JInterval Library: Principles, Development, and Perspectives*

Dmitry Yu. Nadezhin
Oracle Development SPb, Zelenograd, Russia
dmitry.nadezhin@oracle.com

Sergei I. Zhilin
Altai State University, Barnaul, Russia
sergei@asu.ru

Abstract

This paper presents a Java library for interval computations. The design of the bottom layers of the library follows the structure prescribed in the IEEE P1788 Interval Standard draft. The library provides a user with the possibility to choose interval flavor (classic, set-based, Kaucher), interval bounds representation (extended rational, IEEE 754-2008 floating point) and associated rounding policy. The top functional layer of the library implements high-level interval methods such as solvers of systems of interval linear equations and interval regressions. Two applications of JInterval are described: P1788 test suite and interval nodes for KNIME data mining platform.

Keywords: interval analysis, interval arithmetic, Java library, P1788 interval standard, data mining
AMS subject classifications: 65-04, 65G30, 65G40

1 Introduction

The Java language is a general-purpose, concurrent, class-based, object-oriented language specifically designed to have as few implementation dependencies as possible. There is a huge number of applied Java-libraries and standard APIs that make development in Java easy, rapid and cost-effective, not only in business but also in science. Java popularity among developers (and especially among developers of applied systems) results from a combination of Java platform features and properties such as binary portability, safe memory management, network-aware environment, parallel

*Submitted: February 17, 2013; Revised: September 18, 2013; Accepted: January 21, 2014.

and distributed computing tools, strict model of security, standard graphics and user interface, etc.

There are various areas of Java applications. Besides numerous corporate information systems, embedded software and applications for mobile devices, Java has proven its usefulness and importance in scientific areas as well. Some of counterarguments for Java use in scientific computing are still valid (such as language restrictions), but many of them are less so at the moment (such as low performance) [28].

Analysis of large data sets (so-called Big Data) is one of the spheres where Java is playing a key role now. Such highly effective and efficient tools for large-scale fault-tolerant data analysis as Hadoop [2] and its ecosystem, implementing the MapReduce concept [9], have been a major area of Java development over the past few years.

In high-performance computing (HPC), Java is not in the mainstream, but the active research efforts in this area are expected to bring new developments in the near future that will continue stimulating the interest of both industry and academia and increasing the benefits of Java adoption for HPC [28].

Java's significant role in applied and scientific software development and a plethora of applied libraries highlights the absence of advanced interval tools in Java, although interval techniques meet the requirements of a wide variety of applications.

The history of intervals in Java evidently originates from efforts of G. W. Walster and the interval community to add native support of intervals to the Java language shortly after Java's appearance at Sun Microsystems [6]. Unfortunately, the goal had not been achieved.

Other noticeable interval projects in Java are libraries IA_math by Timothy Hickey [13] and Java-XSC developed by Jose Dutra in his master's thesis under the supervision of Prof. Benjamin Bedregal [10, 7]. In both libraries, interval bounds are represented by the double precision floating-point type. The first one provides only classic interval arithmetic (IA) and elementary functions. The functionality of the second library is wider; besides classic IA, it implements complex rectangular IA, classic and complex interval vectors and matrices. At the same time neither Java-XSC nor IA_math contains high-level interval routines required in applications such as solving systems of equations, optimization and so on.

All the above considerations motivated the appearance of our JInterval library for interval computations. The library is intended mainly for developers creating applied Java-based software, so it is aimed not only at low-level interval arithmetic, but at providing high-level interval methods and solvers as well.

This paper is organized in the following way. In Section 2, some basic principles, which JInterval library design relies upon, are discussed. Section 3 gives an overview of the library architecture. Functionality provided by JInterval and several simple examples of its use are described in Section 4. Section 5 presents a couple of JInterval applications. Finally, Section 6 ends the paper with some concluding remarks and perspectives of JInterval development.

2 Principles

The design of our JInterval library is guided by the following basic requirements, ordered by descending priority.

0. *The library must be compliant with the IEEE P1788 standard for interval arithmetic* [22, 3]. Strict P1788 compatibility of the library is of absolute value, but the standard draft is not finished yet and it is being actively modified, so JInterval may

lag the draft on its way to the final IEEE standard. The P1788 standard covers only the low-level part of the foreseen functionality of the library, but these bottom layers shape top level architecture as well.

1. *The library must be clear and easy to use.* No matter how wonderful a software tool is, it will be hardly accepted by developers if it is not transparent and easy to use.

2. *The library should provide flexibility in the choice of interval arithmetic for computations.* The user must be able to choose interval arithmetic (classical, Kaucher, complex rectangular, complex circular, etc.) and to switch from one arithmetic to another if they are compatible. Syntactic differences between using one or another arithmetic should be minimized.

3. *The library should provide flexibility to extend its functionality.* The library must be layered functionally. Three layers should be defined: interval arithmetic operations (according to P1788), interval vector and matrix operations, and, finally, high-level interval methods, such as solvers of equations, optimization procedures, etc. The architecture of the library must allow for extensions at every layer, including the bottom one.

4. *The library should provide flexibility in choosing the precision of interval endpoints and associated rounding policies.* The choice of the interval boundaries representation and the rounding mode could allow the user to tune accuracy and speed of calculations, depending on the problem solved.

5. *The library must be portable.* Cross-platform portability of the library is one of its major strengths, and is a key distinction over its closest competitors. To a large extent, this requirement is ensured by the choice of the Java technology built on the principle “write once, run anywhere”. However, the design must adhere to certain restrictions for practical implementation of this requirement.

6. *The library should provide high performance.* In an era of multicore and multiprocessor systems, a prerequisite for high performance is the ability to use the library safely in a multithreaded environment.

Attempts to simultaneously satisfy all the requirements often run into conflicts. In these situations, the developers followed the declared priority of the requirements.

3 Library Architecture

This section describes core ideas that shape the design of JInterval and its implementation in Java. The section starts with a brief overview of some specific features of the Java language important for understanding further considerations. Subsection 3.2 presents the implementation of a “smart” number type. Interval types based on the number type are discussed in subsection 3.3. Descriptions of number and interval types design in JInterval are preceded by considerations on a mathematical level. Finally, subsection 3.4 presents the modular structure of the library.

3.1 OOP Concepts in Java

This subsection highlights some peculiarities of the implementation of object-oriented programming concepts in the Java language. Knowledge of these issues helps one to understand Jinterval design. A comprehensive description of the Java language can be found, for example, in [4].

One of the specific features of the Java language is the explicit syntactic construction which expresses the notion of class interface. An interface in Java is similar to a class, but it contains no data and exposes behaviors defined as methods. More formally, an interface is an abstract type used to specify an interface (in the generic sense of the term) that classes must implement. A class having all the methods defined in the interface is said to implement that interface. Usually a method in an interface cannot be used directly, but there must be a class which implements it. Different classes can implement the same interface in their own manners. Thus, interfaces in Java are mainly a way to achieve polymorphism. Besides, multiple class inheritance is not allowed in Java, but it can be simulated using interfaces, because a class can implement multiple interfaces.

Another important feature of the Java language's is parameterized types or generics. Generics allow "a type or a method to operate on objects of various types while providing compile-time type safety" [4]. A common use of generics is building container classes which can hold objects of any type. The core idea of generics is to abstract over a type (class or interface) by introducing a so called type parameter (or type variable) in the declaration of the type. A type variable is delimited by angle brackets and follows the class (or the interface) name in the declaration, as in the following example of generic interface.
`Container<T>`:

```
   interface Container<T> {
      void add(T item);
      Iterator<T> iterator();
   }
```

The type variable `T` is used inside the declaration body as a placeholder for some specific type. This specific type is bound to the type variable when an object of generic type is declared. The correctness of type substitution is checked at compile time. In our example, if the class `List<T>` implements interface `Container<T>` then the list of integers can be declared as

```
   Container<Integer> myIntList = new List<Integer>();
```

i.e., type parameter is bound to `Integer` type.

Interfaces and generic interfaces are widely used in JInterval to express the hierarchy of number and interval types described below.

3.2 Basic Number Type

3.2.1 Mathematical Level

Intervals are represented by their endpoints. Achieving flexibility in choosing precision of interval endpoints, associated rounding policies and computational performance (see principle 4) requires a well-designed number data type.

We assume the value set for the basic number data type is equal to the set of rational numbers extended with $\{-\infty, +\infty\}$. This set is denoted by $\overline{\mathbb{Q}}$, and corresponds to the class `ExtendedRational` in the library. Such a supposition is motivated by the urge to build an algebraic system closed under arithmetic operations. This property allows us to obtain precise solutions of many linear algebra problems. It is important to note that the set $\overline{\mathbb{Q}}$ contains value sets of floating-point types both for radix 2 and 10.

Elementary functions are also defined for the basic number type, but their precise results can lie out of $\overline{\mathbb{Q}}$, in contrast to arithmetic operations over $\overline{\mathbb{Q}}$. This issue, together with practical limitations, are reasons why approximate versions of arithmetic operations and elementary functions are necessary.

Results of an approximate version of arithmetic operations and elementary functions belong to some finite subset of $\overline{\mathbb{Q}}$ specified for this version of operations or functions. In the P1788 standard, this subset is named "number format" and is denoted by $\mathbb{F}$. According to P1788, a number format must contain zero, $-\infty$, $+\infty$ and must be symmetric. In our library, the parameters that represent this subset are called a `ValueSet`.

An approximate version of an arithmetic operation $\mathrm{op}(x, y)$ (or $\mathrm{op}(x)$ for unary operation) is defined through the exact operation using a rounding function $\mathrm{rnd} : \mathbb{R} \to \mathbb{F}$ as $\mathrm{rnd}(\mathrm{op}(x, y))$ (or $\mathrm{rnd}(\mathrm{op}(x))$ for unary operation). In a similar way, an approximate version of an elementary function $\mathrm{fun}(x)$ is defined as $\mathrm{rnd}(\mathrm{fun}(x))$.

A binding of a certain version (implementation) to every arithmetic operation and elementary function for the number type signature is called a *context* in JInterval. The library provides the user with a number of contexts. An exact context consists of exact operations and functions. A function of the exact context generates the exception `IrrationalException` if a result of the function is irrational. This enables the user to decide how to organize further computations. When the use of an exact context is computationally too expensive, approximate contexts can be employed. The library provides approximate contexts for different number formats $\mathbb{F}$ and rounding functions $\mathrm{rnd}()$.

3.2.2 Implementation Level

It is natural to expect that the user will employ intervals with bounds represented by numbers of a binary floating-point type (especially of type `double`) more often than ones with rational bounds of general kind. That is why we paid special attention to the representation of the `ExtendedRational` type in these particular cases of rational numbers.

The underlying representation of rational numbers in `ExtendedRational` assumes a reduced form, i.e., there are no common prime divisors for numerator and denominator. Besides, powers of two are factored out of the numerator and denominator. Thus the representation of a rational number has the form $n/d \times 2^{exp}$, and, for $d = 1$, the values of `ExtendedRational` are binary floating-point numbers.

Class `ExtendedRational` encapsulates three different implementations specified as subclasses:

- subcalss `RationalImpl` implements the exact representation of rational numbers in a general form using two `BigInteger` fields for numerator and denominator and an `int` variable for the binary exponent;
- subcalss `BinaryImpl` implements the approximate representation as a binary floatin- point number, and consists of one `BigInteger` field for the mantissa and an `int` variable for the binary exponent;
- subcalss `BinaryDoubleImpl` describes a rational number which can be represented by a value of type `double`.

Implementations of the arithmetic operations and elementary functions over the extended rational numbers are not integrated immediately into `ExtendedRational` as methods. Instead, they are implemented in the separate class `ExtendedRationalOps` for exact versions of operations and functions and in context classes with the interface `ExtendedRationalContext` for approximate versions. So, if `x` and `y` are variables of

`ExtendedRational` type and `ctx` is a context, then, for example, addition can be performed as `ctx.add(x,y)`.

Since the underlying representations of rational numbers in `ExtendedRational` differ, implementations of binary arithmetic operations and elementary functions are based on distinct algorithms for pairs of rational numbers (a) in the general form, (b) in the form of floating-point numbers, and (c) in the form of `double`-values.

To start computations, the user must create an exact context or an approximate context with the necessary number format and rounding mode (see Listing 1).

Contexts are extendable and permit user implementations. For example, the library's contexts permit use of functions from the MPFR library for multiple-precision floating-point computations with correct rounding [11].

3.3 Interval Types

3.3.1 Mathematical Level

One of motives which determine the design of the library is to construct a universal domain of interval operations and elementary functions. The universal domain must be common for intervals of different flavors and must contain sets of classic intervals, set-intervals, and Kaucher intervals.

The universal domain is assumed to consist of

- degenerate intervals $[a, a]$;
- classic intervals $[a, b], a < b$;
- the empty interval;
- the entire interval $(-\infty, +\infty)$;
- semi-infinite intervals $(-\infty, b]$ and $[a, +\infty)$;
- improper intervals $[a, b], a > b$;

where a and b are rational numbers.

The signatures of interval operations and functions are common to intervals of different flavors. However, interval operations and functions of some flavors can give different results for the same arguments. In particular, interval functions of some flavors can be partial. For example,

- for classic intervals, $\sqrt{[-1,4]}$ is not defined;
- for set-valued intervals, square root is a partial function and $\sqrt{[-1,4]} = [0,2]$;
- for Kaucher intervals, $\sqrt{[-1,4]}$ is not defined.

Thus, we cannot unify interval operations and functions and, naturally, we have to logically separate a description of the universal domain and descriptions of interval operations of various flavors.

In the Java language, signatures of interval operations and functions can be expressed using interfaces, and can have different implementations (versions of operations or functions).

Exact versions of interval operations and functions are defined

- for the set-interval flavor, as natural interval extensions of real functions;
- for the Kaucher flavor, as $\mathbb{KR}$-extensions of real functions [12];
- for classic intervals, similar to the case of set-interval flavor excluding intervals with an infinite bound (or bounds) and empty set.

The universal domain is closed on interval arithmetic operations, but some interval functions can give results that lie outside the universal domain (for example, $\sqrt{[1,2]}$ gives an interval with irrational upper bound). Along with practical limitations, this circumstance explains the need for approximate versions of interval operations and functions in addition to mathematically exact ones. In P1788, exact operations and functions correspond to Level 1 while approximate versions match Level 2.

Results of approximate versions of interval operations or elementary functions belong to a finite subset of the universal domain. In terms of P1788, such a subset is called an "interval data type" and is denoted by $\mathbb{T}$. More precisely, the notion "interval data type" means that, at Level 2 of the standard, intervals are represented by a pair <Level 1 interval, type name> which explicitly specifies the value set and the associated version of operations and functions. This differs from the JInterval approach, but the library still provides classes emulating concepts of the standard.

According to P1788, an interval data type is often represented as the inf-sup type derived from a given number format $\mathbb{F}$. The inf-sup type is the bare interval type $\mathbb{T}$ comprising all intervals whose endpoints are in $\mathbb{F}$, together with the empty interval.

There are different ways of approximating an interval $[a, b], a, b \in \mathbb{R}$ corresponding to Level 1 of P1788 by a Level 2 interval $[c, d] \in \mathbb{T}$, $c, d \in \mathbb{F}$:

- containment approximation, which assumes $[c, d] \supseteq [a, b]$, i. e., $c \leq a \leq b \leq d$;
- Hausdorff approximation: $[c, d]$ is an interval closest to $[a, b]$ in the Hausdorff metric, i.e., $c = \mathrm{rnd}_{\mathrm{near}}(a)$, $d = \mathrm{rnd}_{\mathrm{near}}(b)$.

The quality of approximation achieved by an operation/function is indicated by accuracy mode. The P1788 standard distinguishes three accuracy modes for containment approximation: tightest, accurate and valid. 'Tightest' is the strongest of these three modes, and requires the best possible $\mathbb{T}$-interval enclosure as the resulting approximate interval of an operation or a function result. The approximation in 'accurate' accuracy mode also satisfies certain requirements on the width of the enclosure, but these requirements are weaker than in the 'tightest' mode. 'Valid' accuracy mode allows approximation of a mathematically correct result by any interval from $\mathbb{T}$ containing the mathematically exact interval of Level 1.

JInterval defines two additional accuracy modes: exact and so-called "fast" based on Hausdorff approximation. Exact accuracy mode gives a mathematically correct result if the produced interval belongs to the universal domain, i. e., has rational bounds. Fast accuracy mode assumes that the resulting interval from $\mathbb{T}$ has bounds nearest to the endpoints of the exact interval result. In Java, this accuracy mode can be easily implemented because nearest is the only rounding direction natively supported in Java.

Each accuracy mode is assumed to be implemented by a separate set of versions of operations and functions. In JInterval, such a set is called an *interval context.* It binds a certain version to every interval-valued operation and function required by the P1788 standard (hull, constructors, arithmetic operations, intersection, union, elementary functions).

3.3.2 Implementation Level

Following the basic principles, the library is designed to be extensible. Developers can supplement the library with their own interval flavors, underlying interval representations and implementations of interval operations and functions. At the same time, all implementations of different developers must be interoperable.

Interoperability is ensured by the fact that all individual classes representing interval types implement the common interface `Interval` (see Figure 1).

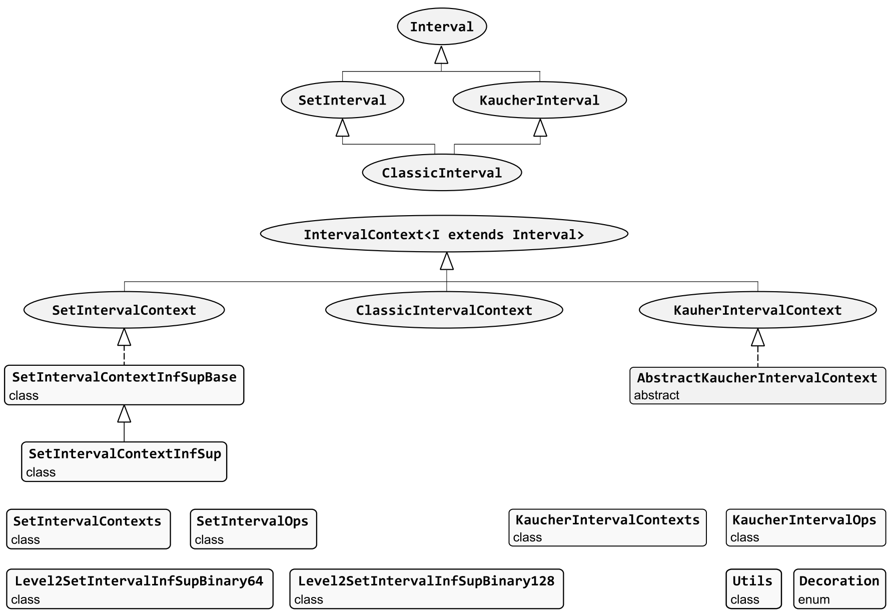

Figure 1: Class diagram for packages `net.java.jinterval.interval.*`

This interface consists of methods `i.inf()` and `i.sup()` returning bounds of an interval `i` as `ExtendedRational`-values and methods `i.doubleInf()` and `i.doubleSup()` which give approximate interval bounds of `i` as values of primitive type `double` independently of the underlying representation of `i`. Furthermore, `Interval` declares all the methods with non-interval result. Methods of one group, such as `i.wid()`, `i.mid()`, etc., return numeric characteristics of an interval `i` as values of `ExtendedRational` or `double` type. Methods of another group in `Interval` are boolean functions like `i.isEmpty()`, `i.containedIn(Interval)`, etc. The method `i.getDecoration()` which gives a decoration for an interval `i` is also declared in the interface `Interval`.

At the moment, the implementation of decorations in JInterval meets P1788 v. 6.1. This version of the standard does not yet explicitly separate interval flavors, and defines common decorations for all flavors. An interval decoration possesses one of the following values: `ILL` (ill-formed), `TRV` (trivial), `DEF` (defined), `DAC` (defined and continuous), and `COM` (common).

The interface `Interval` does not include interval-valued methods such as

- `Interval i.add(Interval)`,
- `Interval i.sin()`,
- etc.

Instead, these methods are declared in the interface `IntervalContext` of an interval context `ictx` as

- `Interval ictx.add(Interval, Interval)`,

- `Interval ictx.sin(Interval)`,
- etc.,

along with other interval-valued functions (constructors, arithmetic operations, elementary functions).

The library provides means to instantiate some predefined interval contexts, but the user may build his own contexts as well. Instances of a predefined interval context can be constructed using a special factory class. For example, to instantiate a context for a set-interval flavor, factory class `SetIntervalContexts` can be used. There exist a number of static methods in a factory class to construct interval contexts supporting different accuracy modes.

- The factory method `getExact()` returns a context with the exact accuracy mode. Functions in this context throw an exception if their exact result does not belong to the universal domain (i.e., at least one bound of the interval is an irrational number, such as `sqrt([1,2])`). Catching the exception allows the user to make a decision on a further computation process.
- The factory method `getInfSup(BinaryValueSet numberFormat)` constructs a context with the tightest accuracy mode for interval type $\mathbb{T}$ = `InfSup_`$\mathbb{F}$, where `numberFormat` specifies $\mathbb{F}$ (`binary16`, `binary32`, `binary64`, `binary128`, etc).
- The factory method `getFast()` constructs a context with the fast accuracy mode for interval type `InfSup_binary64`.

A simple program which illustrates how to create an interval context and perform simple interval calculations is shown in Listing 2 (see Section 4).

The above considerations and Listing 2 concern the use of set-intervals. However, the overall logic of computations using intervals of different flavors is completely analogous.

Interfaces for basic interval types describing different flavors (interface `SetInterval`, interface `KauherInterval`, interface `ClassicInterval`, etc.) are formed as extensions of the common base interface `Interval` (see Figure 1). From `Interval`, they inherit methods common to all flavors.

The signature of interval-valued operations and functions of different flavors is expressed by interfaces named *`Flavor`*`IntervalContext`, where *Flavor* is the name of an interval flavor (interface `SetIntervalContext`, interface `KaucherIntervalContext`, interface `ClassicIntervalContext`, etc.). Flavor interfaces are implemented in corresponding interval flavor contexts.

Interval interfaces *`Flavor`*`IntervalContext` of the different flavors have common features, so they all extend the same generic super-interface `IntervalContext<I extends Interval>` with type parameter `I` by binding `I` to the interval type with corresponding flavor. For example, interface `SetIntervalContext` binds `I` to `SetInterval` and extends interface `IntervalContext<SetInterval>`.

The standard representation of intervals belonging to the universal domain in JInterval internally uses hidden classes that describe both flavor and memory representation of interval bounds. Some of the classes specify classic intervals and, at the same time, implement interfaces of all three flavors: `SetInterval`, `KauherInterval`, `ClassicInterval`. Other classes designate those set-theoretic intervals that are not classic (infinite, semi-infinite and empty) and implement interface `SetInterval` only. The third kind of classes specifies improper intervals and implements only the interface `KauherInterval`. When an operation or a function computes resulting interval bounds, it creates an instance of those hidden classes which represents the result appropriately.

As mentioned above, the P1788 standard adopts another approach, where a Level 2 interval is defined as a pair <Level 1 interval, type name> explicitly specifying its set of values and associated version of operations and functions. JInterval emulates this concept using subclasses of the abstract class `AbstractLevel2SetInterval`. Each subclass designates a specific type of Level 2 intervals. An instance of a certain subclass contains Level 1 interval while type name is specified by the subclass itself because, in Java, every object is equipped with a label of its class. Classes `Level2SetIntervalInfSupBinary64` and `Level2SetIntervalInfSupBinary128` are examples of such subclasses corresponding to interval inf-sup types. Binary arithmetic operations are available only for intervals of the same class. That is why such an operation need no a context, and it is incorporated as a method in this class. For example, class `Level2SetIntervalInfSupBinary64` has the method

`Level2SetIntervalInfSupBinary64 add(Level2SetIntervalInfSupBinary64)`,

while class `Level2SetIntervalInfSupBinary128` contains the method

`Level2SetIntervalInfSupBinary128 add(Level2SetIntervalInfSupBinary128)`.

Number and interval types are employed by classes implementing vectors, matrices and high-level solvers for systems of interval linear equations, interval regressions, etc. The implementation of vectors and matrices in JInterval is at the very beginning and needs further development. At the moment, the library consists of only dense matrices with elements of `ExtendedRational` and `SetInterval` types. Vector and matrix operations have a syntax similar to that of Matlab, but need one more argument to specify a context (see Listing 4).

3.4 Module Structure

Physically, JInterval is a collection of Maven [1] subprojects (modules) with an aggregator project above them. A graph of module dependencies is shown in Figure 2.

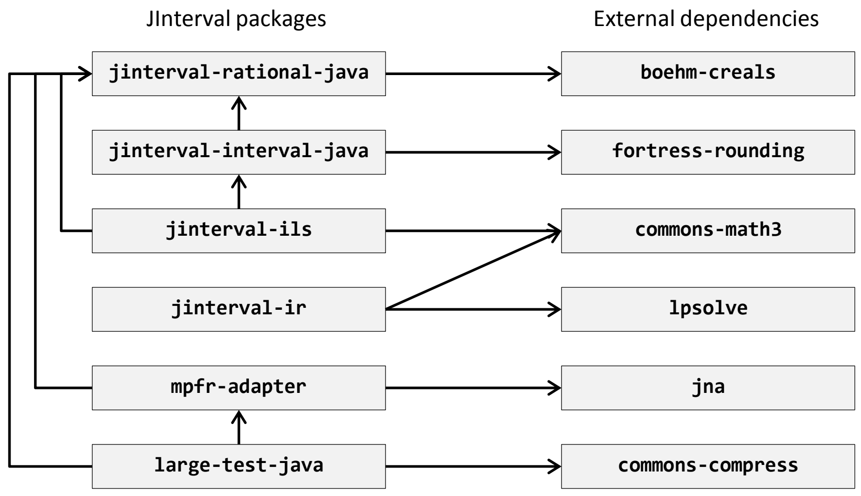

Figure 2: JInterval modules dependency graph.

Modules `jinterval-rational-java` and `jinterval-interval-java` form the core of the library and contain the definition of scalar and interval types. They depend on the rounding class from Fortress (`fortress-rounding`) [14] and constructive reals of H. Boehm (`boehm-creals`) used to construct the tightest interval enclosures of elementary functions. High-level solvers use, besides the core modules, external libraries for some standard methods like linear algebra and linear programming. Module `mpfr-adapter` is a JNA-interface for the native code of the MPFR library [11]. Tests for JInterval functionalities are placed in the subproject `large-test`.

4 Functionality and Examples

JInterval is an ongoing project. Thus, functionality provided by the library constantly grows. At the moment the library provides the user with the following possibilities.

The bottom functional layer is formed by basic numeric types: extended rational and floating-point arithmetic with flexible underlying representation.

Two types of arithmetic for intervals with bounds represented by an appropriate numeric type can be used: real set-based interval arithmetic and Kaucher ("modal interval") arithmetic. JInterval supports bare and decorated intervals, but, as mentioned above, the same decoration class is used for all interval flavors so far.

A list of interval elementary functions available in JInterval meets the P1788 standard, but for the Kaucher interval flavor not all of them have been implemented yet.

Dense vectors and matrices can be used both for extended rational numbers and intervals.

The layer of high-level interval methods is represented by solvers of interval linear equations systems and an interval regressions solver.

Listing 1: Simple rational calculations

```java
package Examples;

import net.java.jinterval.rational.ExtendedRational;
import net.java.jinterval.rational.ExtendedRationalContext;
import net.java.jinterval.rational.ExtendedRationalContexts;

public class EulerNumber {

    public static void evaluateTaylorExpansion(int N){
        ExtendedRationalContext rc;
        rc = ExtendedRationalContexts.exact();
        //rc = ExtendedRationalContexts.mkNearest(BinaryValueSet.BINARY32);

        ExtendedRational one = ExtendedRational.one(), fact = one, e = one;
        for(int n=1; n<=N; n++) {
            System.out.println("e = " + e + "  (" + e.doubleValue() + ")" );
            fact = rc.multiply(fact, ExtendedRational.valueOf(n));
            e = rc.add(e,rc.divide(one,fact));
        }
        System.out.println("e = " + e + "  (" + e.doubleValue() + ")" );
    }

    public static void main(String[] args) {
        // Approximate Euler number e using 10th degree Taylor polynomial for exp(1):
        // e = 1 + 1/2! + 1/3! + ... + 1/10!
        evaluateTaylorExpansion(10);    }
}
```

Interval linear system solvers implement the Gauss–Seidel method with the preliminary Hansen-Bliek-Rohn-Ning-Kearfott enclosure [18] and subdifferential Newton method [26] for finding formal solutions of interval linear systems in Kaucher arithmetic. Two and three dimensional AE-solution sets of interval linear systems can be visualized using a beta-version of a class that implements the boundary interval method by Irene A. Sharaya [25].

The interval regression solver assumes the simplest linear model with an interval uncertainty in response variable only, and allows checking data and model consistency, detecting outliers, computing interval parameters estimates and interval estimate of the response prediction [29, 30].

Hereafter we illustrate a style of JInterval use with several examples.

The first fragment of code (Listing 1) gives an example for the extended rational number type. The program computes a rational approximation to the Euler number using a Taylor series expansion. Initially, a rational context must be created using the factory class `ExtendedRationalContexts`. The precision of the result depends on the used context. In the code, the exact context is employed, but the commented line shows the call of constructor of the approximate context based on the floating-point type. The length of the type is specified by a constant in the argument of the constructor `mkNearest()`.

Listing 2: Evaluation of simple interval expressions

```java
package Examples;

import net.java.jinterval.interval.SetInterval;
import net.java.jinterval.interval.SetIntervalContext;
import net.java.jinterval.interval.SetIntervalContexts;

public class SimpleExpressions {

   private static void evaluateSimpleExpressions() {
        SetIntervalContext ic = SetIntervalContexts.getExact();
        // SetIntervalContext ic = SetIntervalContexts.getInfSup(BinaryValueSet.BINARY16);
        // SetIntervalContext ic = SetIntervalContexts.getInfSup(BinaryValueSet.BINARY32);
        // SetIntervalContext ic = SetIntervalContexts.getInfSup(BinaryValueSet.BINARY128);

        SetInterval x = ic.nums2interval(1,2);          // create x = [1,2] from numbers
        SetInterval y = ic.nums2interval(3,5);          // create y = [2,3] from numbers
        SetInterval z = ic.text2interval("[1/3, 4.5]"); // create interval z from string

        SetInterval s = ic.add(x,y);            // s = x + y
        SetInterval d = ic.divide(s,y);         // d = s / y

        System.out.println("x+y = " + s);
        System.out.println("(x+y)/y = " + d);

        if(d.containedIn(z)) {                  // if d in z
            ExtendedRational a = z.inf();       //    get z inf as a rational number
            ExtendedRational b = z.sup();       //    get z sup as a rational number
            System.out.println("[" + a + "," + b + "] contains " + d);
        } else {                                // else
            double r = z.doubleRad();           //    get double approximation of z radius
            System.out.println("rad z = " + r);
        }
    }

    public static void main(String[] args) {
        evaluateSimpleExpressions();
    }
}
```

Listing 2 shows how to start interval calculations. The user must create an interval context of the chosen flavor (set-interval context `ic` in the example) with the appropriate interval representation and rounding mode (compare the first line of the method **`evaluateSimpleExpressions()`** and possible alternatives in the commented lines below). If the exact context is turned on (using method **`getExact()`**), then interval bounds are represented by the extended rational type and the precision of all subsequent operations is absolute.

After instantiation of a context, the user can construct intervals and compute interval operations or functions calling the context's methods. Number- and boolean-valued characteristics of an interval can be obtained using methods of the interval itself (see the conditional operator in the code).

Listing 3: Interval version of Rump's example

```
package Examples;

import net.java.jinterval.interval.SetInterval;
import net.java.jinterval.interval.SetIntervalContext;
import net.java.jinterval.interval.SetIntervalContexts;

public class IntervalRumpExample {

    private static void evaluateIntervalRumpExpression(String message,
                                                       BinaryValueSet valueSet) {
        System.out.println("=== " + message + " ===");

        SetIntervalContext ic;
        if (valueSet != null) ic = SetIntervalContexts.getInfSup(valueSet);
        else ic = SetIntervalContexts.getExact();

        SetInterval x = ic.nums2interval(77617, 77617),
                    y = ic.nums2interval(33096, 33096),
                    ic11 = ic.nums2interval(11, 11),
                    ic121 = ic.nums2interval(121, 121),
                    ic2 = ic.nums2interval(2, 2),
                    ic5_5 = ic.nums2interval(5.5, 5.5),
                    ic333_75 = ic.nums2interval(333.75, 333.75);

        SetInterval i1 = ic.multiply(ic.subtract(ic333_75, pow(ic, x, 2)), pow(ic, y, 6)),
                    i2 = ic.multiply(pow(ic, x, 2),ic.subtract(ic.subtract(ic.multiply(
                         ic.multiply(ic11, pow(ic, x, 2)), pow(ic, y, 2)),
                         ic.multiply(ic121, pow(ic, y, 4))), ic2)),
                    i3 = ic.multiply(ic5_5, pow(ic, y, 8)),
                    i4 = ic.divide(x, ic.multiply(ic2, y));

        SetInterval i = ic.add(ic.add(ic.add(i1, i2), i3), i4);
        System.out.println("i = " + i);
    }

    public static void testIntervalRumpExpression() {
        evaluateIntervalRumpExpression("BINARY16", BinaryValueSet.BINARY16);
        evaluateIntervalRumpExpression("BINARY32", BinaryValueSet.BINARY32);
        evaluateIntervalRumpExpression("BINARY64", BinaryValueSet.BINARY64);
        evaluateIntervalRumpExpression("BINARY128", BinaryValueSet.BINARY128);
        evaluateIntervalRumpExpression("BINARY256", BinaryValueSet.BINARY256);
        evaluateIntervalRumpExpression("Exact", null);
    }

    public static void main(String[] args) {
        testIntervalRumpExpression();
    }
}
```

The program shown in Listing 3 computes $f(77617, 33096)$ for the expression

$$f(x, y) = (333.75 - x^2)y^6 + x^2(11x^2y^2 - 121y^4 - 2) + 5.5y^8 + x/(2y)$$

which is known as S. Rump's example [23, 16]. Numerical evaluation of this expression gives a misleading result, despite the use of increasing arithmetic precision. Evaluation in interval arithmetic produces wide intervals that contain the correct answer, and thereby exposes the instability. Increasing the precision narrows the width of the resulting intervals.

This example also demonstrates that even not so complex expressions lead to a wordy Java code. This inborn Java problem can be partly solved by defining appropriate wrappers. We have started developing a Scala interface for JInterval to provide the so called syntactic sugar to developers.

Listing 4: Hansen-Bliek-Rohn-Ning-Kearfott enclosure using `MatlabOps`

```
SetIntervalContext ctx = SetIntervalContexts.getInfSup(BinaryValueSet.BINARY64);
ExtendedRationalContext rcInv = ExtendedRationalContexts.mkNearest(BinaryValueSet.BINARY64);
ExtendedRationalContext rcDown = ExtendedRationalContexts.mkFloor(BinaryValueSet.BINARY64);
ExtendedRationalContext rcUp = ExtendedRationalContexts.mkCeiling(BinaryValueSet.BINARY64);
IntervalVector x, dA;
RationalMatrix C, B, v, Cv, R, w;
RationalVector dC, u, d, alpha, beta, dlow, vw;
Rational negunit = Rational.valueOf(-1);           // INTLAB code of HBRNK enclosure

int n = dim(A);                                    // n = dim(A);
dA = diag(A);                                      // dA = diag(A);
C = compmat(A);                                    // C = compmat(A);
B = inv(rcInv, C);                                 // B = inv(C);
v = abs(mul(rcInv, B, ones(n, 1)));                // v = abs(B*ones(n,1));
                                                   // SetRoundDown;
Cv = mul(rcDown, C, v);                            // Cv = C*v;
if (!(Cv.getMinValue().signum() > 0)) {            // if (~all(min(Cv))>0)
   x = new IntervalVector(n,ctx.entire());         //    x = midrad(0,inf+zeros(n,1))
} else {                                           // else
   dC = diag(C);                                   //    dC = diag(C);
   R = sub(rcDown, mul(rcDown, C, B), eye(n));     //    R = C*B - eye(n,n);
                                                   //    SetRoundUp;
   w = zeros(1, n);                                //    w = zeros(1,n);
   for (int i = 0; i < n; i++) {                   //    for i = 1:n
       w = max(w, div_(rcUp,                       //       w = max(w,
          neg(R.getSubMatrix(i, i, 0, n - 1)),     //               (-R(i,:))/
          Cv.getEntry(i, 0)));                     //               Cv(i));
   }                                               //    end;
   vw = mul_(rcUp,                                 //
      v.getColumnVector(0), w.getRowVector(0));    //
   dlow = sub(rcUp, vw, B.diag());                 //    dlow = v.*w-diag(B);
   dlow = neg(dlow);                               //    dlow=-dlow;
   B = add(rcUp, B, mul(rcUp, v, w));              //    B = B + v*w;
   u = mul(rcUp, B, b.mag());                      //    u = B*abs(b);
   d = B.diag();                                   //    d = diag(B);
   alpha = add(rcUp, dC, div_(rcUp, negunit, d));  //    alpha = dC + (-1)./d;
   beta = sub(rcUp, div_(rcUp, u, dlow), b.mag()); //    beta = u./d-abs(b);
   x = div_(ctx,                                   //    x =
     add(ctx, b, midrad(Rational.zero(), beta)),   //        (b + midrad(0,beta))./
     add(ctx, dA, midrad(Rational.zero(), alpha))); //       (dA + midrad(0, alpha);
}                                                  // end
```

Scala is a modern general purpose programming language which mixes features of object-oriented and functional languages [19]. Scala programs run on the Java VM, are byte code compatible with Java and allow calls to Scala from Java and vice versa. Scala is equipped with expressive syntactic features (operator overloading, implicit arguments, etc.) which allow one to cut away unneeded syntactic overhead and make domain-specific language extensions. It enables us to express the relatively complex semantics of interval computations with flexible representation and different interval flavors in the common and natural syntax of an interval formula. Scala code sizes are typically reduced by a factor of 3 when compared to an equivalent Java application.

Another useful wrapper implemented in JInterval is the `MatlabOps` class, which introduces Matlab-like named functions such as `zeros()`, `diag()`, `inv()`, `compmat()`, etc. `MatlabOps`, being inherited in user classes, can essentially shorten a code and simplify the use of JInterval for Matlab/Intlab [24] aware programmers. Listing 4 shows the procedure of Hansen-Bliek-Rohn-Ning-Kearfott coded line-by-line both using JInterval+`MatlabOps` and Matlab+Intlab.

We have restricted ourselves only to several short examples. In the module `jinterval-demo` of the library [5], an interested reader can find more comprehensive examples: the use of interval decorations, iterations with the logistic equation, Hilbert matrix inversion, use of interval linear systems solvers, and so on.

5 Applications

Several applications in different areas were built using JInterval. Two of them are briefly described in this section.

5.1 P1788 Test Framework

On top of JInterval's tightest implementation of interval operations and functions, a simple framework for testing P1788 compliance of third-party libraries is developed.

The main component here is the Launcher, which is able to load dynamic libraries (.so/.dll) with third party implementation of P1788 and to check the results obtained from a library with the tightest results computed internally using JInterval.

The launcher reads tests from plain text files. An example of a simple input test set is presented in Listing 5. The output of the Launcher for filib, boost and MPFI is shown in Listing 6. The lines of the Launcher report have the following format:
`<test> = <expected result> :  <computed result> <comment>`.
The expected result of an operation is calculated using JInterval while the computed result is provided by a tested library.

Listing 5: File of tests

```
* div
[1,2] [0,1]
[1,2] [0,0]

* sqrt
[-Infinity,0]
[-Infinity,Infinity]

* pown
[0,0] 0
```

5.2 KNIME Interval Tools

KNIME [8] is a modular open source data analysis platform that enables the user to visually create data workflows, selectively execute some or all analysis steps, and investigate the results through interactive views of data and models. Due to open interfaces, KNIME allows easy integration of different data loading, processing, transformation, analysis and visual exploration modules without the focus on any particular application

Listing 6: Test launcher report

```
== Filib 3.0.2
div [1.0,2.0] [0.0,1.0] = [1.0,Infinity] : [1.0,Infinity] Ok
div [1.0,2.0] [0.0,0.0] = [EMPTY] : [1.7976931348623157E308,Infinity] NOT TIGHT!
sqrt [-Infinity,0.0] = [0.0,0.0] : [0.0,0.0] Ok
sqrt [-Infinity,Infinity] = [0.0,Infinity] : [-4.9E-324,Infinity] NOT TIGHT!
pown [0.0,0.0] 0 = [1.0,1.0] : [1.0,1.0] Ok
==

== Boost 1.48.0
div [1.0,2.0] [0.0,1.0] = [1.0,Infinity] : [1.0,Infinity] Ok
div [1.0,2.0] [0.0,0.0] = [EMPTY] : [EMPTY] Ok
sqrt [-Infinity,0.0] = [0.0,0.0] : [0.0,0.0] Ok
sqrt [-Infinity,Infinity] = [0.0,Infinity] : [0.0,Infinity] Ok
pown [0.0,0.0] 0 = [1.0,1.0] : [EMPTY] CONTAINMENT FAILURE!!!
==

== MPFI 1.5.1
div [1.0,2.0] [0.0,1.0] = [1.0,Infinity] : [1.0,Infinity] Ok
div [1.0,2.0] [0.0,0.0] = [EMPTY] : [-Infinity,Infinity] NOT TIGHT!
sqrt [-Infinity,0.0] = [0.0,0.0] : [EMPTY] CONTAINMENT FAILURE!!!
sqrt [-Infinity,Infinity] = [0.0,Infinity] : [EMPTY] CONTAINMENT FAILURE!!!
Library has no Operation "pown" in line 7 : * pown
==
```

area. Such popular tools as WEKA (machine learning environment), R Project (statistical programming language), Python (scripting language), ImageJ (image processing and analysis program), Octave, Matlab can be integrated into KNIME data processing and analysis workflows. A standard functionality of KNIME can be extended with user-defined plug-ins. A great number of additional workflow nodes are developed by the user community as well as by commercial vendors for chemical applications, image and text analysis, data mining, etc.

A collection of interval plug-ins for the data mining platform KNIME has been developed. The collection includes an interval regression builder, an outlier detector, and an ILS solver that employs JInterval high-level solvers. KNIME-node "ILS Solver" enables the user to construct inner and outer estimates of the united solution sets. ILS Solver also can be used to visualize 2D and 3D united solution sets of ILS. These views are based upon the Java-applet which has been developed by Gregor Paw in his bachelor's thesis [20], and implements W. Krämer's visualization algorithm [15].

Interval tools can be easily and flexibly combined with the other functional nodes of KNIME to construct analytical procedures. Simple a KNIME workflow that builds an interval regression model and estimates predictions using it is shown in Figure 3. Similarly, a workflow implementing an image recognition algorithm [21] is constructed. This workflow inputs images with bounded noise and detects the best similar pattern among several predefined ones. The metrics used to calculate similarity of images and patterns is based on outer estimates of the united solution sets of interval linear systems of a special kind solved using ILS Solver.

6 Conclusions and Perspectives

In this paper we have presented JInterval, a new library for interval computations in Java. JInterval follows the draft of P1788 standard for interval arithmetic and pro-

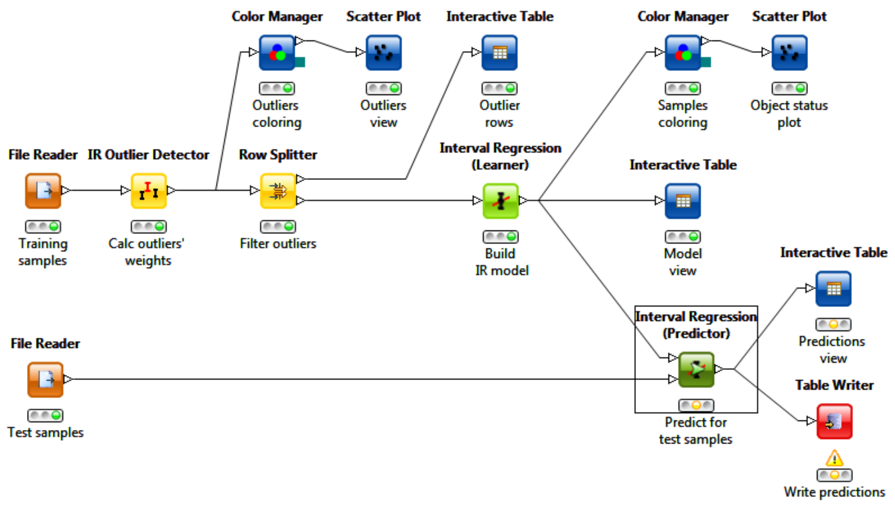

Figure 3: KNIME workflow for interval regression.

vides two interval flavors: set intervals and Kaucher intervals with flexible underlying representation of interval bounds and rounding policy controllable by the user. Dense vectors and matrices are supported on top of the extended rational and interval arithmetics. The most important functional layer of JInterval consists of high-level solvers for typical interval problems: systems of interval linear equations and interval regression.

The functionality of JInterval enables the integration of interval computations and analysis tools into a wide variety of applications and software developed in Java or/and Java Virtual Machine. Data mining, distributed computations, software for mobile devices, and industrial applications may be targets for JInterval employment.

JInterval is a continuing project, and in the nearest future most efforts will be aimed at the maintenance of the compliance of JInterval with the P1788 standard. The standard is close to completion, and JInterval could be the basis for a reference implementation of P1788 in Java.

Yet another goal is to increase the performance of JInterval by providing optional native coded plug-ins optimized for certain platforms.

Considering JInterval as a kind of run-time library for JVM, we plan to develop an API for other languages. Development of a Scala version of JInterval has already started.

The rich content of the high-level functionality of JInterval is one of the most valuable issues for applied software developers. Therefore, an enhancement of JInterval with implementations of interval analysis methods and solvers remains the foreground task. We plan to incorporate the following components in JInterval: ILS tolerable solution set estimators, a global optimization solver [27] and a verified ODE solver [17].

JInterval is an open source project, and new contributors are welcome. The project resources (source code, wiki, mailing list, etc.) are hosted at the `java.net` community web site [5].

Acknowledgements

We would like to acknowledge all the people who contributed to or influenced the JInterval project: Maksim V. Danilov, Kirill S. Dronov, Walter Krämer, Nikita V. Panov, Gregor Paw, Anton E. Sartakov, Andrey S. Samoilov, Irene A. Sharaya, Egor N. Tepikin. We would like to address special thanks to Prof. Sergey P. Shary, who has been strongly supporting JInterval from the very beginning, and to IEEE Interval Standard Working Group [3] for their valuable job.

References

[1] Apache Maven Project. `http://maven.apache.org`.

[2] Hadoop. Documentation and open source release. `http://hadoop.apache.org`.

[3] IEEE Interval Standard Working Group - P1788.
`http://http://grouper.ieee.org/groups/1788`.

[4] The Java Tutorials. Oracle. `http://docs.oracle.com/javase/tutorial/`.

[5] JInterval. Java library for interval computations. `http://jinterval.java.net`.

[6] Let's add intervals to Java (a proposal).
`http://www.cs.utep.edu/interval-comp/java.html`.

[7] Benjamin R. C. Bedragal and José E. M. Dutra. Java-XSC: Estado da arte. In *XXXII Conferencia Latinoamericana de Informatica (CLEI-2006), Santiago, August, 2006*.

[8] Michael R. Berthold, Nicolas Cebron, Fabian Dill, Thomas R. Gabriel, Tobias Kötter, Thorsten Meinl, Peter Ohl, Christoph Sieb, Kilian Thiel, and Bernd Wiswedel. KNIME: The Konstanz Information Miner. In *Studies in Classification, Data Analysis, and Knowledge Organization (GfKL 2007)*. Springer, 2007.

[9] Jeffrey Dean and Sanjay Ghemawat. MapReduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, January 2010.

[10] Jose E. M. Dutra. Java-XSC: Uma Biblioteca JAVA para Computações Intervalares. Master's thesis, Universidade Federal do Rio Grande do Norte, 2000.

[11] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software*, 33(2):13:1–13:15, June 2007.

[12] Alexandre Goldsztejn. Modal intervals revisited, part 1: A generalized interval natural extension. *Reliable Computing*, 16:130–183, 2012.

[13] Timothy J. Hickey, Zhe Qju, and Maarten H. Van Emden. Interval constraint plotting for interactive visual exploration of implicitly defined relations. *Reliable Computing*, 6(1):81–92, 2000.

[14] Guy L. Steele Jr., Eric E. Allen, David Chase, Christine H. Flood, Victor Luchangco, Jan-Willem Maessen, and Sukyoung Ryu. Fortress (Sun HPCS language). In David A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 718–735. Springer, 2011.

[15] Walter Krämer. Computing and visualizing solution sets of interval linear systems. *Serdica Journal of Computing*, 1(4):455–468, 2007.

[16] Eugene Loh and G. William Walster. Rump's example revisited. *Reliable Computing*, 8(3):245–248, 2002.

[17] Dmitry Nadezhin. A differential inequalities method for verifed solution of IVPs for ODEs using linear programming for the search of tight bounds. In *Abstracts of 13th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetics and Verified Numerics SCAN-2008, El Paso, Texas, September 29 - October 3, 2008. – El Paso, Texas, 2008*, pages 78–79.

[18] Arnold Neumaier. A simple derivation of the Hansen-Bliek-Rohn-Ning-Kearfott enclosure for linear interval equations. *Reliable Computing*, 5:131–136, 1999.

[19] Martin Odersky et al. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004. `http://scala-lang.org/docu/files/ScalaOverview.pdf`.

[20] Gregor Paw. Ein intuitiv bedienbares Java-Applet zur Visualisierung exakter Lösungsmengen von mengenwertigen numerischen Problemen. Bachelor-thesis, University of Wuppertal, 2006.

[21] Alexander V. Prolubnikov. An interval approach to pattern recognition of numerical matrices. *Reliable Computing*, 19(1):107–119, 2013.

[22] John Pryce and Cristian Keil (Tech Eds.). P1788: IEEE Draft Standard for Interval Arithmetic.

[23] Siegfried M. Rump. Algorithms for verified inclusions: theory and practice. In R. E. Moore, editor, *Reliability in Computing: The Role of Interval Methods in Scientific Computing*, volume 19, pages 109–126. Academic Press, Boston, 1988.

[24] S.M. Rump. INTLAB — INTerval LABoratory. In Tibor Csendes, editor, *Developments in Reliable Computing*, pages 77–104. Kluwer Academic Publishers, Dordrecht, 1999. `http://www.ti3.tuhh.de/rump/`.

[25] Irene A. Sharaya. Boundary intervals and visualization of AE-solution sets for interval systems of linear equations. In *Book of abstracts of 15th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetics and Verified Numerics SCAN-2012, Novosibirsk, Russia, September 23-29, 2012. – Novosibirsk: Institute of Computational Technologies, 2012*, pages 166–168. Slides: `http://conf.nsc.ru/files/conferences/scan2012/142985/Sharaya-scan2012.pdf`.

[26] Sergey P. Shary. Algebraic approach to the interval linear static identification, tolerance, and control problems, or one more application of Kaucher arithmetic. *Reliable Computing*, 2:3–33, 1996.

[27] Sergey P. Shary. Randomized algorithms in interval global optimization. *Numerical Analysis and Applications*, 1:376–389, 2008.

[28] Guillermo L. Taboada, Sabela Ramos, Roberto R. Exposito, Juan Touriño, and Ramón Doallo. Java in the High Performance Computing arena: Research, practice and experience. *Science of Computer Programming*, 2011. (In press). `http://dx.doi.org/10.1016/j.scico.2011.06.002`.

[29] Sergei I. Zhilin. On fitting empirical data under interval error. *Reliable Computing*, 11(5):433–442, 2005.

[30] Sergei I. Zhilin. Simple method for outlier detection in fitting experimental data under interval error. *Chemometrics and Intelligent Laboratory Systems*, 88(1):60–68, 2007.