System Verification and Validation Plan for MPIR

Xunzhou (Joe) Ye

April 16, 2025

Revision History

Date	Version	Notes
24 February 2025	1.0	Initial draft
11 April 2025	1.1	Refine according to feedbacks

Contents

List of Tables

1 Symbols, Abbreviations, and Acronyms

symbol	description
API	Application Programming Interface
CI	Continuous Integration
MG	Module Guide
MIS	Module Interface Specification
SRS	Software Requirement Specification
T	Test
VnV	Verification and Validation

2 General Information

This section provides a brief description of the project background and introduces the objectives for the VnV plan.

2.1 Summary

MPIR is a sparse linear solver designed to solve large, sparse real matrices efficiently. It uses the General Minimal Residual (GMRES) method for internal matrix solves and iterative refinement techniques to improve both speed and accuracy. The software is intended for use in computational science, engineering, and numerical analysis applications. As a complete library suite, the software also includes example programs to demonstrate the solver interfaces and practical use cases of the solver.

2.2 Objectives

The primary objective of the Verification and Validation (VnV) plan is to ensure the correctness, accuracy, and efficiency of MPIR in solving sparse linear systems. The secondary objective is to verify usability and maintainability of the software for integration with other numerical libraries.

Usability testing for non-expert users is not prioritized is out of the scope of this VnV plan, as MPIR is only intended for domain-expert users. The solver is expected to use an external library for matrix factorization. The example programs will also depend on an external library for reading and writing sparse matrices in Matrix Market Exchange Format (*Matrix Market: File Formats* 2013).

2.3 Challenge Level and Extras

The challenge level remains the same at the general level. The extra is expected to be conducting a usability test which will be discussed in Section 4.2.2.

2.4 Relevant Documentation

See the Software Requirements Specification (Ye 2025c) for MPIR, which details the goals, requirements, assumptions, and theories of the software.

The Module Guide (Ye 2025a) and Module Interface Specification (Ye 2025b) document the design of MPIR.

3 Plan

This section details the plan for the verification of both the documents and the software for MPIR. The primary artifacts being verified are: SRS, software design, VnV Plan, and implementation.

3.1 Verification and Validation Team

The table below summarizes the VnV Team for the project:

Team Member	Role
Xunzhou Ye	Lead developer and tester
Qianlin Chen	"Domain expert", provides feedbacks on documents per course guidelines and document templates
Dr. Smith	Course instructor, provides feedbacks on documents per course requirements
Dr. Nedialkov	Primary stakeholder, oversees project direction and validates all documents

3.2 SRS Verification Plan

The SRS will be verified through an iterative review process with the project supervisor. The objective is to ensure that the documented software requirements accurately reflect the project's goals, are technically sound, and meet quality standards for scientific software development.

3.2.1 Supervisor Walkthrough Meetings

After each major revision of the SRS, a walkthrough meeting will be scheduled within two weeks. The meeting will be led by the author(s) of the SRS and structured to maximize clarity, engagement, and feedback. To avoid the inefficiencies of a full line-by-line review, the walkthrough will instead:

- Begin with a brief overview of the SRS document structure and purpose, reaffirming the role of each section in capturing software requirements.
- Focus the discussion on key functional and non-functional requirements, underlying assumptions, and any newly introduced models or scope changes.
- Use figures, equations, and tables as prompts for discussion and clarification, ensuring their interpretation aligns with the supervisor's understanding.
- Be guided by open-ended questions posed by the author (e.g., "What are your thoughts on the maintainability test design?" or "How well does the portability requirement reflect our deployment targets?"), rather than binary yes/no queries.
- Encourage clarifications, suggestions, and identification of gaps or inconsistencies in the document.

3.2.2 Feedback Management

Feedback from the walkthrough will be documented as GitHub issues assigned to the project owner. Each issue will include a clear summary of the feedback point, context, and the proposed or required action. The project owner is responsible for addressing and closing these issues through successive iterations of the SRS.

3.3 Design Verification Plan

The VnV team will conduct structured reviews of the design, leveraging their professional expertise to provide informed feedback. This evaluation will be guided by the MG and MIS checklists (Smith 2024; Smith 2022a) to ensure that design principles are adhered to and best practices are followed. Similar to the SRS verification process, all feedback will be documented as GitHub issues to maintain transparency and traceability.

3.4 Verification and Validation Plan Verification Plan

The VnV checklist (Smith 2022b) will be used to review each iteration of the VnV Plan. The goal is to uncover any mistakes and reveal any coverage gaps through the supervision and review of the VnV team members. Once the project reaches a deliverable milestone, the VnV team will check whether the documented testing plans and verification processes have been accomplished and the requirements fulfilled. Feedbacks will again be documented as GitHub issues.

3.5 Implementation Verification Plan

Both automated and manual testing will be performed for this project. For automated static code analysis, linters will be integrated as part of the Continuous Integration (CI) pipeline via GitHub Actions. Details on the choice of tools and its use in the project will be discussed in Section 3.6 below. Plans and schemes for automated dynamic tests will be done at various levels of abstraction, including unit tests, system tests. Details are listed in Section 4 and 5.

In addition to automated testing, a manual code walkthrough will be conducted with the project supervisor. This walkthrough will serve as a key verification activity to ensure that the implementation adheres to the research foundation upon which the project is built. The walkthrough will specifically focus on the implementation of the core numerical algorithms, including:

- The iterative refinement process, which is central to the solver's mixed-precision accuracy and convergence behavior.
- The GMRES method, used for solving the inner linear systems during refinement.

During the walkthrough, the author of the implementation will present the relevant source code, explain design decisions, and describe how each function aligns with the intended mathematical formulations. The project supervisor will review the logic, verify consistency with the theoretical foundations, and provide direct feedback.

3.6 Automated Testing and Verification Tools

The software is expected to be implemented in C++. The following are the language specific tools that will be used for automated testing and verifications:

- CMake (*CMake - Upgrade Your Software Build System* 2025) will be used to streamline the building and testing process of the software. CTest, which is part of CMake, will be used to generate code coverage reports for unit tests.
- doctest (*doctest/doctest* 2025) will be used as the unit testing framework for writing test cases.
- clang-format (*ClangFormat — Clang 21.0.0git documentation* 2025) will be used to format source codes based on a set of predefined rules specified in a configuration file. A Git Hook will be deployed to run a format check to ensure that all committed source codes are properly formatted. The goal of formatting codes is to improve code readability and consistency.
- clang-tidy (*Clang-Tidy — Extra Clang Tools 21.0.0git documentation* 2025) will be used as the linter for static code analysis. Committed codes should be free of linter errors and warnings to minimize the chance of having incorrect codes.
- Benchmark (*google/benchmark* 2025) will be used to evaluate the runtime performance of the solver as part of the nonfunctional verification process.

Apart from these language specific tools, the following general purposed tools are also used:

- GitHub Actions will be used as the CI pipeline for the project. Most of the language specific automated tools mentioned above will be integrated as part of CI to ensure that the verification process is reproducible in an isolated, remote environment. This verifies the verification process itself and further improves confidence of the verification process.

3.7 Software Validation Plan

A valid linear solver must not only produce correct results but also effectively serve its intended real-world applications. In this project, validation will focus on demonstrating that the solver meets its intended performance and usability requirements under realistic conditions.

To validate correctness, the solver will be evaluated on a suite of benchmark problems that represent typical sparse linear systems encountered in computational science and engineering. The computed solutions will be compared against established reference solutions to confirm that they satisfy the underlying equations within the predefined tolerance.

Performance validation will involve automated benchmarking tests that measure runtime, memory usage, and scalability across a range of problem sizes. The solver's performance will be compared against existing solvers, ensuring that the mixed-precision approach delivers the expected improvements in computational efficiency.

Dr. Nedialkov has done some investigations on how mixed-precision computing can reduce computation time and memory usage, and it compares various solvers against a double-precision baseline (Bassi, Derakhshan, and Nedialkov 2022). Benchmarks used and results from this paper will be used to validate both solution accuracy and performance of MPIR.

Finally, validation will extend to acceptance testing in simulated real-world scenarios. These tests will involve applying the solver to representative use cases to confirm that its performance and behavior align with the practical needs of its target users. Feedback from these end-use tests will provide critical insights into any adjustments required before deployment, ensuring that the solver is not only correct in a controlled environment but also effective in practice.

4 System Tests

This section outlines the tests that will be performed for MPIR to verify both the functional and nonfunctional requirements specified in the SRS (Ye 2025c). Input specifications and constraints are also listed in the SRS.

4.1 Tests for Functional Requirements

In this section, the system tests that will be conducted are described in detail. These tests will be used to verify the fulfillment of the functional requirements as listed in the SRS (Ye 2025c).

4.1.1 Matrix Inputs and Outputs

This section covers the requirement R3 of the SRS. This includes essentially a "driver" for the solver which loads sparse matrices from a text file in Matrix Market Exchange (.mtx) Format (*Matrix Market: File Formats* 2013) into memory, invokes the solver interfaces, and outputs the results returned from the solver. The tests described below will verify that such a "driver" is functional.

T1: matrix-io

Control: Automatic

Initial State: Solver and driver initiated.

Input: matrix $\mathbf{A}$ in plain text .mtx format of size 100×100, filled from top to bottom, left to right with integers from 1 to 10 000, a random vector $\mathbf{b}$ of size 100. $u_f = u_w = u_r =$ `double`

Output: The elements of $\mathbf{A}$ matches exactly the one in the .mtx file. Some result solution $\mathbf{x}$ of size 100, values does not matter.

Test Case Derivation: N/A

How test will be performed: Automatic

4.1.2 Correctness Tests with Manufactured Solutions

This section covers one of the ways to verify the requirements R1 and R2 of the SRS. This includes tests on the accuracy of the solution from the solver by manufacturing an exact solution $\mathbf{x}_{\text{ref}}$ to the problem $\mathbf{Ax} = \mathbf{b}$. This manufacturing process loosely follows the scheme below:

1. $\mathbf{x}_{\text{ref}} \leftarrow$ some random vector
2. $\mathbf{b} \leftarrow \mathbf{A}\mathbf{x}_{\text{ref}}$
3. Solve $\mathbf{Ax} = \mathbf{b}$

4. $e \leftarrow \dfrac{\|\mathbf{x} - \mathbf{x}_{\text{ref}}\|_2}{\|\mathbf{x}_{\text{ref}}\|_2}$

The relative error e will be used as the accuracy metric. The values of the manufactured reference solution $\mathbf{x}_{\text{ref}}$ in this section is uniformly distributed in the range of $[\min(a_{i,j}), \max(a_{i,j})]$. For the test cases in Sections 4.1.2, 4.1.3, and below, the `bundle1` matrix (M. Lourakis 2006) from the Florida Sparse Matrix Collection (Davis and Hu 2011) will be used as the input matrix $\mathbf{A}$. This matrix has a size of $10\,581 \times 10\,581$ and $770\,811$ non-zeros. The estimated condition number is 1.3×10^4.

T2: generated-double-double

Control: Automatic

Initial State: matrix $\mathbf{A}$ is read from file and stored in memory. An expected exact solution $\mathbf{x}_{\text{ref}}$ and the corresponding $\mathbf{b} = \mathbf{A}\mathbf{x}_{\text{ref}}$ are prepared.

Input: $\mathbf{A}, \mathbf{b}, u_f = u_w = u_r =$ `double`

Output: $\mathbf{x}$ such that $e = \dfrac{\|\mathbf{x} - \mathbf{x}_{\text{ref}}\|_2}{\|\mathbf{x}_{\text{ref}}\|_2} < 1 \times 10^{-10}$

Test Case Derivation: $\mathbf{x}_{\text{ref}}$ is randomly generated. $\mathbf{b} = \mathbf{A}\mathbf{x}_{\text{ref}}$

How test will be performed: Automatic

T3: generated-single-double

Control: Automatic

Initial State: matrix $\mathbf{A}$ is read from file and stored in memory. An expected exact solution $\mathbf{x}_{\text{ref}}$ and the corresponding $\mathbf{b} = \mathbf{A}\mathbf{x}_{\text{ref}}$ are prepared.

Input: $\mathbf{A}, \mathbf{b}, u_f =$ `single`, $u_w = u_r =$ `double`

Output: $\mathbf{x}$ such that $e = \dfrac{\|\mathbf{x} - \mathbf{x}_{\text{ref}}\|_2}{\|\mathbf{x}_{\text{ref}}\|_2} < 1 \times 10^{-10}$

Test Case Derivation: $\mathbf{x}_{\text{ref}}$ is randomly generated. $\mathbf{b} = \mathbf{A}\mathbf{x}_{\text{ref}}$. Matrix factorization is done in `single` precision while working and residual evaluations are done in `double` precision.

How test will be performed: Automatic

4.1.3 Correctness Tests against Trusted Solvers

This section covers the other way to verify the requirements R1 and R2 of the SRS. This includes tests on the accuracy of the yielded solution from the solver by comparing it to an external, trusted solver to the problem $\mathbf{Ax} = \mathbf{b}$. This process loosely follows the scheme below:

1. $\mathbf{x}_{\text{ref}} \leftarrow$ solution by an external solver
2. Solve $\mathbf{Ax} = \mathbf{b}$
3. $e \leftarrow \dfrac{\|\mathbf{x} - \mathbf{x}_{\text{ref}}\|_2}{\|\mathbf{x}_{\text{ref}}\|_2}$

The relative error e will be used as the accuracy metric. For the test cases in this Section, MATLAB® will be used as the external reference solver.

T4: external-double-double

Control: Automatic

Initial State: matrix $\mathbf{A}$ is read from file and stored in memory. The same problem $\mathbf{Ax} = \mathbf{b}$ is passed to the external solver. A reference solution $\mathbf{x}_{\text{ref}}$ is prepared.

Input: $\mathbf{A}, \mathbf{b}, u_f = u_w = u_r =$ `double`

Output: $\mathbf{x}$ such that $e = \dfrac{\|\mathbf{x} - \mathbf{x}_{\text{ref}}\|_2}{\|\mathbf{x}_{\text{ref}}\|_2} < 1 \times 10^{-10}$

Test Case Derivation: $\mathbf{x}_{\text{ref}}$ is randomly generated.

How test will be performed: Automatic

T5: external-single-double

Control: Automatic

Initial State: matrix $\mathbf{A}$ is read from file and stored in memory. The same problem $\mathbf{Ax} = \mathbf{b}$ is passed to the external solver. A reference solution $\mathbf{x}_{\text{ref}}$ is prepared.

Input: $\mathbf{A}, \mathbf{b}, u_f =$ `single`$, u_w = u_r =$ `double`

Output: $\mathbf{x}$ such that $e = \dfrac{\|\mathbf{x} - \mathbf{x}_{\text{ref}}\|_2}{\|\mathbf{x}_{\text{ref}}\|_2} < 1 \times 10^{-10}$

Test Case Derivation: $\mathbf{x}_{\text{ref}}$ is randomly generated. Matrix factorization is done in single precision while working and residual evaluations are done in double precision.

How test will be performed: Automatic

4.2 Tests for Nonfunctional Requirements

In this section, the system tests that will be conducted are described in detail. These tests will be used to verify the fulfillment of the nonfunctional requirements as listed in the SRS (Ye 2025c).

4.2.1 Accuracy

The accuracy of the solver will be assessed by verifying that it converges to a solution within the user-defined tolerance ϵ. The level of accuracy required for computational science and engineering applications will be evaluated through the relative residual norm after convergence. The functional tests T2, T3, T4, T5 are sufficient to verify the nonfunctional requirement NFR1 in the SRS with an accuracy metric of $\epsilon \approx 1 \times 10^{-10}$. Considering that the residual precision $u_r =$ `double`, the chosen $\epsilon \approx 1 \times 10^{-10}$ is reasonably close to the machine epsilon in `double` precise $\epsilon_{\text{mach}} \approx 1.1 \times 10^{-16}$.

4.2.2 Usability

The usability of the solver will be evaluated based on the clarity and accessibility of its public Application Programming Interface (API). The API should be self-contained, readable, and easy to integrate into other software as a dependency. Usability testing will reference the user characteristics section and include developer feedback. The following tests will be performed to verify the nonfunctional requirement NFR2 in the SRS:

T6: nfr-use

Type: Static, Review, Survey-Based

Initial State: The API documentation and usage examples are available. The API documentation of an external, established sparse linear solver is also presented as a comparable reference. Here the sparse solvers concept from the Eigen library is used as the reference (*Eigen: Solving Sparse Linear Systems* 2025).

Input: A detailed usability survey questionnaire will be distributed to all VnV team members. The survey will include both open-ended and Likert-scale questions focused on API clarity, ease of integration, and documentation quality. VnV team members with expertise in numerical software development will serve as the primary reviewers.

Output/Result: A consolidated report of survey responses will summarize feedback on the API's clarity, ease of integration, advantages and drawbacks, and provide specific suggestions for improvements. The report will highlight areas for enhancement based on a comparative analysis with the external reference API.

How test will be performed:

1. The VnV team will first independently review the API documentation and usage examples.
2. Each reviewer will complete the usability survey (see Appendix 6.2).
3. After submitting the survey, a follow-up meeting will be scheduled where each reviewer presents their feedback, and common themes and specific issues are discussed.
4. The discussion and agreed-upon action items will be documented, and suggestions for improvements will be tracked via GitHub issues.

4.2.3 Maintainability

The effort required to modify the solver should be kept minimal. This will be verified by estimating the complexity of implementing likely changes and ensuring that modifications take less than a fraction of the original development time. The following tests will be performed to verify the nonfunctional requirement NFR3 in the SRS:

T7: nfr-mt

Type: Static, Code Review

Initial State: The codebase is available and structured.

Input: Introduce an API extension to support solving matrices stored in Compressed Sparse Row (CSR) format in addition to the existing support for Compressed Sparse Column (CSC) format.

Output/Result: Measure the number of files and lines of code modified, as well as time taken.

How test will be performed:

1. Before starting, the current state of the codebase will be documented. This includes recording the total number of files, the overall lines of code (LOC), and any existing metrics related to code complexity. These baseline measurements serve as a reference for evaluating the changes.
2. The developer will work on introducing the new CSR support. During this phase, the developer is expected to follow established coding standards and utilize existing modules to minimize unnecessary modifications. The process should include:
   - Designing the API changes needed to accommodate CSR format.
   - Modifying relevant modules to handle both CSR and CSC formats where applicable.
   - Integrating new code with the current codebase using modular and reusable practices.
3. After the API extension is implemented, a code review will be conducted. The review team will assess the following:
   - The number of files modified.
   - The total lines of code added, removed, or altered.
   - The time taken from the start of the implementation to the final integration.
   - An evaluation of the complexity changes introduced by the new code.
4. The measured modifications and time are then compared against the predefined threshold (FRACTION of the original development time). If the modifications exceed the acceptable limits, this could indicate potential maintainability issues that need to be addressed.
5. All findings will be documented in detail, and the results will be discussed in a follow-up review meeting. Any issues or suggestions for reducing the modification effort will be recorded as GitHub issues for further refinement of the codebase.

4.2.4 Portability

The solver should run on all actively maintained operating systems, including Windows 10, Windows 11, Linux, and MacOS. Compatibility testing will verify that all required functionalities work across different platforms. The following tests will be performed to verify the nonfunctional requirement NFR4 in the SRS:

T8: nfr-port

Type: Functional, Automated

Initial State: A Linux machine loaded with Docker images of the latest version of Windows 11, Ubuntu LTS (Linux), and MacOS.

Input: Compile both the solver and the example program in each virtual environment. Run both the example program and all automated functional tests described in Section 4.1.

Output/Result: The solver successfully compiles, executes test cases, and produces correct results.

How test will be performed: Automated CI pipelines will test the software across different platforms. Success will be determined by running the test suite in each environment and confirming consistent results.

4.2.5 Performance

To fulfill the nonfunctional requirement for performance NFR5, a manual benchmark test will be conducted to evaluate the solver's runtime efficiency when using mixed-precision arithmetic. The runtime of the functional tests T2 and T3 will serve as the basis for comparing performance between two configurations:

1. Mixed-precision mode: factorization in single precision, internal solves and residual evaluations in double precision.

2. Double-precision mode: all computations in double precision.

Each test will be executed multiple times to account for variability and ensure statistically stable results. This is necessary because clock-time measurements on modern processors can be affected by factors such as CPU caching, thermal throttling, and OS-level scheduling.

T9: nfr-perf

Type: Functional, Manual, Benchmark-Based

Initial State: As described in T2 and T3. Benchmarking tools with high-resolution timing functions (e.g., std::chrono in C++) are available for accurate runtime measurements.

Input: Manually run the solver in both configurations on the selected matrix. The matrix should be solved multiple times (e.g., 10 runs per configuration) to account for performance variability.

Output/Result: Record the average runtime of each configuration for each matrix. The mixed-precision version should demonstrate a consistent runtime reduction of at least PERF_GAIN. Results can be presented in a comparison chart.

How test will be performed:

The benchmark will be conducted on a single machine to ensure consistency. Prior to testing, CPU frequency scaling will be disabled (e.g., using `cpufreq-set -g performance` on Linux), background processes will be minimized, and the solver will be pinned to a single CPU core using tools such as `taskset` to reduce variability from OS-level scheduling. Before measurement, several warm-up runs will be performed in each configuration to stabilize caching behavior and eliminate anomalies from first-time execution. The solver will then be run 10 times on a single representative sparse matrix using both the mixed-precision configuration (single precision for factorization, double precision for solves and residuals) and the full double-precision configuration. Runtime for each run will be measured using embedded high-resolution timers. The average runtime and standard deviation for each configuration will be calculated and compared. A consistent runtime reduction in the mixed-precision mode will confirm that the performance requirement is met. The results will be visualized in a plot to clearly illustrate the performance difference between the two configurations.

4.3 Traceability Between Test Cases and Requirements

	R1	R2	R3	NFR1	NFR2	NFR3	NFR4	NFR5
T1			X					
T2	X	X		X				
T3	X	X		X				
T4	X	X		X				
T5	X	X		X				
T6					X			
T7						X		
T8							X	
T9								X

Table 1: Traceability matrix showing the connections between test cases and requirements

5 Unit Test Description

This section describes the scope of uniting testing for MPIR and the testing strategy used in each tested modules from the list of modules specified in the MIS (Ye 2025b).

5.1 Unit Testing Scope

Unit testing for Iterative Solver Module (M5) is excluded as this module is the entry point of MPIR. Related testing is covered under system tests as specified in Section 4. The Factorization Module (M2) is also excluded because it is implemented in prior research work (Shahrooz Derakhshan et al. 2023). The correctness of M2 is assumed. Unit tests for Floating Point Concepts Module (M3) and Matrix Operations Module (M4) will be covered below.

5.2 Tests for Functional Requirements

All unit tests below will be performed through automated unit testing.

5.2.1 Floating Point Concepts Module

A heuristic strategy will be used to test the exported routines in this module. Specifically, the two primitive floating-point types, `float` and `double`, will be permuted into all possible 3-element tuples to serve as test inputs. These tuples simulate different mixed-precision configurations. The expected results for each case will be precomputed manually to verify the correctness of the routine outputs. Refer to the source code `fp_concepts.hpp` for details of each test cases.

5.2.2 Matrix Operations Module

Since the mathematical models and computations implemented in this module are fundamental to linear algebra, the testing strategy is intentionally minimal yet effective. It is designed to be:

- Focused on verifying correctness, edge cases, and numerical precision with small sized input vectors and matrices;
- Utilizes analytically known results and approximate comparisons for floating-point correctness.

Refer to the source code `ops.hpp` for details of each test cases.

5.3 Tests for Nonfunctional Requirements

Performance of the solver shall be evaluated as a whole. No module needs to be assessed independently.

5.4 Traceability Between Test Cases and Modules

No specific test case is given in this section, but the unit testing strategy has been covered for each tested module.

References

Bassi, Dylan, Shahrooz Derakhshan, and Ned Nedialkov (Dec. 10, 2022). *Investigating Mixed-Precision in Sparse Linear Solvers and Linear Programming*. 1. Hamilton, Ontario, Canada: McMaster University, p. 18.

Clang-Tidy — Extra Clang Tools 21.0.0git documentation (2025). URL: https://clang.llvm.org/extra/clang-tidy/ (visited on 02/25/2025).

ClangFormat — Clang 21.0.0git documentation (2025). URL: https://clang.llvm.org/docs/ClangFormat.html (visited on 02/25/2025).

CMake - Upgrade Your Software Build System (2025). URL: https://cmake.org/ (visited on 02/25/2025).

Davis, Timothy A. and Yifan Hu (Dec. 7, 2011). "The university of Florida sparse matrix collection". In: *ACM Trans. Math. Softw.* 38.1, 1:1–1:25. ISSN: 0098-3500. DOI: 10.1145/2049662.2049663. URL: https://doi.org/10.1145/2049662.2049663 (visited on 04/11/2025).

doctest/doctest (Feb. 25, 2025). original-date: 2014-08-05T21:43:51Z. URL: https://github.com/doctest/doctest (visited on 02/25/2025).

Eigen: Solving Sparse Linear Systems (2025). URL: https://eigen.tuxfamily.org/dox/group__TopicSparseSystems.html (visited on 04/11/2025).

google/benchmark (Feb. 14, 2025). original-date: 2013-12-12T00:10:48Z. URL: https://github.com/google/benchmark (visited on 02/14/2025).

M. Lourakis (2006). *Lourakis/bundle1*. SuiteSparse Matrix Collection. URL: https://sparse.tamu.edu/Lourakis/bundle1 (visited on 04/11/2025).

Matrix Market: File Formats (Aug. 14, 2013). The Matrix Market. URL: https://math.nist.gov/MatrixMarket/formats.html (visited on 02/14/2025).

Shahrooz Derakhshan et al. (Dec. 27, 2023). *Using Mixed-Precision in the Linear Solver of Ipopt and in QDLDL*. 2. Hamilton, Ontario, Canada: McMaster University, p. 19.

Smith, Spencer (Sept. 1, 2022a). *MIS-Checklist · yex33/MPIR*. GitHub. URL: https://github.com/yex33/MPIR/blob/main/docs/Checklists/MIS-Checklist.pdf.

— (Sept. 1, 2022b). *VnV-Checklist · yex33/MPIR*. GitHub. URL: https://github.com/yex33/MPIR/blob/main/docs/Checklists/VnV-Checklist.pdf.

— (Apr. 2, 2024). *MG-Checklist · yex33/MPIR*. GitHub. URL: https://github.com/yex33/MPIR/blob/main/docs/Checklists/MG-Checklist.pdf.

Ye, Xunzhou (2025a). *MG · yex33/MPIR*. GitHub. URL: https://github.com/yex33/MPIR/blob/main/docs/Design/SoftArchitecture/MG.pdf.
— (2025b). *MIS · yex33/MPIR*. GitHub. URL: https://github.com/yex33/MPIR/blob/main/docs/Design/SoftDetailedDes/MIS.pdf.
— (2025c). *SRS · yex33/MPIR*. GitHub. URL: https://github.com/yex33/MPIR/blob/main/docs/SRS/SRS.pdf.

6 Appendix

This is where you can place additional information.

6.1 Symbolic Parameters

The definition of the test cases will call for SYMBOLIC_CONSTANTS. Their values are defined in this section for easy maintenance.

symbol	value
FRACTION	30 %
PERF_GAIN	10 %

Table 2: Symbolic Parameters

6.2 Usability Survey Questions

1. **Did you find the function names and parameters intuitive?**
   - ☐ Very intuitive
   - ☐ Somewhat intuitive
   - ☐ Neutral
   - ☐ Confusing
   - ☐ Very confusing
2. **Did the provided documentation clearly explain how to use the API?**
   - ☐ Yes, everything was clear
   - ☐ Mostly clear, but some aspects were confusing
   - ☐ Neutral
   - ☐ No, the documentation was unclear
   - ☐ No documentation was provided
3. **Were the example use cases sufficient to understand how to integrate the solver?**
   - ☐ Yes, they covered all necessary cases

- ☐ Somewhat, but additional examples would be helpful
- ☐ No, they were insufficient

4. **How easy was it to set up and call the solver in a basic use case?**
   - ☐ Very easy (Just a few function calls)
   - ☐ Somewhat easy (Needed minor adjustments)
   - ☐ Neutral
   - ☐ Difficult (Required significant effort)
   - ☐ Very difficult (I could not get it working)
5. **Did the API provide useful error messages when incorrect inputs were provided?**
   - ☐ Yes, the error messages were informative
   - ☐ Somewhat, but could be improved
   - ☐ No, the error messages were vague or missing
6. **If you encountered any difficulties, what were they? (Open-ended)**

---

7. **How would you rate the overall usability of the API?**
   - ☐ Excellent
   - ☐ Good
   - ☐ Neutral
   - ☐ Poor
   - ☐ Very Poor
8. **What improvements would you suggest to enhance the usability of the API? (Open-ended)**

---

9. **Would you recommend this solver for integration into a larger numerical computing project?**
   - ☐ Yes
   - ☐ No
   - ☐ Maybe, if improvements are made