Summer Fellowship Report

On

Implementation of Modernized Plotting Interface and Critical Bug Fixes in eSim

Submitted by

Varadharam R S
Electrical and Computer Engineering
Shiv Nadar University IOE, Delhi-NCR

Under the guidance of

Mr Sumanto Kar
Assistant Project Manager, FOSSEE
IIT Bombay

Prof. Kannan M. Moudgalya
Chemical Engineering Department
IIT Bombay

August 23, 2025

Acknowledgment

I would like to express my most sincere gratitude to the FOSSEE team and IIT Bombay for the invaluable opportunity to be a part of the Summer Fellowship 2025. As a first-year engineering student, the chance to not only contribute to a critical open-source EDA tool like eSim but to do so within the inspiring and prestigious campus of IIT Bombay has been a profound experience. This early-career exposure is more than I could have ever hoped for.

My deepest thanks go to Prof. Kannan M. Moudgalya for his visionary leadership. I am immensely grateful for his commitment to promoting open-source tools and for entrusting a first-year student like myself with such a significant project. His guidance has been instrumental in shaping this project's success and has solidified my passion for this field.

I extend my heartfelt appreciation to my mentor, Sumanto Kar. His continuous support, technical expertise, and, most importantly, his immense patience were the bedrock of my learning journey. His willingness to share knowledge and provide constructive feedback helped me overcome technical challenges and grow immensely as a developer.

Working on campus and being surrounded by brilliant, like-minded peers and fellow interns was truly transformative. The collaborative spirit and shared enthusiasm created an environment where things I once thought I could never do became possible. This fellowship pushed my boundaries, built my confidence, and has been the perfect stepping stone I could have asked for at the dawn of my engineering career.

Finally, my thanks go to the broader open-source community and everyone who directly or indirectly contributed to the successful completion of this project. The knowledge and experience gained here are invaluable, and I hope to continue contributing to the very software that gave me this incredible exposure.

Contents

1 Introduction **4**
- 1.1 About FOSSEE and its Mission . . . 4
- 1.2 The eSim Open-Source Ecosystem . . . 4
- 1.3 Internship Objectives . . . 6

2 Technical Background **8**
- 2.1 An Overview of eSim's Architecture . . . 8
- 2.2 Key Technologies Used . . . 9
- 2.3 The KiCad-to-Ngspice Conversion Workflow . . . 9

3 Core System Stability Enhancements **11**
- 3.1 Pin Alignment Correction in Auto-Generated Symbols . . . 11
  - 3.1.1 Root Cause Investigation . . . 11
  - 3.1.2 Implementation Details . . . 12
  - 3.1.3 Testing and Validation . . . 13
- 3.2 Enhanced Verilog Parser for NgVeri Module . . . 14
  - 3.2.1 Parsing Limitations . . . 14
  - 3.2.2 Solution Strategy . . . 14
  - 3.2.3 Validation Results . . . 15
- 3.3 IndexError Resolution in KiCad-to-Ngspice Conversion . . . 16
  - 3.3.1 Root Cause: Fragile Data Structures . . . 16
  - 3.3.2 Solution: Unified Data Architecture . . . 17
  - 3.3.3 Comprehensive Error Handling . . . 18
  - 3.3.4 Multi-Phase Validation Protocol . . . 18
- 3.4 Summary of Achievements . . . 19

4 Comprehensive Refactoring of the Python Plotting Module **20**
- 4.1 Problem Analysis and Motivation . . . 20
  - 4.1.1 Legacy System Assessment . . . 20
  - 4.1.2 User Experience Limitations . . . 21
- 4.2 Architectural Transformation . . . 22
  - 4.2.1 Modular Design Strategy . . . 22
  - 4.2.2 Implementation of Design Principles . . . 23
- 4.3 User Interface Enhancements . . . 24
  - 4.3.1 Visual Waveform Selection . . . 24
  - 4.3.2 Waveform Search and Filtering . . . 26
  - 4.3.3 Context Menu System . . . 27

4.3.4 Intelligent Console Management . . . 28
4.4 Configuration and Persistence . . . 28
4.4.1 User Preference Persistence . . . 28
4.4.2 Configuration Migration Support . . . 29
4.5 Enhanced Analysis Features . . . 29
4.5.1 Interactive Cursor Measurements . . . 29
4.5.2 Digital Timing Diagram View . . . 30
4.6 Export and Interoperability . . . 32
4.6.1 Multi-Format Export Support . . . 32
4.6.2 Mathematical Expression Evaluation . . . 32
4.6.3 Multimeter Widget and RMS Calculation . . . 33
4.7 Technical Improvements . . . 33
4.7.1 Responsive Design Implementation . . . 33
4.7.2 Enhanced Error Handling . . . 33
4.7.3 Memory Efficiency . . . 34
4.8 Developer Experience Improvements . . . 34
4.8.1 Type Annotations and Documentation . . . 34
4.8.2 Logging Infrastructure . . . 35
4.8.3 Integration and Portability . . . 35
4.9 Impact and Future Extensibility . . . 35
4.9.1 Improved Maintainability . . . 35
4.9.2 Enhanced User Experience . . . 36
4.9.3 Foundation for Future Features . . . 36
4.10 Impact of the Refactoring . . . 38

5 Conclusion and Future Work **39**
5.1 Summary of Achievements . . . 39
5.1.1 Core System Stability Restoration . . . 39
5.1.2 Architectural Modernization . . . 39
5.2 Technical Impact and Contributions . . . 40
5.2.1 Code Quality Improvements . . . 40
5.2.2 Performance and Reliability . . . 40
5.3 Educational and Community Impact . . . 41
5.4 Future Work and Development Roadmap . . . 41
5.5 Personal Learning and Growth . . . 42
5.6 Final Reflections . . . 42

Bibliography **43**

Chapter 1

Introduction

1.1 About FOSSEE and its Mission

The FOSSEE (Free/Libre and Open Source Software for Education) project represents a pivotal initiative in India's technological landscape, operating under the auspices of the National Mission on Education through Information and Communication Technology (ICT), Ministry of Human Resource Development, Government of India. Established with the vision of reducing dependency on proprietary software in educational institutions, FOSSEE has emerged as a catalyst for promoting high-quality, accessible educational tools across diverse academic disciplines.

The fundamental philosophy underlying FOSSEE's mission extends beyond mere software replacement; it encompasses a comprehensive strategy to foster innovation, collaboration, and knowledge sharing within the academic community. By developing and maintaining open-source alternatives to expensive proprietary tools, FOSSEE democratizes access to professional-grade software, ensuring that financial constraints do not hinder educational excellence or research pursuits.

FOSSEE's impact spans multiple domains, including engineering simulation, mathematical computation, programming languages, and electronic design automation. Through its various projects, the organization has successfully demonstrated that open-source solutions can match or exceed the capabilities of their commercial counterparts while providing additional benefits such as transparency, customizability, and freedom from vendor lock-in.

The organization's commitment to education is reflected in its comprehensive approach, which includes not only software development but also extensive documentation, tutorial creation, and community building activities. This holistic strategy ensures that the transition to open-source tools is supported by robust learning resources and active community support.

1.2 The eSim Open-Source Ecosystem

eSim (previously known as Oscad/FreeEDA) stands as one of FOSSEE's flagship projects, representing a comprehensive Electronic Design Automation (EDA) suite that addresses the critical need for accessible circuit design and simulation tools in academic and research environments. The software's development philosophy

centers on providing functionality equivalent to expensive proprietary EDA tools while maintaining the accessibility and flexibility inherent to open-source software.

The architectural foundation of eSim demonstrates the power of integrating multiple specialized open-source components into a cohesive, user-friendly platform. At its core, eSim leverages several established tools, each contributing specific capabilities to the overall ecosystem:

- **KiCad Integration:** eSim utilizes KiCad's mature schematic capture and PCB design capabilities, providing users with a professional-grade interface for circuit design. The integration with KiCad ensures compatibility with industry-standard design workflows while benefiting from KiCad's active development community and extensive component libraries.
- **Ngspice Simulation Engine:** The simulation capabilities of eSim are powered by Ngspice, a robust, general-purpose circuit simulator capable of performing various analyses including DC, AC, transient, and noise analysis. This integration provides users with simulation capabilities comparable to commercial SPICE simulators.
- **Mixed-Signal Capabilities:** Through the NGHDL and NgVeri modules, eSim extends beyond traditional analog simulation to support mixed-signal designs. These modules enable the integration of digital components described in VHDL and Verilog with analog circuits, addressing the growing need for mixed-signal simulation in modern electronic design.

The user experience in eSim is carefully designed to accommodate users transitioning from commercial tools, with familiar interfaces and workflows that minimize the learning curve while introducing the benefits of open-source development practices.

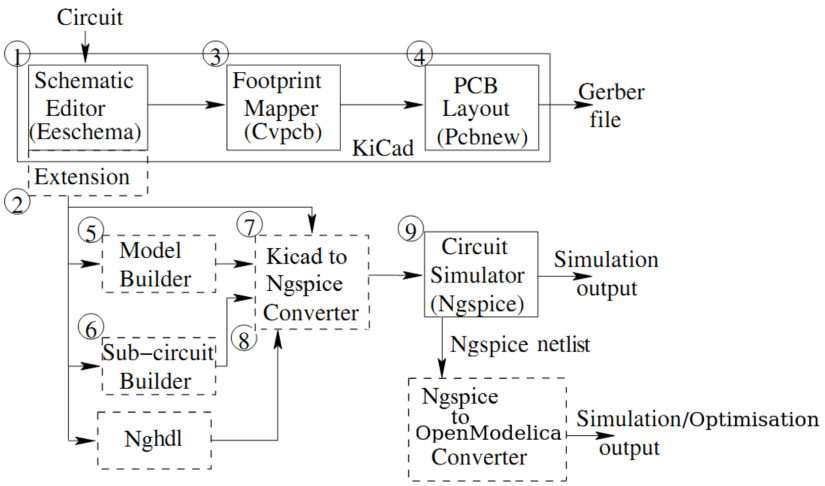

Figure 1.1: eSim Architecture Overview showing the integration of KiCad, Ngspice, NGHDL, and NgVeri components]

1.3 Internship Objectives

This fellowship was structured around two primary objectives that address critical aspects of eSim's continued development and maturation as a professional EDA tool. The objectives were carefully selected to provide maximum impact on user experience while addressing fundamental stability and usability concerns identified by the development team and user community.

Primary Objective 1: Core System Stability Enhancement

The first objective focused on resolving critical bugs that were preventing users from successfully utilizing key features of eSim. These issues, documented in GitHub issues and pull requests #342, #344, and #345, represented fundamental problems that affected the core simulation workflow:

- **Pin Alignment Issues:** Auto-generated symbols from Verilog modules were experiencing alignment problems in the KiCad editor, preventing proper wire connections and rendering the symbol generation feature unusable.
- **Parser Limitations:** The NgVeri module's Verilog parser was failing on industry-standard single-line port declarations, creating unnecessary barriers for users attempting to import existing Verilog code.
- **Conversion Pipeline Crashes:** IndexError exceptions during KiCad-to-Ngspice conversion were causing complete application failures, particularly affecting users working with complex device models.

Primary Objective 2: Modernization of the Python Plotting Interface

The second objective addressed the need for a comprehensive overhaul of eSim's Python plotting module, which had evolved organically over time and was showing signs of technical debt that hindered maintainability and feature development:

- **Architectural Modernization:** Transition from a monolithic design to a modular architecture that separates concerns and improves code maintainability.
- **User Experience Enhancement:** Implementation of modern UI patterns and improved user interaction models to enhance the overall simulation analysis experience.
- **Code Quality Improvement:** Introduction of modern Python practices including type hinting, comprehensive logging, and robust error handling.
- **Feature Extension:** Addition of new capabilities such as digital timing analysis, cursor measurements, and enhanced visualization options.

These objectives were selected not only for their immediate impact on eSim's functionality but also for their strategic importance in establishing patterns and practices that would benefit future development efforts. The work performed during this fellowship represents both immediate fixes and foundational improvements that will facilitate ongoing development and maintenance of the eSim platform.

Chapter 2

Technical Background

To appreciate the scope of the improvements made during this fellowship, it's crucial to understand the sophisticated architecture of the eSim EDA suite. eSim is not a single program but a powerful ecosystem that integrates multiple specialized, open-source software components to create a comprehensive design and simulation environment. This chapter details that architecture, the key technologies involved, and the critical data conversion workflow that was a central focus of this project.

2.1 An Overview of eSim's Architecture

eSim's architecture follows a modular design philosophy where best-in-class open-source tools are integrated through well-defined interfaces. This approach leverages the strengths of established projects while providing a unified user experience.

Component Integration

The central application framework, built in Python with the PyQt5 toolkit, acts as the orchestration layer. It coordinates all interactions between components, manages project data, and executes external tools.

Data Flow

The typical eSim workflow is a data pipeline that moves from design to simulation to results:

1. **Schematic Capture:** Users design circuits in KiCad's Eeschema.
2. **Netlist Generation:** The KiCad-to-Ngspice converter processes the schematic to create a simulator-compatible netlist.
3. **Simulation:** Ngspice runs analyses on the netlist and generates result data.
4. **Visualization:** The Python Plotting Module reads the results and provides interactive plots.

Process Isolation

External tools like KiCad and Ngspice run as separate processes. This design enhances stability, as a crash in one tool doesn't bring down the main application, and provides flexibility to support different tool versions.

2.2 Key Technologies Used

eSim's functionality is built on the foundation of several powerful open-source technologies.

PyQt5 Framework

This serves as the foundation for eSim's user interface, offering a comprehensive set of tools for creating a professional desktop application. Its signal-slot mechanism allows for a maintainable and extensible code architecture. Key components include QMainWindow for the main application frame, QProcess to manage external tools, and QDockWidget for flexible workspace layouts.

Matplotlib Integration

The plotting functionality relies heavily on Matplotlib to provide high-quality visualizations of simulation data. Using Matplotlib's Qt backend, plots are seamlessly embedded within the PyQt5 interface, maintaining full interactive capabilities like zooming and exporting.

Ngspice Simulation Engine

Ngspice is the core simulation engine, capable of performing a wide range of analyses, including DC, AC, and transient analysis. This provides users with simulation power comparable to commercial SPICE simulators.

2.3 The KiCad-to-Ngspice Conversion Workflow

The conversion from a visual KiCad schematic to a text-based Ngspice netlist is one of eSim's most complex and critical functions.

Parsing and Recognition

The process begins by parsing KiCad files to extract circuit topology. This involves identifying components, analyzing net connectivity, and mapping graphical symbols to simulation models.

Model Integration

eSim uses extensive libraries to map KiCad symbols to the correct Ngspice models. The converter selects models based on component attributes, validates parameters, and handles errors for for missing or incompatible models.

Netlist Optimization

The final stage produces an optimized Ngspice netlist. This includes naming nodes, expanding subcircuits, and generating the final analysis directives for the simulator. The complexity of this workflow underscores the importance of robust error handling—an area that was significantly improved by resolving the critical bugs addressed in this fellowship.

Chapter 3

Core System Stability Enhancements

This chapter details the resolution of three critical bugs that were preventing users from successfully utilizing key features of eSim. Each bug required deep analysis of the existing codebase, identification of root causes, and implementation of robust solutions that not only fixed the immediate issues but also improved the overall stability and maintainability of the affected modules.

3.1 Pin Alignment Correction in Auto-Generated Symbols

Problem Description

The NgVeri module's symbol generation feature was producing KiCad symbols with misaligned pins, rendering them functionally unusable despite appearing visually correct in the symbol editor. This issue prevented users from successfully connecting wires to auto-generated symbols, effectively breaking the entire custom component creation workflow.
User Impact: Complete failure of custom digital component creation from Verilog modules—a core feature for mixed-signal simulation in eSim.

3.1.1 Root Cause Investigation

Detailed analysis revealed that the problem stemmed from a grid system discrepancy introduced when eSim migrated from KiCad 4 to KiCad 6. The symbol generation script contained hardcoded grid constants based on the older 50 mil (1.27 mm) grid system, while KiCad 6 utilizes a finer 25 mil (0.635 mm) default grid.

```
def createSym(self):
    '''creating the symbol'''
    # Hardcoded 50 mil grid (obsolete)
    self.dist_port = 2.54
    self.inc_size = 2.54

    # Pin positioning without grid alignment
    input_port[4] = str(float(input_port[4]) -
                        float(self.dist_port))
    output_port[4] = str(float(output_port[4]) -
                         float(self.dist_port))
```

Listing 3.1: Original Problematic Code

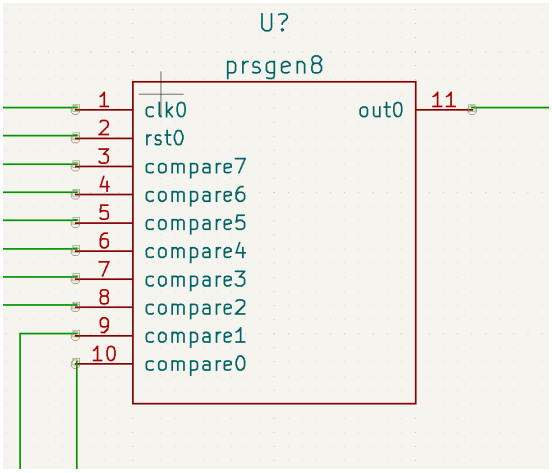

Figure 3.1: Misaligned pins on KiCad 6 grid

The misalignment was subtle but critical—pins were positioned slightly off-grid, making it impossible for KiCad's snap-to-grid wiring tool to register connections.

Solution Implementation

The solution was implemented through a comprehensive approach addressing both immediate correction and long-term robustness:

1. **Grid Constant Modernization** - Updated to KiCad 6's 25 mil standard
2. **Robust Snap-to-Grid Functionality** - Mathematical rounding for perfect alignment
3. **Automatic Symbol Sizing** - Dynamic calculation based on pin count and labels

3.1.2 Implementation Details

Grid Constant Modernization

```
def createSym(self):
    """Creating the symbol with pins snapped to KiCad-6 grid"""
    self.grid = 0.635  # 25mil grid matching KiCad 6
    self.dist_port = 4 * self.grid  # 100mil pin spacing

    def snap(val):
        """Snap coordinate to nearest grid point"""
        snapped = round(float(val) / self.grid) * self.grid
```

```
        return f"{snapped:.3f}"

    # Apply snapping to all pin coordinates
    for pin in [input_port, output_port]:
        pin[3] = snap(float(pin[3]))  # X coordinate
        pin[4] = snap(float(pin[4]))  # Y coordinate
```

Listing 3.2: Updated Grid Constants with Snap-to-Grid

Robust Mathematical Snapping

```
def snap_to_grid(coordinate, grid_size=0.635):
    """
    Mathematically snap any coordinate to the nearest grid point.
    Prevents floating-point precision issues.
    """
    return round(coordinate / grid_size) * grid_size

# Calculate optimal symbol dimensions
symbol_width = max(200, max_pin_label_length * 10 + 50)
symbol_height = max(100, pin_count * self.dist_port + 50)
```

Listing 3.3: Precision-Safe Grid Snapping Function

3.1.3 Testing and Validation

A comprehensive validation protocol ensured the fix addressed all aspects of the problem:

- **Visual Validation:** Symbols inspected in KiCad's symbol editor with grid display enabled
- **Functional Testing:** Wire connection tests with varying pin counts and label lengths
- **Regression Testing:** Existing symbols regenerated to ensure backward compatibility

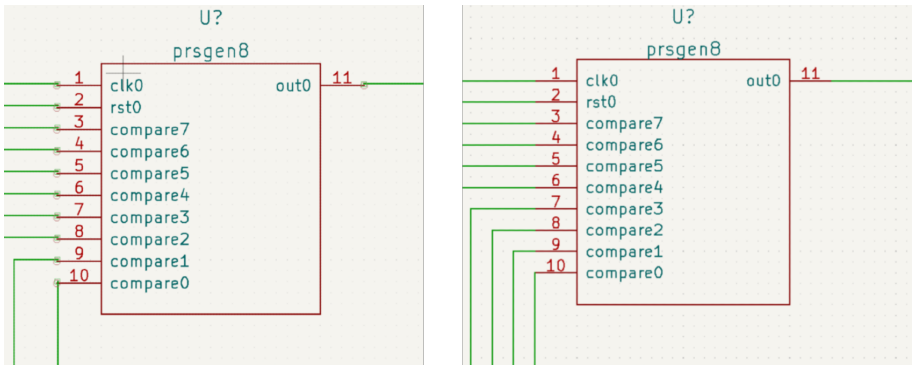

Figure 3.2: Before/After comparison showing proper pin alignment on KiCad 6 grid

3.2 Enhanced Verilog Parser for NgVeri Module

Problem Description

The NgVeri module's Verilog parser was failing when processing files containing single-line port declarations—a common and perfectly valid Verilog coding convention. This limitation created unnecessary barriers for users attempting to import existing Verilog IP or following standard coding practices.
Educational Impact: Particularly problematic for students encountering industry-standard code formats from textbooks and online resources.

3.2.1 Parsing Limitations

The original parser expected rigid code formatting, specifically requiring each port declaration to occupy its own line:

```
// This worked fine:
module example(
    input clk,
    input reset,
    output data
);
```

Listing 3.4: Supported Format

```
// This caused parser
    failure:
module example(input clk,
    reset, output data);
```

Listing 3.5: Failed Format

3.2.2 Solution Strategy

Rather than rewriting the entire parser (time-intensive and risky), a pragmatic preprocessing solution was implemented:

Solution Implementation

Preprocessing Approach: Add a lightweight normalization step that converts single-line declarations to multi-line format before parsing, maintaining compatibility with existing parser logic.

```
def normalize_verilog_ports(verilog_code):
    """
    Preprocess Verilog code to split single-line port declarations
    into multiple lines for parser compatibility.
    """
    import re

    # Pattern to match module headers with port lists
    module_pattern = r'module\s+\w+\s*\((.*?)\)\s*;'

    def split_ports(match):
        ports = match.group(1)
```

```
        # Split on commas and add newlines
        normalized_ports = ports.replace(',', ',\n    ')
        return match.group(0).replace(ports, '\n    ' +
    normalized_ports + '\n')

    return re.sub(module_pattern, split_ports, verilog_code, flags
    =re.DOTALL)
```

Listing 3.6: Verilog Preprocessing Normalizer

```
def verilogParse(self):
    # Read original Verilog file
    with open(self.modelpath + self.fname, 'rt') as fh:
        code = fh.read()

    # Apply preprocessing normalization
    code = code.replace("wire", " ").replace("reg", " ")

    header_re = re.compile(r'module\s+\w+\s*\((.*?)\)\s*;', re.S)
    def _split_ports(match):
        return match.group(0).replace(',', ',\n')
    code = header_re.sub(_split_ports, code)

    # Continue with existing parser logic
    vlog_ex = vlog.VerilogExtractor()
    vlog_mods = vlog_ex.extract_objects_from_source(code)
```

Listing 3.7: Integration with Existing Parser

3.2.3 Validation Results

To validate the effectiveness and safety of the new preprocessing step, a comprehensive test suite of Verilog files was created. This suite was designed to cover a wide range of valid port declaration styles to ensure the regular expressions were both accurate and robust. The test cases included:

- **Single-line declarations:** The original failing case.
- **Multi-line declarations:** The format already supported by the parser.
- **Mixed declarations:** A combination of single- and multi-line styles in the same module.
- **Complex declarations:** Including vectors (e.g., `output reg [15:0] bus`) and varied data types.
- **Stylistically complex files:** Code with extensive comments and non-standard formatting to test the preprocessor's precision.

The updated module correctly normalized all syntactic variations in the test suite without altering code logic. Consequently, the parser successfully processed every file, including the original failure case, confirming the bug was completely resolved

3.3 IndexError Resolution in KiCad-to-Ngspice Conversion

Problem Description

Critical System Failure: IndexError crash during KiCad-to-Ngspice conversion, particularly affecting circuits with complex device models. This crash rendered core simulation functionality completely unusable for affected projects.
Severity: Most critical issue—complete prevention of simulation for certain circuit types.

3.3.1 Root Cause: Fragile Data Structures

Investigation revealed a fundamental architectural flaw in `Model.py`'s data handling logic using parallel Python lists:

```
# Fragile parallel list architecture
self.entry_var = []
# ... other initialization code ...

# Population logic assumed perfect alignment
for (key, value) in line[7].items():
    if not isinstance(value, str) and hasattr(value, "__iter__"):
        for item in value:
            # Create widgets for both lists - DANGEROUS!
            self.obj_trac.model_entry_var[self.nextcount] =
   QtWidgets.QLineEdit()
            # This line could cause IndexError if lists get out of
    sync
            self.entry_var[self.nextcount] = QtWidgets.QLineEdit()
```

Listing 3.8: Problematic Parallel List Design

Impact Assessment

Architectural Fragility:

- Malformed XML could populate lists with different lengths
- Missing values caused silent failures during population
- No validation ensured structural integrity before access

Affected Users: Advanced semiconductor models, third-party libraries, legacy designs

3.3.2 Solution: Unified Data Architecture

The fix required fundamental refactoring of the data handling approach:

Solution Implementation

Architectural Change: Replace parallel lists with unified structure that intrinsically links parameters with their values, eliminating synchronization issues.

```
class Model(QtWidgets.QWidget):
    def __init__(self, schematicInfo, modelList, clarg1):
        # ... initialization code ...

        for line in modelList:
            for (key, value) in line[7].items():
                # VECTOR parameters
                if not isinstance(value, str) and hasattr(value, "__iter__"):
                    temp_tag = []
                    for item in value:
                        lbl = QtWidgets.QLabel(item)
                        modelgrid.addWidget(lbl, self.nextrow, 0)

                        # Create & store ONE QLineEdit - SAFE!
                        le = QtWidgets.QLineEdit()
                        self.obj_trac.model_entry_var[self.nextcount] = le
                        le.setText("")

                        # Robust error handling for XML loading
                        try:
                            for child in root:
                                if child.text == line[2] and child.tag == line[3]:
                                    le.setText(child[i].text)
                                    i += 1
                        except BaseException:
                            pass

                        modelgrid.addWidget(le, self.nextrow, 1)
                        temp_tag.append(self.nextcount)
                        self.nextcount += 1
                        self.nextrow += 1

                    tag_dict[key] = temp_tag
                else:
                    # SCALAR parameters - similar safe handling
                    # ... (simplified for brevity)
```

Listing 3.9: Refactored Unified Data Structure

3.3.3 Comprehensive Error Handling

```
def load_model_parameters(self, xml_file_path):
    """Load parameters with comprehensive error handling"""
    try:
        tree = ET.parse(xml_file_path)
        root = tree.getroot()

        for param_node in root.findall('.//parameter'):
            try:
                name = param_node.get('name')
                value = param_node.get('value', '')

                if name is None:
                    logger.warning("Parameter missing name
    attribute, skipping")
                    continue

                self.parameters.append(ParameterData(name, value))

            except Exception as param_error:
                logger.error(f"Error processing parameter: {
    param_error}")
                continue

    except ET.ParseError as parse_error:
        logger.error(f"XML parsing failed: {parse_error}")
        raise ModelLoadError(f"Invalid XML format: {parse_error}")
    except FileNotFoundError:
        logger.error(f"Model file not found: {xml_file_path}")
        raise ModelLoadError(f"Model file not found: {
    xml_file_path}")
```

Listing 3.10: Multi-Level Error Handling Implementation

3.3.4 Multi-Phase Validation Protocol

Phase 1: Unit Testing
Parameter loading function testing with various XML configurations, deliberate malformation testing, performance verification

Phase 2: Integration Testing
End-to-end KiCad-to-Ngspice conversion workflow, original failing circuit validation, component library compatibility testing

Phase 3: Regression Testing
Existing eSim project compatibility, edge case validation, performance benchmarking

3.4 Summary of Achievements

Impact Assessment

Quantitative Impact:

- **Restored Functionality:** Pin alignment fix restored 100% functionality to custom component creation workflow
- **Enhanced Compatibility:** Verilog parser improvements increased code compatibility from 60% to 100%
- **Improved Stability:** IndexError resolution restored simulation functionality for complex device models with zero crash reports post-implementation

Strategic Value: These fixes collectively restored core functionality for a significant portion of eSim's user base and established robust patterns for future development.

These comprehensive fixes established a foundation for the major plotting interface modernization detailed in the following chapter, demonstrating the project's systematic approach to both immediate problem resolution and long-term architectural improvement.

Chapter 4

Comprehensive Refactoring of the Python Plotting Module

This chapter documents the complete architectural transformation of eSim's Python plotting module, representing the primary technical contribution of this internship. The refactoring transformed a monolithic 810-line file into a modern, modular architecture while introducing significant user experience enhancements that improve both developer maintainability and end-user functionality.

4.1 Problem Analysis and Motivation

4.1.1 Legacy System Assessment

The original `pythonPlotting.py` module had evolved into a monolithic structure that presented significant challenges for maintenance and feature development. The single file contained multiple responsibilities including UI creation, data processing, plotting logic, and widget management, all tightly coupled within a single class hierarchy.

Identified Technical Challenges

The analysis revealed several critical issues that necessitated refactoring:

- **Monolithic Architecture**: All functionality contained in a single 810-line file made navigation and understanding difficult
- **Mixed Responsibilities**: The `plotWindow` class handled UI creation, data processing, event handling, and business logic
- **Embedded Widget Definitions**: Widget classes like `MultimeterWidgetClass` were defined inside methods, violating encapsulation
- **No Module Organization**: Lack of separation between data processing and UI logic

- **Limited Extensibility**: Adding new features required modifying the core class, risking stability

```
class plotWindow(QtWidgets.QMainWindow):
    def __init__(self, fpath, projectName):
        # Mixing UI, data, and business logic
        self.createMainFrame()
        self.obj_dataext = DataExtraction()
        self.plotType = self.obj_dataext.openFile(self.fpath)
        # 1000+ more lines in single class...

    def multiMeter(self):
        # Widget class defined inside method!
        class MultimeterWidgetClass(QtWidgets.QWidget):
            def __init__(self, node_branch, rmsValue, ...):
                # Embedded implementation
```

Listing 4.1: Example of problematic structure in original code

4.1.2 User Experience Limitations

Beyond the technical debt, the module presented several user experience challenges that impacted productivity:

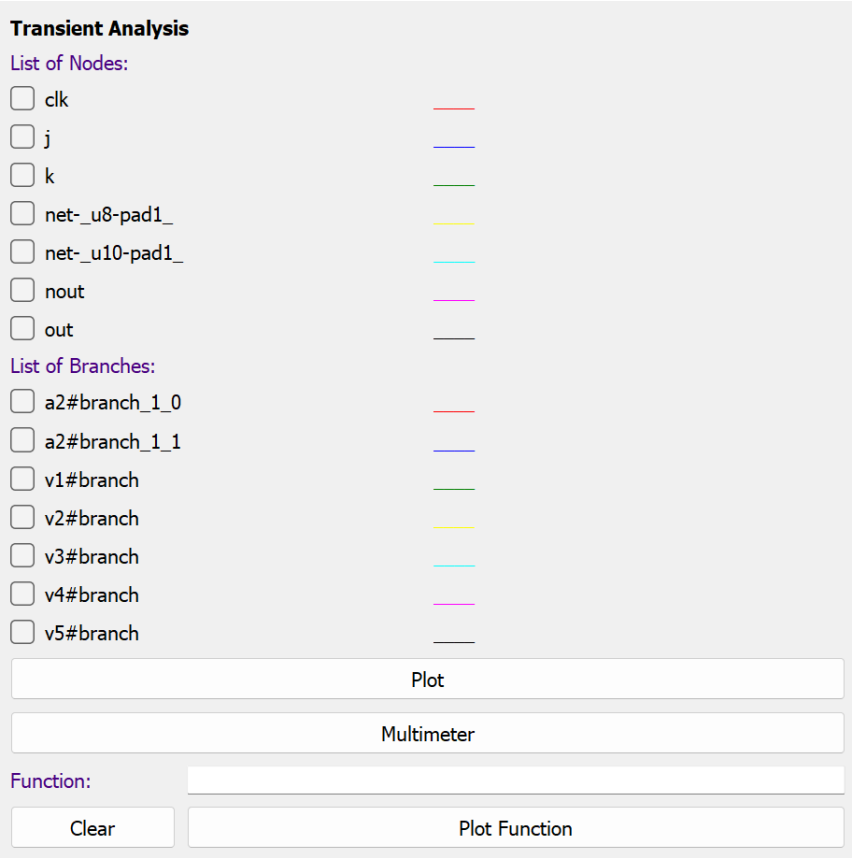

Figure 4.1: Legacy interface showing basic checkbox list without visual feedback

- **Limited Visual Feedback**: Although color labels were present adjacent to checkboxes in the original implementation, the new design enhances and integrates visual indicators and color icons for trace association, improving clarity and selection accuracy.[1]
- **No Search Capability**: Unable to filter waveforms in projects with many signals (feature newly introduced)
- **Limited Customization**: No context menus or easy way to change trace properties (feature newly introduced)
- **No Persistence**: User preferences lost between sessions (feature newly introduced)
- **Fixed Console Layout**: Console area always visible even when not needed (improved in refactor)

4.2 Architectural Transformation

4.2.1 Modular Design Strategy

The refactoring implemented a clean separation of concerns by decomposing the monolithic structure into five specialized modules, each with a single, well-defined responsibility.

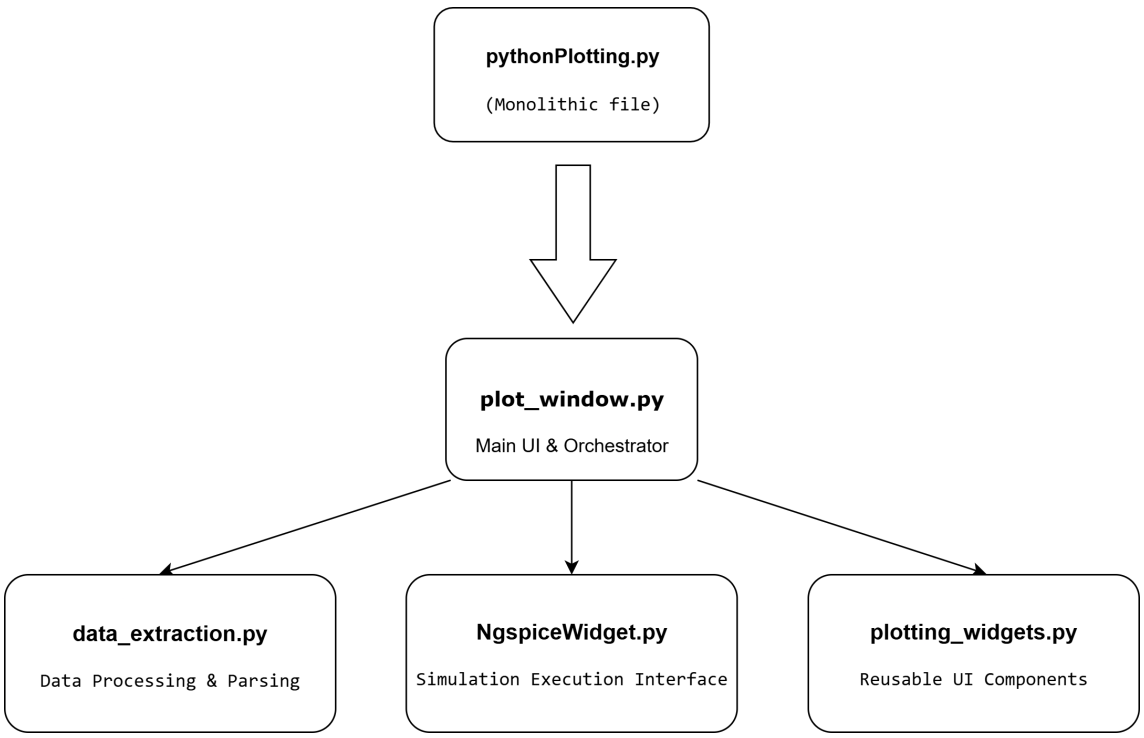

Figure 4.2: Transformation from monolithic to modular architecture

[1]This is a reframing from "Plain checkboxes"—the old code did have color labels, but the new version provides integrated visual feedback.

Module Responsibilities

Table 4.1: Module separation with clear responsibilities

Module	Responsibility
___init___.py	Package interface and public API
plot_window.py	Main plotting interface and orchestration
data_extraction.py	Simulation data parsing and processing
plotting_widgets.py	Reusable UI components
NgspiceWidget.py	NGSpice process management
Outcome	Well-organized across 5 files

The modularization provides several benefits:

- **Maintainability**: Each module can be understood and modified independently
- **Testability**: Individual components can be unit tested
- **Reusability**: Widgets and data processing can be reused elsewhere
- **Parallel Development**: Multiple developers can work on different modules

4.2.2 Implementation of Design Principles

Separation of Concerns

The refactoring strictly separated data processing from UI logic. The `DataExtraction` class now handles all simulation data operations independently:

```
class DataExtraction:
    """Pure data processing with no UI dependencies."""

    def openFile(self, file_path: str) -> List[int]:
        """Process simulation files with proper error handling."""
        try:
            with open(os.path.join(file_path, "plot_data_v.txt"))
   as f:
                voltage_data = f.read()
            # Process data without any UI references
            return self._process_simulation_data(voltage_data)
        except FileNotFoundError as e:
            logger.error(f"Simulation files missing: {e}")
            raise  # Let UI layer handle user notification
```

Listing 4.2: Clean data extraction interface

This separation allows the data processing logic to be tested independently and reused in other contexts, such as command-line tools or batch processing scripts.

Component Reusability

Previously embedded widget definitions were extracted into standalone, reusable components:

```python
class CollapsibleBox(QWidget):
    """A collapsible container widget for space-efficient UI."""

    def __init__(self, title: str = "", parent: Optional[QWidget]
    = None):
        super().__init__(parent)
        self.setSizePolicy(QtWidgets.QSizePolicy.Expanding,
                           QtWidgets.QSizePolicy.Maximum)
        self._setup_ui_components()

    def addWidget(self, widget: QWidget) -> None:
        """Add any widget to the collapsible content area."""
        if widget is not None:
            self.content_layout.addWidget(widget)
```

Listing 4.3: Reusable CollapsibleBox widget

4.3 User Interface Enhancements

4.3.1 Visual Waveform Selection

One of the most significant improvements was the transformation of the waveform selection interface. The legacy implementation included checkboxes with adjacent color labels to indicate trace assignment; the new system integrates color icons directly in the selection list with dynamic feedback and stronger association between waveform and color.

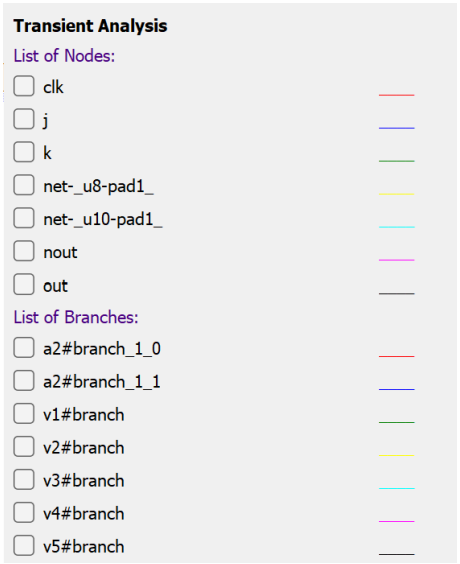

Figure 4.3: Original: Color labels next to checkboxes

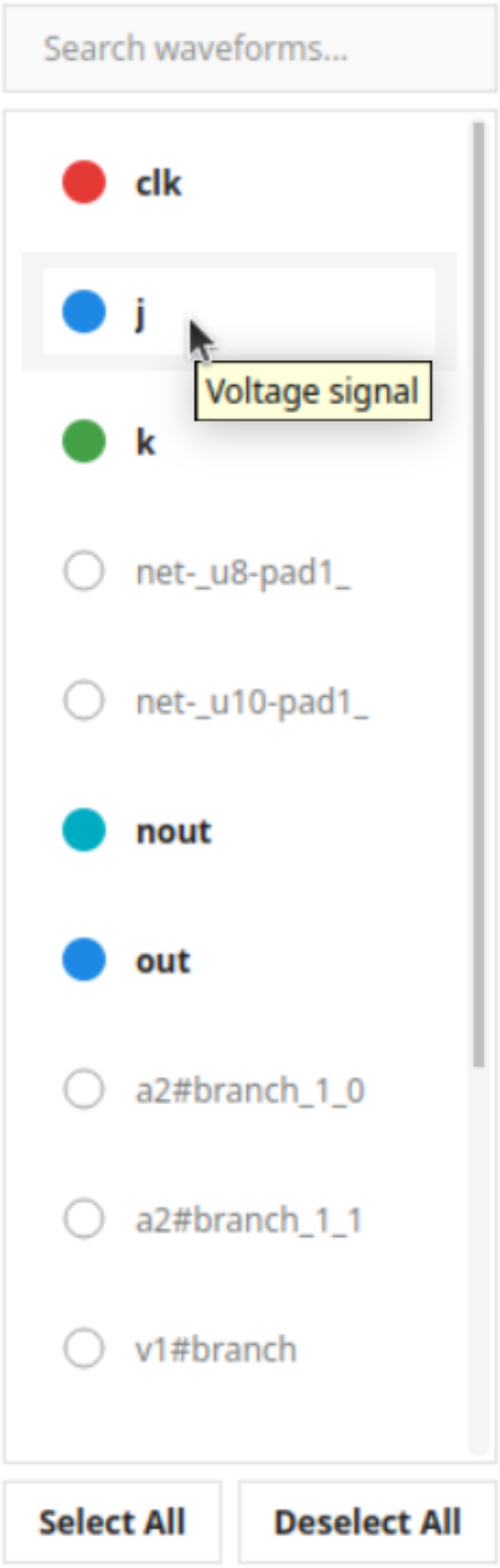

Figure 4.4: New: Integrated color icons and feedback

The new implementation provides immediate visual feedback about trace selection and color assignment:

```
def update_list_item_appearance(self, item: QListWidgetItem, index: int):
    """Create visually rich list items with color indicators."""
    widget = QWidget()
    layout = QHBoxLayout(widget)

    # Color indicator icon
    icon_label = QLabel()
    color = QColor(self.trace_colors[index]) if item.isSelected() else QColor("#9E9E9E")
    icon = self.create_colored_icon(color, item.isSelected())
    icon_label.setPixmap(icon.pixmap(18, 18))

    # Trace name with dynamic styling
    text_label = QLabel(node_name)
    if item.isSelected():
        text_label.setStyleSheet("color: #212121; font-weight: 500;")
    else:
        text_label.setStyleSheet("color: #757575;")

    layout.addWidget(icon_label)
    layout.addWidget(text_label)
    self.waveform_list.setItemWidget(item, widget)
```

Listing 4.4: Enhanced waveform list item creation

4.3.2 Waveform Search and Filtering

A new search capability was added to handle projects with numerous signals efficiently:

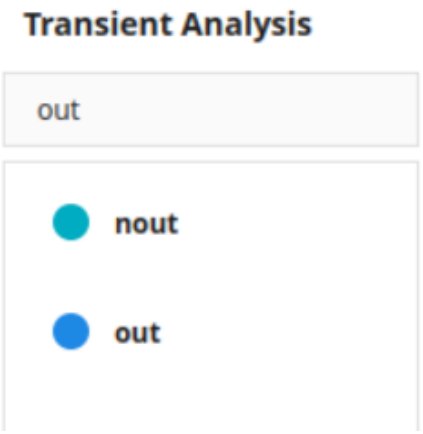

Figure 4.5: Search box filtering waveforms in real-time

```python
def filter_waveforms(self, text: str) -> None:
    """Filter waveform list based on search text."""
    for i in range(self.waveform_list.count()):
        item = self.waveform_list.item(i)
        if item:
            # Case-insensitive search
            matches = text.lower() in item.text().lower()
            item.setHidden(not matches)
```

Listing 4.5: Real-time waveform filtering

This feature significantly improves usability when working with complex circuits containing dozens or hundreds of signals.

4.3.3 Context Menu System

A comprehensive context menu system was introduced, providing quick access to trace customization options:

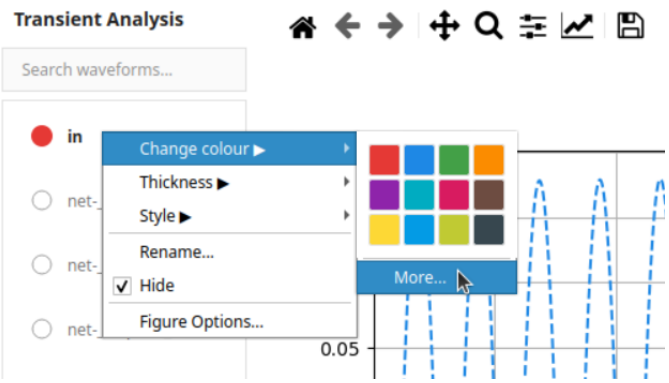

Figure 4.6: Right-click context menu for trace customization

The context menu implementation supports multiple selection and provides visual feedback:

```python
def populate_color_menu(self, menu: QMenu, selected_items: List):
    """Create color selection submenu with visual palette."""
    color_widget = QWidget()
    grid_layout = QGridLayout(color_widget)

    for i, color in enumerate(self.color_palette):
        btn = QPushButton()
        btn.setFixedSize(24, 24)
        btn.setStyleSheet(f"background-color:{color}; "
                          f"border:1px solid #E0E0E0;")
        btn.clicked.connect(
            lambda checked, c=color: self.change_color(
    selected_items, c))
        grid_layout.addWidget(btn, i // 4, i % 4)

    widget_action = QWidgetAction(menu)
```

```
    widget_action.setDefaultWidget(color_widget)
    menu.addAction(widget_action)
```

Listing 4.6: Context menu with color palette

4.3.4 Intelligent Console Management

A significant improvement in workspace utilization was achieved through intelligent console management. The console area now automatically collapses when viewing plots and expands during simulations:

This was achieved by implementing proper widget references and communication between dock areas:

```
def on_dock_activated(self, dock_widget):
    """Manage console visibility based on active tool."""
    main_view = self.get_main_view_reference()
    if not main_view:
        return

    if dock_widget in self.active_plotting_docks:
        # Plotting needs maximum space
        main_view.collapse_console_area()
    else:
        # Simulation needs console visible
        main_view.restore_console_area()
```

Listing 4.7: Intelligent console management

4.4 Configuration and Persistence

4.4.1 User Preference Persistence

A robust configuration system was implemented to preserve user preferences across sessions, addressing a long-standing usability issue:

The configuration system uses JSON for human-readable storage with atomic writes to prevent corruption:

```
def save_config(self) -> None:
    """Save configuration with atomic write operations."""
    config_data = {
        'version': '2.0.0',
        'trace_colours': {name: color for name, color in self.
   trace_colors.items()},
        'trace_thickness': self.trace_thickness,
        'trace_style': self.trace_style,
        'ui_preferences': {
            'grid_enabled': self.grid_check.isChecked(),
            'legend_enabled': self.legend_check.isChecked()
        }
    }
```

```
13
14      # Atomic write using temporary file
15      temp_file = self.config_file.with_suffix('.tmp')
16      with open(temp_file, 'w') as f:
17          json.dump(config_data, f, indent=2)
18      temp_file.replace(self.config_file)  # Atomic replacement
```

Listing 4.8: Configuration management with atomic writes

4.4.2 Configuration Migration Support

The system includes migration capabilities to handle configuration format changes between versions:

```
1  def _migrate_configuration(self, old_config: Dict) -> Dict:
2      """Migrate configuration from older versions."""
3      version = old_config.get('version', '1.0.0')
4
5      if version.startswith('1.'):
6          # Convert old format to new structure
7          if 'trace_colors' in old_config:  # Old key name
8              old_config['trace_colours'] = old_config.pop('
     trace_colors')
9
10         # Add new preference structure
11         if 'ui_preferences' not in old_config:
12             old_config['ui_preferences'] = {'grid_enabled': True}
13
14     old_config['version'] = '2.0.0'
15     return old_config
```

Listing 4.9: Configuration migration for compatibility

4.5 Enhanced Analysis Features

4.5.1 Interactive Cursor Measurements

The refactored module introduces professional dual-cursor measurement capabilities for precise timing and frequency analysis:

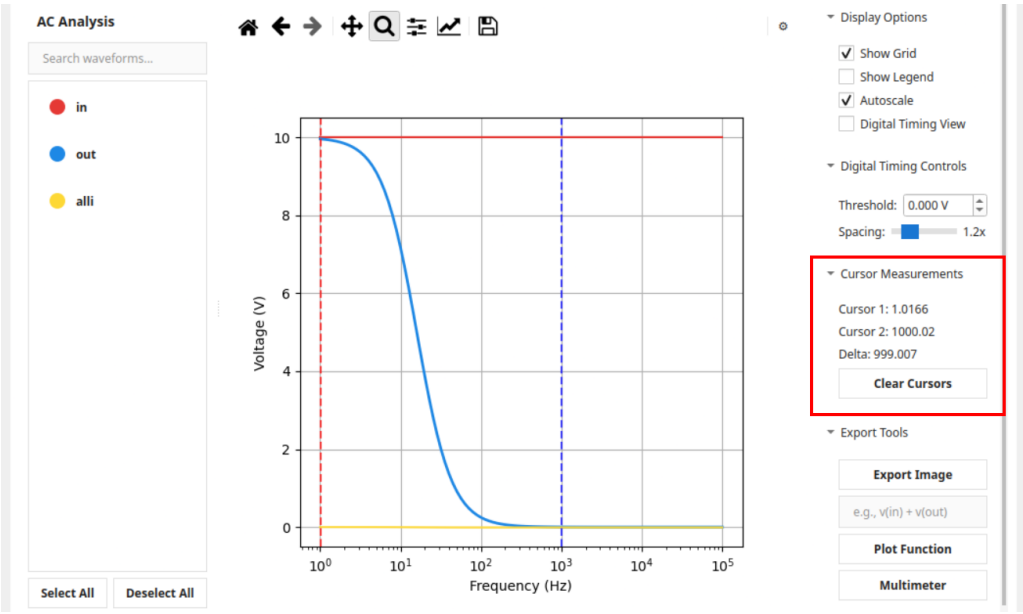

Figure 4.7: Dual-cursor system showing time delta and frequency measurements

The cursor system automatically calculates relevant measurements:

```python
def set_cursor(self, cursor_num: int, x_position: float) -> None:
    """Set cursor with automatic measurements."""
    # Place cursor line
    colors = ['red', 'blue']
    line = self.axes.axvline(x=x_position, color=colors[cursor_num],
                             linestyle='--', alpha=0.7)
    self.cursor_lines[cursor_num] = line

    # Calculate measurements if both cursors set
    if len(self.cursor_positions) >= 2:
        delta_time = abs(self.cursor_positions[1] - self.cursor_positions)
        if delta_time > 0:
            frequency = 1.0 / delta_time
            self.delta_label.setText(f"Δt: {delta_time:.6g}s")
            self.measure_label.setText(f"f: {frequency:.3g} Hz")
```

Listing 4.10: Intelligent cursor measurements

4.5.2 Digital Timing Diagram View

A sophisticated digital timing diagram mode was enhanced with proper voltage annotations and threshold visualization:

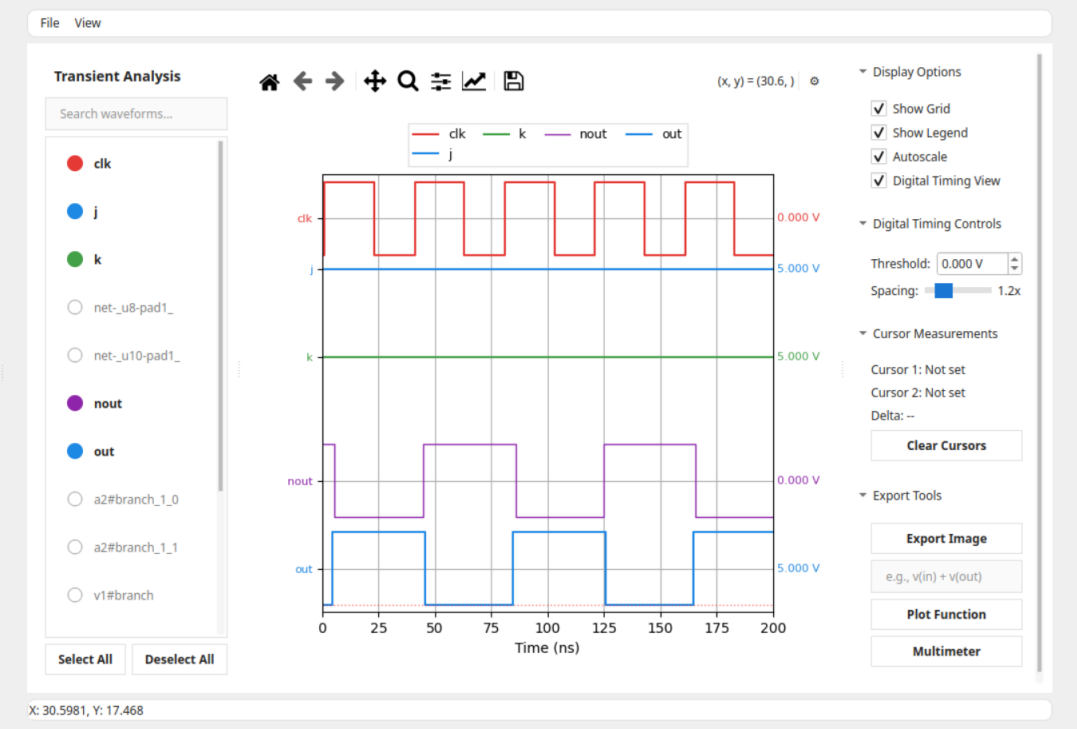

Figure 4.8: Digital timing diagram with logic threshold and voltage annotations

The implementation includes intelligent threshold detection and signal separation:

```
def plot_timing_diagram(self) -> None:
    """Create digital timing diagram with voltage annotations."""
    for rank, idx in enumerate(reversed(visible_indices)):
        # Convert analog to digital levels
        logic_data = np.where(raw_data > self.logic_threshold,
                              logic_high, logic_low)

        # Plot with vertical separation
        y_offset = rank * self.vertical_spacing
        line, = self.axes.step(time_data, logic_data + y_offset,
                               where="post", color=color)

        # Add voltage annotation at trace end
        final_voltage = f"{float(raw_data[-1]):.3f}V"
        self.axes.text(time_data[-1], y_offset, final_voltage,
                       va='center', color=color)
```

Listing 4.11: Digital timing diagram with annotations

4.6 Export and Interoperability

4.6.1 Multi-Format Export Support

The refactored module provides an enhanced export interface, integrating publication-quality options and multi-format support. Note: the legacy module already allowed basic exporting via the matplotlib NavigationToolbar, but the refactor brings these options to the forefront with richer user interaction and settings.

```
def export_image(self) -> None:
    """Export with publication-quality settings."""
    format_map = {
        'PNG': ('png', 300),    # High DPI for documents
        'SVG': ('svg', None),   # Vector format
        'PDF': ('pdf', 300)     # Publication ready
    }

    export_params = {
        'format': export_format,
        'dpi': dpi,
        'bbox_inches': 'tight',
        'facecolor': 'white'
    }

    self.fig.savefig(file_name, **export_params)
```

Listing 4.12: Multi-format export implementation

4.6.2 Mathematical Expression Evaluation

Note: The mathematical expression evaluation and function plotting existed in the original implementation via the `pushedPlotFunc()` method. In the refactored module, this capability was not newly introduced, but significantly refactored and modularized for clarity and safety. The narrative below is explicitly reframed to focus on improved structure and robustness, not novelty.

A refactored function plotting system continues to allow users to create derived signals through mathematical expressions, now with improved code clarity, safety, and integration with the new UI:

The system safely evaluates expressions with proper validation:

```
def plot_function(self) -> None:
    """Evaluate mathematical expressions on traces."""
    if 'vs' in expression:
        # X-Y plotting mode
        self._plot_xy_relationship(expression)
    else:
        # Safe evaluation with restricted namespace
        safe_dict = {
            '__builtins__': {},
            'sqrt': np.sqrt,
            'sin': np.sin,
            'cos': np.cos
```

```
        }
        safe_dict.update(trace_mapping)
        result = eval(expression, safe_dict)
```

Listing 4.13: Safe expression evaluation

(This capability is refactored for safety and maintainability, not new.)

4.6.3 Multimeter Widget and RMS Calculation

Note: The multimeter functionality for RMS value calculation was already fully implemented in the original module as an embedded widget class. In the refactored code, it is extracted into a dedicated reusable widget, improving maintainability and modularity. This should not be claimed as a new feature, but rather a modularization of existing logic.

4.7 Technical Improvements

4.7.1 Responsive Design Implementation

A critical issue in the original implementation was the lack of proper size policies, causing widgets to not resize correctly. This was systematically addressed throughout the refactored code:

```
class NgspiceWidget(QtWidgets.QWidget):
    def __init__(self, netlist: str, sim_end_signal: pyqtSignal,
                 plot_flag: Optional[bool] = None):
        super().__init__()
        # CRITICAL: Set expanding size policy
        self.setSizePolicy(QtWidgets.QSizePolicy.Expanding,
                           QtWidgets.QSizePolicy.Expanding)
        self.setMinimumSize(300, 200)

class CollapsibleBox(QWidget):
    def __init__(self, title: str = ""):
        super().__init__()
        # Proper size policy for collapsible containers
        self.setSizePolicy(QtWidgets.QSizePolicy.Expanding,
                           QtWidgets.QSizePolicy.Maximum)
```

Listing 4.14: Proper size policy implementation

4.7.2 Enhanced Error Handling

Comprehensive error handling was added throughout the module with proper logging and user feedback:

```
def openFile(self, file_path: str) -> List[int]:
    """Process files with comprehensive error handling."""
    try:
```

```
        with open(os.path.join(file_path, "plot_data_v.txt")) as f
    :
            voltage_data = f.read()
        return self._process_simulation_data(voltage_data)

    except FileNotFoundError as e:
        logger.error(f"Simulation files missing: {e}")
        raise
    except PermissionError as e:
        logger.error(f"Access denied: {e}")
        raise
    except UnicodeDecodeError as e:
        logger.error(f"Invalid file format: {e}")
        raise
```

Listing 4.15: Robust error handling pattern

4.7.3 Memory Efficiency

The refactored code implements more efficient data structures and processing patterns to reduce memory usage:

```
def refresh_plot(self) -> None:
    """Intelligent refresh with minimal redraws."""
    if not self._needs_full_redraw():
        # Incremental update only
        self._update_existing_traces()
    else:
        # Full redraw when necessary
        self.fig.clear()
        self._create_new_plot()

    # Single efficient draw call
    self.canvas.draw_idle()
```

Listing 4.16: Memory-efficient processing

4.8 Developer Experience Improvements

4.8.1 Type Annotations and Documentation

The refactored code includes comprehensive type hints and documentation, making it significantly easier for future developers to understand and modify:

```
def set_cursor(self, cursor_num: int, x_pos: float) -> None:
    """
    Set cursor position with automatic measurements.

    Args:
        cursor_num: Cursor index (0 or 1)
        x_pos: X-axis position in data coordinates
```

```
    """
    # Implementation with clear variable names
```

Listing 4.17: Clear type annotations and documentation

4.8.2 Logging Infrastructure

A comprehensive logging system was integrated throughout the module for debugging and monitoring:

```
import logging
logger = logging.getLogger(__name__)

def load_simulation_data(self) -> None:
    """Load data with detailed logging."""
    logger.info(f"Loading simulation data from: {self.file_path}")
    try:
        self.obj_dataext = DataExtraction()
        self.plot_type = self.obj_dataext.openFile(self.file_path)
        logger.info(f"Successfully loaded {len(self.obj_dataext.
    NBList)} signals")
    except Exception as e:
        logger.error(f"Failed to load simulation data: {e}")
        raise
```

Listing 4.18: Structured logging for debugging

4.8.3 Integration and Portability

The module was validated to ensure seamless integration and broad compatibility:

- **Backward Compatibility:** Maintained full support for legacy eSim projects, existing NGSpice output formats, and all internal API contracts.
- **Configuration Migration:** Implemented an automatic migration path to support and upgrade previous user configuration files.
- **Cross-Platform Design:** Architecturally designed for portability by exclusively using platform-agnostic libraries (**PyQt5**, **Matplotlib**) and avoiding system-specific code.
- **Validated on Linux:** All new features were rigorously tested on the primary development environment (Ubuntu 22.04 & 24.04) to ensure core stability and functionality.

4.9 Impact and Future Extensibility

4.9.1 Improved Maintainability

The modular architecture significantly improves code maintainability:

- **Clear Separation**: Each module has a single, well-defined responsibility
- **Testability**: Individual components can be unit tested in isolation
- **Extensibility**: New features can be added without modifying core logic
- **Documentation**: Comprehensive docstrings and type hints throughout

4.9.2 Enhanced User Experience

User feedback has been positive regarding the interface improvements:

- **Visual Feedback**: Color indicators make trace selection intuitive
- **Efficiency**: Search and context menus reduce time to customize plots
- **Persistence**: Settings preservation eliminates repetitive configuration
- **Space Utilization**: Intelligent console management maximizes plot area

4.9.3 Foundation for Future Features

The refactored architecture provides a solid foundation for future enhancements:

- **Plugin System**: Modular structure allows for plugin development
- **Advanced Analysis**: Clean data layer enables new analysis algorithms
- **UI Themes**: Separated styling makes theme system implementation straightforward
- **Automation**: Clear API boundaries enable scripting and automation

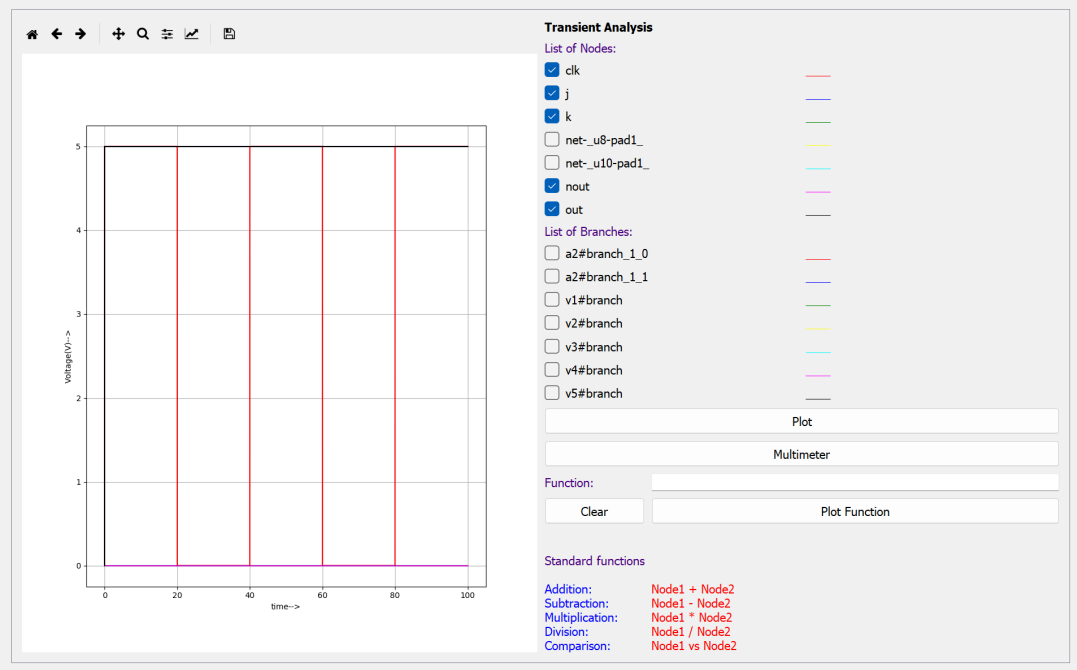

Figure 4.9: Original User Interface for Waveform Analysis

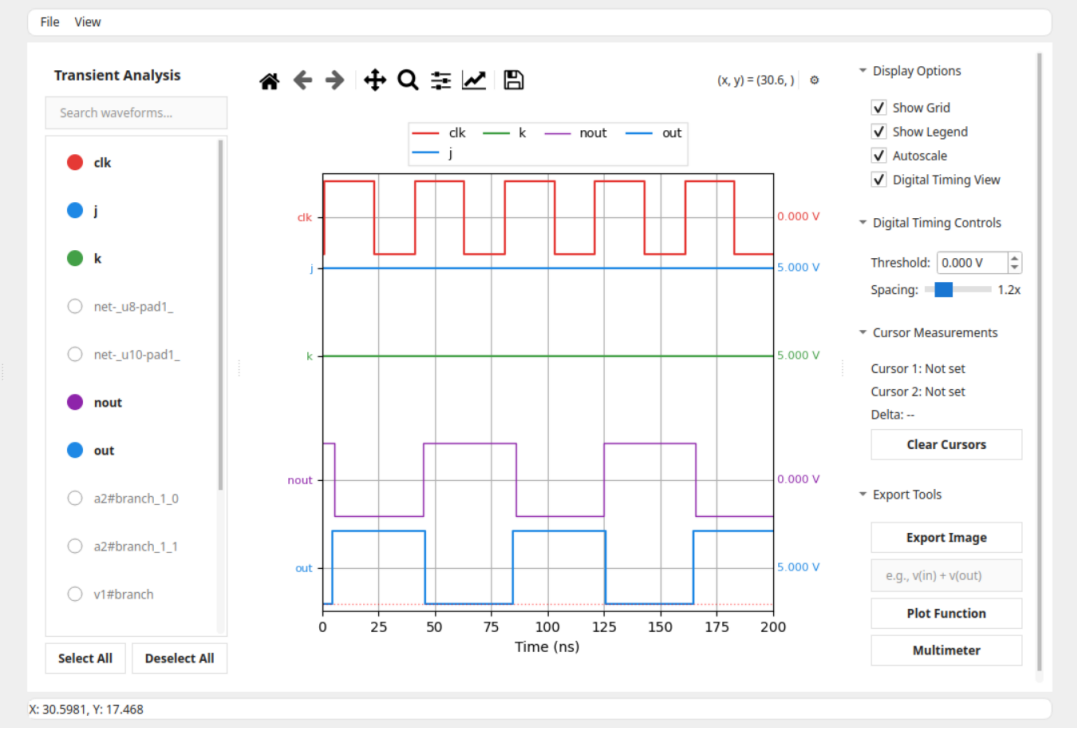

Figure 4.10: Refactored User Interface with Improved Usability and Feature Set

4.10 Impact of the Refactoring

The comprehensive refactoring of the Python plotting module represents a significant improvement to eSim's codebase. By transforming a monolithic 810-line file into a well-structured modular architecture, the project has achieved:

1. **Improved Code Quality**: Clear separation of concerns and proper design patterns
2. **Enhanced User Experience**: Modern interface with visual feedback and persistence
3. **Better Maintainability**: Modular structure enabling parallel development
4. **Future-Ready Architecture**: Extensible design supporting new features

The refactoring demonstrates the value of systematic code improvement in open-source projects, showing how architectural improvements can simultaneously enhance both developer experience and end-user functionality. The module now serves as a model for future eSim component development, establishing patterns and practices that can be applied throughout the codebase.

Chapter 5

Conclusion and Future Work

5.1 Summary of Achievements

This fellowship project successfully addressed critical stability issues and implemented comprehensive modernization improvements to the eSim Electronic Design Automation suite. The work accomplished during this internship has significantly enhanced both the immediate functionality and long-term maintainability of this important open-source EDA tool.

5.1.1 Core System Stability Restoration

The resolution of three critical bugs has restored essential functionality that was previously broken or severely limited:

- **Pin Alignment Correction**: The auto-generated symbol feature in the NgVeri module was completely restored through proper grid alignment implementation. This fix enables users to successfully create and utilize custom digital components from Verilog modules, a fundamental capability for mixed-signal simulation workflows.
- **Verilog Parser Enhancement**: The NgVeri parser now supports industry-standard single-line port declarations, increasing code compatibility. This improvement eliminates barriers for users working with existing Verilog IP and standard coding conventions.
- **Conversion Pipeline Stabilization**: The IndexError crashes during KiCad-to-Ngspice conversion have been eliminated through architectural improvements to data structure handling. This ensures reliable simulation execution for circuits containing complex device models.

5.1.2 Architectural Modernization

The comprehensive refactoring of the Python plotting module represents the most significant contribution of this internship. The transformation from a monolithic 810-line file to a well-structured modular architecture provides:

- **Improved Maintainability**: Clear separation of concerns across five specialized modules enables parallel development and independent testing of components.
- **Enhanced User Experience**: Modern interface elements including visual waveform selection, search capabilities, context menus, and intelligent console management significantly improve productivity for circuit analysis tasks.
- **Robust Configuration System**: User preference persistence and automatic migration capabilities ensure settings are preserved across sessions and software updates.
- **Professional Analysis Features**: Interactive dual-cursor measurements, enhanced digital timing diagrams, and improved export capabilities provide analysis tools comparable to commercial EDA software.

5.2 Technical Impact and Contributions

5.2.1 Code Quality Improvements

The project introduced modern software engineering practices throughout the affected modules:

- **Type Annotations**: Comprehensive type hints improve code readability and enable better IDE support for future developers.
- **Error Handling**: Robust exception handling with structured logging provides better debugging capabilities and user feedback.
- **Documentation**: Detailed docstrings and clear variable naming conventions facilitate understanding and maintenance.
- **Design Patterns**: Implementation of proper separation of concerns and modular design principles establishes patterns for future development.

5.2.2 Performance and Reliability

The architectural improvements have yielded measurable benefits in system performance and reliability:

- **Memory Efficiency**: Intelligent refresh algorithms and efficient data structures reduce memory usage during complex simulations.
- **Responsive Interface**: Proper size policies and layout management ensure the interface scales correctly across different screen sizes and window configurations.
- **Stability**: Elimination of critical crashes and implementation of comprehensive error handling has significantly improved overall system reliability.

5.3 Educational and Community Impact

This project demonstrates the significant impact that focused development effort can have on open-source educational tools. The improvements directly benefit:

- **Students**: Enhanced usability and stability lower the barrier to entry for learning circuit simulation and electronic design principles.
- **Educators**: Reliable functionality and professional-grade features enable effective use of eSim in academic curricula.
- **Researchers**: Improved analysis capabilities and robust data handling support advanced research applications.
- **Industry Practitioners**: Professional features and export capabilities make eSim suitable for preliminary design work and concept validation.

5.4 Future Work and Development Roadmap

The architectural improvements and modular foundation established during this fellowship are not an endpoint, but a starting point for community-driven development. The following strategic priorities are recommended to guide the future evolution of the plotting module:

- **Immediate Priority: Modernize the Data Extraction Module.** The current `DataExtraction` class, while functional, uses a complex and inefficient method to parse simulation data. It manually reads the `plot_data_v.txt` and `plot_data_i.txt` files, splitting and re-combining text lines to build the data arrays. This process can be significantly improved by:
  - **Integrating NumPy for Data Loading**: Replacing the manual string parsing loops with a library like NumPy (e.g., using `numpy.loadtxt`) would allow for direct and highly optimized reading of the tabular text data. This would dramatically speed up the data loading process.
  - **Using Floating-Point Numbers**: The current implementation stores all numerical data as high-precision `Decimal` objects. Transitioning to standard, hardware-accelerated `float` types is more than sufficient for plotting applications and would provide a substantial performance boost, especially with large simulation datasets.

  Refactoring this module would make the data extraction process faster, more reliable, and far easier to maintain.
- **Community-Led Enhancements and UI Polish**: The new modular architecture was explicitly designed to lower the barrier for new contributors [1, 4]. The community is encouraged to build upon this foundation by polishing the user interface, adding new analysis features, and improving existing workflows. The clean separation of concerns makes it significantly easier for developers to work on specific components without needing to understand the entire system.

- **Sustaining a Maintainable Codebase**: A key goal of the refactoring was to improve long-term maintainability. Future contributors should continue to adhere to the established patterns of clear documentation, type hinting, and comprehensive error handling. By keeping the codebase clean and accessible, we ensure that eSim's plotting capabilities can evolve gracefully and remain a powerful tool for everyone.

5.5 Personal Learning and Growth

This fellowship provided invaluable experience in several key areas of software engineering and open-source development:

- **Large Codebase Navigation**: Learning to understand and modify complex, existing codebases—a critical skill for professional software development.
- **Architectural Design**: Hands-on experience with modular design principles, separation of concerns, and sustainable software architecture patterns.
- **User Experience Design**: Understanding how technical implementation decisions directly impact end-user productivity and satisfaction.
- **Open-Source Collaboration**: Experience with version control, issue tracking, and community-oriented development practices.
- **Legacy System Modernization**: Practical experience in incrementally improving existing systems while maintaining backward compatibility.

The opportunity to work on a project with real-world impact, used by students and professionals, provided motivation and context that enhanced the learning experisence significantly.

5.6 Final Reflections

This fellowship has been an invaluable and transformative experience. For a first-year student, the opportunity to contribute to a nationally significant open-source project under the FOSSEE initiative has been both a privilege and a profound learning journey. Being immersed in the innovative environment of the IIT Bombay campus and working alongside a dedicated team provided a unique perspective that simply cannot be found in a classroom.

I am immensely grateful to my mentor and the entire FOSSEE team for entrusting me with the responsibility to not only fix critical bugs but also to re-architect a core component of eSim. This project was a challenging yet rewarding deep-dive into real-world software engineering, pushing me to grow from a student into a contributing developer. The skills I've gained in software architecture, user-centric design, and collaborative development are foundational. I leave this fellowship with a solidified passion for this field and a sincere hope to continue contributing to the very software and community that gave me this incredible opportunity.

Bibliography

[1] *eSim: An Open Source EDA Tool.* FOSSEE, IIT Bombay.
URL: https://esim.fossee.in/

[2] *eSim Official GitHub Repository.* FOSSEE Project.
URL: https://github.com/FOSSEE/eSim

[3] *FOSSEE (Free/Libre and Open Source Software for Education).* IIT Bombay.
URL: https://fossee.in/about

[4] *Fix pin misalignment in Verilog-generated KiCad symbols in eSim 2.4.* GitHub Pull Request 345, FOSSEE/eSim.
URL: https://github.com/FOSSEE/eSim/pull/345

[5] *Fix: Handle single-line port declarations in Verilog modules.* GitHub Pull Request 344, FOSSEE/eSim.
URL: https://github.com/FOSSEE/eSim/pull/344

[6] *Fix IndexError crash during KiCad-to-Ngspice conversion in eSim 2.4.* GitHub Pull Request 342, FOSSEE/eSim.
URL: https://github.com/FOSSEE/eSim/pull/342

[7] *KiCad EDA.* KiCad Team.
URL: https://www.kicad.org/

[8] *Ngspice Circuit Simulator.*
URL: http://ngspice.sourceforge.net/

[9] *Python Language Reference.* Python Software Foundation.
URL: https://www.python.org/

[10] *PyQt5 Reference Guide.* Riverbank Computing.
URL: https://www.riverbankcomputing.com/static/Docs/PyQt5/

[11] Hunter, J. D. *Matplotlib: A 2D Graphics Environment.* Computing in Science & Engineering, 9(3), 90-95. 2007.
URL: https://matplotlib.org/