Evaluating the Reusability and Maintainability in the Evolution of the FMSolvr Project

Technical Report

Jonas Henschel and Matthias Werner

17th Dezember 2025

FMSolvr is a C++-based software bundle to emply the parallel Fast Multipole Method (FMM) for the molecular dynamics. During the course of the project "FMhub – A platform for providing fast multipole solvers for science," the software, which was originally only part of the GROMEX simulator, was revised to achieve better reusability and maintainability. This report analyzes the changes achieved in the project both qualitatively and quantitatively.

1 Introduction

This report presents a comparative analysis of three historical versions of the fmsolvr framework with a focus on software maintainability and reusability. The analyzed versions represent different stages in the evolution of the software:

- Version A: an early research-stage implementation that represents the initial code.
- Version B: the last version before the introduction of the new build system (revision 194c9b240c)
- Version C: the most recent version (revision 541e48ebec)

The goal of this analysis is to evaluate how the software evolved from an early research prototype into a more structured and reusable software framework. Particular attention is given to repository structure, build infrastructure, modular decomposition, code complexity, integration of testing and benchmarking, and suitability for reuse in external projects. The analysis is based on static inspection of the code bases and a set of commonly used software engineering metrics.

2 Methods and Metrics Used in the Analysis

The comparison of the three code bases was carried out using a combination of static inspection of the repository structures and quantitative software metrics.

In order to obtain meaningful software metrics, two measurement perspectives were used. In the first perspective, all files contained in the repository were included. In the second perspective, the analysis focused only on what can be considered the core code, excluding files that primarily contain generated data rather than manually maintained program logic.

Certain large header files contain precomputed numerical tables or automatically generated coefficient arrays. Examples include files such as `helpers/qxyz_*.hpp` and headers located in the tables directory. Although these files are technically part of the repository, they behave more like static data resources. Including them would distort metrics such as lines of code or structural complexity because they contain extremely large blocks of numeric data that are rarely modified manually. For this reason, the analysis distinguishes between full repository statistics and core code statistics that exclude these files.

Several basic software metrics were used. The first group of metrics concerns the size of the code base, including the number of source files and the number of source lines of code (SLOC). SLOC counts were computed after removing blank lines and comments. These values provide an estimate of the size of the code base and allow comparison of the distribution of functionality across subsystems. A second group of metrics concerns the modular structure of the project. The directory hierarchy and the logical grouping of source files were examined in order to identify major subsystems. Modules such as `container`, `math`, `operator`, `simd`, `p2p`, `tree`, and `util` represent functional components of the solver. Examining the size of these modules makes it possible to identify architectural concentrations of responsibility and potential maintenance bottlenecks. The analysis also included an approximate estimation of cyclomatic complexity, also known as McCabe complexity (McCabe 1976). Cyclomatic complexity measures the number of independent execution paths through code and increases with the use of conditional statements, loops, and other control-flow constructs. Because a full C++ parser was not used in this environment, the complexity values were estimated using a heuristic pattern-based approach that counts common control-flow constructs in the source files. The resulting values should therefore be interpreted as approximate indicators rather than exact measurements. They are nevertheless useful for identifying relative differences between the three versions and locating files that concentrate a large amount of control-flow logic. In addition to code metrics, the analysis also evaluated the development infrastructure of the project. This includes the structure of the build system, the organization of source files within the repository, the separation between generated data and maintainable code, and the integration of testing and benchmarking infrastructure. These aspects strongly influence the maintainability and reusability of scientific software even when the core algorithms remain unchanged.

Version	Files	Core SLOC (approx.)
Version A	260	≈16,900
Version B	183	≈19,800
Version C	182	≈19,600

Table 1: Approximate structural metrics of the three versions

3 Code Base Size

A comparison of the basic structural metrics shows that the three versions differ in size but not dramatically. Version A contains approximately 260 source files and about 17,000 lines of core source code. Version B contains roughly 183 files and about 19,800 lines of core code. Version C contains approximately the same number of files as Version B and slightly fewer lines of core code, around 19,600 lines, cf. Table 1.

The increase in size between Version A and Version B reflects the expansion of the solver capabilities as well as the addition of performance measurement infrastructure. The nearly identical size of Versions B and C indicates that the most recent version primarily reorganizes the project rather than introducing major algorithmic changes.

4 Evolution of the Project Structure

The three versions illustrate a gradual maturation of the software project. Version A reflects an early research stage. The code base already implements substantial functionality, but the project structure is relatively flat and several components are not clearly separated. Solver logic, experimental utilities, and infrastructure code are sometimes located close to each other. The build infrastructure is minimal and tailored mainly for local experimentation rather than for reuse.

Version B represents a more mature research framework. The repository now contains clearer subsystem directories that separate numerical operators, utility code, and supporting components. At this stage, the project also introduces unit tests and microbenchmarks used for performance evaluation. These additions indicate that the solver was already being used in a more systematic research environment.

However, Version B still relies on a complex build system consisting of numerous specialized Makefiles for different compilers and hardware architectures. Although this system provides flexibility in high-performance computing environments, it increases maintenance overhead and complicates integration into external projects.

Version C preserves the solver architecture introduced in Version B but significantly improves the surrounding engineering infrastructure. The most notable change is the introduction of a CMake-based build system. This system centralizes build configuration and provides a standardized interface for compiling the solver, running tests, and executing benchmarks. As a result, the project becomes easier to build, maintain, and integrate into other software systems.

5 Modular Structure

The internal architecture of the solver is organized into several logical subsystems that correspond to different aspects of the fast multipole method implementation. Important modules include containers for data structures, mathematical utility functions, SIMD-optimized kernels, and operator implementations that perform the multipole transformations.

Across all three versions, the `operator` subsystem is by far the largest and most complex component of the framework. This module contains the numerical algorithms responsible for the core functionality of the solver. Other modules such as `simd`, `util`, `math`, and `container` provide supporting functionality and are considerably smaller.

The modular structure becomes clearer between Versions A and B as responsibilities are separated more consistently. Version C maintains essentially the same modular organization but improves the infrastructure that connects the modules during the build process.

6 Code Complexity

6.1 Approach

The structural complexity of the code base was assessed using an estimation of cyclomatic complexity, a widely used software metric introduced by McCabe (1976). Cyclomatic complexity measures the number of linearly independent execution paths through a program and is commonly used as an indicator of the structural complexity and maintainability of source code.

Within the context of this study, cyclomatic complexity serves as a proxy for the cognitive effort required to understand and modify code, which is an important aspect of software maintainability. This interpretation is consistent with the maintainability model defined in the ISO/IEC 25010 software quality standard, where code complexity is considered one of the key factors affecting analyzability, modifiability, and testability.

Cyclomatic complexity is defined as the number of independent paths through the control-flow graph of a program. In graph-theoretical terms, it can be expressed as

$$CC = E - N + 2P$$

where
E represents the number of edges in the control-flow graph,
N represents the number of nodes, and
P represents the number of connected components.
In practice, the metric is typically approximated by counting decision points in the source code. Each control-flow construct that introduces branching increases the number of independent execution paths and therefore increases the cyclomatic complexity. Typical constructs contributing to complexity include conditional branches, loops, and logical operators.

In the present analysis, the complexity estimation was performed using a static pattern-based analysis of the C and C++ source files contained in the three versions. The complexity estimation was performed in following steps.

Each source file was scanned for common control-flow constructs that contribute to cyclomatic complexity. These include

- conditional statements (`if`, `else if`)
- loops (`for`, `while`, `do`)
- switch-case constructs
- logical operators used in conditional expressions (`&&`, `||`)
- ternary operators (`?:`)
- exception handling constructs (`catch`)

Each occurrence of such constructs was counted as an additional decision point in the control-flow graph of the program. Third, the counts obtained for individual files were aggregated to produce approximate complexity values for: individual source files, modules, and the overall code base. The complexity estimate for a file therefore corresponds to $CC \approx 1 + N_{\text{decisions}}$ where $N_{\text{decisions}}$ represents the number of detected control-flow constructs.

The complexity analysis was implemented using a lightweight static analysis script operating directly on the extracted repository archives. The script scans source files and counts the relevant control-flow patterns using regular-expression matching. Because the analysis environment did not include a full compiler-based static analysis toolchain, no advanced parsing framework such as Clang tooling or dedicated complexity analysis tools (e.g., Lizard, PMD, or SonarQube) was used. Instead, the pattern-based approach provides an approximate but robust estimate that is sufficient for comparative analysis.

6.2 Results

The estimated cyclomatic complexity of the code base increases from Version A to Version B and remains largely unchanged between Versions B and C. Version A has an estimated complexity value of approximately 1,700, whereas Versions B and C both exceed 2,100. This increase reflects the expansion of the solver functionality and the introduction of additional numerical operators and performance optimizations. The stability of the complexity values between Versions B and C confirms that the newest version does not fundamentally change the algorithmic structure of the solver.

In all three versions, most of the structural complexity is concentrated in a relatively small number of files implementing numerical operators. Examples include the files `lstar_operator.hpp`, `multipole2local_optimized.hpp`, `rotation.hpp`, `ostar_operator.hpp`, and the solver driver `coulomb.cc`. These files implement mathematically sophisticated algorithms and combine template-based abstractions with low-level performance optimizations. As a result, they contain a large number of control-flow constructs and represent the most demanding components from a maintenance perspective. While such complexity is partly unavoidable in high-performance numerical software, these files should be considered critical areas for future refactoring and documentation.

7 Testing and Benchmarking Infrastructure

Beginning with Version B, the project includes a dedicated set of unit tests and performance microbenchmarks. The tests verify the correctness of important algorithmic components, while the microbenchmarks measure performance characteristics of numerical kernels. The benchmarking infrastructure includes experiments that evaluate SIMD performance, timing mechanisms, and hardware performance counters. Such tools are particularly valuable in high-performance computing environments because they help detect performance regressions and evaluate optimization strategies. Version C improves the integration of this infrastructure into the build system, making it easier to execute tests and benchmarks automatically.

8 Assessment

8.1 Reusability

The reusability of the framework improves significantly across the three versions. Version A is primarily designed as a research code base and does not provide clear mechanisms for integration into external software projects. Reusing the solver would require manual adaptation of the build environment and a detailed understanding of the repository structure.

Version B introduces clearer modular boundaries and supporting infrastructure but still relies on a complex set of platform-specific build scripts. This structure makes reuse possible but relatively cumbersome.

Version C substantially improves the situation by adopting a modern CMake-based build system. The solver can now be integrated into external CMake projects with minimal configuration effort, which greatly increases its practical usability as a reusable software component.

8.2 Maintainability

From a maintainability perspective, the project shows steady improvement over time. The introduction of a clearer modular structure, the addition of tests and benchmarks, and the modernization of the build system all contribute positively to the maintainability of the framework. Nevertheless, several challenges remain. The most significant maintenance risks arise from the large operator implementations that concentrate much of the solver's complexity. These files are difficult to understand and modify because they combine mathematical logic, template abstractions, and low-level optimizations. Another issue is the presence of large driver programs that combine multiple responsibilities, including solver initialization, execution control, and performance measurement. Separating these responsibilities into smaller components could improve readability and maintainability.

Aspect	Version A	Version B	Version C
Project structure	limited	improved	well organized
Build system	minimal	complex Makefiles	modern CMake
Testing	minimal	present	integrated
Benchmarking	minimal	extensive	integrated
Reusability	low	moderate	high
Algorithmic complexity	moderate	high	high

Table 2: Comparison of important SW aspects

9 Conclusion

When the three versions are considered together, the project clearly evolves from a research prototype toward a more structured and maintainable software framework. Version A represents the prototype phase in which the main focus is on algorithm development. Version B corresponds to a research framework stage where functionality and performance infrastructure are expanded. Version C represents an engineering-oriented stage where the focus shifts toward improved project infrastructure and long-term maintainability. This progression is typical for scientific software projects that gradually evolve from experimental implementations into reusable research tools. Table 2 shows a comparision of considered software-quality aspects of the considered versions. The comparison of Versions A, B, and C demonstrates a clear trajectory in the development of the fmsolvr framework. While the numerical algorithms themselves remain complex and largely unchanged in the most recent version, the surrounding software infrastructure has improved substantially. The introduction of a modern build system and a cleaner repository structure significantly enhances the maintainability and reusability of the software. These improvements provide a stronger foundation for future development and make the solver easier to integrate into external research projects. Further improvements in maintainability would likely require targeted refactoring of the largest operator implementations, which remain the primary concentration points of structural complexity within the code base.

References

[1] McCabe, T. J. (1976). *A Complexity Measure*. IEEE Transactions on Software Engineering. Vol. SE-2, no. 4, pp. 308-320, Dec. 1976.

[2] ISO/IEC 25010 (2011). *Systems and Software Engineering – Systems and Software Quality Requirements and Evaluation (SQuaRE) – System and Software Quality Models.*