PSE Molecular Dynamics: Sheet 1 Report

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

October 30, 2024

Code

- **Pull Request**: `https://github.com/FlamingLeo/MolSim/pull/2`
- **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**: `-Wall` (to be expanded in the future)
  - **C++ Standard**: C++17 (might be changed in the future)
  - **External Libraries**: `libxerces-c-dev` (to be expanded in the future)
  - **Build Tools**: CMake 3.28.3, GNU Make 4.3

Report

Task 1: "First steps"

Most of Task 1 was solved individually by each student since it mostly concerns setup.

- **.gitignore**: To the `.gitignore` file we added the folder `doxys_documentation`, which is created when generating the *Doxygen* documentation.
- **CMakeLists**: In the `CMakeLists.txt` file we changed the project name to "PSEMolDyn_GroupC".

Task 2: "First Pull Request"

- **Basic Idea**: There's not much to say here, really. We created 2 new branches - `Assignment1` (for the whole worksheet) and `Assignment1_TaskDummyPR` (for this exercise), wrote the `README.md` with build and run instructions and created a pull request from the latter branch to the former.

Task 3: "Completion of the program frame"

- **Implementaion**: As for our implementation of the presented algorithm, there was not much to do excpet for filling in the the methods "calculateF", "calculateX" and "calculateV" of the `Verlet.cpp` class according to the formulas for the force, position and velocity of a particle presented in the first meeting.
- **Command Line Parsing**: Aside from the required t_{end} and Δt parameters, we decided to allow *all* configurable options (incl. t_0, logging output frequency and output type) to be set via the command line. For parsing the arguments themselves, we decided to create a new `CLIParser` namespace and use the `getopt` function from the C standard library. We know it's not ideal, but it's the best option that doesn't involve writing the argument parser ourselves from scratch or including a massive library like *Boost* just for one small feature. `getopt` has a *lot* of downsides aside from being "non-C++"-esque: the `--` argument is handled weirdly, optional arguments don't *really* work, abbreviations are supported for whatever reason (e.g. `--hel` works the same as

`--help`, even though only `--help` is defined - this is why we *don't* use `getopt_long`), and the input file doesn't have to be the last argument for the program to work fine (e.g. `./MolSim -f1000 input.txt -o xyz` works, even though it shouldn't, because `input.txt` is not the last thing we pass it).

Task 4: "Simulation of Halley's Comet"

In this task we used the code from Task 3 to obtain and visualize our simulation data

- **Generating data**: Ran the code with default parameters ($\Delta t = 0.014,\ t_{end} = 1000$) and the given input file using the command `./MolSim ../input/eingabe-sonne.txt`.
- **Visualtization**: Data was visualized with the help of *ParaView* by following the instructions in the course slides. By using a Glyph filter we were able to visualize the movement of the particles and the change in parameters. The values for *mass* and *velocity magnitude* exhibited the expected behaviour.
- **Interpretation**: In order to assign real-world celestial objects to the visualized "particles" we first computed the orbital periods of the objects in the simulation (under the assumption that the static object is the sun and the one on the largest orbit Halley's Comet). By observing and writing down these periods in the virtual "simulation time" and calculating their equivalent in real time we came to the conclusion that the other two objects orbiting the sun must be the Earth and Jupiter. These results were later validated by the mass values in the `eingabe-sonne.txt` file, which indicate the same celestial objects.
- **Videos**: The video was exported from *ParaView* using the *Save Animation* function. We experimented with two formats, `avi` and `mp4`, and settled for `mp4` because of technical issues with `avi` (broken indexes). For the submitted video we chose to show the evolution of velocity magnitude, since it shows changes unlike the mass field and is easier to interpret visually than the force magnitude.

Task 5: "Refactoring and documentation"

- **Brief Idea**: This was arguably the most demanding task of the whole worksheet.
  - **Refactoring**: Separate each logical program aspect into separate folders for easier manageability, encapsulate molecules into new container class as per worksheet instructions, use software design patterns (e.g. factory methods) and OOP techniques (e.g. inheritance) wherever possible.
  - **Documentation**: Get used to Doxygen, customize the Doxyfile and extensively write documentation throughout the code.
- **Folder Structure**: Right off the bat, we knew that the least we could do, regardless of programming language or paradigm, is to categorize the different functionalities of the program and separate them into their own files and even directories respectively. We came up with a new folder structure inside of the `src` directory.
  - `io/{input,output,vtk}`: Functionality for processing file input / output and parsing command line arguments. All functions here terminate the program on error, because file I/O is vital here, otherwise there would be no valid result. We moved the external VTK library to its own directory to avoid clashing with our own code.
  - `objects/`: Physical objects used in our simulations. Currently only includes particles.
  - `simulations/`: The code for the simulations themselves. Currently only includes this week's simulation (Størmer–Verlet).
  - `utils/`: Utility functions for general use across our codebase.
- **Object-Oriented Programming**: Since we're working with C++, we tried to apply some basic principles of OOP and software design patterns to make our code more modular and easily expandable.

- `FileWriter` and `WriterFactory`: All classes which generate some sort of *file output* (currenlty VTK and XYZ output) may inherit common functionality from the `FileWriter` superclass. The user may then choose at runtime which type of output they desire by specifying it via the command line, which then gets passed to the `WriterFactory` method for creating a unique pointer to the desired output writer class. This is an easy and extensible way to support different methods of I/O, since we can simply create a new class which inherits from `FileWriter` whenever we need a new output type and include it in `WriterFactory`.
- `Simulation` and `SimulationFactory`: All *simulation* classes (incl. this worksheet's Verlet simulation, which we moved to its own class with the same name) inherit from the `Simulation` interface class which defines a common `runSimulation()` method. For now, functions for *calculating the position, velocity and effective force on the particle* are stored in the respective simulation class, but this may change in the future. Just like the output type, the user can choose the desired simulation at runtime using the command line, which gets passed to the `SimulationFactory` factory method. This is also done to easily expand the program once more simulations are introduced.
- `ParticleContainer`: This is a completely new class for encapsulating molecules.
  - **Data Structure**: Under the hood, the `ParticleContainer` class stores multiple `Particle` objects contiguously and dynamically in a `std::vector`. We did this because using vectors comes with several benefits compared to lists. We also considered the possibility of using a `std::array` since we know the amount of particles a file contains, but the only slight benefit would be stack-allocation vs. heap-allocation, which is probably not worth the hassle of restructuring the file input process again and also eliminates the possibility of ever working with a dynamic number of particles, should the need for that ever arise. One interesting caveat is that, due to the way a `std::vector` dynamically allocates memory, if you don't reserve a certain amount of spaces before adding objects, the program may generate expensive copies when adding a new object to the vector[1]. We fixed this by first reserving enough spaces for each particle (as specified in the first line of the input file) and then by simultaneously creating and adding a particle into the vector using `emplace_back(...)`.
  - **Iterators**: At first, we tried to implement our own iterator class, but we soon realized that the C++ standard library already provides readily available iterators for `std::vector` - all we needed to do was write the `begin()` and `end()` methods. Thus, one can iterate over a (`const`) `ParticleContainer` using a simple range-based for-loop. We did, however, have to write the `PairIterator` class and functionality ourselves. Essentially, it uses two `std::vector` iterators (inner, outer) to effectively simulate two nested for-loops by progressively incrementing the inner iterator until it reaches the end, which is when the inner iterator gets reset to the beginning of the `std::vector` and the outer iterator is incremented.
  - **Pointers and References**: We basically replaced every occurence where a compound type is passed as a function parameter with a `const` reference to prevent copying (except for `FileReader::readFile(...)`, where we pass a pointer to a `ParticleContainer` object to allow calling the method from inside a `ParticleContainer` object using `this`).

  For a complete class hierarchy and documentation of all class methods and attributes, consider the Doxygen documentation.

- **Doxygen**: Doxygen is a widely known and highly configurable automatic documentation generation tool which works by analyzing the source code for comments following a specific structure using specific "tags" (e.g. `@brief`, `@param`, `@return`...), from which it will generate an easy-to-use HTML documentation page. It's also useful for creating a hierarchical overview of all files, classes and namespaces, as mentioned previously.

  - **Integration**: After writing the necessary documentation comments in our program's header files (as is convention), we used the pre-existing `Doxyfile` to generate our documentation. The most straight-forward way of doing this is by simply calling `doxygen` in the root directory. Alternatively, we incorporated Doxygen in our *CMake* environment by writing a *CMake module* (`cmake/modules/doxygen.cmake`) which checks if Doxygen is installed on the host

[1] You can reproduce this by commenting out `particles->reserve(num_particles);` (`FileReader.cpp:63`).

machine, then automatically adds an *optional* `doc_doxygen` target to the generated `Makefile` (can be turned off with `-DENABLE_DOXYGEN=OFF`; automatically excluded from `make all` via the `EXCLUDE_FROM_ALL` property). The generated HTML documentation will use the repo's `README.md` file (`USE_MDFILE_AS_MAINPAGE = README.md`). LaTeX output is unsupported due to numerous compilation errors on our machines. As said before, the built documentation is *not* included in the repository to avoid needless bloat - you must build it yourself.

  - **Additional Changes**: We decided to only include the *header files* in the generated documentation, since the .cpp files merely provide the implementations, and because there is no .cpp file without a corresponding .h file. We also *updated and finished* the documentation for the pre-existing files to support Doxygen. *Bug tracking and reporting* is also possible by using the `@bug` annotation, which we may continue to use should bugs (inevitably) arise. Finally, we included the *mathematical formulas* via LaTeX (`USE_MATHJAX = YES`) for the documentation of the particles and the simulation because we wanted to keep everything in one place and avoid having to constantly open the slides / worksheet to cross-reference the formulas (and because we thought it looked cool).

  As mentioned above, the latest stable documentation is also available online for ease of access.

- **CMake**: Aside from the custom CMake module described previously, we left `CMakeLists.txt` mostly unchanged, since it already does everything needed for this assignment. This could, however, change in the future, as there are numerous ways to achieve similar goals - some more concise than others (e.g. using `add_compile_options(...)` to add compiler flags).

Miscellaneous

There are a few other additions we made not pertaining to the worksheet to improve the overall development experience.

- **Build Script**: When building the project from scratch, instead of having to manually type in each command to create the directory and build the project using CMake, we wrote a simple shell script that automates this process. We could update this in the future to enable further customization (i.e. disabling Doxygen).
- **GitHub Workflows**: We added an automatic GitHub action which, when creating a pull request to the `master` branch, automatically checks to see if the code is formatted properly using a custom `.clang-format` file in the root of the repository. If the code is not properly formatted, the check will fail. We did this to ensure a certain level of consistency without having to worry about manually formatting the code every single time, since we all use different IDEs with different formatting settings. At first, we wanted to make it so that the code is automatically formatted when pushing to `master`, but the code broke for some reason, so this quick fix will do.