Explanation of the Code

The code structure is derived from a tutorial case called “test_2d_wetting_effects” [5] [6] provided in the official repository. This section of the report provides documentation wherever this code differs from the tutorial case and not the entire structure of the code.

File “wetting.h”

The dimensions of the container are changed to make the path of collision longer. The tank height is increased from 1 m to 10 m.

```
Real DL = 2.0;                    /**< Tank length. */
Real DH = 10.0;                   /**< Tank height. */
```

In the original code, there are two fluids. However, in this case, the second fluid has the same properties as the first fluid. The densities of both the fluids is set to 1.0 kg/m^3.

```
Real rho0_f1 = 1.0; /**< Reference density of the bottom drop of water. */
Real rho0_f2 = 1.0; /**< Reference density of the upper drop of water. */
```

As mentioned before, in order to make the droplets move towards each other, they are given opposite gravitational forces. These values are defined as 1.0 m/s^2 and -1.0 m/s^2.

```
Real gravity_1 = 1.0; /**< Gravity force of the bottom drop of water. */
Real gravity_2 = -1.0; /**< Gravity force of the upper drop of water. */
```

The viscosity of the second fluid is changed to 5.0E-02 Pa s.

```
Real mu_f1 = 5.0e-2; /**< Lower drop viscosity. */
Real mu_f2 = 5.0e-2; /**< Upper drop viscosity. */
```

Instead of the function “createWaterBlockShape”, we now have two functions: “createLowerWaterBlockShape” and “createUpperWaterBlockShape”. This is because, we now need two liquid droplets, and therefore, shapes for each of them. As seen in the code, the first drop is situated at the bottom, while second drop is situated at the top.

```cpp
/** create two water block shapes */
std::vector<Vecd> createLowerWaterBlockShape()
{
    //geometry
    std::vector<Vecd> water_block_shape;
    water_block_shape.push_back(Vecd(0.375 * DL, 0.0));
    water_block_shape.push_back(Vecd(0.375 * DL, 0.35));
    water_block_shape.push_back(Vecd(0.625 * DL, 0.35));
    water_block_shape.push_back(Vecd(0.625 * DL, 0.0));
    water_block_shape.push_back(Vecd(0.375 * DL, 0.0));
    return water_block_shape;
}
std::vector<Vecd> createUpperWaterBlockShape()
{
    //geometry
    std::vector<Vecd> water_block_shape;
    water_block_shape.push_back(Vecd(0.375 * DL, DH-0.35));
    water_block_shape.push_back(Vecd(0.375 * DL, DH));
    water_block_shape.push_back(Vecd(0.625 * DL, DH));
    water_block_shape.push_back(Vecd(0.625 * DL, DH-0.35));
    water_block_shape.push_back(Vecd(0.375 * DL, DH-0.35));
    return water_block_shape;
}
```

In the original code, we have a class called “WaterBlock”. Here, we have two classes: first drop “LowerWaterBlock” and “UpperWaterBlock”.

```
class LowerWaterBlock : public FluidBody
{
public:
    LowerWaterBlock(SPHSystem &sph_system, const string &body_name)
        : FluidBody(sph_system, body_name, makeShared<SPHAdaptation>(1.3, 1))
    {
        /** Geomtry definition. */
        MultiPolygon multi_polygon;
        multi_polygon.addAPolygon(createLowerWaterBlockShape(), ShapeBooleanOps::add);
        body_shape_.add<MultiPolygonShape>(multi_polygon);
    }
};

class UpperWaterBlock : public FluidBody
{
public:
    UpperWaterBlock(SPHSystem &sph_system, const string &body_name)
        : FluidBody(sph_system, body_name, makeShared<SPHAdaptation>(1.3, 1))
    {
        /** Geomtry definition. */
        MultiPolygon multi_polygon;
        multi_polygon.addAPolygon(createUpperWaterBlockShape(), ShapeBooleanOps::add);
        body_shape_.add<MultiPolygonShape>(multi_polygon);
    }
};
```

File “wetting.cpp”

The corresponding “water blocks” are created using the classes defined in the file “wetting.h”. The liquids derive the same properties as defined previously.

```
LowerWaterBlock water_block1(sph_system, "WaterBody1");
FluidParticles water_particles1(water_block1, makeShared<WeaklyCompressibleFluid>(rho0_f1, c_f, mu_f1));

UpperWaterBlock water_block2(sph_system, "WaterBody2");
FluidParticles water_particles2(water_block2, makeShared<WeaklyCompressibleFluid>(rho0_f2, c_f, mu_f2));
```

The body relation map has to be modified since now we have two liquid droplets inside the container, while the rest of the domain is empty. We now have two “ComplexBodyRelations”: “water_water_complex1” and “water_water_complex2”. We also have two “BodyRelationContacts” called “water_wall_contact1” and “water_wall_contact2”.

```
ComplexBodyRelation water_water_complex1(water_block1, {&water_block2});
BodyRelationContact water_wall_contact1(water_block1, {&wall_boundary});
ComplexBodyRelation water_water_complex2(water_block2, {&water_block1});
BodyRelationContact water_wall_contact2(water_block2, {&wall_boundary});
```

Now, we need to define two “Gravity” objects for allocating different gravities to the two droplets. We use the “gravity_1” and “gravity_2” values defined in the file “wetting.h”.

```
/** Define external force. */
Gravity gravity1(Vecd(0.0, gravity_1));
Gravity gravity2(Vecd(0.0, gravity_2));
```

We have two initialize the time step for the two water blocks using the previously defined “Gravity” objects.

```
/** Initialize particle acceleration. */
TimeStepInitialization initialize_a_water_step1(water_block1, gravity1);
TimeStepInitialization initialize_a_water_step2(water_block2, gravity2);
```

We now evaluate the density by summation approach.

```
/** Evaluation of density by summation approach. */
fluid_dynamics::DensitySummationFreeSurfaceComplex
    update_water_density_by_summation1(water_water_complex1.inner_relation_, water_wall_contact1);
fluid_dynamics::DensitySummationFreeSurfaceComplex
    update_water_density_by_summation2(water_water_complex2.inner_relation_, water_wall_contact2);
```

The time step size and pressure and density relaxation is calculated for both fluids.

```
/** Time step size with considering sound wave speed. */
fluid_dynamics::AcousticTimeStepSize get_water_time_step_size1(water_block1);
fluid_dynamics::AcousticTimeStepSize get_water_time_step_size2(water_block2);
/** Pressure relaxation for water by using position verlet time stepping. */
fluid_dynamics::PressureRelaxationRiemannWithWall
    water_pressure_relaxation1(water_water_complex1.inner_relation_, water_wall_contact1);
fluid_dynamics::PressureRelaxationRiemannWithWall
    water_pressure_relaxation2(water_water_complex2.inner_relation_, water_wall_contact2);
fluid_dynamics::DensityRelaxationRiemannWithWall
    water_density_relaxation1(water_water_complex1.inner_relation_, water_wall_contact1);
fluid_dynamics::DensityRelaxationRiemannWithWall
    water_density_relaxation2(water_water_complex2.inner_relation_, water_wall_contact2);
/** Viscous acceleration. */
fluid_dynamics::ViscousAccelerationMultiPhase
    water_viscous_acceleration1(water_water_complex1);
fluid_dynamics::ViscousAccelerationMultiPhase
    water_viscous_acceleration2(water_water_complex2);
```

In the original code, only one of the fluids was a liquid. However, now both are liquids. So, while calculating the surface tension and wetting effects, we need to consider both of the fluids.

```
/** Suface tension and wetting effects. */
fluid_dynamics::FreeSurfaceIndicationComplex
    surface_detection1(water_water_complex1.inner_relation_, water_wall_contact1);
fluid_dynamics::FreeSurfaceIndicationComplex
    surface_detection2(water_water_complex2.inner_relation_, water_wall_contact2);
```

Now, we compute the color gradient for both of the complex body relations.

```
fluid_dynamics::ColorFunctionGradientComplex
    color_gradient1(water_water_complex1.inner_relation_, water_wall_contact1);
fluid_dynamics::ColorFunctionGradientComplex
    color_gradient2(water_water_complex2.inner_relation_, water_wall_contact2);
fluid_dynamics::ColorFunctionGradientInterplationInner
    color_gradient_interpolation1(water_water_complex1.inner_relation_);
fluid_dynamics::ColorFunctionGradientInterplationInner
    color_gradient_interpolation2(water_water_complex2.inner_relation_);
```

Similarly, while computing the surface tension acceleration and the wetting norm we need to consider both fluids instead of one.

```
fluid_dynamics::SurfaceTensionAccelerationInner
    surface_tension_acceleration1(water_water_complex1.inner_relation_, tension_force);
fluid_dynamics::SurfaceTensionAccelerationInner
    surface_tension_acceleration2(water_water_complex2.inner_relation_, tension_force);
/** Wetting effects. */
fluid_dynamics::SurfaceNormWithWall
    wetting_norm1(water_wall_contact1, contact_angle);
fluid_dynamics::SurfaceNormWithWall
    wetting_norm2(water_wall_contact2, contact_angle);
```

As described above, we modified the body relation map. So, we now update the water blocks, complex body relations and body relation contacts, if the starting time is not zero.

```
if (sph_system.restart_step_ != 0)
{
    GlobalStaticVariables::physical_time_ = restart_io.readRestartFiles(sph_system.restart_step_);
    water_block1.updateCellLinkedList();
    water_block2.updateCellLinkedList();
    water_water_complex1.updateConfiguration();
    water_wall_contact1.updateConfiguration();
    water_water_complex2.updateConfiguration();
    water_wall_contact2.updateConfiguration();
}
```

The runtime of the simulation is changed to 30 s to simulate and visualize the collision in detail.

```
Real End_Time = 30.0;          /**< End time. */
```

Now we consider the main loop. We first initialize the time steps for both the fluids.

```
initialize_a_water_step1.parallel_exec();
initialize_a_water_step2.parallel_exec();
```

We now calculate the advection time steps for the fluids.

```
Real Dt_f1 = get_water_advection_time_step_size1.parallel_exec();
Real Dt_f2 = get_water_advection_time_step_size2.parallel_exec();
```

Now, we update the densities of the fluids. Since both fluids in our case are liquids, the transport correction is not applied.

```
update_water_density_by_summation1.parallel_exec();
update_water_density_by_summation2.parallel_exec();
```

We now apply the viscous acceleration for both the fluids.

```
water_viscous_acceleration1.parallel_exec();
water_viscous_acceleration2.parallel_exec();
```

Since we created functions for surface detection, color gradient, wetting norm and surface tension acceleration for both of the fluids, all of these functions are run. The corresponding code is as follows.

```
surface_detection1.parallel_exec();
surface_detection2.parallel_exec();
color_gradient1.parallel_exec();
color_gradient2.parallel_exec();
color_gradient_interpolation1.parallel_exec();
color_gradient_interpolation2.parallel_exec();
wetting_norm1.parallel_exec();
wetting_norm2.parallel_exec();
surface_tension_acceleration1.parallel_exec();
surface_tension_acceleration2.parallel_exec();
```

We now calculate time step size and apply the relaxation for pressure and density.

```
while (relaxation_time < Dt)
{
    Real dt_f1 = get_water_time_step_size1.parallel_exec();
    Real dt_f2 = get_water_time_step_size2.parallel_exec();
    dt = SMIN(SMIN(dt_f2, dt_f2), Dt);

    water_pressure_relaxation1.parallel_exec(dt);
    water_pressure_relaxation2.parallel_exec(dt);

    water_density_relaxation1.parallel_exec(dt);
    water_density_relaxation2.parallel_exec(dt);

    relaxation_time += dt;
    integration_time += dt;
    GlobalStaticVariables::physical_time_ += dt;
}
```

We now update the linked lists for the water blocks. We update the configuration of the modified complex body relations and body relation contacts.

```
water_block1.updateCellLinkedList();
water_water_complex1.updateConfiguration();
water_wall_contact1.updateConfiguration();

water_block2.updateCellLinkedList();
water_water_complex2.updateConfiguration();
water_wall_contact2.updateConfiguration();
```

Output

The simulation is run for 30 seconds. The drops, which were initially at rest, are accelerated towards each other. The drops finally collide and scatter into smaller particles. Since we applied a different gravitational force to each water block, the particles belonging to the lower block move towards the lower end of the container, and those belong to the upper block, move up. This is visible in Fig. 4.

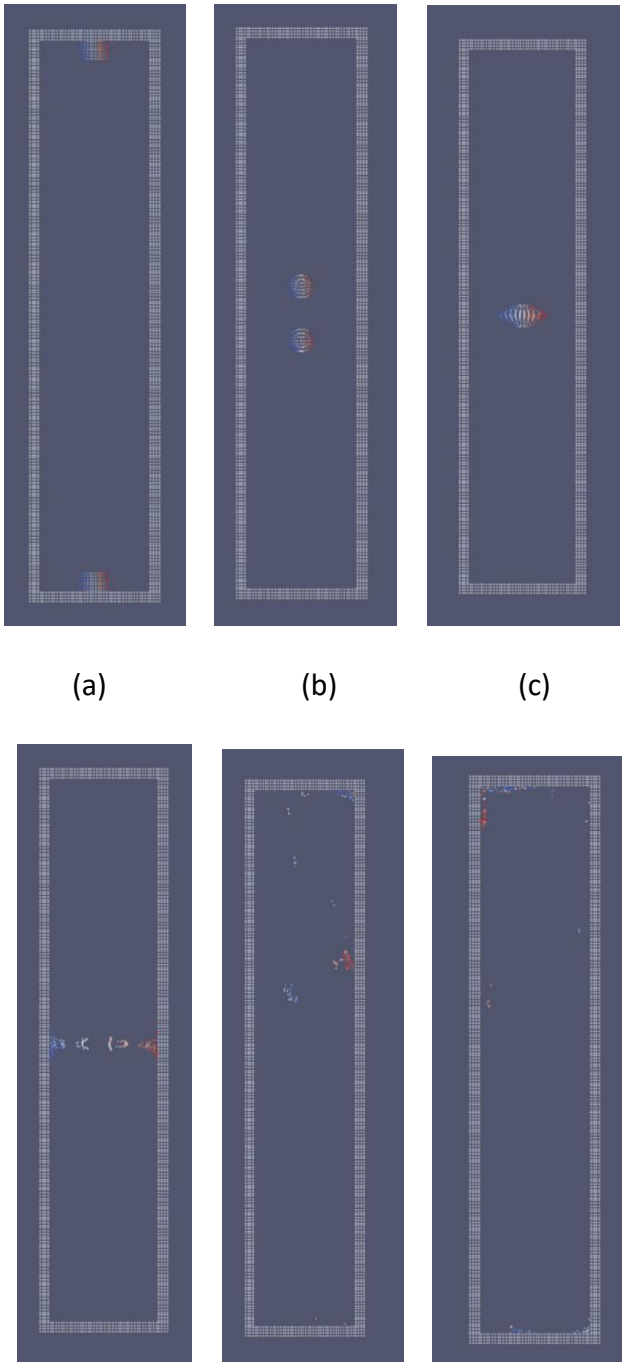

Fig. 4. The snapshots for the output visualized in Paraview.