Reflection and Traceability Report on MPIR

Xunzhou (Joe) Ye

1 Changes in Response to Feedback

All feedback Github issues were closed with a comment. It either provides a link to the commit addressing the issue or explains why the issue was rejected. Please refer to Closed Github issues for tracability.

1.1 SRS and Hazard Analysis

1.2 Design and Design Documentation

1.3 VnV Plan and Report

2 Challenge Level and Extras

2.1 Challenge Level

Copied from the problem statement:

> The challenge level of this project is expected to be general, leaning toward the advanced level for the following reasons:
>
> 1. This is a continuation of a Master's level research project. The associated thesis is still working in progress.
> 2. The choice of algorithm and mathematical technique is predefined as part of the project. The mathematical knowledge required for understanding and implementing these algorithms (e.g. GMRES) is graduate level.
> 3. Working with multiple floating point precisions requires knowledge on computer architectures, low-level floating point representations in computer, as well as advance implementation techniques to support manipulations of precisions.
> 4. Testing, benchmarking, and evaluating the performance gain on applying these advanced algorithms and techniques are also very involved.

2.2 Extras

Usability testing. Please refer to the VnVReport for a detailed report.

3 Design Iteration (LO11 (PrototypeIterate))

The final design and implementation of MPIR evolved significantly from the initial prototype, guided by feedback, usability insights, and practical constraints encountered during development. The initial version focused on quickly prototyping the core numerical algorithm—preconditioned GMRES with iterative refinement—without strong emphasis on modularity, extensibility, or type safety.

As development progressed, a key insight was the need to reduce the complexity of the interface. Early versions exposed many low-level numerical and type parameters, which were error-prone and hard to use correctly. In response, later iterations introduced the `Refinable` concept to encapsulate type constraints and reduce redundant template parameters. This abstraction simplified the solver API while maintaining flexibility across mixed-precision configurations.

Usability testing with internal team members highlighted pain points in documentation clarity and error messages. Based on this, improvements were made to ensure better self-documentation of function names, improved error reporting, and clearer usage examples. The solver interface was also made more idiomatic by replacing raw pointer manipulation with standard containers such as `std::vector`, greatly improving readability and safety.

Performance benchmarking also shaped the final implementation. Manual timing and profiling helped restructure certain steps (e.g., Gram-Schmidt and Givens rotations) into modular helpers, making them easier to optimize and validate independently.

While the project did not undergo multiple full development cycles due to time constraints, targeted refinements were made throughout the process based on stakeholder feedback and empirical testing.

4 Design Decisions (LO12)

The design of MPIR was shaped by a combination of research goals, technical constraints, and assumptions inherited from prior work.

Limitations Time and manpower limitations constrained the scope of the features implemented. For example, although support for multiple sparse matrix formats was considered, only Compressed Sparse Column (CSC) was supported in the final release. Similarly, maintainability testing and full documentation coverage were deprioritized in favor of ensuring correctness and performance.

Assumptions The project assumed that the input matrices would be symmetric and quasi-definite, a constraint imposed by the use of QDLDL as the preconditioner. It also assumed that users of the library are domain experts, which influenced decisions around error handling and user interface simplicity.

Constraints The software was designed to interface with a previously developed research prototype, which constrained design flexibility. Adhering to this compatibility requirement meant retaining certain data structures and algorithmic sequences, even when alternatives might have offered better abstraction or performance.

Overall, design decisions were made to balance innovation with alignment to the existing research direction, prioritizing numerical correctness, performance validation, and software quality within the defined constraints.

5 Economic Considerations (LO23)

MPIR is intended as a research-grade open-source numerical library rather than a commercial software product. As such, its primary value lies in academic and scientific use, particularly for researchers working with sparse linear systems and mixed-precision methods.

There is a niche but growing community in computational science, optimization, and numerical linear algebra that could benefit from such a solver—particularly in the context of accelerating computation with mixed-precision techniques. Potential users include graduate researchers, academic labs, and developers of scientific computing software.

To attract users, the focus would be on clear documentation, published benchmarks, and integration examples that demonstrate its advantages over existing libraries. Publishing results in a peer-reviewed venue and promoting through academic mailing lists and GitHub repositories would be essential for adoption.

If commercialized, the costs would include packaging, support, performance tuning, and robust cross-platform support. However, due to its specialized nature and open-source orientation, monetization is unlikely to be the focus. Instead, success would be measured by citations, usage in downstream research, and community contributions.

6 Reflection on Project Management (LO24)

6.1 How Does Your Project Management Compare to Your Development Plan

At the outset, there was no formal development plan. As a solo project, responsibilities such as implementation, documentation, and testing were all managed independently and informally. As the project matured, informal task lists and GitHub issues were introduced to organize priorities, which improved the workflow during later phases.

6.2 What Went Well?

The use of GitHub for version control, issue tracking, and continuous integration was particularly effective. Lightweight tools such as CMake, clang-format, and clang-tidy helped enforce code quality without introducing too much overhead. Despite the absence of a team, consistent incremental development and documentation practices enabled steady progress and maintainability.

6.3 What Went Wrong?

Without a structured development plan from the beginning, certain tasks—especially verification planning, usability testing, and code cleanup—were deferred until late in the timeline. This led to some last-minute work and limited the opportunity for multiple design iterations. The solo nature of the project also meant that there was limited opportunity for collaborative feedback or delegation of tasks, which might have improved efficiency.

6.4 What Would you Do Differently Next Time?

For future projects, even solo ones, creating a lightweight but structured development plan early on would help manage scope and track progress more effectively. This plan would include estimated timelines, checkpoints, and clearer criteria for success. Introducing formal retrospectives or progress reviews at regular intervals would also provide opportunities to reflect and adjust the approach as needed.

7 Reflection on Capstone

Not applicable for CAS 741

7.1 Which Courses Were Relevant

N/A

7.2 Knowledge/Skills Outside of Courses

N/A