Smoothed Particle Hydrodynamics

A Technical Report by Team Atlantic:

Oliver Boom

Xianzheng Li

Keer Mei

John Walding

Yujie Zhou

Introduction

Team Atlantic have created a simulation program that uses a Smoothed Particle Hydrodynamics (SPH) method to solve the Navier-Stokes problem. The problem begins with a column of water simulating a dam break scenario. The simulation then calculates the position, velocity, pressure, and density for each fluid particle until a run time of 30 seconds.

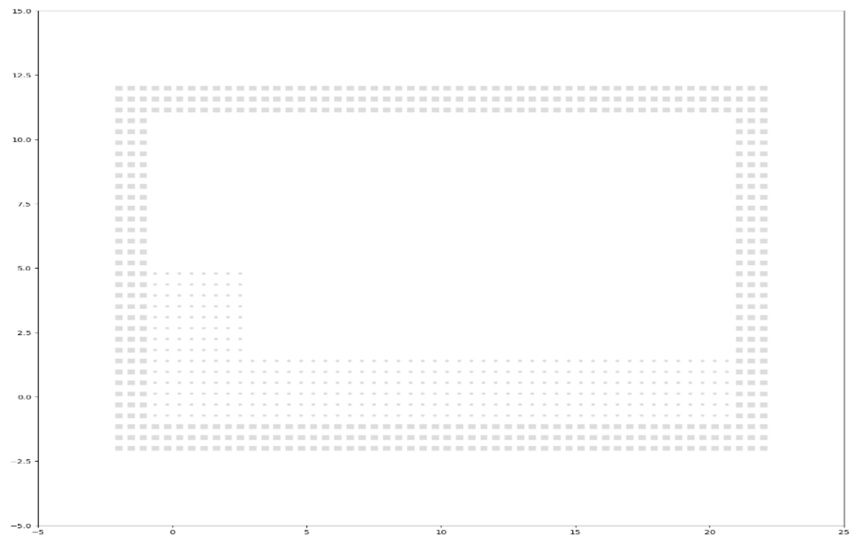


Figure 1. Initial simulation plot. Color scaled based on velocity

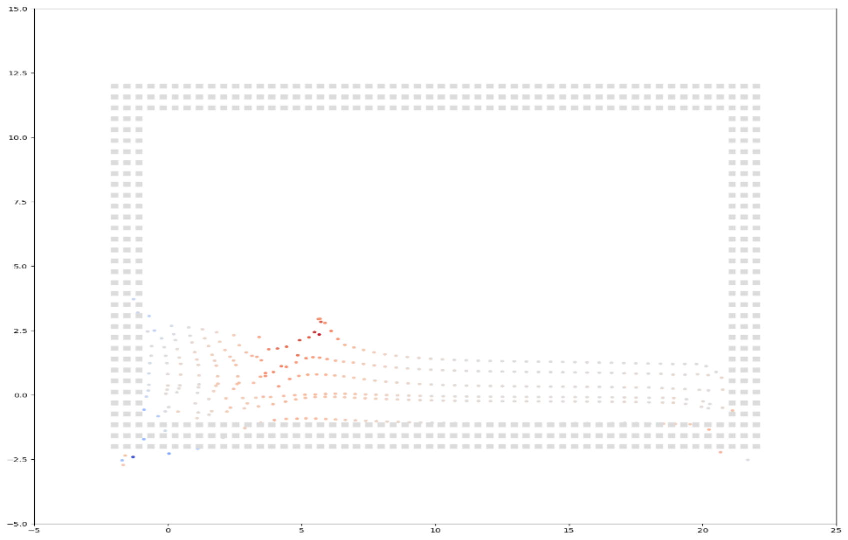
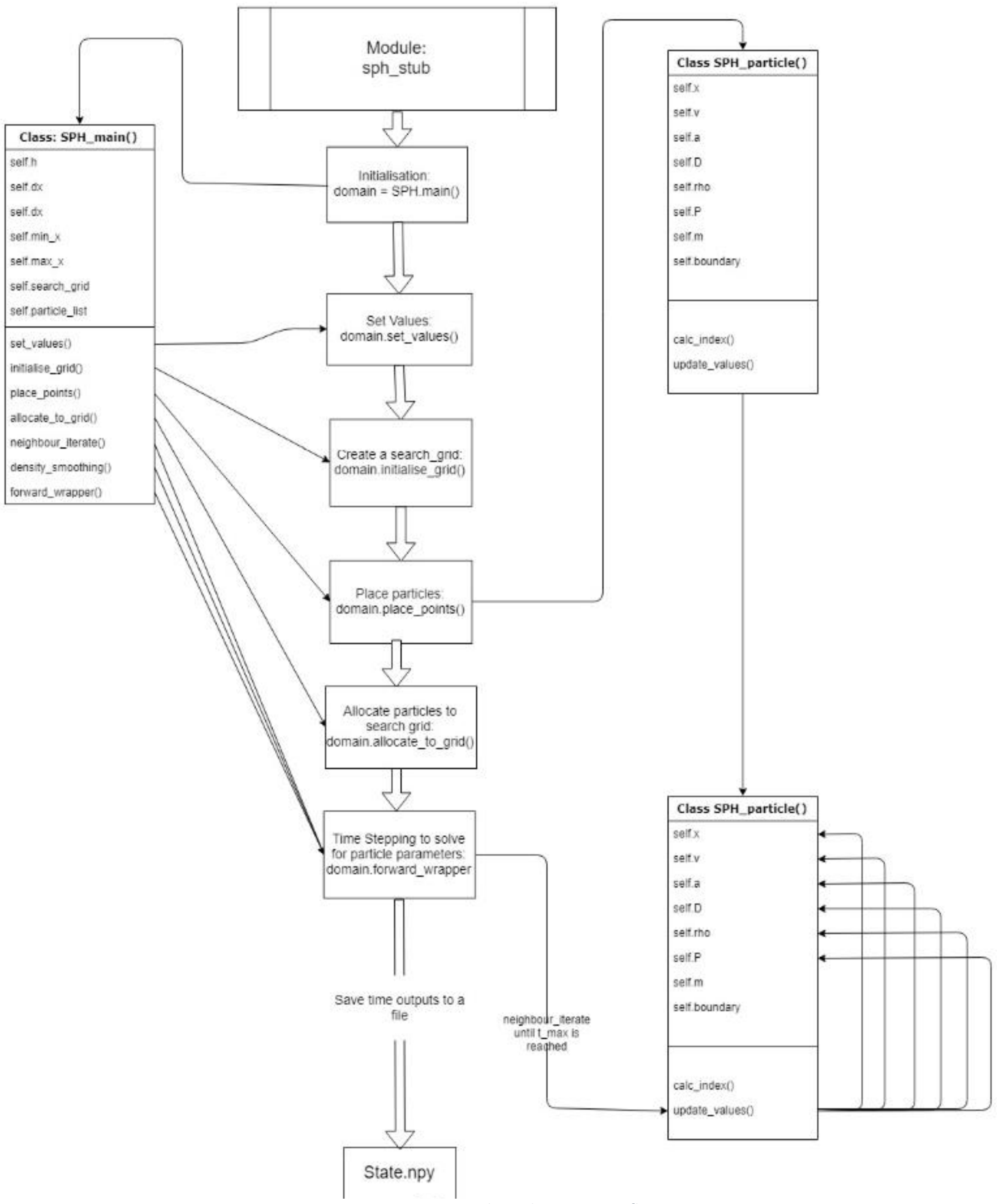


Figure 2. Simulation run at 200 time steps. Color scaled based on velocity

The simulation program includes a front end user interafce, a numerical simulator and a post-processor to create animations. The front end interface allows users to interact with the simulator. The simulator produces the particle parameters at each time step and outputs a ‘State.npy’ file into the user’s local directory. The post-processor loads the ‘State.npy’ file from the local directory and is able to produce an animation of particle locations and their physical parameters including velocity, pressure and density at each time step.

Algorithm Flowchart



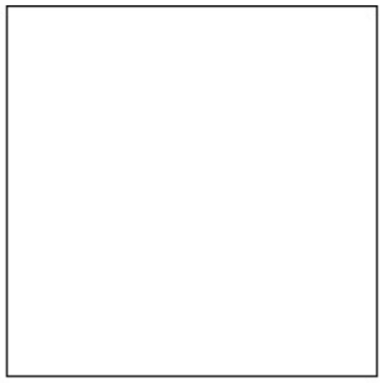
Module: sph_stub
Class: SPH_main()
self.h
self.dx
self.dx
self.min_x
self.max_x
self.search_grid
self.particle_list
set_values()
initialise_grid()
place_points()
allocate_to_grid()
neighbour_iterate()
density_smoothing()
forward_wrapper()
Initialisation: domain = SPH.main()
Set Values: domain.set_values()
Create a search_grid: domain.initialise_grid()
Place particles: domain.place_points()
Allocate particles to search grid: domain.allocate_to_grid()
Time Stepping to solve for particle parameters: domain.forward_wrapper
Save time outputs to a file
State.npy
neighbour_iterate until t_max is reached
Class SPH_particle()
self.x
self.v
self.a
self.D
self.rho
self.P
self.m
self.boundary
calc_index()
update_values()

Figure 3. Numerical simulator process flow

The simulator module executes a series of commands to create the numerical solutions to the SPH problem. Each module execution step is explained in detail below. The final output of the simulator is a 'State.npy' file which can then be read into a post-processing module to create an animation.

Algorithm Pseudo Code Description

Step 1. Create domain class: SPH_main()



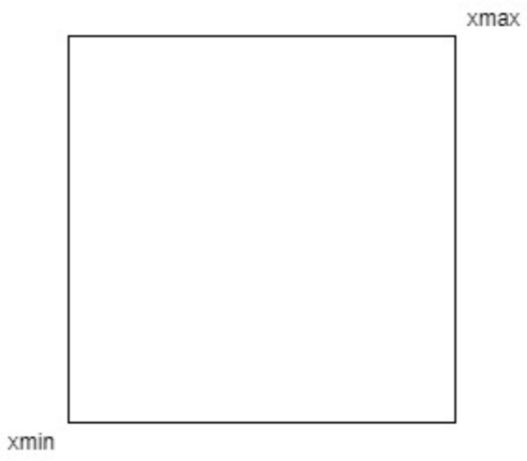
Description:

The program is initialised by creating an Object class called SPH_main(). This object contains the following attributes:

- self.h – a grid spacing
- self.h_factor – a factor for the grid spacing
- self.dx – an interparticle spacing
- self.min_x – the lower bound of the domain
- self.max_x – the upper bound of the domain
- self.max_list – the size of a search grid that will be created later
- self.particle_list – an array that contains all the particle objects

Inside the class SPH_main(), functions are called in order to manipulate the particle objects.

Step 2. Create domain parameters: set_values()

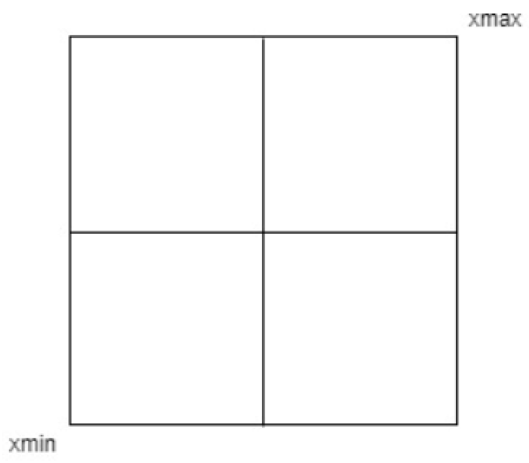


Description:

The first function, SPH_main.set_values() explicitly states the initial conditions of the domain. It provides the upper and lower bounds, the intergrid and interparticle spacing.

A reference density (rho0), fluid speed of sound (c0), as well as the final domain time (t_max) are provided to the simulation.

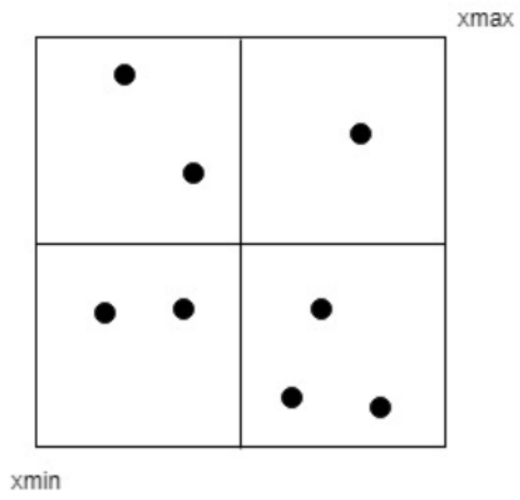
Step 3. Initialise the grid: initialise_grid()



Description:

Initialise grid creates an attribute array called self.search_grid[]. Self.search_grid[] is used when time-stepping each particle parameter and the contributions of the neighbours are quantified. The search_grid() provides the locations of all nearby grid points to the current particle.

Step 4.Create a particle list: place_points()



Description:

Two arrays, x and y , are created given the upper and lower bound of the domain. Starting from the lower left corner, iterating at a value of dx, a particle is created using the coordinate locations by calling the SPH_particle() class. The SPH_particle() is an object with the following attributes:

- self.x – particle location in (x, y) coordinates
- self.v – particle velocity in (x, y) directions
- self.a – particle acceleration in (x, y) directions
- self.D – the particle's instantaneous change in density with respect to time
- self.rho – particle density
- self.P –particle pressure

An index relative to the search grid is created for each particle. Each particle is then appended into the domain's particle list.

Pseudo code:

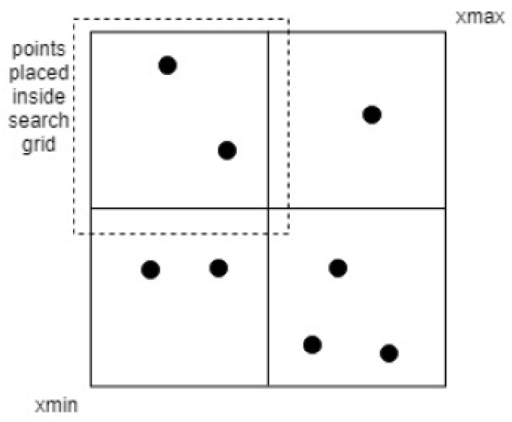
```
• for i, x in enumerate(x_arr):
    o for j, y in enumerate(y_arr):
        ▪ particle = SPH_particle(self, np.array([x,y])
            • particle.calc_index()
            • self.particle_list.append(particle)
```

Padding:

In the function place_points() is the ability to create dynamic boundaries. If geometry='default', place_points() allocates three particles to a boundary layer which surrounds an initial wave formation as described by the problem statement. These boundary particles are stored inside an array called initial[i, j].

Once the boundary particles are in defined, the fluid particles are then appended into the particle_list[] within

Step 5. Allocate particles to grid: allocate_to_grid()



Description:
For each particle in the particle list, they are assigned a search_grid() value. This search_grid() value will be used in the neighbour_iterate() function.

Step 6. Time stepping: forward_wrapper()

Description:
Forward_wrapper() is the function which updates the values of the particle objects inside the particle_list[] at a time step.

Pseudo-code:

```
• While t < domain.t_max:
    o pickle.dump(self.particle_list, file))
    o pickle.load(file)
    o append(pickle.load(file))
    o If interval = 20:
        ▪ For particle in particle_list[]:
            • domain.density_smoothing(particle) # this function is used to resolve
              any density fluctuations after every 20 time steps
    o For particle in particle_list[]:
        ▪ domain.neighbour_iterate(particle) # this function looks at all the neighbours
          within the search_grid of the particle and calculates the derivative
          approximations to the navier-stokes equations.
    o For particle in particle_list[]:
        ▪ particle.update_values(particle) # this function updates the attributes of the
          particle using the derivatives calculated in the neighbour_iterate() function from
          the previous step
    o t  = t + dt
```

Dynamic Time-stepping

Initially, forward_wrapper() simulated a fixed time step at a value proportional to the ratio of h and the numerical speed of sound. Dynamic time steps were later incorporated, which will be discussed after the pseudocode.

Pseudo-code:

```
• While t < domain.t_max:
    o pickle.dump(self.particle_list, file))
    o pickle.load(file)
    o append(pickle.load(file))
    o If interval = 20:
        ▪ For particle in particle_list[]:
            • domain.density_smoothing(particle) # this function is used to resolve
              any density fluctuations after every 20 time steps
    o For particle in particle_list[]:
```

        - domain.neighbour_iterate(particle) # this function looks at all the neighbours within the search_grid of the particle and calculates the derivative approximations to the navier-stokes equations.
    - For particle in particle_list[]:
        - particle.update_values(particle) # this function updates the attributes of the particle using the derivatives calculated in the neighbour_iterate() function from the previous step
    - t = t + 0.3*minimum of (t_cfl, t_A, t_f) where t_cfl, t_A and t_f are based on the current particle properties, particle.a (acceleration) and particle.rho (density)

A dynamic time step was included and the time step was taken as a minimum of the time steps appropriate that adhere to the CFL condition and conditions relating to the maximum force per unit mass on a particle and the divergence of the velocity.

The dynamic time step should result in higher dt values, resulting in a faster running simulation. However, it is a time step that is set by trying to adhere to stability conditions. Hence, for simulations that are unstable then the time step ends up being smaller than the initially set 'safe' time step. If the simulation starts displaying instabilities, as frequently occurred in testing, then the simulation slowed down significantly (almost an order of magnitude lower than the initially fixed time step).

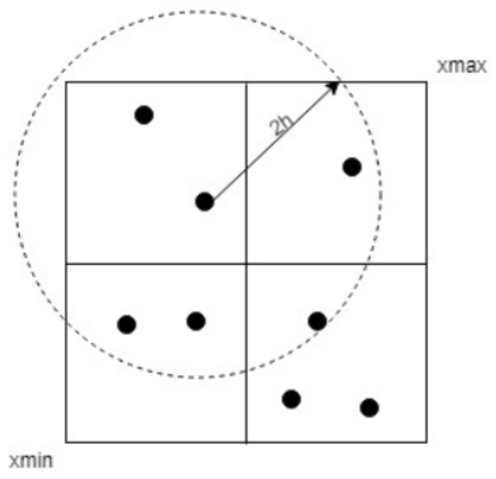
This was circumvented by setting an additional condition that meant that if the dynamic time step was lower then the fixed initial time step, then the fixed initial time step was used. This is effectively using a time step that by definition is unstable, but as longer convergence plots were required to see how behavior unfolded, it was deemed a necessary decision in testing.

Density smoothing

At regular, the density is smoothed to assist in stability. As the pressure field is extremely coarse, several smoothing intervals were considered and it was established that results improved and performance degraded by decreasing the intervals.

This is because smoothing inherently fights instability but adds extraneous computational overhead.

Step 6.1 Neighbour Iteration: neighbour_iterate(self, part)



Description:

The neighbour_iterate(particle) function is the main algorithm used to find the neighbouring particles to the current particle. The current particle's parameters will be updated based on a smoothing kernel used to approximate the derivatives in the Navier-Stokes equation.

Only neighbouring particles within a 2h grid space to the current particle have a contribution on the current particle. The relative distance to all particle's in the neighbouring search_grid[] is calculated from the current particle. The magnitude of the relative distance enables the calculation for the smoothing kernel and the approximate derivatives.

Pseudo-code:

```
• for index1 in range of 1 unit left and 1 unit right of particle.list_num
    o for index2 in range of 1 unit above and 1 unit below particle.list_num
        ▪ for other particle in search_grid[index1, index2]
            • if particle is not other particle:
                o calculate relative distance r_ij
                o calculate magnitude of relative distance mag_r_ig
                o if mag_r_ij < 2*h:
                    ▪ calculate particle derivatives
```

Boundary Condition Forcing

For an additional boundary forcing term the Lennard-Jones potential was used at the boundaries. The pseudocode for this is as follows.

Pseudo-code:

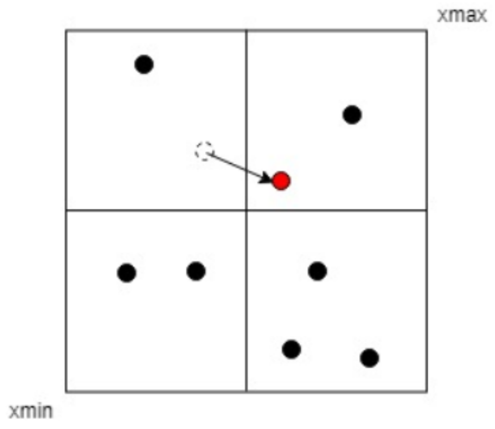
```
• if distance to other particle < 2h and other particle is boundary:
    o particle can_see_wall is set to true and add to list of near wall particles
• iterate through list of near wall particles:
    o calculate perpendicular distance to each wall by dot product of unit normal vector for
      each wall and distance to the wall
```

    - Then a q distance term was allocated based which was 0.9 * dx. So if the particle is under one resolution size of the boundary then it is given an extra acceleration factor, to simulate the boundary.
    - Scale this term with Lennard Jones potential to find additional acceleration

The Lennard Jones potential was calcalated for the default geometry, but was not calculated for the non-uniform geometry. In order to calcalate for non-uniform geometry the unit normal to each wall should be taken into account and the distance calculated from each wall as before. However there should be additional conditions with respect to when the unit normal should be calculated based on position.

For sustainability and readability, this function (neighbour_iterate) has been split down into neighbour_iterate, calc_aD, adaptive and wall_forcing.

Step 6.2 Updating Values: update_values(self, B, rho0, gamma, dt)



Description:

updates_values() is a function within the SPH_particle() class. It updates the particle's attribues self.x, self.v, self.rho and self.P based on the derivatives calculated in the neighbour_iterate(particle) function.

Back-stopping

In a scenario where particles are leaking outside the boundary, a numerical back stop was implemented. This means that when the coordinates of the fluid particles in the next time step exceed the domain limits, then the velocity of the particles are hard reset to 0 with a bounce factor to the normal of the boundary.

Pseudo-Code

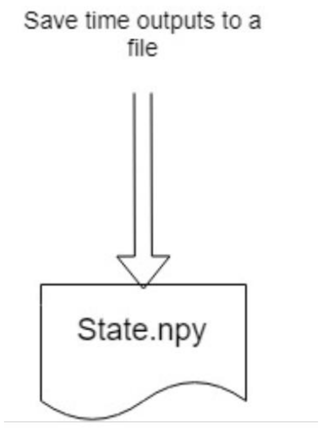
```
• particle.new_location = location + velocity * dt
    o if new location is outside boundary:
        ▪ velocity = bounce factor * velocity
• particle.new_location = particle.old_location + velocity * dt
```

In situations when particle densities exceed an expected range, a numerical back stop was implemented to reset the particle density. This is done to ensure that the particles do not become unstable due to instability elsewhere in the simulation.

Pseudo-Code

```
if particle.density < 400:
    particle.density = 400
if particle.density > 1500:
    particle.density = 1500
```

Step 6.3 Pickling time solutions



Description:

To log the particle parameters at each different time step, the entire particle_list[] at each time step are pickled, loaded and then appended into a list. The list is then saved into a file on the local directory called 'State.npy'.

The dumping and loading of the particles at each time step ensures that the dynamic objects stored inside the particle_list[] at that specific time do not update themselves when update_values() is called later on within the forward_wrapper() loop.

Installation instructions

For the end user, the following additional modules are required to be installed for the simulatior and post-processor to function:

- matplotlib.pyplot, matplotlib >= 2.2.2
- matplotlib.animation , matplotlib >= 2.2.2
- numpy >= 1.14.3
- pickle

To run the numerical simulator, download the module *sph.py* from Team Atlantic's GitHub repository found at: https://github.com/msc-acse/acse-4-project-2-atlantic. Calling the module in the command prompt should automatically produce a 'State.npy' file into the current working directory of the user which can then be read by the post-processor.

To create an animation of the results, download the module *postprocessor.py* from Team Atlantic's GitHub repository found at: https://github.com/msc-acse/acse-4-project-2-atlantic. Ensure that *sph.py* is already downloaded before trying to operate postprocessor.py.

Within a python kernel, the user can create the animation by the following steps:

1. import postprocessor
2. create a variable
3. execute the following command: variable = postprocessor.read_file_plot('State.npy', option, time=-1, image=False)
    a. if option=1 : creates an animation of the location of particles and the x-direction velocity from the initial time to the final simulation time
    b. if option=2 : creates an animation of the location of particles and the y-direction velocity from the initial time to the final simulation time
    c. if option=3 : creates an animation of the location of particles and the Pressure from the initial time to the final simulation time
    d. if option=4: creates an animation of the location of particles and the density from the initial time to the final simulation time
    e. if image=True: if the user selects this parameter to be true, then the postprocessor will save a series of snapshots at each time step of the simulation and save them into the user's current working directory

Front-end user interaction

A module called *front_end.py* allows the user to interact directly with the simulation program. The user should download the module *front_end.py* from Team Atlantic's GitHub repository found at: https://github.com/msc-acse/acse-4-project-2-atlantic. Ensure that *sph.py* and *postprocessor.py* is already downloaded before trying to operate.

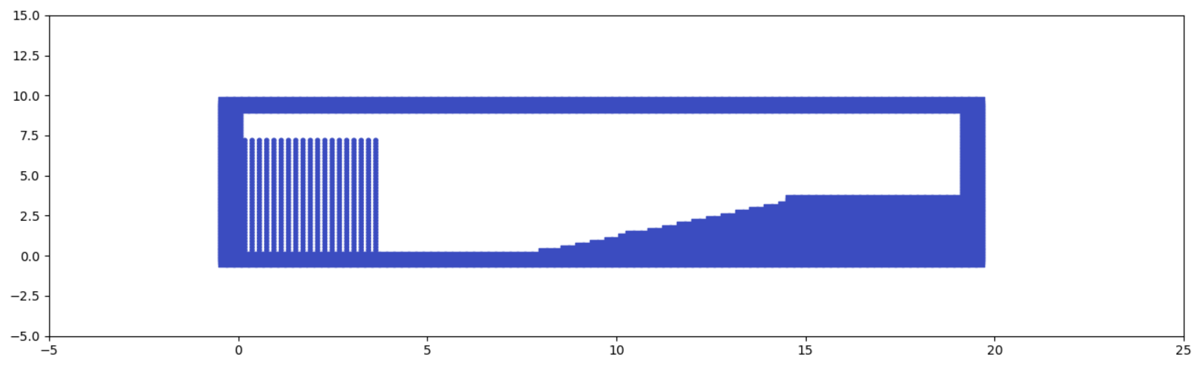


Figure 4. Example Wave_2 geometry specification as specified by the front_end

Testing and code sustainability

Travis is implemented for continuous integration of Team Atlantic's simulator. A pytest module called *pytest_SPH.py* is written for the simulator which considers the following:

- whether any particles have been stored inside the problem domain

- whether the velocity of any particle inside the particle list have exceeded the fluid speed of sound, c0
- whether any particle has exceeded the reference density by a factor of 1.5
- whether any particle has exceeded the domain boundary by initialise_grid()

Inside github, the file *.travis.yml* will automatically run the script for *pytest_SPH.py* for every update to build.

Pseudo code for the tests:

test_grid()

```
• create a domain
    o for particle_list in domain:
        • assert len(particle_list) != 0
```

test_speed()

```
• create a domain and solve the domain until end time
    o load solutions from local directory
        ▪ for particle_list at each time within solutions:
            • for particle in particle_list[]:
                o assert particle.velocityx <= c0 (domain speed of sound)
                o assert particle.velocityy <= c0 (domain speed of sound)
```

test_density()

```
• create a domain and solve the domain until end time
    o load solutions from local directory
        ▪ for particle_list at each time within solutions:
            • for particle in particle_list[]:
                o assert particle.density > 0 (no negative density is given)
                o assert particle.density <= 1.5 * domain.rho0 (domain reference
                  density)
```

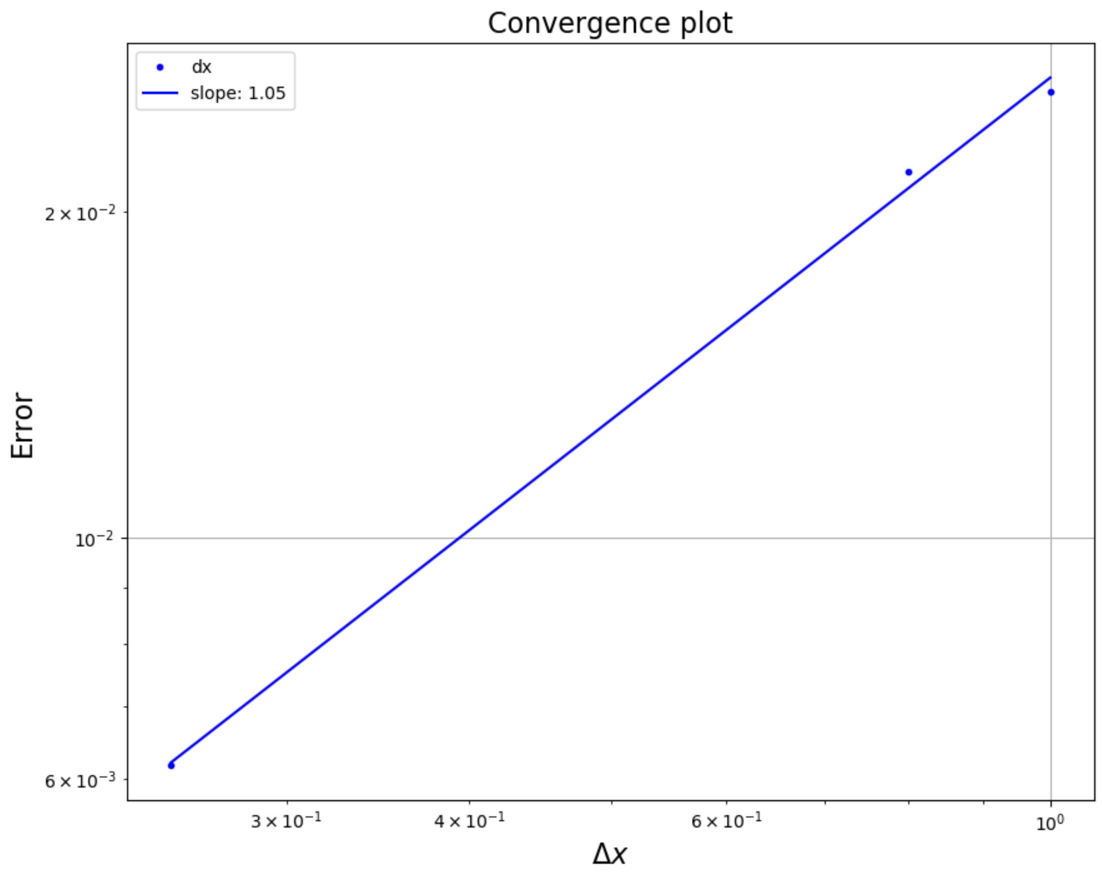
test_mass_conserve()

```
• create a domain and solve the domain until end time
    o load solutions from local directory
        ▪ for particle_list at each time within solutions:
            • for particle in particle_list[]:
                o assert domain.x_min <= particle.xcoordinate <= domain.x_max
                o assert domain.y_min <= particle.ycoordinate <= domain.y_max
```

Error Convergence

A convergence test written for the simulation is within the module convergence_test.py. The test considers three different resoulutions by varying the Δx of the domain and solves the problem 200 time steps forward. The exact solution for the error is assumed to be the results from the simulation with the lowest Δx (highest resolution) and all errors from the other simulations are compared against this exact solution.

The metric used for the error convergence test is the average wave speed – obtained using the highest wave particles' location as a function of time.



From the results shown above, our simulation is linearly converging towards the exact solution as a function of resolution, although not at a high order.

Appendix A: Question 5 Solutions

When dx=0.5, dt= 2.0,

We have ymax (at every timestep, the y location of mid-point of the crest).

Please see the specific results of each interval of time step from Appendix A.1.

discrepancy = 1.520700500049179

Why there might be a discrepancy?

Answer:

When we calculated the time step, we used a constant delta x, because we already know the parameters which are used to calculate the dt, such as c0 and h_fac. Then we can get a function between delta x and time step, which is,

$$\mathrm{dt} = 4\mathrm{dx}$$

However, for the dx in the function below, the dx on the numerator is not the same as what the dx we mentioned in the function above. The dx on the numerator is the x change we got from our function. Thus, when we calculated each speed between two different points, the discrepancy will occur from our functions. What I want to clarify is that the dx on the numerator and the denominator are not the same, so the dx cannot be removed, and that is why the discrepancy changed with each different time step.

$$\frac{x_{i+1} - x_i}{4dx} - \sqrt{gh_0} = Discrepancy$$

We found that:

When dx = 1, dt = 4.0, discrepancy = 1.5377842115798945

When dx = 0.5, dt = 2.0, discrepancy =1.520700500049179

When dx = 0.25, dt= 1.0, discrepancy = 1.5181498089026981

For the dt we calculated above, the dx we used is a constant number, it is different with the Δx(movement) we used at the numerator.

When dx (dt) gets smaller, the discrepancy also gets smaller.

Appendix A.1

```
ymax =[4.78787879 4.78787879 4.78763354 4.78714304 4.78640729 4.78542631
 4.78420011 4.7827287  4.7810121  4.77905033 4.77684338 4.77439124
 4.7716939  4.76875129 4.76556334 4.76212993 4.7584509  4.75452604
 4.75035552 4.74594104 4.74128134 4.73637645 4.7312267  4.72583215
 4.72019285 4.71430886 4.70818028 4.70180725 4.6951899  4.68832844
 4.68122309 4.67387415 4.66628194 4.65844688 4.65036942 4.64205013
 4.63348963 4.62468866 4.61564807 4.60636916 4.59686347 4.58712493
 4.57716953 4.5669995  4.55661708 4.54602459 4.53522441 4.52421898
 4.51301088 4.50160279 4.48999752 4.47819809 4.46620767 4.45402968
 4.44166776 4.42912582 4.41640805 4.40351896 4.39046336 4.37733312
 4.36406888 4.35067158 4.33721195 4.32369815 4.31013732 4.29653557
 4.28289794 4.26922838 4.25552979 4.24180405 4.22805207 4.21427389
 4.20046876 4.18663525 4.17277136 4.15900775 4.14560607 4.13223529
 4.11889628 4.10558972 4.09231618 4.07907512 4.06593165 4.05288629
 4.039938   4.02708419 4.01432075 4.00164217 3.9890416  3.97651103
 3.96404136 3.95238185 3.94106131 3.9298277  3.91866801 3.90762857
 3.89709503 3.886616   3.87617732 3.86576527 3.85536669 3.84496784
 3.83462021 3.82431333 3.81403662 3.80377943 3.79353114 3.78328119
 3.77301922 3.76273505 3.75241877 3.74206076 3.73165176 3.72118284
 3.71064548 3.70003154 3.68933326 3.67854327 3.66765458 3.65666054
 3.64555482 3.63433376 3.62297078 3.61146007 3.59979625 3.58797433
 3.5759897  3.56383806 3.55151544 3.53901814 3.52634268 3.51348578
 3.50044433 3.48721535 3.47379598 3.46018346 3.44637511 3.43236833
 3.41816062 3.40374955 3.38913286 3.37430837 3.35926266 3.34399394
 3.32850094 3.3127831  3.29684052 3.2806741  3.26428556 3.24767749
 3.2308534  3.21381779 3.1965761  3.17913479 3.16150133 3.1436842
 3.12569288 3.10753783 3.08923042 3.07078292 3.05220843 3.03351981
 3.01474864 2.99590976 2.97701782 2.95808707 2.9391313  2.92016372
 2.90119683 2.88224235 2.86331113 2.84441306 2.85130717 2.86316669
 2.87506637 2.8869942  2.89893727 2.91088192 2.92530439 2.94026677
 2.95493023 2.96928763 2.98353918 2.99766013 3.0116251  3.0254083
 3.03898375 3.05232549 3.0654078  3.07820541 3.09069373 3.10284902
 3.11464861 3.12607105 3.13709634 3.14770604 3.15788342 3.16761362
 3.17688376 3.18568298 3.19400259]
```