PSE Molecular Dynamics: Sheet 3 Report

Group C: Flavius Schmidt, Luca-Dumitru Drindea, Mara Godeanu

December 3, 2024

Code

- **Pull Request**: `https://github.com/FlamingLeo/MolSim/pull/4`
- **Technical Details**: This is a short description of our main development environments.
  - **Operating System**: Ubuntu 24.04 (also tested on Ubuntu 22.04 via WSL)
  - **Compiler**: Clang++ 18.1.3 (also tested with Clang++ 14)
  - **Compiler Flags (Release)**: `-Wall -Wextra -Wpedantic -Werror (-O3 -DNDEBUG)`
  - **C++ Standard**: C++17 (might be changed in the future)
  - **External Libraries**: `libxerces-c-dev libgtest-dev libbenchmark-dev libspdlog-dev libxsd-dev`
  - **Build Tools**: CMake 3.28.3, GNU Make 4.3

Report

Task 1: ”XML input”

- **General Idea**: Implemented XML input using CodeSynthesis XSD, a tool for automatically generating C++ code (classes) from a given XML schema (XSD). Not much else to say here; a description of the XML syntax and features is provided in `README.md`.
- **Format**: The generated files have the `.cpp` and `.h` extensions and use `UpperCamelCase` for type names and `lowerCamelCase` for function names to comply with our current style. `Generating the Files`: We created a script, `scripts/createXSD.sh`, to automatically generate the C++ files and format them. Interestingly, because CodeSynthesis XSD recently updated to version 4.2.0 (after 9 years since the last update) and the version in the repository still uses an older version, the version mismatch error may be triggered. As such, the script simply removes the check from the generated header file, as the code works regardless.
- **XMLReader**: We created a new type of file reader, `XMLReader`, which uses the generated `SimulationXSD.cpp` backend to read and parse XML files into our simulation arguments and particle container classes. The tests primarily check if the XSD is parsed correctly, especially where optional tags may or may not be present, and check trying to parse invalid XML files (e.g. missing, invalid or incomplete tags).
- **Documentation**: Adding documentation to be generated via Doxygen is done using `<xs:annotation>` and `<xs:documentation>` in the `.xsd` file and by passing `--generate-doxygen` to the tool. To be honest, we were kind of against adding documentation in such a manner. It clutters the XSD file and makes it much harder to read, it’s more tiresome to add compared to a simple C++ comment, it blends in with the rest of the XML code compared to comments being clearly distinguished from C++ code in most compilers, and finally, it’s mixed in with already generated default documentation. In the end, we decided to add it to comply with the worksheet tasks,
- **Miscellaneous**: The other options (`--std`, `--generate-serialization`) are not really all that interesting. The former is used to determine the type of smart pointer to use, the latter is used to enable marshalling and unmarshalling between C++ and XML data.

Task 2: ”Linked-Cell Algorithm”

- **General Idea**: In order to implement the linked cell data structure we created two new classes: `Cell` and `CellContainer`. `Cell` represents individual cells, in which lists with pointers to particles are stored, showing which particles are in that “region” of space. `CellContainer` is the main structure and exists as a single object. It actually stores the particles and manages “inter-cell” interactions.
- **Cell**: This is the simplest of the two classes. It stores a list of pointers `m_particles` to the particles that are inside it, along with the dimensions of the cell, its position and its type (Inner, Border, Halo). Halo cells only extend one past the actual domain. `Cell` offers the functions `addParticle()` and `removeParticle()` for managing the list of stored particle pointers.
- **CellContainer**: This class acts as a “manager” for the individual cells. The first difficulty arose in deciding for a 2D model (to be expanded later) or a fully 3D model. We chose the latter and later scaled back due to implementation issues. The first step was to handle the initialization. When dividing the domain into cells, we thought the following model makes the most sense:
  - All cells are of equal size (no special cases to be handled)
  - If possible, the cell size is the cutoff radius (as to avoid unnecessary searches in too far-away cells). If, however, the cutoff radius does not perfectly divide the domain, we make the cells slightly larger than the cutoff radius so as to always restrict the search to the neighboring cells (again, no special cases).
- **Initialization**: Once the number of cells is determined, they are created with the given size, position, type (Inner, Border, Halo) and added to a vector of cells `cells`. `haloLocation` determines the cardinal direction(s) of the cell if it is a halo cell (north, south, east, west, above, below; a cell may have multiple, such as north and west representing northwest). Finally, after all cells are created, `addParticle()` is called on all particles, and their references are added to the appropriate cells (based on their position). It is worth mentioning that the `Particle` class has been expanded with the attributes `cellIndex` (index of the particle's cell in the `cells` vector) and `active` (denotes whether the cell is still active within the simulation or not). This concludes the initialization.
- **Indexing**: In order to support much of the class functionality, it was necessary to create functions that allow to convert between 3D/2D positions of cells and particles in the simulation and the 1D positions (index) of cells in the `cells` vector. For this purpose, `getCellIndex(position)` returns the index of the cell in which the given coordinates lie. By this function we can always identify the cell in which a particle lies. The function `getVirtualCellCoordinates(index)` returns based on the cell index the 2D/3D coordinates of the cell. With these “conversions” assured, we first implemented the `getNeighbours(cellIndex)` function. This returns all the neighbouring cells of the cell at `cellIndex`, meaning all the cells that need to be considered for the force calculations.
- **Moving Particles**: In order to allow particles to migrate from cell to cell, we created the `addParticle()` and `deleteParticle()` functions. The first adds the particle reference to the cell it is *factually* in and changes the `cellIndex` in the particle accordingly. The latter removes the particle reference from the cell it was stored in (based on the particle's `cellIndex` field) and sets `cellIndex` to -1. Used in succession (see `moveParticle`), the two functions allow the movement of particles between cells. It is important to note, the actual particle objects are never created or deleted after the initalization. Only their references get added to and deleted from different cell objects.
- **Deleting Particles**: With `deleteParticle()` we can also completely remove the particles that are in halo cells. In `removeHaloCells()` the particles that are still marked as active and are located in the halo, have their reference deleted from the cell by `deleteParticle()` and are then marked as inactive.
- **Deleting Sucks**: There were many issues with actually `deleting` particle objects, so we just mark them as inactive. The point is, we tried to use `std::vector::erase`, but this caused issues with not removing the correct element (rather, removing the *last* element?), especially *while* iterating over the particle container, so we bit the bullet amidst time constraints and implemented this as a workaround.
- **Iterators**: We implemented custom iterators for iterating over the particles of the border cells and halo cells respectively. Both of these rely on the `SpecialParticleIterator`, which has an outer iterator for iterating over vectors of `Particle*` forward lists, for which there is a separate inner iterator. If a cell has no particles, the outer iterator advances until it finds one with at least one particle. As for iterating over pairs of particles, we opted for the cop-out method and just use the `PairIterator` of the underlying `ParticleContainer`. In all seriousness, having a new `PairIterator` that only iterates over distinct particles within a cell neighborhood was on the bottom of our priority list, and we already expressed our dismay with using a `PairIterator` as a response in our previous pull request. To summarize: explicitly iterating, the ”traditional” way, is (in our

opinion) more explicit and readable (i.e. the user sees *exactly* which particles / cells are being iterated over) and less error-prone (since we're just using what's already given to us and we're not relying on our own implementation).

- **Putting It All Together**: Now that we've defined the new data structure for using the linked cell algorithm, we've incorporated it in a new Lennard-Jones potential simulation type, `LennardJonesLC`. Aside from using the linked cell data structure alongside the main particle container, we've created new position- and force-calculating functions.
  - The position calculation function is mostly the same as the non-linked-cell function, except it moves a particle to a new cell (if necessary), checks if a particle enters a halo cell and applies the corresponding boundary condition. This is further elaborated in Task 3.
  - The force calculation functions (with and without using Newton's third law) incorporate the cutoff radius and rely on the following algorithm:

    ```
    for all cells ic:
        for all active particles i in ic:
            for all cells kc in ic's neighborhood:
                for all active particles j in kc:
                    if (i != j):
                        if(dist(i,j) <= cutoff):
                            calc. and add F(i,j) to F(i)
    ```

    When applying Newton's third law, we only iterate over distinct pairs of particles by comparing the *memory addresses* of particles i and j against eachother ($\&i < \&j$). This is possible because the addresses of two distinct particles are *always unique* and *always remain the same* inside the vector, as long as we don't change the vector dimensions or explicitly move them. An alternative would be to assign a unique ID to each particle.

  Note that we decided not to implement tests for the simulation as a whole this time; rather, we test each individual component (especially the physics functions and the boundary conditions), which should mostly guarantee functional execution of the simulation as a whole.

- **Benchmarking**: Once again, benchmarking has been performed using Google's *Benchmark* library.
  - **Environment**: Asus ZenBook 13, Ubuntu 24.04 LTS, Linux kernel version 6.8, 16 GB RAM, AMD Ryzen 7 5800U (8 cores, 16 threads) @ 4.51 GHz, Clang++ 18.1.3 with -O3 (Release build).
  - **Method**: The code used to benchmark the simulation is included in the `bench` directory. We ran the benchmark 10 times, then took the average value for each implementation - this is done because only measuring one iteration is inconsistent, since the CPU may be doing other things. All other irrelevant processes were closed while benchmarking. For our input, we generated 2D squares of differing sizes and let the simulation run for a few iterations. Figures 1, 2, 3 show the benchmark results when comparing different implementations.
  - **Cutoff**: As an aside, we also implemented functions for calculating the Lennard-Jones potential *without* the linked cell method, but *with* a specified cutoff radius, to see the improvement the radius alone would bring. While it is noticeable in the beginning, due to it still being $O(N^2)$, the benefit gradually disappears.

- **CellUtils**: For the sake of completeness, certain cell metadata types (the cell type, cardinal direction and boundary condition) are stored in `CellUtils`, a file / namespace which additionally provides utility functions for working with these new enums (primarily converting to and from strings).

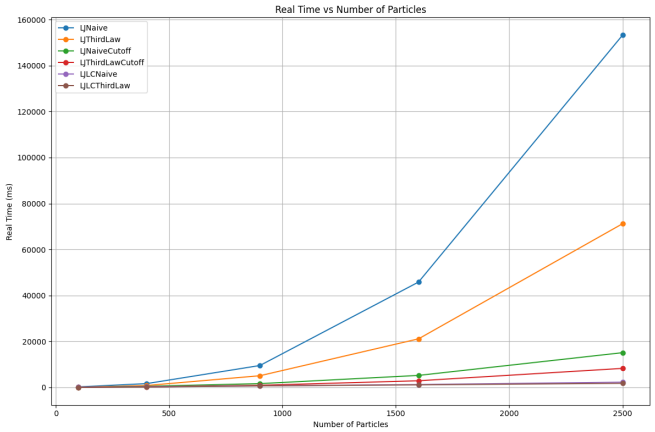


Figure 1: The benchmark results, comparing *all* implementations of the Lennard-Jones simulation. The x axis shows the total number of particles used in the benchmark, the y axis shows the time it took in ms. The naive implementation is by far the slowest, followed by the version using Newton's third law. The variants which only use the cutoff radius but *not* the linked cell method are noticeably better and quite similar to the linked cell methods, albeit not for long. *The graph goes up to 2500 particles here.*

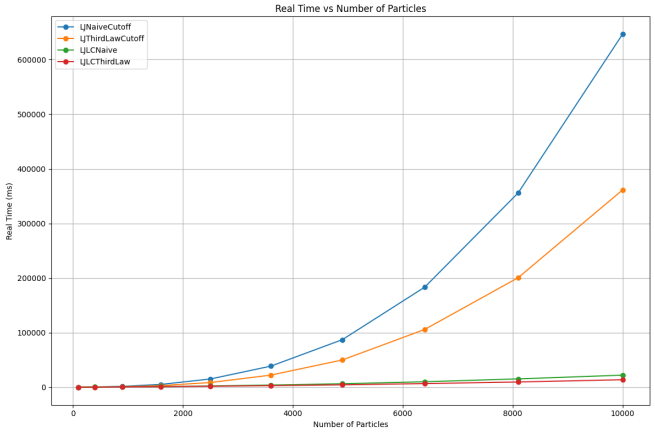


Figure 2: The benchmark results, comparing the implementations of the Lennard-Jones simulation that use the cutoff radius. The non-linked-cell methods start strong, but eventually, they become dramatically slower. *The graph goes up to 10000 particles here.*

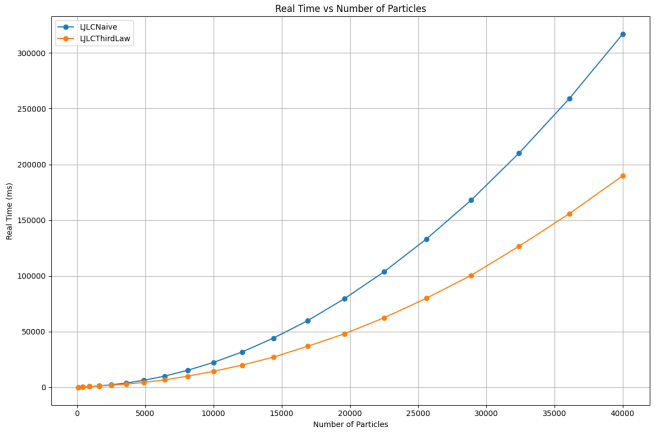


Figure 3: The benchmark results, comparing the implementations of the Lennard-Jones simulation that use the linked cell method. We are realistically able to use *many* more particles. *The graph goes up to 40000 particles here.*

Task 3: "Boundary conditions"

- **General Idea**: As mentioned previously, boundary conditions are applied once a particle enters a halo cell. Outflow boundaries are relatively naive; it's the reflective boundaries that involved a deeper thought process (and require some pictures for easier explanations).
- **Functionality**: Functionality for implementing boundary condition handling is located in the `strategies` directory, in `BoundaryConditions.cpp`.
- **Outflow**: Outflow simply removes particles (i.e. marks them inactive and invalidates their cell index) once they enter a halo cell.
- **Reflective**: Reflective boundaries simulate collision by mirroring the particle's position from inside the halo cell back in bounds and inverting the velocity in the corresponding direction.
  - **Opposite Neighbors and Position Mirroring**: In order to understand how reflective boundaries work, we first need to take a look at two more `CellContainer` functions: `getOppositeNeighbor()` and `getMirrorPosition()`.
    - **Opposite Neighbors**: The first step in implementing reflective boundaries is getting the opposite neighbor of a cell in a certain direction (rather, getting the neighbor of a cell in the opposing direction). It will soon become apparent why we need the *opposing* direction. Figure 4 shows this using example cells and indices.

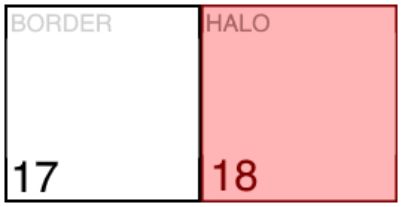


Figure 4: Two neighboring cells. Cell 18 is to the **east** of cell 17; therefore, we need to call `getOppositeNeighbor(18, east)` to get index 17. Similarly, cell 17 is to the **west** of cell 18. Thus, we need to call `getOppositeNeighbor(17, west)` to get index 18.

    - **Mirroring**: Mirroring from one cell into a neighboring cell is performed by first calculating the offset in a given direction, i.e. the distance of the particle to the border we want it mirrored against. This is calculated by subtracting the cell size with the particle's relative position inside the cell. Then, in the neighboring cell, the relative position of the mirrored particle is simply the previously calculated offset for our desired dimension. For all other dimensions, the relative position stays the same (i.e. if we're interested in mirroring the particle horizontally, the vertical position intuitively stays the same). This is explained using an example in figure 5.

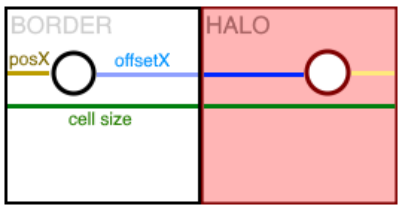


Figure 5: Suppose we wish to mirror the particle from the left cell into the right cell. We calculate the horizontal offset by subtracting the particle's relative position from the cell size. The mirrored particle's relative position inside the opposing cell in the original particle's horizontal offset. We proceed analogously for vertical mirroring.

  - **Reflective Boundary Basics**: With these two components, we're ready to implement reflective boundaries. Once a particle enters a reflective halo cell, we first need to find the opposing cell back inside the simulation domain. This is why we need to get the neighbor in the *opposing* direction; if, for example, we're entering an eastern halo cell, we want to find the neighbor opposing the east, i.e. the western neighbor. After we found the neighboring cell, to simulate reflection, we mirror the particle's position from inside the halo cell to the neighboring cell. Figure 6 helps visualize this. Because we can calculate the opposing neighbor cell in *multiple* directions (e.g. opposite of northwest is southeast), this also works with corner halo cells, as seen in Figure 7.

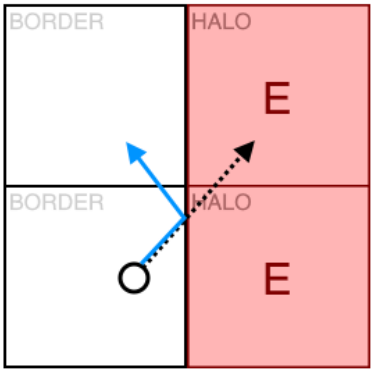


Figure 6: The basic reflection algorithm. The black arrow marks the ”true” trajectory of the particle as it enters the halo cell. The blue arrow marks the resulting position after calling the position calculation function. First, since we enter the halo cell to the **east**, we need to find the neighbor of the halo cell **opposite the east** direction (so, the western neighbor cell). Once we have the index of the neighboring cell, we mirror the particle into it.

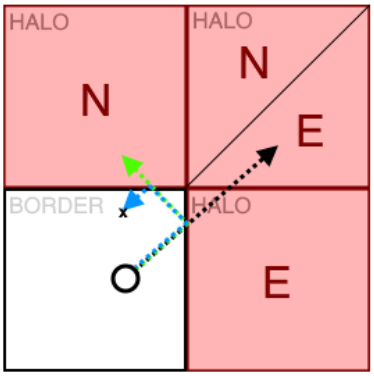
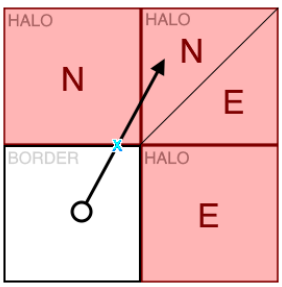


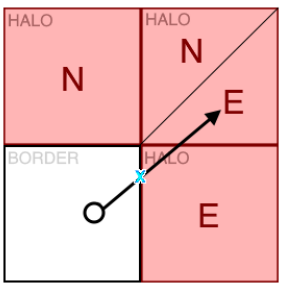
Figure 7: Reflecting also works for corner halo cells. The opposite cell of the **northeast** halo cell is located **southwest** of it. The mirroring occurs similarly, except we mirror both horizontally and vertically now. The black arrow marks the ”true” trajectory of the particle as it enters the halo cell. The green arrow is only for there for visualization purposes, to show that we need to mirror vertically aswell. The blue arrow marks the resulting position after calling the position calculation function.

- **Corner Halo Cells**: On the topic of corner halo cells... if we enter a corner halo cell, how do we determine which border the particle hits first? After all, it could be that one boundary has a different condition to another, so we need to apply the correct one. To do this, we split corner cells into two halves, and check to see which half the particle enters. Figure 8 shows what is meant by this.

  For three dimensions, this becomes a bit more complicated, both to implement and to visualize. As such, only the two-dimensional version is currently implemented.



(a) If we’re in the top-left half of the corner cell, it means we hit the northern border first, so we need to apply the northern boundary condition.



(b) If we’re in the bottom-right half of the corner cell, it means we hit the eastern border first, so we need to apply the eastern boundary condition.

Figure 8: Determining which boundary is hit first.

Task 4: ”Simulation of a falling drop - Wall”

- **General Idea**: We implemented a new type of particle cluster generator which generates 2D spheres (discs) of particles in a grid formation. We tried multiple different broken implementations until settling on one which (hopefully) works.
- **Attempt 1**: A first attempt was made by creating concentric rings of particles with distance h between the particles. This both went against the grid-based approach suggested by the problem statement and didn't work for bigger radii. Also, particles would overlap eachother.

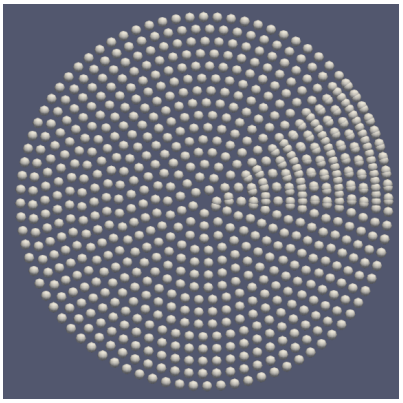


Figure 9: Our first attempt at making a particle disc. It's relatively accurate, but the molecules are not arranged in a grid, and there is noticeable particle overlap (top right).

- **Attempt 2**: We devised another algorithm based on the *Midpoint Circle Algorithm* which respects the grid approach and doesn't spawn particles over each other but instead nicely orders them evenly spaced between each other with respect to h. We first calculated how many particles we would need for the circumference of the disc based on the given radius r and then filled it with particles within the boundaries of said disc. We achieved this by treating the center particle as if it were the center of a square where one side is two times the value of the radius and by checking for each discrete point of the square if it is within the bounds of the circumference.

  However, this implementation was quite buggy. For starters, the borders were messy and incomplete. This probably had something to do with casting from `double` to `int`. Most jarringly, particles would behave erratically - particles would shoot across the domain, and the middle particle would seemingly disappear for no discernible reason (probably because particles were too close to eachother or overlapping?). This would also frequently lead to runtime errors while attempting to convert floating point numbers which could not fit in integer types.

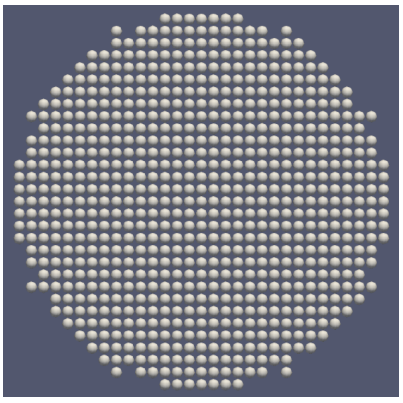


Figure 10: Our second attempt at making a particle disc. The particles are now arranged in a grid, but the border is quite inaccurate. Overall, this implementation was very buggy.

- **Attempt 3**: Finally, we went for a much simpler algorithm. Here, we simply iterate from the left-most x coordinate to the right-most y coordinate, moving by h units each iteration; we check if the position is within the circle area, and if so, we add a particle there. In the end, after running the simulation, we observe something which resembles a drop of water whose particles ”splash” (reflect) when touching the ground.

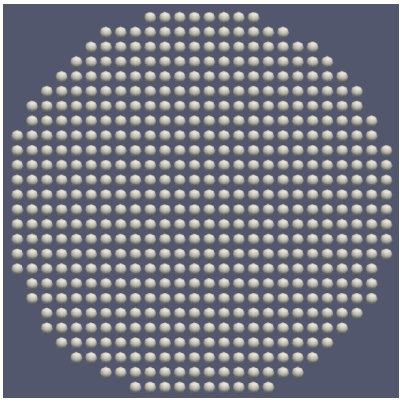


Figure 11: Our final attempt at making a particle disc. The particles are neatly arranged in a grid, and the borders look fine. Above all else, there are no more weird position and velocity issues.

Miscellaneous

- **Outlook**: There are a lot of features we meant to include for this worksheet, but couldn't, simply because many of us did not have time. This is not an excuse, it just so happened that all of us were occupied with other things, all at the same time. Possible future additions include, but are not limited to:
  - Adding an active particle iterator to prevent having to check the particle activity every time.
  - Adding the possibility to plot inactive particles (for visualization and debugging purposes).
  - Adding an extra optimization to the linked cell algoritm: instead of checking *every* neighboring cell, we only check those within the particle's cutoff radius, thus skipping possibly entire cells worth of particles.
  - Adding *full* support for 3D cell containers.
  - Adding an option to disable randomizing the velocity of cuboid and disc particles.
  - Further cleaning up and modularizing the codebase.
- **Minor Changes**: Minor changes were made since the last worksheet, mostly following suggestions left in the previous pull request.
  - We refactored the code in the `src/simulations` subdirectory to avoid needless repetition. Now, the initialization and time integration loop code are pre-defined in the `Simulation` superclass; the subclasses now only contain extra simulation-specific data structures (e.g. cell container for `LennardJonesLC`) and simulation-specific logs.
  - We renamed the files in `src/strategies` to make them more descriptive (e.g. `ForceCalculation.cpp` instead of `F.cpp`).
  - For testing file input using empty files, instead of keeping an empty file in the repository, we instead create a new temporary file and access that instead. For this, we use `tmpnam` to get the name of the function. However, this produces the following warning: `the use of 'tmpnam' is dangerous, better use 'mkstemp'`. For now, the code still works just fine, but if the warning becomes too annoying, we might change the functionality.
  - We briefly tried updating to C++20 to use `std::format` for easily creating formatted strings. However, this is still highly experimental, and the compiler used in the GitHub workflow doesn't even support this yet. So, we went back to C++17.