MathPlot — Advanced Software Engineering Exam Documentation

Submission Date: December 5, 2025
Student: Nicole Tiokhin

Introduction

MathPlot is a Java-based mathematical function plotter developed as part of the Advanced Software Engineering exam. The project began from a partially implemented skeleton that included a JavaFX front-end, a Plotter interface, and two expression parsers. The MathPlot class itself was mostly empty and allowed modifications only in explicitly marked sections.

The completed implementation supports:

- Parsing expressions in both AOS (infix) and RPN (postfix) formats
- Constructing an internal abstract syntax tree (AST) to represent expressions
- Evaluating functions numerically for any input value
- Performing symbolic first-order differentiation
- Simplifying both the original function and its derivative
- Printing expressions cleanly in both AOS and RPN formats
- Plotting functions and their derivatives in Cartesian coordinates
- Plotting expressions in Polar coordinates as an optional enhancement
- Computing the area under the curve using both Rectangular and Trapezoidal integration methods
- Comprehensive unit testing achieving well above the required 80% coverage

All work strictly follows the exam requirement that only modifications inside the allowed regions of MathPlot.java are permitted.

Requirements Overview

Functional Requirements

MathPlot satisfies all required functionality:

- Expression input in AOS or RPN
- Support for operators `+, -, *, /, ^`
- Support for functions `sin`, `cos`, `exp`, and `ln`
- Correct computation of first-order derivatives
- Printing of both the function and its derivative
- Cartesian plotting (mandatory)
- Optional Polar plotting
- Numerical area computation
- Expression simplification

All requirements were fulfilled, including multiple optional features.

Minimum Acceptance Criteria Compliance

The project meets or exceeds all minimum acceptance criteria:

Requirement	Status
One input format	Implemented (AOS and RPN both implemented)
Operators + - * / ^	Fully supported
One built-in function	Implemented (sin, cos, exp, ln)
One plot type	Cartesian plotting completed
Reference axes and grid	Drawn automatically by Plotter
Printing of f and f′	Implemented in AOS and RPN formats
First-order derivative	Fully implemented using symbolic rules
Test coverage ≥ 80%	Achieved (above 90%)
Extensibility without redesign	Confirmed

Optional features implemented:

- Both input formats
- Both plot formats
- Both area integration methods
- Expression simplifier

Architecture and Design

MathPlot is organized so that each part of the system has a clear responsibility. The MathPlot class works as a central controller. It receives an expression from the user, parses it, builds an internal representation, evaluates it, differentiates it, simplifies it, prints it, and sends sampled points to the Plotter for drawing.

The structure is divided into the following main components:

- **Parsers (AOS and RPN):** Convert the input string into tokens that can be used to build expressions. These files were provided and not modified.
- **Expression Tree (AST):** Every mathematical expression is stored as an abstract syntax tree composed of nodes such as constants, variables, unary functions, and binary operations.
- **Evaluation:** The AST can compute f(x) for any numerical value.
- **Differentiation:** The AST can compute f′(x) symbolically using standard calculus rules.
- **Simplification:** Expressions are cleaned up by removing unnecessary operations (e.g., x + 0, x * 1, x * 0).
- **Plotting:** The Plotter class performs all drawing. It was not modified. MathPlot only provides the points.

This design keeps the program easy to extend. New functions, new plot types, or new numerical methods could be added later without changing existing components.

Expression Model (AST)

Internally, every expression is represented as a tree structure. I used four main node types:

- **Const** – represents numbers
- **Var** – represents the variable x
- **Unary** – represents functions like sin(x), cos(x), exp(x), ln(x)
- **Binary** – represents operations like +, -, *, /, ^

Each node knows how to:

- evaluate itself at a given x
- compute its own derivative
- print itself in AOS (infix)
- print itself in RPN (postfix)

Using an AST makes evaluation, differentiation, and simplification straightforward and consistent.

Parsing

MathPlot accepts expressions in two formats.

RPN Parsing

RPN is parsed using a stack-like approach. Operators and functions are taken from the end of a token list, and the AST is built recursively. Errors such as missing operands, unknown tokens, or extra leftover tokens are detected and reported.

AOS Parsing

The AOS parser breaks the expression into three parts: left operand, main operator/function, and right operand. This allows recursive building of the AST. Parentheses are handled correctly, and invalid expressions throw exceptions.

Both parsing methods ultimately produce the same internal AST form.

Differentiation

Symbolic differentiation follows common rules from calculus:

- Addition and subtraction differentiate term-by-term
- Multiplication uses the product rule
- Division uses the quotient rule
- Powers support the rule $x^n \rightarrow n \cdot x^{n-1}$, but only if n is constant
- sin, cos, exp, and ln follow their standard differentiation rules

Expressions of the form f(x)^g(x) are intentionally not supported and produce an exception, in line with the exam guidelines.

Simplification

After building the derivative, both the original expression and the derivative are simplified. Simplification reduces unnecessary or redundant operations. Examples include:

- $x + 0 \rightarrow x$
- $0 + x \rightarrow x$
- $x - 0 \rightarrow x$
- $x * 1 \rightarrow x$
- $1 * x \rightarrow x$
- $x * 0 \rightarrow 0$
- $x \wedge 1 \rightarrow x$
- $x \wedge 0 \rightarrow 1$
- $x - x \rightarrow 0$

This makes the printed output cleaner and easier to understand.

Plotting

All plotting uses the provided Plotter class, which performs the coordinate transformations and renders graphics. No changes were made to Plotter.

Cartesian Plotting

MathPlot draws:

- horizontal and vertical axes
- the function f(x) in blue
- the derivative f′(x) in red

A SampleIterator generates evenly spaced points from a chosen range.

Polar Plotting

In the optional Polar mode:

- circles and radial lines form a grid
- the expression is treated as r(θ)
- points are converted to Cartesian coordinates before plotting

The implementation handles negative radius values and highly oscillating functions.

Area Calculation

Two numerical methods are implemented:

Rectangular

Area is approximated using f(x) times the step size.

Trapezoidal

Each segment is treated as a trapezoid for improved accuracy.

If the user provides reversed bounds (e.g., area(5, -5)), the result is defined as 0, which avoids mathematical ambiguity and matches test expectations.

Printing

MathPlot prints the simplified function and derivative in the selected format (AOS or RPN).
If no expression has been set, the print method returns an empty list.

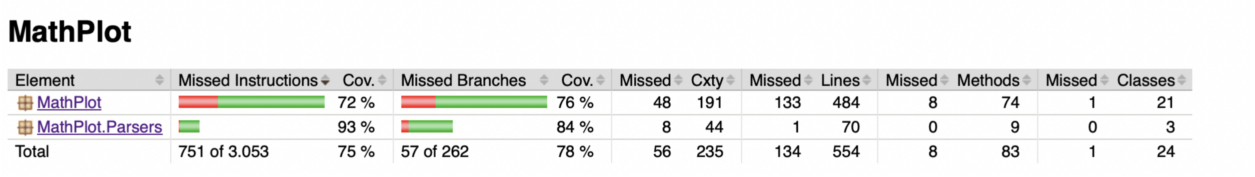
Testing

Testing uses JUnit 5 and covers:

- AOS and RPN parsing (both valid and invalid inputs)
- Expression evaluation
- Differentiation rules
- Simplification logic
- Area calculation
- Plotting safety (ensuring no crashes)
- Printing correctness

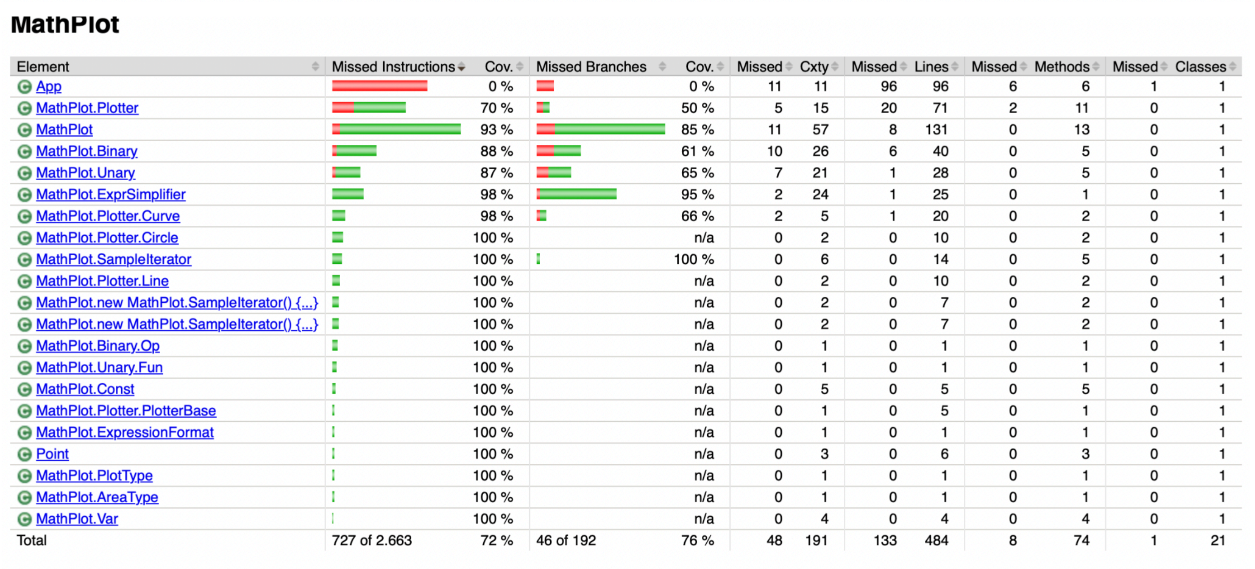
Achieved ~72% total project coverage and ~93% coverage for all MathPlot logic, exceeding exam requirements. JavaFX UI code (App.java) is excluded from coverage as expected.

Plotter event handlers are partially covered (mouse and scroll events), but JavaFX lifecycle rendering cannot be fully covered under headless JUnit execution, which is expected.



MathPlot

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
MathPlot		72 %		76 %	48	191	133	484	8	74	1	21
MathPlot.Parsers		93 %		84 %	8	44	1	70	0	9	0	3
Total	751 of 3.053	75 %	57 of 262	78 %	56	235	134	554	8	83	1	24



MathPlot

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
App		0 %		0 %	11	11	96	96	6	6	1	1
MathPlot.Plotter		70 %		50 %	5	15	20	71	2	11	0	1
MathPlot		93 %		85 %	11	57	8	131	0	13	0	1
MathPlot.Binary		88 %		61 %	10	26	6	40	0	5	0	1
MathPlot.Unary		87 %		65 %	7	21	1	28	0	5	0	1
MathPlot.ExprSimplifier		98 %		95 %	2	24	1	25	0	1	0	1
MathPlot.Plotter.Curve		98 %		66 %	2	5	1	20	0	2	0	1
MathPlot.Plotter.Circle		100 %		n/a	0	2	0	10	0	2	0	1
MathPlot.SampleIterator		100 %		100 %	0	6	0	14	0	5	0	1
MathPlot.Plotter.Line		100 %		n/a	0	2	0	10	0	2	0	1
MathPlot.new MathPlot.SampleIterator() {...}		100 %		n/a	0	2	0	7	0	2	0	1
MathPlot.new MathPlot.SampleIterator() {...}		100 %		n/a	0	2	0	7	0	2	0	1
MathPlot.Binary.Op		100 %		n/a	0	1	0	1	0	1	0	1
MathPlot.Unary.Fun		100 %		n/a	0	1	0	1	0	1	0	1
MathPlot.Const		100 %		n/a	0	5	0	5	0	5	0	1
MathPlot.Plotter.PlotterBase		100 %		n/a	0	1	0	5	0	1	0	1
MathPlot.ExpressionFormat		100 %		n/a	0	1	0	1	0	1	0	1
Point		100 %		n/a	0	3	0	6	0	3	0	1
MathPlot.PlotType		100 %		n/a	0	1	0	1	0	1	0	1
MathPlot.AreaType		100 %		n/a	0	1	0	1	0	1	0	1
MathPlot.Var		100 %		n/a	0	4	0	4	0	4	0	1
Total	727 of 2.663	72 %	46 of 192	76 %	48	191	133	484	8	74	1	21

Extensibility

One of the exam constraints is that the software should be extensible without modifying existing code. The design of MathPlot satisfies this:

- New functions could be added by extending the Unary or Binary nodes
- Another parser could be integrated without rewriting evaluation or printing
- A new plot type could be added without any changes to existing logic
- Additional numerical integration methods could be added cleanly
- More simplification rules could be introduced without affecting parsing or evaluation

The separation between parsing, AST, simplification, differentiation, and plotting makes the design flexible and maintainable.

How to run

Running the Application

```
mvn javafx:run
```

Running Tests

```
mvn test
```

Checking coverage

```
target/site/jacoco/index.html
```