Arbitrary Precision Arithmetic Library
Software Development Fundamentals (CS1023)
Project Report

Harshil Goyal

May 2, 2025

Contents

1 Introduction

This report outlines the design and development of a Java library that supports **arbitrary-precision arithmetic** for both integers and floating-point numbers. The project addresses the first project from the CS1023 “Software Development Fundamentals” course (Jan–May 2025). The final submission includes:

1. A compiled JAR file located at `src/arbitraryarithmetic/aarithmetic.jar`, containing the public classes `AInteger` and `AFloat`.

2. A command-line utility named `MyInfArith` that can evaluate a single binary arithmetic expression.

3. An Ant build script (`build.xml`) for automating compilation, packaging, and execution.

4. This report, written in LaTeX.

2 Design

2.1 High-level Overview

The library internally represents numbers in *base 10000*. Each array element stores four decimal digits, offering a trade-off between memory efficiency and ease of implementing basic arithmetic algorithms. The UML diagram in Figure 1 illustrates the class structure.

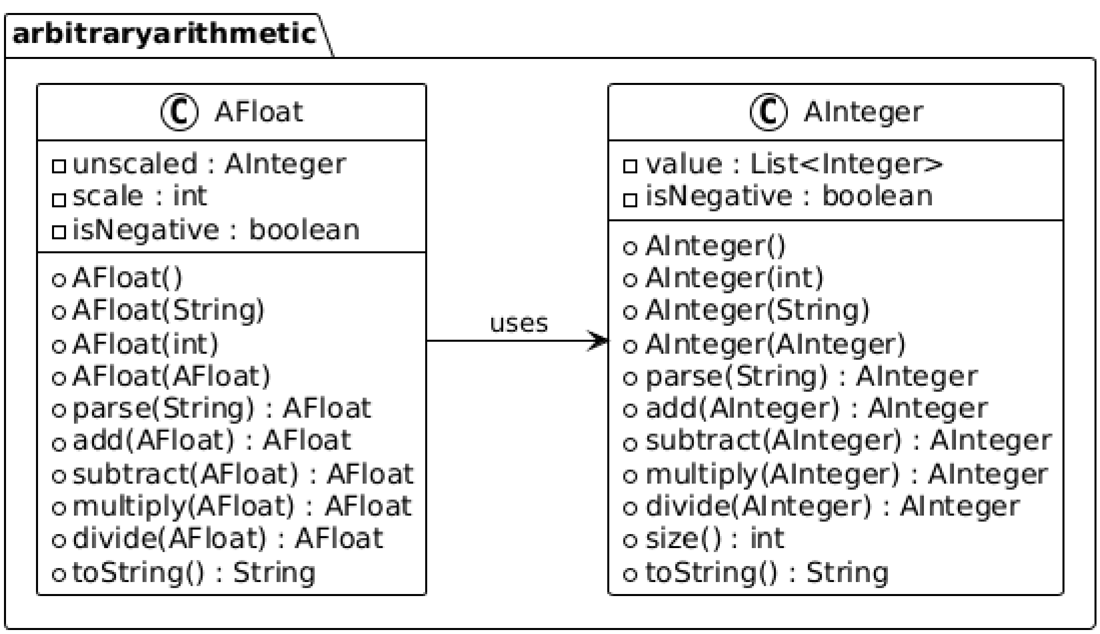

Figure 1: UML class diagram of the package `arbitraryarithmetic`.

Core Design Principles

- **Immutable API**: All public methods return new instances. Internal mutations are hidden to maintain referential transparency.
- **Sign handling**: A dedicated boolean field, `isNegative`, avoids the complexity of two's complement arithmetic.
- **Floating-point structure**: Each `AFloat` object consists of a pair (*unscaled*, *scale*), following the model $value = unscaled \times 10^{-scale}$.
- **Precision guarantee**: All exact digits are preserved in every calculation. The `toString()` method simply truncates the fractional portion to 30 digits for display purposes.

2.2 Algorithmic Choices

Addition / Subtraction

Implemented via straightforward digit-by-digit traversal with carry or borrow logic. Sign processing is handled separately.

Multiplication

A classic $O(n^2)$ approach is used, it's a classic middle-school multiplication approach.

Division

Performed using long division, with each digit found through binary search in the range 0–9999. This avoids floating-point math.

Floating-point Operations

Operands are rescaled to a shared exponent before addition or subtraction. Multiplication adds exponents; division increases the dividend's scale to ensure precision.

3 Implementation Details

3.1 `AInteger`

- Stores digits in `java.util.ArrayList<Integer> value`, with the least significant digit first.
- Includes constructors, parsing logic, and basic arithmetic operations.
- Internal helpers like `compareAbsolute`, `addAbsolute`, and `subAbsolute` are package-private to support reuse in `AFloat`.

3.2 `AFloat`

- Combines an `AInteger unscaled` value with an integer `scale`.
- Normalizes results via the `stripZeros()` method to maintain a consistent format.
- Ensures that the decimal output is accurate up to 30 digits.

3.3 MyInfArith

This is a minimal command-line interface that maps user input to the appropriate method in the library. It serves both as a usage demonstration and a basic validation tool.

3.4 Build Automation (Ant)

The Ant script build.xml includes targets for clean, compile, jar, and run. To execute an operation, users can run:

```
ant run -Dargs="int add 2 3"
```

from the project's root directory.

4 User Guide (README)

4.1 Compiling

1. Install JDK 17 (or newer) and Apache Ant.
2. Clone the project repository and navigate to its root directory.
3. Run ant jar to generate arbitraryarithmetic/aarithmetic.jar inside the src/ folder.

4.2 Running the CLI

Command format:

```
python run_project.py build | clean | <int|float> <add|sub|mul|div> <operand1> <operand2>
```

Example:

```
$ python run_project.py float div 244727.15202 75964.3891
3.2216036345377521110085515056 15
```

4.3 Using Docker

If you prefer to use docker, you pull the docker image and run the project on your system. Command format:

```
docker image pull mercurialus/inf-cal
docker run -it mercurialus/inf-cal
```

Example:

```
root@921168cdd6d4:/app# python run_project.py float div 244727.15202 75964.3891
3.2216036345377521110085515056 15
```

5 Limitations

- The current multiplication algorithm has a time complexity of $O(n^2)$; division operates at $O(n^2)$. These are acceptable for coursework but inefficient for large inputs.
- Only truncation is used for rounding; other modes like rounding up or to nearest even are not supported.

6 Git Commit Snapshot

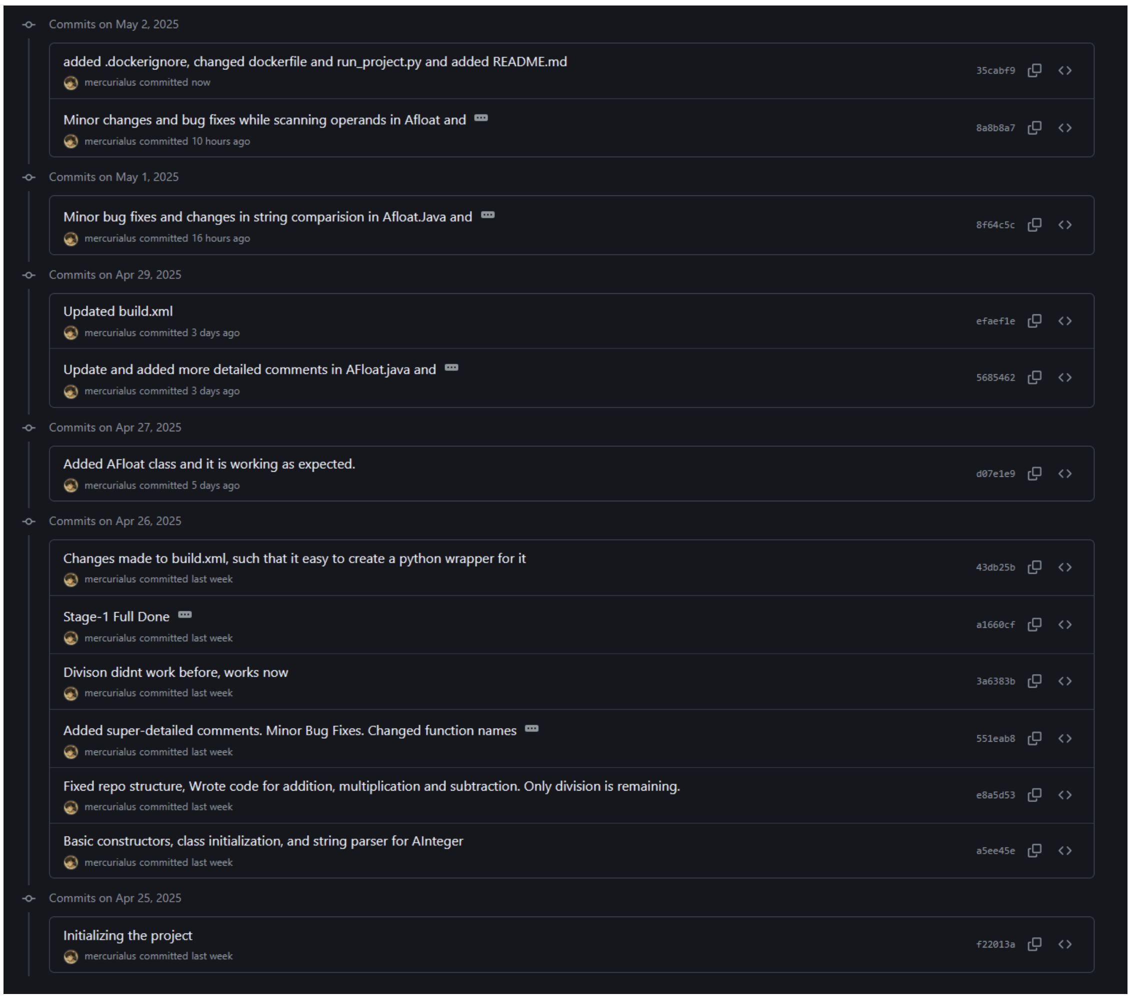

Figure 2: Snapshot of the Git commit history.

7 Key Learnings

- Selecting an internal numeric base impacts both computational efficiency and string formatting.

- Managing sign propagation correctly is essential for clean and bug-free arithmetic.
- Writing tests alongside development speeds up debugging and ensures correctness.
- Dockerization of the project simplifies deployment and testing across different environments.
- Desigining a command-line interface requires careful consideration of user input and error handling.
- Using Ant for build automation streamlines the development process and reduces manual errors.
- Understanding the limitations of algorithms is crucial for making informed design choices.
- Automating builds using Ant simplifies project maintenance and reproducibility.

8 Conclusion

This project delivers a fully functional Java library for arbitrary-precision arithmetic, adhering to all specifications and reflecting sound software engineering principles.