nodes around the object are defined and then particles are created only in the nodes needed to draw the desired geometry. Figure 9-2

illustrates how this mesh is used; in this case a triangle is generated in 2D. First the nodes of a mesh are defined starting from the maximum dimensions of the desired triangle, then the edges of the triangle are defined and finally particles are created at the nodes of the Cartesian mesh which are inside the triangle.

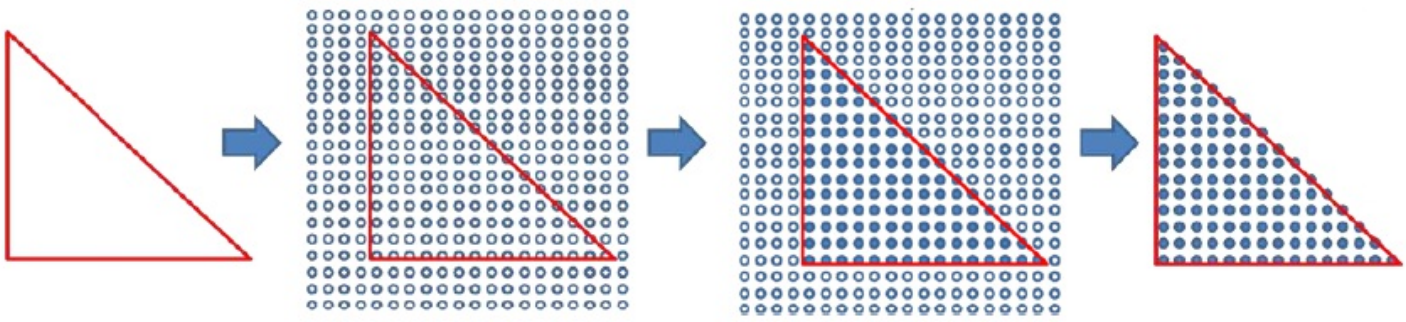


**Figure 9-2.** Generation of a 2-D triangle formed by particles using GenCase.

All particles are placed over a regular Cartesian grid. The geometry of the case is

defined independently to the inter-particle distance. This allows the discretization of each test case with a different number of particles simply by varying the resolution (or particle size) dp. Furthermore, GenCase is very fast and able to generate millions of particles only in seconds on the CPU.

Very complex geometries can be easily created since a wide variety of commands (labels in the XML file) are available to create different objects; points, lines, triangles, quadrilateral, polygons, pyramids, prisms, boxes, beaches, spheres, ellipsoids, cylinders, waves (`, , , , ,

, , , , ,

, , , , , , , `).

Once the mesh nodes that represent the desired object are selected, these points are stored as a matrix of nodes. The shape of the object can be transformed using a translation ( `<move />` ), a scaling ( `<scale />` ) or a rotation ( `<rotate />, <rotateline />` ). With the generation process creating particles at the nodes, different types of particles can be created; a fluid particle ( `<setmkfluid />` ), a boundary particle ( `<setmkbound />` ) or none ( `<setmkvoid />` ). Hence, mk is the marker value used to mark a set of particles with a common feature in the simulation. Note that values of the final mk are different to mkfluid, mkbound and mkvoid following the rules:

mk for boundaries = mkbound + 11

mk for fluid particles = mkfluid + 1

Particles can be created only at the object surface ( `<setdrawmode mode="face" />` ), or only inside the bounds of the objects ( `<setdrawmode mode="solid" />` ) or both ( `<setdrawmode mode="full" />` ).

The set of fluid particles can be labelled with features or special behaviours ( `<initials />` ). For example, initial velocity ( `<velocity />` ) can be imposed for fluid particles or a solitary wave can be defined ( `<velwave />` ). Furthermore, particles can be defined as part offloating object ( `<floatings />` ).

Once boundaries are defined, filling a region with fluid particles can be easily obtained using the following commands: ( `<fillpoint />, <fillbox />, <fillfigure />, <fillprism />` ). This works also in the presence of arbitrarily complex geometries.

In cases with more complex geometries, external objects can be imported from 3DS files (Figure 9-3) or CAD files

(Figure 9-4). This enables the use of realistic geometries generated by 3D designing application with the drawing commands of GenCase to be combined. These files (3DS or CAD) must be converted to STL format (`<drawfilestl />`), PLY format (`<drawfileply />`) or VTK format (`<drawfilevtk />`), formats that are easily loaded by GenCase. Any object in STL, PLY or VTK (object.vtk, object.stl or object.ply in Figure 9-1) can be split in different triangles and any triangle can be converted into particles using the GenCase code.

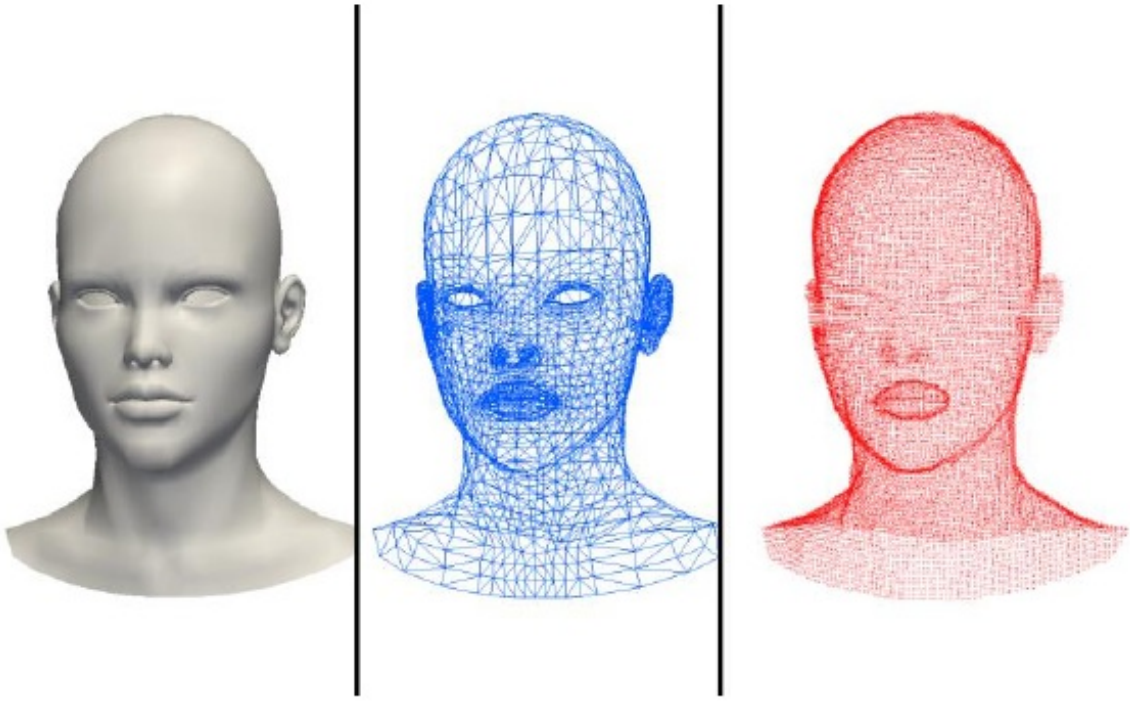


**Figure 9-3.** Example of a 3D file imported by GenCase and converted into particles.

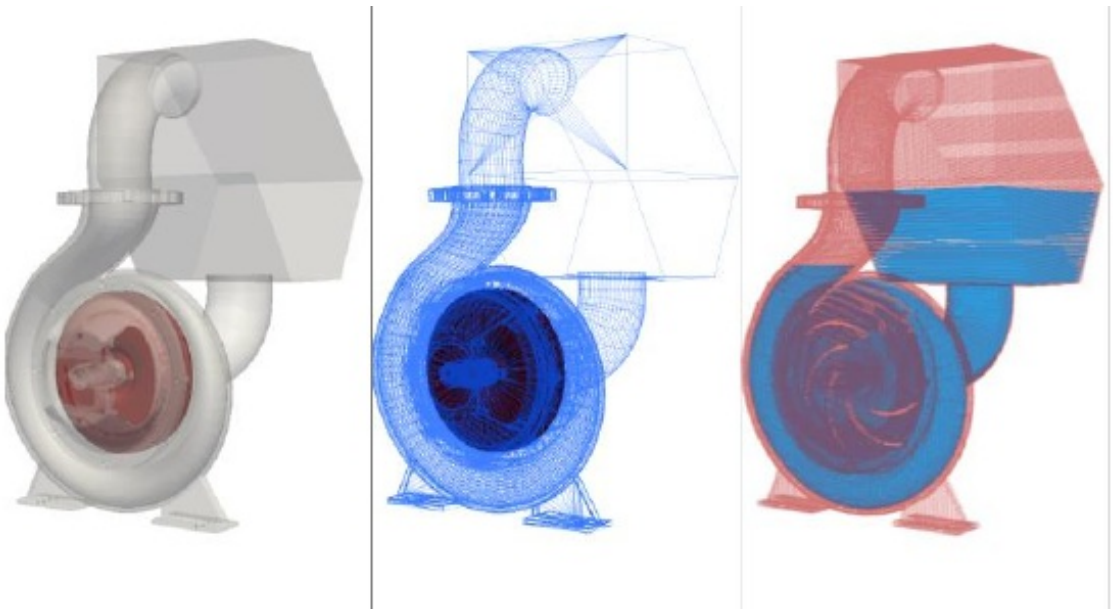


**Figure 9-4.** Example of a CAD file imported by GenCase and converted into particles.

Different kinds of movements can be imposed to a set of particles; linear, rotational, circular, sinusoidal, etc. To help users define movements, a directory with some examples is also included in the DualSPHysics package. Thus, the directory MOTION includes:

- Motion01: uniform rectilinear motion (`<mvrect />`) that also includes pauses (`<wait />`)
- Motion02: combination of two uniform rectilinear motion (`<mvrect />`)

- Motion03: movement of an object depending on the movement of another (hierarchy of objects)
- Motion04: accelerated rectilinear motion ( `<mvrectace />` )
- Motion05: rotational motion ( `<mvrot />` ). See Figure 9-5.
- Motion06: accelerated rotation motion ( `<mvrotace />` ) and accelerated circular motion ( `<mvcirace />` ). See Figure 9-6.
- Motion07: sinusoidal movement ( `<mvrectsinu />, <mvrotsinu />, <mvcirsinu />` )
- Motion08: prescribed with data from an external file (time , position) ( `<mvfile/>` )
- Motion09: prescribed with data from an external file (time , angle) ( `<mvrotfile />` )

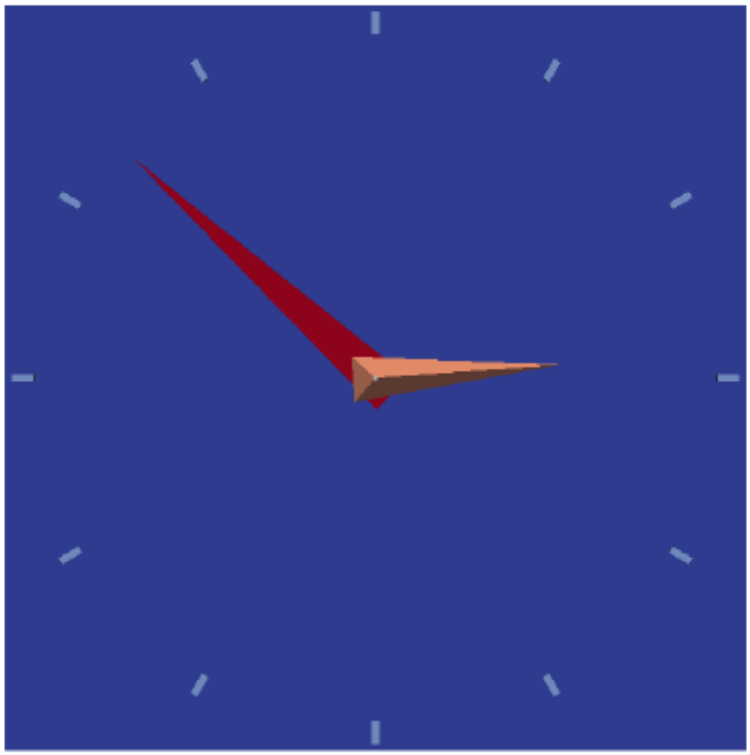


**Figure 9-5.** Example of rotational motion.

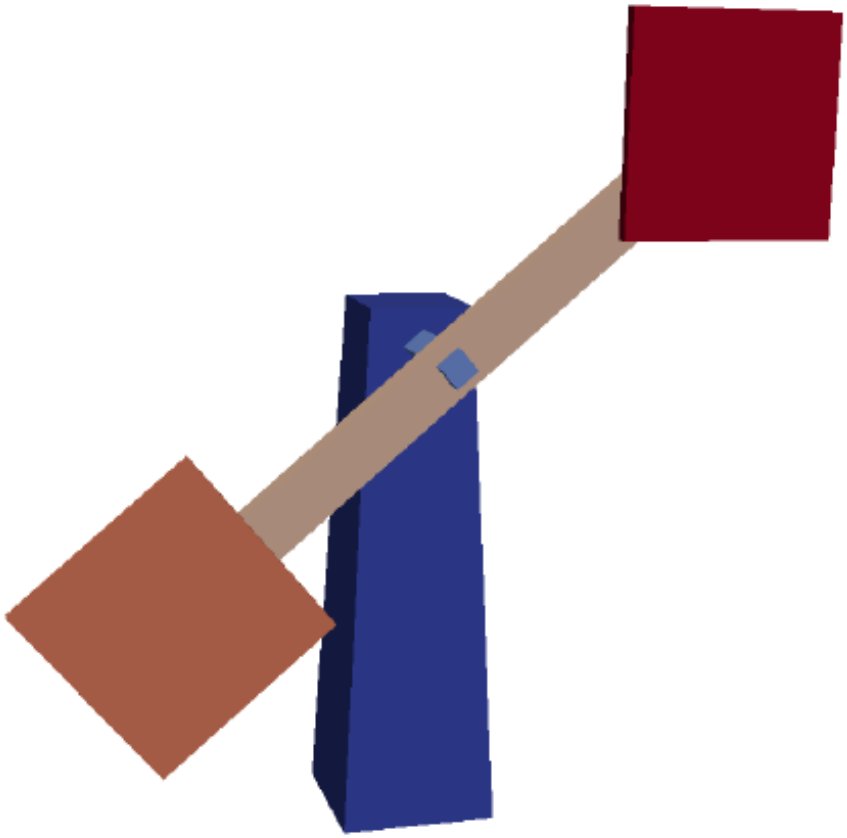


**Figure 9-6.** Example of accelerated rotation motion and accelerated circular motion.

# Processing

The main code which performs the SPH simulation is named DualSPHysics.

The input files to run DualSPHysics code include one XML file (Case.xml in Figure 10-1) and a binary file (Case.bi4 in Figure 10-1). Case.xml contains all the parameters of the system configuration and its execution such as key variables (smoothing length, reference density, gravity, coefficients to compute pressure starting from density, speed of sound...), the number of particles in the system, movement definition of moving boundaries and properties of moving bodies. The binary file Case.bi4 contains the particle data; arrays of position, velocity and density and headers. The output files consist of binary format files with the particle information at different instants of the simulation (Part0000.bi4, Part0001.bi4, Part0002.bi4 ...) file with excluded particles (PartOut.obi4) and text file with execution log (Run.out).

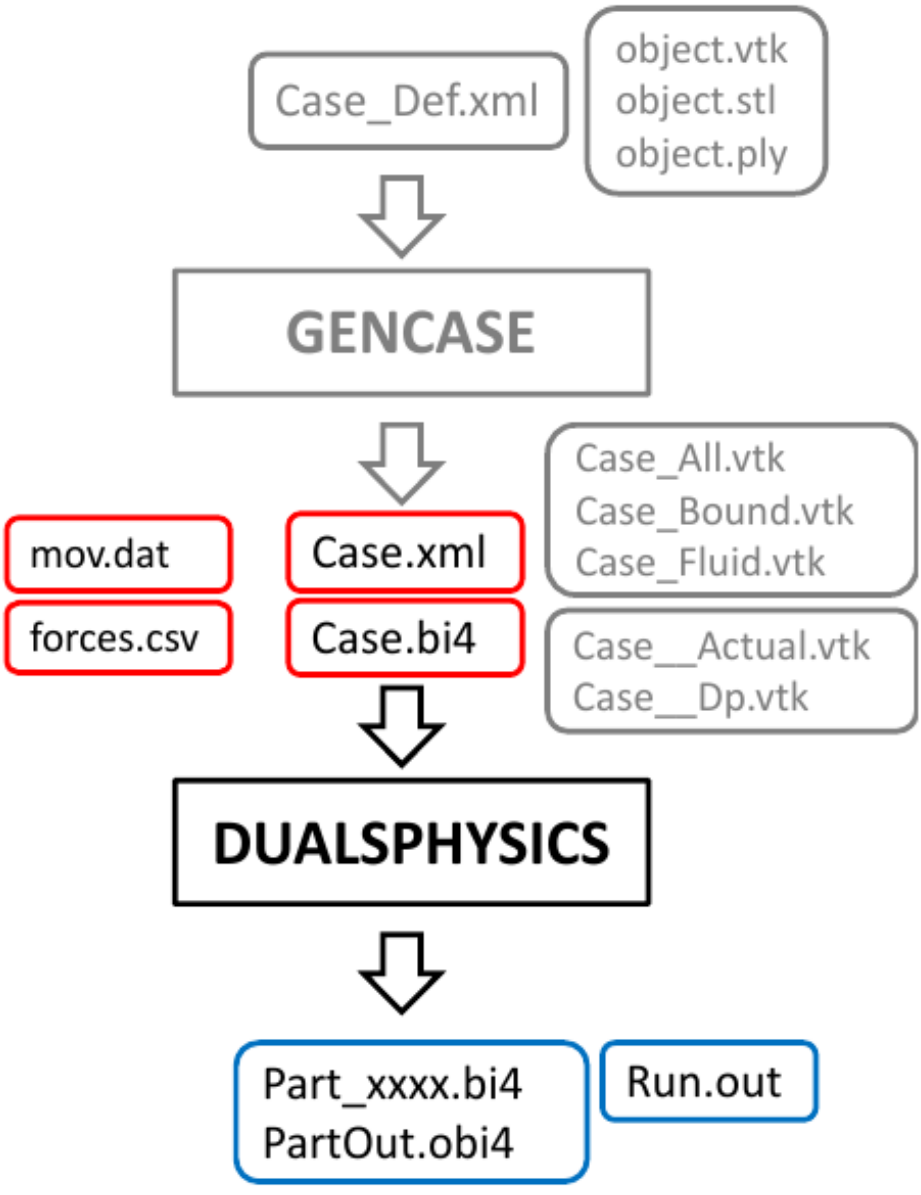



**Figure 10-1.** Input (red) and output (blue) files of DualSPHysics code.

Different parameters defined in the XML file can be be changed using executions parameters of DualSPHysics: time stepping algorithm specifying Symplectic or Verlet (-symplectic, -verlet[:steps]) , choice of kernel function which can be Cubic or Wendland (- cubic, -wendland), the value for artificial viscosity (-viscoart: ) or laminar+SPS viscosity treatment (-viscolamsps:), activation of the Delta-SPH formulation (- deltasph: ), use of shifting algorithm (-shifting: ) the maximum time of simulation and time intervals to save the output data (-tmax:, -tout:). To run the

code, it is also necessary to specify whether the simulation is going to run in CPU or GPU mode (- cpu, -gpu[:id]), the format of the output files ( -sv:[formats,...], none, binx, ascii, vtk, csv), that summarises the execution process (-svres:<0/1>) with the computational time of each individual process (-svtimers:<0/1>). It is also possible to exclude particles as being out of limits according to prescribed minimum and maximum values of density (-rhopout:min:max) or that travel further than maximum Z position (-incz:).

For CPU executions, a multi-core implementation using OpenMP enables executions in parallel using the different cores of the machine. It takes the maximum number of cores of the device by default or users can specify it ( -ompthreads:). On the other hand, different cell divisions of the domain can be used (-cellmode: ) that differ in memory usage and efficiency.

One of the novelties version 4 of DualSPHysics is the use of double precision in variables of position of the particles (-posdouble:) for the computation of particle interactions. The particle interaction is one of the most time-consuming parts of the simulation, hence the precision in this part can be controlled using the -posdouble parameter, which takes the following values:

- 0: particle interaction is performed using single precision for position variables (x, y, z) When “dp” is much smaller than size of the domain, the user is recommended to choose one of the following:
- 1: particle interaction is performed using double precision for position variables but final position is stored using simple precision
- 2: particle interaction is performed using double precision for position variables and final position is stored using double precision.

Other important novelty in v4.0 is the determination of the optimum BlockSize for the CUDA kernels that execute particle interaction (-blocksize:):

- Fixed (-blocksize:0): A fixed block size of 128 threads is used. This value does not always provides the maximum performance but it usually offers good performance for those type of kernels.
- Occupancy (-blocksize:1): Occupancy Calculator of CUDA is used to determine the optimum block size according to the features of the kernel (registers and shared memory however data used in the kernels are not considered). This option is available from CUDA 6.5
- Empirical (-blocksize:2): Here, data used in the CUDA kernels is also considered. The optimum BlockSize is evaluated every certain number of steps (500 by default). In this way, block size can change during the simulation according to input data.

# Postprocessing

## Visualization of particle output data

The PartVTK code is used to convert the output binary files of DualSPHysics into different formats that can be visualised and /or analysed. Thus, the output files of DualSPHysics, the binary files (.bi4), are now the input files for the post-processing code PartVTK.

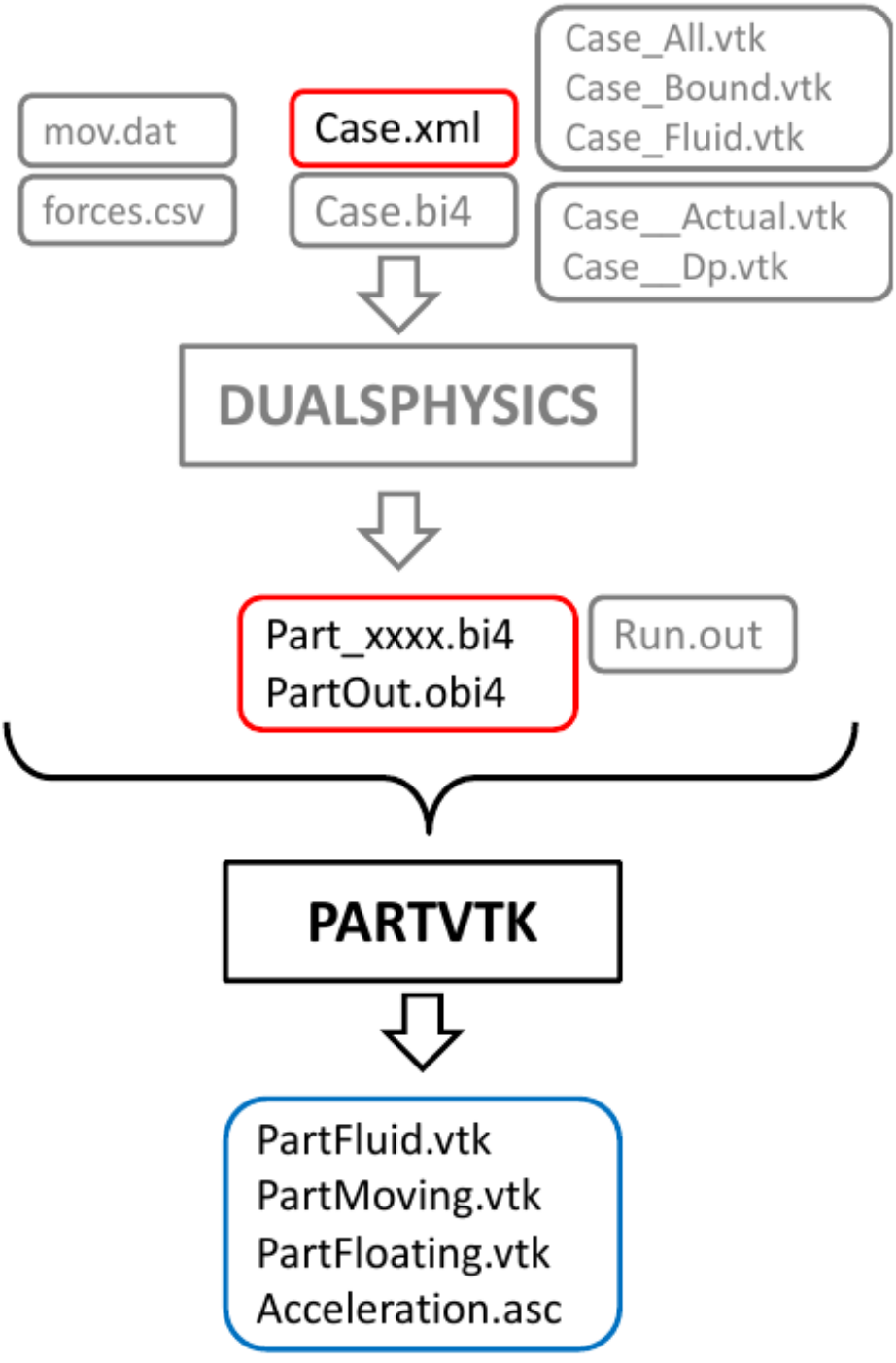



**Figure 11-1.** Input (red) and output (blue) files of PartVTK code.

The output files can be VTK-binary (-savevtk), CSV (-savecsv) or ASCII (-saveascii). In this way the results of the simulation can be plotted using Paraview, gnuplot, Octave, etc.... For example; PartVtkBin_0000.vtk,... These files can be generated by selecting a set of particles defined by mk (-onlymk:), by the id of the particles (-onlyid:), by the type of the particle (-onlytype:), by the position of the particles (-onlypos: and -onlyposfile) or by the limits of velocity of the particles (-onlyvel:), so we can check or uncheck all the particles (+/-all), the boundaries (+/-

bound), the fixed boundaries (+/-fixed), the moving boundaries (+/-moving), the floating bodies (+/-floating) or the fluid particles (+/-fluid). The output files can contain different particle data (-vars:); all the physical quantities (+/-all), velocity ( +/-vel), density ( +/-rhop), pressure ( +/-press), mass (+/-mass), volume (+/-vol), acceleration (+/-ace), vorticity ( +/-vor), the id of the particle (+/-idp), the mk of the particle (+/-mk) and the type (+/-type:). The user can define new variables in DualSPHysics and make reference to those in PartVTK using -vars:NewVar or -vars:all.

In addition, the PartVTKOut code is used to generate files with the particles that were excluded from the simulation (stored in PartOut.obi4). The output file of DualSPHysics, PartOut.obi4 is the input file for the post-processing code PartVTKOut. Information with excluded particles can be stored in CSV files ( -savecsv: -SaveResume) and VTK (-savevtk:) can be generated with those particles.

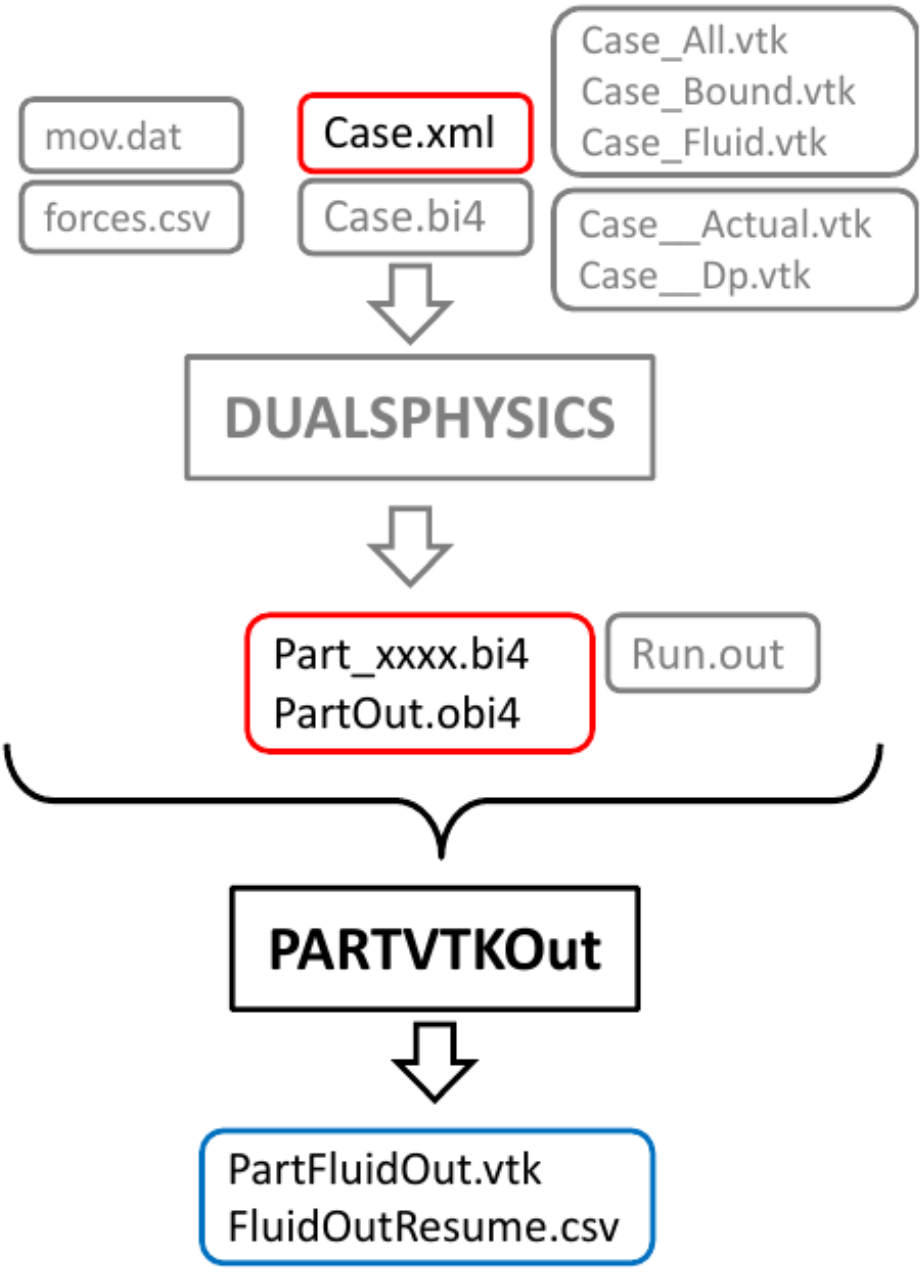



**Figure 11-2.** Input (red) and output (blue) files of PartVTKOut code.

Particles can be excluded from the simulation for three reasons:

- Position: Limits of the domain are computed starting from particles that were created in GenCase. Note that these limits are different from pointmin and pointmax defined in section of the input XML and can be also

changed by the user when executing DualSPHysics code. The actual limits of the domain can be seen in Run.out: MapRealPos(final). Therefore, when one particle moves beyond those limits, the particle is excluded. Only in the Z+ direction, can particles move to higher positions according to parameter IncZ., where new cells are created to contain the particles.

- Position: Valid values of particle density are between RhopOutMin (default=700) and RhopOutMax (default=1300), but the user can also change those values.
- Velocity: One particle can be also removed from the system when its displacement exceeds 0.9*Scell during one time step (Scell is the size of the cell).

# Visualization of boundaries

In order to visualise the boundary shapes formed by the boundary particles, different geometry files can be generated using the BoundaryVTK code. The code creates triangles or planes to represent the boundaries.

As input data, shapes can be loaded from a VTK file (-loadvtk), a PLY file (-loadply) or an STL file (-loadstl) while boundary movement can be imported from an XML file (- loadxml file.xml) using the timing of the simulation (-motiontime) or with the exact instants of output data (-motiondatatime ). The movement of the boundaries can also be determined starting from the particle positions ( -motiondata). The output files consist of VTK files (-savevtk), PLY files (-saveply) or STL files (-savestl) with the loaded information and the moving boundary positions at different instants. For example the output files can be named motion_0000.vtk, motion_0001.vtk, motion_0002.vtk... These files can be also generated by only a selected object defined by mk (-onlymk:), by the id of the object (-onlyid:).

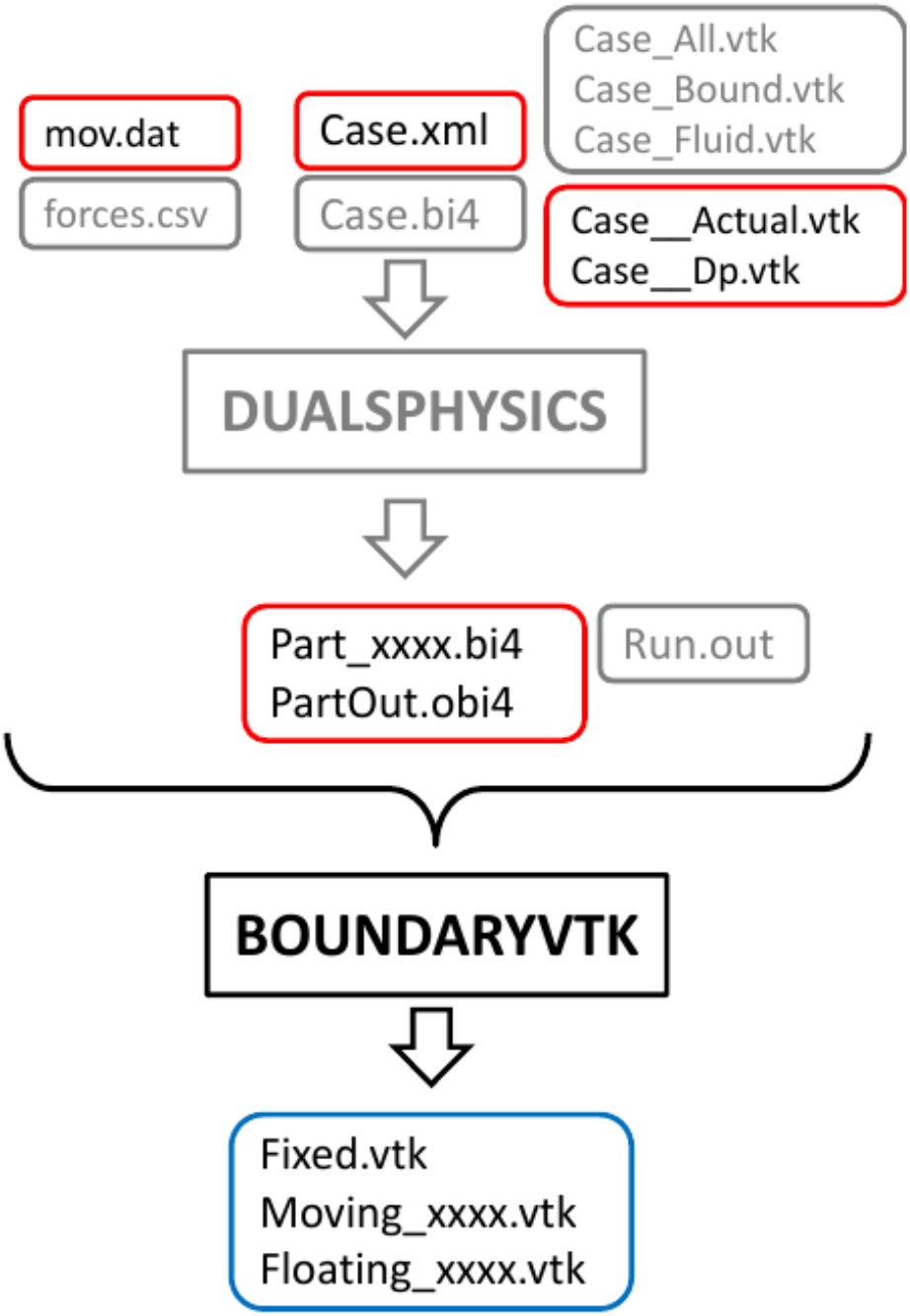



**Figure 11-3.** Input (red) and output (blue) files of BoundaryVTK code.

## Analysis of numerical measurements

To compare experimental and numerical values, a tool to analyse these numerical measurements is needed. The MeasureTool code allows different physical quantities at a set of given points to be computed. The binary files (.bi4) generated by DualSPHysics are the input files of the MeasureTool code and the output files are again VTK-binary or CSV or ASCII. The numerical values are computed by means of an SPH interpolation of the values of the neighbouring particles around a given position.

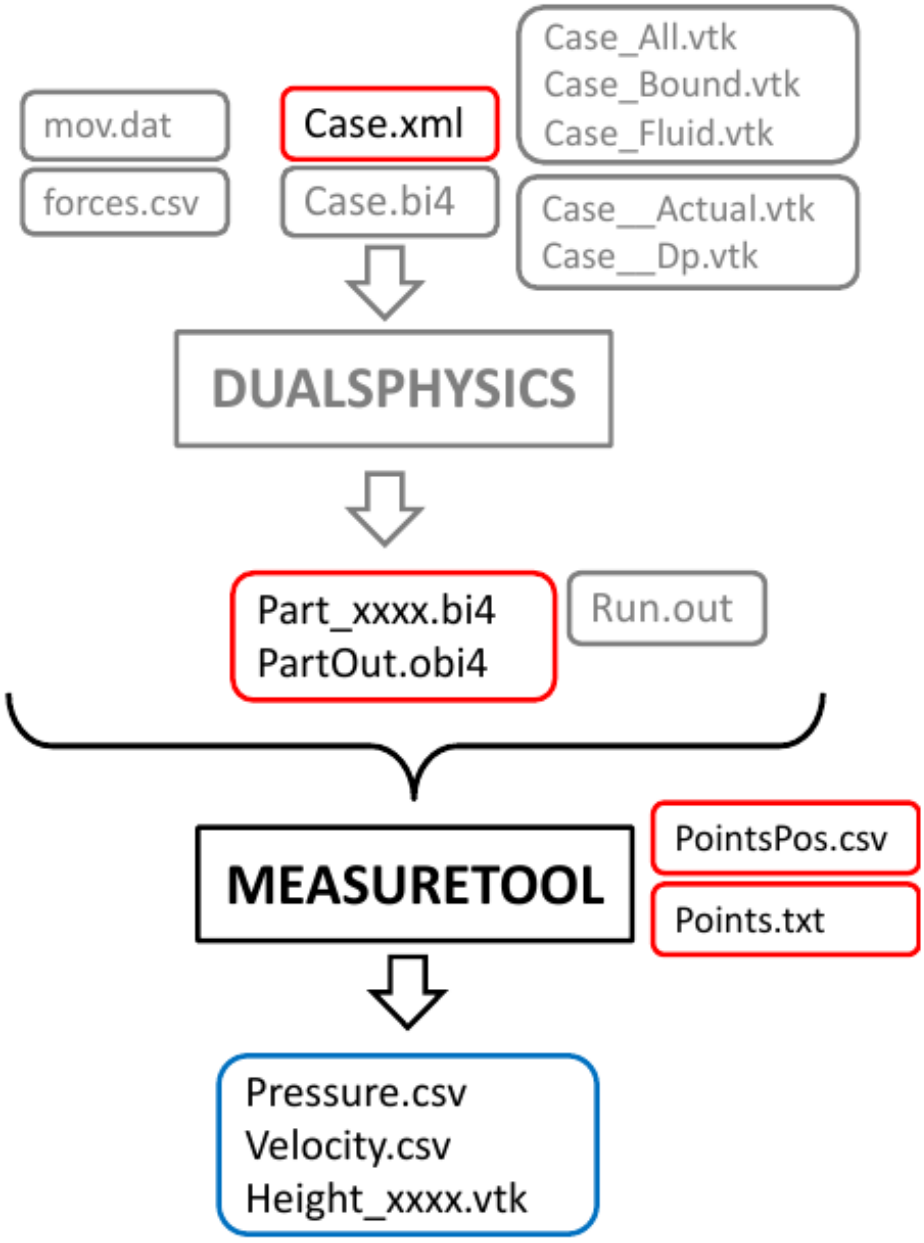



**Figure 11-4.** Input (red) and output (blue) files of MeasureTool code.

The interpolation is computed using the Wendland kernel. Kernel correction is also applied when the summation of the kernel values around the position is higher than a value (-kclimit:) defining a dummy value if the correction is not applied ( -kcdummy:). The positions where the interpolation is performed are given in a text file for fixed position (-points ) or with position that changes in time (-pointspos ). Variables can be also computed at the position of existing particles with a given mk ( -particlesmk:) or by indicating their id (-particlesid: ). The distance of interpolation can be 2h (the size of the kernel) or can be changed (-distinter_2h:, -distinter:). The interpolation is carried out using a selected set of particles, so the same commands for PartVTK can be used ( - onlymk: , -onlyid:, -onlytype:, -onlypos:, -onlyposfile). Different interpolated variables (-vars) can be numerically calculated; all available ones (+/-all), velocity ( +/-vel), density (+/- rhop), pressure (+/-press), mass (+/-mass), volume (+/-vol), id (+/-idp), vorticity ( +/-vor), acceleration (+/-ace) the summation of the kernel multiplied by volume ( +/-kcorr), and variables defined by the user (+/-XXX). The maximum water depth can be also computed. Height values (-height:) are calculated according to the interpolated mass, if the nodal mass is higher than a given reference mass, that Z-position will be considered as the maximum height. The reference value can be calculated in relation to the mass values of the selected particles (-height:0.5 , half the mass by default in 3D and -height:0.4 in 2D) or can be given in an absolute way (-heightlimit:).

# Force computation

The ComputeForces code is employed to compute the force exerted by the fluid onto a boundary object. The value of force is calculated as the summation of the acceleration values (solving the momentum equation) multiplied by the mass of each boundary particle (see also PostprocessingCalculations.pdf).

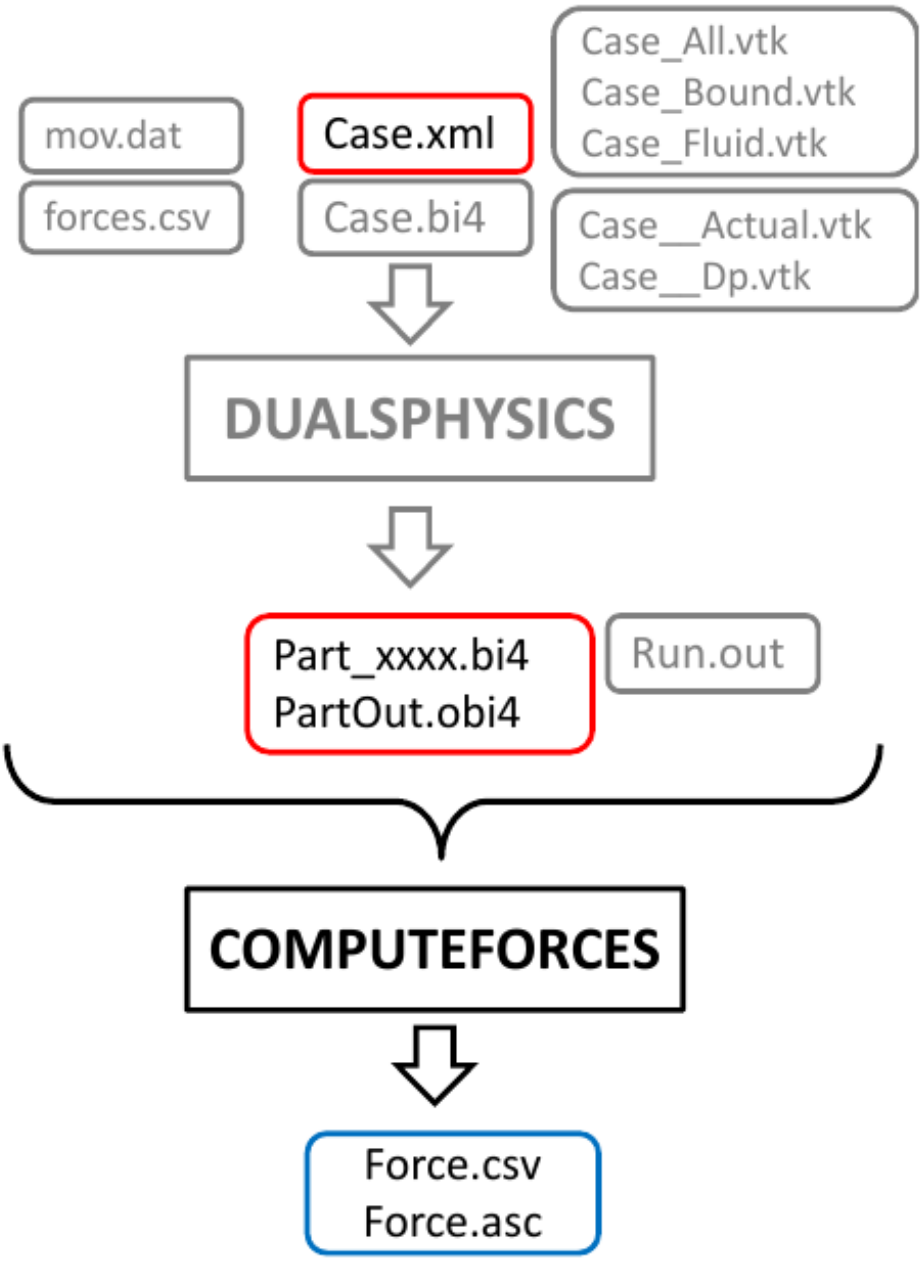
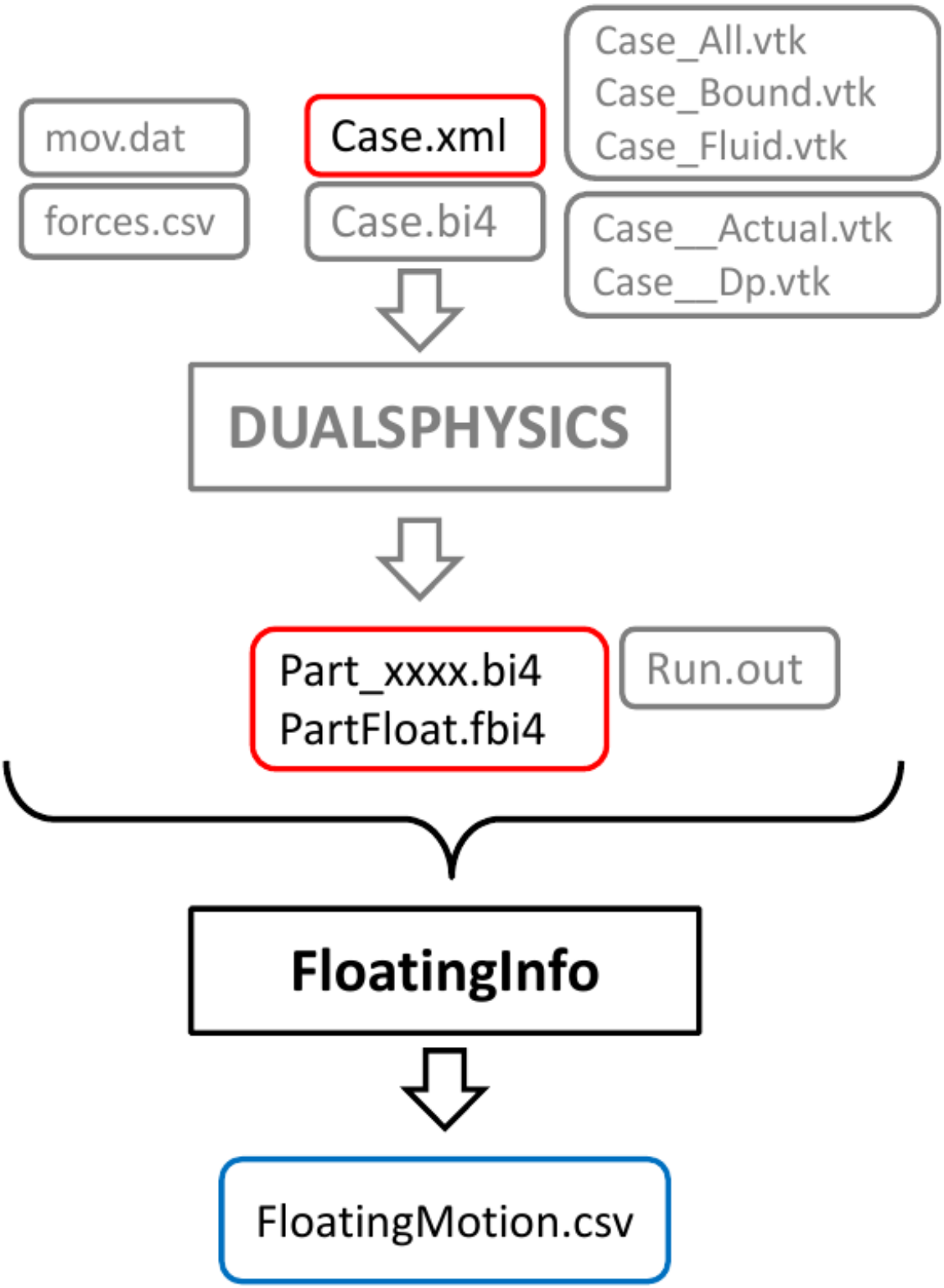



**Figure 11-5.** Input (red) and output (blue) files of ComputeForces code.

The momentum equation to solve the acceleration values is computed using the Wendland kernel. The distance of interpolation can be 2h (the size of the kernel) or can be changed (-distinter_2h:, -distinter:). The interpolation is carried out using a selected set of particles, so the same commands for PartVTK can be used (-onlymk:, -onlyid:, - onlypos:). The output files can be VTK-binary (-savevtk ), CSV (-savecsv) or ASCII (- saveascii).

# Analysis of floating data

The FloatingInfo code is employed to obtain different data of the floating objects such as linear velocity, angular velocity, displacement of the center, motions and angles of rotation. The binary files (.bi4) generated by DualSPHysics and the file PartFloat.fbi4 are the input files of the FloatingInfo code and the output files are CSV files.




By default, the code always saves time, fvel (linear velocity), fomega (angular velocity), center. Motions and rotation can be also obtained (-savemotion) in 2D (surge, heave, roll) and in 3D (surge, sway, heave, roll, pitch, yaw). A CSV file will be created with the information of each floating in the simulation, but user can choose a given floating object (-onlymk:) or the name of the file (-savedata).

## Surface representation

Using a large number of particles, the visualization of the simulation can be improved by representing surfaces instead of particles. To create the surfaces, the marching cubes algorithm is used [Lorensen and Cline, 1987].

This computer graphics technique extracts a polygonal mesh (set of triangles) of an isosurface from a 3-D scalar field.

Figure 11-6, represents a 3-D dam-break simulation using 300,000 particles. The first snapshot shows the particle representation. Values of mass are interpolated at the nodes of a 3-D Cartesian mesh that covers the entire domain using an SPH interpolation. Thus a 3-D mesh vertex that belongs to the free surface can be identified. The triangles of this surface (generated by means of the marching cubes algorithm) are represented in the second frame of the figure. The last snapshots correspond to the surface representation where the colour corresponds to the interpolated velocity value of the triangles.

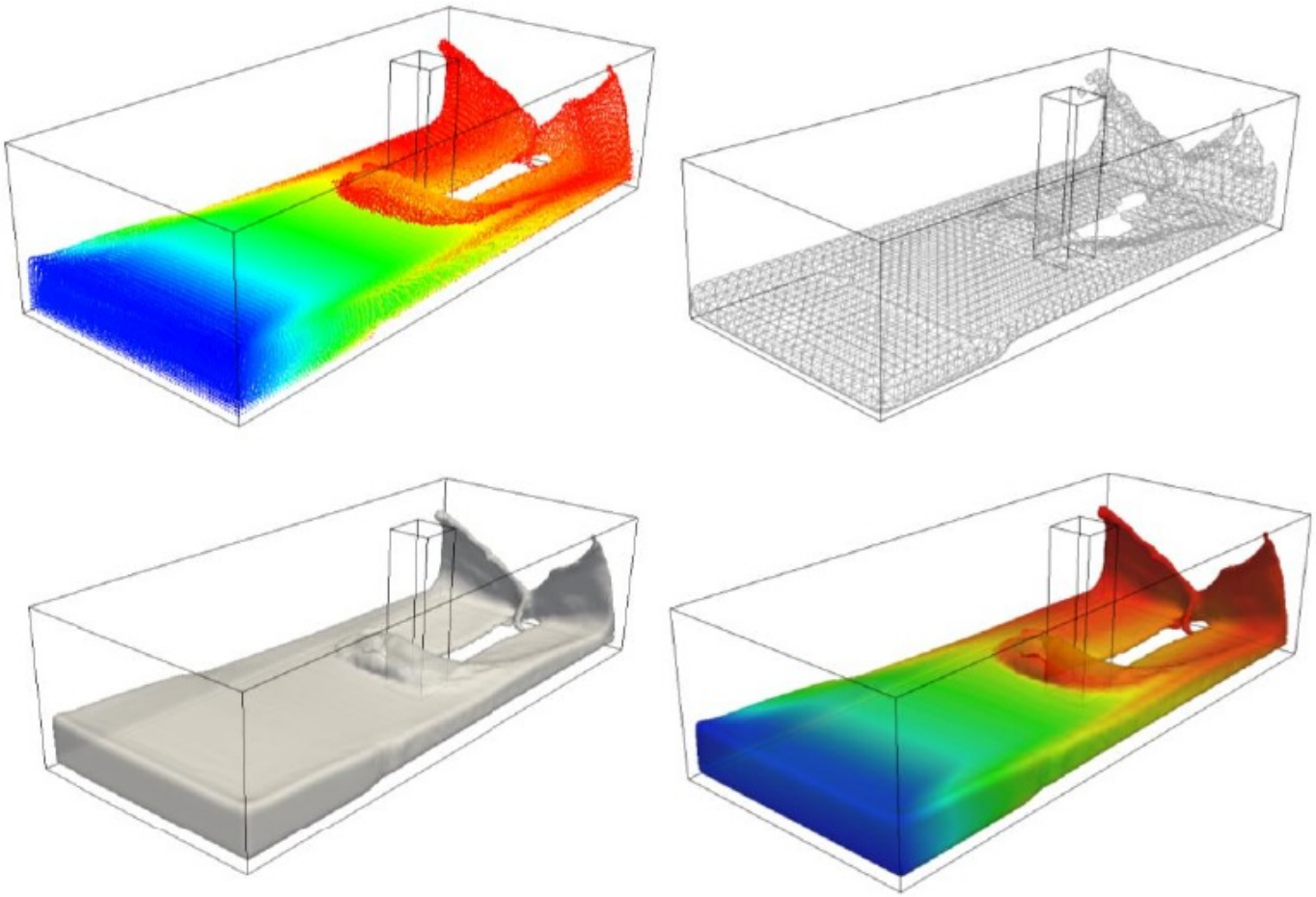


**Figure 11-6.** Conversion of points to surfaces.

The output binary files of DualSPHysics are the input files of the IsoSurface code and the output files are VTK files (-saveiso[:*]) with the isosurfaces calculated using a variable ( ) or can be structured points (-savegrid) with data obtained after the interpolation. The Cartesian mesh size can be defined by specifying the discretisation size using either the particle size dp (-distnode_dp: ) or by specifying an absolute internode distance (-distnode:). On the other hand, the maximum distance for the interaction between particles to interpolate values on the nodes can be also defined depending on 2h (-distinter_dp:) or in an absolute way ( -distinter:). The particles to be considered to create the isosurface can be defined by the user using positions (-onlypos & -onlyposfile) or the limits of the isosurface can be indicated (-iso_limits:)*

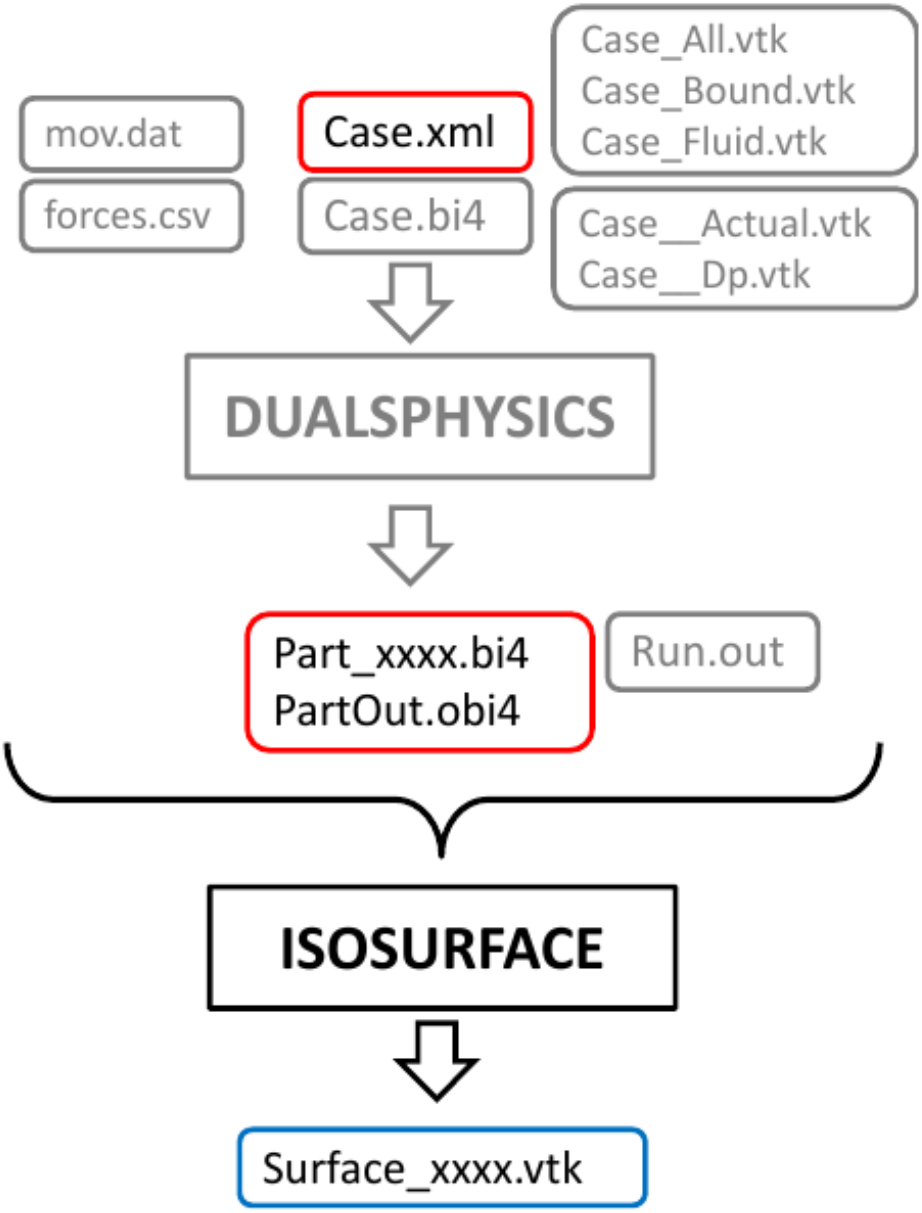



**Figure 11-7.** Input (red) and output (blue) files of IsoSurface code.

# FAQ

# What do I need to use DualSPHysics? What are the hardware and software requirements?\

DualSPHysics can be executed either on CPU or GPU. In order to use DualSPHysics code on a GPU, you need a CUDA-enabled Nvidia GPU card on your machine (http://developer.nvidia.com/cuda-gpus).

If you want to run GPU simulations (i.e. not develop the source code) the latest version of the driver for your graphics card must be installed. If no source code development is required, there is no need to install any compiler to run the binary of the code, only the driver must be updated. If you also want to compile the code you must install the nvcc compiler and a C++ compiler. The nvcc compiler is included in the CUDA Toolkit that can be downloaded from the Nvidia website and must be installed on your machine.

## Why DualSPHysics binary is not running?

If you are trying to run the executable GPU version on a CUDA-enabled Nvidia GPU card, the error message: Exception (JSphGpuSingle::SelecDevice) Text: Failed getting devices info. (CUDA error: CUDA driver version is insufficient for CUDA runtime version) can be solved by installing the latest version of the driver for the GPU card.

## How can I compile the code with different environments/compilers?

The provided source files in this release can be compiled for linux using a 'makefile' along with gcc and nvcc compilers, and for windows using a project for Visual Studio (both VS2010 and VS2013 are provided). In case you use another compiler or other environment, you can adjust the contents of the makefile or you can also use CMAKE.

Please refer to section 6 on Compiling DualSPHysics

## How many particles can I simulate with the GPU code?

The amount of particles that can be simulated depends on (i) the memory space of the GPU card and (ii) the options of the simulation.

## How should I start looking at the source code?

Section DualSPHysics open-source code of the guide introduces the source files including some call graphs for a better understanding and it is also highly recommended that you read the documentation generated with Doxygen.

## How can I create my own geometry?

Users can follow the provided example cases in the package. Those input XML files can be modified following XML_GUIDE_v4.0.pdf. Different input formats of real geometries can be converted using ExternalModelsConversion_GUIDE.pdf. This manuscript also describes in detail the input files of the different test cases.

# How can I contribute to the project?

You can contribute to the DualSPHysics project by reporting bugs, suggesting new improvements, citing DualSPHysics [See the answer to Question 24] in your paper if you use it, submitting your modified codes together with examples.

To read more info on that, please refer to the repository main page

# How can I modify the code of the precompiled libraries?

Some code is provided in precompiled libraries to reduce the number of source files and to facilitate the comprehension of only the SPH algorithms by the users. These precompiled code covers secondary aspects during SPH simulations that are only used in specific simulations so they are not used in most of the cases. If the user wants to modify some of the codes included in the precompiled libraries, he can just replace that library by his own implementation.

# How does the code define the limits of the domain?

In the input XML file, the parameters pointmin and pointmax only define the domain to create particles beyond these limits fluid or boundary particles will not be created. The limits of the computational domain are computed at the beginning of the DualSPHysics simulation and use the initial minimum and maximum positions of the particles that were already created with GenCase. In order to modify the limits automatically computed by DualSPHysics, different execution parameters can be used: -domain_particles[:xmin,ymin,zmin,xmax,ymax,zmax] -domain_particles_prc:xmin,ymin,zmin,xmax,ymax,zmax -domain_fixed:xmin,ymin,zmin,xmax,ymax,zmax so that the limits can be specified instead of using initial particle positions.

# How can I define the movement of boundaries?

Examples of the different type of movements that can be described with DualSPHysics are addressed in directory MOTION. Different kind of movements can be defined such as rectilinear, rotational, sinusoidal or circular motion and they can be uniform or accelerated, with pauses or with hierarchy of movements. And a final option is to load the movement from an external file with a prescribed movement (info of time, X- position, Y-position, Z-position and velocities) or with rotational movement (info of time, angle) that will be interpolated at each time step during the simulation (see CaseSloshingMotion).

# How can I include a new type of movement?

A new movement can be always defined by using an external file with mvfile or mvrotfile where the desired movement can be computed in advance and loaded from that file. However if a user wants to create the new type of movement in the source code, this should be implemented in the functions JSphCpu::RunMotion() for CPU or JSphGpu::RunMotion() for GPU, since the code implemented now is in the library JSphMotion.

# How do I prevent the boundary particles from going outside of the domain limits when applying motion?

As explained in the previous question, the limits of the computational domain are computed starting from the initial minimum and maximum positions of the particles. Since these values use the initial configuration, any

movement of boundaries that implies positions beyond these limits will give us the error 'boundary particles out the domain'. The solutions to solve this problem and to avoid creating larger tanks or domains are:

- defining boundary points `<drawpoint>` at the minimum and maximum positions that the particles are expected to reach during the simulation. The option `<drawpoint>` will create a particle at the given location x,y,z.
- using the parameters of DualSPHysics execution mentioned in the answer to Question 9.

## Why do I observe a gap between boundaries, floating bodies and fluid in the solution?

The gap is a result of pressure overestimation across density discontinuities. It is inherent to the boundary formulation used in DualSPHysics. The forces exerted by the boundary particles create a small gap between them and fluid particles (1.5 times h). Note that by increasing the resolution (i.e. using a smaller "dp") the gap is reduced however new boundary conditions are being developed and should be available in future releases [Domínguez et al., 2015].

## How do I prevent the fluid particles from penetrating inside floating bodies?

Validation of floating using experimental data from [Hadzic et al., 2005], as shown in http://dual.sphysics.org/index.php/validation/wavesfloatings/, has been performed only for floating objects created with `<setdrawmode mode="full" />` or `<setdrawmode mode="solid" />`, therefore the use of these options is suggested. In this way, floating particles are created in the faces of the object and inside the object so no particle penetration will be observed.

## How do I numerically compute the motion and rotations of floating bodies?

The new code FloatingInfo allows to obtain different variables of interest of the floating objects during the simulation; positions of the center, linear velocity, angular velocity, motions (surge, sway and heave) and angles of rotation (pitch, roll and yaw). As shown in http://dual.sphysics.org/index.php/validation/wavesfloatings/, the study of a floating body subjected to a wave packet is validated with data from [Hadzic et al., 2005].

## When are fluid particles excluded from the simulation?

Fluid particles are excluded during the simulation: (i) if their positions are outside the limits of the domain, (ii) when density values are out of a range (700-1300 by default), (iii) when particles moves beyond 0.9 times the cell size during one time step.

## How do I create a 2-D simulation?

DualSPHysics can also perform 2-D simulations. To generate a 2-D configuration you only have to change the XML file; imposing the same values in the Y-direction that

define the limits of the domain where particles are created (pointmin.y=1 and pointmax.y=1).

## How can I solve the error “Constant 'b' cannot be zero”?

Constant ‘b’ appears in the equation of state (Eq. 14). Constant ‘b’ is zero when fluid height (hswl) is zero (or fluid particles were not created) or if gravity is zero. First, you should check that fluid particles have been created. Possible errors can appear in 2-D simulations when the seed point of the option `<fillbox>` is not located at the correct y-position. Other solution is to specify the value of ‘hswl’ in `<constantsdef>` `<hswl value="0.8" auto="false" />`). When using gravity zero, the value of ‘b’ needs to be specified in `<constantsdef>` (`<b value="1.1200e+05" auto="false" />`) as occurs in CaseMovingSquare.

## How can I create a case without gravity?

When using gravity zero, the value of ‘b’ needs to be specified in `<constantsdef>` (`<b value="1.1200e+05" auto="false" />`) as occurs in CaseMovingSquare. Otherwise, ‘b’ is zero and gives the error shown in Question 18.

## How can I define the speed of sound?

By default, the speed of sound (speedofsound=coefsound*speedsystem) is calculated as 10 · sqrt(g*hswl). In order to calculate a more suitable ‘speedofsound‘ for a particular case requires the user to set the parameters ‘coefsound‘ and ‘speedsystem‘.

## What is the recommended alpha value in artificial viscosity?

The value of α=0.01 has proven to give the best results in the validation of wave flumes to study wave propagation and wave loadings exerted onto coastal structures [Altomare et al., 2015a; 2015c]. However in the simulation of other cases such as dam-breaks, the interaction between fluid and boundaries during dam propagation becomes more relevant and the value of α should be changed according to the resolution (“dp”) to obtain accurate results.

## How can I define new properties of the particles?

The file format XML offers several resources to define new general parameters or specific properties for different type of particles. In order to load parameters from the section `<parameters>`, the user can mimic how this is also carried out by DualSPHysics. If different properties will be defined for different fluid volumes, section `<properties>` can be used (that is also explained in the XML guide).

## How can I store new properties of the particles (e. g. Temperature)?

The new file format (.bi4) and the post-processing tools have been designed to include new properties defined by the user for its own implementation. The function JSph::SavePartData() already includes an example of how to store new particle data. Then, the post-processing codes will automatically manage all variables included in the .bi4 files.

## How must I cite the use of the code in my paper?

Please refer to the code if you use it in a paper with reference [Crespo et al., 2015].