PSE Molecular Dynamics: Sheet 2 Report

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

November 14, 2024

Code

- **Pull Request**: `https://github.com/FlamingLeo/MolSim/pull/3`
- **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`
  - **Build Tools**: CMake 3.28.3, GNU Make 4.3

Report

Task 1: "Unit tests"

- **Usage** (inside `build/tests`): `ctest` (alternative, albeit with some unwanted output: `./tests`)
- **Basic Idea**: GoogleTest (`gtest`) is a popular C++ testing framework used, among other things, to write unit tests to individually test components of a program. We used `gtest` to write unit tests that try to encompass most logical cases in the execution of our own implemented features.
- **Integration**: `gtest` integrates seamlessly with CMake via its test driver program, `ctest`.
  - The integration happens automatically using the `enable_testing()` command. This allows the user to simply write `ctest` inside the test executable directory to run all tests.
  - There's no need to write our own `main()` function for running all tests, since we link with `gtest_main`, which automatically gathers all defined unit tests and executes them sequentially.
  - To avoid having to install `gtest` system-wide, our project uses CMake's `FetchContent` module alongside the `find_package()` command to detect if it is already installed and to fetch and install the latest version from its respective online repository without further input otherwise[1].
- `ASSERT` **vs.** `EXPECT`: The difference lies in how test case execution is handled.
  - `ASSERT` macros are used to indicate **fatal** execution errors when evaluating the specified condition and expressions inside the macros. In other words, if an **assertion** fails, the test case fails and is immediately stopped afterwards.
  - `EXPECT` macros are used to indicate **non-fatal** execution errors when evaluating the specified condition and expressions inside the macros. In other words, if an **expectation** fails, the test case fails, but it is not immediately stopped, allowing code that follows the macro to still be executed.

[1]Running the build script will still install the missing libraries system-wide on the users machine. We explain the reasoning below under **Miscellaneous**.

- As such, we decided to use `ASSERT` macros when continuing with the test case would crash the program or lead to unexpected results (e.g. assert that an array has size n before attempting to access elements at indices $0, ..., n-1$). In all other cases, we used `EXPECT`, which allows us to evaluate all macros in a test case, even if only one (or a few) failed.

- **Mocking**: A *mock object* is a part of a program frequently used when testing as a placeholder for real objects. It supplies the same interface as the real object, but with a "fake" implementation (i.e. you specify the return values or how often a function should be called, for instance). However, we decided against using mocks in our project (at least for now).
  - The mocked classes should contain *virtual* methods to be mocked. This would mean having to create new classes for everything we want to mock, or having to refactor the entire codebase / rewrite every class again just to accommodate mocking. Mocking non-virtual methods using templates is possible, but much more tedious.
  - In most cases, mocking wouldn't really work / be straightforward, since classes frequently use the functionality of other classes for their purposes (e.g. `CuboidGenerator` relies on a `FileWriter`, which itself relies on the functionalities of `Cuboid` and `ParticleContainer`, which itself relies on `Particle`).
  - Mocking wouldn't bring any huge performance benefits either, since most test cases only initialize very few objects, which shouldn't be too memory-intensive.

- **The Tests Themselves**: We wrote unit tests for every new component added this worksheet and for some previously implemented components aswell, including `ParticleContainer`. Each test has a corresponding explanation comment. We would have written more tests but ran out of time.
  - The tests are located in a separate directory, `tests`, which mimics the structure of the `src` directory. Test input files are located in `tests/files`.
  - Generally, the tests involve basic **expected functionality** (both programmatic, such as constructors, destructors, operator overloading; and semantic, such as calculating the force, position or velocity of a particle), **edge-case testing** (e.g. integer limits where applicable, reading empty files) and **invalid values**.
  - For testing the simulations, our approach was to test them with the two different force-calculation functions (naive, Newton's third law) against each other *and* against hardcoded, precomputed values. We do *not* run the full simulations, because otherwise it would take way too long to run the tests. We could, however, expand this in the future to be able to selectively and exclusively run the full simulations and progressively check against precomputed values.
  - Most tests use simple `TEST` cases which involve manually initializing each object. However, for some tests where the initialization of objects is identical throughout all tests, we used test fixtures (`TEST_F`) to avoid needlessly repeating code.
  - Because invalid input for most function calls results in program termination in our project, we use `EXPECT_DEATH` in these cases. While expensive, it's basically the only way to do so when checking erroneous values.
  - Testing file I/O was a bit tricky because of the filesystem structure. We could not use relative paths to access the input files, as that would assume the executable always remains in `build/tests` (absolute paths are impossible for obvious reasons). Trying to read a non-existent file would abort the tests. As such, the code first checks if it can find the input files by checking for the root "`MolSim`" directory, and only executes I/O tests if they are found relative to the project root.
  - In the GitHub `gtest` workflow, the code is compiled and ran using sanitizers. While slower, this additionally checks for any memory leaks or undefined behavior which occurs while running the tests. If a memory leak happens, the test automatically fails.

NOTE: There should be way more test cases for reading invalid files, but we didn't get around to completely fixing file input yet. For now, please use correctly formatted input files with valid numbers.

Task 2: ”Continuous Integration”

- **Basic Idea**: We already experimented with GitHub actions last week to automatically check our code’s formatting. Now, we’ve expanded it to automatically run tests, test the build script, attempt to build the documentation and to perform some basic linting.
  - `build-docs`: Builds the documentation following the steps explained last worksheet. This is done to ensure that the documentation comments are valid; otherwise, a warning is shown (while it doesn’t necessarily *fail* the test, you can still check the GitHub output to avoid having to recompile the documentation yourself).
  - `build-script`: Builds the project using the build script without any preinstalled libraries (first by letting CMake fetch the missing dependencies, then by installing them automatically using the system’s package manager). This ensures that building the project works on brand-new machines. This is **only** done on pull request to `master` because it is the most time-consuming and resource-intensive.
  - `format`: Checks code formatting by running `clang-format` on all input files, then seeing if there are any uncommited changes. This was already introduced last week.
  - `lint`: Does simple linting using `cpplint` with special arguments. This only checks a few minor things, such as missing `include` statements, redundant semicolons etc.
  - `test`: Builds the code with sanitizers enabled and runs tests.
- **Upholding CI**: To disallow circumventing the above checks and to ensure that all tests pass when merging to `master`, we checked the ”Require status checks to pass before merging” (with all actions) and ”Do not allow bypassing the above settings” options in the `master` branch protection rule. We also checked the ”Require approvals” and ”Require conversation resolution before merging” options to enforce addressing supervisor suggestions. Directly pushing to `master` is also disabled with ”Require a pull request before merging”.

Task 3: ”Logging”

- **Basic Idea**: `spdlog` is a C++ library used for fast, extendable, configurable and customizable logging. We implemented `spdlog` with various different log levels throughout our program.
- **Integration**: Integration is done in the same way as with `gtest`, using CMake’s `find_package()` and `FetchContent`. See *Task 1* for how it works.
- **Macros vs. Functions**: The main difference between macros and functions is that macros are defined at *compile time*, and turning them off via different logging levels effectively removes them completely from the code (by setting them to`(void)0;`). For this performance gain, we decided to use macros - it’s very slight, but it adds up over time, and guarantees that the code does not exist in the final executable, regardless of compiler optimizations. This also makes benchmarking a lot more accurate, since we’re *only* benchmarking the algorithm-specific code. The only potential drawback is the fact that the user must set the log level at compile time, and changing it requires recompiling the whole program. Another noticeable difference is the more succinct, albeit less customizable syntax, which is enough for us momentarily since we only log to `stdout`, and if *really* need the logs for something, we can pipe them from `stdout` to a file separately using other tools.
- **Structure**: We’ve used almost every provided log level, each with a different purpose.
  - Using the build script, by default (when not specifying a `spdlog` level), compiling a debug build will set `SPDLOG_ACTIVE_LEVEL` to 1 (`SPDLOG_DEBUG` or higher). Compiling a release build will set the active level to 2 (`SPDLOG_INFO` or higher).
  - Another small thing we’ve done is change the `spdlog` format depending on whether or not you use a debug or a release build. On debug builds, the *date, time, log level, file name and line* will be printed alongside the log message. On release builds, only the *log level* will be printed alongside the log message.
  - **Trace**: `SPDLOG_TRACE` is used for *very* granular, low-level operations in order to trace operations which may have a high performance impact, such as constructors, destructors or certain function entries. Because these macros could potentially be called very often - such

as when initializing a simulation with hundreds of particles - the messages will easily clog up the console output. As such, they are turned off by default and must be explicitly turned on by selecting the corresponding log level when compiling the program.

  - **Debug**: `SPDLOG_DEBUG` is used for slightly higher-level, more abstract operations that don't have that much to do with the nitty-gritty of C++ internals, but are still mostly irrelevant for the average user (e.g. which CLI arguments have been parsed, which line is has currently been read from a file etc.). These are operations you'd still want to track in a debug build in order to check if the main functionality works as intended, but you (momentarily) don't care about performance impact.
  - **Info**: `SPDLOG_INFO` is currently only being used to show the user which simulation is performed (and using which arguments) and when a file has successfully been written. These are the operations we consider important enough for the user to see, since visual feedback about what the program is doing is always a good thing to have.
  - **Error**: `SPDLOG_ERROR` is only used in the `error()` method in `CLIUtils.h`, which is called whenever the program encounters an error that is most likely caused by the user (e.g. invalid file input, invalid command line arguments).
  - Currently, we don't use `SPDLOG_WARN` or `SPDLOG_CRIT`, since we haven't found a good reason for either yet. However, we might implement warnings in the future for simulations which - while still possible - make little to no sense (e.g. simulations where the start and end time are the same). As for critical errors, since we halt program execution after most (if not all) errors anyway, there's little to no reason to distinguish between non-critical and critical errors.

Task 4: "Collision of two bodies"

- **Usage** (inside `build/src`): `./MolSim ../../input/input-lj.txt`
  The default simulation has been changed from Verlet integration to the Lennard-Jones potential, since most future simulations will use this according to the worksheet.

- **Structure**: In order to adhere to the structure created for the whole project, we decided to create a new class for the Lennard-Jones simulation (`LennardJones`). To implement the actual Cuboid functionality, two more classes were created: `CuboidGenerator` to initialize all Cuboids according to the input data, and `Cuboid` to handle the individual Cuboids.

- **Cuboid**: `Cuboid` holds all the meta-data pertaining to a Cuboid (position, start velocity etc.) and importantly a reference `particles` to a central `ParticleContainer` object, in which all particles of the simulation are stored. We decided to store all particles centrally as to make iterating over them easier, when calculating the forces. Naturally this is also more computationally efficient from the perspective of memory access, since all particles are stored contiguously. Based on the given meta-data, the function `initializeParticles()` initializes all particles and adds them to `particles`. Since the worksheet was not specific, we decided to individually calculate the velocity for each particle by summing the input velocity with the output of the provided `maxwellBoltzmannDistributedVelocity()` function.

- **CuboidGenerator**: The `CuboidGenerator` class serves as an interface between the main Lennard-Jones class and the individual Cuboids. It only has two attributes: the name of a file `m_filename` and a reference `m_particles` to the central `ParticleContainer`. Its only function (specifically, `generateCuboids()`) creates a `FileReader` and calls its `readCuboids()` function. We decided to implement file input because we considered too many variables are involved for comfortable command line input. The function reads the input file, creates the specified Cuboids and calls `initializeParticles()` on every one. Thus with a single function call in the simulation class, the setup is completed and all particles are to be found in `particles`.

- **LennardJones**: The `LennardJones` class incorporates the data and functionality needed for running the simulation. It uses a `CuboidGenerator` instance to generate the particles from the individual cuboids and iterates over each particle from the the container in order to calculate their respective position, force and velocity. The implementation of this class is not much different from the `Verlet` class, the main difference being that the Lennard-Jones potential is used in the force calculation. We tested both a naive approach of calculating the forces and a more optimized one where the forces are not calculated twice for each pair of particles, utilizing Newton's third law. The final program uses the optimized version.

- **Benchmarking**: Benchmarking has been performed using Google's *Benchmark* library.
  - **Integration**: Integrating the library is done identically to the integration of `spdlog` and `gtest`. See Task 1 for further details. Note that, in order to compile a benchmark build, *logging* must be turned off and a *release* build must be compiled. This ensures that the tests are as accurate as possible. Also note that, when compiling a benchmark build, you cannot run the main executable - only tests and the benchmark executable (`bench/bench`). This is because all logging and file output is disabled, which effectively makes the program useless.
  - **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/worksheet` 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.
  - **A Small Optimization...?**: Prior to the final benchmark, we found an optimization idea when performing the full simulation. Before, in order to reinitialize the forces for each calculation, we traversed every particle in the container and set the forces at the beginning of the force calculation method.

```
void calculateF_LennardJonesThirdLaw(...) {
    // set previous f for each particle and reinitialize to zero
    for (auto &p : particles) {
        p.setOldF(p.getF());
        p.setFToZero();
    }
    ...
}
```

    This seemed pretty inefficient, since we have to traverse the entire particle container. We thought we could improve this by setting the force effective on each particle immediately after calculating the positions, since this was the logical order in the simulation, thus saving us a traversal.

```
 void calculateX(...) {
    for (auto &p : particles) {
        ...
        // set previous f for each particle and reinitialize to zero
        // for the upcoming force calculation
        p.setOldF(p.getF());
        p.setFToZero();
    }
}
```

    However, the results were a little disappointing.
  - **Results**: Figure 1 shows the results of the benchmark. The naive implementation took roughly 127.2 seconds, while the improved version using Newton's third law took around half as much time, at roughly 68.52 seconds, as is expected. The optimization mentioned above brought almost no difference at all in this case (in some isolated runs, it was even very slightly faster), which was probably a result of various optimizations performed under the hood, or because the time complexity of both functions is $O(n^2)$, where iterating over all particles first adds a negligible $O(n)$ step. Overall, the execution time feels really slow - partially to be expected, since we're working with way more particles than in the previous simulation for 25000 iterations, and because the force calculation formula is a lot more complicated and computationally expensive. There is still definite room for improvement, though (especially parallelization).

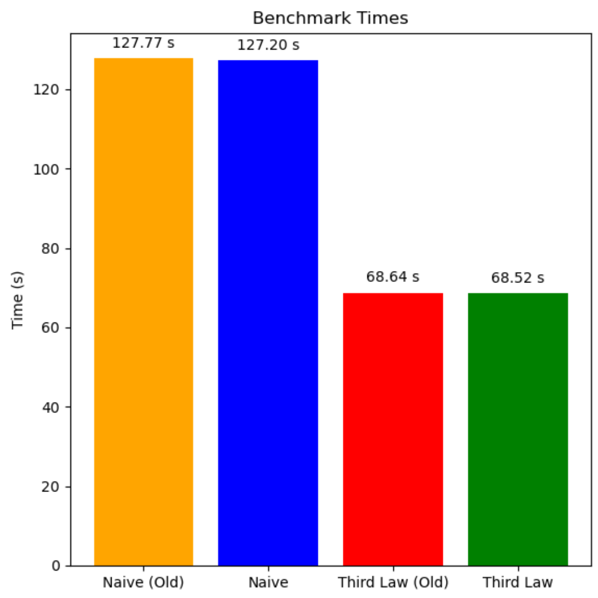
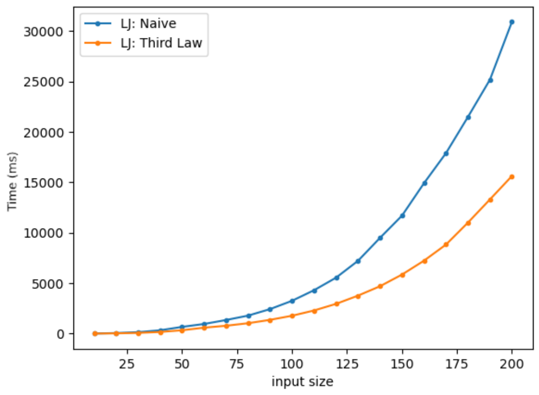


Figure 1: The benchmark results, comparing the naive implementation with the version that uses Newton's third law. The results marked with "(old)" do not have the force calculation optimization mentioned above included, the ones without do. There's basically no difference.

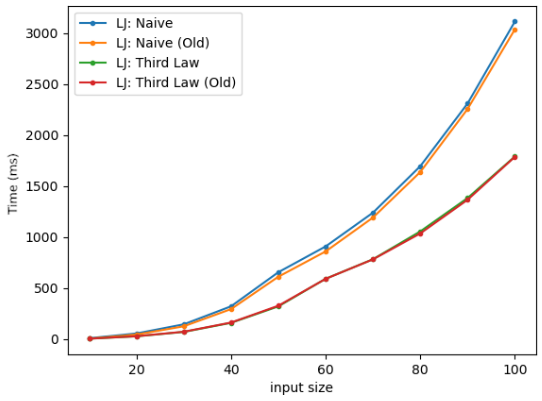
  - **Profiling**: We've also done a bit of profiling with `perf` to confirm the slowdown sources for our program. The force calculation has the largest performance impact by far at 38.21%.

Miscellaneous

- **2D vs. 3D**: We started out with a 3D simulation and then realized we should switch to 2D. A first (ugly) solution was to manually set the 3rd coordinate of the velocity to 0 after every new calculation. Looking more closely, we realized that the given `maxwellBoltzmannDistributedVelocity()` function can be called with only 2 dimensions.
- **Build Script Improvements**: The `build.sh` script has been completely overhauled.
  - **Command Line Arguments**: Because of the highly customizable nature of the compilation process, the build script now supports options passed via the command line. These either serve as shorthand for lengthier CMake arguments (e.g. `-b` instead of `-DCMAKE_BUILD_TYPE`) or change certain attributes of the build process (e.g. `-l` disables automatically installing missing libraries). A full list of options can be found in `README.md` or by using `-h`.
  - **Missing Libraries**: By default, running the build script on Debian-based systems will automatically install missing libraries system-wide using `apt-get` (except Google Benchmark, which installs a debug version through the package manager for some reason). This is done to speed up the compilation process and reduce the size of the `build` directory. Note that, in accordance with the first and third task statements, this can be turned off using `-l` to still let CMake fetch and install the dependencies locally.
  - **Parallel Jobs**: Compilation has been made parallel using Make's `-j` option with the number of available cores.
- **Benchmarking**: In addition to the required benchmarking performed in Task 4, we wrote some general benchmarks for testing the Lennard-Jones simulation for a varying, linearly increasing amount of particles. These are found in the `bench` directory. By default, benchmarking is turned off and must be manually turned on when compiling. For visualizing the data as seen in Figure Figure 2, we used this script.
- **More Shell Scripts**: Scripts to automatically format and lint the code have been added to the `scripts` directory. The build script remains in the root directory for ease of use and access.



(a) Benchmarking of the LJ simulation using the two different force calculation methods for 10 to 200 particles.



(b) Benchmark comparing the current method of resetting the force for each particle immediately after calculating the position with the old method. Once again, there is virtually no difference.

Figure 2: Results using the benchmark code provided in the repository on the machine specified above.

- **Various Code Improvements**: We've also made a few more QoL changes to the source code.
  - The default simulation arguments $(t_0, t_{end}, \Delta t)$ are chosen automatically depending on the simulation if not specified by the user, in accordance with the worksheet arguments.
  - Force, velocity and position calculations have been moved to their own files inside a new directory, `strategies`. A new `StrategyFactory` has been created to return the corresponding 3-tuple of functions depending on the simulation. This makes testing and benchmarking more straightforward and removes a lot of redundant code.
  - A new output writer, `NullWriter`, has been added. Instead of writing to a file, the result is printed to `stdout` using `SPDLOG_INFO` (or, if `SPDLOG_ACTIVE_LEVEL` is set to 6, nothing gets printed). This speeds up execution and makes benchmarking and debugging easier, at the cost of not being able to directly use Paraview for visualization.
  - When running a simulation again, if previous file output exists, the contents inside the corresponding output directory automatically get deleted. This prevents old data from still existing and guarantees a fresh batch of data for each execution.
- `libxerces-c-dev` has been retroactively configured to be automatically fetched by CMake if it is not available. As such, the user will hopefully not need *any* preinstalled system-wide dependencies to build and run the program.