Project on performance oriented programming

J. Daniel Garcia (coordinator)
Computer Architecture
Computer Science and Engineering Department
University Carlos III of Madrid

2023

1 Objective

The fundamental goal of this project is that students get familiar with **sequential programs optimizations**.

Specifically, the project will focus in the development of sequential software using the C++ programming language (including improvements up to C++20).

2 Introduction

Fluid flows simulation can be achieved by multiple methods. One of them is Smoothed Particle Hydrodynamics (SPH). SPH solves the Navier-Stokes equations at time steps. A fluid is modeled as a collection of particles. Those particles have interactions impacting their acceleration, position and velocity.

Only particles at short distance have real interaction. To speedup simulation time, long distance interactions are discarded. To achieve this, the 3D space is divided in a grid of blocks and only particles in blocks in the nearby are considered.

Simulation includes the following steps:

- Reading initial state of the simulation.
- Simulation of particles for each time step or iteration:
  - Repositioning of particles in the grid.
  - Computing forces and accelerations for each particle.
  - Processing particle collisions with boundaries.
  - Particles movement.
  - Reprocessing box boundaries interactions.
- Writing final state of the simulation.

3 General overview

3.1 Introduction

The goal of this project is to develop an application that performs a fluid simulation during a number of time steps.

The application takes the following command line parameters:

- **Iterations**: Number of iterations or time steps to be computed.
- **Input**: Data file with the description of the initial state of the fluid.
- **Output**: Output file with the description of the final state of the fluid.

IMPORTANT: Although input and output files contain information using single precision floating point numbers, all internal computations shall use double precision floating point numbers.

3.2 File format

A fluid file contains all the information of fluid state at a given point in time. Files are stored in binary format with the following requirements:

- **Integer**: All integer values use 4 bytes.
- **Floating-point**: All floating point values use IEEE-754 single precision representation.

A file is composed of a **header** and a **body**. The header includes general information while the body contains information for each particle in the fluid.

The header contains the fields as specified in Table 1.

Field	Type	Description
ppm	**Floating-Point**	Particles per meter
np	**Integer**	Number of particles

Table 1: Fields in the file header.

Note that the **number of particles** specifies how many particles are stored in the rest of the file. The information and format for each particle in the file **body** is specified in Table 2.

Field	Type	Description
px	**Floating-Point**	Position x-coordinate.
py	**Floating-Point**	Position y-coordinate.
pz	**Floating-Point**	Position z-coordinate.
hvx	**Floating-Point**	Vector hv x-coordinate.
hvy	**Floating-Point**	Vector hv y-coordinate.
hvz	**Floating-Point**	Vector hv z-coordinate.
vx	**Floating-Point**	Velocity x-coordinate.
vy	**Floating-Point**	Velocity y-coordinate.
vz	**Floating-Point**	Velocity z-coordinate.

Table 2: Fields in the file body for each particle.

3.3 Simulation constants

Table 3 shows the constants used in the simulation.

Constant	Description	Value
r	**Radius multiplier**	1.695
ρ	**Fluid density**	10^3
p_s	**Stiffness pressure**	3.0
s_c	**Stiffness collisions**	$3 \cdot 10^4$
d_v	**Damping**	128.0
μ	**Viscosity**	0.4
d_p	**Particle size**	$2 \cdot 10^{-4}$
Δt	**Time step**	10^{-3}

Table 3: Simulation scalar constants.

In addition, Table 4 shows vector constants to be used in the simulation. Note that the **external acceleration** ($\vec{g}$) is the gravity acceleration which in this example acts across the y-axis. The **box upper bound** ($\vec{b}_{max}$) and **box lower bound** ($\vec{b}_{min}$) define two opposite vertex in a rectangular cuboid. This is the closed domain where the simulation happens. $\vec{b}_{max}$ is given by vector $(x_{max}, y_{max}, z_{max})$ and $\vec{b}_{min}$ is given by vector $(x_{min}, y_{min}, z_{min})$.

Constant	Description	x	y	z
$\vec{g}$	**External acceleration**	0.0	-9.8	0.0
$\vec{b}_{max}$	**Box upper bound**	x_{max} = 0.065	y_{max} = 0.1	z_{max} = 0.065
$\vec{b}_{min}$	**Box lower bound**	x_{min} = -0.065	y_{min} = -0.08	z_{min} = -0.065

Table 4: Simulation vector constants.

Figure 1 shows a possible grid divided in blocks and box bounds b_{min} and b_{max}.

Note that some mathematical constants (e.g. π) may be needed. In that case the preferred value shall be the one provided by the C++20 standard library. Please, refer to the namespace **std::numbers** for such constants (see `https://en.cppreference.com/w/cpp/header/numbers`).

3.4 Simulation parameters

A number of parameters depend on the amount of particles per meter (**ppm**).

The particle mass m depends on the density ρ and the amount of particles per meter (**ppm**).

$$m = \frac{\rho}{ppm^3}$$

Another relevant parameter is the **smoothing length** (h), which depends on the **radius multiplier** (r) and amount of particles per meter (**ppm**).

$$h = \frac{r}{ppm}$$

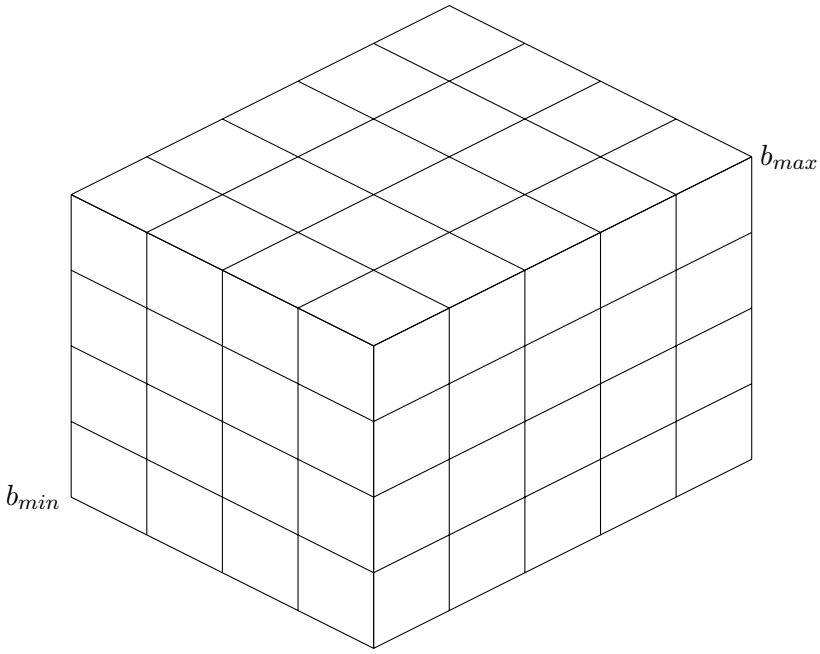

Figure 1: Cube dimension.

3.5 The simulation grid

To avoid excessive computations the whole simulation box is divided in smaller blocks. Note that the simulation box is a rectangular cuboid with different lengths in each of the three dimensions.

To determine the number of blocks in each dimension the length int that dimensions is divided by parameter h.

$$n_x = \lfloor \frac{x_{max} - x_{min}}{h} \rfloor$$
$$n_y = \lfloor \frac{y_{max} - y_{min}}{h} \rfloor$$
$$n_z = \lfloor \frac{z_{max} - z_{min}}{h} \rfloor$$

The size of grid block $\vec{s}\ (s_x, s_y, s_z)$ is then given by

$$s_x = \frac{x_{max} - x_{min}}{n_x}$$
$$s_y = \frac{y_{max} - y_{min}}{n_y}$$
$$s_z = \frac{z_{max} - z_{min}}{n_z}$$

Given a particle with position vector $\vec{p}$ the block indices are computed by considering its distance in each dimension to the box minimum coordinate and the size of a grid block ($\vec{s}$).

$$i = \lfloor \frac{p_x - x_{min}}{s_x} \rfloor$$
$$j = \lfloor \frac{p_y - y_{min}}{s_y} \rfloor$$
$$k = \lfloor \frac{p_z - z_{min}}{s_z} \rfloor$$

Note that for each dimensions the corresponding index value cannot be less than 0 or greater or equal than the corresponding number of blocks in that dimension:

$$i \in [0, n_x - 1]$$
$$j \in [0, n_y - 1]$$
$$k \in [0, n_z - 1]$$

If an index value is outside its corresponding range, it shall be adjusted to the corresponding limit.

The key role of simulation blocks is that only particles in the same block or contiguous blocks will be computed. In general, for any given block there are 26 contiguous blocks. This number may be lower when the block is in the limits of the grid. For example, when $i = j = k = 0$, the number of contiguous blocks is 7.

4 Simulation stages

The simulation application structure is the following:

1. **Parameter parsing**: Command line input parameters are parsed.
2. **Simulation initialization**: Initial simulation state is loaded from a file and data structures are initialized.
3. **Time steps processing**: For the specified number of time steps the new state for each particle is computed.
4. **Results storage**: Final simulation state is saved into a file.

4.1 Program arguments parsing

The application will be given exactly three parameters:

- **nts**: Number of time steps. This parameter specifies how many time steps shall be the simulation executed.
- **inputfile**: Input file with initial simulation state.
- **outputfile**: Output file where results shall be written.

For example, the command:

```
$ fluid 2000 init.fld final.fld
```

Loads the file **init.fld** runs **2000** time steps and generates an output file named **final.fld**.

If the number of arguments is not exactly three arguments, an error message shall be generated and the program shall exit with error code **-1**.

```
$ fluid
Error: Invalid number of arguments: 0.
$ fluid 4
Error: Invalid number of arguments: 1.
$ fluid 2000 init.fld
Error: Invalid number of arguments: 2.
$ fluid 2000 init.fld final.fld 45
Error: Invalid number of arguments: 4.
```

If the first argument is not an integer number, an error message shall be generated and the program shall exit with error code **-1**.

```
$ fluid hello init.fld final.fld
Error: time steps must be numeric.
```

If number of time steps is a negative number, an error message shall be generated and the program shall exit with error code **-2**.

```
$ fluid -3 init.fld final.fld
Error: Invalid number of time steps.
```

If the input file cannot be opened for reading, an error message shall be generated and the program shall exit with error code **-3**.

```
$ fluid 1 init.fld final.fld
Error: Cannot open init.fld for reading
```

If the output file cannot be opened for writing, an error message shall be generated and the program shall exit with error code **-4**.

```
$ fluid 5 init.fld final.fld
Error: Cannot open final.fld for writing
```

4.2 Simulation initialization

During initialization, the input file is opened and the main simulation parameters are read (particles per meter and number of particles). Then, other parameters depending on them are computed.

Those parameters include:

- The **smoothing length** (h).
- The **particle mass** (m).
- The **grid size vector** ($\vec{n} = (n_x, n_y, n_z)$).
- The **grid block size** ($\vec{s} = (s_x, s_y, s_z)$).

The information for each particle is read from the input file. Although input files use single precision for floating point, as soon as a value is read it shall be converted to double precision. Particles will be assigned a consecutive integer numeric identifier starting at **0**, keeping the same order that is used in the input file.

Once a particle has been read, its corresponding block in the grid is determined and the particle is added to the set of particles in that block.

If the number of particles found in the header is **0** or a negative value, the program shall exit with code **-5** and an error message shall be generated.

```
$ fluid 2000 init.fld final.fld
Error: Invalid number of particles: 0.
```

If the number of particles found in the file does not match with the parameter **number of particles** (**np**), an error message shall be generated and the program shall exit with error code **-5**. For example, if the header has the value **4750** for **np** but the file contains **4700** particles, the following message shall be sent to the standard error output.

```
$ fluid 2000 init.fld final.fld
Error: Number of particles mismatch. Header: 4750, Found: 4700.
```

As another example, if the header has the value **4750** for **np** but the file contains **5000** particles, the following message shall be sent to the standard error output.

```
$ fluid 2000 init.fld final.fld
Error: Number of particles mismatch. Header: 4750, Found: 5000.
```

If the file is successfully read, the values for the parameters shall be sent to the standard output.

```
$ fluid 2000 init.fld final.fld
Number of particles: 4800
Particles per meter: 204
Smoothing length: 0.00830882
Particle mass: 0.00011779
Grid size: 15 x 21 x 15
Number of blocks: 4725
Block size: 0.00866667 x 0.00857143 x 0.00866667
```

4.3 Simulation processing

The simulation processing stage performs computations on all particles in the simulation for every time step. Consequently, this stage is repeated as many times as time steps are specified.

For every time step the following sub-stages are executed:

1. Reposition each particle in the grid
2. Compute forces on each particle.
3. Process collisions.
4. Move particles.
5. Process boundaries.

4.3.1 Repositioning particles

During this stage coordinates for every particle are checked to make sure that they are in the right block of the simulation grid. If a particle is not in the right block, it is moved to the corresponding block.

Note that a the position of a particle may have changed. Consequently, it may be necessary to update the block in which the particle is stored.

4.3.2 Accelerations computation

The following steps are performed

1. Initialize accelerations.
2. Densities increase.
3. Density transformation.
4. Acceleration transfer.

Initialize densities and accelerations . During this step an initial value is given to the density (ρ_i) and the vector acceleration ($\vec{a_i}$) for each particle.

The density is initially set to **0**.

$$\rho_i = 0$$

The vector acceleration is initially set to the **external acceleration**.

$$\vec{a}_i = \vec{g}$$

Densities increase In this step, for every block in the simulation grid all contiguous blocks as well as the current block are considered. For every pair of particles i and j, the densities are increased.

$$\Delta\rho_{ij} = \begin{cases} (h^2 - \|\vec{p_i} - \vec{p_j}\|^2)^3 & \text{if } \|\vec{p_i} - \vec{p_j}\|^2 < h^2 \\ 0 & \text{if } \|\vec{p_i} - \vec{p_j}\|^2 \geq h^2 \end{cases}$$

Note that density increase $\Delta\rho_{ij}$ is to be applied to both particles: i and j.

$$\rho_i = \rho_i + \Delta\rho_{ij}$$
$$\rho_j = \rho_j + \Delta\rho_{ij}$$

However, special care needs to be taken to avoid that a particle is increased twice with the same increment (as $\Delta\rho_{ij} = \Delta\rho_{ji}$).

Density transformation For every computed density, a linear transformation is performed.

$$\rho_i = (\rho_i + h^6) \cdot \frac{315}{64 \cdot \pi \cdot h^9} \cdot m$$

Acceleration transfer In this step, for every block in the simulation grid all contiguous blocks as well as the current block are considered. For every pair of particles i and j, distances are evaluated.

If the two particles are near enough ($\|\vec{p_i} - \vec{p_j}\|^2 < h^2$) acceleration is updated.

$$dist_{ij} = \sqrt{\max(\|\vec{p_i} - \vec{p_j}\|^2, 10^{-12})}$$

$$\Delta\vec{a}_{ij} = \frac{(\vec{p_i} - \vec{p_j}) \cdot \frac{15}{\pi \cdot h^6} \cdot \frac{3 \cdot m \cdot p_s}{2} \cdot \frac{(h - dist_{ij})^2}{dist_{ij}} \cdot (\rho_i + \rho_j - 2 \cdot \rho) + (\vec{v_j} - \vec{v_i}) \cdot \frac{45}{\pi \cdot h^6} \cdot \mu \cdot m}{\rho_i \cdot \rho_j}$$

$$\vec{a}_i = \vec{a}_i + \Delta\vec{a}_{ij}$$

$$\vec{a}_j = \vec{a}_j - \Delta\vec{a}_{ij}$$

However, special care needs to be taken to avoid that a particle is increased twice with the same increment (as $\vec{a}_{ij} = \vec{a}_{ji}$).

4.3.3 Particles collisions

In this stage, possible collisions of particles with box walls are considered, in order to update vector acceleration. Given that indices for a grid block are c_x, c_y and c_z, the following options are considered:

- $c_x = 0$
- $c_x = n_x - 1$
- $c_y = 0$
- $c_y = n_y - 1$
- $c_z = 0$
- $c_z = n_z - 1$

Collisions with x-axis bounds When c_x is fixed, for any combination of c_y and c_z all particles are processed. First x coordinate of the new position is computed.

$$x = \vec{p}_x + \overrightarrow{hv}_x \cdot \Delta t$$

Then, the difference to the bound is compared to the **particle size** (d_p).

$$\Delta x = \begin{cases} d_p - (x - x_{min}) & \text{if } c_x = 0 \\ d_p - (x_{max} - x) & \text{if } c_x = n_x - 1 \end{cases}$$

If Δx is greater than 10^{-10}, the acceleration is updated, considering the **stiffness collisions** (c_s) and **damping** (d_v).

$$a_x = \begin{cases} a_x + (c_s \cdot \Delta x - d_v \cdot v_x) & \text{if } c_x = 0 \\ a_x - (c_s \cdot \Delta x + d_v \cdot v_x) & \text{if } c_x = n_x - 1 \end{cases}$$

Collisions with y-axis bounds When c_y is fixed, for any combination of c_x and c_z all particles are processed. First y coordinate of the new position is computed.

$$y = \vec{p_y} + \overrightarrow{hv_y} \cdot \Delta t$$

Then, the difference to the bound is compared to the **particle size** (d_p).

$$\Delta y = \begin{cases} d_p - (y - y_{min}) & \text{if } c_y = 0 \\ d_p - (y_{may} - y) & \text{if } c_y = n_y - 1 \end{cases}$$

If Δy is greater than 10^{-10}, the acceleration is updated, considering the **stiffness collisions** (c_s) and **damping** (d_v).

$$a_y = \begin{cases} a_y + (c_s \cdot \Delta y - d_v \cdot v_y) & \text{if } c_y = 0 \\ a_y - (c_s \cdot \Delta y + d_v \cdot v_y) & \text{if } c_y = n_y - 1 \end{cases}$$

Collisions with z-axis bounds When c_z is fixed, for any combination of c_x and c_y all particles are processed. First z coordinate of the new position is computed.

$$z = \vec{p_z} + \overrightarrow{hv_z} \cdot \Delta t$$

Then, the difference to the bound is compared to the **particle size** (d_p).

$$\Delta z = \begin{cases} d_p - (z - z_{min}) & \text{if } c_z = 0 \\ d_p - (z_{maz} - z) & \text{if } c_z = n_z - 1 \end{cases}$$

If Δz is greater than 10^{-10}, the acceleration is updated, considering the **stiffness collisions** (c_s) and **damping** (d_v).

$$a_z = \begin{cases} a_z + (c_s \cdot \Delta z - d_v \cdot v_z) & \text{if } c_z = 0 \\ a_z - (c_s \cdot \Delta z + d_v \cdot v_z) & \text{if } c_z = n_z - 1 \end{cases}$$

4.3.4 Particles motion

In this stage, all particles are updated.

$$\begin{aligned} \vec{p_i} &= \vec{p_i} + \overrightarrow{hv_i} \cdot \Delta t + \vec{a_i} \cdot (\Delta t)^2 \\ \vec{v_i} &= \overrightarrow{hv_i} + \frac{\vec{a_i} \cdot \Delta t}{2} \\ \overrightarrow{hv_i} &= \overrightarrow{hv_i} + \vec{a_i} \cdot \Delta t \end{aligned}$$

4.3.5 Box boundaries interactions

In this stage, collisions of particles with box boundaries are considered. Given that indices for a grid block are c_x, c_y and c_z, the following options are considered:

- $c_x = 0$
- $c_x = n_x - 1$
- $c_y = 0$
- $c_y = n_y - 1$
- $c_z = 0$
- $c_z = n_z - 1$

Collisions with x-axis bounds When c_x is fixed, for any combination of c_y and c_z all particles are processed.

$$d_x = \begin{cases} \vec{p}_x - x_{min} & \text{if } c_x = 0 \\ x_{max} - \vec{p}_x & \text{if } c_x = n_x - 1 \end{cases}$$

Only when d_x is negative there is a collision. In such case, p_x, v_x and hv_x shall be updated.

$$p_x = \begin{cases} x_{min} - d_x & \text{if } c_x = 0 \\ x_{max} + d_x & \text{if } c_x = n_x - 1 \end{cases}$$
$$v_x = -v_x$$
$$hv_x = -hv_x$$

Collisions with y-axis bounds When c_y is fixed, for any combination of c_x and c_z all particles are processed.

$$d_y = \begin{cases} \vec{p}_y - y_{min} & \text{if } c_y = 0 \\ y_{max} - \vec{p}_y & \text{if } c_y = n_y - 1 \end{cases}$$

Only when d_y is negative there is a collision. In such case, p_y, v_y and hv_y shall be updated.

$$p_y = \begin{cases} y_{min} - d_y & \text{if } c_y = 0 \\ y_{max} + d_y & \text{if } c_y = n_y - 1 \end{cases}$$
$$v_y = -v_y$$
$$hv_y = -hv_y$$

Collisions with z-axis bounds When c_z is fixed, for any combination of c_x and c_y all particles are processed.

$$d_z = \begin{cases} \vec{p}_z - z_{min} & \text{if } c_z = 0 \\ z_{max} - \vec{p}_z & \text{if } c_z = n_z - 1 \end{cases}$$

Only when d_z is negative there is a collision. In such case, p_z, v_z and hv_z shall be updated.

$$p_z = \begin{cases} z_{min} - d_z & \text{if } c_z = 0 \\ z_{max} + d_z & \text{if } c_z = n_z - 1 \end{cases}$$
$$v_z = -v_z$$
$$hv_z = -hv_z$$

4.4 Results storage

During storage, the output file is opened for writing in binary mode. First, the general parameters (particles per meter and number of particles) are written. Then, all the particles are written to the file. Note that for each particle only vectors $\vec{p}$, $\overrightarrow{hv}$, $\vec{v}$ are written to the file. The order in which particles shall be written is ascending order of identifiers. That is, the same order in which the where read.

IMPORTANT: Although all internal computations shall be executed using double precision, final results shall be stored in single precision.

5 Task

5.1 Software Development

5.1.1 Program versions

This task consists of the development of different sequential versions of the described application using C++20. Please, note that no use of multi-threading or parallelism is allowed.

To develop the required program teams may consider different alternatives as using structures of arrays, arrays of structures or hybrid alternatives combining both approaches.

5.1.2 Components

The following components shall be developed:

- **sim**: Library with all components used from main program.
- **utest**: Unit tests of all components in library **sim**.
- **ftest**: Functional tests of the application.
- **fluid**: Executable program with the application.

Below, some components are described in more detail:

sim : Simulation library components.
It will contain, at least, the following source files:

- **progargs.hpp, progargs.cpp**: Argument handling for **main()** parameters.
- **grid.hpp, grid.cpp**: Grid representation.
- **block.hpp, block.cpp**: Block representation.

utest : Unit tests.
It will contain, at least, the following source files:

- **progargs_test.cpp**: Unit tests for **progargs**.
- **grid_test.cpp**: Unit tests for **grid**.
- **block_test.cpp**: Unit tests for **block**.

fluid : Program with the application.
It will contain a single source file:

- **fluid.cpp**: It shall only contain a **main()** function for the application. It shall not include any additional function.

5.1.3 Compiling the project

All source files must compile without problems and shall not emit any compiler warning. A CMake configuration file shall be included with the following options:

Main CMakeLists.txt

```
cmake_minimum_required(VERSION 3.26)
project(fluid LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

# Set compiler options
add_compile_options(-Wall -Wextra -Werror -pedantic -pedantic-errors)
set(CMAKE_CXX_FLAGS_RELEASE "${CMAKE_CXX_FLAGS_RELEASE} -march=native")

# Enable GoogleTest Library
include(FetchContent)
FetchContent_Declare(
        googletest
        GIT_REPOSITORY https://github.com/google/googletest.git
        GIT_TAG v1.14.0
)
set(gtest_force_shared_crt ON CACHE BOOL "" FORCE)
FetchContent_MakeAvailable(googletest)

# Enable GSL Library
FetchContent_Declare(GSL
        GIT_REPOSITORY "https://github.com/microsoft/GSL"
        GIT_TAG v4.0.0
        GIT_SHALLOW ON
)
FetchContent_MakeAvailable(GSL)
```

```
# Run clang-tidy on the whole source tree
# Note this will slow down compilation.
# You may temporarily disable but do not forget to enable again.
set(CMAKE_CXX_CLANG_TIDY clang-tidy -header-filter=.*)

# All includes relative to source tree root.
include_directories(PUBLIC .)

# Process cmake from sim and fluid directories
add_subdirectory(sim)
add_subdirectory(fluid)

# Unit tests and functional tests
enable_testing()
add_subdirectory(utest)
add_subdirectory(ftest)
```

The CMake configuration file for directory **sim** shall include the following options:

CMakeLists.txt for sim library

```
# Add to this list all files related to sim library
add_library(sim
        progargs.hpp
        progargs.cpp
        grid.cpp
        grid.hpp
        block.cpp
        block.hpp
)

# Use this line only if you have dependencies from stim to GSL
target_link_libraries(sim PRIVATE Microsoft.GSL::GSL)
```

The CMake configuration file for directory **fluid** shall include the following options:

CMakeLists.txt for fluid program

```
add_executable(fluid fluid.cpp)
target_link_libraries(fluid sim)
```

These rules are an example.

Keep in mind that all evaluations shall be performed with the compiler optimizations enabled with CMake option **-DCMAKE_BUILD_TYPE=Release**.

IMPORTANT: The only allowed library is the C++ standard library. No external library is allowed.

EXCEPTION: The C++ Core Guidelines Support Library (commonly referred as the **GSL**) is also allowed. The latest version can be obtained at: `https://github.com/microsoft/GSL`.

5.1.4 Code quality rules

Source code must be well structured and organized, as well as appropriately documented. The **C++ Core Guidelines** (`http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines`) are recommended but not fully required.

A set of rules will be published in a separate document.

5.1.5 Unit tests

A set of unit tests shall be defined and included in the submission. You are recommended, but not required, to make use of GoogleTest (`https://github.com/google/googletest`).

In any case, evidence of enough unit tests shall be included in the submission.

In the case you use GoogleTest as a unit testing framework, below you will find an example CMake file:

CMakeLists.txt for unit tests

```
# Executable for all unit tests with list of sources
# For example, you may have one *_test.cpp for each *.cpp in sim
add_executable(utest
        particle_info_test.cpp
        math_vector_test.cpp
        block_test.cpp)

# Library dependencies
target_link_libraries(utest
        PRIVATE
        sim
        GTest::gtest_main
        Microsoft.GSL::GSL)

# Discover all tests and add them to the test driver
include(GoogleTest)
gtest_discover_tests(utest)
```

5.1.6 Use of AI tools

The use of AI powered tools during the project is allowed. However, please keep in mind the following:

- If you use any AI tool, please declare such uses in the design section of your project report. No negative grading will be given to such uses, provided that they are declared in the project report.
- No support will be given to the use of such tools. In particular, if you have questions about code, you need to be able to explain your own code.
- Beware that some AI tools may generate unsafe code or code with poor performance.
- You may be required by any of your instructors to explain your own code.

5.2 Performance and energy evaluation

This task consists of a performance evaluation of the application. To conduct the performance evaluation, the total execution time shall be measured. In addition, energy use must be also measured. The power use must be derived.

All performance evaluations shall be performed in a node from the **avignon** cluster.

Represent in a graphic all total execution times, energy use and power for images with different number of iterations.

The project report shall include conclusions drawn from results. Please, do not limit to simply describing data. Try to find convincing explanations of those results.

6 Grading

Final grades for this project is obtained in the following way:

- Performance: 15%.

- Energy use: 15%.
- Unit tests: 10%.
- Functional tests: 5%.
- Design quality: 5%.
- Code quality: 10%.
- Performance and energy evaluation in the report: 15%.
- Contributions from each team member: 20%.
- Conclusions: 5%.

RELEVANT WARNINGS:

- If the submitted code does not compile, the final grade for the project will be 0.
- If a quality coding rule is ignored without justification, the final grade for the project will be 0.
- In case of copying all implied groups will get a grade of 0. Beside the head of the school will be notified for the corresponding disciplinary actions.

7 Submission procedures

Project submission will be performed through Aula Global. Two submission links will be enabled:

- **Report**. It will contain the project report, which will be a file in PDF format named **report.pdf**.
- **Source code**: It will contain all the source code that is needed to compile.
  - It shall be a compressed file (ZIP format) named **fluid.zip**.

The project report shall not exceed 15 pages with a font size of 10 points or higher, including title-page and all sections. If there are more than 15 pages, page 16 and above will not be considered during grading. The report shall contain, at least, the following sections:

- **Title-page**: It shall contain the following information:
  - Project name.
  - Reduced group where students are enrolled.
  - Team number.
  - Name and NIA of all authors.
- **Original design**. It shall include the design for each component. In this section and explanation and justification for the main design decisions must be included.
- **Optimization**. It shall contain a discussion of applied optimizations and their impact. In particular, it shall specify optimizations performed on the original source code, as well as optimizations enabled with additional compiler flags.

- **Performed tests**: Testing plan description to ensure correct execution. It shall include both unit tests and functional tests.
- **Performance and energy evaluation**: It shall include performance and energy evaluations that have been carried out.
- **Work organization**: It must describe the work organization among team members, making explicit the tasks that each person in the team has performed.
  - It must contain a breakdown of the project into tasks.
  - Tasks must be small enough so that a task is assigned to a single person.
  - All team members must perform relevant contributions to design and implementation of software components.
  - A task shall not be assigned to more than one person. In such a case the task must be divided into sub-tasks.
  - The report must include the time dedicated by each person to each task.
- **Conclusions**. Special value will be given to conclusions derived from performance evaluation results, as well as those conclusions relating the performed work with contents of the course.