Documentation

ness

February 28, 2020

Contents

This tool has been implemented in C++ with Python as a means to analyze data and test the output of the code. Bellow is the documentation of the functions used in the tool and an analysis of the testing pipline.

1 Code structure, Class variables and functions

The tool is composed of two classes *SPH_particle* and *SPH_main*. *SPH_particle* is an object that represents each individual particle in our simulation and holds parameters and values that are specific to every particle (eg. position, velocity, etc). *SPH_main* represents the domain the simulation is using and holds all the functions, globals and data strucutres needed to run the simulation.

1.1 *SPH_particle*

1.1.1 Class variables:

x[] (double) Particle position variables. This array has two entries, one for the x-direction and one for the y-direction.

v[] (double) Particle velocity variable. This array has two entries, one for the x-direction and one for the y-direction.

rho (double) Particle density variable.

P (double) Particle pressure variable.

a[] (double) Particle acceleration variable. This array has two entries, one for the x-direction and one for the y-direction.

D (double) Particle rate of change of density variable.

rho2 (dobule) Particle density variable. Used in the *density_field_ smoothing* function.

vij_half[] (double) The largest difference of velocities between one particle and any of its neighbours.

x_half[] (double) Particle half-step position variable. This is an array that stores the position of the particle after a half step in the second order time-stepping method. This array has two entries, one for the x-direction and one for the y-direction.

v_half[] (double) Particle half-step velocity variable. This is an array that stores the velocity of the particle after a half step in the second order time-stepping method. This array has two entries, one for the x-direction and one for the y-direction.

rho_half (double) Particle half-step density variable. This is a variable that stores the density of the particle after a half step in the second order time-stepping method.

a_half[] (double) Particle half-step acceleration variable. This is an array that stores the acceleration of the particle after a half step in the second order time-stepping method. This array has two entries, one for the x-direction and one for the y-direction.

D_half (double) Particle half-step rate of change of density variable. This variable stores the rate of change of density after a half step in the second order time-stepping method.

numerator (double) Is (literaly) the numerator from density smoothing function.

denominator (double) Is (literaly) the denominator from the density smoothing function.

main_data (staic SPH_main *) Link to SPH_main class so that it can be used to calc_index

list_num[] (int) Index in neighbour finding array.

is_boundary (bool) Set true if the particle is part of the boundary

1.1.2 Class functions:

calc_index(*void*) (void) This function is responsible for populating the *list_index[]* array.

1.2 *SPH_main*

1.2.1 Class variables:

h (double) Smoothing length

h_fac (double)

dx (double) Particles initial spacing. This is a variable that is responsible for the space between neighbouring particles when the initial configuration is set.

c0 (double) Speed of sound. As this is a simple simulation, a different speed of sound has been implemented.

dt (double) Timestep. This is a variable that defines the size of the timestep in the time-stepping solution.

g[] (double) Gravity constant array. This array stores the initial accelaration values that describe our sytem. In this case it is only gravity and it points in the negative y-direction.

mu (double) Viscocity variable. This variable holds the viscocity value of our system.

rho0 (double) Initial density variable. This variable stores the initial density value for all the particles.

B (double) Initial, or reference density of the fluid, given by the formula:

$$B = \frac{\rho_o c_o^2}{\gamma} \tag{1}$$

gamma (double) A constant used in a number of formulas throughout the code. The value is given as 7.

mass (double) Mass variable. This variable stores the mass of each particle in the simulation.

For dynamic time stepping

v_max (double) Maximum velocity of all particles in one iteration.

a_max (double) Maximum acceleration of all particles in one iteration.

rho_max (double) Maximum density of all particles in one iteration.

dt_cfl (double) Constant used in the dynamic time-stepping, according to the formula:

$$\Delta t_{CFL} = min\Big\{\frac{h}{|v_{ij}|}\Big\} \tag{2}$$

dt_f (double) Constant used in the dynamic time-stepping, according to the formula:

$$\Delta t_F = min\Big\{\sqrt{\frac{h}{|a_i|}}\Big\} \tag{3}$$

dt_a (double) Constant used in the dynamic time-stepping, according to the formula:

$$\Delta t_A = min\Big\{\frac{h}{c_o\sqrt{(\rho/\rho_o)^{\gamma-1}}}\Big\} \tag{4}$$

cfl (double) A constant used to scale the constants above. Its range should be between 0.1 - 0.3.

min_x[], max_x[] (double) Dimensions of simulation region

**grid_count (vector<vector<int») ** Decrements the total number of neighbour particles we need to saerch through.

max_list[] (int) Maximum index of grid.

particle_list (vector<SPH_particle>) List of all the particles

search_grid (<vector<vector<vector<SPH_particle*> > >) Outer two vectors are the grid, inner vector is the list.

1.2.2 Class functions:

SPH_main() Main constructor.

cubic_spline(*double r[]*) (double) Cubic Spline calculation function.

- *r[]* contains the distance between two points.

Calculates the cubic spline according to three cases:

$$W(r,h) = \frac{10}{7\pi h^2}\begin{cases} 1 - \frac{3}{2}q^2 + \frac{3}{4}q^3 & \text{if } 0 \leq q \leq 1 \\ \frac{1}{4}(2-q)^3 & \text{if } 1 \leq q \leq 2 \text{ Where } q = \dfrac{r}{h} \\ 0 & \text{if } q > 2 \end{cases} \tag{5}$$

cubic_spline_first_derivative(*double r[]*) (double) Cubic Spline First Derivative calculation function.

- *r[]* contains the distance between two points.
  Calculates the cubic spline according to three cases:

$$\nabla W(r,h) = \frac{10}{7\pi h^3} \begin{cases} -3q + \frac{9}{4}q^2 & \text{if } 0 \le q \le 1 \\ -10\frac{3}{4}(2-q)^2 & \text{if } 1 \le q \le 2 \text{ Where } q = \dfrac{r}{h} \\ 0 & \text{if } q > 2 \end{cases} \tag{6}$$

update_gradients(*double r[], SPH_particle* part, SPH_particle* other_part*) (void)
Updating the values of rate of change of speed (acceleration) and density.

- *part* is a pointer to the particle that we calculate the change for.
- *other_part* is a pointer to the neighbouring particle used in the calculation. The calculations are done according to the following functions:

$$\alpha_i = -\sum_{j=1}^{N} m_j \left(\frac{P_i}{\rho_i^2} + \frac{P_j}{\rho_j^2}\right) \frac{dW}{dr}(r_{ij},h) e_{ij} + \sum_{j=1}^{N} m_j \left(\frac{1}{\rho_i^2} + \frac{1}{\rho_j^2}\right) \frac{dW}{dr}(r_{ij},h) \frac{v_{ij}}{r_{ij}} \tag{7}$$

$$D_i = \sum_{j=1}^{N} m_j \frac{dW}{dr}(r_{ij},h) v_{ij} \cdot e_{ij} \tag{8}$$

density_field_smoothing(*SPH_particle* part*) (void) Smoothing out the Density/Pressure field. This is done because the integration of the continuity equation will result in variations in density and pressure based on the macroscopic velocity gradients, but also on local variations in particle spacing and velocity; resulting in a “rough” density and pressure distributions. To get around this problem, we implemented a density smoother.

- *part* is a pointer to the particle whose density is to be smoothed.

$$\rho_i = \frac{\sum_{j=1}^{n} W(r_{ij},h)}{\sum_{j=1}^{n} \frac{W(r_{ij})}{\rho_j}} \tag{9}$$

set_values(*double delta_x*) (void) Setting simulation parameters.

- *delta_x* the distance between particles in the initial configuration.

initialize_grid(*void*) (void) This function initializes the search grid used to find neighbours. This is done by dividing the points into cell grids of size double the initial distance between particles and allocating them to the corresponding cell.

place_points(*double min0*, *double min1*, *double max0*, *double max1*, *bool type*) (void)
This function is responsible for initializing the points to the domain, by setting all the variables (as defined in the *SPH_particle* variables).

- *min0* minimum position value for the x-direction.
- *min1* minimum position value for the y-direction.
- *max0* maximum position value for the x-direction.
- *max1* maximum position value for the y-direction.
- *type* boolean that defines weather or not the particle is on the boundary.

allocate_to_grid(*void*) (void) Allocates all the points to the search grid (assumes that index has been appropriately updated). This function is called in every iteration step, as the movement of the particles might change their corresponding search grid cell.

neighbour_iterate(*SPH_particle* part*) (void) The main function that searches the search grid to find the corresponding neighbours for the given particle. Iterates through a particle's neighbours within 2h with a stencil scheme, and update the gradients for pairs of particles. This function is called twice in one iteration (one per half-step).

- *part* the particle for which the neighbours are searched.

update_particle(*SPH_particle* part*) (void) Function that updates the position, velocity, density and pressure, according to the results of the *update_gradients* functions.

- *part* the particle for which the variables are updated.

reset_grid_count() (void) This function resets the grid count of the domain after every iteration.

update_rho(*SPH_particle* part*) (void) This function updates the density of the particle after each half and whole timestep.

- *part* the particle for which the density is updated.

store_initial(*SPH_particle* part*) (void) This function stores the initial velocity, density and position for the second order time-stepping scheme.

- *part* input particle to store

time_dynamic() (void) Finds the timestep for next iteration by finding the minimum of three other timesteps calculated using the maximum velocity, density and acceleration. This function is called in every iteration.

full_update(*SPH_particle* part*) (void) The final update of particle for each iteration using the second order method.

- *part* particle to receive full update.

get_new_max(*SPH_particle* part*) (void) Gets the new global maximum velocity acceleration and density.

- *part* particle to get new max from.

repulsion(*SPH_particle* part*, *double dist*) (double) Calculate the repulsive force to a particle by the boundary.

- *part* particle to apply the repulsive force to.
- *dist* distance between the particle and the nearby boundary.

update_particle_FE(*SPH_particle* part*) update function for the first order method.

2 Parallelization with OpenMP

OpenMP was used to parallelize the computation of the simulation. The *#pragma omp for loops* exist on the *SPH_Snippet*, parallelizing the looping over all particles. Since our *update_particle* and *neighbout_iterate* functions operate on two particles in the same time, the *#pragma omp atomic* directive was added at the parts of the code where we update the particle values, to ensure that no more than one process accesses and edits them at one time.

3 Running the Simulation, Post-processing and Output scripts

A sample C++ file (*SPH_Snippet.cpp*) has been provided in the package that runs the simulation for the parameters required for the class excericse. That file is responsible for the entirety of the simulation and serves as a template for any future simulations anyone would want to run using this tool. Moreover, a number of post processing scripts have been implemented in C++ and Python for the purpose of outputing the simulation states in a suitable format for both visualization and data manipulation.

3.1 file_writer.cpp

A simple C++ file that outputs simulation steps as *.vtp* files; to be used with ParaView or other software that is build upon the *VTK* library.

3.2 post.py

A simple Python script that takes the *.vtp* files created by *file_writer.cpp*, creates a Pandas DataFrame for every iteration step and outputs them in a HDF5 file (for easy data transport and data manipulation). A similar version of the script is used in the testing pipeline in the step where the testing moves from C++ to Python.

4 Testing

Testing on this tool is done by both C++ and Python. For C++ the BOOST library is used and for Python a custom test file has been written.

4.1 C++ Testing

The C++ side of the testing handles all the mathematical functions defined in the *SPH_main* class. Namely *cubic_spline*, *cubic_spline_first_derivative* and *update_gradients*. A set of BOOST test cases has been set that depends on the possible outputs of the spline functions. Note, that the same principle is applied for *update_gradients* as the cubic spline functions play an important role in the calculation of acceleration and rate of change of pressure. Moreover, the tests are conducted independently of the model parameters (as they are defined in the *set_vales* functions), by setting the desired parameters before the function call.

4.2 Python Testing

The Python part of the testing deals with validating the behaviour of the simulator; checking that the particles stay within the boundaries and that the particles are present at their proper positions after N iterations steps. This part of the testing can be used for validating that the input parameters by the user fall within the power of the simulator (ie that the simulator does not become unstable).

4.3 Testing pipeline

The teting pipeline goes as follows:

1. The objects and C++ classes get compiled.
2. The Python script *run_tests.py* is being run, which is responsible for running both the C++ and Python tests.
3. The C++ test file *test_SPH_2D.cpp* is run, followed by *test_file_writer.cpp* which outputs a set of *.vtp* files, containing the iteration steps of the test simulator.
4. *test_post.py* is called (a specialised version of *post.py*) which processes the *.vtp* files and creates the *output_test.h5* file, containing all the Pandas DataFrames to be used by the Python part of the tests.
5. *python_tests.py* is run with *output_test.h5* as input; completing a number of tests on the behaviour of the simulator.

Note: The following commands are to be entered from the base repository of the package. The complete testing pipeline can be run by using:

```
make runtests
```

To run only the C++ tests you can use:

```
./tests/bin/test_SPH_2D
```

The *–log_level=unit_scope* flag can be used to give a detailed report of the testing suite and *–list_content* provides the user for a description of each testing case for easier debugging.

To produce the *.vtp* files you can use:

```
./tests/bin/test_file_writer
```

To produce the *.h5* test files you can use:

```
python ./tests/test_post.py
```

(It goes without saying that the *.vtp* files need to already exist for the *.h5* files to be produced.)

5 Arbitrary Boundaries (beta)

This is a separate scheme for arbitrary convex polygon shaped domains. To use it, substitute the relevant files in the source file with the ones under arbitrary/. It applies Forward Euler time-stepping and direct push back boundary mechanism. Its member functions are mostly the same with the original scheme, with the following differences:new attributes in the SPH_main class:

- vector<vector<double» boundaries, vertices, areas:

stores information of boundaries and vertices of the domain polygon and rectangular areas containing moving particles.

- set<vector<int» near_boundary:

stores the indices of grids that are near the boundary.

- int vcount:

the number of vertices.is_inside: This function decides if a boundary particle lies inside the domain polygon. It iterates through each boundaries, performs cross product on the vectors from boundary endpoints to the particle to determine which side it lies on.place_points_new: For boundary particles (type = 1), the first 4 arguments describe the coordinates of the two end points of a line. Boundary particles will be placed every dx on the line, and 3 more lines of particles will be placed outside, parallel to this line each at a distance of dx. For moving particles (type = 0), the first 4 arguments still define the bottom-left and top-right points of a rectangle. The particles in the rectangle will be placed if its inside the polygon (determined using is_inside)