

## 1 Objectives

The objectives of this project are the following:

- write an implementation of an **event-driven** simulation that simulates the evolution of 2D particles subjected to elastic collisions law and trapped in a box
- read an initial state of the simulation from a text file
- use an efficient data structure to manage the events of the simulation

Each objective is presented in the following.

## 2 Presentation

Let us imagine a set of particles trapped in a box. We want to simulate the evolution of the particles supposing that they are only subjected to the law of elastic collisions (i.e. there is no gravity, no friction force etc). Their movement is thus rectilinear. We will use a Graphical User Interface (GUI) to view the simulation.

One solution is to use *time-driven* simulation: a time step  $dt$  is fixed and starting from a time  $t$ , you compute the positions of the particles at time  $t + dt$ . You have then to check if two particles occupy the same position, “go back” in time to the collision time and update the velocities and the particles position. This approach suffers of two main drawbacks:

- if you consider  $n$  particles, a 1s simulation takes a time proportional to  $\frac{n^2}{dt}$  (checking for collisions is quadratic) which is not suitable for large  $n$
- if  $dt$  is too big, you may miss collisions

## 3 Event-driven simulation

Another approach is to use a simulation guided by *events*, particularly *collision events*, because it is easy to update the particles positions between two collisions. The main loop of an event-driven simulation will be the following:

1. find the next collision event  $e$  to process (i.e. the one that will occur first among all possible collision events)
2. update the particles positions to the time of event  $e$  and update the state of the particles concerned by the collision
3. go back to step 1 until you reach a fixed time limit

A first naive implementation of such an event-driven simulation will compute all possible collisions at step 1 and thus has a quadratic time complexity: for each particle, you have to compute the expected time for the particle to collide a wall or another particle.

You will find on the website of the lecture two videos showing such an approach: the one in which we simulate the evolution of 200 random particles is rather fluid, but if the simulation consists of 2000 particles, it freezes at each update because of the quadratic complexity of the algorithm.

## 4 An efficient data structure: priority queue

The bottleneck of the algorithm is clearly step 1 in which all the possible collisions are computed. We need an efficient data structure to store the events, in order to be able to efficiently retrieve the earliest event and also to efficiently store new events.

A **priority queue** is an Abstract Data Type similar to a queue but with a priority order associated to the elements that it stores. It has two operations:

- an **insert** operation that stores an element with an associated priority
- an **extract-min** operation that removes the element with the highest priority from the queue and returns it

We will implement our priority queue using a **binary heap** (cf. 6.5) which guarantees a worst case execution time in  $O(\log n)$  for both operations. See section 6.5 for details on the implementation.

The priority queue will store the events and the priority order will be given by the dates associated to the events: the highest priority element will be the one with the earliest date. We will thus be able to retrieve the earliest event in  $O(\log n)$ .

One important point is to be taken into account: some events will be invalidated by other ones. For instance, let us suppose that a colliding event  $e$  between particle  $p_1$  and particle  $p_2$  happens at time  $t_e$ . Suppose that the earliest event to be treated is  $e'$  involving  $p_1$ . After having executed  $e'$ , the event  $e$  is likely to be no more valid, as  $p_1$  velocity has surely changed. In order to avoid to go through the entire priority queue to find all events involving  $p_1$ , we will add to each event the number of collisions in which the particles of the event have been implicated at the date the event is created. For instance, let us suppose that  $e$  is computed at time  $t$  and that the number of collisions in which  $p_1$  and  $p_2$  have been implicated at  $t$  are respectively  $c_1$  and  $c_2$ . Then, when extracting event  $e$  from the queue, if the number of collisions of  $p_1$  is not equal to  $c_1$ ,  $e$  is invalid: it means that another collision involving  $p_1$  has been treated between  $t$  and  $t_e$ . The priority queue will thus contain *invalid events*, but treating them is easy (just compare the actual number of collisions of the involved particles with those stored in the event) and extracting them is inexpensive (logarithmic time). We will also store in the priority queue *refreshing* events that simply command to refresh the GUI.

The main loop of the simulation using a priority queue is presented on algorithm 4.1. The content of the main loop is now clearly linear, because the bottleneck is to find all possible collisions for at most two particles.

## 5 Particle collisions

You will have to handle elastic collisions between particles or between particles and a wall. You will find in this section all the necessary “mathematical” equations.

First, you will have to check if a particle  $p$  with position  $(x, y)$ , velocity  $(vx, vy)$ , radius  $r$  and mass  $m$  will collide an vertical wall. Suppose that the box bottom-left corner has coordinates  $(0, 0)$  and that the box top-right corner has coordinates  $(1, 1)$ . Then

- if  $v_x > 0$ ,  $p$  will collide a vertical wall in  $\frac{1 - x - r}{vx}$  unit of time
- if  $v_x < 0$ ,  $p$  will collide a vertical wall in  $\frac{r - x}{vx}$  unit of time
- otherwise it will not collide a vertical wall

We now need to compute the new velocity of the particles involved in the collision. In the case of the collision of a particle  $p$  with velocity  $(vx, vy)$  with a vertical wall, momentum and kinetic energy conservation lead to a new velocity of  $(-vx, vy)$ .

The reasoning is exactly the same for computing the time to collide an horizontal wall.

Now, let us suppose that you have to check if a particle  $p_1$  with position  $\vec{p}_1 = (x_1, y_1)$ , velocity  $\vec{v}_1 = (vx_1, vy_1)$ , radius  $r_1$  and mass  $m_1$  collides a particle  $p_2$  (we use the same notations for the characteristics of  $p_2$ ). Let us denote  $\Delta\vec{p} = \vec{p}_2 - \vec{p}_1$  and  $\Delta\vec{v} = \vec{v}_2 - \vec{v}_1$ . The amount of time  $\Delta t$  in which  $p_1$  and  $p_2$  will collide is a solution of the following equation:

$$||\Delta t \Delta\vec{v} + \Delta\vec{p}|| = r_1 + r_2$$

and therefore  $\Delta t$  is a solution of

$$(\Delta\vec{v} \cdot \Delta\vec{v})\Delta t^2 + 2(\Delta\vec{p} \cdot \Delta\vec{v})\Delta t + (\Delta\vec{p} \cdot \Delta\vec{p}) - (r_1 + r_2)^2 = 0 \quad (1)$$

where  $\cdot$  represents the scalar product of 2D vectors.

---

**Algorithm 4.1:** main loop of the event-driven simulation

---

**Input:** a set of initialized particles

```

1 create an empty priority queue Q;
2 foreach particle p do
3   foreach future collision implicating p do
4     create an event corresponding to the collision and insert it in Q;
5   end
6 end
7 insert a refreshing event with associated time 0;
8 while Q is not empty do
9   extract the earliest event e from Q ;                               /* time associated to e is t */
10  if e is valid (check number of collisions) then
11    update the particles positions up to time t;
12    if e is a refreshing event then
13      redraw the particle in the GUI ;
14      add a refreshing event at time t + refresh rate in Q;
15    end
16    if e is a collision between p and a vertical wall then
17      compute new velocity of p;
18      compute new possible collisions involving p and add the corresponding events in Q;
19    end
20    if e is a collision between p and an horizontal wall then
21      compute new velocity of p;
22      compute new possible collisions involving p and add the corresponding events in Q;
23    end
24    if e is a collision between p1 and p2 then
25      compute new velocity of p1 and p2;
26      compute new possible collisions involving p1 and add the corresponding events in Q;
27      compute new possible collisions involving p2 and add the corresponding events in Q;
28    end
29  end
30 end

```

---

The discriminant  $\Delta = 4(\Delta\vec{p} \cdot \Delta\vec{v})^2 - 4 \times (\Delta\vec{v} \cdot \Delta\vec{v}) \times ((\Delta\vec{p} \cdot \Delta\vec{p}) - (r_1 + r_2)^2)$  must be positive for equation (1) to have real solutions. If so, we are only interested in the first solution of the equation (the second one representing the collision after the particles have been through one of each other) which is

$$\Delta t = -\frac{\Delta\vec{p} \cdot \Delta\vec{v} + \sqrt{(\Delta\vec{p} \cdot \Delta\vec{v})^2 - (\Delta\vec{v} \cdot \Delta\vec{v}) \times ((\Delta\vec{p} \cdot \Delta\vec{p}) - (r_1 + r_2)^2)}}{\Delta\vec{v} \cdot \Delta\vec{v}} \quad (2)$$

$\Delta t < 0$  means that the two particles are departing one from the other (they have collided in the past and will not collide again unless they have other collisions first). Of course, such “past” collisions will not be taken into account. Applying the principles of momentum and kinetic energy conservation gives us the new velocities  $\vec{v}_1'$  and  $\vec{v}_2'$  of the particles:

$$\begin{aligned} \vec{v}_1' &= \vec{v}_1 + \frac{2 \times m_2 \times (\Delta\vec{p} \cdot \Delta\vec{v})}{(m_1 + m_2) \times (r_1 + r_2)^2} \Delta\vec{p} \\ \vec{v}_2' &= \vec{v}_2 - \frac{2 \times m_1 \times (\Delta\vec{p} \cdot \Delta\vec{v})}{(m_1 + m_2) \times (r_1 + r_2)^2} \Delta\vec{p} \end{aligned}$$

## 6 Implementation details

### 6.1 Particles

Particles will be defined by a structure consisting of the following fields: position, velocity, mass, radius, number of collisions in which the particle has been implicated and color. Color is represented by an **int** value comprised between 0 and 7. You may define a structure representing a 2D vector, but this is not mandatory.

When computing the expected amount of time to a collision, if the collision does not exist (root of equation (1) is negative for instance), return **INFINITY**. This is a valid value for floating point numbers, and for any **double** value  $v, c < \text{INFINITY}$ . Pay attention when comparing floating point numbers with  $0.0$ :  $v < 0.0$  will give you (in most cases) the right result, but  $v == 0.0$  will not. Use an **EPS** value like  $1\text{E-}16$  and the absolute value function **fabs** to compare a floating point number to  $0.0$ : **fabs(v) < EPS**.

You should implement at least the following functions for particles:

- a function that updates a particle position given an amount of time
- a function that computes the expected time for a particle to collide a vertical wall (the same for an horizontal wall)
- a function that updates the velocity of a particle after the collision with a vertical wall (the same for an horizontal wall)
- a function that computes the expected time of the collision between two particles
- a function that updates the velocities of two particles after their collision

It is strongly advised to use pointers to particles as parameters of these functions. You may also write a function to print a particle (for debugging purpose for instance).

In order to validate your implementation, the following (minimal) test data is given: a set of particles represented on table 1 and collision times and new velocities on table 2 (v means vertical wall and h means horizontal wall). Beware, all values are given with a precision of  $10^{-6}$ .

You must provide in your project a `test-particle.c` file with a program showing that your implementation respects these test cases with a precision of  $10^{-4}$ . You must use the **assert** function to verify the assertions. In order to ease your implementation, you will find all values presented in table 2 in the `data/particles-tests.txt` file in your repository.

### 6.2 Read an initial situation from a text file

You will finally have to develop a loader for an initial situation. These files will be text files with the following format:

- the first line of the file is the description of the file and may not be used
- the second line of the file contains the number of particles

name	$\vec{p}$	$\vec{v}$	$m$	$r$
p1	(0.25, 0.25)	(0.5, 0)	0.5	1E-2
p2	(0.25, 0.25)	(-0.5, 0)	0.5	1E-2
p3	(0.25, 0.25)	(0, 0.5)	0.5	1E-2
p4	(0.25, 0.25)	(0, -0.5)	0.5	1E-2
p5	(0.25, 0.25)	(0.25, -0.4)	0.5	1E-2
p6	(0.50, 0.25)	(0, 0)	0.8	5E-3
p7	(0.75, 0.25)	(-0.25, 0)	0.5	1E-2
p8	(0.6, 0.8)	(0.25, -0.4)	0.8	5E-3

Table 1: Particles for tests

object A	object B	time to collision	new velocities
p1	v	1.48	(-0.5, 0.0)
p2	v	0.48	(0.5, 0.0)
p3	v	INFINITY	
p4	v	INFINITY	
p5	v	2.96	(-0.25, -0.4)
p1	h	INFINITY	
p2	h	INFINITY	
p3	h	1.48	(0.0, -0.5)
p4	h	0.48	(0.0, 0.5)
p5	h	0.6	(0.25, 0.4)
p1	p6	0.47	(-0.115385, 0.0) (0.384615, 0.0)
p1	p7	0.64	(-0.25, 0.0) (0.5, 0.0)
p1	p8	1.352274	(0.067993, -0.329141) (0.520004, -0.194287)
p7	p8	INFINITY	

Table 2: Time and new velocities for collisions

- each following line will describe a particle with the following format:

$x, y, vx, vy, m, r, c$

where  $(x, y)$  is the position of the particle,  $(vx, vy)$  its velocity,  $m$  its mass,  $r$  its radius and  $c$  its color. You may also add the color of the particle if you want. You must allocate *dynamically* enough memory to store all particles.

You will find examples of such files in the `data` directory of your repository. Look at appendix E to understand how to read data from a file.

You must provide a test for your loader in a `test-loader.c` file (simply print the features of the particles defined in the text file). Your program must take a filename as a command line argument (see appendix D to understand how to pass arguments from the command line). There are several example files in the `data` directory of your repository.

### 6.3 Events

Events will be simple structures with the following fields: a time, a pointer to a particle `p_particle_a`, a pointer to a particle `p_particle_b`, the number of collisions in which the particle pointed by `p_particle_a` has been implicated in at the creation of the event, the same for the particle pointed by `p_particle_b`.

You may wonder how to define refreshing events and events representing a collision with a wall. We will use the fact that fields are pointers and adopt the following convention:

- if `p_particle_a` and `p_particle_b` are `NULL`, then the event is a refreshing event

- if `p_particle_a` (resp. `p_particle_b`) is `NULL`, then the event is a collision with a vertical (resp. horizontal) wall
- otherwise the event is a collision between two particles

You must define a function to create an event and a function to verify if an event is valid or not using the `count` field of particles.

You must provide in your project a `test-event.c` file with a program showing that your function verifying the validity of an event is correct. You can use particles P1 and P2 previously defined and test

- collision events with a horizontal wall and p1. Check that event with count for p1 equal to 0 is valid, event with count for p1 equal to 1 is not valid, increment p1 count and check that event with count for p1 equal to 1 is now invalid.
- the same for collision events with a vertical wall and p2.
- collisions events for p1 and p2. Check that events with p1 and p2 with both counts equal to 1 are valid, increment p1 count, check that previous events are not valid and check that event with correct count is valid.

## 6.4 Naive simulation

### 6.4.1 Main algorithm of naive simulation

You will first develop a naive simulation using the algorithm presented in section 3. A more detailed view is presented on algorithm 6.1.

---

#### Algorithm 6.1: main loop of the event-driven naive simulation

---

**Input:** a set of initialized particles

**Input:** a time limit  $\mathcal{T}$

**Input:** a time period  $\mathcal{T}_R$  for GUI refreshing

```

1 time := 0;
2 time_next_redraw :=  $\mathcal{T}_R$ ;
3 while time <  $\mathcal{T}$  do
4   draw the particles on the GUI;
5   while time < time_next_redraw do
6     next_event := redrawing event at time_next_redraw;
7     foreach particle p do
8       foreach possible collision c between p and walls or other particles do
9         if time to collision c is less than time of next_event then
10          | next_event := c;
11        end
12      end
13    end
14    foreach particle p do
15      | update p position up to event next_event;
16    end
17    treat event next_event (update velocities of particles if necessary) ;
18    time := time of next event ;
19  end
20  time_next_redraw = time +  $\mathcal{T}_R$  ;
21 end

```

---

You must implement your naive simulation in a `simulation-naive.c` file.

### 6.4.2 Graphical display

The GUI you will have to develop will use the SDL library and functions declared and documented in the `disc.h` file. A complete example of use is given in the `snow.c` file, you can build the corresponding executable with `make snow` and execute it with `./snow`. A window size of  $900 \times 900$  is a good choice, at least on the SI computers.



When using SDL, you will have memory leaks (please complain to SDL developers). In order to correctly use Valgrind to check your memory allocation/deallocation, you should deactivate SDL GUI. This can be done using the `NOGUI=1` option with your Makefile.

For instance

```
make snow NOGUI=1
valgrind --leak-check=full ./snow
```

should produce no errors, whereas

```
make snow
valgrind --leak-check=full ./snow
```

produces lots of errors.

Do not forget to use the `NOGUI=1` option when compiling your program to be tested with Valgrind.

### 6.4.3 Naive simulation program

You must write a program in a `clash-of-particles-naive.c` file that launches your naive simulation taking a file name on command line to use a file with a starting configuration, e.g.

```
./clash-of-particles-naive ./data/newton-simple.txt
```

Your program should use a simulation total time of 50000 time units and a refreshing step of 2 time units.

If you want to create your own data files, use the following values for parameters in order to have a viewable simulation on a  $900 \times 900$  screen:

particle velocity	components value between $-0.0005$ and $0.0005$
particle radius	value between $0.004$ and $0.01$
particle mass	value between $0.4$ and $0.8$

## 6.5 Simulation with priority queue

Priority queue are classically implemented using an array, but we will choose in this project to implement our priority queue with a [binary heap](#).




### 6.5.1 Binary heaps

We will consider in the following priority queues containing `int` values and that the `extract` operation is a `extract_min` operation (we are interested in the element with the minimum value in the priority queue).

Binary heaps are an implementation of priority queues that use a binary tree. They respect the following two properties:

- they are *complete* trees, i.e. all levels of the tree are full except the last one which is filled from the left
- each node of the tree has a lower value than all its sons

For instance, figure 1 is a binary heap: the tree is complete and all nodes respect the second property. Notice that the

node  cannot be placed under the node  or the node  even if the second property is respected as the last level of the tree has to be filled from the left.

Of course, the minimum value is at the root of the tree and we can get it in  $O(1)$ , but how can we guarantee  $\log n$  complexity for `extract_min` and `insert` operations?

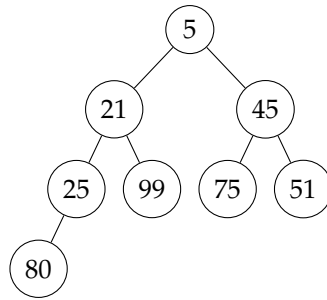


Figure 1: An example of binary heap

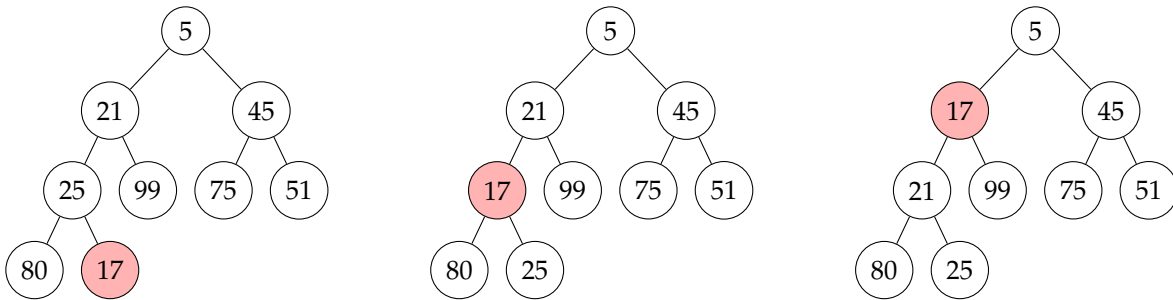


Figure 2: Inserting the value 17 in the previous binary heap

- inserting a new node is done as follows: find the first free place in the tree, put a node with the new value here and make the node go up in the tree while its parent node is greater than it. For instance, the steps for inserting the value 17 in the previous binary heap are shown on figure 2.

As the tree is complete, the node will go up in at most  $\log_2(n)$  steps.

Notice that it should be a good idea for each node to know its parent node in order to ease the exchange.

- the `extract_min` operation works as follows: get the minimum value at the root of the binary heap, put the last value in the binary heap at the root and make it goes down the tree while it is greater than at least one of its child (exchange it with the child with the lowest value). For instance, the steps for extracting the minimum value from the tree presented on figure 1 are shown on figure 3.

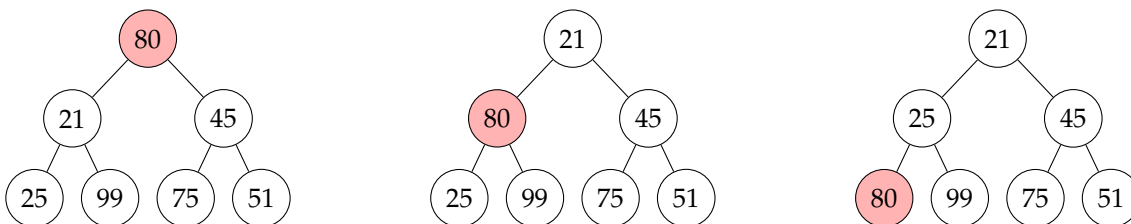


Figure 3: Extracting the minimum value in the previous binary heap

Of course, to implement these algorithm, you should know where is the last node in the binary heap or where to insert a new node! It is rather simple. Let us look at an example. Suppose that the nodes are numbered as in figure 4 (this numbering is rather natural). Consider now the 9<sup>th</sup> node in the heap. 9 in binary notation is 1001. Remove the leading 1, start from the root and consider that 0 means “left” and 1 means “right”. You arrive exactly at the ninth node! Try it on several examples, it works 😊!

Algorithm 6.2 presents the algorithm to find the father node of node numbered  $n$  in a binary heap. Notice that:

- `>>` is a bit shifting operation and `&` is a bitwise AND operator (see [bitwise operations in C](#))
- $\log_2$  is implemented as the `log2` operation in the `math.h` library. To use it in C, you must cast the result to `int`. For instance

```
int depth = (int) log2(n);
```



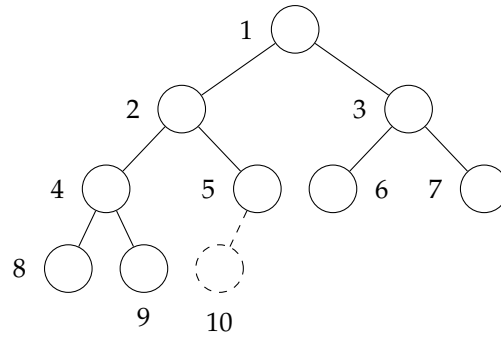


Figure 4: Numbering the nodes of a binary heap

**Algorithm 6.2:** finding the father of node  $n$  ( $n > 1$ )

---

```

1 depth =  $\log_2(n)$ ;
2 initialize p_father_node with a pointer to the root of the tree;
3 for  $i \in \{\text{depth} - 1, \dots, 1\}$  do
4   if  $(n \gg i) \ \& \ 1 = 0$  then
5     p_father_node := pointer to left child of p_father_node;
6   else
7     p_father_node := pointer to right child of p_father_node;
8   end
9 end
10 return p_father_node;
```

---

**6.5.2 Structures needed to represent a binary heap**

In order to represent a binary heap in C, you will need:

- a node structure containing a value, a pointer to its left child (possibly `NULL`), a pointer to its right child (possibly `NULL`) and a pointer to its parent node (possibly `NULL` for the root of the binary heap)
- a structure representing the heap containing the number of nodes in the heap and a pointer to the root node of the heap

For instance, the heap presented in figure 1 could be represented in memory as in figure 5.

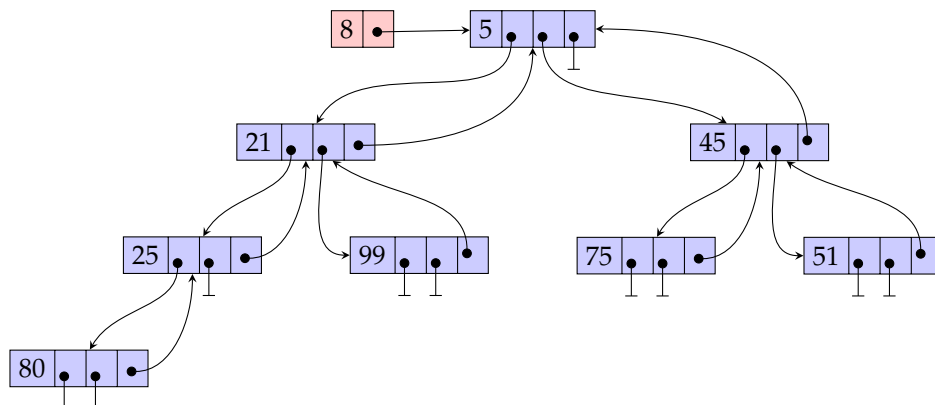


Figure 5: A possible memory representation for binary heap from figure 1

**6.5.3 Binary heap validation**

In order to validate your implementation, you will produce two programs:

- a `test-heap-correctness.c` file that contains a program showing that your heap is correct. In this program, you will insert 50 events with time from 1 to 50 in a heap and verify that when you call `extract-min` 50 times on the heap you get the events in the right order.
- a `test-heap-complexity.c` file that contains a program showing that the operations `insert` and `extract-min` have the right complexity. In order to do so, you will measure the time needed to insert  $n$  values and to call `extract-min`  $n$  times, with  $n$  starting at 20 and finishing at 10000000 for instance (simply double  $n$  at each experiment).

To measure time, use the `clock` function declared in `time.h` (you should therefore include this file). For instance, here is a snippet of code measuring the time to call a function `foo`  $n$  times:

```
clock_t start = clock();

for (int i = 0; i < n; i++) {
    foo();
}

clock_t end = clock();

double elapsed_time = (double) (end - start) / CLOCKS_PER_SEC;
```

You will write the results in a `data_complexity_heap.csv` file. Each line of the file will have the following format:

```
nb,time_to_insert,time_to_extract
```

where `nb` is the number of values inserted and extracted. Refer to appendix E to understand how to write data to a file. The Python script `plot_heap_complexity.py` in the `scripts` directory will plot your data.

#### 6.5.4 Simulation with binary heaps

The simulation will execute the loop presented on algorithm 4.1 until a time limit is reached (therefore it is not necessary to take into account events that are beyond this time limit). Your simulation must be implemented in a `simulation-bh.c` file.

#### 6.5.5 Programs using binary heap

You should write a program in `clash-of-particles-bh.c` using the same guidelines as these given in section 6.4.3.

### 6.6 Development plan

You may apply the following development plan:

1. `particle.h`, `particle.c` and `test-particle.c`: define necessary structures and functions as defined in section 6.1.
2. `loader.h`, `loader.c` and `test-loader.c`: define the loader and test it.  
Do not forget to free memory if necessary!
3. `event.h`, `event.c` and `test-event.c`: define necessary structure for events, the function to verify if an event is valid and test it accordingly to section 6.3.
4. `simulation-naive.h`, `simulation-naive.c`, and `clash-of-particles-naive.c`: implement naive simulation and an application to verify it.
5. `heap.h`, `heap.c`, `test-correctness-heap.c` and `test-complexity-heap.c`: define and test a binary heap.
6. `simulation-bh.h`, `simulation-bh.c`, and `clash-of-particles-bh.c`: implement simulation with binary heap and an application to verify it.



Do not try to implement several functionalities at once, it will not work! Implement **and test** your code incrementally.  
See section 9 to see how you will be graded.

## 6.7 Summary of files to produce

	specification	implementation
particle	particle.h	particle.c
loader	loader.h	loader.c
event	event.h	event.c
test of particles impl.		test-particle.c
test loader		test-loader.c
test of events		test-event.c
naive simulation	simulation-naive.h	simulation-naive.c
naive app		clash-of-particles-naive.c
heap	heap.h	heap.c
correctness test of heap		test-correctness-heap.c
complexity test of heap		test-complexity-heap.c
simulation with binary heap	simulation-bh.h	simulation-bh.c
app with binary heap		clash-of-particles-bh.c

## 7 Acknowledgements

This project is inspired from an exercise done during Prof. Sedgewick course on algorithms. I strongly recommend to read his book [3] or to attend his online lectures [4, 5] if you are interested in algorithms and programming.

## 8 Work to do and requirements

What you have to do and the project requirements are summarized in the following points. The project is due **03/18/2022 23h59**. **The code retrieved from your repository at this date will be considered as the final version of your project.**

You will have to commit working code for the naive simulation of particles evolution on **03/04/2022 23h59**. If not, penalty will be applied on your final grade.

### 8.1 Requirements about functionalities

- [R1] you must implement the following functionalities: a loader for an initial position, an event-driven naive simulation of particles, a heap to handle events, an event-driven simulation with a binary heap and tests as defined in section 6.
- [R2] you may implement one of the following extensions: add forces between particles, zoom to a particular region of the simulation using SDL, computing  $\pi$  digits using your simulation program (see <https://www.youtube.com/watch?v=jsYwFizhncE>). You may also find yourself an interesting extension. These extensions will be taken into account in your final grade only if the basic functionalities are correctly working.

### 8.2 Requirements about implementation

- [R3] your implementation must be written in C.
- [R4] your code must compile with the `-std=c99 -Wall -Werror` options and at least GCC version 10.
- [R5] you must use the provided Makefile (see appendix B). The rule `compile-all` should compile all your executables and tests. The rule `launch-tests` should run all your tests. See section B for more details.
- [R6] your code must work on the ISAE computers under Linux.

[R7] you must write

- a program in the `clash-of-particles-naive-file.c` that loads a file representing an initial situation and simulates the particles evolution in a GUI with the naive algorithm
- a program in the `clash-of-particles-naive-random.c` that takes a number of particles to randomly generate and simulates the particles evolution in a GUI with the naive algorithm
- a program in the `clash-of-particles-bh-file.c` that loads a file representing an initial situation and simulates the particles evolution in a GUI with the algorithm using binary heap
- a program in the `clash-of-particles-bh-random.c` that takes a number of particles to randomly generate and simulates the particles evolution in a GUI with the algorithm using binary heap

[R8] you must write the following test files as defined in section 6: `test-particle.c`, `test-loader.c`, `test-event.c`, `test-correctness-heap.c` and `test-complexity-heap.c`.

[R9] you may add programs to test your implementation. They must be located in files beginning with `app-`.

[R10] your data files (initial state) should be located in the `data` directory and have a `.txt` suffix.

[R11] your code must not produce errors when using the Valgrind tool.

[R12] you may reuse code **YOU** have implemented during the lab sessions.

### 8.3 Requirements about code conventions

[R13] your header files must be documented, possibly with the Doxygen tool (look at lab sessions and the corresponding refcard on LMS). Do not document preconditions or postconditions, only functions behavior, their returns and their parameters. You should add normal C comments in your code to explain your implementation if they are relevant.

[R14] your C code must follow the code conventions explained during the lecture.

### 8.4 Requirements about your repository

[R15] you must commit your work at each successful functionality implementation. An intelligible message should be added to the commit.

[R16] you must follow the following directories convention: your `.c` files must be in the `src` directory or in the `tests` directory for tests and your `.h` files in the `include` directory. You may put the `.o` and executable files wherever you want. See section C for more details.

## 9 Grading

Your final grade will be calculated as presented on table 3. **Beware, your tests will be used to evaluate your work!** A successful extension implementation will give you up to 4 points **if the basic functionalities are correctly implemented.**

Notice that you may have a really good grade even if you do not implement all the functionalities: **the quality of your code is really important!**



Your code will be analyzed by JPlag [2], a code plagiarism detector. **DO NOT TAKE CODE FROM ANOTHER STUDENT, EVEN IF YOU TRY TO DISGUISE IT.**  
If you are convinced of plagiarism, you will obtain the “R” grade for the module.

## A Testing functions

In this section, I will present how to write tests for your project. If you want to test a function or a property for the project, you should

- create a file whose name begins with `test-` in the `tests` directory

Part	Details	Grade	Comments
Basics	particles	2	
	loader	2	
	event	1	
Naïve simulation	simulation	4	
Binary heap simulation	binary heap	4	
	simulation	1	
Code quality	overall quality	4	variables naming, documentation, no unuseful code duplication etc.
	tests quality	2	
	your code does not compile	-5	
	your code produces errors with valgrind	-2	maximum malus, depends on the number of errors
	incorrect use of SVN	-1	unuseful commit messages etc.

Table 3: Grade details

- add a rule to build the corresponding executable in your Makefile and add the test to the global `launch-tests` variable (see section B)
- use the C `assert` macro to verify properties
- define one or several functions, each one corresponding to a specific test or property
- call these functions into your `main` function
- use simple `printf` commands to indicate that the tests are satisfied

You will find in your `tests` directory a `test-dummy.c` file with two dummy tests about integer arithmetics that can serve as an example. There is a corresponding rule in your Makefile to build the executable.

If your test needs external information such as a filename, use `main` arguments (cf. section D). Do not add the test to the `launch-tests` variable of your Makefile.

## B Makefile

In order to compile your applications and tests, you should use the Makefile provided in your repository. Here are some explanations:

- the `doc` rule generates the Doxygen documentation from your `.h` files. The documentation is located in the `doc` directory
- the `clean` rule removes all generated `.o` files and executables
- the C files in the `src` and `tests` directories can be automatically compiled. For instance, if you want to produce the `.o` file from `src/my_file.c`, just execute the following command:

```
make my_file.o
```

- you should put all the `.o` file names that can be generated from your `.c` files as prerequisites of the `check-syntax` rule. Therefore, executing `make check-syntax` should compile all your C source files
- you should create a compilation rule for each of your applications or tests (see examples). You can use the `.o` files as prerequisites as the provided rules generate them from your `.c` files. Do not forget to add the needed header files as prerequisites

- you should add all your executables (applications and tests) as prerequisites of the `compile-all` rule. Therefore, executing `make compile-all` should generate all your applications and tests
- you should add all your test executables in the `ALL_TESTS` variables. Therefore, executing `make launch-tests` should execute all your tests.

**Beware:** do not add tests that need an argument on the command line

- the `OPT` variable can be used to add the `-O3 -f1` to options to GCC (and thus optimize aggressively your code). To use it, call make with `OPT=1` as for instance

```
make compile-all OPT=1
```

- you can put debug messages in your code using `printf`. In order to avoid “polluting” the execution of your applications, you can encapsulate these messages in a preprocessor directive:

```
#ifdef DEBUG
printf("this is a debug message...\n");
#endif
```

When compiling your applications, all code between the `#ifdef DEBUG` and `#endif` directives will be wiped except if you use make with `DEBUG=1` as for instance

```
make compile-all DEBUG=1
```

- the `svn-add-all-files` rule add all `.c`, `.h` and `.txt` files that are in the expected directories. Do not hesitate to use it!

## C Repository organization

Your repository is organized as follows:

- the `include` directory contains your header files (`.h` files)
- the `src` directory contains your C source files, except tests
- the `tests` directory contains your C test files
- the `doc` directory contains the documentation generated by Doxygen

You should compile your files using the provided Makefile (see section B).

You must include a `README.txt` file at the root of your repository to give some explanations on your project, particularly how to run executables that need command-line arguments.

## D Passing parameters to a program from command line

You have seen in session M0 that you can pass parameters to a Python program using the `sys.argv` list. There is a similar mechanism in C when you use the following signature for `main`:

```
int main(int argc, char *argv[])
```

`argc` represents the length of `argv` and `argv` is an array of strings containing the parameters given on the command line and the name of the executable in position 0.

Let us take an example: let us suppose that you have such a `main` function in a C file compiled to a `example-main` executable. Then, when executing

```
./example-main hello world
```

- `argc` has the value 3, as there is two parameters
- `argv[0]` is `./example-main`, the name of the executable
- `argv[1]` is `hello`
- `argv[2]` is `world`

You will find in your `src` directory an `example-main.c` file that prints these informations when executing the corresponding executable.

## E File Input/Output in C

If you want to read or write data to a file in C, you should use the structures and functions provided by the `stdio.h` header file. We will present here only simple cases when wanting to read or write formatted data from or to a text file. You may find more information in [6] or manual pages for the functions that are described in the following.

### E.1 Opening and closing files

In order to read from files (or write to files), you must first open them. `stdio.h` provides a type, `FILE`, to represent a data stream from or/and to a particular file. Let us suppose that we want to read the content of a file `data.txt`. To open the file, we will use the `fopen` function and get a pointer to a `FILE` object:

```
FILE *p_file = fopen("data.txt", "r+");
```

Some remarks:

- `"data.txt"` represents the path to the file, it is here a relative path (the file `data.txt` must be in the current directory).
- `"r"` is the mode of the stream and specifies what you want to do with the file. You will mainly use the following modes:
  - `"r"` to read a file
  - `"w"` to write to a file
  - `"r+"` to read and write to a file
- if there are some errors when opening the file, the returned pointer is `NULL`. You should therefore test `p_file`.

When you have finished working with a file, **you should close it**. This is particularly true when writing to a file, as the file may have been put in central memory and the modifications you have done to the file might not be committed by the operating system (if you want to know more, you may refer to [1]). To close a file, simply call the `fclose` function on your pointer:

```
fclose(p_file);
```

### E.2 Reading from files

#### E.2.1 Reading formatted input

If your file contains only **readable characters** and you know that data will follow a given format, then you may use the `fscanf` function. `fscanf` behaves like the `scanf` function we have seen in lab M1, but it works on files instead of standard input.

Let us look at the `fscanf` signature. It needs:

- a `FILE *` pointer to the file you want to read from
- a **format string** in the same format than the `printf` function
- several pointer variables to store what you read

Let us suppose that we want to read **int** values from a file with the following format:

```
2 - 4
4 - 12
5 - 42
13 - 2323
```

Each line of the file has two **int** values separated by the string `" - "`. Let us suppose that `first_value` and `second_value` are **int** variables in which we want to store the read values, then the call to `fscanf` may be:

```
fscanf(p_file, "%d - %d", &first_value, &second_value);
```

The format string is `"%d - %d"`: we expect an integer, then a space, then `-`, then a space, then another integer. `fscanf` will return the number of input items successfully matched and assigned to the variables. Therefore, if `fscanf` returns 2, then the line has been correctly read and the variables assigned.

You may wonder how you may know when the file has been completely read. The `EOF` special value is returned by `fscanf` when it encounters the end of the file.



When using `fscanf`, the pointer `p_file` changes! After calling for instance the previous instruction, `p_file` will then “point” after the two integers. Therefore, you should not reuse directly `p_file` supposing that it points at the beginning of the file.

You will find in the `src` directory of your repository a simple program in the `read-file-formatted.c` file that tries to print all lines of a file respecting the previous syntax (cf. listing 1). You can compile it with the following Makefile command:

```
make read-file-formatted
```

You can execute it on the file `data-toread-formatted.txt` from your `data` directory with the following command<sup>1</sup>:

```
./read-file-formatted data/data-toread-formatted.txt
```

Try it on different inputs to verify that it works correctly, particularly when there are errors in the data file (see `data-toread-error.txt` in the `data` directory).

#### Listing 1: A simple program to read formatted text files

```
/**
 * @file read-file-formatted.c
 *
 * @brief Simple program to explain how to read
 *        formatted data from file
 *
 * @author C. Garion
 *
 */

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    FILE *p_file = NULL;

    p_file = fopen(argv[1], "r");

    if (p_file == NULL) {
        fprintf(stderr, "Cannot read file %s!\n", argv[1]);
        exit(EXIT_FAILURE);
    }

    int first_int    = 0;
    int second_int   = 0;
    int line_nb      = 1;
    int fscanf_result = 0;

    fscanf_result = fscanf(p_file, "%d - %d", &first_int, &second_int);
```

<sup>1</sup>data-toread-formatted.txt must be in the `data` directory in the command, but you can use relative or absolute paths to call `read-file-formatted` on files that are not in the current directory.



```

while (fscanf_result != EOF) {
    if (fscanf_result != 2) {
        fprintf(stderr, "Line number %d is not syntactically correct!\n",
                line_nb);
        exit(EXIT_FAILURE);
    }

    printf("first value at line %d: %d\n", line_nb, first_int);
    printf("second value at line %d: %d\n", line_nb, second_int);

    line_nb = line_nb + 1;
    fscanf_result = fscanf(p_file, "%d - %d", &first_int, &second_int);
}

fclose(p_file);

p_file = NULL;

return 0;
}

```

### E.2.2 Reading text files

If you want to read simple text files<sup>2</sup>, do not use `fscanf` but rather the `fgets` function. `fgets` takes as input:

- a `char *` string `s` that is used as a buffer, i.e. `s` will contain after the call to `fgets`
- an `int` value `size` that indicates the number of characters to be *at most* read
- a `FILE *` argument that represents the file to be read

Notice that `fgets` stops when encountering a newline or the `EOF` character representing the end of the file. Notice also that you should know the maximum length of the lines of the file in order to have correct `s` and `size` arguments. `fgets` will return `NULL` when encountering the end of the file.

Let us take an example and consider the following file:

```

Sing, O goddess, the anger of Achilles son of Peleus, that brought
countless ills upon the Achaeans. Many a brave soul did it send
hurrying down to Hades, and many a hero did it yield a prey to dogs
and vultures, for so were the counsels of Jove fulfilled from the day
on which the son of Atreus, king of men, and great Achilles, first
fell out with one another.

```

We know that the length of each line will not be more than 80 characters, therefore we could read the file and prints each line as presented on listing 2. Compile the executable with the corresponding Makefile rule and use it on the `data/iliad.txt` file in your repository.

#### Listing 2: A simple program to read text files

```

/**
 * @file read-file-text.c
 *
 * @brief Simple program to explain how to read
 *        text lines from file
 *
 * @author C. Garion
 *
 */

```

<sup>2</sup>Or part of a file that contains only text

```

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    FILE *p_file = NULL;

    p_file = fopen(argv[1], "r");

    if (p_file == NULL) {
        fprintf(stderr, "Cannot read file %s!\n", argv[1]);
        exit(EXIT_FAILURE);
    }

    // declaration of a buffer of 81 chars: we suppose that each line
    // of the file has no more than 80 chars and we add 1 char for the
    // final '\0' character
    char buffer[81];
    int line = 0;

    // while not having encountered end of file, read lines
    while (fgets(buffer, 80, p_file) != NULL) {
        line++;

        printf("%2d: %s", line, buffer);
    }

    fclose(p_file);

    p_file = NULL;

    return 0;
}

```

### E.3 Writing to files

If you want to write formatted content to a file, you must first open the file with the `"r+"` or `"w"` modes with `fopen`. You then use the `fprintf` function. It behaves like the `printf` function, but it takes as first argument a `FILE *` pointer to the file you want to write to.

For instance, the program presented on listing 3 writes the factorial of the 10 first natural numbers to the file `fact.txt`. The program is available in the `src` directory of your repository and a compilation target is provided in your Makefile.

Listing 3: A simple program to write some computations in a text file

```

/**
 * @file write-fact.c
 *
 * @brief Simple program to explain how to write
 *        text into a file
 *
 * @author C. Garion
 *
 */

#include <stdio.h>
#include <stdlib.h>

int main(void) {

```

```
FILE *p_file = NULL;

p_file = fopen("fact.txt", "w");

if (p_file == NULL) {
    fprintf(stderr, "Cannot write to file fact.txt!\n");
    exit(EXIT_FAILURE);
}

int fact = 1;

for (int i = 0; i < 10; i++) {
    fprintf(p_file, "%d! = %d\n", i, fact);

    fact = fact * (i + 1);
}

fclose(p_file);

p_file = NULL;

return 0;
}
```

## References

- [1] Ulrich Drepper. "What Every Programmer Should Know About Memory". Nov. 21, 2007. URL: <http://www.akkadia.org/drepper/cpumemory.pdf>.
- [2] Karlsruhe Institute of Technology. *JPlag – Detecting Software Plagiarism*. 2016. URL: <https://jplag.ipd.kit.edu/>.
- [3] Robert Sedgewick and Kevin Wayne. *Algorithms*. 4th. Pearsons Education, 2011. URL: <https://algs4.cs.princeton.edu/home/>.
- [4] Robert Sedgewick and Kevin Wayne. *Algorithms, Part I*. 2018. URL: <https://online.princeton.edu/node/201>.
- [5] Robert Sedgewick and Kevin Wayne. *Algorithms, Part II*. 2018. URL: <https://online.princeton.edu/node/166>.
- [6] Wikibooks. *C Programming, File IO — Wikibooks, The Free Textbook Project*. 2015. URL: [https://en.wikibooks.org/wiki/C\\_Programming/File\\_IO](https://en.wikibooks.org/wiki/C_Programming/File_IO).

## License CC BY-NC-SA 3.0



This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported license (CC BY-NC-SA 3.0)

You are free to Share (copy, distribute and transmute) and to Remix (adapt) this work under the following conditions:



**Attribution** – You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).



**Noncommercial** – You may not use this work for commercial purposes.



**Share Alike** – If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

See <http://creativecommons.org/licenses/by-nc-sa/3.0/> for more details.