GPUSPH User Guide

version 4.0 — October 2016

Contents

1 Introduction

To run simulations with your own setup, you must create a new `Problem`. This is done by creating a new `cu`source file, with the associated header (e.g. `MyProject.cu` and `MyProject.h`), placing them under `src/problems/user`. Beginners should use one of the provided sample files in `src/problems` as a template for their project. There are two main samples available in the `src/problems` directory: `ProblemExample` (for Lennard-Jones or dynamic boundaries) and `CompleteSaExample` (for semi-analytical boundaries).

2 Anatomy of a project

Below are the steps required to build a new project and run it with GPUSPH:

1. In case you use semi-analytical boundary conditions, follow the steps described in section 3 for the pre-processing;
2. Create `MyProject.cu` and `MyProject.h` files in the `src/problems/user` directory.
3. In the GPUSPH folder, compile the code for your project:
   ```
   make problem=MyProject
   ```
4. Execute GPUSPH:
   ```
   ./GPUSPH
   ```
5. Follow the steps described in section 6 to visualize and post-process the results.

3 Specific pre-processing for semi-analytical boundaries

Several types of boundary conditions are available in GPUSPH. They are described in the theory guide. With classical boundary conditions (Lennard-Jones, dynamic boundaries) the problem geometries are defined and filled with particles by GPUSPH itself. For simulations involving complex objects and/or open boundaries, the semi-analytical boundaries can be used. They make it possible to import a geometry in

`h5sph` format. These geometries are generated using a mesher (*e.g.* `SALOME`) and `CRIXUS`, an open-source software which is able to fill the computational domain with particles.
The pre-processing steps specific to semi-analytical boundaries are the following:

1. create a mesh of the boundaries in SALOME and export it as a binary STL file;
2. run the `testTriangle` script to know the minimum interparticle distance to set in the simulation, with the following syntax:

   ```
   testTriangle NameOfTheSTLFile.stl 0.1
   ```

   where 0.1 could be any number (the program just checks if the distance between the center and the vertex of a triangle is bigger than the specified number, and returns the maximum and minimum found for that dimension);
3. run CRIXUS to fill the domain with particles:

   ```
   Crixus NameOfTheINIFile.ini
   ```

4. copy the resulting `h5sph` file(s) containing the initial particles to the directory `data_files`.

These steps are described with more detail below.

3.1 Preparing the geometry with SALOME

The input files for CRIXUS are meshes of:

- the total domain's boundaries
- the free-surface
- the special boundaries (for moving objects and/or open boundaries)

These meshes must be composed of triangles and can be generated using SALOME, which is an open source software very useful for 3D modeling and meshing.

It is important to generate meshes with homogeneous triangle size, otherwise the quality of results may be affected.

In order to start a new project in SALOME, click on `File/New`. When you save your project, SALOME creates a file with the `.hdf` extension, which stores all the geometry elements and meshes that you design for your project.

Building the geometry

The complete SALOME documentation for the GEOMETRY module can be found here:
`http://docs.salome-platform.org/7/gui/GEOM/`

Designing is easy in SALOME. To start building the geometry elements, click on the "Geometry" module. There are 7 types of basic geometrical elements:

1. VERTEX: it can be created by providing its coordinates, by clicking on the vertex of another geometry element, by using another point as reference... There are many ways which are described in the "Point Construction" window which appears when we click on "Create a point"

2. SEGMENT: it can be created providing two points that were previously stated, or using the intersection of two planar elements.

3. WIRE: a wire is just a series of segments. It can be a closed wire or if the end matches the start, or an open one.

4. FACE: a face is just a limited plane

5. SHELL: a shell is a series of faces. SALOME would consider it a closed shell when it encloses a volume

6. SOLID: a solid is just a limited part of the 3D space; it can be easily created on the basis of a closed shell

7. COMPOUND: a compound is just the combination of two or more elements of different type, merged into one single element

Apart from these basic geometry types, we can find three special types, which are DIVIDED DISK, DIVIDED CYLINDER, and SMOOTHING SURFACE. Among these three special types, the most interesting is the smoothing surface, as it is useful to create 3D surfaces from a point cloud. Finally, we can find auxiliary geometry elements such as circles, ellipses, arcs, vectors, sketches, polylines, cylinders, cones, spheres, cubes, torus, disks, T shape pipes, etc.

SALOME makes it possible to import geometries from a wide range of file types: STL, BREP, STEP,etc. It is possible to import a geometry in STL format (generated with another 3D modeling software, such as Autocad, SolidWorks, Catia, etc.).

Caution: STL files are ASCII or binary files in which geometry is described by triangles. Each element of an STL file is composed by the 3 coordinates of each 3 vertex of the triangle, and the 3 components of the triangle's normal vector. This means that when we export some geometry elements in STL format, triangles would be automatically created. When importing this file in SALOME, the geometry is then composed of triangular faces and that the meshing operations to be implemented afterwards are influenced by this previous and automatic discretization of the geometry. This results in bad mesh quality. So when importing geometry on a STL format, a redesigning of it is necessary in order to obtain a good mesh quality.

Other very useful tools of SALOME are the boolean operations on solids. It is possible to fuse, intersect solid objects, use a solid object as a cutting tool for another one, etc. It is also possible to perform operations like rotation, translation, etc. on the geometrical objects.

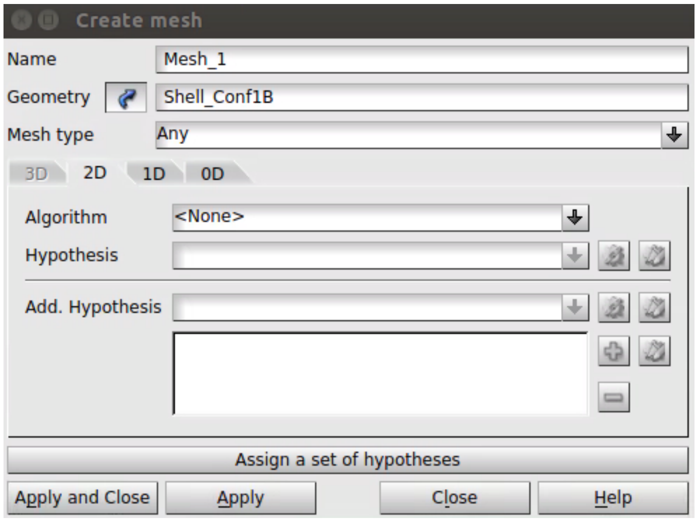
Generating the mesh

The complete SALOME documentation for the MESH module can be found here:
`http://docs.salome-platform.org/latest/gui/SMESH/index.html`

Once the geometry is defined, it can be meshed using SALOME. In order to access the meshing tools, change from the Geometry module to the Mesh module. To create a mesh from a geometry element, click on `Create Mesh`, and a window opens (see Figure 1) in which the following options are available:

- Name: the name of the mesh which is going to be created
- Geometry: the geometry element that we want to mesh
- 3D/2D/1D/0D: it's the nature of the mesh that we are going to create, it automatically chooses the correct one depending on the type of element that we have specified in Geometry
- Algorithm: is the meshing method's algorithm. Netgen 1D-2D works well for shell meshing.

- Hypothesis: here we can specify the hypothesis to be used by the algorithm method. Clicking on the first icon on the right, we can specify the parameters of the algorithm. See the SALOME documentation for more details about the options. For example, for the Netgen 1D-2D algorithm, a window opens with all the options shown in the Figure 2.



Name: Mesh_1

Geometry: Shell_Conf1B

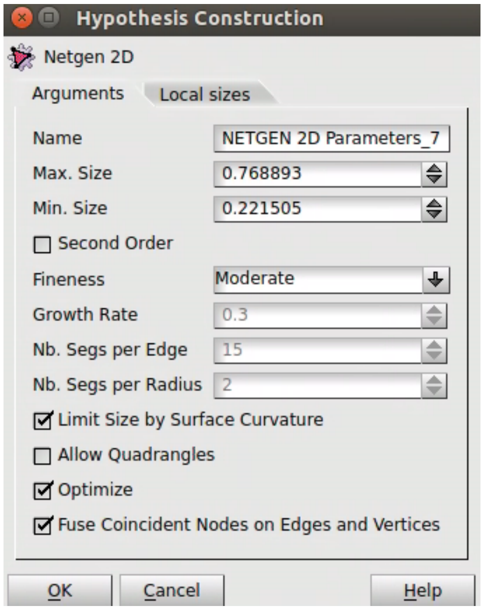
Mesh type: Any

Algorithm: <None>

Figure 1: Screenshot of the mesh options window in SALOME.

The most relevant mesh options with the Netgen 1D-2D algorithm are Max Size and Minimum Size. It is important to note that SALOME usually respects the Max Size, whereas the minimum size is often ignored due to geometry-mesh adaptation problems. In addition, the minimum size would be always delimited by the characteristic size of the geometry, that is to say, the minimum length of the faces composing the shell. Regarding the option Fineness, Fine works usually well. Once we press OK and then Apply, an element of mesh type will appear in the Object Browser on the left side of the screen. The icon will appear with an exclamation mark on it: that means that the mesh has not been computed yet. To do so, we click on the icon Compute or we just do right click and we select the option Compute. The algorithm will now begin iterating until a solution has been found. The mesh is then prepared to be exported as an STL file in order to be used as an input for CRIXUS.

Caution: STL files for CRIXUS need to be binary, so when exporting the mesh, make sure you choose the correct file format at the bottom tab.



Name: NETGEN 2D Parameters_7
Max. Size: 0.768893
Min. Size: 0.221505
☐ Second Order
Fineness: Moderate
Growth Rate: 0.3
Nb. Segs per Edge: 15
Nb. Segs per Radius: 2
☑ Limit Size by Surface Curvature
☐ Allow Quadrangles
☑ Optimize
☑ Fuse Coincident Nodes on Edges and Vertices

Figure 2: Screenshot of the Netgen 1D-2D hypothesis window in SALOME.

3.2 Check the triangles' dimensions with `testTriangle`

In order to know the triangle dimensions that we have just created in SALOME, we can run the TESTTRIANGLE script, available with CRIXUS.
To compile TESTTRIANGLE, use this command in the `Crixus.git/scripts` directory:

```
gcc test-triangle-size.c -lm -o testTriangle
```

It is recommended to add the path to the TESTTRIANGLE binary to your `$PATH` environment variable. Add this line in `~/.bashrc`:

```
export PATH=/your_path/Crixus.git/scripts/testTriangle:$PATH
```

where `/your_path` is your path to the CRIXUS directory.
To work correctly, the distance between particles (dr) of a given simulation should not be less than the maximum distance between the center and the vertex of a triangle. This is why TESTTRIANGLE is used to get the maximum value of this magnitude for all triangles of the main geometry STL file. Run the program with the command:

```
./testTriangle NameOfTheSTLFile.stl 0.1
```

Where 0.1 could be any number. The program just checks for each triangle of the mesh whether the radius is bigger than the specified number, and returns the maximum and the minimum value found for the radius of each triangle. The user is supposed to set the dr of the simulation equal or superior to the maximum value got by TESTTRIANGLE. However, it should not be much bigger, as the number of neighbors would increase enormously, driving the simulation slower in terms of computation time.

3.3 Fill the geometry with particles using CRIXUS

CRIXUS is an open source software which performs the initialization of the fluid as a previous step for GPUSPH. As input, it basically needs the STL files describing the model geometry and, if necessary, an STL file describing the free-surface and/or the special boundary meshes. In addition, we have to specify the distance between particles of the simulation and other options as described below.

Remark: what follows is just a summary of the CRIXUS manual, available in the README file of CRIXUS.

CRIXUS takes an INI file as an input. This is essentially a text file with `.ini` extension where we specify all the necessary STL files and other parameters. The INI file follows the following structure:

```
[mesh]
  stlfile=salome_box_0.02_with_floating_box.stl
  dr=0.017634
  fshape=sa_box_fshape.stl
[special_boundary_grids]
  mesh1=sa_box_sbgrid_1.stl
  mesh2=sa_box_sbgrid_2.stl
[fill_0]
  option=geometry
  xseed=0.5
  yseed=0.5
  zseed=0.2
  dr_wall=0.018
[output]
  format=h5sph
  name=xcomplete_sa_example
```

```
split=yes
```

Every section of the file starts with the title `[XXX]` and all following statements refer to that section. These sections are described below.

Section [mesh]

Here is where the main geometry STL file will be specified. The value of stlfile is the name of the STL file containing the geometry of the walls and other elements limiting the movement of the fluid. The value of dr is the distance between particles which CRIXUS will use to fill the domain with fluid. swap_normals enables the user to change the orientation of the shell's faces of the STL file. Faces should be inner-fluid oriented for the simulation to work correctly. fshape is optional and specifies the free surface of the fluid which we want the simulation to start with. The geometry of the surface has to be represented by a STL file, whose name is the value of fshape. If no initial surface is specify, CRIXUS would fill the domain until the Z limit is reached. Note that this time the STL file need to be binary, but its meshing is irrelevant since its only goal is to limit the domain's filling.

Section [fill_number]

In this section we tell CRIXUS how to fill the domain. There are two ways of doing that: we can fill the domain by boxes (option=box) or by a seed point (option=geometry). The first one just fills the specified box with fluid, whereas the second one starts filling the domain from the specified seed point and only stops if it reaches a wall or the free surface. We can call the filling algorithm as many times as wished, even with different filling option. This is useful when we want to initialize the fluid in two areas which are not self connected.

Section [special_boundary_grids]

Here we are able to specify which boundaries of the domain are open boundaries, that is to say, boundaries where the fluid can get in and out. It is important to note that these boundaries have to be part of the geometry represented by the STL file specified as stlfile at the beginning. Then, the open boundaries will be implemented by specifying the names of their STL files.

Section [output]

In this section we simply choose the format of the output file. This could be either `.h5sph` or `.vtu` format. The first one is a table ready to be used as the $t = 0$ parameters for the simulation in GPUSPH, whereas the second one is the file format that can be read by ParaView. In a normal situation, in this section we would always write `format=h5sph`, since this gives us the file to be implemented on GPUSPH. Nevertheless, if we want first to observe the results of the filling processes in ParaView, we would write `format=vtu`. The H5SPH files can be opened in HDF, a program to visualize data in tables. It is important to know, once again, that these are just the basic options for CRIXUS, sufficient to launch most of the simulations. It could happen, however, that we needed to specify more options in order to customize our filling process: in this case, the rest of the information, including the developer's e-mail can be found in the mentioned README file.

In order to run CRIXUS, follow these steps:

1. Open the Linux Terminal
2. Place the directory in the folder where we have all the STL files used by CRIXUS, as well as the INI file:

   ```
   cd Directory/Of/The/Files
   ```

3. Launch Crixus:

   ```
   Crixus NameOfTheINIFile.ini
   ```

4. Once Crixus has finished, copy the resulting H5SPH files to the `data_files` directory of GPUSPH. The geometry (i.e., the H5SPH file) is now ready to be used by GPUSPH.

4 Setting up the simulation

As said before, the simulation setup only involves manipulating the .cu and the .h files of your problem in order to specify all its parameters before running GPUSPH. The structure of a problem, is in fact the structure of the .cu file, which could be defined as follows:

1. GEOMETRY. As the mesh geometry has previously been created by CRIXUS, we only have to specify the file containing this information: the .h5sph file.
2. SIMULATION PARAMETERS. There are several simulation parameters that need to be specified, concerning the time, the frequency of output writing or specific SPH parameters.

3. INITIAL CONDITIONS. We need to specify an initial value for each of the fields to be implemented on each particle.
4. BOUNDARY CONDITIONS. Boundary conditions have also to be stated before running GPUSPH.

In the following sections we develop each section to help the readers write the .cu file in order to build their own simulation.
Creating new problems in GPUSPH is done using the `XProblem` class. `XProblem` does much of the work of defining and placing objects of common shape (cubes or parallelepipeds, spheres, etc) in the domain. These shapes can be solid or fluid or filled with fluid. `Xproblem` takes care of filling the appropriate volume(s) with particles. It can also insert h5sph objects coming from the Crixus preprocessing step. It has the added feature of taking care of the setup for floating or moving bodies that are handled through a dynamics library: `Project Chrono`. It also takes care of the open boundaries identification, and their type (pressure driven or velocity driven). Simple initializations can be automatically performed by `XProblem`, in particular to prescribe an initial hydrostatic pressure or to change the mass of some objects.
The prescription of the moving bodies velocities can be done in the function:
`moving_bodies_callback`
while an advanced user initialization can be coded in the function:
`initializeParticles`.

4.1 Problem Examples

The supplied Problem examples are located in the src/problems directory:

- `AccuracyTest`:
  schematic single-fluid dam break case on a flat bottom;
- `Bubble`:
  two-phase flow case, representing the motion of a bubble lighter than the surrounding fluid;
- `BuoyancyTest.cu`:
  a rectangular tank of still water with a submerged torus that is released when the problem begins;
- `DamBreak3D.cu`:
  schematic single-fluid dam break case with an obstacle;

- `DamBreakGate`:
  same case as the previous one but the dam break is managed by a vertically sliding gate;

- `DynBoundsExample`:
  double-periodic channel flow;

- `InputProblem`:
  several problems are included in this one, all based on the semi-analytical boundaries.

- `OffshorePile`:
  waves propagating in a y-periodic channel and hitting a cylinder;

- `OilJet`:
  oil contained in a tube propelled by a piston towards the top and flowing on a plate;

- `OpenChannel`:
  channel flow (with y-periodicity or side walls);

- `Seiche`:
  sloshing case;

- `SolitaryWave`:
  solitary wave generated by a piston. Possibility to add cylinders on the waves path;

- `Spheric2LJ`:
  schematic dam break case on a obstacle (available measurements) – Lennard-Jones boundary conditions;

- `Spheric2SA`:
  schematic dam break case on a obstacle (available measurements) – semi-analytical boundary conditions;

- `StillWater`:
  still water case;

- `TestTopo`:
  example of use of a topography file for the geometry;

- `WaveTank`:
  wave generation on an inclined bed;
- `Objects`:
  example showing how to handle moving objects;
- `CompleteSaExample.cu`:
  generic example for semi-analytical boundaries;
- `ProblemExample.cu`:
  generic example for dynamic and Lennard-Jones boundaries.

Each of these examples can be run by typing `make problem=ProblemName`, in the top level GPUSPH directory. It is recommended that the user try them to ensure everything checks out in terms of CUDA and GPUSPH.

Some of the problems are described with more details below.

`BuoyancyTest.cu` includes a rectangular tank of still water with a submerged torus (or by changing object_type, a cube or sphere) that is released when the problem begins. As time advances, the torus rises through the water column as it has a density half that of water and then it reaches the free surface and floats.
The output of `BuoyancyTest` is written every 0.01 seconds into a file in the directory tests designated by the problem name and the date and time. The files are in VTU format that can be read by ParaView. Alternative formats, such as text, can be chosen by changing the writer in the `add_writer` command.

`ProblemExample.cu` shows how a matrix of objects can easily be added to a problem. The basic problem is a semi-infinite domain with a plane used as a floor (`addPlane`). A 4 x 4 array of solid cubes is set-up using the addCube command multiple times. The `GT_FIXED_BOUNDARY` (GT=Geometry Type) means that the cubes are solid. The cubes are also rotated 45 degrees by a rotate command. Then a smaller array of spheres of fluid are defined. `GT_FLUID` is used in the `addSphere` command. Note for the fluid the `setDensityByMass` establishes the fluid density.

In `DamBreak3D.cu` `makeUniverseBox()`, which has as its arguments two opposite corners of the project domain—the first corner is the origin. This command sets up the domain using analytical planes as boundaries. These planes do not require the use of particles. Water is added to the domain with the `addBox()` command – note that the fluid is denoted by `GT_FLUID` (GT=GeometryType). The fluid behind the dam is 0.4

m deep. A variety of obstacles can be added in front of the dam. As provided, there is just a single object, but by invoking the model with `./GPUSPH --num_obstacles 3` three obstacles will be in front of the dam. These obstacles can be rotated from their original position by `./GPUSPH --num_obstacles 3 --rotate_obstacle true`

Another run-time option includes `--wet true or false`, which puts a $0.1m$ layer of water around the obstacles (and in front of the dam).

`CompleteSaExample.cu` is an example using the Semi-Analytical boundary conditions (SA). This type of boundary condition was chosen in the `SETUP_FRAMEWORK`, which is a class that contains the various simulation choices. For example it contains `boundary<SA_BOUNDARY>` as the choice. This example consists of a tank with a free surface and a submerged inlet. There is a floating cube as well. The example requires data files that are available on the `www.gpusph.org` web site:

```
wget http://www.gpusph.org/downloads/data_files_XCompleteSaExample.tgz
```

or, in your browser,

`www.gpusph.org/downloads/data_files_XCompleteSaExample.tgz`

The file (`data_files_XCompleteSaExample.tgz`) is uncompressed in the root GPUSPH directory. It will create a directory `data_files`, containing four `.h5sph` to set up the fluid and boundaries and one `.stl` file to define the cube. (In addition there are five files that were used to generate the input files using Crixus, an open source pre-processor). The problem is large and will take some time as it involves the semi-analytical boundaries. There are 122,642 particles in total, of which 56821 are fluid particles and the rest are boundary and vertex particles.

4.2 Generic Example

To write your own example, you can use one of the examples as a template, but they all have a similar format as Example. For example, looking at the text file, `BuoyancyTest.cu` in the directory `src/problems`, we see that, after the appropriate `includes`, including `BuoyancyTest.h`, the example is defined as a child of the `XProblem` class. Then the setup is done following the structure below.

4.2.1 Framework setup

The `SETUP_FRAMEWORK` function enables to change the general options of the simulation:

```
SETUP_FRAMEWORK(
  kernel<WENDLAND>,
  formulation<SPH_F1>,
  viscosity<DYNAMICVISC>,
  boundary<SA_BOUNDARY>,
  periodicity<PERIODIC_NONE>,
  add_flags<ENABLE_INLET_OUTLET | ENABLE_DENSITY_SUM
      | ENABLE_MOVING_BODIES | ENABLE_FERRARI>
);
```

1. The first item enables to choose the type of kernel:
   QUADRATIC
   CUBICSPLINE
   WENDLAND
   GAUSSIAN

2. The second item enables to choose the type of SPH formulation:
   SPH_F1
   SPH_F2
   SPH_GRENIER
   where SPH_F1 is a WCSPH single-fluid formulation, SPH_F2 is a WCSPH multi-fluid formulation and SPH_GRENIER is another mutli-fluid formulation based on the Grenier formulation.

3. The third parameter is the viscosity model. There are 5 options for this term:
   ARTVISC
   KINEMATICVISC
   DYNAMICVISC
   SPSVISC
   KEPSVISC

4. The fourth parameter is the type of boundary. There are 4 options for this term:
   LJ_BOUNDARY
   MK_BOUNDARY
   DYN_BOUNDARY
   SA_BOUNDARY

5. The fifth item makes it possible to enable periodicity:
   PERIODIC_NONE
   PERIODIC_X
   PERIODIC_Y
   PERIODIC_XY
   PERIODIC_Z
   PERIODIC_XZ
   PERIODIC_YZ
   PERIODIC_XYZ

6. Finally, the `add_flags` term enables the implementation of some extra functions in the simulation, such as the extra Ferrari diffusion term, the in & out boundaries or the water depth function, which computes the flow depth at a given set of X, Y:
   ENABLE_DTADAPT
   ENABLE_XSPH
   ENABLE_PLANES
   ENABLE_DEM
   ENABLE_MOVING_BODIES
   ENABLE_INLET_OUTLET
   ENABLE_WATER_DEPTH
   ENABLE_FERRARI
   ENABLE_DENSITY_DIFFUSION
   ENABLE_DENSITY_SUM
   ENABLE_GAMMA_QUADRATURE
   ENABLE_INTERNAL_ENERGY

4.2.2 Generic simulation parameters

```
// Initialization of simulation parameters
m_name = "XCompleteSaExample";
set_deltap(0.02f);
physparams()->r0 = m_deltap;
// Set world size and origin.
// HDF5 file loading does not support bounding box
// detection yet
const double MARGIN = 0.1;
const double INLET_BOX_LENGTH = 0.25;
// size of the main cube, excluding the
// inlet and any margin
double box_l, box_w, box_h;
box_l = box_w = box_h = 1.0;
// world size
double world_l = box_l + INLET_BOX_LENGTH
    + 2 * MARGIN; // length is 1 (box) + 0.2 (inlet box length)
double world_w = box_w + 2 * MARGIN;
double world_h = box_h + 2 * MARGIN;
m_origin = make_double3(- INLET_BOX_LENGTH - MARGIN,
    - MARGIN, - MARGIN);
m_size = make_double3(world_l, world_w ,world_h);
// time parameters
simparams()->tend = 40.0;
simparams()->dt = 0.00004f;
simparams()->dtadaptfactor = 0.3;
// open boundary information
simparams()->numOpenBoundaries=2;
```

- `m_name` is the problem name.
- `deltap` is the distance between particles used in the current simulation. For consistency reasons, it has to be the same that we have set in the INI file for CRIXUS.
- `m_size` and `m_origin` are the size and the origin of the domain (defined in SALOME in case semi-analytical boundaries are used).
- `tend` is the time at which the simulation should stop.
- `dt` is the size of the first time-step or the time-step size if the ENABLE_DTADAPT flag is not activated.
- `dtadaptfactor` is the CFL coefficient, usually taken as 0.3.
- `numOpenBoundaries` is the number of open boundaries in the simulation.

4.2.3 SPH parameters

```
// Initialization of SPH parameters
simparams()->maxneibsnum = 352;
// buildneibs at every iteration
simparams()->buildneibsfreq = 1;
// Slightly extend kernel radius for gamma computation
simparams()->nlexpansionfactor = 1.1;
// Ferrari correction
simparams()->ferrari = 1.0;
```

- `maxneibsnum` is simply a limit for the number of neighbors computed in the SPH method. It allows the user control the amount of calculus done for each particle at each iteration. If the mesh of the geometry is coherent with `deltap`, the maximum neighbor number should not be over 280, so setting this number to 300 would be a good choice.

- `buildneibsfreq` is the neighbor counting frequency, in terms of number of time steps.

- `nlexpansionfactor` is the factor increasing the area where we count the neighbor particles for the computation of γ with SA boundaries.

- `ferrari` is the Ferrari diffusion coefficient, which is used to potentiate diffusion dissipation (0 for no extra diffusion, 1 for the maximum)

4.2.4 Physical parameters

```
physparams()->gravity = make_float3(0.0, 0.0, -9.81);
size_t water = add_fluid(1000.0);
set_kinematic_visc(0, 1.0e-2f);
set_equation_of_state(water, 7.0f, 50.0f);
setHydrostatic();
```

This specifies the gravity field, the fluid density (through the `add_fluid` function), and the equation of state to be used through `set_equation_of_state`. In this function, the first argument is the fluid considered, the second one is the exponent in the equation of state (usually 7), and the third one is the numerical speed of sound. The numerical speed of sound can also be set by specifying reference velocity and water height:

```
// Reference quantities for speed of sound computation
setWaterLevel(0.5);
```

```
setMaxParticleSpeed(7.0);
```

An initial hydrostatic pressure is prescribed in the domain. The code automatically finds out which is the highest particle in the domain, and initializes the pressure based on that value. To change the water level to be considered in the initialization, the function `setWaterLevel` can be used. To disable the hydrostatic initialization, set:

```
m_hydrostaticFilling = false;
```

4.2.5 Results parameters

```
  // Drawing and saving times
  add_writer(VTKWRITER, 1e-1f);
```

The writer (for the output data) is chosen with the `add_writer` command (usually the VTK writer, which provides files to be read by ParaView). The file writing frequency (in terms of simulated seconds) can also be specified. That is, in this case, we will have a VTU file every 0.1 simulated second for example. It is important to note that, since some simulations could become too large, this frequency is essential in order to limit the size of the result files.

4.3 Building and initializing the particle system

With classical boundary conditions:

With DYNAMIC or LENNARD-JONES boundary conditions, the problem geometry and the filling with particles is done inside GPUSPH. An example of generation of arrays of cubes and spheres in the computational domain is given in `XProblemExample.cu`. The geometrical objects can be added using functions like:

```
addCube(GT_FIXED_BOUNDARY, FT_BORDER,
            Point(X,Y,Z),cube_size);
```

The geometry type (GT) may be fluid, fixed boundary, open boundary, floating body, moving body, plane, and testpoint, eg. `GT_OPENBOUNDARY`. They are enumerated in `XProblem.h`.

GPUSPH has a variety of geometrical objects that can be used to generate Problems. The geometrical objects are defined in the `src/geometry` folder. The XProblem class makes it possible to rotate or shift them after they were defined. They can be assigned a mass and a center of gravity. In two dimensions, the objects (in `C++` terms, classes) include *Point, Vector, Segment, Rect (rectangle), Circle.* In three dimensions, there are additional objects: *Cone, Cube, Cylinder, Sphere and TopoCube.* Using these objects, many types of Problems can be constructed. For the three dimensional case, the bottom (bathymetry) of the problem domain can be input via a file, using the TopoCube object and a dem file.

The *Point* object is usually used as a three dimensional object containing the location of a point in three dimensions. All numbers are double precision. Associated with the Point object are functions that determine distance (or distance squared) of a point from the origin or the distance from another point.

A *Vector* object is a three dimensional double precision object of three space coordinates, x,y, and z. Vector has a number of associated and useful functions, such as Vector.norm, for the length of the vector.

The *Cube* object is really a parallelepipeds, defined by an origin, given by a Point object, and three vectors are used to define the size and orientation of the cube. For example, here is a box that delimits an experimental domain (taken from the DamBreak3D.cc example), called *experiment_box.*

```
experiment_box = Cube(Point(0, 0, 0),Vector(1.6, 0, 0),Vector(0, 0.67, 0), Vector(0,
0, 0.4));
```

This box has a corner located at the origin of the domain, with $(x, y, z) = (0, 0, 0)$, and three vectors from this point describe the cube, which happens to be 1.6 m long in the x direction, 0.67 m long in the y direction, and 0.4 in the z direction.

So far we have only defined the cube *experimentbox*, we have given it no properties. For this particular box, which bounds the computational domain, its bottom and four sides will be set as boundary particles, as we will see later.

Associated with the Cube object are commands to fill the inner part of the box with particles, or to fill the boundaries as with boundary particles.

The *Cylinder* object is defined by a point that determines the location of the center of the disk that forms its base, a vector that defines the radius about the point, and then another vector that defined the height of the cylinder. The cylinder object also has fill and FillBorder commands. For example,

```
jet = Cylinder(Point(0.,0.,0.), Vector(0.5,0.,0.), Vector(0.,0.,1.));
```

would define a cylinder located at the origin with radius 0.5 and height 1.0 with the name jet. The Cylinder object can be used to define a cylindrical column of fluid, using the `jet.Fill` command for the defined cylinder, jet. The mass of the particles forming jet is set by `jet.SetPartMass` function. If the jet was supposed to be a pipe, the `jet.FillBorder`, with suitable arguments, would use boundary particles for the pipe called jet. Two of the arguments (Booleans: true or false) of the method determine if the cylinder is closed on the bottom or the top.

The *Sphere* object is defined by a point that determines the center of the sphere, a vector that determines its radius (and equatorial normal), and a vector pointing to the sphere's pole. For a sphere, these two vectors have equal magnitude and are normal to each other. The Sphere object uses the Circle object in layers to create a sphere.

A *TopoCube* object is used to define a domain that has the bottom of the cube provided by a data file. The geometry of the TopoCube is determined the same was as in the Cube object. The data file has a strict format; for example:

north: 13.2
south: -0.2
east: 43.2
west: 0.54
rows: 134
cols: 432
{data in 134 rows with 432 entries per line; numbers space separated}

The numbers following the compass directions are the length of the domain described by the data, in meters. (North and south correspond to the +Y axis and the -Y axis, while E and W are aligned with the +X and -X directions.) The internal variables (see problem TestTopo.cc) $nsres$ and $ewres$ are grid resolutions determined by $nsres = (north - south)/(nrows - 1)$ and $ewres = (east - west)/(ncols - 1)$.

The data file is read using the TopoCube.SetCubeDem function, which is called with arguments (float H, float *dem, int ncols, int nrows, float nsres, float ewres, bool interpol), where H is the depth of the cube, *dem points to the array of bathymetric data in the data file, ncols and nrows are the number of columns and rows in the dem data set, nsres and ewres is the spacing between the bathymetric data in the north/south direction and the east/west direction, and interpol (not the police) is the boolean variable for interpolation. FillBorder will fill a face with particles–the particular face is determined by face_num, which takes on the values of (0,1,2,3), for the front face, the right side face, the back face, and the left side face (facing the -x direction) for a rectangular box.

Other objects can be defined and added to the source directory to allow for additional flexibility.

With semi-analytical boundary conditions:

The fluid initialization performed by CRIXUS and stored in the H5SPH files is used by GPUSPH to start the simulation. The specification of the file containing the fluid particles occurs with the following statement:

```
addHDF5File(GT_FLUID, Point(0,0,0),
"./data_files/XCompleteSaExample/0.my_project.fluid.h5sph",
NULL);
```

The specification of the file containing the special boundary particles occurs with the following statement:

```

// Main container
GeometryID container =
addHDF5File(GT_FIXED_BOUNDARY, Point(0,0,0),
  "./data_files/MyProject/0.my_project.boundary.kent0.h5sph",
  NULL);
disableCollisions(container);

// Inflow boundary
GeometryID inlet =
  addHDF5File(GT_OPENBOUNDARY, Point(0,0,0),
  "./data_files/MyProject/0.my_project.boundary.kent1.h5sph",
  NULL);
disableCollisions(inlet);

GeometryID cube =
  addHDF5File(GT_FLOATING_BODY, Point(0,0,0),
  "./data_files/MyProject/0.my_project.boundary.kent2.h5sph",
  "./data_files/MyProject/MyProject_object_file.stl");
// output forces on the cube
enableFeedback(cube);
// set the cube density
setMassByDensity(cube, 500);
```

In order to specify whether the open boundary is pressure driven or velocity driven, the following lines are used:

```
setVelocityDriven(inlet, 1);
setVelocityDriven(outlet, 0);
```

Once again the GT (GeometryType) can be fluid, fixed boundary, open boundary, floating body, moving body, plane, or testpoint, eg. `GT_OPENBOUNDARY`.

5 Running your simulation

To run your simulation you first need to compile GPUSPH for your problem. To do so, in the GPUSPH folder, run:

```
make problem=MyProblem
```

Remark:

- If you are running a multi-node simulation, do not forget to add the option `mpi=1`.
- If your are running a simulation with moving objects, do not forget to add the option `chrono=1`.

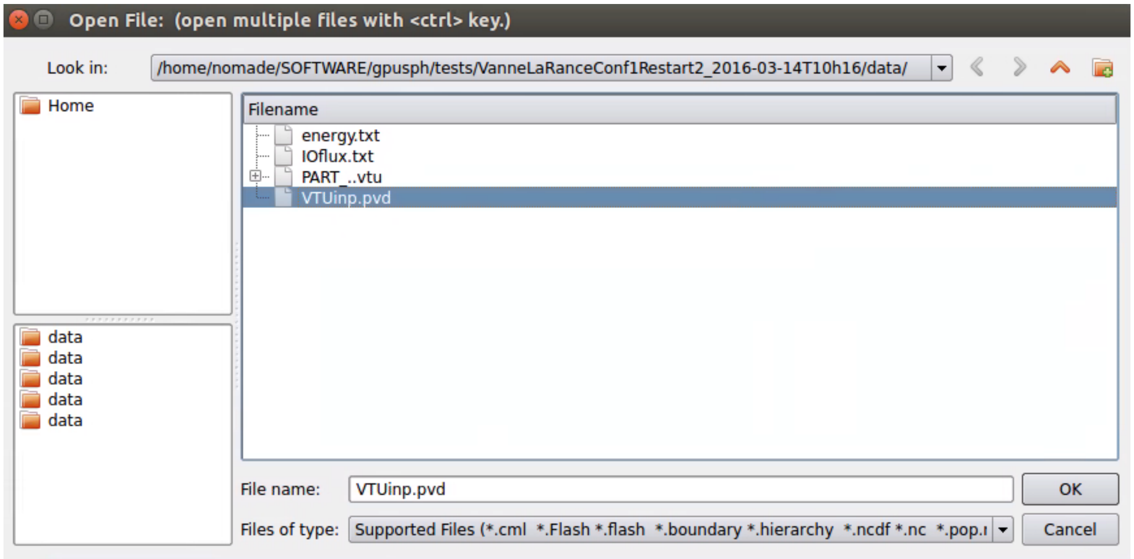
See the installation guide or run `make --help` for the complete list of compilation options.

6 Visualizing the results

The results of the simulation are stored in a directory under `tests`, named after the used Problem and the date of execution (e.g. `tests/DamBreak3D_2014-6-12T13h23`). Data files (found in a `data` sub-directory of the specific test directory) are normally written in VTK Unstructured Grid format (`.vtu`) and can be visualized with ParaView.
The files necessary to hotstart the simulation are also stored in the `tests/MyProject_2014-6-12T13h23/data` folder.
The run directories and their content are preserved until manually removed. The `scripts/rmtests` auxiliary script can be used to clean up the `tests` directory.
A tutorial to start using ParaView is available here:
`http://www.paraview.org/Wiki/Beginning_ParaView`

To open a file, click on the first upper icon on the left. The VTU files are named as `PART_00025.vtu` where the number corresponds to the output files numbering. PARAVIEW allows the user to visualize at the same time all the VTU files, just clicking on VTUinp.pvd or selecting the all set of `PART_..vtu` files (see the Figure

3). The set of VTU files can be analyzed as a movie by clicking on the play buttons at the top of the screen.



Open File: (open multiple files with <ctrl> key.)

Look in: /home/nomade/SOFTWARE/gpusph/tests/VanneLaRanceConf1Restart2_2016-03-14T10h16/data/

Filename

energy.txt

IOflux.txt

PART_..vtu

VTUinp.pvd

File name: VTUinp.pvd

Files of type: Supported Files (*.cml *.Flash *.flash *.boundary *.hierarchy *.ncdf *.nc *.pop.ı

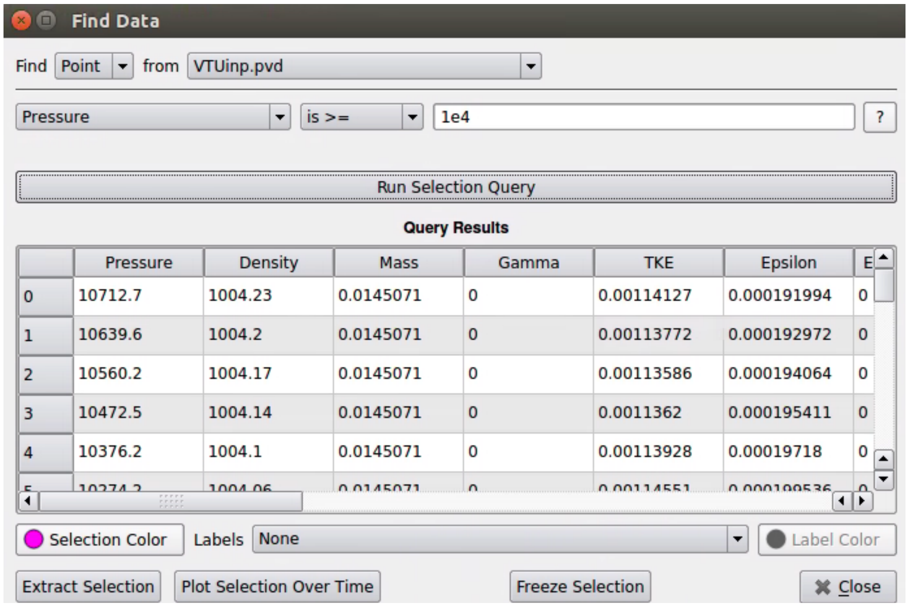
Figure 3: Screenshot of the window for file opening in PARAVIEW.

After selecting the `VTUinp.pvd` or the desired `PART` files, a window appears at the bottom-left part of the screen. Press `Apply` to confirm the file opening: the set of particles appears at the center of the screen. When pressing `Apply`, a window opens with three main sections: `Properties`, `Display` and `Information`. The second one enables the user to decide which field should be printed (first tab of the section `Coloring`). With the `Show` option, we can make the color legend appear, and with the `Edit` one, we can customize it. Below this section we find a set of options that enables us to manage the plotting as desired. The `Information` section provides information like the total size of the dataset.

Below some useful filters of ParaView are listed. The filters are all available through the `Filters` tab at the top of the screen, in the section `Alphabetical`, or through shortcuts in the main window.

- **Find Data**
  Find data by scalar value makes it possible to select particles on the basis of the value of their fields. When activating a filter a window opens. In order to

select the particles we want, in that window (see the Figure 4), we have to set the Find tab on Point in that window. In the left tab we can choose the desired field, whereas in the right one we can set the value (note that there are several conditions: >=, <=, =, between, etc.). Once we have set all the options, we press on Run Selection Query and we will be able to see a table containing all the particles that match the imposed condition. In addition; we will be able to see these particles on the Layout, colored with the chosen Selection Color.



Find Point from VTUinp.pvd

Pressure is >= 1e4

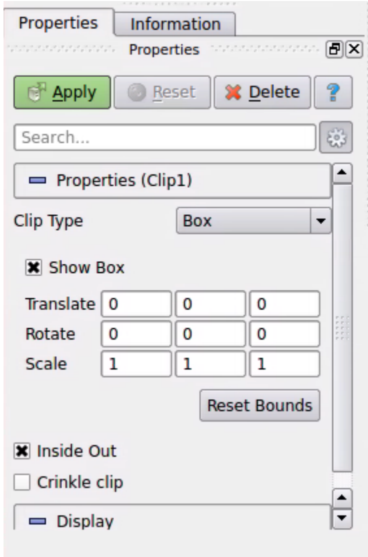
Labels None

Query Results

	Pressure	Density	Mass	Gamma	TKE	Epsilon	E
0	10712.7	1004.23	0.0145071	0	0.00114127	0.000191994	0
1	10639.6	1004.2	0.0145071	0	0.00113772	0.000192972	0
2	10560.2	1004.17	0.0145071	0	0.00113586	0.000194064	0
3	10472.5	1004.14	0.0145071	0	0.0011362	0.000195411	0
4	10376.2	1004.1	0.0145071	0	0.00113928	0.00019718	0

Figure 4: Screenshot of the `Find Data` window in PARAVIEW.

- **Clip**
  In order to visualize a section of the domain, since the fields are discrete, we are obliged to perform a `Clip`. The `Clip` window is shown in the Figure 5. By changing the `Clip Type` tab into `Box`, it is possible to set the dimensions and position of the box. It is important to click on the `Inside Out` button to select the particles that are inside the box. Once ready, click on `Apply` to get a new `Clip` object in the Pipeline Browser. You can manipulate it in the same way as the main dataset. You can also perform clips with planes.
  Remark: the Slice option does not work because the flow fields are not continuous. To make a slice, we currently apply a thin box-type clip to the dataset.



Clip Type: Box

☒ Show Box

Translate 0 0 0

Rotate 0 0 0

Scale 1 1 1

☒ Inside Out

☐ Crinkle clip

Figure 5: Screenshot of the `Clip` window in PARAVIEW.

- Other useful filters are the **Threshold**, **Calculator**, **Scatter Plot**, etc.

Saving your results

Save Data
You can generate a table in CSV format containing the values of the fields for each particle for each PART file or filtered dataset. If you click on `File/Save Data`, a window appears where you can set the name and other options for the result file. There are many options for the format of the file, but the recommended one is .csv, as you can visualize it on the Linux LibreOffice Calc or Windows Excel. In addition, you can change the format of the file to .dat in order to open it with a text editor.

Save State
You can save your PARAVIEW postprocessing state in a file by clicking on `File/Save State`. The state file is in ascii format so you can edit it with a text editor. You can also apply it to other datasets than the original one, which is very useful in order to avoid having to repeatedly perform the same filtering operations.