Interval Computation with Java-XSC

Renato V. Ferreira[1], Bruno J. T. Fernandes[1], Edmo S. R. Bezerra[1], Marcília A. Campos[1]

[1]Centro de Informática – Universidade Federal de Pernambuco (UFPE)
Recife – PE – Brazil
{rvf,bjtf,ersb,mac}@cin.ufpe.br

1. Introduction

Many great engineering and scientific advances of recent decades would not have been possible without the floating-point capabilities of digital computers [Bush, 1996]. Still, some results of floating-point calculations look strange, even to people with years of mathematical and scientific computation experience.

A great problem faced by the actual computational model is the numeric representation, due to the density of real numbers. These numbers can not be entirely discretely represented, so they are represented as floating-point numbers which can express a range of values exactly.

The computer floating-point unit works internally in base 2, binary. For example, the decimal fraction 0.1 cannot be precisely represented in binary. It is a repeater fraction 0.00011001100110... It is like the repeater fraction 1/3 = 0.33333… in base 10. When you add 0.333333... to 0.666666... you get 0.999999... rather than 1.0, even though you just added 1/3 + 2/3 to get 1. Yet, with floating-point binary representation, when you add 0.1 + 0.1 you will probably get something other than 0.2.

A crucial conclusion is that floating point is by nature inexact [Green, 2005], because no digital number representation system can handle every real number, since real numbers are continuous and machines use discrete data. It is very important to realize that any binary floating-point system can represent only a finite number of floating-point values in exact form. All other values must be approximated by the closest representable value [Microsoft, 2003] [IEEE, 1985].

The objective of this project is to implement the Interval type and interval operations in Java to develop a library with unary and binary interval functions and to compare results from this work with others from MapleInt.

This paper is organized as follows: Section 2 provides a broad outline of the floating-point in Java; Section 3 describes the interval type and the interval operations in Java and results from Java-XSC and MapleInt. Finally, conclusion follows in Section 4.

2. Floating-point in Java

Java has gained enormous popularity since it first appeared. Its rapid ascension and wide acceptance can be traced to its design and programming features, particularly in its promise that you can write a program once, and run it anywhere. As stated in Java language white paper by Sun Microsystems: "Java is a simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, multithreaded, and dynamic" [Choudhari, 2001] [Sun Microsystems, 2005].

Java offers simple inheritance with the use of interfaces when compared to C++ and has eliminated the use of pointers. Another main advantage to Java programmers, is that Java automatically manages memory allocation and garbage collection. Java is designed to make distributed computing easy with the networking capability that is inherently integrated into it. Writing network programs in Java is like sending and receiving data to and from a file. Due to all that Java, nowadays, is the most popular programming language in the world.

2.1. Floating-point errors

For representing floating-point numbers, the Java programming language offers two primitive types, *float* and *double* and the wrapper classes *Float* and *Double* from the *java.lang* package [Sun Microsystems, 2005]. The double primitive type supports 16 decimal digits but it only offers correctness for the 14 most significant digits [Gosling, Joy, Steele, 1996].

Example 1: This example intends to perform a simple subtraction between two floating-point numbers, then compare the result of the performed operation with the expected result and print the result of the comparison.

```
double d = 3.9-3.8;
if(d==0.1)
System.out.println("equals")
;
else
System.out.println("not
equals");
```

Picture **l:** Example of source code

The example should clearly output `equals` at *stdout* (standard output), but instead of that, it outputs `not equals` because d was equal to `0.10000000000000009` showing that subtraction returned an incorrect value.

Example 2: This example intends to perform a sequence of ten additions, and finally print the aggregated result.

```
double d = 0.0;
for(int i = 0; i < 10; i++){
      d += 0.4;
}
System.out.println(d);
```

Picture **2:** Example of source code

The example makes a loop that adds 0.4 ten times to the variable d, so it should output 10 x 0.4 that is obviously 4.0, but the number outputted at *stdout* was `3.9999999999999996`.

Most of the people will not be affected by precision errors. No one would complain if their cabinet maker made a desk 2.000000000001 meters long, but for scientific computation, which needs high precision, this error margin is most of the times not acceptable, for it can cause wrong results, as seen [Ferreira, Fernandes, Campos, 2004].

2.2. Java floating-point implementation

The Java floating point implementation follows partially the IEEE 754 (1985) standard for floating point arithmetic [The Macaulay Institute, 2004]. As said previously, the floating-point types are float and double, representing the single-precision 32-bit and double-precision 64-bit format IEEE 754 values and operations as specified in *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985 (IEEE, New York).* The bit representation for *float* goes as

1 bit	8 bits	23 bits
sign	exponent	significand

Picture **3:** Float bit representation

and for *double* type

1 bit	11 bits	52 bits
sign	exponent	significand

Picture **4:** Float bit representation

The *float* representation gives 6 to 9 digits of decimal precision while *double* gives 15 to 17 digits of decimal precision.

Java requires that floating-point arithmetic behave as if every floating-point operator rounded its floating-point result to the result precision. Inexact results must be rounded to the representable value nearest to the infinitely precise result; if the two nearest representable values are equally near, the one with its least significant bit zero is chosen. This is the IEEE 754 standard's default rounding mode known as *round to nearest* [Gosling, Joy, Steele, 1996].

Java uses *round toward zero* when converting a floating value to an integer, which acts, in this case, as though the number were truncated, discarding the mantissa bits. Rounding toward zero chooses at its result the format's value closest to and no greater in magnitude than the infinitely precise result [Gosling, Joy, Steele, 1996].

Java floating-point operators produce no exceptions. An operation that overflows produces a signed infinity, an operation that underflows produces a signed zero, and an operation that has no mathematically definite result produces NaN (Not a Number).

The inadequacies of Java floating-point implementation are that it fails to fully implement the requirements of IEEE 754 standard. Including three main problems which are: (i) Do not support IEEE 754 sticky bit exception flags; (ii) Do not provide support for IEEE 754 directed roundings; (iii) Do not provide a machine interval arithmetic datatype.

The exception flags are Invalid Operation, Overflow, Division-by-Zero, Underflow, Inexact Result, and without handling with these flags and support directed rounding it is impossible to

conform to IEEE 754 [The Macaulay Institute, 2004].

3. Java-XSC

According to Donald Knuth in volume 2 of The Art of Computer Programming, p. 241, "Interval arithmetic provides truly reliable error estimates. Since the intermediate values in a calculation often depend on each other, the final estimates obtained with interval arithmetic will tend to be pessimistic. However, the prospects for effective use of interval arithmetic look very good, so efforts should be made to increase its availability." [Knuth, 1997]

Regarding all that, the motivation of this project is to build an API (Application Programming Interface) called Java-XSC to provide automatic floating point error control, implementing directed rounding modes and the interval type and interval arithmetic in Java, working towards to build a calculator to intervals.

Instead of approximating a real number **x** to a floating point machine number, the real is approximated to an interval **X** that contains **x**, **X** has an upper bound and a lower bound represented as floating pointing machine numbers. Calculations are done using intervals, so we substitute arithmetic operations for interval operations, which guarantee results [Moore, 1996][Campos, 1997].

The Interval type, that we built, in Java, has two properties which are the lower bound and the upper bound, both of them are represented by the *double* primitive type.

The library was modeled using the Rational Rose Enterprise Edition [IBM Rational Software, 2002] and implemented using the Eclipse IDE (Integrated Development Environment) [Eclipse.org, 2005] versions 2.1, 3.0 and 3.1. Java version used was 1.4.2 and 5.0.

3.1. Directed Rounding

Before defining the Interval type, we implemented the Rounding type which contains two operations (*methods*) to do the directed rounding, the rounding up and the rounding down which round a given *double* with a specified precision.

public static double roundDown(double d, int dec);
Method that rounds down the number d to a specified number of decimal digits.

public static double roundUp(double d, int dec);
Method that rounds up the number d to a specified number of decimal digits.

3.2. Interval Type

3.2.1. Unary Operations

public boolean equals(Object arg0);
Method which overrides the equals method from java.lang.Object and compares two intervals [x1, x2], [x3, x4] and returns true if x1 = x3 and x2 = x4, this comparison tolerates precision errors.

public double width();
Method that calculates the width of the Interval. For a given interval X = [x1,x2], the width is defined as x2-x1.

public boolean isEmpty();
Method that verifies if the Interval is empty.

public boolean pertains(double arg0);
Method that verifies if a machine floating point number is contained in the Interval. For a given interval X = [x1,x2], d pertains to X if d >=x1 and d <=x2.

public Interval symmetric();
Method that returns the symmetric interval. The symmetric of an interval X = [x1, x2] is Y = [-x2, -x1]

public Interval reciprocal();
Method that returns the reciprocal interval. The reciprocal of an interval X = [x1, x2] is Y = [1/x2, 1/x1]

3.2.2. Binary Operations

public static Interval add(Interval arg0, Interval arg1);
Method that returns an interval that is equal to arg0 + arg1. The addition is defined as [x1, x2] + [x3, x4] = [x1 + x3, x2 + x4]

public static Interval sub(Interval arg0, Interval arg1);
Method that returns an interval that is equal to arg0 - arg1. The subtraction is defined as [x1, x2] - [x3, x4] = [x1 - x3, x2 - x4]

public static Interval mult(Interval arg0, Interval arg1);
Method that returns an interval that is equal to arg0 * arg1. Multiplication is defined as [x1, x2]

* [x3, x4] = [min(x1*x3, x1*x4, x2*x3, x2*x4), max(x1*x3, x1*x4, x2*x3, x2*x4)]

public static Interval div(Interval arg0, Interval arg1);
Method that returns an interval that is equal to arg0 / arg1; Division is defined as [x1, x2] / [x3, x4] = [x1, x2] * [1/x4, 1/x3]

public static Interval intersection(Interval arg0, Interval arg1) ;
Method that returns the intersection between the intervals received as parameters.
public static Interval union(Interval arg0, Interval arg1);
Method that returns the union between the intervals received as parameters.

public static double abs(Interval arg0);
Method that returns the absolute value of an interval.

public static double distance(Interval arg0, Interval arg1);
Method that returns the distance between the intervals received as parameters.

public static boolean isIn(Interval arg0, Interval arg1);
Method that returns true if the interval arg0 is contained in the interval arg1.

3.3. Results

Java - XSC	MapleInt
Round down: 2.0. 10^{-9} Precision	
1.999999999	1.999999999
Round down: 3.88888888899. 10^{-9} Precision	
3.888888888	3.888888888
Round down: -1.6564564876648. 10^{-9} Precision	
-1.656456488	-1.656456489
Round up: 2.0. 10^{-9} Precision	
2.000000001	2.000000001
Round up: 3.88888888899. 10^{-9} Precision	
3.888888890	3.888888890
Round up: -1.6564564876648. 10^{-9} Precision	
-1.656456486	-1.656456487
Width of [1.0, 2.5]	
1.5	1.5
Width of [0.0, 8.9]	
8.9	8.9
Width of [-5.6, 4.9]	
10.5	10.5
Reciprocal of [-3.0, 8.4] 10^{-9} Precision	
[-Infinity,	[-Infinity,
Infinity]	Infinity]
Reciprocal of [-5.0, -2.0] 10^{-9} Precision	
[-0.500000001, -0.199999999]	[-0.500000001, -0.199999999]
Reciprocal of [1.0, 4.0] 10^{-9} Precision	
[0.249999999, 1.000000001]	[0.249999999, 1.000000001]
0.0 pertains [-0.1, 0.1]	
true	true
0.2 pertains [-0.1, 0.1]	
false	false
0.1 pertains [-0.1, 0.1]	
true	true
Add: [1.0, 2.5] + [0.0, 8.9] 10^{-9} Precision	
[0.999999999, 11.400000001]	[0.999999999, 11.400000001]
Add: [-8.9, 0.0] + [0.0, 8.9] 10^{-9} Precision	
[-8.900000001, 8.900000001]	[-8.900000001, 8.900000001]
Sub: [-8.9, 0.0] - [0.0, 8.9] 10^{-9} Precision	
[-0.000000001, 17.800000001]	[0.000000000, 17.800000001]
Sub: [1.0, 3.0] - [4.0, 5.0] 10^{-9} Precision	
[-4.000000001, -0.999999999]	[-4.000000001, -0.999999999]
Mult: [1.0, 3.0] * [4.0, 5.0] 10^{-9} Precision	
[3.999999999, 15.000000001]	[3.999999999, 15.000000001]
Mult: [-8.9, 0.0] * [1.0, 2.5] 10^{-9} Precision	
[-22.250000001, 0.000000001]	[-22.250000001, 0.000000000]
Div: [-8.9, 0.0] / [1.0, 2.5] 10^{-9} Precision	
[-Infinity, Infinity]	[-Infinity, Infinity]
Div: [4.0, 5.0] / [1.0, 2.5] 10^{-9} Precision	
[1.599999999, 5.000000001]	[1.599999999, 5.000000006]
Intersection: [1.23,1.89]∩[1.1,1.29]	
[1.23,1.29]	[1.23,1.29]
Intersection: [1.23,1.89]∩ [1.5,1.6]	
[1.5,1.6]	[1.5,1.6]
Union: [1.23,1.89]∪[1.1,1.29]	
[1.0, 1.89]	[1.0, 1.89]
Union: [1.23,1.89] ∪ [1.5,1.6]	
[1.23, 1.89]	[1.23, 1.89]
Is in: [1.23,1.89] ⊇ [1.1,1.29]	
false	false
Is in: [1.23,1.89] ⊇ [1.5,1.6]	
true	true
Distance: [1.23,1.89], [1.5,1.6]	
0.27	Method not found in MapleInt
Absolute: |[1.23,1.89]|	
1.89	Method not found in MapleInt

Tabela l: Comparação dos resultados Java-XSC x MapleInt, intpakX

4. Conclusion

Computational systems are incapable of representing all real numbers, because of their density. Digital representation is discrete and only covers a range of real numbers. Java, being a programming language, suffers from this fact, together with that, Java implementation has ignored important aspects of IEEE specification. Because of all that, Java presents errors when dealing with floating point numbers.

Since Java is, nowadays, the most popular programming language in the world, due to a range of advantages offered, it tends to be used for scientific applications. Regarding to minimize and control floating-point errors, a library called Java-XSC(eXtension for Scientific Computation) was developed, using interval methods.

Comparing with the Maple interval extension, the intpakX, MapleInt [Wuppertal, 2004], the Java-XSC has presented similar outputs in the majority of the test cases, and presented better results in some operations.

The library will be extended to contain not only arithmetic and logical operations, but also logarithmic, trigonometric and statistic operations.

References

Bush, B.M., (1996), "The Perils of Floating Point", http://www.lahey.com/float.htm, accessed in June 10, 2005.

Green, R., (2005), "Java Glossary: Floating Point", http://mindprod.com/jgloss/floatingpoint.html, accessed in June 12, 2005.

Microsoft (2003), "Tutorial to Understand IEEE Floating-Point Errors", http://support.microsoft.com/kb/q42980/, accessed in January 12, 2005.

IEEE Computer Society (1985), "IEEE Standard for Binary Floating-Point Arithmetic", IEEE Std 754-1985.

Choudhari, P., (2001), "Java Advantages & Disadvantages", http://arizonacommunity.com/articles/java_32001.shtml, accessed in June 12, 2005.

Sun Microsystems, "Java Language Overview", Java 2 Platform, Standard Edition, White Papers, http://java.sun.com/docs/overviews/java/java-overview-1.html, accessed in January 13, 2005.

Sun Microsystems, "JavaTM 2 Platform, Standard Edition, v 1.4.2 API Specification", http://java.sun.com/j2se/1.4.2/docs/api/, accessed in January 9, 2005.

Gosling, J., Joy, B., Steele G., (1996), "The Java Language Specification", Addison Wesley.

Ferreira, R. V., Fernandes, B. T., Campos, M. A., (2004), "Evaluating the floating-point in Java Virtual Machine", Proceedings ENNEMAC 2004.

The Macaulay Institute, (2004), "Floating Point Arithmetic and Java", http://www.macaulay.ac.uk/fearlus/floating-point/javabugs.html, accessed in December 8, 2004.

Knuth, D., (1997) "The Art of Computer Programming", Volume 2.

Moore, R. E. (1996), "Interval Analysis" Englewood Cliffs: Prentice-Hall

Campos, M. A., (1997), "Uma Extensão Intervalar para a Probabilidade Real" Departamento de Informática – UFPE

IBM Rational Software, (2002) – Rational Rose Enterprise Edition, http://www306.ibm.com/software/rational/offerings/reqanalysis.html, accessed in January 13, 2005

Eclipse.org, (2005), http://www.eclipse.org/, accessed in June 30, 2005.

MapleSoft.com, http://www.maplesoft.com/, accessed in January 22, 2005.

Wuppertal, 2004, "Verified Numerics meets Computer Algebra" http://www.math.uni-wuppertal.de/~xsc/software/intpakX/