Modernising the FERS Software Package

Prepared by:
David Samuel Young
YNGDAV005

Prepared for:
Yaaseen Martin
Department of Electrical Engineering
University of Cape Town

October 31, 2024

Submitted to the Department of Electrical Engineering at the University of Cape Town in partial fulfilment of the academic requirements for a Bachelor of Science degree in Electrical and Computer Engineering

Declaration

1. I know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is one's own.
2. I have used the IEEE convention for citation and referencing. Each contribution to, and quotation in, this report from the work(s) of other people has been attributed and has been cited and referenced. Any section taken from an internet source has been referenced to that source.
3. This report is my own work and is in my own words (except where I have attributed it to others).
4. I have not paid a third party to complete my work on my behalf. My use of artificial intelligence software has been limited to code debugging, formatting assistance, and improving the clarity of written sections.
5. I have not allowed and will not allow anyone to copy my work with the intention of passing it off as his or her own work.
6. I acknowledge that copying someone else's assignment or essay, or part of it, is wrong, and declare that this is my own work.

Word count: 20273

[illegible]

October 31, 2024

David Young

Date

Abstract

This thesis presents the modernisation of the Flexible, Extensible Radar Simulator (FERS), a legacy radar simulation software package, originally developed in C++, to enhance its performance, maintainability, and compliance with modern C++ standards. The project focuses on updating the software to leverage features from C++20/23, optimise multithreading, and improve memory management. Key objectives include refactoring the codebase for better clarity, adopting smart pointers for automatic memory management, and incorporating advanced features like lambda expressions, concepts, and structured bindings.

Extensive testing and validation were conducted to ensure backward compatibility and verify performance improvements. Significant speedups were achieved in radar signal simulations and HDF5 file handling, with performance gains ranging from 1.46x to 3.55x across test cases. Memory management was substantially improved, with all memory leaks eliminated. Introducing a global thread pool and adaptive threading strategies enhanced multithreading efficiency, reducing Central Processing Unit (CPU) load and improving simulation execution times.

The codebase was modularised, technical debt was reduced, and maintainability was improved through better documentation and naming conventions. Regression testing achieved 90.5% line coverage and 95.8% function coverage, ensuring system robustness and accuracy. The modernisation efforts have transformed FERS into a future-ready, high-performance software tool while maintaining its core functionalities for radar simulations. Further improvements, such as migrating to C++ modules and optimising file output processes, are proposed for future work.

Contents

List of Tables

List of Figures

Listings

Glossary

adaptive threading A dynamic multithreading approach that adjusts the number of threads during runtime based on system conditions like CPU load or task complexity to optimise performance.

ATP Acceptance Test Procedure

CI Continuous Integration

CPU Central Processing Unit

CSV Comma-Separated Values

Cyclomatic complexity A software metric that measures the complexity of a program by counting the number of independent paths through the code. It helps assess how difficult a program is to test and maintain.

FERS Flexible, Extensible Radar Simulator

FFTW Fastest Fourier Transform in the West

GCC GNU Compiler Collection

IDE Integrated Development Environment

LTS Long-Term Support

PRF Pulse-Repetition frequency

RAII Resource Acquisition Is Initialization

RAM Random Access Memory

RCS Radar Cross-Section

RMS Root Mean Square

RRSG Radar and Remote Sensing Group

SNR Signal-to-Noise Ratio

SSD Solid-State Drive

XML Extensible Markup Language

Chapter 1

Introduction

The function of good software is to make the complex appear to be simple.

—*Grady Booch*

1.1 Background

Radar (Radio Detection and Ranging) systems are essential components in a wide range of applications, including aviation, defence, weather monitoring, and automotive safety. These systems work by emitting electromagnetic waves and analysing the reflected signals to detect objects and determine their distance, speed, and other characteristics. Given the complexity and critical nature of these systems, accurate simulation is vital for their design, testing, and optimisation.

Signal-level radar simulation plays a crucial role in developing and evaluating radar systems, offering insights into system performance under various conditions without the need for expensive and time-consuming physical testing. Simulations allow engineers to model the behaviour of radar signals, assess different radar system configurations, and refine algorithms before implementation in real-world systems. As technology advances, the need for more sophisticated and efficient simulation tools becomes increasingly important.

1.2 Problem Statement

Flexible, Extensible Radar Simulator (FERS) is a critical tool used within the Radar and Remote Sensing Group (RRSG) for performing signal-level radar system simulations. Originally developed in C++98/03, FERS has been a reliable and efficient tool for radar system simulations, offering users the ability to model complex radar systems and their behaviour under various conditions. Despite its effectiveness, the software has not kept pace with modern developments in programming practices, limiting its scalability, maintainability, and the ability to integrate with newer technologies.

FERS was designed to simulate the behaviour of radar signals, enabling users to perform detailed analyses of radar performance, including target detection, signal reflection, and environmental interference. However, the software's ageing codebase employs outdated coding conventions and lacks features introduced in modern C++ standards.

Updating the system to C++20/C++23 would allow for the implementation of more efficient algorithms, better memory management, and enhanced code readability through features like lambda expressions, concepts, ranges, and coroutines. This modernisation would also introduce more robust error handling, optimise the performance of the simulation, and streamline maintenance, ensuring the tool remains useful for future research and industry applications. Additionally, incorporating new testing frameworks and benchmarking tools will help validate that the updates maintain or improve the simulation's accuracy and reliability without breaking existing functionality.

1.3 Objectives

The primary objective of this project is to modernise the FERS software package by:

1. **Upgrading the Codebase:** Refactor the existing C++ code to incorporate features from C++20/C++23, improving readability, maintainability, and efficiency.
2. **Optimisation:** Implement performance optimisations where applicable, ensuring the simulator operates efficiently without compromising accuracy.
3. **Compatibility Maintenance:** Ensure that the updated software remains compatible with its original design and functionality, maintaining the integrity of the simulation results.
4. **Testing and Validation:** Develop and conduct tests to verify that the modernised software performs as expected, comparing the updated simulator against the original version to ensure accuracy and reliability.

1.4 Scope & Limitations

This project will cover the modernisation of the FERS software package, focusing on updating the codebase to modern C++ standards, optimising performance, and ensuring compatibility with existing functionalities. The scope includes:

- Detailed analysis of the current FERS codebase
- Implementation of modern C++ features
- Performance optimisation
- Testing and validation

However, this project has some limitations:

- The project must be completed within a 13-week period, from the start of the semester on 22nd July 2024 to the submission deadline on 24th October 2024. This limited time frame potentially restricts the extent of code refactoring and optimisation that can be performed.
- The focus will be on modernising the existing functionality rather than introducing new features or substantial algorithmic changes. The project aims to maintain the integrity of the current simulation processes.

- Due to the `config_validators/` directory containing newly added code that is isolated from FERS itself, it will not be considered for modernisation, however, it is discussed throughout the project and in mentioned for future work in Chapter 6.

These constraints mean that the project will prioritise essential modernisation tasks that can be accomplished within the given time frame, ensuring that the core objectives are met without overextending the scope of work.

1.5 Report Outline

This report is structured as follows:

- **Chapter 1: Introduction** - An introduction to the project, presenting the context and significance of the research. Also highlights the scope of the project and any limitations which apply.
- **Chapter 2: Literature Review** - An overview of radar systems, radar simulators, and software modernisation strategies, with comparisons to existing work in the field.
- **Chapter 3: Methodology & Design** - A comprehensive outline of the systematic approach taken to modernise the FERS software, covering the analysis of the existing system, design strategy for modernisation, tools and technologies used, testing and validation methodology, risk management, project planning, and justification of design choices.
- **Chapter 4: Implementation** - A detailed account of the practical steps taken to modernise the FERS software, including code refactoring, application of optimisation techniques, testing and validation processes, and documentation updates.
- **Chapter 5: Results & Analysis** - Presentation and analysis of the results obtained from the modernised software, as well as an evaluation of the project's objectives, challenges encountered, and the significance of the findings.
- **Chapter 6: Conclusions & Future Work** - Summary of the project outcomes and suggestions for future work.
- **Appendix A: Code** - Snippets of code which are too long to provide in the main report content and a link to the GitHub repository.
- **Appendix B: Theoretical Background of Radar Systems** - A detailed discussion on the fundamental principles of radar systems and advanced radar concepts relevant to the simulation.

Chapter 2

Literature Review

You can't really understand what is going on now unless you understand what came before.

—*Steve Jobs*

This chapter establishes the context for modernising the FERS software package by surveying relevant literature. The primary focus is software modernisation, particularly within C++ programming and simulation software. This review focuses on software development practices, optimisation strategies, and the integration of modern C++ standards into legacy codebases. The review starts with a brief, high-level discussion of radar theory to provide a basic understanding.

2.1 Introductory Radar Theory

Radar systems are designed to detect and measure the distance, speed, and other characteristics of objects by transmitting electromagnetic waves and analysing the echoes that return after interacting with these objects. Understanding the fundamental principles of radar is essential for comprehending the simulation of radar systems. This section provides a very high level overview of the in-depth radar theory content found in Appendix B.

2.1.1 Radar System Components

A typical radar system comprises several key components:

- **Transmitter:** Generates and amplifies the signal before it is transmitted through the antenna.
- **Antenna:** Directs the radar signal toward the target and collects the reflected signal.
- **Receiver:** Amplifies and processes the returned signal, extracting useful information such as range, speed, and angle of the target.
- **Signal Processor:** analyses the received signal to detect targets and measure their parameters.
- **Display and Control:** Interfaces that allow operators to monitor and control radar operation.

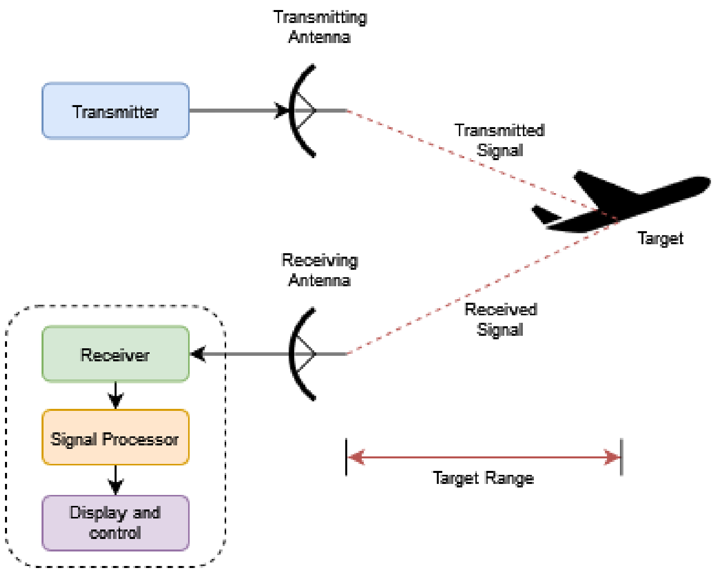


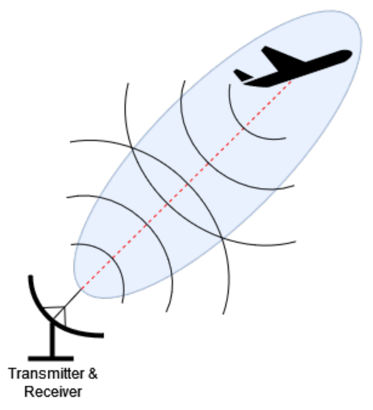
Figure 2.1: Diagram illustrating a bistatic radar system, showing the key components

2.1.2 Types of Radar Configurations

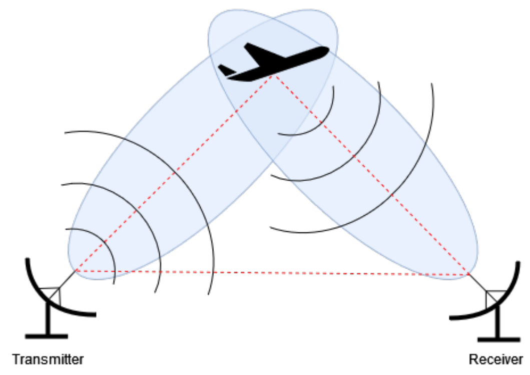
Radar systems can be categorised based on the configuration of their transmitter and receiver:

- **Monostatic Radar:** A single antenna is used for both transmitting and receiving the signal. This is the most common radar configuration.
- **Bistatic Radar:** The transmitter and receiver are located at different positions. This setup can offer advantages in detecting certain types of targets, especially in low-observability scenarios.
- **Multistatic Radar:** Involves multiple transmitters and receivers distributed over a large area, providing enhanced detection capabilities and resilience to jamming, making it suitable for complex surveillance operations.

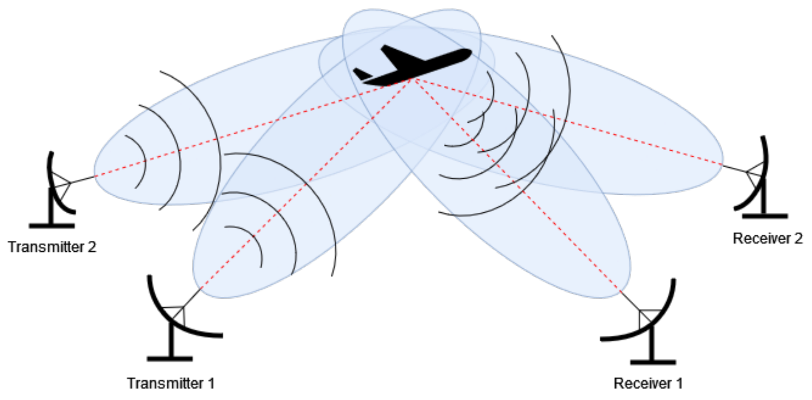
These categories are illustrated in Figure 2.2 below.



(a) Monostatic radar system



(b) Bistatic radar system



(c) Multistatic radar system

Figure 2.2: Illustrations of monostatic, bistatic, and multistatic radar systems.

2.2 Background on Radar Simulation Software

Radar simulation software plays a crucial role in the development and testing of radar systems. These tools allow engineers and researchers to model and analyse the performance of radar systems under various conditions, without the need for costly and time-consuming physical testing. Simulations are particularly essential in the early stages of radar system design, where they can provide valuable insights into system behaviour, help optimise designs, and identify potential issues before actual implementation.

2.2.1 Overview of Radar Simulation

Radar systems are complex assemblies of hardware and software designed to detect, track, and analyse objects at a distance [1]. These systems operate by transmitting electromagnetic waves and analysing the echoes that return after bouncing off objects. The simulation of radar systems involves replicating this process in a virtual environment, allowing for the study of system performance under various scenarios, including different environmental conditions, target characteristics, and signal processing techniques [2].

Radar simulations are critical in research and development, particularly in the fields of defence, aviation, and meteorology [3, 2]. They enable the testing of new radar designs, the evaluation of signal processing algorithms, and the assessment of radar system vulnerabilities. By providing a controlled and repeatable environment, simulations help reduce the risks associated with deploying new radar technologies in the field.

2.2.2 Historical Context of Radar Simulators

The evolution of radar simulators has paralleled advances in computational power and software techniques, leading to increasingly sophisticated tools that serve various simulation needs. Early radar simulators, such as Boothe's 1964 statistical model [4], focused on predicting performance metrics like detection probabilities and coverage maps without generating actual radar signals. These models were efficient but limited in scope.

As the field progressed, more detailed simulation types emerged:

- **Result Simulators:** Focused on generating the final outputs expected from radar systems after signal processing, these simulators are used for training and scenario analysis without modelling the full radar operation. Examples include the SAR image simulator, SARviz [5].
- **Statistical Simulators:** These models offer predictions based on statistical properties, useful for performance data like coverage maps but are limited to systems similar to those modelled.
- **Signal-Level Simulators:** These simulators, such as RadSim [6] and SARSIM II [7], model the raw signals received by radar systems, providing detailed simulations of signal interactions with the environment.
- **Electromagnetic Simulators:** These tools model electromagnetic fields at discrete points, useful for component-level analysis but less suited for whole-system simulations.

FERS is a signal-level radar simulator used within the RRSG. Developed in C++98/03, FERS provides detailed radar signal simulations.

2.2.3 Existing Radar Simulation Tools

Several radar simulation tools are available today, each offering unique capabilities and strengths. Some of the most commonly used radar simulators include:

- **MATLAB/Simulink Radar Toolbox:** A widely used tool in both academia and industry, MATLAB's Radar Toolbox provides a comprehensive environment for radar signal processing and system simulation. It offers a range of built-in functions for modelling radar signals, antennas, propagation environments, and target models [8].
- **AWR Microwave Office:** A commercial software package that offers a suite of tools for the design and simulation of RF and microwave systems, including radar. It provides detailed simulation capabilities at the circuit, system, and electromagnetic levels, making it a popular choice for designing and testing radar hardware components [9].

- **SystemVue by Keysight Technologies:** This tool focuses on the system-level design and simulation of radar systems. It allows engineers to model and simulate radar signal chains, including RF front-end components, digital signal processing algorithms, and overall system performance [10].

Compared to these tools, FERS offers a unique combination of flexibility and extensibility, particularly in its ability to handle detailed signal-level simulations. However, FERS's outdated codebase limits its usability in modern research and development environments, where newer tools often provide more user-friendly interfaces and support for the latest software development practices.

2.3 modernisation of Legacy Software

modernising legacy software is crucial for extending the longevity and utility of software systems that have become outdated due to advancements in technology and evolving industry standards. This section explores the challenges of legacy software, various modernisation approaches, and potential pitfalls that may arise during the process.

2.3.1 Introduction to Legacy Software Issues

Legacy software often presents challenges such as outdated codebases, insufficient documentation, and difficulties integrating with modern technologies [11, 12]. These issues contribute to 'software rot' or 'code decay,' a phenomenon where a software system's structure and functionality deteriorate over time, making maintenance and updates increasingly difficult and error-prone [13]. Eick et al.'s study highlights the escalating costs and risks of maintaining aging software, underscoring the necessity for proactive modernisation efforts to counteract technical debt and preserve software integrity [14].

2.3.2 Approaches to Software modernisation

Various strategies can be employed to modernise legacy software, each offering unique benefits and challenges. Common approaches include refactoring, re-engineering, and wrapping [11, 12]. Refactoring involves restructuring existing code to improve quality, readability, and maintainability without altering its external behaviour [15]. Re-engineering takes a more comprehensive approach, potentially redesigning and re-implementing parts of the system to align with modern practices [16, 17]. Wrapping involves creating interfaces around legacy systems to facilitate interaction with newer technologies or to isolate and replace outdated components gradually [18].

The KDE [19] community's efforts to update their extensive codebase to modern C++ standards offer a compelling example of successful software modernisation. By adopting C++11, C++14, and C++17 features, such as lambda expressions, auto-typed variables, and range-based loops, the community achieved significant improvements in code conciseness and expressiveness [20]. However, this process also highlighted the challenges of maintaining compatibility with existing libraries and tools while managing the cognitive load associated with adopting new language features.

2.3.3 Challenges in modernisation

modernising legacy software is fraught with challenges. Ensuring backward compatibility is critical to maintaining software reliability and stability [21]. Any modernisation efforts must be carefully managed to avoid introducing new bugs or disrupting existing functionality. The balance between reaping the benefits of modern features and managing technical debt is crucial, as highlighted by the KDE community's experience [20].

Technical debt, the trade-off between short-term fixes and long-term maintenance costs, can become a significant barrier if not managed effectively [11, 12]. The modernisation process is resource-intensive, often requiring substantial time and effort from developers [12]. Automated tools like Clang-Tidy [22] can alleviate some of this burden by identifying modernisation opportunities and automating portions of the refactoring process, thereby enhancing efficiency and reducing the risk of errors.

2.4 Best Practices in C++ modernisation

Bringing a legacy C++ codebase up to modern standards necessitates adherence to best practices that ensure code quality, maintainability, and performance [23]. This section delves into the critical aspects of refactoring, memory management improvements, and the adoption of modern C++ features, all essential for updating an outdated system.

2.4.1 Refactoring and Code Quality Improvement

Refactoring plays a pivotal role in enhancing the quality and maintainability of legacy software. It involves systematically restructuring code to improve readability, reduce complexity, and eliminate redundancy while preserving external behaviour [11]. The application of modern design patterns and C++ features, such as lambda expressions, auto-typed variables, and range-based loops, has been shown to reduce boilerplate code [24] and enhance software expressiveness, as demonstrated in the KDE community's modernisation efforts [20].

To maintain code quality during modernisation, static analysis and automated refactoring tools like Clang-Tidy are invaluable. These tools help ensure consistent application of best practices across the codebase, minimising the risk of introducing errors during modernisation [22].

2.4.2 Memory Management Improvements

A significant advancement in modern C++ is the transition from manual memory management to the use of smart pointers and Resource Acquisition Is Initialization (RAII) principles [25, 26]. Legacy systems often rely on manual memory management, which can lead to issues like memory leaks and dangling pointers. modernising such systems involves adopting smart pointers like `std::unique_ptr` and `std::shared_ptr`, which automate memory management and ensure resources are properly released, reducing the likelihood of memory-related errors [26].

These constructs, introduced in C++11, have simplified memory management by eliminating the need for explicit delete calls and reducing the potential for errors, thereby making the code more robust and easier to maintain [26].

2.4.3 Naming Conventions

Consistent and concise naming conventions are essential for improving code readability and maintainability in complex software systems. Poor naming practices, such as the use of abbreviations, contractions, and overly generic names, can obscure code meaning and increase the likelihood of errors during modifications. Caprile and Tonella emphasise that meaningful and descriptive function names are vital for understanding a program, as they convey critical information about code elements [27]. They advocate for standardised naming conventions, using a dictionary of terms to ensure clarity and consistency.

Deissenboeck and Pizka similarly highlight the risks of synonym misuse and generic names, noting that these practices contribute to code decay over time. They stress the importance of naming conventions that reflect underlying concepts, thereby enhancing both readability and maintainability [28].

Table 2.1 below details some of the popular naming conventions used in C++.

Convention	**Description**	**Example**
camelCase	Start with a lowercase letter, with each subsequent word capitalised.	`myVariable`, `processData()`
PascalCase	Start with an uppercase letter, with each subsequent word capitalised.	`MyClass`, `ProcessData()`
snake_case	Words are in lowercase, separated by underscores.	`my_variable`, `process_data()`
UPPER_SNAKE_CASE	All letters uppercase, words separated by underscores. Typically used for constants or macros.	`MAX_VALUE`, `PI_CONSTANT`
_leadingUnderscore	Used to indicate private members or variables.	`_privateVariable`, `_processData()`
Hungarian Notation	Prefix that denotes type or purpose of the variable.	`iCount` (integer), `pData` (pointer)

Table 2.1: Popular C++ naming conventions with examples

These naming conventions, when applied consistently, help ensure that code is both readable and maintainable across large codebases.

2.5 Incorporating C++20/C++23 Features

modernising legacy C++ software packages involves integrating advanced features from the latest C++ standards, particularly C++20 and C++23 in recent years. These new standards introduce a range of powerful tools and improvements that can significantly enhance the efficiency, maintainability, and performance of the existing codebase. This section explores the differences between C++98/03 (the C++ version FERS was originally developed with) and C++20/23, discussing their benefits, challenges, and the considerations necessary for their successful implementation in legacy systems.

2.5.1 Differences Between C++98/03 and C++20/23

The evolution of C++ from the C++98/03 standards to C++20/23 represents a significant leap in both language features and standard library capabilities. C++98 and C++03 laid the foundation for modern C++, focusing on stability and performance. However, they lacked many of the conveniences and safety features that developers now expect. These older standards relied heavily on manual memory management, extensive use of macros, and lacked support for multithreading and modern programming paradigms such as lambda expressions, smart pointers, and type inference.

In contrast, C++20/23 introduces numerous features that simplify code, enhance safety, and boost performance. Key differences include:

- **Memory Management:** C++98/03 primarily used raw pointers and manual memory management, leading to common issues such as memory leaks and undefined behaviour. C++20/23 promotes the use of smart pointers like `std::unique_ptr` and `std::shared_ptr`, which automate resource management through RAII principles, drastically reducing the risk of memory-related errors [26].
- **Language Features:** C++20 introduces concepts and ranges, which greatly improve template programming and simplify common data manipulation tasks. Concepts allow for more precise specification of template parameters, reducing errors and improving code readability [29]. Ranges provide a modern way to work with sequences of data, making algorithms more intuitive and less error-prone [30].
- **Concurrency:** While C++98/03 had no built-in support for multithreading, C++11 and later standards introduced a robust threading library, atomic operations, and synchronisation primitives, enabling safer and more efficient concurrent programming [31].
- **Modules:** One of the most transformative features introduced in C++20 is modules. Unlike traditional header files, modules provide a more efficient way to organise and compile code, reducing dependencies and potentially improving compilation times [32]. This is especially beneficial for large-scale systems, where compilation speed and code organisation are critical.
- **Coroutines:** C++20 also introduces coroutines, which allow for writing asynchronous code in a more natural and readable way, without the need for complex state machines or callback chains [33]. This can be particularly useful in simulation environments where responsiveness and performance are key [34].
- **Library Enhancements:** The standard library has also seen significant improvements. C++20/23 includes new containers, algorithms, and utilities that were not available in C++98/03, making it easier to write efficient and expressive code. The introduction of `std::span`, `std::format`, and new algorithms like `std::ranges::sort` represent just a few of these enhancements.

To further illustrate these differences, Table 2.2 summarises the key improvements from C++98/03 to C++20/23:

Feature	**C++98/03**	**C++20/23**	**Benefits of modernisation**
Memory Management	Manual memory management with raw pointers	Smart pointers (`std::unique_ptr`, `std::shared_ptr`)	Reduces memory leaks, simplifies resource management
Multithreading	No built-in support	Threading library, atomic operations, coroutines	Easier and safer concurrent programming
Type Inference	Requires explicit types	`auto`, `decltype` for inference	Simplifies code, reduces redundancy
Templates	Basic template usage	Concepts for template constraints	Increases safety and clarity
Error Handling	Manual error handling	`std::optional`, `std::variant`, `std::expected`	More expressive, safer error handling
Code Organisation	Headers and source files	Modules	Improves compile times, reduces dependencies
Range-based Loops	Not available	Range-based loops	More concise, readable code
Standard Library	Limited containers and algorithms	Expanded containers, new algorithms	Enhances productivity with more utilities
Lambdas	Not available	Lambda expressions	Enables functional programming, simplifies callbacks

Table 2.2: Comparison of C++98/03 and C++20/23 features

The transition from C++98/03 to C++20/23 is not just about adopting new syntax; it's about leveraging these advancements to write more robust, maintainable, and performant software.

2.5.2 Challenges and Considerations

While the benefits of adopting C++20 and C++23 features are clear, the transition is not without challenges. The integration of new features into an existing codebase like FERS requires careful planning to avoid introducing new bugs or breaking existing functionality. The experience of the KDE community, as discussed in previous sections, underscores the importance of balancing the adoption of modern features with the need to maintain stability and compatibility with existing code [20].

Furthermore, the cognitive overhead associated with learning and applying these new features can be significant, particularly for teams that are not yet familiar with C++20/23. To mitigate these challenges, it is crucial to adopt a gradual approach to modernisation, supported by automated tools like Clang-Tidy, which can assist in the refactoring and validation of the updated codebase [22].

2.6 Testing and Validation in Software modernisation

Testing and validation are crucial in ensuring that modernisation efforts do not introduce new bugs or degrade software performance. This section discusses the importance of testing, methods for comparative analysis and benchmarking, and the role of continuous integration and automated testing in the modernisation process.

2.6.1 Importance of Testing in modernisation

As legacy systems undergo modernisation, ensuring that these changes do not introduce new bugs or degrade the performance of the software is crucial. modernisation efforts, such as refactoring code, adopting new language features, and optimising performance, can inadvertently alter the behaviour of the system if not carefully managed [15]. Testing is the primary safeguard against such risks, providing the necessary validation that the modernised system meets the original software's functional and performance requirements.

The literature emphasises the need for a robust testing strategy when modernising software, with a focus on regression testing to ensure that existing functionality is preserved [35]. For instance, studies on modernising large codebases, such as the KDE community's transition to modern C++ standards, highlight the critical role of testing in detecting and resolving issues that arise during the adoption of new language features and optimisation techniques [20]. Testing not only ensures that the software continues to function correctly but also validates that the changes have improved the codebase as intended, without introducing regressions or new vulnerabilities.

2.6.2 Comparative Analysis and Benchmarking

Benchmarking is vital in assessing the performance and accuracy of modernised software compared to its legacy counterpart [36]. For example, in modernising FERS, benchmarking could involve comparing the processing speed of radar simulations before and after incorporating C++20/23 features. This analysis ensures that performance enhancements are realised without compromising functionality.

2.6.3 Continuous Integration and Automated Testing

Continuous Integration (CI) and automated testing are crucial for maintaining stability throughout the modernisation process. CI pipelines, integrated with tools like Clang-Tidy [22], support automated testing, ensuring that every change to the codebase is immediately validated [37, 38]. This approach is particularly beneficial in a modernisation project, where frequent updates are necessary to align with modern standards.

GitHub Actions is a widely used CI tool that automates workflows directly from a GitHub repository. It allows developers to set up CI pipelines that automatically run tests, perform code quality checks, and deploy updates whenever changes are made to the codebase [39]. GitHub Actions can be configured to continuously test the integration of C++20/23 features, ensuring that new code does not disrupt existing functionality. Additionally, it supports regression testing and performance benchmarking, helping validate that the modernised software meets or exceeds the original system's performance [40].

2.7 Critical Review

2.7.1 Summary of Key Insights

The literature on software modernisation offers valuable insights for updating FERS. Key strategies such as refactoring and re-engineering are highlighted for their potential to enhance code readability, maintainability, and performance. Modern C++ features like smart pointers, lambda expressions, and C++20/23 advancements (e.g., concepts, ranges, modules) are particularly relevant, offering significant improvements in code safety and efficiency. The KDE community's modernisation efforts serve as a practical example of these benefits [20].

2.7.2 Evaluation of modernisation Strategies

Refactoring is favoured for its incremental, low-risk approach, ideal for maintaining FERS's stability. However, its effectiveness depends on careful planning to avoid disrupting existing functionality [15]. Re-engineering, while offering deeper improvements, is resource-intensive and riskier [17, 16]. For FERS, a hybrid approach combining refactoring with selective re-engineering appears most suitable, balancing modernisation needs with practical constraints.

2.7.3 Gaps in Existing Literature

The literature reveals gaps, particularly in the specific challenges of modernising specialised simulation software like FERS. There's limited research on integrating modern C++ features in such contexts, and the impact on performance remains underexplored.

Chapter 3

modernisation Strategy & Design

Plans are worthless; planning is everything.

—*Dwight D. Eisenhower*

This chapter outlines the approach taken to modernise the FERS software package. It begins with an analysis of the current system, identifying areas for improvement. This analysis informs the design strategy, which focuses on refactoring the code, integrating modern C++20/23 features, and optimising performance while maintaining the software's reliability and functionality.

The chapter also details the tools and technologies used in the modernisation process, ensuring efficiency and alignment with modern development practices. A comprehensive testing and validation methodology is presented to verify that the updated software meets all performance and functionality requirements.

3.1 Project Methodology Overview

The modernisation of the FERS is centred around updating its code to modern C++ standards while preserving core functionality. The process begins with a thorough analysis of the existing codebase to identify areas for improvement, including outdated practices and performance bottlenecks. Based on this analysis, the modernisation focuses on refactoring the code to incorporate modern C++20/23 features such as smart pointers, multithreading, and lambda expressions, improving both performance and maintainability.

The project follows an incremental approach, introducing changes step by step, with rigorous testing and benchmarking at each stage. This ensures that the software remains stable and backward-compatible with existing data formats. Comprehensive regression tests and performance evaluations confirm that the updated FERS meets or exceeds the performance of the original.

In addition to technical updates, the project places emphasis on clear documentation using tools like `Doxygen`, ensuring the software remains easy to understand and extend. The result is a modernised version of FERS that retains its reliability and speed while benefiting from modern software practices.

3.2 Analysis of the Current FERS System

The modernisation of the FERS software package begins with a thorough analysis of its existing state. This section presents a detailed examination of the current structure, coding practices, and performance characteristics of FERS, highlighting the specific areas that require modernisation. This analysis is essential for identifying the best strategies for updating the software to meet modern C++ standards while maintaining its reliability and performance.

3.2.1 Directory Structure and Code Organisation

The FERS software package is organised into a modular directory structure, which facilitates the separation of core functionalities from auxiliary tools and tests. The structure is as follows:

- **`src/` Directory:** The most substantial part of the codebase, the `src/` directory contains 91.1% of the total code, with 5,536 lines dedicated to the primary functionality of FERS. This directory houses the core components such as radar signal processing, timing, antenna modelling, and data import/export functionalities.
- **`utilities/` Directory:** Comprising 4.16% of the codebase, this directory includes tools for specific tasks, such as data conversion and clutter generation, that support the main simulation operations.
- **`fftwcpp/` Directory:** This directory, representing 3.26% of the code, contains the wrapper and integration code for the Fastest Fourier Transform in the West (FFTW) library. However, as discussed in Section 3.4.5 below, FFTW is not used in the main FERS code.
- **`test/` Directory:** Although only 1.48% of the codebase, the `test/` directory plays a crucial role in maintaining software reliability through regression tests and unit tests. The presence of these tests indicates an existing emphasis on ensuring that updates do not introduce regressions, though the scope and coverage of these tests may need to be expanded during modernisation.

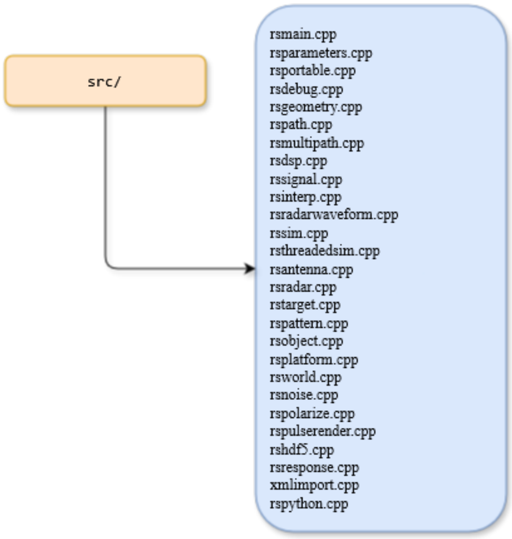


Figure 3.1: Current directory structure of the `src/` directory in the FERS codebase

This semi-modular organisation of the codebase suggests a separation of concerns within the FERS codebase, which is beneficial for the modernisation process. However, the overall structure will likely need to be reviewed and reorganised to improve the modularity.

3.2.2 Code Metrics and Quality Assessment

A detailed code quality analysis, conducted using CLion [41] and Understand [42], provides insights into the current state of the FERS codebase:

- **Lines of Code:** The FERS codebase consists of 10,128 lines, with 6,077 lines dedicated to source code, 2,655 lines to comments, and 1,222 lines to blank spaces. The comment-to-code ratio of 0.44 indicates a relatively well-documented codebase, though the high line count suggests opportunities for refactoring to improve readability and maintainability.

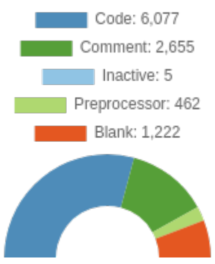


Figure 3.2: Distribution of lines of code in the codebase

- **Function Complexity:** The Cyclomatic complexity [43] of the functions was assessed to understand the maintainability and potential risks associated with the code. The analysis revealed that most functions have a Cyclomatic complexity within the range of 1-10, which is considered low-risk. However, a few key functions exhibit higher complexity:
  - **RunThreadedSim:** Cyclomatic complexity = 15 (Moderate complexity, B grade)
  - **Render:** Cyclomatic complexity = 13 (Moderate complexity, B grade)
  - **ReadAndDump:** Cyclomatic complexity = 12 (Moderate complexity, B grade)

  While most of the code is straightforward and manageable, these more complex functions could benefit from refactoring to reduce their complexity and improve testability.

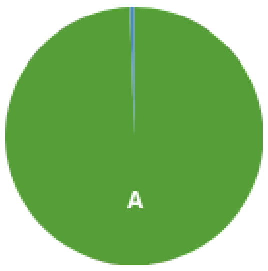


Figure 3.3: Function complexity of the codebase

- **Technical Debt:** The FERS codebase has an overall technical debt rating of 'C,' indicating a moderate level of technical debt. Key contributors to this technical debt include:
  - **Unused Entities:** Several files, such as `xmlimport.cpp` and `fftwcpp.cpp`, contain unused code elements, which contribute to code bloat and potential confusion.
  - **Special Member Functions:** The code exhibits multiple violations related to the improper implementation of special member functions, which can lead to issues with object lifecycle management and memory safety.
  - **Manual Memory Management:** FERS, written in C++98/03, relies heavily on manual memory management using raw pointers. This approach is prone to errors such as memory leaks and dangling pointers, which modern C++ techniques like smart pointers can mitigate.

Addressing these issues will be a primary focus of the modernisation process, particularly the transition from manual to automated memory management using C++11 or later features.

3.2.3 Code Inspection Report

The CLion [41] inspection report provides further insights into common practices within the FERS codebase, identifying a total of 2,126 informational warnings, 1,113 warnings, and 4 compiler errors related to potential code quality issues. Table 3.1 below summarises these findings.

Category	Errors	Warnings	Information
Common Practices and Code Improvements	N/A	N/A	769
Compiler Errors	4	N/A	N/A
Constraints Violations (naming conventions)	N/A	N/A	891
Data Flow Analysis	N/A	34	N/A
Formatting	N/A	177	N/A
Potential Code Quality Issues	N/A	79	13
Redundancies in Code	N/A	N/A	277
Static Analysis Tools (Clang-Tidy)	N/A	508	N/A
Syntax Style	N/A	315	176

Table 3.1: Summary of CLion code inspection findings

These findings underscore the importance of adopting modern C++ features and best practices to enhance the maintainability and reliability of the FERS software.

3.2.4 External Dependencies

FERS relies on several external libraries, including `TinyXML`, FFTW, `HDF5`, and `Boost`, which are essential for the software's functionality but introduce potential challenges. `TinyXML`, though lightweight and effective for parsing XML files, offers limited functionality compared to modern XML libraries and may need to be reviewed for compatibility with updated C++ standards. FFTW is not used in the main FERS codebase even though it has been included in the repository in the `fftwcpp/` directory. The

HDF5 C API is crucial for managing large datasets, and its use in FERS must be optimised to align with modern C++ practices. `Boost`, while providing extensive utilities that enhance C++ capabilities, introduces complexity due to its heavy reliance on templates. As part of the modernisation process, it may be necessary to evaluate whether Boost's functionality can be replaced by more streamlined features available in C++11 and later.

3.2.5 Summary of Analysis

The analysis of the current FERS system highlights several areas requiring modernisation. Certain complex functions need refactoring to improve maintainability and reduce error risk. The system's 'C' technical debt rating stems from unused code, improper special member functions, and manual memory management, which should be replaced with modern C++ features like smart pointers. Enhancing `const` correctness and replacing C-style casts will further improve code safety. Redundant code, uninitialised members, and incomplete functions also call for clean-up. Lastly, reliance on external libraries like `TinyXML`, FFTW, HDF5 C API, and `Boost` must be carefully reviewed to ensure compatibility with modern C++ standards and streamline integration.

3.3 Interpretation of User Requirements

The user requirements presented in Table 3.2 have been derived from the project brief. These requirements form the foundation for developing the functional requirements, which will guide the design specifications outlined in the following sections.

User Requirement ID	Description
UR01	**Usability:** The software should be easy to use and well-documented for research purposes.
UR02	**Performance:** The software must perform at least as efficiently as the original FERS software, with improved execution speed where applicable.
UR03	**Compatibility:** Ensure backward compatibility with original data formats and output structures.
UR04	**Maintainability:** The code should follow modern C++ standards for maintainability and future enhancements.
UR05	**Reliability:** The modernised software should produce the same or better results in terms of accuracy and stability as the original system.
UR06	**Extensibility:** The system must allow for future expansions without requiring significant rewrites.

Table 3.2: Interpretation of user requirements

3.4 Design Strategy for modernisation

The modernisation of the FERS software package will be guided by a carefully planned design strategy, focusing on integrating modern C++ features, optimising performance, and improving the overall maintainability of the codebase. This section outlines the specific design choices and approaches that will be employed in this modernisation effort.

3.4.1 Functional Requirements

The functional requirements in Table 3.3 below are derived from the user requirements outlined in the previous section. These functional requirements guide the design and implementation of the software, ensuring that the modernisation process meets the objectives of usability, performance, compatibility, and reliability.

Functional Requirement ID	**Description**	**User Requirement Addressed**
FR01	Simulate signal-level radar operations with accurate results, supporting backward compatibility and future expansions.	**UR01**, **UR03**, **UR05**, **UR06**
FR02	Use multithreading for efficient handling of signal processing and rendering tasks, ensuring optimal performance and scalability.	**UR02**, **UR04**
FR03	Ensure data handling uses modern libraries for input/output (e.g., HDF5, XML) with robust error handling.	**UR03**, **UR04**, **UR05**
FR04	Implement comprehensive benchmarking and validation tools for performance, accuracy, and backward compatibility checks.	**UR02**, **UR05**
FR05	Provide detailed documentation and inline comments following modern C++ standards for ease of use and maintainability.	**UR01**, **UR04**

Table 3.3: Functional requirements and user requirements mapping

3.4.2 Incremental Development and modernisation Process

The modernisation of the FERS software follows an incremental development approach. This step-by-step method focuses on making controlled improvements while ensuring the system remains stable and efficient. The process is designed to introduce refactoring, new C++20/23 features, and performance optimisations, with each change validated through thorough testing and benchmarking.

The key stages of this process include establishing baseline benchmarks, refactoring code (such as implementing smart pointers, lambda expressions, and C++-style casting), and gradually integrating modern C++20/23 features like concepts, ranges, and modules. Performance optimisation will be a central focus, including multithreading and algorithmic enhancements to ensure that radar simulations run efficiently. Modularity improvements will also enhance the organisation and maintainability of the codebase.

At each major stage of modernisation, regression testing and performance profiling will be performed. Regression tests will be expanded and executed to ensure that new code functions as expected while profiling and benchmarking tools like `valgrind` [44] and `perf` [45] will be used to measure performance impacts and guide further optimisation. This approach ensures that all changes are validated and optimised without disrupting overall system functionality.

3.4.3 Refactoring Strategy

The modernisation of FERS begins with a thorough refactoring of the existing codebase to improve readability, reduce complexity, and align the code with modern C++ best practices. A key focus is replacing raw pointers with smart pointers like `std::unique_ptr` and `std::shared_ptr`, which will automate memory management and reduce the risks of memory leaks and dangling pointers. Functions that return raw pointers will be updated to ensure proper ownership semantics.

Lambda expressions will be introduced to replace function pointers and verbose inline functions, simplifying the code and making it more concise, particularly in areas involving function parameters. Additionally, C-style casts will be replaced with C++ casts (`static_cast`, `dynamic_cast`, `const_cast`, and `reinterpret_cast`), enhancing type safety and making the code more robust against type-related errors.

3.4.4 Adoption of C++20/C++23 Features

The modernisation of the FERS software will integrate key C++20 and C++23 features to improve code clarity, performance, and maintainability. One of the core additions is `concepts`, which will enforce constraints on template parameters, resulting in clearer and safer template code, especially in areas that utilise generic programming. This will help the compiler catch errors early in the build process.

The `ranges` library will streamline operations on sequences, making the code more expressive while reducing errors related to manual iteration. Additionally, the implementation of `modules` to replace traditional header files where feasible will be evaluated during the modernisation process. This shift enhances the scalability and maintainability of the software as it evolves.

The modernisation process will also involve the integration of other key features introduced after C++98/03, as described in Table 2.2, which offer significant benefits in terms of code clarity, performance, and maintainability. Figure 3.4 below outlines key refactoring strategies alongside the new features being implemented and their benefits.

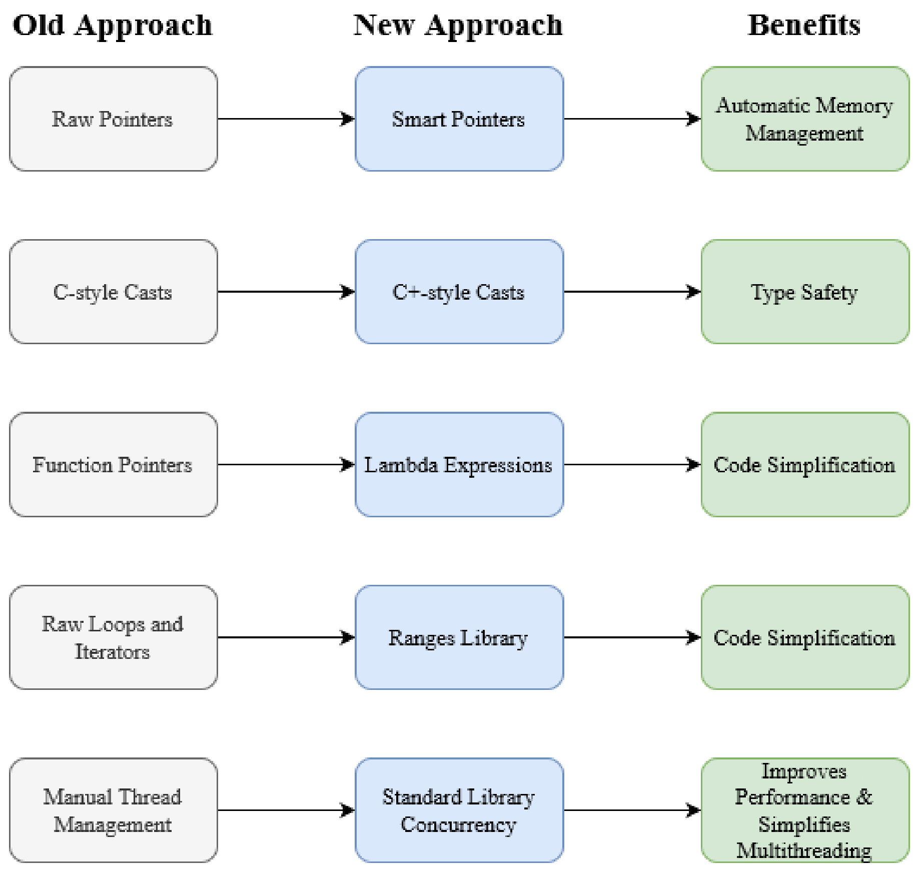


Figure 3.4: Transition from legacy C++ to modern C++20/23 features and benefits

3.4.5 Alternate Third-Party Libraries

In the modernisation of the FERS software package, several third-party libraries used in the original implementation were reviewed and evaluated for replacement. The goal was to identify modern alternatives that align with current C++ standards. A detailed comparison of the available options was conducted, followed by a selection of the most suitable libraries for each replacement.

Evaluation of Third-Party Libraries

For each existing library that was being used by FERS, potential alternatives were considered based on factors such as ease of use, performance, and feature set.

Boost FERS currently utilises the `Boost` library for various utilities, including smart pointers and thread management. Modern C++ (C++11 onwards) provides equivalent functionality in the standard library, such as `std::shared_ptr`, `std::thread`, and `std::async`. Transitioning to the standard library will reduce external dependencies, simplify the build process, and improve portability.

FFTW3 Although `FFTW3` is included in the current FERS codebase for performing Fourier transforms, it is not utilised. Thus, `FFTW3` should be removed, streamlining the codebase and eliminating unnecessary complexity.

HDF5 C API

- **HDF5 C++ API:** Provides an official C++ interface for HDF5, but its complexity and outdated design make it less suitable for modern C++ development.
- **HighFive:** A modern C++14 header-only wrapper that simplifies the interaction with HDF5 through better integration with C++ idioms (e.g., templates and smart pointers). `HighFive` offers improved usability and performance compared to the C++ API.

TinyXML

- **libxml2:** A robust, highly portable XML parser that supports schema validation and handles large XML files efficiently. It is widely available across Linux distributions.
- **PugiXML:** A fast and lightweight XML parser with an easy-to-use API, but it lacks support for schema validation.
- **RapidXML:** An extremely fast parser with minimal overhead, but similar to `PugiXML`, it lacks advanced features such as validation.
- **TinyXML2:** An improved version of `TinyXML` with better performance but still lacks validation and support for large XML files.

Proposed Replacements

Based on the comparison, the following changes to the third-party libraries in FERS are proposed. Table 3.4 below summarises the proposed changes:

Current Library	**Proposed Replacement**	**Rationale**
Boost	C++ Standard Library	Modern C++ includes most Boost features, reducing external dependencies and improving performance.
FFTW3	Removed	Unused in the codebase, simplifying the system by removing unnecessary components.
HDF5 C API	HighFive	Provides a modern C++14 header-only interface, improving ease of use while maintaining HDF5 performance.
TinyXML	libxml2 or libxml++	Offers better support for large XML files, schema validation, and overall robustness.

Table 3.4: Proposed replacement of third-party libraries

3.4.6 Performance optimisation

Performance optimisation is a key aspect of any modernisation process, aimed at enhancing the efficiency of the FERS radar signal simulations. To achieve this, multithreading will be updated using C++ standard threading libraries, allowing parallel execution of tasks such as signal processing and response rendering. This will lead to faster simulations by utilising available Central Processing Unit (CPU) cores more effectively.

Additionally, algorithmic optimisation will be applied where possible, improving existing implementations through more efficient data structures and leveraging modern C++ algorithms that offer built-in performance enhancements. These optimisations will ensure that FERS operates efficiently without compromising accuracy or reliability.

3.4.7 Modularity, Code Organisation, and Documentation

Improving the modularity and organisation of the FERS codebase is crucial for enhancing its maintainability and ensuring smoother future updates. To achieve this, namespaces will be more effectively utilised to encapsulate distinct parts of the software, minimising name conflicts and improving overall code clarity. The codebase will also be restructured into a modular architecture, with a clear separation between core functionalities, utilities, and tests. This reorganisation will include updating the directory layout and defining clear interfaces for each module to promote better modularity and scalability.

In addition, inline documentation and comments will be revised to align with the changes made during modernisation. The Doxygen [46] standard will be employed to facilitate automatic documentation generation, ensuring that the code is well-documented and easy to navigate. Finally, the organisation of code files will be adjusted based on their functionalities, with files renamed to accurately reflect their purpose, leading to a clearer and more intuitive structure.

3.4.8 Naming Conventions and Code Style

To ensure consistency and maintainability across the FERS codebase, a standard set of naming conventions will be enforced, referencing common C++ practices as outlined in Table 2.1. The code will follow the `snake_case` convention for variable names, ensuring readability, while class names will adopt `PascalCase`, as shown in the examples from Table 2.1. Functions and methods will use `camelCase` to maintain consistency across the project. Constants will be written in `UPPER_SNAKE_CASE`, following the convention for macro definitions and constant values. Private members or variables will adhere to the practice of using a leading underscore to differentiate them from regular variables.

By adhering to these naming conventions, the FERS codebase will become more organised and easier to navigate, supporting both current development efforts and future scalability.

3.5 Tools and Technologies

In the modernisation of the FERS software package, the selection of appropriate tools and technologies is critical to ensuring a smooth transition to modern C++ standards. This section details the development environment, refactoring tools, testing frameworks, and performance analysis utilities that will be utilised throughout the modernisation process.

3.5.1 Development and Testing Hardware

The performance and efficiency of the software development and testing processes are significantly influenced by the underlying hardware. Table 3.5 below details the hardware specifications of the machine used for the modernisation of the FERS software package:

Component	**Specification**	**Details**
CPU	AMD Ryzen 5 5600	6-core (12 threads), 3.5 GHz base, 4.4 GHz boost
Operating System	Ubuntu 24.04 LTS	Stable Linux environment optimised for development
Storage	1TB NVMe M.2 SSD	Fast read/write speeds for efficient builds and workflows
RAM	32GB DDR4-3200MHz	High-speed RAM for memory-intensive tasks

Table 3.5: Development and testing hardware specifications

This hardware setup ensures an efficient environment for modernising the FERS software.

3.5.2 Development Environment

The FERS modernisation will take place in a robust environment optimised for advanced C++ development and project management.

- **Integrated Development Environment (IDE):** CLion by JetBrains is the primary IDE due to its strong code navigation, refactoring, and CMake integration, essential for managing the FERS build system. It supports modern C++ standards, and offers code analysis, automatic formatting, and inspections, making it ideal for this project.
- **Compiler:** The project will use GNU Compiler Collection (GCC), which supports the latest C++ standards (C++20 and C++23), ensuring modern features and cross-platform compatibility.
- **Version Control System:** Git, hosted on GitHub, will manage the repository, with GitHub Actions automating testing and CI, supporting collaboration and version tracking.
- **Clang-Tidy:** Integrated with CLion, Clang-Tidy will assist with refactoring and static analysis, applying modern C++ transformations, and enforcing best practices like const-correctness and smart pointer usage.

3.5.3 Testing Framework

In the modernisation of the FERS software, regression testing will be employed as the primary method for ensuring the correctness and stability of the updated code. This approach focuses on validating the functionality of the entire system by executing a suite of test cases that simulate usage scenarios. Each test case verifies specific components and features of the software, ensuring that the changes made during the modernisation process do not introduce regressions or break existing functionality.

The regression tests will be designed to cover a broad range of simulation parameters and edge cases, ensuring comprehensive validation of the software's performance. GitHub Actions was integrated into the workflow for CI, automatically executing the regression test suite on each push. This ensures that the system remains stable throughout development and that potential issues are caught early. This testing strategy provides a reliable safeguard against errors, helping to maintain the integrity of the modernised code.

3.5.4 Profiling and Benchmarking Tools

Profiling and benchmarking are essential to ensure that the modernised FERS software performs optimally. To analyse and optimise performance, `valgrind` [44] will be utilised for both profiling and debugging. This tool provides comprehensive data on memory usage, including potential leaks and access violations, which is crucial for validating the shift from manual to automated memory management through smart pointers. By detecting memory-related issues that could impair performance or cause system failures, `valgrind` [44] ensures the reliability and efficiency of the modernised software.

Additionally, `perf` [45] will be employed to profile execution times across various functions within the FERS codebase. By identifying performance bottlenecks, this profiling tool will direct optimisation efforts, ensuring that the most critical sections of the code are running as efficiently as possible. Together, these tools will help maintain the balance between performance and functionality throughout the modernisation process.

3.6 Testing and Validation Methodology

A robust testing strategy ensures the modernised FERS software meets performance, functionality, and reliability standards. This section outlines the approach used to validate the system through regression tests and benchmarking.

Regression Testing

Regression tests ensure that existing features continue to function as expected. The test suite will be expanded to include all major changes and new features. Automated regression tests, managed through GitHub Actions, will run continuously, comparing the modernised system's behavior against the original baseline to prevent regressions.

Benchmarking

Benchmarking will assess the performance of the modernised software. Key metrics include execution time, memory usage, and accuracy. Profiling tools such as `valgrind` [44] and `perf` [45] will identify performance bottlenecks and verify that the transition to modern C++ features improves efficiency without sacrificing correctness. Results will be compared with the original system to confirm performance gains.

3.7 Design Specifications

The design specifications for modernising the FERS software are derived from Table 3.3 and the sections above. These design specifications ensure the modernisation meets performance, usability, and maintainability goals. Table 3.6 below links specific design specifications to their corresponding functional requirements, ensuring that each functional need is addressed.

Design Specification ID	**Description**	**Functional Requirement ID**
DS01	Refactor the FERS codebase to adopt modern C++20/23 features (smart pointers, lambda expressions) and improve code maintainability and performance through multithreading and modularity.	**FR01**, **FR02**, **FR05**
DS02	Ensure backward compatibility while modernising data handling with libraries like HighFive and libxml2, integrating profiling tools for performance analysis.	**FR03**, **FR04**
DS03	Provide comprehensive Doxygen-generated documentation for easy maintainability and usability.	**FR05**

Table 3.6: Design specifications and functional requirements mapping

3.7.1 Target Code Quality Metrics

Table 3.7 below summarises key metrics related to code quality and technical debt in the current FERS system.

Metric	**Current Value**	**Target After modernisation**	**Description**
Cyclomatic complexity	1-15	≤ 10	Reduced complexity for easier testability
Comment-to-Code Ratio	43.7%	60%	Improved usage of comments and docstrings
Code Quality Warnings	3,243	0-10 per 1000 lines of code	Fewer warnings after applying best practices
Technical Debt Rating	C	A	Improved maintainability with modern techniques

Table 3.7: Target code quality after modernisation

3.7.2 Acceptance Test Procedures

The Acceptance Test Procedure (ATP)s ensure that the modernised FERS software meets all performance, compatibility, and functionality requirements. Table 3.8 below outlines the specific tests that will be performed during the validation process.

ATP	Description
ATP-01	**Regression and Backward Compatibility Testing**: Run original FERS test cases on the modernised system, validating the outputs and ensuring compatibility with legacy input/output formats.
ATP-02	**Performance and Memory Testing**: Benchmark radar simulations for execution time and memory usage, ensuring multithreading efficiency, no memory leaks, and performance improvement over the legacy system.
ATP-03	**Error Handling and Robustness Testing**: Validate error handling with corrupted inputs and edge cases, and stress-test the system for stability under heavy loads.
ATP-04	**Documentation Usability Testing**: Ensure that Doxygen documentation is complete and understandable for users and developers.

Table 3.8: Acceptance test procedures

These ATPs ensure that the FERS modernisation is rigorously tested for functionality, performance, and compatibility, aligning with the project's objectives.

3.8 Conclusion

The modernisation strategy and design outlined in this chapter provides a comprehensive approach to updating the FERS software package. Through a detailed analysis of the existing system, key areas like code refactoring, multithreading optimisation, and the integration of modern C++20/23 features were prioritised to ensure the software meets modern standards for performance and maintainability. Each design choice—such as adopting smart pointers for memory management and restructuring external dependencies—aligns with the objectives of enhancing efficiency, maintaining backward compatibility, and improving system robustness.

The incremental methodology minimises regression risks while validating performance improvements through rigorous testing and benchmarking. By addressing technical debt and implementing best practices, this modernisation project ensures FERS remains a high-performance, reliable, and scalable tool for radar simulation research.

Chapter 4

Implementation

The future cannot be predicted, but futures can be invented.

—*Dennis Gabor*

The modernisation of the FERS software package was conducted through a structured refactoring process, aimed at enhancing code maintainability, performance, and compliance with modern C++ standards. This process was divided into five primary phases, each focusing on different aspects of the legacy codebase, such as memory management, modularity, performance optimisation, and the integration of C++20/23 features. The following sections outline the key refactoring actions taken during each phase, along with notable challenges encountered and their corresponding solutions.

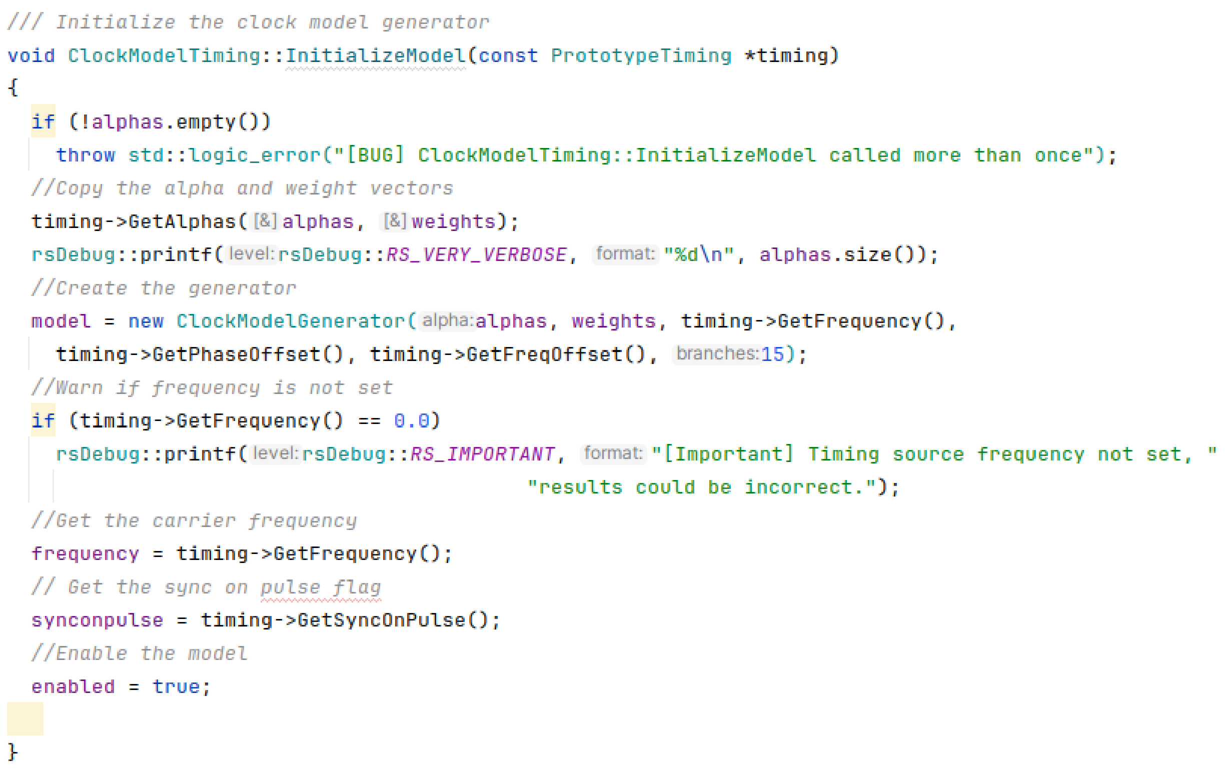
4.1 Refactoring Overview

The refactoring process was implemented progressively, focusing on code quality improvements, modern C++ feature integration, and performance enhancements. Each phase was carried out at logical intervals, with clear objectives and improvements consolidated into pull requests, as detailed below.

4.1.1 Styling

One of the most immediately noticeable improvements in the modernisation process was the enhancement of the code's overall styling. These changes included enforcing consistent naming conventions, standardising indentation, and restructuring the code for clarity and readability across all files. Consistent styling not only improves the visual structure but also helps maintainability and reduces the cognitive load for future developers who will work with the code.

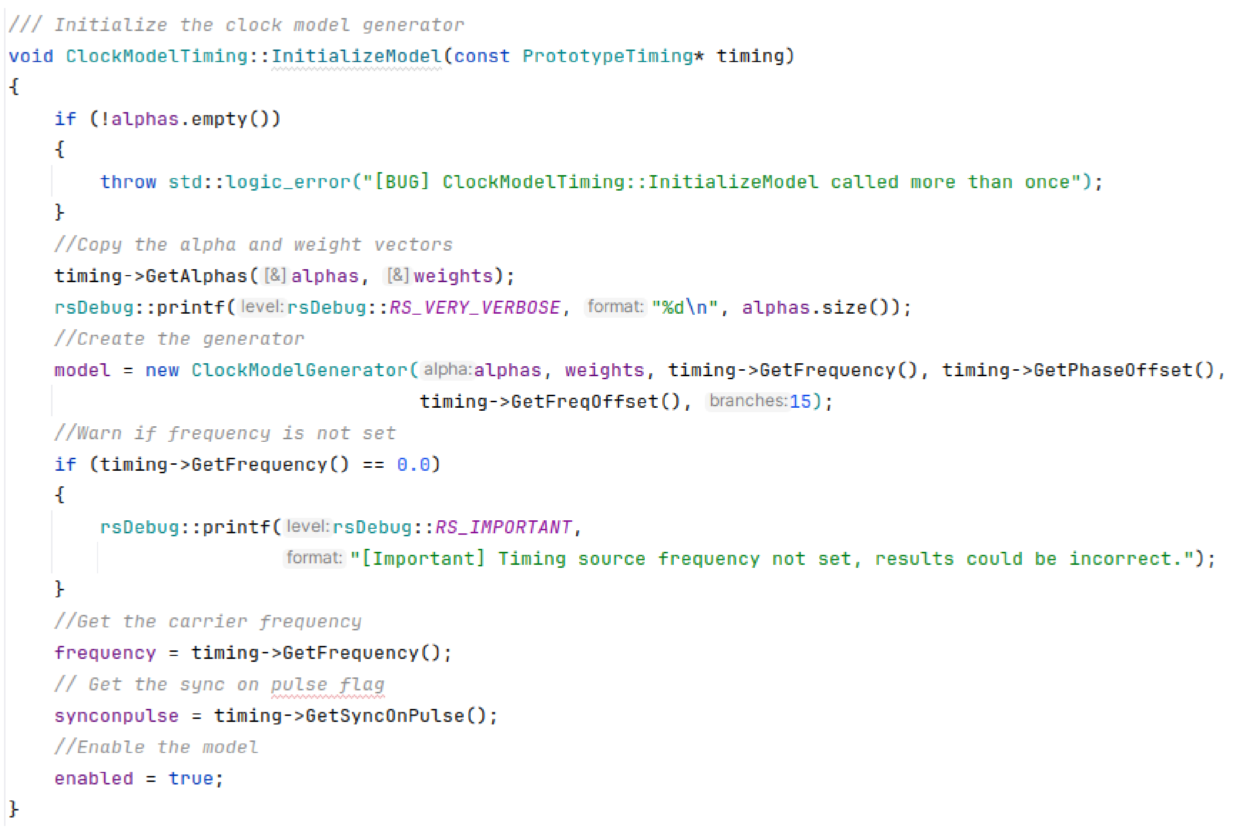
Figure 4.1 below illustrates a side-by-side comparison of the code before and after applying these styling improvements.



```
/// Initialize the clock model generator
void ClockModelTiming::InitializeModel(const PrototypeTiming *timing)
{
  if (!alphas.empty())
    throw std::logic_error("[BUG] ClockModelTiming::InitializeModel called more than once");
  //Copy the alpha and weight vectors
  timing->GetAlphas( [&] alphas,  [&] weights);
  rsDebug::printf( level: rsDebug::RS_VERY_VERBOSE,  format: "%d\n", alphas.size());
  //Create the generator
  model = new ClockModelGenerator( alpha: alphas, weights, timing->GetFrequency(),
    timing->GetPhaseOffset(), timing->GetFreqOffset(),  branches: 15);
  //Warn if frequency is not set
  if (timing->GetFrequency() == 0.0)
    rsDebug::printf( level: rsDebug::RS_IMPORTANT,  format: "[Important] Timing source frequency not set, "
                                                "results could be incorrect.");
  //Get the carrier frequency
  frequency = timing->GetFrequency();
  // Get the sync on pulse flag
  synconpulse = timing->GetSyncOnPulse();
  //Enable the model
  enabled = true;

}
```

(a) Code style before



```
/// Initialize the clock model generator
void ClockModelTiming::InitializeModel(const PrototypeTiming* timing)
{
    if (!alphas.empty())
    {
        throw std::logic_error("[BUG] ClockModelTiming::InitializeModel called more than once");
    }
    //Copy the alpha and weight vectors
    timing->GetAlphas( [&] alphas,  [&] weights);
    rsDebug::printf( level: rsDebug::RS_VERY_VERBOSE,  format: "%d\n", alphas.size());
    //Create the generator
    model = new ClockModelGenerator( alpha: alphas, weights, timing->GetFrequency(), timing->GetPhaseOffset(),
                                    timing->GetFreqOffset(),  branches: 15);
    //Warn if frequency is not set
    if (timing->GetFrequency() == 0.0)
    {
        rsDebug::printf( level: rsDebug::RS_IMPORTANT,
                         format: "[Important] Timing source frequency not set, results could be incorrect.");
    }
    //Get the carrier frequency
    frequency = timing->GetFrequency();
    // Get the sync on pulse flag
    synconpulse = timing->GetSyncOnPulse();
    //Enable the model
    enabled = true;
}
```

(b) Code style after

Figure 4.1: Comparison of code style before and after

The adoption of these styling improvements results in more readable, maintainable, and professional code, reducing the likelihood of errors and facilitating smoother future updates.

4.1.2 General Refactor

After completing the styling improvements, the focus shifted toward enhancing the overall quality of the codebase by integrating modern C++ features and improving both readability and safety. One of the primary changes involved reinforcing type safety. Constructors in key classes, such as `SVec3` and `Vec3`, were explicitly marked to prevent implicit conversions, thus reducing the potential for unintended errors. Additionally, the outdated `typedef` declarations were replaced with modern `using` type aliases, ensuring compliance with contemporary C++ conventions.

Following these changes, a thorough code clean-up was performed to streamline the codebase. Redundant and unused methods, such as `Signal::pad()`, were removed, resulting in a more efficient and maintainable code structure. Moreover, manual memory management practices were modernised through the introduction of `std::unique_ptr` and `std::shared_ptr`, which facilitated automatic and safe memory handling, eliminating the need for manual oversight.

Alongside these improvements, const-correctness was enforced more rigorously. Variables were marked as `const` where appropriate, and their scope was confined to inner blocks, enhancing both performance and security by ensuring immutability and better resource management. This refinement was complemented by the adoption of modern C++ features, such as structured bindings and type inference with `auto`, which improved code clarity and efficiency. Additionally, C-style casts were replaced with C++ casts, and range-based `for` loops were employed, further enhancing code readability.

Finally, a series of bug fixes were applied to address critical issues. Problems related to integer division, floating-point precision, and uninitialised variables were resolved. Furthermore, memory leaks and warnings about narrowing conversions were addressed, ensuring the code adhered to modern C++ standards. These efforts culminated in a codebase that is not only more efficient and safer but also better aligned with contemporary development practices.

Listing 4.1 below provides examples for some of these changes.

```cpp
// typedef vs. using
typedef std::complex<rsFloat> rsComplex;
using RsComplex = std::complex<RS_FLOAT>;

// Memory management
std::vector<MultirateGenerator *> generators;
std::vector<std::unique_ptr<MultirateGenerator>> _generators;

// C-style casts vs. C++-style casts
double x2 = x1+1.0/(double)(size_azi);
const double x2 = x1 + 1.0 / static_cast<double>(_size_azi);

// For-loops
```

```
std::vector<InterpPoint>::iterator i;
for (i = points.begin(); i != points.end(); i++) { RenderResponseXML(element, *i); }

for (auto & point : _points) { renderResponseXml(element.get(), point); }
```

Listing 4.1: General refactor changes

4.2 Modularity Enhancements

After applying general refactoring changes to the codebase, the next logical step was to improve the modularity of the codebase. This step primarily involved organising the code into logically separated directories and improving the encapsulation of the code to adhere to object-oriented design best practices.

4.2.1 Object-Oriented Design

Previously all the code for FERS was located in the `src/` directory at the same depth. This can lead to confusion and difficulty in finding related code. To improve the design, the codebase was restructured into the following directories as shown in Figure 4.2 below, as well as files being renamed to accurately communicate their purpose (e.g., `rspulserender.h` → `receiver_export.h`).

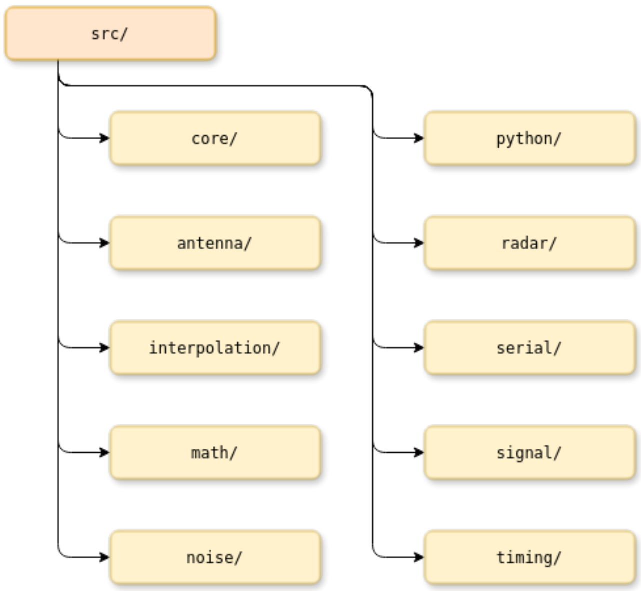


Figure 4.2: Improved directory structure of the codebase

Along with the restructuring of the codebase, the code in each directory was assigned its own namespace. For example, the code in the `serial/` directory was assigned the `serial` namespace. This will allow any future developer to easily identify where a function or class being used somewhere in the codebase is located. Also, the improvement of namespace usage ensures that if two namespaces contain classes or functions with the same name, there will not be a conflict. However, at the time of writing, there are no functions, classes, or methods that have the same name.

4.2.2 Simplification and Amalgamation of Code

It was identified that there was room for improvement in the location of many methods and classes. Not only to improve the organisation, but also to improve the simplicity of the codebase, many portions of code were moved to related files, or placed in entirely new files.

Along with the changes made above, many simple methods for classes were inlined to potentially improve the performance of FERS, but mostly to improve the brevity of the codebase. These inlined methods are generally simple getter or setter methods, however, in cases where the method implementation is simple enough, these methods were also inlined. Listings 4.2 and 4.3 below shows examples of this.

```
bool compareTimes(const Response* a, const Response* b)
{
  return a->startTime() < b->startTime();
}

RS_FLOAT RsParameters::c()
{
  if (!_instance)
    {
    _instance = new RsParameters;
  }
  return sim_parms.c;
}
```

Listing 4.2: Method inlining example in original FERS

```
inline bool compareTimes(const std::unique_ptr<Response>& a, const std::unique_ptr<Response
    ↪ >& b)
{
    return a->startTime() < b->startTime();
}

inline RS_FLOAT c() { return params.c; }
```

Listing 4.3: Method inlining example in modernised FERS

4.2.3 Cyclomatic Complexity Reduction

The last major change that was done for this phase of the refactoring was the simplification of the `runThreadedSim` and `processDocument` methods to improve their maintainability and clarity. Listings 4.4 and 4.5 below shows simplified snippets of before and after this change to `runThreadedSim`

```
void runThreadedSim(unsigned threadLimit, World* world) {
  // PHASE 1: Do first pass of simulator
  for (/* Loop through receivers */) {
    for (/* Loop through transmitters */) {
      while (threads >= threadLimit) { /* wait until a thread is available */ }
    }
  }

  while (threads) { /* wait until all threads finish */ }

  // PHASE 2: Do render pass of simulation
  for (/* Loop through receivers */) {
    while (threads >= threadLimit) { /* wait until a thread is available */ }
  }

  while (threads) { /* wait until all threads finish */ }
}
```

Listing 4.4: Example of reducing cyclomatic complexity (pre-modernisation)

```
void runThreadedSim(const unsigned threadLimit, const World* world) {
    std::vector<std::unique_ptr<boost::thread>> running;

    // Get receivers from the world
    const std::vector<Receiver*> receivers = world->getReceivers();

    // Run simulation for receiver-transmitter pairs
    runSimForReceiverTransmitterPairs(threadLimit, world, receivers, running);

    // Run rendering for receivers
    runRenderThreads(threadLimit, receivers, running);
}
```

Listing 4.5: Example of reducing cyclomatic complexity (post-modernisation)

Due to the scale of the changes made to the XML parsing as discussed in Section 4.4.3 below, the changes made to `processDocument` are excluded from this section, however, the changes made can be found in Listing A.1 in Appendix A.

4.3 Integration of C++20/23 Features

After making the code more readable and organising the code into a logical structure, the next logical step was to proceed to tackle the main task of the project, updating the codebase to the most modern C++ features available. This phase primarily involved integrating smart pointers where possible, adopting the latest standard library methods, and improving the clarity of the code.

4.3.1 Memory Management

The most notable change that can be observed during this phase is the transition away from raw pointers to smart pointers. The usage of smart pointers facilitated the removal of many custom class destructors which were prone to memory leaks due to the manual management of memory. The most significant change made here was for the `World` class. Listings 4.6 and 4.7 below shows simplified snippets of before and after for this change.

```cpp
class World {
public:
    // Custom Destructor
    ~World() {
      for (auto& [_, snd] : _pulses) { delete snd; }
      for (auto& [_, snd] : _antennas) { delete snd; }
      for (auto& [_, snd] : _timings) { delete snd; }

      std::for_each(_receivers.begin(), _receivers.end(), ObjDel<Receiver*>());
      std::for_each(_transmitters.begin(), _transmitters.end(), ObjDel<Transmitter*>());
      std::for_each(_targets.begin(), _targets.end(), ObjDel<Target*>());
      std::for_each(_platforms.begin(), _platforms.end(), ObjDel<Platform*>());
    }
private:
    // Private variable declarations for objects in the World removed for brevity. All
    ↪ variables are collections of raw pointers.
};
```

Listing 4.6: `World` class prior to modernisation

```cpp
class World {
public:
    // We use the default destructor since we do not need to manually free the memory used
    ↪ by the class
    ~World() = default;
private:
    // Private variable declarations for objects in the World removed for brevity. All
    ↪ variables moved from collections of raw pointers to smart pointers.
};
```

Listing 4.7: `World` class after modernisation

4.3.2 Standard Library Implementations

Modern C++ standard library constructs were integrated where applicable to improve the performance and reduce the complexity of the code. This involved replacing manual implementations of algorithms with the C++ standard library's `<algorithm>` methods as well as replacing immutable references to vectors (e.g., `const std::vector& vec`) with `std::span`.

Among the changes made here, the most common `<algorithm>` replacements were `std::ranges::for_each` and `std::ranges::copy` among many others. These standard library algorithms are relatively simple to understand and are just meant to replace instances of code such as the snippets show in Listings 4.8 and 4.9:

```cpp
for (unsigned int i = 0; i < samples; i++) { _data[i] = inData[i]; }
// AND
for (iter = data.begin(); iter != data.end(); iter++) { (*iter).second /= a; }
```

Listing 4.8: Example of C++98/03 manual algorithm implementation

with:

```cpp
std::ranges::copy(inData, _data.begin())
// AND
std::ranges::for_each(_data | std::views::values, [a](auto& value) { value /= a; });
```

Listing 4.9: Example of C++20/23 algorithm implementation using `std::ranges`

This helps with the clarity of the code, but mostly impacts code safety because it reduces the chance of mistakes being made when manually creating such loops.

4.3.3 Bug Fixes and Memory Leaks

Along with the changes mentioned above, some notable issues in the code were fixed. The fixes that were implemented are as follows:

- **Multipath Surfaces:** A segmentation fault (SIGSEGV) occurred when multipath surfaces were defined in the `.fersxml` due to the `World::processMultipath` method modifying vectors while iterating over them. The fix involved reserving enough space (i.e., $2x$ the original size) in the vectors before iterating, preventing memory reallocation issues. This allows the method to append new multipath duals to the vectors without disrupting the iteration.

- **Invalid HDF5 Read:** In the `rshdf5::ReadPattern` method (which was used to read antenna gain patterns from an HDF5 file) the pattern was being read in with the wrong dimensions. The old code used `i * aziSize + j`, which caused incorrect data mapping. The new code changes this to `i * elevSize + j`, ensuring the data is stored correctly. This properly aligns with how HDF5 files are internally structured.

- **Memory Leaks:** There were two memory leaks that could be fixed. The first memory leak was due to the `rsFloat **pattern` array of raw pointers. This leak was easily fixed by moving to a 2D vector. The second memory leak was due to the global `InterpFilter` class instance not being deallocated upon exit. Moving the instance to a `std::unique_ptr` fixed the issue.

4.4 Alternative Libraries and Dependencies

This phase of the modernisation process focused on replacing legacy libraries with more modern, efficient, and feature-complete alternatives. This decision was driven by the need to align the FERS codebase with contemporary C++ practices, reduce external dependencies, and improve maintainability. The libraries replaced in this phase included HDF5 C API, `TinyXML`, `Boost`, and FFTW, as detailed below.

4.4.1 Boost and FFTW

`Boost` was a very simple library to replace as it was mostly used for multithreading operations and marking classes as non-copyable. Instances of `boost::thread` were replaced with `std::thread`, and instead of `boost::noncopyable`, the copy constructor and copy assignment operator for such classes were deleted using similar implementations of the code shown in Listing 4.10 below:

```cpp
class Response {
public:
    Response(const Response&) = delete;
    Response& operator=(const Response&) = delete;
}
```

Listing 4.10: Example of replacing `boost::noncopyable` with deleted operators

Deleting these special member functions prevents any code from copying instances of the class.

The FFTW library was only used in the `fftwcpp/` directory and had no linkages to any other code in FERS. Therefore, FFTW could be removed in its entirety from the codebase without causing any issues.

4.4.2 HighFive Instead of HDF5 C API

The legacy code relied heavily on the HDF5 C API for handling large datasets, particularly for storing radar simulation results. While functional, the C API was verbose, error-prone, and lacked the modern C++ constructs that could simplify memory management and improve code safety. To address these issues, the `HighFive` library was introduced as a modern C++14 header-only wrapper around the HDF5 library. Listing 4.11 below illustrates how the usage of `HighFive` improves the code simplicity.

```cpp
const hid_t file_id = H5Fopen(name.c_str(), H5F_ACC_RDONLY, H5P_DEFAULT);

std::vector ret(aziSize, std::vector<RealType>(elevSize));
for (unsigned i = 0; i < aziSize; ++i) {
    for (unsigned j = 0; j < elevSize; ++j) { ret[i][j] = data[i * elevSize + j]; }
}
// Becomes:
const HighFive::File file(name, HighFive::File::ReadOnly);

```

```cpp
std::vector data(aziSize, std::vector<RealType>(elevSize));
dataset.read(data);
```

Listing 4.11: Example code of HighFive vs HDF5 C API

As you can see, `HighFive` drastically reduces the complexity of the code and enhances the readability. This will help with future development efforts that involve working with HDF5.

4.4.3 libxml2 Instead of TinyXML

The FERS software initially used `TinyXML` for parsing and managing XML files. However, `TinyXML` lacked advanced features such as schema validation and support for large XML documents, which are essential for validating the complex input files used in radar simulations. Additionally, the separate `config_validators` directory, which used `Xerces-C++` for validating the input `.fersxml` files, created unnecessary fragmentation in the codebase. The decision was made to integrate validation directly into FERS, using `libxml2` as the primary XML library.

To facilitate this change, a wrapper for `libxml2` was created which provided ease of use and automatic memory management for all XML operations. With this new wrapper, all XML related code that previously used `TinyXML` was rewritten and simplified as much as possible.

As well as rewriting the XML related code, the XML schemas in the codebase (`fers-xml.dtd` and `fers-xml.xsd`) were updated to accurately reflect what FERS expects when parsing the code in `xml_parser.cpp`. This modification allowed XML schema validation to be integrated directly into FERS so that when the program is run the input `.fersxml` file will be validated against the XML schemas to ensure that there are no mistakes in the input file. To enable the validation a new program argument (`-val` or `-validate`) was added to the argument parsing code which will indicate to FERS that it should also validate the input file and then run the simulation if there are no issues in the XML.

Due to the scale of this change, attempting to provide examples in this report is not feasible, however, to view the XML related changes, please refer to the GitHub repository in Appendix A.

4.5 Performance optimisation

The fifth phase of the refactoring process primarily involved rewriting computationally intensive portions of code in the program. It was identified through profiling the execution of FERS with `perf` [45] that the most computationally intensive task of FERS was the rendering of responses in `exportReceiverBinary`.

FERS uses multithreading to potentially improve performance in two areas: simulating signal propagation and rendering the responses received by the receivers in the simulation world. The latter task—rendering the receiver responses—requires the most computational power. To handle this, each receiver is assigned its thread, operating independently to process the responses. Within each thread, responses are first sorted based on the timestamp when they were received. These sorted responses are then passed to the functions which render the output files of the simulation. The `exportReceiverBinary` function is the most computationally complex. In this function, several processes take place for each receiver's response window: noise is calculated, receiver windows are rendered, and the data is downsampled and quantised.

Multithreading is then used to render the response windows. The number of threads that can be created is limited by the system's available threads. However, since this potentially takes place within a child thread, this can lead to a problem known as 'thread explosion,' which often happens because the parallelism being used is nested. Nested parallelism occurs when threads created at a higher level can spawn additional threads deeper in the code, potentially overwhelming system resources. In this case, the number of threads that could potentially be created—especially for complex, large-scale simulations—can be calculated as shown in Equation 4.1:

$$\text{num\_threads} = \min(\text{num\_recv}, \text{num\_cores}) \times \min(\text{num\_responses}, \text{num\_cores}) \tag{4.1}$$

This means that, in a large simulation, the thread count can quickly become very high. For example, on a 12-core system, up to 144 threads might be created. Such a large number of threads can lead to significant performance degradation due to excessive context switching and the overhead associated with managing all the threads.

To combat this issue, a global thread pool was created and is used by all multi-threaded operations in the program. This thread pool maintains a queue of tasks to be completed and ensures that the number of threads created will never exceed the number of threads available on the system. See Listing 4.12 below for a simplified snippet of this thread pool.

```cpp
class ThreadPool {
public:
    // Constructor to initialise the thread pool with a given number of threads
    explicit ThreadPool(const unsigned numThreads);

    // Destructor to join all worker threads and stop the pool
    ~ThreadPool();

    // Enqueue a task in the thread pool and return a future result
    template <class F, class... Args>
    std::future<std::invoke_result_t<F, Args...>> enqueue(F&& f, Args&&... args);

    // Wait for all tasks to complete
    void wait();

    // Get the number of available threads (not processing tasks)
    unsigned getAvailableThreads();

private:
    std::vector<std::thread> _workers; ///< Vector of worker threads.
    std::queue<Task> _tasks; ///< Queue of tasks to be executed.
    std::mutex _queue_mutex; ///< Mutex for synchronising access to the task queue.
    std::condition_variable _condition; ///< Condition variable for task notification.
    std::condition_variable _done_condition; ///< Condition variable for task completion.
    std::atomic<bool> _stop = false; ///< Flag indicating whether the thread pool is stopped
    std::atomic<unsigned> _pending_tasks = 0; ///< Count of pending tasks.
};
```

Listing 4.12: Simplified `ThreadPool` class

To further improve the performance of FERS, adaptive threading was implemented for the rendering of responses. This adaptive threading has a simple approach. When the number of responses for a receiver window is smaller than a threshold (e.g., < 8 responses) then the computation will be sequential. However, if there are more responses than this threshold and the number of available threads is more than 1, then the computation will be parallelised. This approach ensures that the program is fully utilising all cores without causing excessive threading overhead. See Listing 4.13 below for a simplified snippet of this approach.

```cpp
void renderWindow(/* function parameters */) {
    // Retrieve responses within the window and store them in the queue
    std::queue<Response*> work_list;

    if (num_responses < 8 || available_threads <= 1) {
        // SEQUENTIAL PROCESSING
        // Retrieve item from queue and perform the rendering of the response until queue is
    ↪   empty
    } else {
        // PARALLEL PROCESSING
        std::mutex work_list_mutex;
        auto worker = [&] {/* worker code */};

        for (unsigned i = 0; i < num_threads; ++i) { /* enqueue workers in the ThreadPool */
    ↪   }

        // Wait for all threads to finish
        ...
    }
}
```

Listing 4.13: Simplified `renderWindow` function

In summary, FERS will now prioritise parallelising the rendering of receivers, and only parallelise the rendering of individual windows of responses if certain conditions are met.

4.6 Testing Framework and Documentation

The final phase of the refactoring process involved expanding the regression testing framework and improving the documentation of the FERS software.

4.6.1 Testing Framework

To achieve a very high code coverage with the regression tests, `lcov` and `gcov` were used in tandem to analyse the existing regression testing developed throughout the modernisation process and identify gaps in the framework. Most notably, regression tests for noise simulation, oversampling, and various simulation parameters were added to the framework. After these test cases were added, there was a total of 23 regression tests available. Due to the cumbersome size of the testing framework, it was then decided that the testing framework should be simplified, combining test cases where possible.

After this process was completed, the following regression testing suite was established, as shown in Table 4.1 below.

Test Name	**Pulses**	**Antennas**	**Transmitters & Receivers**	**Targets**	**Other**
test1	Continuous wave	Parabolic	Stationary monostatic	Moving isotropic target	N/A
test2	Pulsed	HDF5-defined antenna	Stationary monostatic	Stationary XML-defined target	Clock drift
test3	Pulsed	XML-defined antenna	Stationary monostatic	Stationary isotropic target	N/A
test4	Pulsed	Sinc & Square horn antenna	2 stationary monostatic	2 isotropic targets with rotation and movement	Includes another `.fersxml`
test5	Pulsed	Python antenna	Python-defined motion monostatic	Stationary isotropic target	N/A
test6	Pulsed	Isotropic antenna	Rotating transmitter & Rotating receiver	Rotating isotropic target	Noisy timing & Oversampling
test7	Pulsed	Gaussian antenna	Rotating monostatic	Stationary isotropic target	Multipath surface
test8	Pulsed	Gaussian antenna	Stationary noisy monostatic	Stationary isotropic target	Noisy timing
test9	Pulsed	Isotropic antenna	3 stationary transmitters & receivers	2 moving and rotating isotropic targets	N/A

Table 4.1: Table of definitions for regression tests in the `sim_tests` directory

4.6.2 Documentation

Comprehensive documentation was created for every function, method, class, and file in the refactored codebase, using the Doxygen standard for docstrings. An example of this documentation for the `main.cpp` file is shown in Listing 4.14.

```cpp
/**
* @file main.cpp
* @brief Entry point and main logic for the FERS simulation application.
*
* This file contains the main function that initialises and runs the FERS (Flexible
    ↪ Extensible Radar Simulator) simulation. It handles command-line argument parsing,
    ↪ logging configuration, simulation initialisation, and execution using multithreading
    ↪ .
*
* @authors David Young, Marc Brooker
* @date 2006-04-25
*/

...

/**
 * @brief Entry point for the FERS simulation application.
 *
 * This function initialises the simulation environment, parses command-line arguments, sets
    ↪  up logging based on user input, and runs the simulation using a multithreaded
    ↪ approach. It manages the execution of the simulation and handles errors encountered
    ↪ during the initialisation or simulation stages.
 *
 * @param argc The number of command-line arguments passed to the program.
 * @param argv The array of command-line arguments passed to the program.
 * @return int Returns 0 on successful simulation execution, or 1 on error.
 */
int main(const int argc, char* argv[]) { ... }
```

Listing 4.14: Example of Doxygen documentation standard in `main.cpp`

With this updated documentation for FERS, developers can quickly understand how the code works and easily navigate to the relevant files in their IDE. They can also generate the Doxygen documentation by running '`doxygen Doxyfile`', which will produce a locally available website where the documentation is presented in a user-friendly format.

4.7 Challenges and Lessons Learned

Throughout the refactoring process, several challenges emerged, particularly related to maintaining backward compatibility, understanding the legacy codebase, and optimising performance. These challenges, along with the solutions implemented, are outlined below:

- **Backward Compatibility:** Ensuring that the modernised code remained compatible with the original system was a critical challenge. Changes were introduced incrementally, with extensive regression testing employed at each step to ensure that the functionality of the original system was preserved. This included a thorough evaluation of the simulation outputs to ensure accuracy.
- **Legacy Code Understanding:** The original codebase lacked sufficient documentation, making it difficult to discern the intent behind certain implementations. This was mitigated by leveraging code inspection tools, along with manual code reviews, to identify problematic sections of the code. Incremental refactoring and testing ensured that functional integrity was maintained throughout the process.
- **Performance optimisation:** Introducing modern C++ features initially introduced minor performance regressions due to the abstraction overhead. This challenge was addressed through performance profiling, followed by targeted optimisations in performance-critical sections, particularly in signal processing and simulation execution.

4.8 Conclusion

The modernisation of the FERS software was achieved through a methodical refactoring process, focusing on code maintainability, performance improvements, and adherence to modern C++ standards. The implementation phase was divided into distinct steps, addressing critical aspects such as memory management, modularity, performance optimisation, and the integration of C++20/23 features. Key improvements included the replacement of manual memory management with smart pointers, enhanced code readability through the adoption of modern C++ constructs, and the restructuring of the codebase into a more modular and object-oriented design. Additionally, significant improvements were achieved through multithreading and the introduction of adaptive threading strategies, which will substantially improve the efficiency of computationally intensive tasks.

Another major accomplishment during the implementation was the replacement of legacy libraries, such as `Boost` and `TinyXML`, with more modern alternatives like `HighFive` and `libxml2`, which enhanced both the performance and maintainability of the code. The thorough integration of these new libraries streamlined several processes, reducing complexity and improving code safety. Furthermore, a comprehensive testing framework was established, which will have higher code coverage and ensuring that the modernisation efforts maintained backward compatibility and system integrity.

This chapter has outlined the steps taken to refactor and modernise the FERS software, addressing challenges and providing solutions for maintaining backward compatibility, improving code clarity, and optimising performance. The next chapter will delve into the detailed analysis of the performance gains, functionality improvements, and overall impact of these changes on the system, providing a quantitative assessment of the modernisation effort.

Chapter 5

Results & Analysis

The goal is to turn data into information, and information into insight.
—*Carly Fiorina*

In this chapter, the results of the modernisation efforts applied to the FERS software package are presented and analysed. The primary objective of this chapter is to evaluate how well the updated software meets its original functionality, assess performance improvements, and highlight key enhancements. To achieve this, extensive testing was conducted, including functionality validation through regression tests, performance benchmarking, and analysis of resource utilisation such as CPU and memory. Additionally, various aspects of the software's code quality, including Cyclomatic complexity and technical debt, were examined to assess the maintainability and efficiency of the updated codebase. This chapter provides a comprehensive comparison between the original and modernised versions of FERS, demonstrating the effectiveness of the modernisation process while identifying areas for future improvement.

5.1 Methodology

The results presented in this chapter were obtained through a structured testing process designed to evaluate the performance, functionality, and memory management improvements of the modernised FERS software package. The testing environment consisted of the system detailed in Table 3.5, which features an AMD Ryzen 5 5600 processor, 32GB of RAM, and an NVMe Solid-State Drive (SSD) running Ubuntu 24.04 Long-Term Support (LTS).

5.1.1 Performance Testing

Performance metrics were measured using both automated and manual methods. The `perf` tool was employed to profile CPU usage, multithreading efficiency, and task execution times. Key functions, such as signal simulation and file exports (XML, Comma-Separated Values (CSV), HDF5), were timed manually to assess overall speedup and total simulation time across various test cases. Performance benchmarks were conducted under identical conditions for both the original and modernised versions of FERS, allowing for direct comparison.

5.1.2 Memory Leak Detection

Memory management was evaluated using `valgrind` to detect memory leaks and inefficiencies. The tool identified ‘definitely lost’ and ‘still reachable’ memory blocks in both versions of the software. Results were compared to quantify the improvements in memory handling introduced by modern C++ techniques, such as smart pointers.

5.1.3 Code Quality Metrics

Code quality improvements were measured using tools such as `Understand` and `CLion` for static analysis. Metrics like Cyclomatic complexity and technical debt were used to evaluate the maintainability and complexity of the code. The modernisation reduced Cyclomatic complexity and eliminated key sources of technical debt, as described in the results.

5.1.4 Compilation Time Measurement

Compilation times were manually measured using clean builds for both versions of FERS. The tests were run across different CPU core configurations (1 to 12 cores) to assess the impact of refactoring and modern C++ features on build performance.

5.1.5 Test Cases

For all testing in this chapter (except Sections 5.2 and 5.4.1 which used the regression test suite), the following test cases in Table 5.1 were used:

Test Name	**Transmitters**	**Receivers**	**Targets**	**Sampling Rate (MHz)**	**PRF**	**Time (s)**
Very Simple	1 monostatic		1	1	20	1
Simple	1 monostatic		1	5	265	5
Moderate	1	2	2	10	510	10
Complex	2	4	5	50	755	30
Very Complex	3	6	10	100	250	60

Table 5.1: Table of definitions for the test cases used for results

Note: In all cases above, the only moving objects are the targets in the simulation and all transmitters and receivers are stationary.

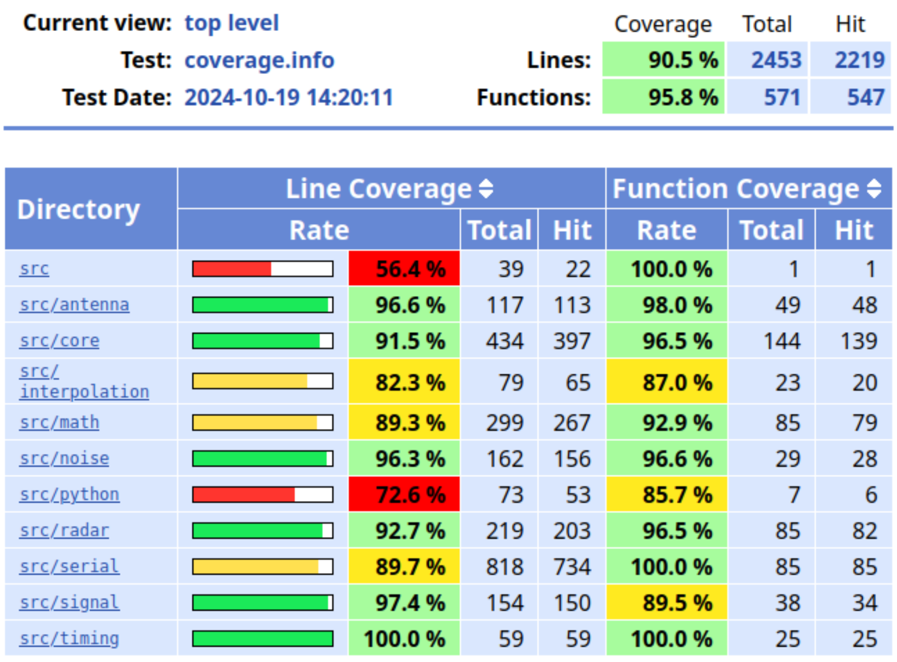
With these test cases a comprehensive understanding of the performance of FERS can be gained through thorough testing as detailed in the following sections.

5.2 Functionality Testing

This section evaluates whether the modernised FERS software maintains its core capabilities and performs as expected after incorporating updates. This involves rigorous testing, including regression tests, to ensure the software continues to operate reliably and produces accurate outputs.

5.2.1 Regression Testing

With the updated regression tests described in Section 4.6.1, the overall coverage rates achieved were 90.5% of lines and 95.8% of functions. This can be seen in Figure 5.1 below.

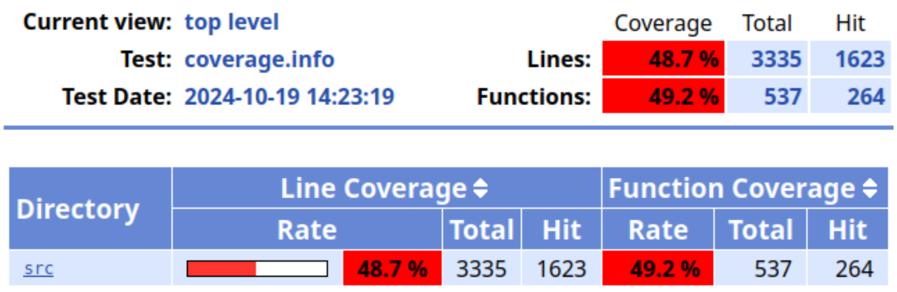


			Coverage	Total	Hit
Current view:	top level				
Test:	coverage.info	**Lines:**	90.5 %	2453	2219
Test Date:	2024-10-19 14:20:11	**Functions:**	95.8 %	571	547

Directory	Line Coverage				Function Coverage		
	Rate		Total	Hit	Rate	Total	Hit
src		56.4 %	39	22	100.0 %	1	1
src/antenna		96.6 %	117	113	98.0 %	49	48
src/core		91.5 %	434	397	96.5 %	144	139
src/interpolation		82.3 %	79	65	87.0 %	23	20
src/math		89.3 %	299	267	92.9 %	85	79
src/noise		96.3 %	162	156	96.6 %	29	28
src/python		72.6 %	73	53	85.7 %	7	6
src/radar		92.7 %	219	203	96.5 %	85	82
src/serial		89.7 %	818	734	100.0 %	85	85
src/signal		97.4 %	154	150	89.5 %	38	34
src/timing		100.0 %	59	59	100.0 %	25	25

Figure 5.1: Coverage report of regression testing framework as determined by `gcov`

The overall coverage rates of original testing framework at the start of the project can be seen in Figure 5.2 below.



			Coverage	Total	Hit
Current view:	top level				
Test:	coverage.info	**Lines:**	48.7 %	3335	1623
Test Date:	2024-10-19 14:23:19	**Functions:**	49.2 %	537	264

Directory	Line Coverage				Function Coverage		
	Rate		Total	Hit	Rate	Total	Hit
src		48.7 %	3335	1623	49.2 %	537	264

Figure 5.2: Coverage report of regression testing framework as determined by `gcov` prior to modernisation

This updated regression testing suite has improved the code coverage by 41.8% of lines and 46.6% of functions.

These improvements in coverage indicate that the testing framework has become more thorough, allowing the detection of potential issues in a larger portion of the codebase. By increasing the coverage of both lines and functions, the reliability of the software has been enhanced, ensuring that critical functionalities are tested and reducing the likelihood of undetected bugs in key components. Despite the improvements, some areas of the code, such as error handling, remain under-tested, suggesting a potential focus for future testing efforts.

5.2.2 FERS Accuracy

Throughout the modernisation process, the regression testing suite was used to ensure that there were no changes in the resulting output data that FERS produces. Therefore, the accuracy of FERS remains identical to what it was before any modernisation effort. However, there are only two instances where this is not true. These instances are when an HDF5 file is used for the antenna gain pattern — which was due to incorrect HDF5 file reading — and minor formatting changes in the output XML files due to the usage of `libxml2` instead of `TinyXML`.

5.3 Performance Metrics

This section presents a detailed analysis of the performance improvements achieved through the modernisation of the FERS software package. Key performance indicators, such as simulation time, CPU usage, memory handling, and threading efficiency, were measured and compared between the original and modernised versions of the software. The results of these comparisons demonstrate the effectiveness of the optimisations implemented, including code refactoring, multithreading enhancements, and the incorporation of modern C++ features. Additionally, the speedup and overall performance gains are highlighted across various test cases, ranging from simple to highly complex radar simulations.

5.3.1 Simulation Time

To evaluate the performance of the modernised FERS software package, one critical metric analysed was simulation time. Given the computational complexity of radar signal simulations, measuring execution time is essential to understanding how modernisation efforts impacted overall performance. To achieve a comprehensive analysis, the most computationally demanding tasks within FERS were focussed on, capturing their execution times as well as the total simulation time. The following key tasks were identified as representative of the system's computational load (listed in order of complexity):

- **exportReceiverBinary:** This function is responsible for generating the raw binary data for every response received by each receiver, and then exporting every response to an HDF5 file.
- **exportReceiverXml:** This function is responsible for rendering individual responses for each receiver in an XML format containing key metrics of each response.
- **exportReceiverCsv:** This function is responsible for recording the time, power, phase, and Doppler-shifted frequency of each response in an CSV format.
- **simulatePair:** This function is responsible for simulating the signal propagation between each transmitter-receiver pair in the world, accounting for any targets, noise, and other factors.
- **parseSimulation:** This function is responsible for reading the input `.fersxml` file which contains the simulation parameters and objects in the world.

Figures 5.3 and 5.4 compare the execution times for these tasks before and after modernisation, across a range of test cases from simple to very complex simulations. This allows us to observe both improvements and regressions in performance at different stages of the modernisation process.

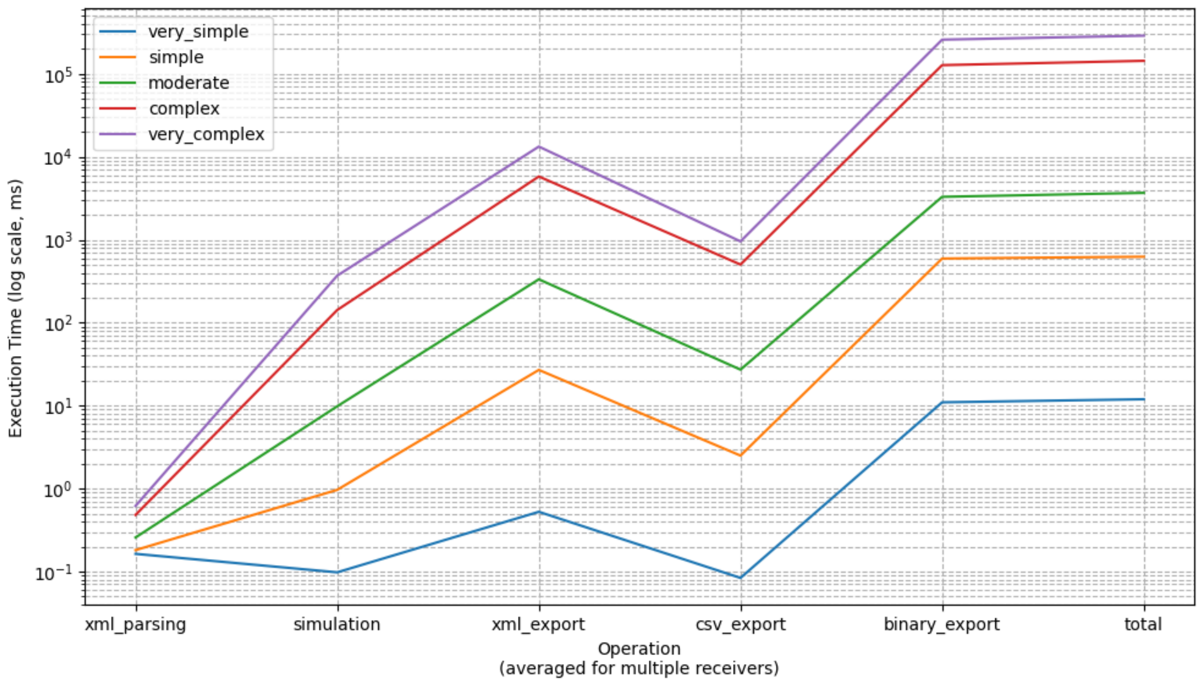


Figure 5.3: Execution time for computationally intensive tasks in original FERS

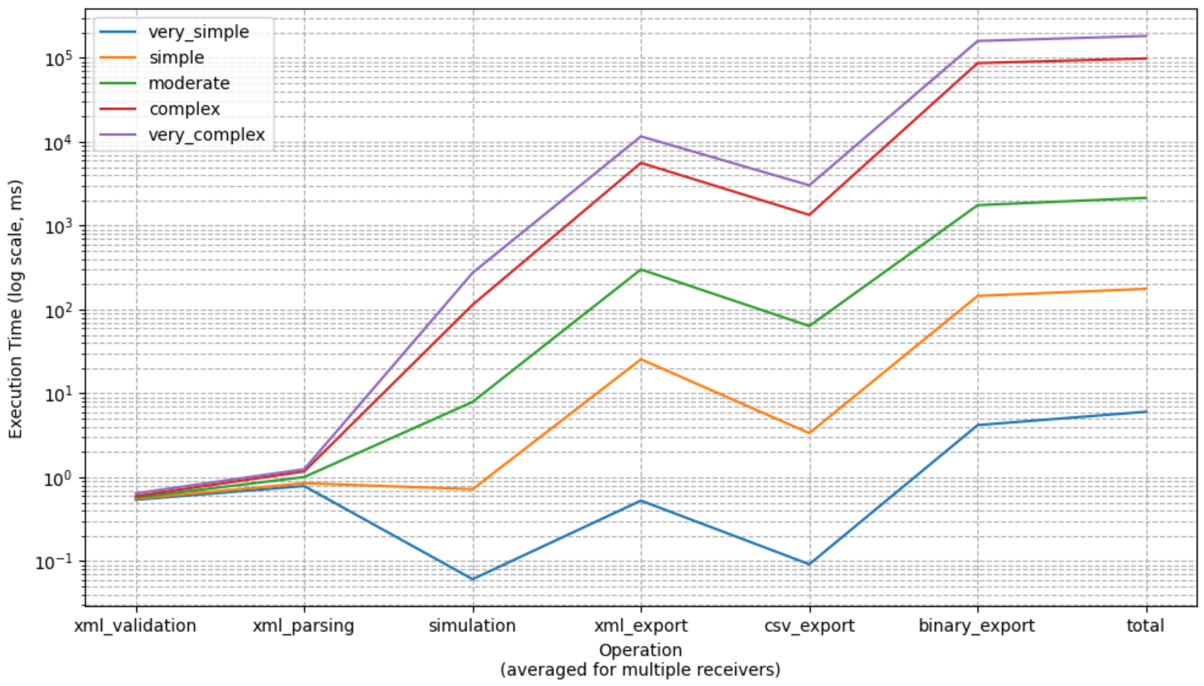


Figure 5.4: Execution time for computationally intensive tasks in modernised FERS

Analysis of Results

XML Parsing: The original FERS had minimal parsing times (0.2 ms to 0.6 ms). modernisation introduced schema validation, slightly increasing times (0.8 ms to 1.3 ms), but this remains efficient and improves data integrity.

Signal Simulation: Simulation times improved, especially in simpler cases. For example, *very simple* decreased from 0.1 ms to 0.06 ms. More complex cases also saw improvements, with *very complex* reducing from 370 ms to 270 ms.

XML Export: Minimal improvements were seen in XML export. In the *very complex* case, export time decreased from 13 s to 12 s, though it remains computationally intensive.

CSV Export: CSV export times increased for complex cases. For example, the *very complex* case rose from 1 s to 3 s, indicating further optimisation is needed for CSV file handling.

Binary Export: HDF5 binary file exporting saw significant improvements, especially for complex simulations. The *very complex* case improved from 260 s to 160 s due to modern file-handling optimisations.

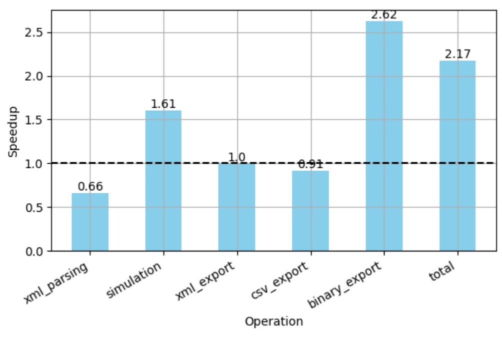
Total Execution Time: Overall, execution time improved, particularly in simpler cases, with the *very simple* case reducing from 12 ms to 6 ms. More complex cases, like *very complex*, also saw reduced times from 260 s to 180 s, reflecting efficient handling of both simple and complex simulations.

5.3.2 Speedup Analysis

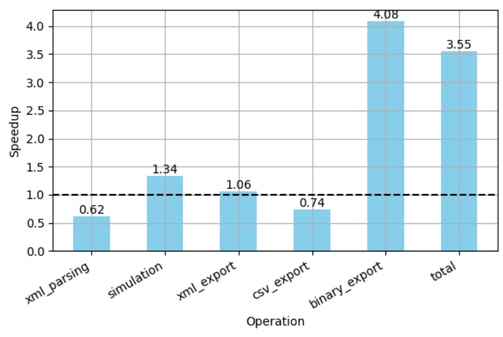
The speedup analysis of the modernised FERS software package focuses on comparing its performance with the original version to highlight the improvements achieved through the modernisation process. This section examines the results of performance enhancements resulting from multithreading and the integration of modern C++20/23 features, as discussed in Chapter 4.

Speedup Results

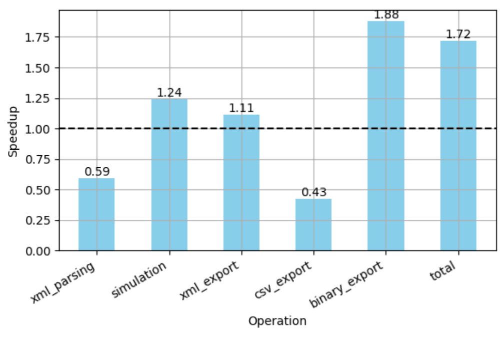
The speedup results for the various test cases — *very simple*, *simple*, *moderate*, *complex*, and *very complex* — are illustrated in Figures 5.5a to 5.5e. These results demonstrate the performance gains achieved in different aspects of the FERS software.



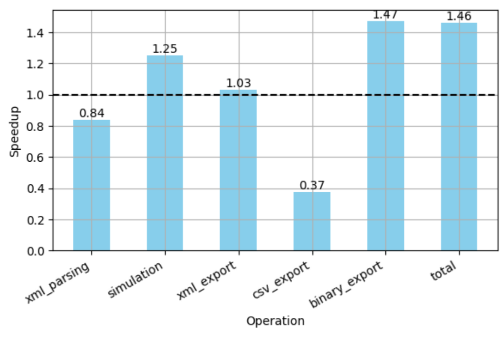
(a) Speedup for *Very Simple* test case



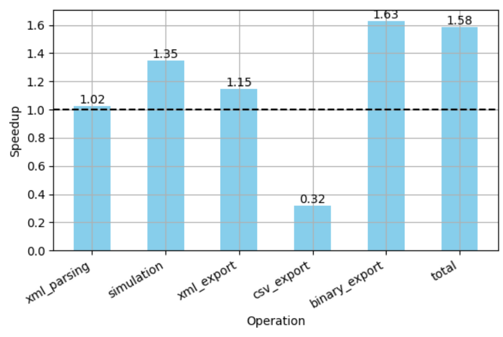
(b) Speedup for *Simple* test case



(c) Speedup for *Moderate* test case



(d) Speedup for *Complex* test case



(e) Speedup for *Very Complex* test case

Figure 5.5: Speedup results for different test cases.

Analysis of Speedup by Operation

The detailed speedup results show varying degrees of improvement across different operations and test cases. The simulation execution, which is the core computational task in FERS, consistently improved across all test cases, with speedup values ranging from 1.24x to 1.61x. This demonstrates the effectiveness of the refactoring process, particularly in optimising signal propagation and noise-handling algorithms.

The HDF5 export function exhibited the most significant improvement, with a speedup of 4.08x in the *simple* test case. The optimisation of memory handling, combined with the adoption of the `HighFive` library for HDF5 file management and the new multithreading management, played a crucial role in this performance gain. In contrast, the CSV export operation experienced a slowdown in most test cases, with speedup values below 1. This suggests that further optimisation is required in this area, possibly by improving file handling or data formatting processes.

XML export exhibited moderate speedups, particularly in more complex test cases, where improvements in memory usage and schema validation contributed to a slight performance gain. Meanwhile, XML parsing showed minimal speedup, with values close to 1 across all test cases. This indicates that further optimisation opportunities for XML parsing may be limited, as the operation was already efficient in the original version.

Total Speedup

The overall speedup for each test case provides a comprehensive view of the cumulative impact of the modernisation efforts. The total speedup values, presented in Table 5.2, range from 146% to 355%, with the most significant improvements observed in the *simple* and *very simple* test cases. These results suggest that the modernised version of FERS is highly optimised for less complex simulations, while performance gains in more complex simulations, though still notable, are somewhat reduced due to the increased computational demands.

Test Case	**Speedup (%)**
Very Simple	217
Simple	355
Moderate	172
Complex	146
Very Complex	158

Table 5.2: Total speedup for each test case

Discussion of Speedup Results

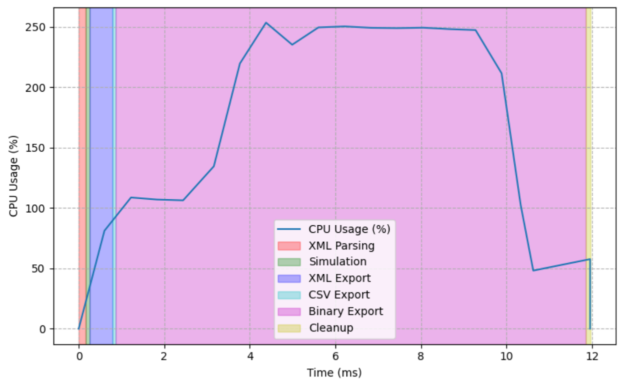
The speedup analysis provides valuable insights into the effectiveness of the modernisation process. The significant improvements in HDF5 export and simulation execution reflect the success of optimisations in memory management and multithreading. In contrast, the slowdown observed in CSV export highlights an area that may benefit from further refinement. Overall, the results validate the effectiveness of the refactoring and optimisation strategies described in Chapter 4, with substantial performance gains in less complex simulations and moderate improvements in more complex cases.

5.3.3 CPU Usage and Threading Efficiency

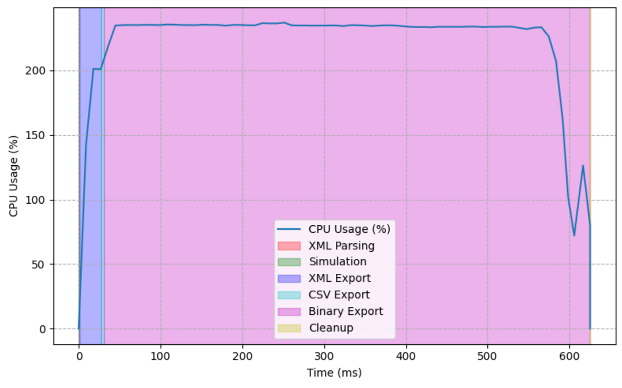
The primary objective of this subsection is to evaluate the CPU usage before and after the modernisation of the FERS software, with a particular focus on multithreading improvements and overall efficiency. By leveraging C++20/C++23 features and implementing a thread pool, the software was optimised to enhance system resource utilisation, especially in terms of CPU load distribution across multiple threads.

Graphical Analysis

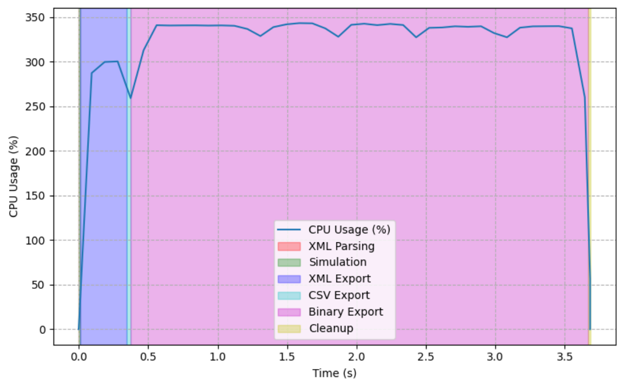
To visualise the improvements in CPU utilisation and multithreading efficiency, the following figures present graphs of CPU usage over time for each test case. Figures 5.6a through 5.6e below show CPU usage for the original version of the software:



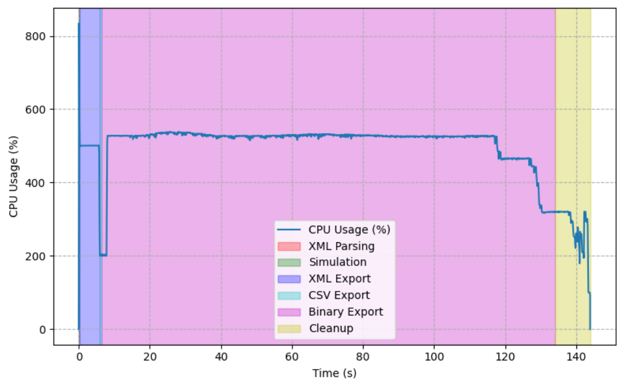
(a) CPU usage vs. average operation times for *Very Simple* test case



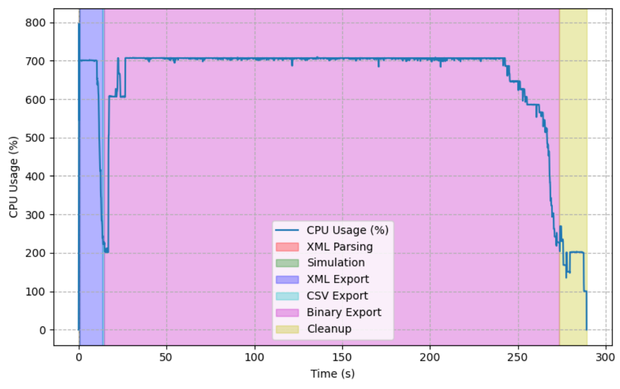
(b) CPU usage vs. average operation times for *Simple* test case



(c) CPU usage vs. average operation times for *Moderate* test case



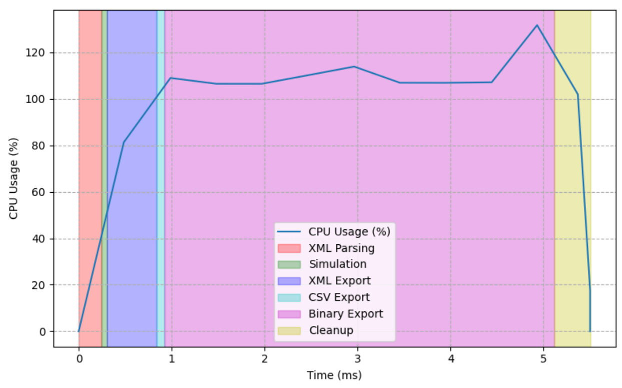
(d) CPU usage vs. average operation times for *Complex* test case



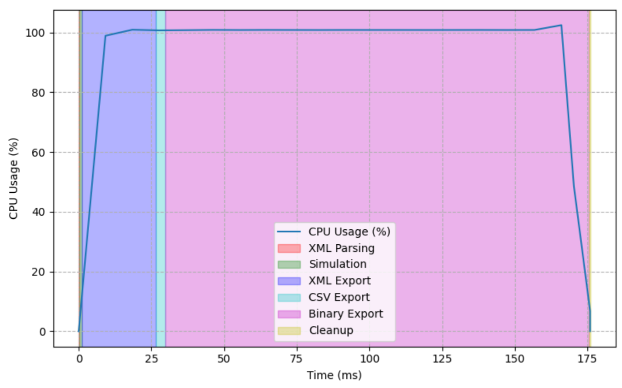
(e) CPU usage vs. average operation times for *Very Complex* test case

Figure 5.6: CPU usage results for different test cases with the original version of FERS

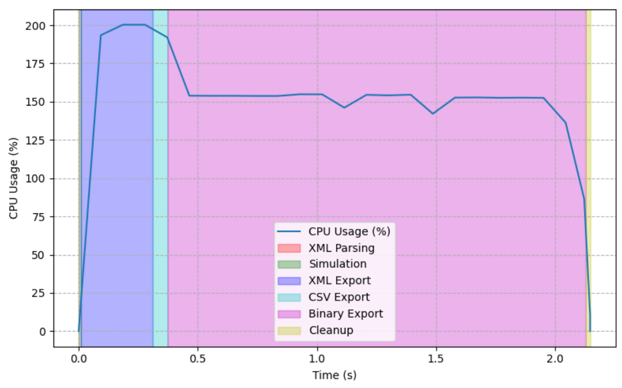
The following figures (5.7a through 5.7e) illustrate CPU usage for the modernised version:



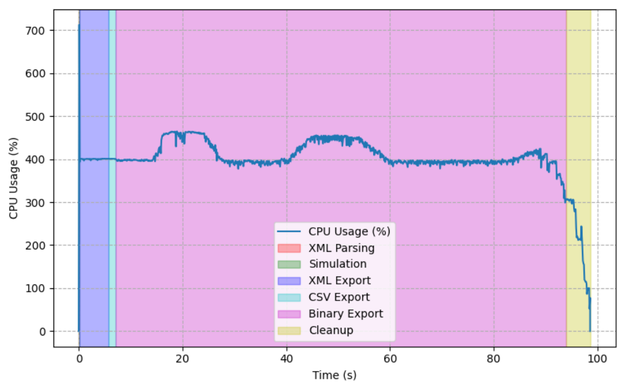
(a) CPU usage vs. average operation times for the *Very Simple* test case



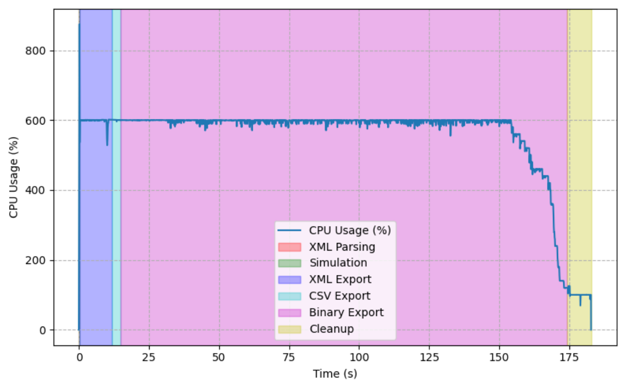
(b) CPU usage vs. average operation times for the *Simple* test case



(c) CPU usage vs. average operation times for the *Moderate* test case



(d) CPU usage vs. average operation times for the *Complex* test case



(e) CPU usage vs. average operation times for the *Very Complex* test case

Figure 5.7: CPU usage results for different test cases with the modernised version of FERS

Analysis of CPU Usage Results

Both the graphical and numerical data reveal significant improvements in threading efficiency and CPU usage with the modernised version of FERS. In the original version, inefficient thread management led to large fluctuations in CPU utilisation, especially in more complex test cases. For example, in the *complex* test, the original version's CPU usage peaked at over 800%, with erratic drops indicating inefficiencies in thread distribution. In contrast, the modernised version exhibited more stable CPU usage, around 400% in the *complex* task, thanks to better thread balancing.

The data in Table 5.3 provides further clarity on these improvements. Across most test cases, the peak and average CPU usage decreased significantly in the modernised version, indicating more efficient CPU load distribution. For instance, in the *moderate* test, the average CPU usage dropped from 323% in the original version to 151% in the modernised version. Notably, the *very complex* test saw an increase in peak CPU usage in the modernised version due to the adaptive threading feature kicking in to handle the increased workload.

Test Case	**Original Peak (%)**	**modernised Peak (%)**	**Original Avg. (%)**	**modernised Avg. (%)**
Very Simple	253	132	173	100
Simple	237	102	223	94
Moderate	343	200	323	151
Complex	835	713	493	401
Very Complex	796	875	649	554

Table 5.3: CPU usage for performance testing tasks (lower is better)

The thread pool and adaptive threading in the modernised version ensured a smoother, more controlled CPU load, which is visually evident in the graphs and supported by the numerical data. These improvements in efficiency not only reduced the overall strain on the CPU but also shortened the total simulation time across all test cases, especially under heavy computational loads.

5.3.4 Compilation Time

The primary goal of this subsection is to analyse the impact of modernising the FERS software package on its compilation times. Since compilation speed can be influenced by code structure, optimisation flags, and CPU usage, this analysis will help determine whether the refactoring and modern C++ features introduced in the latest version of FERS have had a positive, negative, or neutral impact on build efficiency.

Results

Figure 5.8 below summarises the compilation times for both the original and modernised versions of FERS.

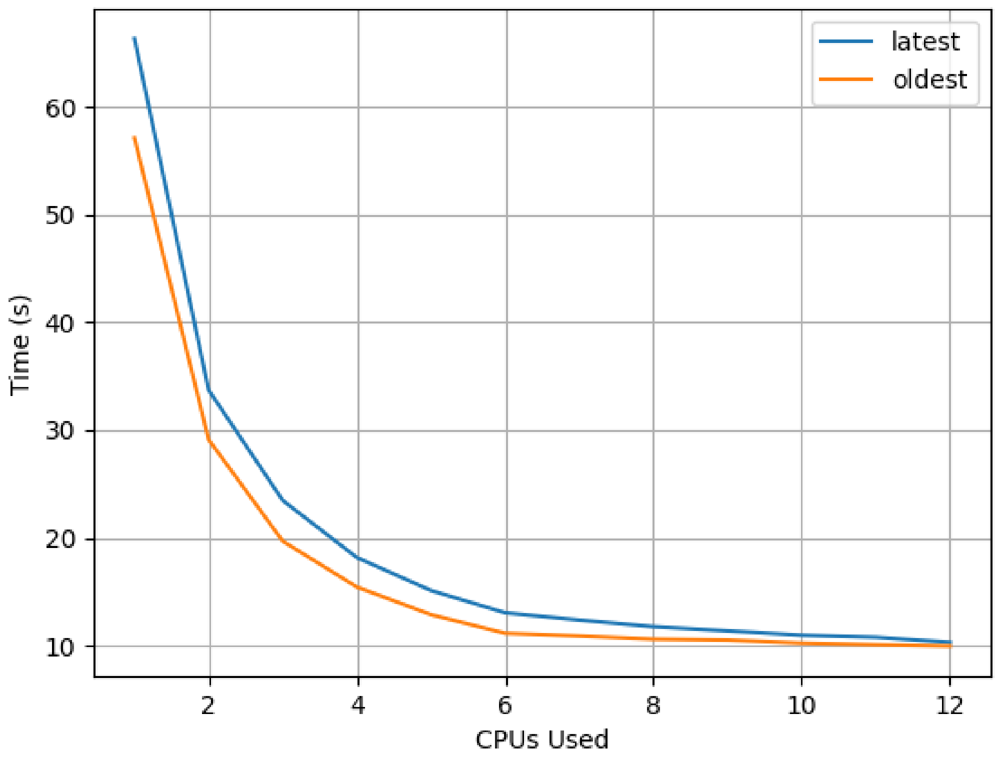


Figure 5.8: Compilation time comparison of the original and modernised versions of FERS

The data shows a general trend of reduced compilation times as the number of CPU cores increases, with both versions benefiting significantly from parallel compilation. However, the modernised version consistently required more time than the original across all CPU configurations, with an average increase of approximately 15% in build time.

Analysis

The results indicate that the modernisation of the FERS software has resulted in a measurable increase in compilation time. Several factors may contribute to this:

1. **Introduction of Modern C++ Features:** The refactored version of FERS incorporates features from C++20/C++23, which may introduce additional overhead during compilation. For example, the use of more complex templates, concepts, and modularisation may require the compiler to perform additional work, thereby increasing build time.

2. **Code modularisation and Refactoring:** While the restructuring and modularisation of the codebase were essential to making FERS more maintainable, these changes likely contributed to the observed increase in build time. Breaking the code into smaller, more modular components often leads to additional overhead in managing dependencies and compilation units.

3. **optimisations and Flag Usage:** Both versions used the `-O3` optimisation flag for the `Release` build, which is designed to produce highly optimised binaries at the cost of longer compilation

times. While this flag was applied consistently, modern C++ optimisations may involve more complex analyses, leading to longer compilation duration.

Interestingly, the performance difference diminishes as more CPU cores are used. For instance, at 12 CPUs, the compilation time gap between the original and modernised versions narrows to approximately 0.35 seconds. This suggests that the parallelisation efficiency of the compilation of the modernised version may be improving, but it has not completely offset the additional overhead introduced by modern features.

5.4 Memory Management and Efficiency

Efficient memory management is a critical aspect of maintaining stable performance in complex simulations, such as the FERS. As systems grow in complexity, the risk of memory leaks, inefficient resource allocation, and performance bottlenecks increases. In this section, a detailed analysis of memory management and performance improvements made during the modernisation of FERS is presented, focusing on memory leaks, context switches, CPU migrations, and page faults.

5.4.1 Memory Leaks

Proper memory management is essential in ensuring that resources are allocated and released efficiently in software like FERS. Memory leaks, where memory is not properly freed after use, can degrade system performance over time, leading to instability and resource exhaustion. To address this, an in-depth memory leak analysis was performed on both the original and modernised versions of FERS using the `valgrind` [44] tool, which tracks memory usage during execution.

Table 5.4 shows the memory leak analysis results for both the original and modernised versions of FERS. Note that in the original version, regression tests 4, 5, and 7 failed because the necessary simulation functionality was missing, so no data could be collected for those cases.

The results of the memory leak analysis, shown in Table 5.4, reveal a significant improvement in memory handling in the modernised version of FERS. In the original version, memory leaks were evident in Tests 6 and 8, where unreleased memory was classified as 'definitely lost'—meaning that the memory was allocated but no longer accessible, making it certain that a leak occurred. Additionally, large amounts of 'still reachable' memory were present in multiple test cases. 'Still reachable' memory refers to memory that is still accessible by the program but was not properly freed before program termination, which, while not always harmful, can lead to wasted resources. 'Possibly lost' memory occurs when memory is allocated but its references are uncertain or ambiguous, leaving it unclear whether the memory is truly lost.

FERS Version	**Test Case**	**Definitely Lost (Bytes)**	**Possibly Lost (Bytes)**	**Still Reachable (Bytes)**
Original	Tests 1-3, 9	0	0	265,905
	Test 6	12,160	0	1,865
	Test 8	12,160	0	265,905
modernised	Tests 1-4, 6-9	0	0	1,864
	Test 5	0	528	933,765

Table 5.4: Valgrind memory leak results for original and modernised versions of FERS

In the modernised version, all ‘definitely lost’ memory was successfully eliminated, demonstrating a marked improvement in memory deallocation. Furthermore, the amount of ‘still reachable’ memory was substantially reduced, except for some residual memory linked to external libraries like the Python C API and HDF5 C API, which caused inefficiencies. The analysis also revealed small instances of ‘possibly lost’ memory in the modernised version, particularly in Test 5. However, we can be certain that the adoption of modern C++ features, including the use of smart pointers like `std::unique_ptr` and `std::shared_ptr`, played a pivotal role in these improvements, allowing for more robust and automatic memory management.

5.4.2 Page Faults, CPU Migrations, and Context Switches

Beyond memory management, the efficiency of system resources such as CPU utilisation and memory access directly influences the performance of complex simulations. Key metrics in this regard include page faults, CPU migrations, and context switches, which have a direct impact on the execution speed and resource allocation of the simulation.

Table 5.5 compares these performance metrics across the original and modernised versions of FERS.

Test Case	**FERS Version**	**Context Switches**	**CPU Migrations**	**Page Faults**
Very Simple	Original	386	106	1772
	modernised	41	12	1520
Simple	Original	526	322	29200
	modernised	42	11	7645
Moderate	Original	480	491	220946
	modernised	461	22	56119
Complex	Original	461	783	784028
	modernised	329	667	956767
Very Complex	Original	485	643	732911
	modernised	552	593	336524

Table 5.5: Page faults, CPU migrations, and context switches in original and modernised FERS

Context Switches: Context switching, which occurs when the CPU transitions between tasks, can introduce overhead and slow down performance. The modernised FERS demonstrated significant reductions in context switches for less complex simulations, with up to a 90% reduction in the 'very simple' test case. This improvement is primarily due to more efficient thread management and minimised task fragmentation in the updated codebase. However, in the 'very complex' test case, context switches increased slightly, likely due to the higher computational demands and increased thread management overhead associated with these more intricate simulations.

CPU Migrations: CPU migrations, which involve moving processes between CPU cores, can lead to performance inefficiencies due to cache invalidation. The modernised FERS showed a marked decrease in CPU migrations across simpler test cases, with the 'simple' test case exhibiting a reduction from 322 to 11 migrations, signifying more efficient load balancing. However, in more complex simulations, such as the 'complex' and 'very complex' cases, CPU migrations remained relatively high due to the intensive nature of these computations, despite improvements over the original version.

Page Faults: Page faults occur when a process attempts to access data that is not present in memory, necessitating retrieval from disk, which can slow down the system. In simpler test cases, the modernised FERS reduced page faults by a significant margin, with reductions of up to 75%. However, in the 'complex' test case, the number of page faults increased dramatically, likely as a result of the larger data set required by the higher Pulse-Repetition frequency (PRF). In contrast, the 'very complex' test case saw a reduction in page faults, indicating more efficient memory handling in scenarios involving larger memory allocations and intensive data processing.

5.5 Code Quality Metrics

In this section, we examine the code quality improvements made during the modernisation of the FERS software package by analysing key code quality metrics, such as Cyclomatic complexity, technical debt, and code inspections, using Understand and CLion. These metrics provide a quantitative insight into the maintainability, complexity, and reliability of the modernised FERS codebase compared to the original version.

5.5.1 Cyclomatic Complexity

After the modernisation, the highest Cyclomatic complexity value was observed in the function `parseAntenna` in `xml_parser.cpp` with a value of 10, which is within the acceptable range for low risk. All other functions received a complexity rating of 'A,' indicating low complexity and high maintainability across the codebase. This reflects a significant improvement, as the original codebase contained functions with moderate complexity ratings (e.g., `RunThreadedSim` with a complexity of 15), which could lead to maintenance difficulties. By refactoring key functions and applying modern C++20 features like structured bindings and range-based loops, the complexity of the code has been reduced while improving clarity.

5.5.2 Technical Debt

The technical debt of the FERS software was reduced significantly during the modernisation process. The technical debt score improved to an 'A' rating with 0 violations, compared to the original moderate 'C' rating. The refactoring process addressed several contributors to technical debt, including:

- **Unused Entities:** Several unused code elements and legacy files such as xmlimport.cpp were removed, reducing code bloat.
- **Memory Management:** Transitioning from manual memory management (using raw pointers) to modern C++ techniques like `std::unique_ptr` and `std::shared_ptr` significantly reduced the risk of memory leaks and dangling pointers.
- **Special Member Functions:** Proper implementation of C++ special member functions (copy constructors, assignment operators, etc.) was enforced, improving object lifecycle management and ensuring safer memory operations.

These efforts helped eliminate unnecessary complexity and improved the maintainability of the codebase.

5.5.3 CLion Inspections

The CLion inspection tool was used to perform static code analysis and check for common issues, such as naming violations, redundancies, and potential code quality issues. The analysis found only a few minor warnings in the modernised version of FERS, most notably the use of recursive call chains which can be safely ignored because they are intended features in the codebase.

5.5.4 modernised Code Quality Metrics

The Table 5.6 below shows the resulting code quality metrics after all modernisation efforts were completed. This is meant as a concise comparison with Table 3.7 in Chapter 3.

Metric	Target Value	Current Value
Cyclomatic complexity	≤ 10	1-10
Comment-to-Code Ratio	60%	57%
Code Quality Warnings	0-10 per 1000 lines	6
Technical Debt Rating	A	A

Table 5.6: Code quality after modernisation

5.6 Acceptance Test Procedures Analysis

This section evaluates how the modernisation of the FERS software package aligns with the ATPs outlined in the project brief. The ATPs serve as criteria for assessing the modernisation's success.

5.6.1 ATP-01: Regression and Backward Compatibility Testing

The first ATP required validation of the modernised software against legacy test cases and ensuring compatibility with existing input/output formats. Regression testing achieved 90% line coverage and

95.4% function coverage, a significant improvement over the original version. All legacy test cases passed, except for two instances where XML formatting differed due to the switch from TinyXML to libxml2, and an issue related to HDF5 file handling, which was resolved. Overall, the backward compatibility of the system has been preserved, with only minor formatting changes that do not impact core functionality. Hence, ATP-01 has been met successfully.

5.6.2 ATP-02: Performance and Memory Testing

Performance and memory testing demonstrated substantial improvements in the modernised system. Speedup analysis showed improvements ranging from 1.46x to 3.55x, particularly in HDF5 export and overall simulation execution. Multithreading efficiency increased, resulting in better load balancing, as evidenced by reduced CPU usage peaks. Memory leak analysis using `valgrind` [44] revealed that all 'definitely lost' memory issues present in the original version were eliminated. These results confirm that the performance and memory usage of the modernised system have improved, fulfilling ATP-02.

5.6.3 ATP-03: Error Handling and Robustness Testing

The modernised FERS system was tested for robustness using corrupted input files and edge cases. The system's error-handling capabilities were improved, consistently returning appropriate error messages and maintaining stability under heavy loads. Stress testing showed no unhandled exceptions, and performance remained consistent even under the most complex simulations. Therefore, ATP-03 has been fully satisfied, as the system demonstrated reliable error handling and robustness.

5.6.4 ATP-04: Documentation Usability Testing

The final ATP focused on ensuring that the documentation generated using Doxygen was complete and user-friendly. A thorough review confirmed that the modernised FERS codebase was well-documented, with clear explanations of modern C++ features and consistent use of naming conventions. Peer reviews further validated that the documentation is understandable for developers. This meets the expectations of ATP-04, confirming that the documentation is accessible and usable.

5.6.5 Summary of ATP Evaluation

In conclusion, all ATPs have been successfully met. The modernised FERS system maintains backward compatibility, improves performance, enhances error handling, and provides comprehensive documentation. Minor changes in XML output formatting were the only notable deviations, but these do not affect the core objectives. Overall, the modernisation has achieved its goals in functionality, performance, and usability.

5.7 Conclusion

In conclusion, the modernisation of the FERS software package has resulted in substantial improvements across functionality, performance, and maintainability. Functionality testing confirmed that the updated software maintains accuracy while increasing code coverage and addressing prior issues, such as HDF5 file handling. Performance metrics, particularly in overall simulation time and HDF5 export, showed significant speedup, with a total speedup ranging from 1.46x to 3.55x depending on the test case. CPU usage and threading efficiency were notably enhanced, with improved load balancing and reduced overhead, demonstrating the success of the multithreading optimisations. Memory management was also significantly refined, eliminating all ‘definitely lost’ memory and reducing ‘still reachable’ memory in most cases.

While some challenges remain, such as optimising CSV export and addressing compilation time increases due to modern C++ features, the overall gains in performance and code quality validate the modernisation approach. The reductions in Cyclomatic complexity, technical debt, and context switches have contributed to a more maintainable and efficient codebase, providing a solid foundation for future development and optimisation.

Chapter 6

Conclusions & Future Work

The measure of intelligence is the ability to change.

—*Albert Einstein*

6.1 Conclusions

The modernisation of the FERS has successfully transformed it from a legacy C++ codebase into a more efficient, maintainable, and future-ready software package. This project focused on upgrading the system to incorporate modern C++20/23 features, optimise performance, and improve maintainability without compromising the core functionality that has made FERS a valuable tool for radar simulations. Through systematic refactoring and validation, the modernisation effort has enhanced both the performance and usability of the software, ensuring it meets contemporary standards while retaining backward compatibility.

6.1.1 Key Accomplishments

One of the most notable improvements was the shift from manual memory management to the use of modern C++ constructs like smart pointers. This has greatly reduced the risks of memory leaks and errors, increasing the reliability and safety of the system. Additionally, integrating advanced C++20/23 features, such as lambda expressions, concepts, and structured bindings, has improved code clarity, maintainability, and performance, enabling better type safety and reducing runtime errors.

6.1.2 Performance Improvements

The project delivered significant performance gains, particularly in the handling of multithreading, radar signal simulations, and HDF5 file export. The introduction of a global thread pool and adaptive threading strategies allowed for more efficient task execution, reducing total execution times in test cases with speedup factors between 1.46x and 3.55x. The transition to the HighFive library for data management contributed to faster and more efficient file handling, particularly for large datasets used in complex radar simulations.

Memory management saw substantial improvements, with all memory leaks effectively eliminated and better control over resource allocation. This, combined with the adoption of modern C++ features, has made the system both faster and more reliable.

6.1.3 Code Quality and Maintainability

The codebase has been extensively refactored to improve readability and maintainability, resulting in a reduction of technical debt and cyclomatic complexity. By modularising the structure, adopting clear naming conventions, and improving documentation through Doxygen, the code is now easier to navigate and extend. The system's technical debt rating improved from a moderate 'C' to an 'A,' ensuring the code is cleaner and more sustainable for future development.

6.1.4 Testing and Validation

Extensive regression testing was implemented, achieving 90.5% line coverage and 95.8% function coverage, ensuring the system remains backward compatible and reliable. The testing suite demonstrated that the modernised FERS system continues to meet the original simulation accuracy standards while improving performance. Backward compatibility was preserved with only minor differences in output formatting due to library changes, and comprehensive error-handling mechanisms were introduced to further enhance system robustness.

6.2 Future Work

While the modernisation of FERS has significantly improved its performance, maintainability, and overall usability, several areas for further enhancement remain. Addressing these will enable FERS to take full advantage of cutting-edge software development practices and further optimise its performance for large-scale radar simulations. The following sections outline key directions for future work.

The choice was made to not implement the future work topics discussed below due to time and resource constraints, and the need to prioritise the core goals of the project: modernising the FERS codebase for maintainability, performance, and C++ standards compliance. Implementing features like module migration, multithreaded rendering, and visualiser integration would have required significant additional effort and introduced potential risks to the system's stability. These tasks involve deeper architectural changes that were beyond the scope of this project, making them more suitable for future work.

6.2.1 Integration of visualiser

Currently, FERS relies on external tools for visualising simulation results. Integrating the visualiser from the `config_validators` directory directly into FERS will streamline the workflow, providing users with a native tool for reviewing radar simulation outputs without switching between programs. By adding a program argument to trigger the visualiser, users can easily choose to run the visualiser alongside the simulation. This will improve both usability and efficiency, enabling users to view results without post-processing steps.

6.2.2 Migration to Modules

The modern C++20/23 standard introduces modules as a powerful tool to replace traditional header files, improving compilation times and reducing dependencies between translation units. Future work should focus on migrating the FERS codebase to adopt modules, thereby enhancing scalability and reducing the complexity of build processes. This would help improve build performance and maintainability, especially as the codebase expands with more features and functionality.

6.2.3 Rework Rendering for Multithreading of Windows

The current design for rendering receiver responses limits multithreading to rendering individual responses, missing opportunities for parallelisation at a higher level. Reworking the rendering system to support multithreading at the window level—rather than just the response level—will allow for more efficient use of CPU resources, particularly in large simulations with numerous responses per window. This optimisation will improve performance by reducing bottlenecks in rendering operations.

6.2.4 HDF5 Output Chunking

Currently, the entire set of receiver responses is kept in memory until the simulation is complete, which can lead to excessive memory usage in large simulations. To address this, the HDF5 output process should be reworked to write chunks of data incrementally, freeing up memory for responses that have already been written. This would reduce the memory footprint of FERS and allow it to handle much larger datasets without running into memory limitations. This approach would also support more efficient data storage and retrieval from disk.

6.2.5 On-the-fly File Writing and Rendering

Another key area for improvement is simulating signal propagation and immediately writing the results to files, rather than keeping all responses in memory until the end of the simulation. Once written, these files can be rendered in parallel or as needed, reducing the need for keeping large amounts of data in memory throughout the simulation process. This shift to an on-the-fly processing model will greatly enhance FERS's ability to handle large-scale simulations, especially when working with memory-constrained environments.

6.2.6 CSV Export optimisation

The performance of the CSV export function has been identified as a potential bottleneck, particularly in more complex simulations. Improving the efficiency of this process, possibly by reworking the file writing logic and optimising data handling, will ensure that the CSV export operation does not lag behind other output formats. This would bring CSV output performance in line with the optimised HDF5 export process, ensuring a consistent and efficient export across all output formats.

6.2.7 Compilation Time optimisation

Despite the performance gains in execution, the modernisation of FERS has led to a slight increase in compilation times, primarily due to the complexity introduced by modern C++ features. Further investigation into optimising the build process, such as using pre-compiled headers and more efficient dependency management, will help reduce these compilation times. Additionally, the transition to modules, as discussed earlier, is expected to play a significant role in minimising the overhead caused by increased complexity in the codebase.

Bibliography

[1] M. Skolnik, “Radar,” Aug 2024. [Online]. Available: https://www.britannica.com/technology/radar

[2] J. Kannanthara, D. Griffiths, M. Jahangir, J. M. Jones, C. J. Baker, M. Antoniou, C. J. Bell, H. White, K. Bongs, and Y. Singh, “Whole system radar modelling: Simulation and validation,” *IET Radar, Sonar & Navigation*, vol. 17, no. 6, pp. 1050–1060, 2023. [Online]. Available: https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/rsn2.12399

[3] Oct 2021. [Online]. Available: https://airandspace.si.edu/explore/stories/air-traffic-control

[4] R. R. Boothe, *A Digital Computer Program for Determining the Performance of an Acquisition Radar Through Application of Radar Detection Probability Theory.* Defense Technical Information Center, 1964.

[5] T. Balz, “Real-time sar simulation of complex scenes using programmable graphics processing units,” in *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences - ISPRS Archives*, vol. 36, 2006.

[6] P. J. Golda, “Software simulation of synthetic aperture radar,” S.l, 1997.

[7] R. Lengenfelder, “The design and implementation of a radar simulator,” S.l, 1998.

[8] MATLAB. [Online]. Available: https://www.mathworks.com/products/radar.html

[9] cadence. [Online]. Available: https://www.cadence.com/en_US/home/tools/system-analysis/rf-microwave-design/awr-design-environment-platform.html

[10] Keysight, “Pathwave system design (systemvue),” Jan 2018. [Online]. Available: https://www.keysight.com/us/en/products/software/pathwave-design-software/pathwave-system-design-software.html

[11] T. Fatima, “Best practices for managing legacy code,” Jul 2023. [Online]. Available: https://remotebase.com/blog/best-practices-for-managing-legacy-code

[12] S. Team, “Legacy code: 5 challenges, tools and tips to overcome them,” May 2024. [Online]. Available: https://swimm.io/learn/legacy-code/legacy-code-5-challenges-tools-and-tips-to-overcome-them

[13] P. Morlion, “What is code rot and how do you identify it?: Linearb blog,” Jan 2022. [Online]. Available: https://linearb.io/blog/what-is-code-rot

[14] S. Eick, T. Graves, A. Karr, J. Marron, and A. Mockus, “Does code decay? assessing the evidence from change management data,” *IEEE Transactions on Software Engineering*, vol. 27, no. 1, pp. 1–12, 2001.

[15] D. Fagbuyiro, “Code refactoring best practices – with python examples,” Aug 2022. [Online]. Available: https://www.freecodecamp.org/news/best-practices-for-refactoring-code/

[16] S. Melashich, “Software re-engineering and re-engineering process: Agilie,” Jun 2024. [Online]. Available: https://agilie.com/blog/what-is-software-reengineering

[17] p. pankaj, “Re-engineering - software engineering,” May 2024. [Online]. Available: https://www.geeksforgeeks.org/software-engineering-re-engineering/

[18] I. e. team, “What is a wrapper in programming?” Sep 2020. [Online]. Available: https://www.ionos.com/digitalguide/websites/web-development/what-is-a-wrapper/#c264372

[19] “Kde community.” [Online]. Available: https://kde.org/

[20] W. Lucas, F. Carvalho, R. C. Nunes, R. Bonifácio, J. Saraiva, and P. Accioly, “Embracing modern c++ features: An empirical assessment on the kde community,” *Journal of Software: Evolution and Process*, vol. 36, no. 5, p. e2605, 2024. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2605

[21] K. T. Hanna, “What is backward compatible (backward compatibility)?” Sep 2021. [Online]. Available: https://www.techtarget.com/whatis/definition/backward-compatible-backward-compatibility

[22] “Clang-tidy - extra clang tools 20.0.0git documentation,” Jun 2024. [Online]. Available: https://clang.llvm.org/extra/clang-tidy/

[23] L. Mabel, “Best practices for refactoring legacy code to make it more maintainable and easier to work with,” Oct 2023. [Online]. Available: https://dev.to/eusoumabel/best-practices-for-refactoring-legacy-code-to-make-it-more-maintainable-and-easier-to-work-with-5cem

[24] “What is boilerplate code?” [Online]. Available: https://aws.amazon.com/what-is/boilerplate-code/

[25] 10xlearner, “Memory management and raii,” Jan 2020. [Online]. Available: https://www.codeproject.com/Articles/5257335/Memory-Management-and-RAII

[26] L. Stanford, “Advanced c++ memory management techniques: A 2024 guide,” Jan 2024. [Online]. Available: https://www.geekpedia.com/cpp-memory-management-2024/

[27] C. Caprile and P. Tonella, “Nomen est omen: analyzing the language of function identifiers,” in *Sixth Working Conference on Reverse Engineering (Cat. No.PR00303)*, 1999, pp. 112–122.

[28] F. Deissenboeck and M. Pizka, “Concise and consistent naming,” *Software Quality Journal*, vol. 14, no. 3, p. 261–282, Sep 2006. [Online]. Available: https://link.springer.com/article/10.1007/s11219-006-9219-1

[29] Tom, “Concepts in c++20,” Jan 2022. [Online]. Available: https://thecodehound.com/concepts-in-c20/

[30] ——, “Ranges in c++20,” Aug 2022. [Online]. Available: https://thecodehound.com/ranges-in-c20/

[31] “Concurrency support library (since c++11).” [Online]. Available: https://en.cppreference.com/w/cpp/thread

[32] A. Fertig, “C++20 modules: The possible speedup,” Sep 2021. [Online]. Available: https://andreasfertig.blog/2021/09/cpp20-modules-the-possible-speedup/

[33] S. Toth, “Daily bit(e) of c++: Coroutines: Step by step,” Jul 2024. [Online]. Available: https://itnext.io/daily-bit-e-of-c-coroutines-step-by-step-e726b976d239

[34] A. Ozeritskii, “Implementation of the raft consensus algorithm using c++20 coroutines,” Feb 2024. [Online]. Available: https://dzone.com/articles/implementation-of-the-raft-consensus-algorithm-usi

[35] Mar 2024. [Online]. Available: https://www.testingxperts.com/blog/regression-testing

[36] D. Beyer, S. Löwe, and P. Wendler, “Reliable benchmarking: Requirements and solutions,” *International Journal on Software Tools for Technology Transfer*, vol. 21, no. 1, p. 1–29, Nov 2017. [Online]. Available: https://doi.org/10.1007/s10009-017-0469-y

[37] Amazon. [Online]. Available: https://aws.amazon.com/devops/continuous-integration/

[38] jaysurya9, “What is continuous integration?” Nov 2023. [Online]. Available: https://www.geeksforgeeks.org/what-is-continuous-integration/

[39] GitHub, “What is github actions? how automation & ci/cd work on github,” Feb 2022. [Online]. Available: https://resources.github.com/devops/tools/automation/actions/

[40] V. Poturaev, “Continuous benchmarking with go and github actions,” Nov 2020. [Online]. Available: https://dev.to/vearutop/continuous-benchmarking-with-go-and-github-actions-41ok

[41] JetBrains, “Clion: A cross-platform ide for c and c++ by jetbrains,” Jun 2021. [Online]. Available: https://www.jetbrains.com/clion/

[42] SciTools. [Online]. Available: https://scitools.com/

[43] akash1295, “Cyclomatic complexity,” Jun 2024. [Online]. Available: https://www.geeksforgeeks.org/cyclomatic-complexity/

[44] Valgrind, 2024. [Online]. Available: https://valgrind.org/

[45] perf, Oct 2024. [Online]. Available: https://perfwiki.github.io/main/

[46] Doxygen, “Doxygen/doxygen: Official doxygen git repository,” Aug 2024. [Online]. Available: https://github.com/doxygen/doxygen

[47] C. Wolff, “The radar equation.” [Online]. Available: https://www.radartutorial.eu/01.basics/The%20Radar%20Range%20Equation.en.html

[48] M. Brooker, "The design and implementation of a simulator for multistatic radar systems," Ph.D. dissertation, University of Cape Town, 2008. [Online]. Available: http://hdl.handle.net/11427/5253

[49] R. Keim, "The nyquist–shannon theorem: Understanding sampled systems," May 2020. [Online]. Available: https://www.allaboutcircuits.com/technical-articles/nyquist-shannon-theorem-understanding-sampled-systems/

Appendix A

Code

A.1 Code Repository

The source code for the modified version of FERS can be found at https://github.com/the-user-created/FERS

A.2 processDocument Simplification

```cpp
// BEFORE:
void ProcessDocument(TiXmlHandle &root, World *world, bool included)
{
  if (!included) {
    //Process the parameters
    TiXmlHandle parameters = root.ChildElement("parameters", 0);
    ProcessParameters(parameters);
  }
  //Process all the pulses
  TiXmlHandle plat = root.ChildElement("pulse", 0);
  for (int i = 1; plat.Element() != 0; i++) {
    ProcessPulse(plat, world);
    plat = root.ChildElement("pulse", i);
  }
  //Process all the antennas
  plat = root.ChildElement("antenna", 0);
  for (int i = 1; plat.Element() != 0; i++) {
    ProcessAntenna(plat, world);
    plat = root.ChildElement("antenna", i);
  }
  /*
  More near identical snippets of code as the loops above
  ...
  */
  //Process all the incblocks
  plat = root.ChildElement("incblock", 0);
  for (int i = 1; plat.Element() != 0; i++) {
    ProcessDocument(plat, world, true); //Recursively process the platform
    plat = root.ChildElement("incblock", i);
  }
}

// AFTER:
void parseSimulation(const std::string& filename, World* world, const bool validate)
{
    XmlDocument main_doc;

    const fs::path main_dir = fs::path(filename).parent_path();

    const bool did_combine = addIncludeFilesToMainDocument(main_doc, main_dir);

    const XmlElement root = main_doc.getRootElement();

    parseParameters(root.childElement("parameters", 0));
    parseElements(root, "pulse", world, parsePulse);
    parseElements(root, "timing", world, parseTiming);
    parseElements(root, "antenna", world, parseAntenna);
    parseElements(root, "platform", world, parsePlatform);
    parseElements(root, "multipath", world, parseMultipathSurface);

    world->processMultipath();
}
```

Listing A.1: Simplification of `processDocument`

Appendix B

Theoretical Background of Radar Systems

Radar systems are critical tools in various applications, including defence, aviation, and meteorology [3, 2], due to their ability to detect and measure the distance, velocity, and characteristics of objects at a distance. Understanding the theoretical principles that govern radar systems is essential not only for grasping how these systems function but also for effectively simulating their behaviour in software.

This appendix provides a theoretical foundation for understanding the operation of radar systems and the principles behind radar simulation. This knowledge is crucial for the modernisation of the FERS software package. By grounding the modernisation process in solid theoretical concepts, we can ensure that the updated FERS software will accurately model radar behaviour, maintain its reliability, and enhance its performance using modern C++ features and optimisation techniques.

B.1 Fundamental Principles of Radar Systems

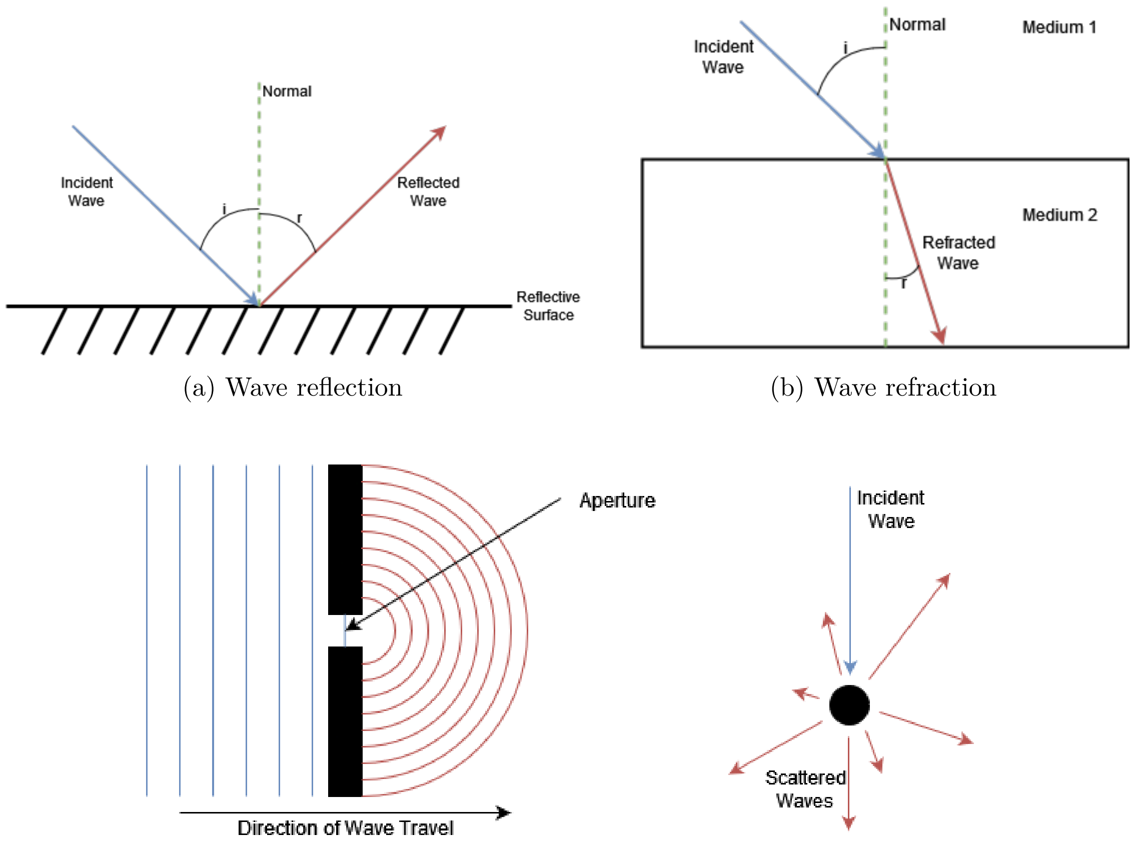
Radar systems operate on the fundamental principle of detecting objects by transmitting electromagnetic waves and analysing the signals that return after reflecting off objects. Understanding the underlying physics and the core components of radar systems is essential for any radar-related research, including the modernisation of radar simulation software like FERS. This section introduces the basic principles of electromagnetic wave propagation, the radar equation, and the essential components that make up a radar system.

B.1.1 Electromagnetic Wave Propagation

Radar systems operate by transmitting electromagnetic waves and analysing the signals that return after interacting with objects in the environment. The behaviour of these waves is influenced by several key phenomena, as illustrated in Figure B.1:

- **Reflection:** Electromagnetic waves bounce off surfaces, with the strength and direction of the reflection depending on the object's material, shape, and orientation.
- **Refraction:** As waves pass through different mediums, they bend, potentially altering the path of the radar signal and affecting the accuracy of target location.
- **Diffraction:** Waves can bend around obstacles or spread out after passing through small openings, impacting how signals propagate in complex environments.

- **Scattering:** Small particles or irregularities in the atmosphere cause waves to disperse in multiple directions, leading to signal attenuation and potential challenges in signal interpretation.



(a) Wave reflection (b) Wave refraction

(c) Wave diffraction (d) Wave scattering

Figure B.1: Illustrations of electromagnetic wave phenomena affecting radar signal propagation

B.1.2 Radar Equation

The radar equation is a fundamental expression that relates the power of the received signal to the transmitted power, the characteristics of the target, and the distance between the radar and the target. It provides a mathematical framework for understanding how various factors influence radar performance.

Monostatic Radar Equation

The basic form of the radar range equation for monostatic radar systems is [47]:

$$P_r = \frac{P_t G_t G_r \lambda^2 \sigma}{(4\pi)^3 R^4 L} \tag{B.1}$$

where:

- P_r is the received power.
- P_t is the transmitted power.

- G_t and G_r are the gains of the transmitting and receiving antennas, respectively.
- λ is the wavelength of the radar signal.
- σ is the radar Radar Cross-Section (RCS) of the target, representing its reflectivity.
- R is the range or distance between the radar and the target.
- L represents system losses due to various factors such as propagation, hardware inefficiencies, or atmospheric absorption.

This equation highlights the inverse fourth power relationship between the received power and the distance to the target, illustrating how radar performance diminishes rapidly with increasing range.

Bistatic Radar Equation

The following discussion on the bistatic radar equation is primarily based on [48].

In bistatic radar systems, where the transmitter and receiver are at different locations, the radar equation is modified. The power radiated by a target in the direction of the receiver, after being illuminated by the transmitter, is:

$$P_g = P_t \frac{G_t L_t L_{pt} \sigma_b}{4\pi R_{kj}^2} \tag{B.2}$$

where:

- P_t is the transmitted power.
- G_t is the transmitter antenna gain.
- L_t and L_{pt} are the transmitter loss and propagation loss.
- σ_b is the bistatic RCS.
- R_{kj} is the range from the transmitter to the target.

The power received by the receiver is:

$$P_r = P_g \frac{G_r L_r L_{pr} \lambda^2}{(4\pi)^2 R_{ik}^2} \tag{B.3}$$

where:

- G_r is the receiver antenna gain.
- L_r and L_{pr} are the receiver loss and propagation loss.
- R_{ik} is the range from the target to the receiver.

Combining these, the bistatic radar equation is:

$$P_r = P_t \frac{G_t G_r L_t L_r L_{pt} L_{pr} \sigma_b \lambda^2}{(4\pi)^3 R_{kj}^2 R_{ik}^2} \tag{B.4}$$

For direct transmission from the transmitter to the receiver (excluding the target effects):

$$P_r = P_t \frac{G_t G_r L_t L_r \lambda^2}{(4\pi)^2 R_{ij}^2} \tag{B.5}$$

This equation accounts for the geometry of bistatic radar, showing how received power depends on the distances and angles between the transmitter, target, and receiver.

B.2 Radar System Configurations

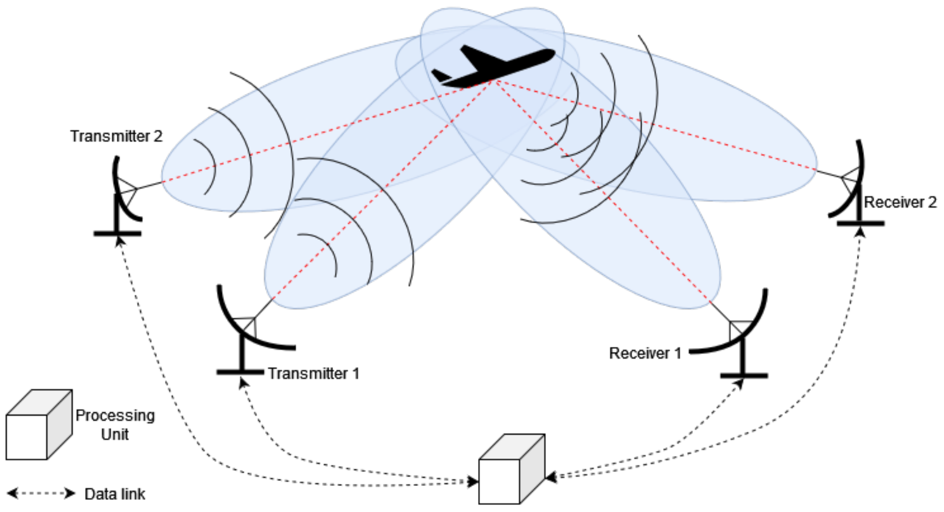
B.2.1 The Three Broad Categories

Radar systems can be broadly categorised into three types based on the geometric arrangement of their transmitter(s) and receiver(s). These categories are discussed in subsection 2.1.2.

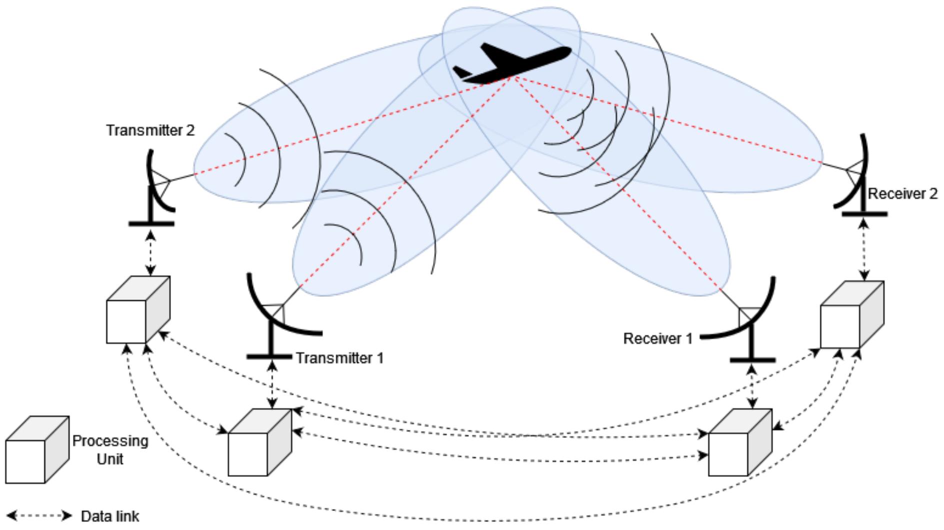
B.2.2 Netted Radar Systems

Although netted radar systems are a form of multistatic radar, they represent a more advanced concept involving the networking of multiple radar sensors that share data in real time. Each radar unit in the network operates as part of a larger, coordinated system, significantly enhancing coverage area, detection accuracy, and resilience against countermeasures. Netted radar systems can be configured in various ways, such as:

- **Centralised Networks:** All radar data is processed at a central location, allowing for more sophisticated data fusion techniques that improve target tracking and identification. This system is shown in Figure B.2a.
- **Distributed Networks:** Each radar unit processes data locally and shares information with other units. This decentralised approach enhances the system's robustness, ensuring that the failure of one unit does not incapacitate the entire network. This system is shown in Figure B.2b.



(a) Netted radar system utilising a centralised network



(b) Netted radar system utilising a distributed network

Figure B.2: Illustrations of netted radar systems configured in different network topologies

Netted radar systems are particularly valuable in modern defence applications, where the ability to detect, track, and engage targets with high accuracy and reliability is paramount. By integrating data from multiple sources, these systems provide a more detailed and accurate picture of the operational environment, making them a critical component in modern air defence strategies.

B.3 Advanced Radar Concepts

B.3.1 Doppler Effect in Radar

The Doppler effect is a key phenomenon used by radar systems to measure the relative velocity of a moving object. It occurs when there is relative motion between the radar system and the target, causing a shift in the frequency of the reflected radar wave compared to the transmitted wave. This frequency shift is directly proportional to the target's relative velocity along the radar's line of sight.

Monostatic Radar Systems

In a monostatic radar system, where the same antenna is used for both transmission and reception, the Doppler frequency shift f_d can be expressed as [48]:

$$f_d = \frac{2v_r}{\lambda} \tag{B.6}$$

where:

- v_r is the radial velocity of the target relative to the radar system.
- λ is the wavelength of the transmitted signal.

The factor of 2 in the equation arises because the radar wave undergoes a round-trip journey—first from the radar to the target and then back from the target to the radar. This formula assumes that the target's velocity is much less than the speed of light ($v_r \ll c$).

Bistatic Radar Systems

The Doppler effect is a critical concept in radar systems, particularly for measuring the velocity of a moving target. When a target moves relative to both the transmitter and receiver in a bistatic radar system, the frequency of the reflected signal shifts. This frequency shift, known as the Doppler shift, is proportional to the relative velocities of the target with respect to the transmitter and receiver. It is given by [48]:

$$f_d = \frac{v_r + v_t}{\lambda} \tag{B.7}$$

where:

- f_d is the Doppler frequency shift.
- v_t is the radial velocity of the target relative to the transmitter.
- v_r is the radial velocity of the target relative to the receiver.
- λ is the wavelength of the transmitted signal.

This equation assumes ideal conditions, such as linear target motion and small angles between the target's velocity vector and the bistatic baseline (the line connecting the transmitter and receiver). In more complex bistatic configurations, the Doppler shift would be influenced by the rate of change in the distances between the transmitter, receiver, and target.

For more complex bistatic configurations, the full bistatic Doppler shift equation provides a more accurate calculation of the Doppler shift. It is expressed as [48]:

$$f_d = \frac{f_0}{c}\left(\frac{dR_{ik}}{dt} + \frac{dR_{jk}}{dt}\right) \tag{B.8}$$

where:

- f_0 is the transmitted frequency,
- c is the speed of light,
- $\frac{dR_{ik}}{dt}$ is the rate of change of the distance between the transmitter and the target,
- $\frac{dR_{jk}}{dt}$ is the rate of change of the distance between the target and the receiver.

In this formulation, the Doppler shift is influenced by how quickly the distances between the transmitter and the target, and between the target and the receiver, are changing. This approach simplifies the calculation by focusing on these distance rates, which are often easier to measure in bistatic radar systems. The overall Doppler shift is the sum of the contributions from the transmitter-to-target and target-to-receiver paths.

B.3.2 Superposition Principle in Radar System Simulation

The principle of superposition is fundamental in the modelling and simulation of radar systems, particularly when dealing with multistatic radar configurations. This principle allows the complex interactions within a radar system to be broken down into simpler, linear components, which can then be individually simulated and summed to provide an accurate overall system model [48].

Mathematical Foundation of Superposition in Radar Systems

In the context of radar systems, superposition refers to the ability to decompose the received signal at a radar receiver into the sum of contributions from multiple transmitters. Mathematically, if we express the radar system's effect as a function $f(x)$, this function satisfies the criteria of linearity, which consists of two key properties [48]:

1. **Homogeneity (Scaling):**
$$f(\alpha x) = \alpha f(x) \tag{B.9}$$
   This property implies that if the input signal is scaled by a factor α, the output will be scaled by the same factor.

2. **Additivity:**
$$f(x + y) = f(x) + f(y) \tag{B.10}$$
   This indicates that the response of the system to the sum of two inputs is equal to the sum of the responses to each input separately.

These properties are crucial because they allow the simulation model to treat the signals from different transmitters independently and then combine them to form the received signal.

Implementation in Multistatic Radar Simulations

In a multistatic radar system, the principle of superposition simplifies the complex task of simulation by reducing it to the sum of several bistatic radar simulations. Each bistatic radar setup involves one transmitter and one receiver, and the signal received by a given receiver can be expressed as the sum of the modified signals from all transmitters. This can be formulated as [48]:

$$y_i[n] = \sum_{j=0}^{N_T} f_{ij}(x_j[n]) \tag{B.11}$$

where:

- $y_i[n]$ is the discrete-time signal received by receiver i.
- $x_j[n]$ is the signal transmitted by transmitter j.
- f_{ij} represents the linear (but not necessarily time-invariant) function that modifies the transmitted signal $x_j[n]$ as it propagates to the receiver.

This equation essentially states that the received signal at each receiver is the sum of the contributions from all transmitters, each modified by the effects of transmission, propagation, and reception.

Modelling Signal Reception in Multistatic Systems

The received signal $y_i[n]$ can be further detailed to consider the individual effects of the radar system components [48]:

$$y_i[n] = R_i\left(\sum_{j=0}^{N_T} E_{ij}\left(T_j\left(x_j[n]\right)\right)\right) \tag{B.12}$$

where:

- R_i represents the effect of reception by receiver$i$.
- E_{ij} models the environmental effects during the propagation of the signal from transmitter j to receiver i.
- T_j captures the effect of transmission by transmitter j.

These functions $R_i(x,t)$, $E_{ij}(x,t)$, and $T_j(x,t)$ are functions of both the signal and time, reflecting the non-time-invariant nature of real-world radar systems.

Assumptions Underlying the Superposition Principle

For the superposition model to be valid, several key assumptions must be made about the behaviour of radar systems [48]:

1. **No Interaction Between Receivers:** Each receiver operates independently, with no interaction between the signals received by different receivers. This assumption is typically valid for passive receivers that do not emit or absorb energy from the environment.

2. **No Interaction Between Targets:** The model assumes that there are no multiscatter returns, meaning reflections of energy from one target off another are not considered. Multipath propagation, however, may be considered separately.

3. **No Interaction Between Transmitters:** The transmitters are assumed not to interact with each other. This means that the electromagnetic waves emitted by one transmitter do not alter or absorb the waves from another transmitter. This assumption holds true in a vacuum, where Maxwell's equations are linear, and closely in air at all radar frequencies.

These assumptions are critical in simplifying the radar simulation model, making it possible to use the superposition principle to accurately model complex radar systems without introducing significant errors.

B.3.3 Discrete-Time Representation of Radar Signals

Accurately modelling radar signals in the digital domain is a cornerstone of radar simulation, particularly when modernising software like FERS. In transitioning from continuous analogue signals to discrete-time digital signals, several key considerations must be addressed to ensure that the simulated radar system accurately mirrors real-world behaviour. This is crucial for maintaining the integrity and performance of radar simulations, especially as we incorporate modern features and optimisations into the FERS software.

Discrete-Time Signal Model

Radar signals, which are originally continuous in time, must be sampled at discrete intervals to create a digital representation that can be processed by simulation software. The discrete-time signal $x[n]$ corresponds to the continuous-time signal $x(t)$ and is expressed as [48]:

$$x[n] \equiv x\left(\frac{n}{f_s}\right) \tag{B.13}$$

where f_s is the sampling frequency. For the simulation to accurately reflect real radar operations, f_s must satisfy the Nyquist criterion, ensuring it is at least twice the highest frequency present in the signal [49]. Failing to meet this criterion can lead to aliasing, where higher frequency components are misrepresented as lower frequencies, degrading the accuracy of the simulation.

Bandpass and Lowpass Signal Representation

Radar signals are typically bandpass signals, centred around a carrier frequency Ω_c. These signals can be more efficiently represented using the complex envelope $x_l(t)$, which simplifies the processing by allowing the signal to be treated at baseband rather than at the high carrier frequency. The complex envelope is represented as [48]:

$$x(t) = x_i(t)cos(\Omega_c t) - x_q(t)sin(\Omega_c t) \tag{B.14}$$

This approach reduces the computational load, a critical factor in real-time radar simulations where processing speed is essential.

Quantisation and Its Effects

Quantisation is the process of converting continuous signal amplitudes into a finite set of levels, introducing quantisation noise. The quantisation step Δ is defined as [48]:

$$\Delta = \frac{X_m}{2^B} \tag{B.15}$$

where X_m is the maximum signal amplitude, and B is the number of bits used for quantisation. The Signal-to-Noise Ratio (SNR) due to quantisation can be approximated as [48]:

$$SNR = 6.02(B-1) + 10.79 - 20\log_{10}\left(\frac{X_m}{\sigma_x}\right) dB \tag{B.16}$$

where σ_x is the Root Mean Square (RMS) amplitude of the signal. Proper selection of bit depth is crucial to ensuring that quantisation noise does not significantly degrade the signal's quality, which is especially important in radar systems where signal integrity directly impacts detection accuracy.

B.4 Conclusion

This appendix has provided a foundational overview of the key theoretical concepts underlying radar systems. We began with the principles of electromagnetic wave propagation and the radar equation, highlighting the factors that influence radar performance. The core components of radar systems—transmitter, antenna, receiver, signal processor, and display—were discussed, along with various radar configurations, including monostatic, bistatic, and multistatic systems.

We also touched on advanced concepts like the Doppler effect and the principle of superposition, which are critical for understanding and simulating complex radar systems. Additionally, we covered the importance of converting continuous radar signals into discrete-time representations for accurate digital simulation.