SVT 2024-25
Lab 1 Assignment

Testing process (Component Testing)

Date: **14/03/2025**

TEAM: **06**

Students:

Yan Chen Zhou

Darío Antonio Dueñas Loscos

Ivan Ocheretianyi

Tovahn Harvest Vitols

Candela Torrescusa Alonso

TABLE OF CONTENTS

1 Introduction **3**
1.1 Purpose of the document 3
1.2 Document Overview 3
1.3 References 3
2 Test Objectives **5**
3 Test Analysis, Design and Implementation **6**
3.1 Test Basis 6
3.2 Test Conditions 9
3.3 Test Cases 10
3.4 Test Procedures 18
3.5 Test Environment 20
3.6 Test Data 21
4 Test Execution and Results **22**
4.1 Test Case Results 22
4.2 Test Summary Report 25
5 Lists & Tables **27**
5.1 Traceability test basis – test conditions 27
5.2 Traceability test conditions – test cases 28
5.3 Traceability test cases – test procedures 29

1 Introduction

1.1 Purpose of the document

The purpose of this document is to describe the information gathered about the IEEE-754 floating point standard and the specifications provided for different types of operations (arithmetic, logical,...) using special values such as infinity, NaN,...
Based on this information, the document seeks to explain how the IEEE-754 floating point standard is followed by a series of tests implemented in Groovy 3 and 4.

The reader should be familiar with floating point representation in computers, arithmetic and logic. The reader is advised to know about testing phases and how to interpret test results.

1.2 Document Overview

The rest of this document contains a complete summary of not only our tests of the IEEE-754 floating point standard, but also of the process and results so as to allow for reproducibility and outside analysis.

To accomplish this, the document is organized into sections corresponding to Test Objectives, Test Analysis, Design, and Implementation (which itself is broken into a variety of subcategories relating to different aspects of Test Design), Test Execution and Results, and a final section containing useful lists and tables.

Test Objectives thoroughly outlines the reasoning behind this testing as well as the specific goals of this exercise.

Test Analysis, Design, and Implementation contains subsections pertaining to the background information on the IEEE-754 floating point standard as well as its implementation in Java and Groovy, the specific test conditions and case we are analyzing, the environment and procedure for running the tests, and the data provided as input for the tests.

Test Execution and Results contains details of the test execution itself as well as a formal analysis of the results.

The contents of Lists and Tables is determined to be self-evident.

1.3 References

General:

“IEEE Standard for Floating-Point Arithmetic”, Downloaded on February 18,2025 - https://standards.ieee.org/ieee/754/6210/

https://www.validlab.com/goldberg/paper.pdf

https://www.geeksforgeeks.org/ieee-standard-754-floating-point-numbers/

https://stackoverflow.com/questions/71224184/ieee-754-floating-point-comparisons-with-special-cases

https://rosettacode.org/wiki/Extreme_floating_point_values#Groovy

https://www.researchgate.net/publication/356802440_Early_Prediction_of_DNN_Activation_Using_Hierarchical_Computations - https://doi.org/10.3390/math9233130

For the test cases:
https://es.parasoft.com/blog/how-to-write-test-cases-for-software-examples-tutorial/

For math theory:

https://procomun.intef.es/articulos/limites-con-indeterminaciones

https://en.wikipedia.org/wiki/Indeterminate_form

2 Test Objectives

The primary test objective is to test float operators (*, /, +, -, sqr, exp, ...) for indeterminacies under extreme conditions (division by zero or infinity, square root of negative numbers, ...)

These tests will check if Groovy has an accurate and consistent behaviour and if it adheres to **the IEEE 754 floating-point standard**, as Java does.

Testing will be done **in different environments**:

- **Apache Groovy 3.0.19 + Spock 2.3**
- **Apache Groovy 4.0.25 + Spock 2.4-M5**

3 Test Analysis, Design and Implementation

3.1 Test Basis

Our test basis consists on the acquired knowledge on Groovy and Java, the IEEE 754 standard floating-point format and some mathematical theory:

1. **Groovy and Java Floating-Point Behaviour:**
   Groovy is a language that supports unit testing for the Java Virtual Machine. It builds upon the strengths of Java with some additional features, making modern programming features for Java developers.
   Since Groovy complements Java, both languages have similar behaviours when working with floating-point numbers.

   Java uses a subset of the IEEE 754 binary floating-point standard to represent floating-point numbers and define the results of arithmetic operations.

2. **IEEE 754 standard floating point specifications:**
   Floating point representation:

   In IEEE 754 format a floating point number consists of three parts:

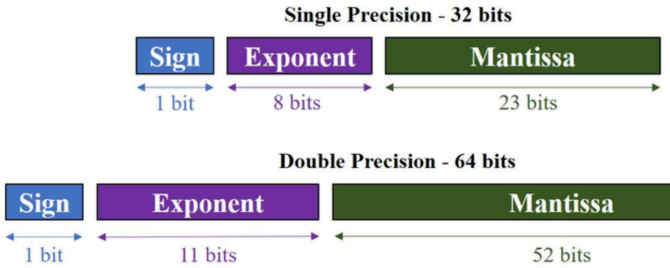
   - **Sign bit (S):** The Most Significant Bit, which is 0 for positive numbers, and 1 for negative numbers.
   - **Exponent (E):** A fixed number of bits assigned to represent the exponent, stored with a bias (127 for single precision and 1023 for double precision). This bias is needed to represent negative exponents then, the actual exponent is E - bias.
   - **Mantissa (M):** Contains the remaining bits. It represents the fractional part stored in the normalized form. The actual value in binary is 1 + M.

   Based on these rules, the floating point value represented using the S, E and M values in the IEEE-754 format is:

$$F = (-1)^S \times 2^{(E-127)} \times (1 + M)$$

   The two commonly used forms of the IEEE-754 format are:

   - Single precision format: 1 sign, 8 exponents, and 23 mantissa bits.
   - Double precision format: 1 sign, 11 exponents, and 52 mantissa bits.



Value	Exponent	Mantissa
Zero	0	0
Denormalized	0	Not 0
Infinity	Max value	0
NaN	Max value	Not 0

Logical operations, comparisons:

There are four mutually exclusive relations: less than, equal, greater than, and unordered.

The last case arises when at least one operand is NaN.

- Comparisons ignore the sign of zero (+0 == −0).
- Infinite operands of the same sign shall compare *equal*.
- Comparisons of infinity to the same infinity or zero to either zero should yield false for inequalities.
- Any inequality comparison (*less than* or *greater than)* with **NaN** should yield **false**.
- Every NaN shall compare unordered with everything, including itself.

Additionally, IEEE 754 has a total ordering function:

- **boolean totalOrder(source, source) ->** If x< y, then totalOrder(x, y) is true.

Mathematical operations with extreme and special values:

Because IEEE 754 takes into account both positive and negative zero (signed 0), it is specified that these zeros cause certain operations to return different values such as:

- $\frac{1}{-0}$ returns negative infinity.
- $\frac{1}{+0}$ returns positive infinity

IEEE 754 requires infinities to be handled in a reasonable way in mathematical operations, such as:

- (+∞) + (+7) = (+∞)
- (+∞) × (−2) = (−∞)
- (+∞) × 0 = NaN – there is no meaningful thing to do

IEEE 754 also specifies that NaN be returned as the result of certain "invalid" operations, such as:

$$\frac{0}{0} \qquad \infty \times 0 \qquad \sqrt{-1}$$

In general, most operations involving a NaN will result in a NaN, although functions that would give some defined result for any given floating-point value will do so for NaNs as well, such as:

$$NaN^{0} = 1$$

3. Mathematical theory:

In mathematics, some operations result in an indeterminacy, in other words, the result is unknown with the usual arithmetic operations.
To resolve them, we use special techniques that provide a numerical result.

These techniques include:

- **Algebraic manipulation:** There are several methods, such as multiplying and dividing by the **conjugate** or **simplifying** the expression, taking **common factors**...
- **Limits:** A tool for evaluating a mathematical expression as a variable tends to some value (which can be finite, infinite, zero,...).
- **Logarithms**: Useful for exponential expressions, which can **transform** them into an expression where we can apply L'Hopital's rule.
- **L'Hopital's rule:** Really useful for solving indeterminacies with limits. It consists of computing separately the derivatives of the numerator and denominator and re-evaluating the limit on that result, repeating the process until obtaining a result.

Some common indeterminacies are:

- $\frac{0}{0}$: can be often solved using L'Hopital's rule.

- 0^0: can be often solved using logarithms.
- **∞−∞:** can be solved with the conjugate technique if it gives an indeterminacy.
- 1^∞: can be often solved using logarithms. We also have this formula:

$$1^\infty = \lim_{x\to\infty}\left(\frac{f(x)}{g(x)}\right)^{h(x)} = \lim_{x\to\infty} e^{h(x)\left(\frac{f(x)}{g(x)}-1\right)}$$

- ∞^0: can be often solved using logarithms.

4. **Common Mathematical Operations**

   IEEE 754 also ensures that common mathematical operations such as trigonometric functions, as well as the above stated logarithmic functions, exponentials, or roots, behave well with floating point numbers.

3.2 Test Conditions

In this context of floating-point calculations in Groovy, each condition will be a specific numerical operation or scenario evaluated to verify adherence with the IEEE 754 standards.

They include testing arithmetic and logical operations, edge cases, and indeterminate forms. These conditions ensure that floating-point computations behave correctly across different environments and prevent unexpected inaccuracies or inconsistencies in calculations since we will learn about how Groovy behaves while working with floating-point.

A few of the test conditions with some input cases are:

- **Test comparisons with NaN and Infinity**:
  - **inf==inf** and **-inf==-inf:** should return true.
  - **inf>inf:** should return false.
  - **NaN==NaN:** should return false.
- **Comparison tests to verify the behaviour of signed zero:**

  In IEEE 754 floating-point, +0 is treated equally to -0. Based on that we have that:

  - **+0==-0:** should return true.

- **+0>-0:** should return false.
- **+0<-0:** should return false.

- **Operations with Infinities:**
  - **inf + inf:** should return +inf.
  - **inf - inf:** should return NaN because this operation is indeterminate.
  - **inf * (-inf):** should return -inf**.**
- **Indeterminate Forms**: For operations where the result is indeterminate.
  - **0/0** and **inf/inf** should return NaN as they are indeterminacies.
  - **0^0** should return 1.0 because it is defined like this in IEEE 754.
- **Extreme floating-point cases:** Extreme input cases that handle very large numbers to check they are handled correctly to prevent overflows.
  - **1/0** and **exp(inf)** should return infinity.
- **Boundary value tests** like operations on very small numbers, ± 0 to prevent underflows**.**
- **Trigonometric Tests**
  - **Sin(±inf)** and **Cos(±inf)** and **Tan(±inf)** should return NaN
  - **Arctan(±inf)** should return ± pi/2
- **Root Tests**
  - **Sqrt(inf)** should return positive infinity
  - **Sqrt(NaN)** should return NaN
  - The square root of any negative number should return NaN

3.3 Test Cases

Arithmetic operation tests:

1. **Addition/Subtraction with Infinity and Boundary values:**
   - **Test Case ID:** TC001.

- **Description:** Check that adding or subtracting a value to/from infinity returns infinity.
- **Preconditions:**
  - The programming environment (Groovy) supports IEEE 754 floating-point arithmetic.
  - Infinity (∞) is represented correctly in Java as floating-point `Float.POSITIVE_INFINITY`.

Operation	Expected result
0.0 + ∞	+ ∞
0.0 - ∞	- ∞
∞ + ∞ same as ∞ - (- ∞)	+∞
- ∞ - ∞ same as - ∞ + (- ∞)	- ∞
∞ - ∞	NaN
float maximum value + float maximum value	+∞
float minimum value + float maximum value	float maximum value
-float maximum value - float maximum value	- ∞
NaN + 1	NaN

2. **<u>Multiplication and division with Infinity and boundary values:</u>**

- **Test Case ID:** TC002.
- **Description:** Check that multiplying a non zero value (either finite or infinite) with infinity returns infinity. Check that infinite times 0 is not a number.
- **Preconditions:**
  - The programming environment (Groovy) supports IEEE 754 floating-point arithmetic.

- Infinity (∞) is represented correctly in Java as floating-point `float.POSITIVE_INFINITY`.

Operation	Expected result
± ∞ × ± ∞	± ∞
± ∞ ÷ ± ∞	NaN
0.0 × ± ∞	NaN
float maximum value × 2	+∞
float minimum value × 0.1	0.0
float maximum value × -2	- ∞
float minimum value × -0.1	0.0

3. **Division with 0:**

- **Test Case ID:** TC003.
- **Description:** Check that dividing a nonzero finite number by zero results in infinity and the division of zero by zero.
- **Preconditions:**
  - The programming environment (Groovy) supports IEEE 754 floating-point arithmetic.
  - Infinity (∞) is represented correctly in Java as floating-point `float.POSITIVE_INFINITY`.

Operation	Expected result
0.0 ÷ 0.0	NaN
1.0 ÷ 0.0	+∞
1.0 ÷ - 0.0	-∞
-1.0 ÷ 0.0	-∞
-1.0 ÷ -0.0	+∞

4. **Root of index n:**

- **Test Case ID:** TC004.
- **Description:**
  - Verify that cero remains unchanged for any root.
  - Check that a positive value always has a root and that a negative value gives a numerical result for an odd index.
  - Verify that NaN always gives a result of NaN for any index root.
- **Preconditions:**
  - The programming environment (Groovy) supports IEEE 754 floating-point arithmetic.

root of index n	index n	expected result
+0	any	+0
+1	even	+1
-0	any	-0
-1	even	NaN
-1	odd	-1
+∞	any	+ ∞
- ∞	even	NaN
- ∞	odd	- ∞
NaN	any	NaN

5. **Exponential Cases:**

- **Test Case ID:** TC005.
- **Description:**
  - Verify the correct behavior of the exponential function for special floating-point cases.
- **Preconditions:**
  - The programming environment (Groovy) supports IEEE 754 floating-point arithmetic.
  - The `Math.exp()` function is used for exponentiation.

Operation	Expected result
exp(+∞)	+∞

exp(-∞)	0.0
exp(NaN)	NaN
exp(710)	+∞ (overflow)
exp(-745)	0.0 (underflow)

6. **Exponential Cases:**

- **Test Case ID:** TC006.
- **Description:**
  - Verify the power function for special cases with Nan.
- **Preconditions:**
  - The programming environment (Groovy) supports IEEE 754 floating-point arithmetic.
  - The `Math.pow()` function.

Operation	Expected result
NaN ^ 0	1

7. **Logarithmic Cases:**

- **Test Case ID:** TC007.
- **Description:**
  - Verify the natural logarithm function for special cases like 0, -1, NaN and infinity.
- **Preconditions:**
  - The programming environment (Groovy) supports IEEE 754 floating-point arithmetic.
  - The `Math.log()` function is used for logarithms.

Operation	Expected result
ln(0)	-∞
ln(-1)	NaN
ln(+∞)	+∞

ln(NaN)	NaN

8. **Trigonometric Cases:**
    - **Test Case ID:** TC008.
    - **Description:**
        - Verify special cases of trigonometric operations with infinity and NaN.
    - **Preconditions:**
        - The programming environment (Groovy) supports IEEE 754 floating-point arithmetic.
        - The `Math.sin(), Math.cos(), Math.tan()` functions are defined and used in the tests.

Operation	Expected result
sin(NaN)	NaN
cos(NaN)	NaN
tan(NaN)	NaN
arctan(-NaN)	NaN
sin(+∞)	NaN
cos(+∞)	NaN
tan(+∞)	NaN
arctan(+∞)	π/2
arctan(-∞)	-π/2

Logical operation tests (comparisons):

1. **Is equal tests:**
    - **Test Case ID:** TC009.
    - **Description:**
        - Verify that equality of special values yield expected results from standard IEEE 754 floating-point arithmetic.
    - **Preconditions:**
        - The programming environment (Groovy) supports IEEE 754 floating-point arithmetic.

- The operator '==' is well used in Groovy and follows the IEEE 754 rules for special values.

COMPARISON	EXPECTED RESULT
Nan == Nan	FALSE
+inf == +inf	TRUE
- inf == - inf	TRUE
inf == - inf	FALSE

2. **Is less than equal tests (<):**

- **Test Case ID:** TC0010.
- **Description:**
  - Verify that equality of special values yield expected results from standard IEEE 754 floating-point arithmetic.
- **Preconditions:**
  - The programming environment (Groovy) supports IEEE 754 floating-point arithmetic.
  - The operator '<' is well used in Groovy (respects syntax) and follows the IEEE 754 rules for special values.

x	y	Expected Result
-∞	y	TRUE

Special cases:

<	NaN	-∞	-0	+0	+∞
NaN	FALSE	FALSE	FALSE	FALSE	FALSE
-∞	FALSE	FALSE	TRUE	TRUE	TRUE
-0	FALSE	FALSE	FALSE	FALSE	TRUE
+0	FALSE	FALSE	FALSE	FALSE	TRUE
+∞	FALSE	FALSE	FALSE	FALSE	FALSE

3. **Is greater than equal tests (<):**

- **Test Case ID:** TC0011.

- **Description:**
  - Verify that equality of special values yield expected results from standard IEEE 754 floating-point arithmetic.
- **Preconditions:**
  - The programming environment (Groovy) supports IEEE 754 floating-point arithmetic.
  - The operator '>' is well used in Groovy (respects syntax) and follows the IEEE 754 rules for special values.

x	y	Expected Result
+∞	y	TRUE

Special cases:

>	NaN	-∞	-0	+0	+∞
NaN	FALSE	FALSE	FALSE	FALSE	FALSE
-∞	FALSE	FALSE	FALSE	FALSE	FALSE
-0	FALSE	TRUE	FALSE	FALSE	FALSE
+0	FALSE	TRUE	FALSE	FALSE	FALSE
+∞	FALSE	TRUE	TRUE	TRUE	FALSE

4. **totalOrder tests:**
   - **Test Case ID:** TC0012.
   - **Description:**
     - Verify that equality of special values yield expected results from standard IEEE 754 floating-point arithmetic.
   - **Preconditions:**
     - The programming environment (Groovy) supports IEEE 754 floating-point arithmetic.
     - The `compareTo()` function, used for these tests is defined and maintains a proper ordering for all floating-point values.

x	y	Expected Result
-0	+0	TRUE
+0	-0	FALSE
same floating-point datum with negative exponent (EXP		TRUE

X >= EXP Y)		
same floating-point datum with positive exponent (EXP X <= EXP Y)		TRUE
-Nan	y	TRUE
x	-Nan	FALSE
x	+Nan	TRUE
+Nan	y	FALSE
-Nan	+Nan	TRUE
+Nan	-Nan	FALSE

3.4 Test Procedures

1. Precondition

Install all the necessary software and set it. To know the software needed, look at the following chapter Test environment.

The source code with implementation of floating point operations to be tested and some basic tests defined should be provided.

IEEE754 documentation should be analyzed to know expected results of tests and to be able to find problems in implementation.

2. Test Execution

As the tests should be performed with two different settings, the following steps must be performed twice: once for Groovy 3.0.19 with Spock 2.3 and another for Groovy 4.0.25 and Spock 2.4-M5

Step 1 - Initialize the project

- Check that the correct versions of Groovy, Spock and Java are used.
- Initialize the project, invoke the command **`gradle init`**.
- There would be some selection.
    - Type of build: Library (better for testing framework Spock)
    - Implementation language: Groovy

- Java version: 17 and above
- Project name: “the one needed by user”
- Build script DSL: Groovy
- Generate build using new APIs and behavior: no

- build.gradle could be modified
  - For better output of the test calls. Parts to be added are explained in Step 4 - Verify tests
  - For different versions of Groovy and Spock, the following parts should be added into the `dependencies` part
    - For Groovy 3.0.19 and Spock 2.3

```
implementation 'org.codehaus.groovy:groovy-all:3.0.19'

testImplementation
'org.spockframework:spock-core:2.3-groovy-3.0'
```

    - For Groovy 4.0.25 and Spock 2.4-M5

```
implementation 'org.apache.groovy:groovy:4.0.25'

testImplementation
'org.spockframework:spock-core:2.4-M5-groovy-4.0'
```

Step 2 - Add test files

- The code to be tested (**operations definition**) should be located at the directory **src/main/groovy.**
- The code with the **test cases** should be located at the directory **src/test/groovy.**
- Another approach, which is the one we finally followed, is to store both the functionality to be tested (operation definition in our case) and the tests in one file stored in **src/test/groovy**.

Step 3 - Execute Test

There are two possible ways:

1. **Test all the cases at the same moment**.
   - To do so, the command **`gradle test`** should be invoked. **`gradle build`** could be called to compile, test and package the application. But it is enough to

call **`gradle test`** as it compiles and runs test, which are enough for this project.

2. **Test each test condition separately.**
   To perform tests with regards to boundary values, like adding a positive number to the maximum float value, the command **`gradle test --tests '*BoundaryValuesTests'`** should be invoked, where the third parameter passed is the class that contains tests with regard to condition.
   The same actions should be performed to test other conditions.

Step 4 - Verify tests

Analyze the outputs of the tests, comparing results of operations performed with the expected theoretical output. Classify unexpected results as false positives or false negatives.

To have tests results better explained, three options are possible:

- read build/reports/tests/test/index.html that confirms whether the test was passed or failed, shows error details, and test coverage.
- Invoke command `gradle test --info` to show more information about failed tests
- Modify build.gradle file to contain the following code. This will provide additional information: all the tests passed will be displayed; more information about failed tests would be shown. Call the command `gradle test`

```
tasks.withType(Test) {
      testLogging {
            events "passed", "skipped", "failed"
            exceptionFormat "full" // Show full stack trace for
            failed tests
            showStandardStreams = true // Print output to
            console
      }
}
```

3. Compare results obtained with different settings (Groovy 3 and Groovy 4)

Analyze results obtained from both test environments. It is recommended to either compare .html files, or to have tables, etc containing results of the same tests with different versions.

3.5 Test Environment

The tests can be conducted on any of the three major operating systems: **Windows, macOS, or Linux**.

From a software perspective, **Java** must be installed first. It is recommended to use Java 17 or newer, as versions beyond 17 remain compatible. In our setup, we tested with Java 17 and Java 23.

After installing Java, **Apache Groovy** should be installed. The tests must be executed in two separate Groovy environments: Apache Groovy 3.0.19 and Apache Groovy 4.0.25

Each Groovy version requires a corresponding **Spock framework version**:

- Groovy 3.0.19 ⟶ Spock 2.3
- Groovy 4.0.25 ⟶ Spock 2.4-M5

To ensure efficient and consistent test execution across different environments, **Gradle** should be used as a build automation tool. Gradle simplifies dependency management, test execution, and reporting, making the testing process more reliable and scalable.

3.6 Test Data

As we are testing the default behavior of Groovy and Java, all our data is selected from Java's built in values.

For example, we used the built in maximum and minimum values and infinities for floats. As well as using floating point numbers, such as 0.0f or 1.0f, and special values like POSITIVE_INFINITY and NaN where required.

We did not use outside data.

4 Test Execution and Results

4.1 Test Case Results

Overall we have 3 fails in Groovy:4.0.25 and 15 fails in Groovy:3.0.19

		Groovy 3	Groovy 4
Test	**Expected result**	**Obtained result**	**Obtained result**
0.0 + ∞	+ ∞	CORRECT	CORRECT
0.0 - ∞	- ∞	CORRECT	CORRECT
∞ + ∞ same as ∞ - (- ∞)	+∞	CORRECT	CORRECT
- ∞ - ∞ same as - ∞ + (- ∞)	- ∞	CORRECT	CORRECT
∞ - ∞	NaN	CORRECT	CORRECT
float maximum value + float maximum value	+∞	CORRECT	CORRECT
float minimum value + float maximum value	float maximum value	CORRECT	CORRECT
-float maximum value - float maximum value	- ∞	CORRECT	CORRECT
NaN + 1	NaN	CORRECT	CORRECT
± ∞ × ± ∞	± ∞	CORRECT	CORRECT
± ∞ ÷ ± ∞	NaN	CORRECT	CORRECT
0.0 × ± ∞	NaN	CORRECT	CORRECT
float maximum value × 2	+∞	CORRECT	CORRECT
float minimum value × 0.1	0.0	CORRECT	CORRECT

float maximum value × -2	- ∞	CORRECT	CORRECT
float minimum value × -0.1	0.0	CORRECT	CORRECT
0.0 ÷ 0.0	NaN	CORRECT	CORRECT
1.0 ÷ 0.0	+∞	CORRECT	CORRECT
1.0 ÷ - 0.0	-∞	CORRECT	CORRECT
-1.0 ÷ 0.0	-∞	CORRECT	CORRECT
-1.0 ÷ -0.0	+∞	CORRECT	CORRECT
root(+0, any index)	+0	CORRECT	CORRECT
root(+1, any_index)	+1	CORRECT	CORRECT
root(-0, any index)	-0	CORRECT	CORRECT
root(-1, even index)	NaN	CORRECT	CORRECT
root(-1, -odd index)	-1	CORRECT	CORRECT
root(+∞, any index)	+ ∞	CORRECT	CORRECT
root(- ∞, even index)	NaN	CORRECT	CORRECT
root(- ∞, odd index)	- ∞	CORRECT	CORRECT
root(NaN, any index)	NaN	CORRECT	CORRECT
exp(+∞)	+∞	CORRECT	CORRECT
exp(-∞)	0.0	CORRECT	CORRECT
exp(NaN)	NaN	CORRECT	CORRECT
exp(710)	+∞ (overflow)	CORRECT	CORRECT
exp(-745)	0.0(underflow)	CORRECT	CORRECT
NaN ^ 0	1	CORRECT	CORRECT
ln(0)	-∞	CORRECT	CORRECT
ln(-1)	NaN	CORRECT	CORRECT

ln(+∞)	+∞	CORRECT	CORRECT
ln(NaN)	NaN	CORRECT	CORRECT
sin(NaN)	NaN	CORRECT	CORRECT
cos(NaN)	NaN	CORRECT	CORRECT
tan(NaN)	NaN	CORRECT	CORRECT
arctan(-NaN)	NaN	CORRECT	CORRECT
sin(+∞)	NaN	CORRECT	CORRECT
cos(+∞)	NaN	CORRECT	CORRECT
tan(+∞)	NaN	CORRECT	CORRECT
arctan(+∞)	π/2	CORRECT	CORRECT
arctan(-∞)	-π/2	CORRECT	CORRECT
+inf == +inf	TRUE	CORRECT	CORRECT
- inf == - inf	TRUE	CORRECT	CORRECT
inf == - inf	FALSE	CORRECT	CORRECT
NaN == NaN	FALSE	TRUE	TRUE
-0.0/1.0	-0.0f	0.0f	0.0f
-0.0/-1.0	0.0f	-0.0f	0.0f
-0.0f/1.0f	-0.0f	0.0f	-0.0
-0.0f/-1.0f	0.0f	-0.0f	0.0
Min_value*-0.1f	TRUE	FALSE	CORRECT
NaN > 0.0f	TRUE	FALSE	CORRECT
NaN > -0.0f	TRUE	FALSE	CORRECT
NaN > PositiveInfinity	TRUE	FALSE	CORRECT
NaN > NegativeInfinity	TRUE	FALSE	CORRECT

Float.compare(-0.0f, 0.0f)	-1	0	CORRECT
Float.compare(0.0f, -0.0f)	1	0	CORRECT
Float.compare(-0.0f/1.0f, 0.0f)	-1	0	CORRECT
1.0/-0.0f	Negative Infinity	Positive Infinity	CORRECT

4.2 Test Summary Report

While, as seen above, the majority of test cases passed, there are some notable exceptions that reveal how Groovy treats floats differently than the IEEE 754 standard and differences between Groovy versions.

Fails

In both environments (Groovy: 3.0.19 and Groovy: 4.0.25) we have found the following inconsistencies:

- There is no distinction between positive and negative values of zeros when compared against each other using **==**. In both versions of Groovy, **-0.0f == 0.0f** returns true.
  This does not comply with IEEE 754 as both +0 and -0 should be different. However, when using the function `Float.compare(a, b)`, Groovy is able to discriminate the signs correctly.

- Moreover, as stated in the IEEE754 documentation, NaN should not be equal to any number, not even itself. Yet, we discovered in the test cases that **NaN == NaN** returns true. This characteristic of NaN does not comply with the IEEE754 standard.

- We have found that Groovy treats NaN as the biggest floating point number, even bigger than +Infinity, which complies with the IEEE 754 standard assuming we are considering +NaN.
  However, as we mentioned previously -NaN should also exist and be considered less than -Infinity. As there is no -NaN, this is not the case.

We have detected slight differences in how Groovy:3.0.19 and Groovy:4.0.25 treat floating point numbers, which are:

- Groovy: 3.0.19 translates a negative zero **-0.0f** directly into a positive zero **+0.0f**. This does not happen in Groovy: 4.0.25. However, Groovy: 3.0.19 can preserve the sign of a negative zero if it is the direct result of a computation like **0.0f / -1.0f** or **0.0f*(-1.0f)**.
- Groovy : 3.0.19 treats **NaN** as the biggest floating point number when using comparators **<** or **>**, even greater than Positive Infinity. While Groovy:4.0.25 returns false for every comparison with **NaN**.
- In Groovy:4.0.25, the sign of the zero is only preserved if it is specified as **-0.0f**. If we use **-0.0**, this zero will lose its sign and become positive. See “ArithmeticTests.0/b Divisions with 0.0 and 0.0f”.

Overall, it appears that Groovy complies with the vast majority of the IEEE 754 standard with the main discrepancies appearing in signed versions of strange values such as 0 and NaN. Again, the main differences between versions of Groovy seem to appear when dealing with the same values.

5 Lists & Tables

The following tables show the traceability of a representative selection of test cases. Not all test cases are shown as the full list of test cases is available previously in this document and the below examples are believed to cover each type of test case well enough that tests not documented specifically can still be traced using this table.

The first table shows test conditions based on specific parts of IEEE 754. The second table shows the implemented test cases in Java based on specific conditions. The last table shows the grouping for test procedure/execution based on the test case.

5.1 Traceability test basis – test conditions

Test Basis	Test Conditions
Comparisons ignore the sign of zero	+0 = -0
Infinite operands of the same sign shall compare equal.	±∞ = ±∞
Comparisons of infinity to the same infinity or zero to either zero should yield false for inequalities.	±∞ ⊁ ±∞, ±0 ⊁ ±0, ±∞ ≮ ±∞, ±0 ≮ ±0
Any inequality comparison (less than or greater than) with NaN should yield false.	NaN ≮ anything, NaN ⊁ anything
Every NaN shall compare unordered with everything, including itself.	NaN ≠ anything, NaN ≠ NaN
1/(−0) returns negative infinity	1/(-0) = -∞
1/(+0) returns positive infinity	1/(+0) = +∞
infinities should be handled in a reasonable way during mathematical operations	±∞ + (±float) = ±∞, +∞ × (-float) = -∞, ±∞ × 0 = NaN
NaN should be returned as the result of certain "invalid" operations	0/0 = NaN, ∞×0 = NaN, sqrt(−1) = NaN, log(-1) = NaN
Functions that would give some defined result for any given floating-point value will do so for NaNs as well	NaN ^ 0 = 1
Normal mathematical functions should behave as expected	Sin(∞) = NaN, Arctan(∞) = π/2, sqrt(∞) = ∞, log(0) = -∞, log(NaN) = NaN, Log(∞) = ∞

5.2 Traceability test conditions – test cases

In the following test cases, x represents a variable of type float with any valid value such as:

Float x = 9.32;

Test Condition	Test Cases
+0 = -0	`+0 == -0`
±∞ = ±∞	`Float.PositiveInfinity == Float.PositiveInfinity, Float.NegativeInfinity == Float.NegativeInfinity`
±∞ ⩾ ±∞, ±0 ⩾ ±0, ±∞ ⩽ ±∞, ±0 ⩽ ±0	`Float.PositiveInfinity !> Float.PositiveInfinity, Float.NegativeInfinity !> Float.NegativeInfinity, Float.PositiveInfinity !< Float.PositiveInfinity, Float.NegativeInfinity !< Float.NegativeInfinity 0 !> 0, -0 !> -0, 0 !< 0, -0 !< -0`
NaN ⩽ anything, NaN ⩾ anything	`Float.NaN !< x, Float.NaN !> x`
NaN ≠ anything, NaN ≠ NaN	`Float.NaN != Float.NaN, Float.NaN != x, Float.NaN !< Float.NaN, Float.NaN !> Float.NaN, Float.NaN !< x, Float.NaN !> x`
1/(-0) = -∞	`1/(-0) == Float.NegativeInfinity`
1/(+0) = +∞	`1/(+0) == Float.PositiveInfinity`
±∞ + (±float) = ±∞	`Float.PositiveInfinity + x ==Float.PositiveInfinity, Float.NegativeInfinity - x ==Float.NegativeInfinity`
+∞ × (-float) = -∞	`Float.PositiveInfinity*(-x)==Float.NegativeInfinity`
±∞ × 0 = NaN	`Float.PositiveInfinity * 0 == Float.NaN, Float.NegativeInfinity * 0 == Float.NaN`
0/0 = NaN, sqrt(−1) = NaN, sqrt(−∞) = NaN, log(-1) = NaN	`0/0 == Float.NaN, Math.sqrt(-1.0) = Float.NaN, sqrt(Float.NegativeInfinity) = Float.NaN, log(-1.0) == Float.NaN`
NaN ^ 0 = 1	`Float.NaN ** 0 == 1`
Sin(∞) = NaN Cos(∞) = NaN, Tan(∞) = NaN	`sin(Float.PositiveInfinity) == Float.NaN, cos(Float.PositiveInfinity) == Float.NaN, tan(Float.PositiveInfinity) == Float.NaN`
Arctan(±∞) = ±π/2	`arctan(Float.PositiveInfinity) == Math.PI / 2, arctan(Float.NegativeInfinity) == - Math.PI / 2`

sqrt(∞) = ∞	`sqrt(Float.PositiveInfinity)==Float.PositiveInfinity`
log(0) = -∞, log(NaN) = NaN, Log(∞) = ∞	`log(0) == Float.PositiveInfinity, log(Float.NaN) == Float.NaN, log(Float.PositiveInfinity)==Float.PositiveInfinity`

5.3 Traceability test cases – test procedures

Test Cases	Test Procedure *with* *gradle test --tests "*____"*
`2 * Float.MAX_VALUE == Float.POSITIVE_INFINITY, 2 * Float.MAX_VALUE == Float.NEGATIVE_INFINITY,`	BoundaryValueTests
`Float.PositiveInfinity + x ==Float.PositiveInfinity, Float.NegativeInfinity - x ==Float.NegativeInfinity, Float.PositiveInfinity*(-x)==Float.NegativeInfinity, Float.NaN ** 0 == 1, log(-1.0) == Float.NaN, log(0) == Float.PositiveInfinity, log(Float.NaN) == Float.NaN,`	ArithmeticTests
`sin(Float.PositiveInfinity) == Float.NaN, cos(Float.PositiveInfinity) == Float.NaN, tan(Float.PositiveInfinity) == Float.NaN, arctan(Float.PositiveInfinity) == Math.PI / 2, arctan(Float.NegativeInfinity) == - Math.PI / 2`	TrigonometricTests
`Float.NaN != Float.NaN, Float.NaN != x, Float.NaN !< Float.NaN, Float.NaN !> Float.NaN, Float.NaN !< x, Float.NaN !> x, +0 == -0, Float.NaN !< x, Float.NaN !> x, Float.PositiveInfinity !> Float.PositiveInfinity, Float.NegativeInfinity !> Float.NegativeInfinity, Float.PositiveInfinity !< Float.PositiveInfinity, Float.NegativeInfinity !< Float.NegativeInfinity 0 !> 0, -0 !> -0, 0 !< 0, -0 !< -0`	ComparisonTests
`1/(-0) == Float.NegativeInfinity, 1/(+0) == Float.PositiveInfinity, 0/0 == Float.NaN, Float.PositiveInfinity - Float.PositiveInfinity == Float.NaN, Float.PositiveInfinity * 0 == Float.NaN, Float.NegativeInfinity * 0 == Float.NaN,`	IndeterminateTests

`1^Float.PositiveInfinity == Float.NaN`
`0^0 == 1`
`PositiveInfinity^0 == 1`	

`Math.sqrt(-1.0) = Float.NaN,`
`Math.sqrt(Float.PositiveInfinity)==Float.PositiveInfinity,`
`Math.sqrt(Float.NegativeInfinity) = Float.NaN,`
	RootTests