University of Stuttgart
Institute for Combustion Technology (ITV)

Simulation of Particle Collisions

Final Project for Introduction to C++ for Engineering (ICE)

Chapter 1

General Problem Description

Molecular dynamics (MD) simulation is a powerful computational technique used to study the behavior of atoms and molecules over time. It provides insights into the dynamic processes of molecular systems by numerically solving the classical equations of motion for each particle within the system.
In this tutorial, we will delve into the fundamentals of molecular dynamics simulation, considering only the elastic collision of molecules in 2D. In the following we will treat the molecules as spherical particles with a given mass, radius, and velocity. In general, to solve this MD simulation following steps are required:

1. **Initialize:** Initialize particle distribution with a given velocity in a given domain.

2. **Integration:** Integrate the particle velocity in time to get the new position of the particles.

3. **Force Calculation:** Calculate the forces on the particles acting in the given time step. In this project only the collision force will be considered, forces normal to the collision vector, e.g., friction and other molecular forces are not considered.

4. **Update:** Integrate the equation of motion using the calculated forces to get the new velocities.

A common approach to integrate the particle evolution in time is the so-called event-driven simulation. These types of simulations compute the next collision event and advance time immediately to this point, not considering in between states. In this project we will use the event-driven approach, as a fast and simple algorithm.
To represent the MD simulation in a C++ program, we will need a few classes including,

Particle: A particle class which stores it's properties, here in this tutorial, the position, velocity, and an integer representing an ID. The last part is required to visualize different particle populations. Further, the class will need methods to calculate the forces acting on it during particle-particle and particle-wall collisions.

Cloud: A class to group together a particle population in a cloud, handling the evolution of the particles in time.

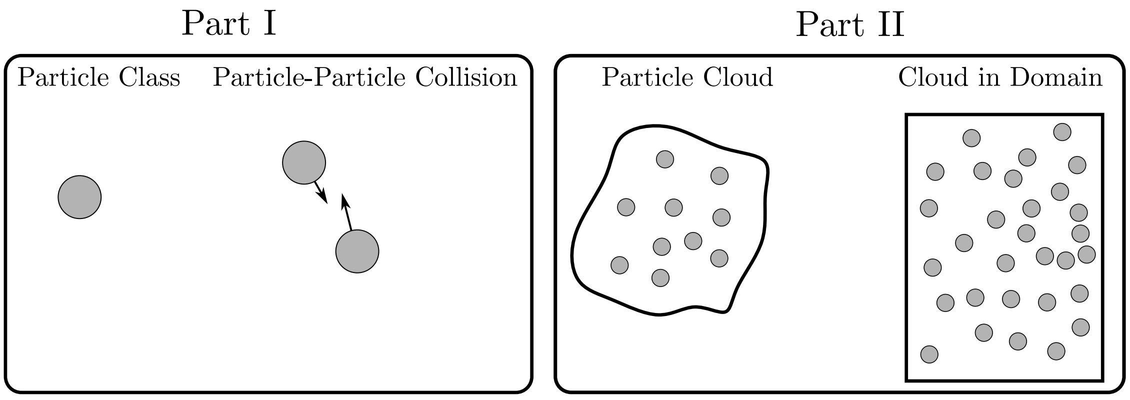
Domain: A class representing the physical domain built up by straight walls.

Wall: A wall expressed as an infinite line with an origin and a normal direction.

To solve this complex task, it is divided in two main parts:

Part I Develop a particle class, handle particle-particle collision

Part II Add a cloud which contains all particles and a domain class to represent a physical space



Chapter 2

Part I: Particle Class and Collision

In the first part of the project the particle class is developed considering particle-particle collision and evolving one to three particles in time (Steps 2 to 3 in Ch. 1). The particle class contains the main particle properties of position, velocity, and a label ID representing a color. The last part will be required later to visualize the results. However, the particle class does not only store the information of position and velocity but also provides functions to modify the state of the particle, e.g., a move function to evolve the particle in space, a collision function to calculate the force on the particle and an update function to update the new velocities due to the forces on the particle.

2.1 Particle-Particle Collision

The elastic collision of two particles must conserve the energy and momentum:

$$\text{Energy:} \qquad m_1 \vec{u}_1^2 + m_2 \vec{u}_2^2 = m_1 \left(\vec{u}_1'\right)^2 + m_2 \left(\vec{u}_2'\right)^2 \tag{2.1}$$

$$\text{Momentum:} \qquad m_1 \vec{u}_1 + m_2 \vec{u}_2 = m_1 \vec{u}_1' + m_2 \vec{u}_2' \tag{2.2}$$

Here the superscript $'$ denotes the state after collision. To solve the particle collision of two particles in 2D, the velocities are split up in a velocity component in the direction of the collision, $\vec{u}_d$, and a normal component, $\vec{u}_n$, see Fig. 2.1. Here, no friction between particles is considered, hence the normal component of the velocity, $\vec{u}_n$, is unchanged. With this, the 2D collision problem reduces to a 1D collision. Combining the momentum and energy conservation equations leads then to,

$$\vec{u}_{d,1}' = 2\left(\frac{m_1 \vec{u}_{d,1} + m_2 \vec{u}_{d,2}}{m_1 + m_2}\right) - \vec{u}_{d,1}, \tag{2.3}$$

$$\vec{u}_{d,2}' = 2\left(\frac{m_1 \vec{u}_{d,1} + m_2 \vec{u}_{d,2}}{m_1 + m_2}\right) - \vec{u}_{d,2}, \tag{2.4}$$

with $\vec{u}_{d,1}'$ and $\vec{u}_{d,2}'$ representing the velocity component in the direction of the hit after the collision. Combining again the unaltered normal component with the new velocity u_d' results in the velocity of the particle after the collision,

$$\vec{u}_{\text{new}} = \vec{u}_n + \vec{u}_d'. \tag{2.5}$$

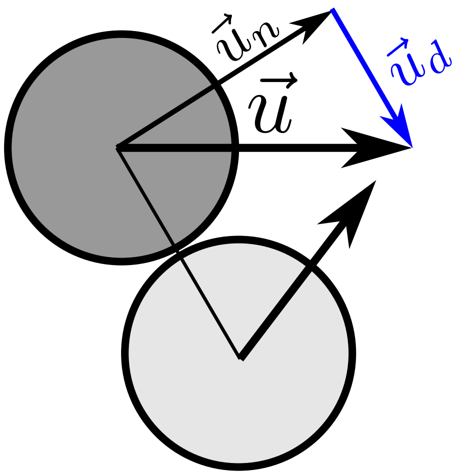


Figure 2.1: Sketch of the collision of two particles and their velocity decomposition.

2.2 Evolve Particles in Time

To evolve the particle in time, first the new particle positions are calculated based on their current velocities and a given discrete time step,

$$\vec{x}_{\text{new}} = \vec{x}_{\text{old}} + \vec{u}\Delta t. \tag{2.6}$$

Hereby, it is important to chose the time step Δt in such a way, that particles are not advanced to far in time leading to missed collisions. Therefore, we will utilize the event-driven approach and set the time step Δt to the point in time when the first collision occurs. As the particles move in a vacuum with no other force than collision acting on them, the time to collision, τ_{coll}, between two particles i and j can be calculated with following formula [1],

$$\tau_{\text{coll}} = \frac{-b - \sqrt{b^2 - u_{ij}^2\left(r_{ij}^2 - \sigma^2\right)}}{u_{ij}^2}, \tag{2.7}$$

with

$$\text{Distance between particles:} \qquad \vec{r}_{ij} = \vec{x}_j - \vec{x}_i, \tag{2.8}$$

$$\text{Relative velocity:} \qquad \vec{u}_{ij} = \vec{u}_j - \vec{u}_i, \tag{2.9}$$

$$\text{Relative motion:} \qquad b = \vec{r}_{ij} \cdot \vec{v}_{ij} \tag{2.10}$$

and

$$\sigma_{ij} = r_i + r_j. \tag{2.11}$$

Note that a collision of the two particles only occurs if $b < 0$ and the discriminant, $b^2 - u_{ij}^2\left(r_{ij}^2 - \sigma^2\right) \geq 0$.
After finding the particle pair i and j, that will collide first, the particles are advanced in time with Eq. (2.6) using $\Delta t = \min\left(\tau_{\text{coll}}\right)$. As now the collision event between particles i and j occurs, the collision force on the particles can be computed. Note, that the time step has been set to the smallest collision time,

hence no other collision could have occured than the one between i and j. At last the equation motion has to be solved to update the particle velocities,

$$
\begin{align}
\vec{F} &= m\frac{\partial \vec{u}}{\partial t}, \tag{2.12}\\
&= m\frac{\vec{u}_{\text{new}} - \vec{u}_{\text{old}}}{\Delta t}, \tag{2.13}\\
\Rightarrow \vec{u}_{\text{new}} &= \vec{u}_{\text{old}} + \frac{\vec{F}}{m}\Delta t. \tag{2.14}
\end{align}
$$

The algorithm to evolve the particles is also sketched in Fig. 2.2.

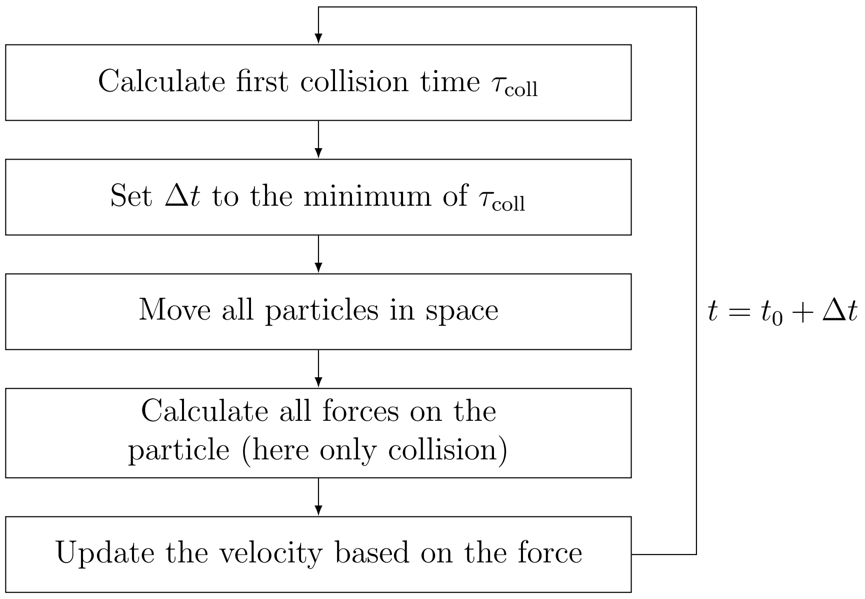

Calculate first collision time τ_{coll}
Set Δt to the minimum of τ_{coll}
Move all particles in space
Calculate all forces on the particle (here only collision)
Update the velocity based on the force
$t = t_0 + \Delta t$

Figure 2.2: Algorithm to advance the particles in time.

2.3 Tasks

First navigate to the directory "`Mod01-elasticShock/`", which contains all required scripts and source code files for part I. In this folder you find a directory called "`src/`" which contains the source files you need to modify, see also Fig. 2.3.

Figure 2.3: Directory structure of Mod01.

```
project dir
 ├── data - Output of the results
 ├── PlotScript - Script for plotting results
 ├── src - Source files you need to modify
 ├── tests - Do not modify!
 ├── Allrun - Script to execute the tests
 └── CMakeLists.txt - Required for compilation
```

2.3.1 Particle class

Your first task is to complete the particle class definition and implementation in "`src/particle.hpp`" and "`src/particle.cpp`", marked with a ToDo comment.

Private member variables The private member variables of the particle class are listed in Tab. 2.1. To represent the vectors of position, velocity, and force a `vector2D` class is used. This class is provided with the source code and allows simple vector arithmetic, see also Sec. 2.4 for further details on how to use this class. Further, here the notation of an underscore after the variable name is used to denote that this is a private class variable!

Table 2.1: Private member variables of the particle class.

Variable	Type	Description
`pos_`	vector2D	Position [m]
`u_`	vector2D	Velocity [m/s]
`F_`	vector2D	Forces [N]
`r_`	double	Radius [m]
`m_`	double	Mass [kg]
`color_`	int	ID to represent the color

Particle class constructor The constructor of the particle class takes the position, velocity, radius, mass, and color as an input and sets the private member variables to these values. For the vectors position and velocity the components in x and y direction will be used instead of a `vector2D` object.

Public accessor functions To access the private member variables functions for the position, velocity, radius, mass and color are provided. However, it shall be noted that these functions should only return **constant references** to the private member variables, disallowing any modification of them. The state of the particle, e.g., its position should only be modified over the move function. To allow the accessor functions to be called on a constant particle object,

```cpp
const particle p(...);
// Get the radius of the const declared particle
double r = p.r();
```

the functions have to be declared constant as well. This is done by putting the keyword const **after** the function definition:

```cpp
const double& r r() const;
```

Complete the class definition with the accessor functions of Tab. 2.2 in "`src/particle.hpp`" and "`src/particle.cpp`".

Table 2.2: Public member accessor functions of the particle class.

Function Name	Return Type	Description
`pos()`	const vector2D&	Position [m]
`vel()`	const vector2D&	Velocity [m/s]
`r()`	const double&	Radius [m]
`m()`	const double&	Mass [kg]
`color()`	int	ID to represent the color

move(): Move a particle in space To evolve the particle position in time the `move()` function is used. This is a void function which takes the time step Δt as an input. Eq. (2.6) describes how the position of the particle is updated with current velocity. Note that the move function does not alter the velocities! This is done in the `update()` function. The function is declared in "`particle.hpp`" and must be defined in "`particle.cpp`".

update(): Update the particle velocities The `update()` function is used to update the particle velocity based on the current forces with Eq. (2.14). Note, this function only modified the velocity component, it does **not** change the position. After calculating the new velocities the force vector is set to zero, see also Fig. 2.2.

collisionForce(): Calculate the collision force To handle particle collisions the `collisionForce()` function is used. This function serves two purposes, first it checks if a collision occurs with a given particle p and second calculates the equivalent force that results in the velocity over the time step Δt.
Here we define a particle collision if the distance between the two particles is less than their added radii,

$$\text{collision if: } |\vec{x}_2 - \vec{x}_1| \leq r_1 + r_2. \tag{2.15}$$

If a collision occurs the new velocity is calculated with the equations presented in Sec. 2.1. To do so, the velocity is first decomposed in the direction of the collision and normal to it,

$$\vec{u}_{d,1} = \vec{d}^* \left(\vec{u}_1 \cdot \left(\vec{d}^*\right)\right), \tag{2.16}$$

$$\vec{u}_{n,1} = \vec{n}\left(\vec{u}_1 \cdot (\vec{n})\right). \tag{2.17}$$

Here, $\vec{n}$ is the unit normal vector to the distance between the two particle p1 and p2 with,

$$\vec{n} = \begin{pmatrix} -d_y^* \\ d_x^* \end{pmatrix} \tag{2.18}$$

where d_x and d_y are the normalized components of the distance vector $\vec{d}^* = \frac{\vec{p}_1 - \vec{p}_2}{|\vec{p}_1 - \vec{p}_2|}$. The decomposed velocities are then used to calculate the new velocity of the particle with,

$$\vec{u}_{d,1}^{\text{new}} = 2\left(\frac{m_1 u_{d,1} + m_2 u_{d,2}}{m_1 + m_2}\right) - u_{d,1}, \tag{2.19}$$

$$\vec{u}_1^{\text{new}} = \vec{u}_{d,1}^{\text{new}} + \vec{u}_{n,1}. \tag{2.20}$$

Note that only the velocity of the particle on which `collisionForce()` is called is modified. The velocity and force of particle p2 is unchanged and will be modified once it's own `collisionForce()` function is called at a later point.
To calculate the force on the particle Newton's law and the time step Δt is used to calculate the equivalent velocity and force,

$$\vec{F} = m_1 \vec{a} \tag{2.21}$$
$$= m_1 \frac{\Delta \vec{u}}{\Delta t} \tag{2.22}$$
$$= m_1 \frac{\vec{u}_1^{\text{new}} - \vec{u}_1^{\text{old}}}{\Delta t} \tag{2.23}$$

timeToCollision(): Calculate time to collision Calculate the time to collision to a given particle with Eq. (2.7).
Note: This function does not modify the state of the particle, can you declare it const?

2.3.2 Evolve particles in time

The particle class has now all the required functions to simulate a particle-particle collision in time. Now a function is needed to evolve the particle position in time and to check for potential collision, called `evolveParticles()`. The definition of the function is given in `functions.hpp` and the implementation should be written in the `functions.cpp` file. The pseudo-code for the algorithm is shown in Lst. 2.1.

Listing 2.1: Pseudo code for evolve function

```
input:  std::vector<particle> particleList,
        const double minDeltaT.
        const double maxDeltaT

output: deltaT

// Set deltaT to maxDeltaT
deltaT = maxDeltaT

// Store particle indices i and j
indices i, j

// Get the minimum time step
for p1 in particleList
    for p2 in particleList
        collision time of p1 and p2
        if tau < deltaT
            deltaT = tau
            Store i and j index of p1 and p2

deltaT = max(deltaT,minDeltaT)


// Move particles
for p in particleList
    call move() on p

// Calculate collision force
if collision occured
    set collisionForce() of particle i
    set collisionForce() of particle j
```

```

// Update velocities based on forces
for p in particleList
    call update() on p
```

2.3.3 Compile & Test

To compile and run the tests use the provided `Allrun.sh` script. This script will compile your source code and link it to a test library which provides different test cases you can choose. For this project the test cases available are:

- **vector2D:**
  This test case will test the vector2D class provided for this project.
- **particleClass:**
  This test case will test your particle class for
  - Constructor sets all private member variables correctly
  - The `move()` function moves the particle in space
  - Time to collision is calculated correctly
- **headOnCollision:**
  In this test case two particles will collide head on. The test will check that the velocities are set correctly after the collision and that the kinetic energy is conserved.
  In addition this test case writes a file to the `data/` directory which is then visualized with a python script.
- **rightAngleCollision:**
  Two particles collide in a right angle. The test will check that the velocities are set correctly after the collision and that the kinetic energy is conserved.
  In addition this test case writes a file to the `data/` directory which is then visualized with a python script.
- **threeParticleCollision:**
  Three particles collide at the same time. The test will check that the algorithm will calculate first the collision between two particles and then between one of the particles and the third, checking that the kinetic energy and momentum is conserved.

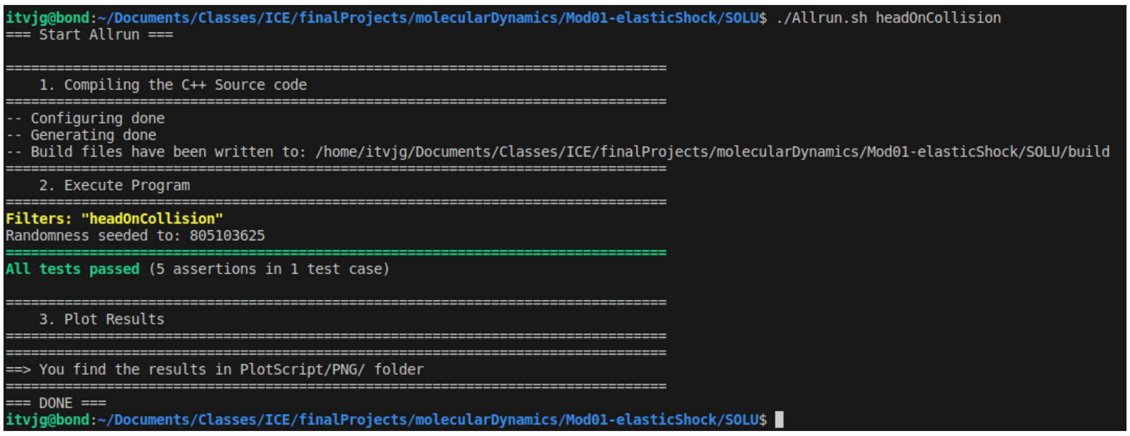
To run a test case, e.g., the right angle collision test execute the Allrun script with:

```
./Allrun rightAngleCollision
```

You can also run all test cases with the ”all” keyword:

```
./Allrun all
```

An example of how a successful test case run looks like is shown in Fig. **??**.



```
itvjg@bond:~/Documents/Classes/ICE/finalProjects/molecularDynamics/Mod01-elasticShock/SOLU$ ./Allrun.sh headOnCollision
=== Start Allrun ===

================================================================================
    1. Compiling the C++ Source code
================================================================================
-- Configuring done
-- Generating done
-- Build files have been written to: /home/itvjg/Documents/Classes/ICE/finalProjects/molecularDynamics/Mod01-elasticShock/SOLU/build
================================================================================
    2. Execute Program
================================================================================
Filters: "headOnCollision"
Randomness seeded to: 805103625
================================================================================
All tests passed (5 assertions in 1 test case)

================================================================================
    3. Plot Results
================================================================================
================================================================================
==> You find the results in PlotScript/PNG/ folder
================================================================================
=== DONE ===
itvjg@bond:~/Documents/Classes/ICE/finalProjects/molecularDynamics/Mod01-elasticShock/SOLU$ 
```

Figure 2.4: Screenshot of a successful test run.

2.4 2D Vector Class

The calculation of the particle movement, collision and velocities requires some vector arithmetic. To make the programming easier and closer to the mathematical representation a `vector2D` class is provided with the source code. In the following examples how to construct and use the vector are given.

Listing 2.2: Create a 2D vector

```cpp
// Construct with default constructor
// By default the x and y value is initialized to zero
vector2D vec;

// Construct from components
// x-component: 1.0
// y-component: 2.0
vector2D vec2(1.0,2.0);
```

Listing 2.3: Access the vector components

```cpp
// The x and y component of the vector can be accessed with
double x = vec.x();
double y = vec.y();

// Note: The vector::x() and vector::y() function
// return references
// You can use them to modify the values
vec.x() = 2.1;  // set x-value to 2.1
vec.y() = 1.1;  // set y-value to 1.1
```

Listing 2.4: Vector arithmetic – Addition

```cpp
// Vector addition
vector2D a(1.0,2.0);
vector2D b(0.5,0.5);

// Vector c has now
// x-value: 1.5
// y-value: 2.5
vector2D c = a+b;

// The addition can also be used with the
// += operator
b += a;
// Now b has the
// x-value: 1.5
// y-value: 2.5
```

Listing 2.5: Vector arithmetic – Subtraction

```
// Vector subtraction
vector2D a(1.0,2.0);
vector2D b(0.5,0.5);

// Vector c has now
// x-value: 0.5
// y-value: 1.5
vector2D c = a-b;

// The subtraction can also be used with the
// -= operator
b -= a;
// Now b has the
// x-value: -0.5
// y-value: -1.5
```

Listing 2.6: Vector arithmetic – Scalar product

```
// The scalar product of a vector a and b is calculated with
// the operator overload of operator&()
```

// $s = \vec{a} \cdot \vec{b}$

```
vector2D a(1.0,2.0);
vector2D b(0.5,0.5);

// s is the scalar product of a and b
double s = a & b;
```

Listing 2.7: Vector arithmetic – Vector times a scalar

```
vector2D a(1.0,2.0);
double s = 0.5;

// vector b has now
// x-value: 0.5
// y-value: 1.0
vector2D b = a*s;

// or
vector2D b = s*a;
```

Chapter 3

Part II: Particle Cloud and Domain

In part II of the project you will collect the particles in a cloud which will serve two purposes. Firstly, the cloud will own and manage all particles including the time evolution. Secondly, the cloud knows about the domain and can handle the particle-wall collisions.
To achieve this goal you will need to write three additional classes: wall, domain, and cloud class.

3.1 Domain and Wall Class

In this project we define a domain as a region bounded by a list of walls which provides following functions:

walls(): Returns constant access to the walls bounding the domain.

bounds() Returns the x and y dimension of the domain, see also Fig. 3.1.

genPointInside(): Generates a random point inside of the domain with a given buffer space to the wall.

inside(): Returns true if the provided particle is inside the domain boundary, not bounds!

A domain may come in any shape, e.g. a rectangle, triangle, hexagon etc. Due to the potential unlimited design of domains we will use a general pure abstract domain class of which then the more specific domains, e.g. a rectangular one is derived. The abstract base class domain declares the functions described above, however does not give an implementation. The implementation details are the task of the specific class derived from the domain class.
For this project only rectangular domains, consisting of four walls named left, right, top and bottom, will be considered. The walls are represented, as outlined above: straight infinite lines with a normal and origin vector. The rectangular domain then constructs four walls defined by the origin of the domain as the lower left corner and a width and height. Further, we will follow the convention that the normal direction of the walls **always** points towards the domain, see also Fig. 3.2.

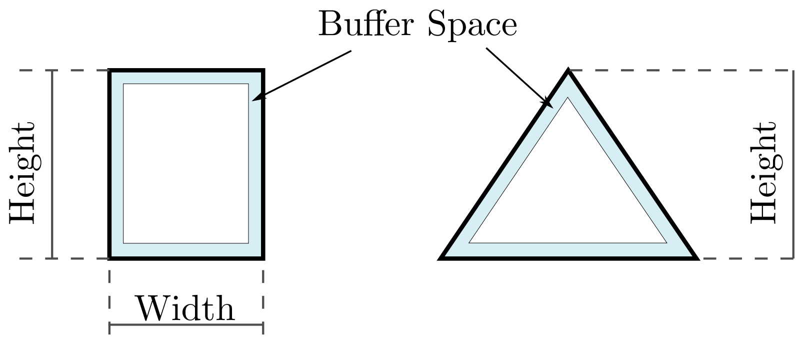


Figure 3.1: Sketch of the general domain class for an example rectangular and triangle domain. The bounds of the domain is given as the width and height. Within the light blue shaded buffer area no particles may be generated with `genPointInside()`.

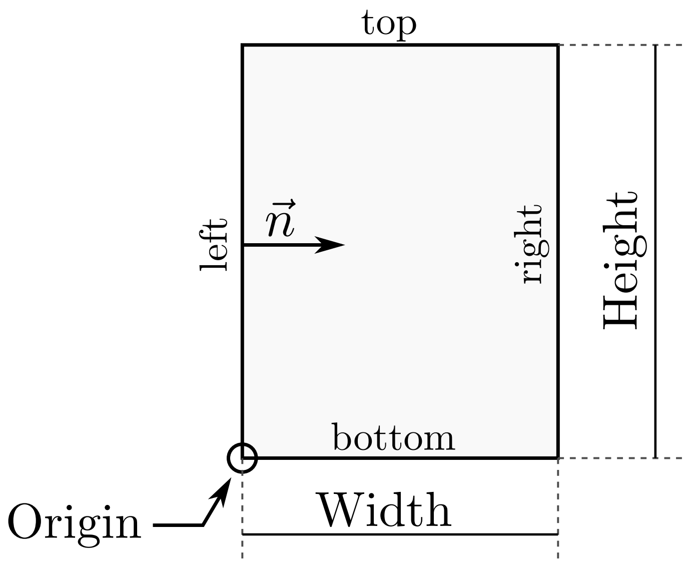


Figure 3.2: Sketch of the rectangular domain design with origin, width and height. The wall normal direction points towards the domain.

3.2 Group particles in a cloud

In Mod01 the particles had been constructed and stored in a vector which was then passed to the `evolveParticles()` function. Now, in Mod02, particle management is transferred to an own class, the `cloud` class. The main idea is to group together the data and the methods in one class, which is essentially the core aspect of object oriented programming. In addition, the cloud will not operate in an infinite space as the evolve function of Part I, but within a given physical domain. Therefore, the cloud owns the particles and has knowledge of the domain. The cloud however does not own or manages the domain. These dependencies are sketched in Fig. 3.3.

To populate the cloud with particles two approaches exist in this project:

1. `addParticle()`: Takes an existing particle as the input and adds it to the cloud if it does not overlap with another particle.

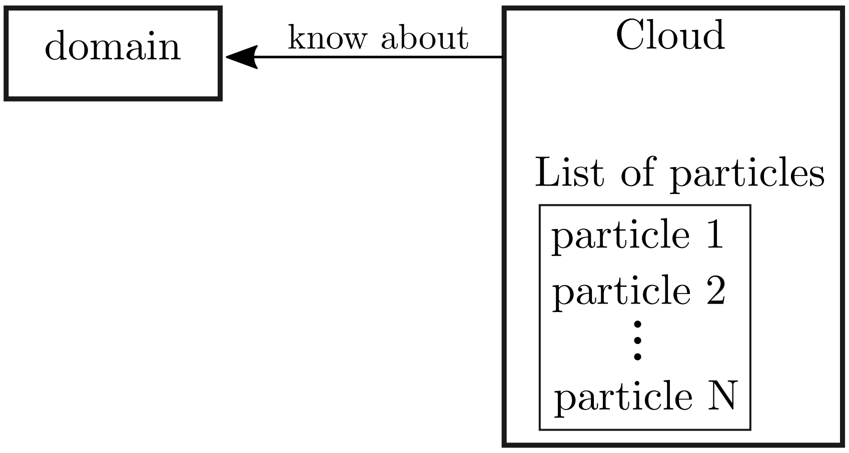


Figure 3.3: Sketch of cloud, domain, and particle dependencies.

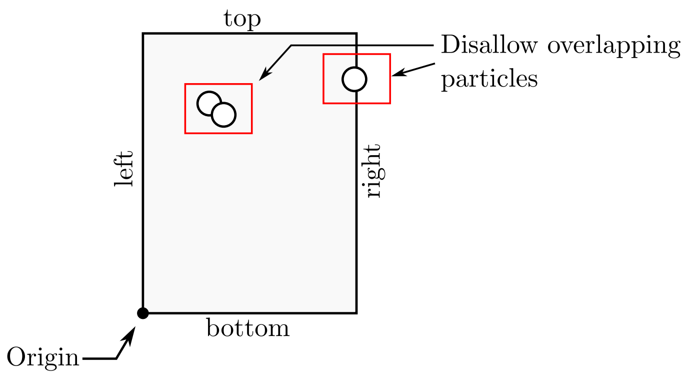


Figure 3.4: Sketch of initializing particles in the cloud avoiding overlaps.

2. `init()`: This function takes a domain as a bounding box, default parameters of particle radius, mass, color, and maximum velocity, as well as the number of particles to generate as an input. This function then attempts to insert the given number of particles in the given domain box, at random locations, with a random velocity of $\vec{u} = [-\vec{u}_{\max}, \vec{u}_{\max}]$ avoiding any overlapping particles, see Fig 3.4.

3.3 Tasks

In the Part II of this project we will describe the public member functions of the classes, their names, input and return types. These functions serve as the interface through which external components, like the test library, interact with these classes. It is crucial to adhere strictly to the given name, input and return type of the public member functions. Otherwise the program will not compile or will not work properly. This approach mirrors the practices commonly encountered in larger software projects, where strict adherence to predefined protocols is essential for seamless data exchange and system interoperability.

3.3.1 Wall class:

The files `wall.hpp` and `wall.cpp` in the `src/` directory already provide a general structure which you will need to complete. The constructor of the wall class takes exactly two arguments, the wall origin and normal vector,

```
class wall constructor
input: vector2D origin
       vector2D normal
```

The order of first origin vector and then the normal vector is important!
The public member functions of this class are listed in Tab. 3.1 Note that once the wall is constructed with the origin and normal, it can no longer be modified, as the public functions only allow a const access and no modification.

Table 3.1: Public member functions of `wall` class.

Name	Return Type	Input	Description
`origin() const`	const vector2D&	-	Return origin
`normal() const`	const vector2D&	-	Return normal

3.3.2 Rectangular domain class:

The pure abstract domain class is given in `domain.hpp`. This class is already complete and does not require any modification. Your task is now to derive a new `rectDomain` class from `domain` and implement the pure abstract functions. Further some specific rectangular domain functions are added.
The rectangular domain is constructed from the width, height and origin, see also Fig. 3.2,

```
Domain Constructor
input: double width
       double height
       vector2D origin
```

Think about if you can declare the input as const.
The public member functions in addition to the ones of the domain class and their descriptions are listed in Tab. 3.2. The `inside()` function returns true if the center point of the given particle is inside the domain bounds and false if the center of the particle is outside the domain bounds.

3.3.3 Expand particle class:

First copy the particle class of Mod01 to the `src` directory of Mod02. In Mod01 the particle could only collide with other particles and the force is calculated with the `collisionForce()` member function. Now, the particle can also collide with the domain wall, which requires to add a `collisionForce()` function for the particle-wall interaction. Here, we will overload the existing `collisionForce()` function to work for particles and walls. An example how function overload works is given in Listing 3.1.

Table 3.2: Additional public member functions of `rectDomain` class.

Name	Return Type	Input	Description
`walls() const`	const std::vector<wall>&	-	Return the walls
`origin() const`	const vector2D&	-	Return origin of the domain
`width() const`	double	-	Return the width
`height() const`	double	-	Return the height
`top() const`	const wall&	-	Return top wall
`bottom() const`	const wall&	-	Return bottom wall
`left() const`	const wall&	-	Return left wall
`right() const`	const wall&	-	Return right wall

Listing 3.1: Function overload in C++

```cpp
// A general function to calculate a force
// the implementation details are here of no interest
double force(const particle& p);

// Overload the function by using the same name,
// but different input arguments
double force(const wall& w1);

...

// Call the function
// The compiler automatically chooses the correct
// force function depending on the input arguments
double fParticle = force(p1);
double fWall = force(w1);
```

Similar to the particle-particle collision, a pure elastic collision is assumed and hence only the wall normal velocity is of interest, see Fig. 3.5. Think about how you need to set the force on the particle to achieve the change in the velocity.

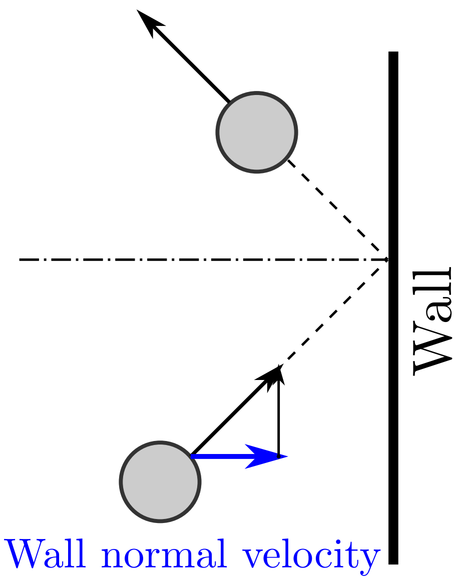


Figure 3.5: Sketch of an elastic collision of a particle with a wall.

3.3.4 Cloud class:

The cloud class is the main task of Part II. As described in Sec. 3.2 its purpose is to store and manage the particle population. The public member functions you need to implement are listed in Tab. 3.3. Note, that you probably will need a few private member functions to fulfill the class functions. However, as private member functions are only known to the class, they are not presented to the outside, thus, this is entirely in your hands what you need and how you implement it.
For the `init()` function you need to generate the particles with a random position and velocity. An example how random numbers can be generated in C++ is given in the Listing 3.2.

Table 3.3: Public member functions of `cloud` class.

Name	Return Type	Input	Description
`cloud()`	-	-	Default constructor
`cloud()`	-	domain&	Construct with given domain
`init()`	-	const domain& domain	Init. particle distribution
	-	const int nParticles	
	-	const double maxU	
	-	const double r	
	-	const double m	
	-	const int color	
`addParticle()`	bool	const particle&	Add a single particle
`setDomain()`	-	const domain& domain	Set the domain
`nParticles() const`	int	-	Return # of particles
`particles() const`	const vector<particle>&	-	Return the particle list
`validDomain() const`	bool	-	Return if the domain is set [1]
`evolve()`	double	const double minDeltaT	Evolve particle population in time
		const double maxDeltaT	and return used deltaT

Listing 3.2: Example how to generate random numbers

```cpp
#include <random>

...

// Create a random generator
std::random_device dev;
std::mt19937 gen(dev());

// Here the first entry marks the lower limit of
// the distribution and the second entry the upper limit.
std::uniform_real_distribution<> disX(1.1,2.2);

// Generate a random uniform distributed value
// between 1.1 and 2.2
double xRandom = disX(gen);
```

3.3.5 Compile & Test

To compile and run the tests use the provided `Allrun.sh` script. This script will compile your source code and link it to a test library which provides different test cases you can choose. For this project the test cases available are:

- **wallClass:**
  Tests the `wall` class constructor and access functions.

- **rectDomainClass:**
  Test the `rectDomain` class including the constructor, access functions and the `inside()` check.

- **cloudClass**
  The `cloud` class tests checks the default constructor, and construction with a domain for a cloud. It further checks if particles can be added to the cloud and if the cloud is correctly initialized.
  It does not check the evolve function!

- **particleDiffusionTest:**
  In this test a rectangular domain is initialized with 10 particles of color ID 1 in the top part of the domain, and 10 particles with color ID 2, but the same mass, radius, and maximum velocity as before in the bottom part. The simulation is then started, and the two different particle.
  You will find the results as PNG images in the `PlorScript/PNG` directory. In addition, a video in MP4 format is generated which you can also use to visualize the results.
  In addition, the conservation of the kinetic energy is checked in each time step, stopping the simulation if the kinetic energy is no longer conserved.

- **particleDiffusionLargeTest:**
  This test case resembles the particle diffusion test, however with a larger domain and 100 particles for each particle population.

- **particleHitWallTest:**
  This test checks that the particle is reflected correctly by the wall.

To run a test case, e.g., the right angle collision test execute the Allrun script with:

```
        ./Allrun rightAngleCollision
```

You can also run all test cases with the ”all” keyword:

```
        ./Allrun all
```