Interval Extensions of Multivalued Inverse Functions

The Implementation of Interval Relational Arithmetic in gaol

FRÉDÉRIC GOUALARD

Université de Nantes, Nantes Atlantique Université, CNRS, LINA, FRE 2729

2 rue de la Houssinière, BP 92208, F-44000 NANTES

The implementation of inverse functions provided by most interval arithmetic software libraries is restricted to bijective functions and to the principal branch of multivalued functions. On the other hand, some algorithms—most notably, constraint propagation algorithms—require multivalued inverse functions as well. We present in details in this paper the algorithms to implement interval arithmetic extensions of the following multivalued inverse functions: the inverse integral power, the inverse cosine, the inverse sine, the inverse tangent, the inverse hyperbolic cosine, and the inverse multiplication. The issues raised by their effective as well as efficient implementation with floating-point numbers in the gaol C++ library are carefully addressed.

1. INTRODUCTION

Given the relation $y = \cos x$, where x lies in the interval $[10, 14]$, *interval arithmetic* [Moore 1966] will readily allow us to compute the possible values for y by considering the monotonic subdomains of the cosine function over $[10, 14]$: $y \in [\cos 10, 1] \approx [-0.84, 1]$. On the other hand, what is the possible domain for an unknown x if the domain for y is $[-0.3, 0.2]$? Most interval arithmetic libraries will fix it at $[\mathrm{acos}\, 0.2, \mathrm{acos}\, {-0.3}] \approx [1.36, 1.87]$ because they consider branch cuts of the multivalued inverse cosine to return principal values in the domain $[0, \pi]$ only. Now, what if we knew that x lay in the domain $[20, 26]$? The aforementioned inverse cosine interval function would not be of much help here, while considering a multivalued inverse cosine would allow us to restrict the domain of x to $[6\pi + \mathrm{acos}\, 0.2, 8\pi - \mathrm{acos}\, 0.2] \approx [20.22, 23.77]$ (see Figure 1).

Such a use of relations between variables together with domains of possible val-
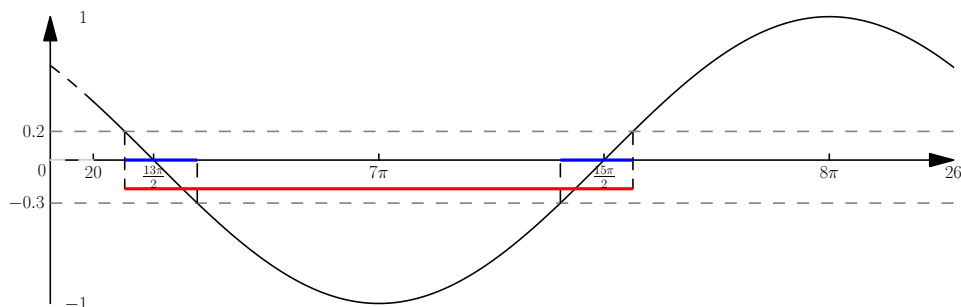
Fig. 1.   Computing the inverse cosine of $[-0.3, 0.2]$ with respect to the interval $[20, 26]$

ues to infer tighter consistent domains is the core principle of *Constraint Programming* [Dechter 2003; Sutherland 1963]. *Interval Constraint Programming* [Benhamou 2001; Davis 1987] is an instance of this framework that considers relations over continuous variables whose domains are represented by intervals with floating-point number bounds. It combines smart propagation algorithms [Mackworth 1977] with original domain-contracting algorithms to solve linear and nonlinear continuous problems, with very good performances on some difficult problems when compared to traditional mathematical methods [Granvilliers and Benhamou 2001]. Some of these contracting methods (most notably, HC3 [Benhamou 1995; Lhomme 1993], HC4 [Benhamou et al. 1999], and their derivatives), make extensive use of interval extensions of the inverse of usual mathematical functions, be they bijective or not. Since the pioneering work of Cleary [Cleary 1987] on interval relational arithmetic, algorithms to compute the interval extensions of inverse functions have been repeatedly devised for the exclusive benefit of various solving systems (BNR Prolog [Older and Vellino 1990], Prolog IV [Benhamou and Touraïvane 1995], Ilog Solver [Puget 1994], and RealPaver [Granvilliers and Benhamou 2006], among others) up to the point they are almost considered as part of the computing lore. To our knowledge, however, no detailed description of these algorithms has ever been published—to the important exception of the algorithm for the inverse multiplication [Hickey et al. 2001]. As a consequence, the existing implementations may vary widely in reliability, speed, and in the functions supported.

This paper is an attempt to straighten this situation out at a moment when interval arithmetic is being considered for addition to the standard library of such an ubiquitous language as C++ [Brönnimann et al. 2006b] . We stress the differences between standard interval inverse operators and interval multivalued inverse operators in Section 2, retracing meanwhile the history of interval relational arithmetic, which motivated their definition. Related work is discussed in Section 3, where we present the tools and libraries that implement interval relational arithmetic, contrasting their approach with our own work in the *gaol* C++ library [Goualard 2007]. The algorithms for the inverse power function, the inverse cosine, the inverse sine, the inverse tangent, and the inverse hyperbolic cosine are presented in Section 4; for the sake of completeness, we also give the formulas already published by Hickey and others [Hickey et al. 2001] for the inverse multiplication. The ac-

tual implementation of these algorithms in *gaol* is considered in details in the same section. It is then compared in Section 5 with respect to quality and performance with another freely available library.

2. FROM INTERVAL ARITHMETIC TO INTERVAL RELATIONAL ARITHMETIC

Though its history predates the development of computers,[1] *interval arithmetic* gained most of its momentum as a drop-in replacement for *floating-point numbers* [IEEE 1985] to reason over sets and to overcome the weaknesses of computer arithmetic [Warmus 1956; Sunaga 1958; Moore 1966]. Therefore, most of the early works in the domain focused on extensions to intervals of well-known mathematical methods such as Newton-Raphson's [Moore 1966] and Gauss-Seidel's [Hansen and Sengupta 1981]. As a consequence, the interval operators defined were, for the most part, straight extensions to intervals of their floating-point counterpart, with multivalued inverse functions restricted to their principal branch.

Even now, most of the software libraries available [Yohe 1979; Aberth and Schaefer 1992; Knüppel 1994; Wiethoff 1996; Rump 1999; Microsystems 2002; Revol and Rouillier 2005; Lerch et al. 2006] only offer the limited set of interval operators necessary to support a *functional approach* to mathematical algorithms.

Such an approach is well suited to the *imperative programming* paradigm, where *assignment* is the crux of the instruction set. There are, however, some paradigms, such as *Logic Programming* [Kowalski 1974], based not on the idea of functions, but on relations: In a language like Prolog [Clocksin and Mellish 1981], the programmer should not write statements like:

$$y \leftarrow \cos x$$

that sets the variable y to the cosine of x, as a Prolog program should not have definite input and output. Hence the use, instead, of relations binding the values of x and y:

$$y = \cos x,$$

where both y and x may take the role of the unknown.

Until the work by Cleary [Cleary 1987], however, Prolog's arithmetic was not up to Logic Programming expectations since it was using the infamous "`is/2`" predicate, which was purely functional in nature. Cleary introduced the use of intervals to enclose real values with certainty, together with rules to project an arithmetic relation onto each of the variables involved. The relation $y = \cos x$, for example, would then be translated into two functional statements:

$$\begin{cases} \mathbf{I_x} \leftarrow \mathbf{I_x} \cap (\cos^{-1} \mathbf{I_y}) \\ \mathbf{I_y} \leftarrow \mathbf{I_y} \cap (\cos \mathbf{I_x}) \end{cases} \tag{1}$$

making use of the inverse cosine "$\cos^{-1}$," with $\mathbf{I_x}$ and $\mathbf{I_y}$ the domains of x and y respectively.

---

[1]Refer to the web site *Interval Computations* at `http://www.cs.utep.edu/interval-comp/` and, in particular, to its page *Early Papers on Interval Computations* for references to the relevant works.

Obviously, the interval extension of "$\cos^{-1}$" has to be markedly different from that of the usual arccosine function as it shall not be restricted to the principal branch of the inverse cosine. By definition, we have:

$$\begin{cases} \text{acos}\,\mathbf{I_y} & = \text{cch}\left(\{x \in [0, \pi] \mid \exists y \in \mathbf{I_y} : x = \text{acos}\,y\}\right) \\ \cos^{-1}\mathbf{I_y} & = \text{cch}\left(\{x \in \mathbb{R} \mid \exists y \in \mathbf{I_y} : y = \cos x\}\right) \end{cases} \tag{2}$$

where the "cch" operator is a function from the power set $\mathcal{P}(\mathbb{R})$ of $\mathbb{R}$ to the *set of intervals with floating-point bounds* $\mathbb{I}$ that returns the smallest interval (in the sense of set inclusion) containing its argument.[2]

The definition of "$\cos^{-1}$" given in Eq. (2) is evidently useless since the only interval such a function would return for any argument $\mathbf{I_y}$ satisfying $\mathbf{I_y} \cap [-1, 1] \neq \varnothing$ would be $[-\infty, +\infty]$.[3] The same would hold for all periodic functions. A simple way to tackle this problem is to consider an $(n+1)$-ary inverse operator $\top^{-1}$ for any n-ary operator $\top$, which computes the inverse operation on some n-tuple $(x_1, \ldots, x_n)$ with respect to some variable x_{n+1}. For example, for the cosine function, we obtain the inverse cosine "acos_rel" defined by:

$$\text{acos\_rel}(\mathbf{I_y}, \mathbf{I_x}) = \text{cch}\left(\{x \in \mathbf{I_x} \mid \exists y \in \mathbf{I_y} : y = \cos x\}\right).$$

Cleary's seminal paper restricted itself to addition and multiplication relational forms, and did not consider cosine or any other trigonometric relation. More to the point, his paper did not give any algorithm in sufficient details to implement the inverse operators needed. Following Cleary's work on a better Prolog arithmetic, Davis [Davis 1987], Hyvönen [Hyvönen 1989; 1995], and Older & Vellino [Older and Vellino 1990] investigated the use of interval arithmetic to solve constraint systems, which led them to consider interval relational arithmetic too. Except for some considerations on the inversion of the square function, Davis' paper does not give algorithms to implement inverse functions; the same holds for Hyvönen's papers, even though *reversed mappings*—his name for inverse functions—appear prominently in the LIA InC++ software library manual [Hyvönen 1995]. In 1989, Older [Older 1989] gave one of the first published algorithms to handle the relation $x \times y = z$ with the three functional forms:

$$\begin{cases} \mathbf{I_x} \leftarrow \text{div\_rel}(\mathbf{I_z}, \mathbf{I_y}, \mathbf{I_x}) \\ \mathbf{I_y} \leftarrow \text{div\_rel}(\mathbf{I_z}, \mathbf{I_x}, \mathbf{I_y}) \\ \mathbf{I_z} \leftarrow \mathbf{I_x} \times \mathbf{I_y} \end{cases}$$

making use of the ternary inverse multiplication "div_rel" defined by:

$$\text{div\_rel}(\mathbf{I_z}, \mathbf{I_y}, \mathbf{I_x}) = \text{cch}\left(\{x \in \mathbf{I_x} \mid \exists y \in \mathbf{I_y} \exists z \in \mathbf{I_z} : xy = z\}\right).$$

It is, however, presented as a Prolog program without any explanation or proof of correctness, nor does it take into account the delicate problem of *outward rounding*.

---

[2]This operator is often called "hull" (and sometimes noted "□") in the interval constraint programming literature. In this paper, however, we adopt the notations sponsored by Kearfott and others [Kearfott et al. 2005].

[3]We consider here the compactification of $\mathbb{R}$ as the real projective line, as described by the IEEE 754 standard [IEEE 1985].

In the nineties, Benhamou and Lhomme concurrently formalized the works by Older and Vellino on BNR(Prolog) and clp(BNR), establishing the foundations for Algorithms HC3 [Benhamou 1995] and Ref_filtering [Lhomme 1993], which both rely on interval relational arithmetic. Since then, these two algorithms have been used repeatedly as building blocks for ever complex methods to solve difficult nonlinear problems with increasing performances [Benhamou et al. 1999; Ceberio and Granvilliers 2000; Granvilliers and Benhamou 2006; Fränzle et al. 2007].

Meanwhile, the available information on how to implement interval relational arithmetic remained scarce, and was mostly restricted to a series of papers by Hickey and his co-authors: An algorithm for the relation $y = \exp x$ was presented in 1996 [Hickey and Ju 1996] (the paper was primarily focused on performances, as the computation of both exp and $\exp^{-1}$ do not require a complex machinery *per se*, the exponential function being invertible); algorithms to handle the multiplication relation $x \times y = z$ were presented in two subsequent papers [Hickey and Ju 1997; Hickey et al. 2001], the last one being considered as the definitive work on the subject. The work by Hickey and others on interval relational arithmetic led to the implementation of the *smath* software library [Hickey 2005], one of the few freely available libraries offering multivalued inverse functions. This library, along with other libraries and tools, is considered in the next section.

3.   AVAILABILITY OF INTERVAL RELATIONAL ARITHMETIC

Original implementations of interval relational arithmetic are present in several constraint solvers, such as *Mozart* [Mejias et al. 2006], *ECLIPSe* [Cheadle et al. 2007], *Realpaver* [Granvilliers 2004], *INCLP(ℝ)* [De Koninck et al. 2006] or *Prolog IV*, to name a few of the most prominents and most recents.[4] When embedded in such tools, interval relational arithmetic is usually not directly accessible to users. One has to choose among the few software libraries that implement it to get an unfettered access:

—*Ilog Solver* [Ilog Inc. 2007] is a closed-source commercial product. It is much more than an interval library only, as it also offers many constraint propagation algorithms;

—*smath* [Hickey 2005] is the C library by Hickey alluded to in the previous section. It lacks some operators (e.g., hyperbolic inverse functions) but it is a fast and lightweight library that can be mastered in its entirety very quickly;

—*Boost* [Dawes et al. 2007] is a huge C++ library seeking to be a reference implementation of many algorithms for the C++ stan dard. It contains an interval arithmetic section [Melquiond et al. 2007; Brönnimann et al. 2006a], in which interval relational arithmetic is still, however, at an early stage. It has indeed not found its way into the current release yet.[5] Still, it is a promising, very customizable library. For better or for worse, it relies on advanced features of the C++ standard that are not widely available yet.

To our knowledge, most of the algorithms used by these tools and libraries have never been published. There may then be significant differences in reliability and

---

[4]Note that *Prolog IV* is no longer publicly available, though.
[5]This is the case as of Version 1.34.1 of the library.

performances of the various implementations. They also differ in the relations supported (*INCLP(ℝ)*, for example, does not implement trigonometric, hyperbolic and other transcendental inverse functions).

Originating from our work on constraint programming algorithms, *gaol*[6] [Goualard 2007] is a C++ library implementing interval extensions of functional as well as relational operators. It has served as a model for Section 26.6.15 on interval relational operators of the proposal to add interval arithmetic to the C++ standard library [Brönnimann et al. 2006b]. It has been freely available on Sourceforge since October 2002 with a *GNU Lesser General Public license*,[7] and has been downloaded ca. 1400 times as of November 2007. To our knowledge, it is being used in the following constraint solvers: *OpenSolver* [Zoeteweij and Arbab 2004], *Constraint Explorer* [Zimmer 2004] by Dassault Aviation, and *Elisa* [Christie et al. 2005]. It is also being used in various research projects (e.g., Feydy and Stuckey's work on using an interval approach to solve linear equations [Feydy and Stuckey 2007]).

We present in the next section what we believe to be the first detailed description of the algorithms to implement interval relational arithmetic in the hope it can serve as a common ground for future works on the subject, and as a starting point for better, faster algorithms. In our view, the only freely available and mature library that offers roughly the same features as *gaol* is *smath*. We therefore compare in Section 5 our implementation with *smath*'s as for performances and tightness of the results.

4.   THE IMPLEMENTATION OF INTERVAL RELATIONAL ARITHMETIC IN GAOL

Gaol is a software library of around 20,000 lines of C++ code (comments included). It used to be available on SUN Sparc/Solaris, Microsoft Windows/ix86 and Linux/i86. For lack of time, only the last two platforms are currently actively supported. Porting it to other platforms should not be difficult, though, provided a C99 standard-compliant compiler [C99 1999] be available, as we rely on several C99 functions for rounding operations.

For the sake of readability the following algorithms are presented in a simple pseudo-language that is almost, but not quite, entirely unlike *Python* [Python Software foundation 2007]. In addition, we use the following notations: The left (resp. right) bound of an interval $\mathbf{I}$ is written $\underline{\mathbf{I}}$ (resp. $\overline{\mathbf{I}}$). The bounds of the intervals are IEEE 754 floating-point numbers in the `double` format (that is, representation on 64 bits, with 1 sign bit, 11 bits for the biased exponent, and 52 bits of fractional part, as specified by the standard [IEEE 1985]). Let $\mathbb{F}$ be the set of such floating-point numbers. Any number that is not perfectly representable with a `double` is rounded: $\mathrm{fl}_{\bigtriangledown}(x)$ (resp. $\mathrm{fl}_{\bigtriangleup}(x)$) rounds the real x to the largest `double` smaller or equal to x (resp. smallest `double` larger or equal to x). The empty interval is represented by "∅." It is implemented by intervals where one of the bounds at least is an NaN (*Not A Number*) [Goualard 2000]. An interval whose left bound is strictly

---

[6]The game of the name: The first name of *gaol* was *jail*, standing for *Just Another Interval Library*, until we realized that having relational operators as well as functional ones—a rare trait among interval libraries—, *gaol* was no longer just another interval library...

[7]See the page at http://www.gnu.org/licenses/lgpl.html for what that means and entails.

greater than its right bound is also considered empty.

Switching the rounding direction incurs a high cost because it disrupts the stream-lining of instructions by requiring the emptying of the FPU pipelines. To counter this, *gaol* relies almost entirely on upward rounding only, achieving downward rounding when necessary by using simple floating-point properties (e.g., $\mathrm{fl}_\triangledown(x \times y) = -\mathrm{fl}_\triangle((-x) \times y)$). This approach minimizes rounding switches without eliminating them entirely, as some floating-point functions (most notably trigonometric, hyperbolic and other transcendental functions) often require a particular rounding direction to compute their value with the proper rounding. For better readability, however, we use both downward and upward rounding in the following algorithms.

The IEEE 754 standard requires addition, subtraction, multiplication, division, and square root to be *correctly rounded*, which means that the value computed should be the closest floating-point number less or greater (depending on the current rounding direction) than the true result. There is no such requirement for the other functions. The *gaol* library is insulated from the actual floating-point library idiosyncrasies by *proxy functions* with the expected behavior (e.g., `cos_dn()` and `cos_up()` are *gaol* functions that call the actual `cos()` function and ensure that the result returned is rounded downward or upward). As expected, the implementation of the proxies depends on the floating-point library used. The current version of *gaol* uses the IBM Accurate Portable Mathematical Library [Ziv et al. 2001]. The accuracy of the proxy functions is tied to the one of the underlying floating-point library. They may, therefore, return values that are not the floating-point numbers closest to the true results in the current rounding direction. As a consequence, the algorithms presented below return, in the general case, an enclosing interval of the closed convex hull of the inverse operators.

4.1 Inverse Multiplication

The inverse multiplication div_rel is used in the processing of the relation $x \times y = z$ to determine new domains for x and y:

$$\mathrm{div\_rel}(\mathbf{I_z}, \mathbf{I_y}, \mathbf{I_x}) = \mathrm{cch}\left(\{x \in \mathbf{I_x} \mid \exists y \in \mathbf{I_y} \exists z \in \mathbf{I_z} : xy = z\}\right).$$

As said previously, the algorithm to compute it has already been published by Hickey and others [Hickey et al. 2001]. For the sake of completeness, we recapitulate in Table I the various possible cases and refer the reader to their paper for more information.

A fine-grained analysis leads to 36 cases depending on the position of the dividend and of the divisor on the real line. In addition, if at least one of the operands is empty, the result is the empty interval.

It is possible to take advantage of SIMD instructions on modern CPUs to vectorize the tests and to perform in parallel the various divisions needed. However, depending on the authors, the expected gain does not seem worth it [Wolff von Gudenberg 2002; Malins et al. 2006].[8]

---

[8]Some extra work in the field appears in order, though, as some authors [Lambov 2006] report large gains with some instruction sets (in particular, *Streaming SIMD Extensions 2* by Intel in the paper cited).

Table I.   Inverse multiplication div_rel($\mathbf{I_z}, \mathbf{I_y}, \mathbf{I_x}$) for non-empty operands

$[\underline{\mathbf{I_z}}, \overline{\mathbf{I_z}}]$ / $[\underline{\mathbf{I_y}}, \overline{\mathbf{I_y}}]$						
	$\mathbf{I_x} \cap [\frac{\overline{\mathbf{I_z}}}{\underline{\mathbf{I_y}}}, \frac{\underline{\mathbf{I_z}}}{\overline{\mathbf{I_y}}}]$	$\mathbf{I_x} \cap [0, \frac{\underline{\mathbf{I_z}}}{\overline{\mathbf{I_y}}}]$	$\mathbf{I_x} \cap [0,0]$	$\mathbf{I_x} \cap [\frac{\overline{\mathbf{I_z}}}{\overline{\mathbf{I_y}}}, \frac{\underline{\mathbf{I_z}}}{\overline{\mathbf{I_y}}}]$	$\mathbf{I_x} \cap [\frac{\overline{\mathbf{I_z}}}{\overline{\mathbf{I_y}}}, 0]$	$I \cap [\frac{\overline{\mathbf{I_z}}}{\underline{\mathbf{I_y}}}, \frac{\underline{\mathbf{I_z}}}{\underline{\mathbf{I_y}}}]$
	$\mathbf{I_x} \cap [\frac{\overline{\mathbf{I_z}}}{\underline{\mathbf{I_y}}}, +\infty]$	$\mathbf{I_x}$	$\mathbf{I_x}$	$\mathbf{I_x}$	$\mathbf{I_x}$	$I \cap [-\infty, \frac{\underline{\mathbf{I_z}}}{\underline{\mathbf{I_y}}}]$
	$\varnothing$	$\mathbf{I_x}$	$\mathbf{I_x}$	$\mathbf{I_x}$	$\mathbf{I_x}$	$\varnothing$
	$\mathrm{cch}\left((\mathbf{I_x} \cap [-\infty, \frac{\overline{\mathbf{I_z}}}{\overline{\mathbf{I_y}}}]) \cup (\mathbf{I_x} \cap [\frac{\overline{\mathbf{I_z}}}{\underline{\mathbf{I_y}}}, +\infty])\right)$	$\mathbf{I_x}$	$\mathbf{I_x}$	$\mathbf{I_x}$	$\mathbf{I_x}$	$\mathrm{cch}\left((\mathbf{I_x} \cap [-\infty, \frac{\underline{\mathbf{I_z}}}{\underline{\mathbf{I_y}}}]) \cup (\mathbf{I_x} \cap [\frac{\underline{\mathbf{I_z}}}{\overline{\mathbf{I_y}}}, +\infty])\right)$
	$\mathbf{I_x} \cap [-\infty, \frac{\underline{\mathbf{I_z}}}{\overline{\mathbf{I_y}}}]$	$\mathbf{I_x}$	$\mathbf{I_x}$	$\mathbf{I_x}$	$\mathbf{I_x}$	$\mathbf{I_x} \cap [\frac{\underline{\mathbf{I_z}}}{\overline{\mathbf{I_y}}}, +\infty]$
	$\mathbf{I_x} \cap [\frac{\underline{\mathbf{I_z}}}{\underline{\mathbf{I_y}}}, \frac{\overline{\mathbf{I_z}}}{\overline{\mathbf{I_y}}}]$	$\mathbf{I_x} \cap [\frac{\underline{\mathbf{I_z}}}{\underline{\mathbf{I_y}}}, 0]$	$\mathbf{I_x} \cap [0,0]$	$\mathbf{I_x} \cap [\frac{\underline{\mathbf{I_z}}}{\underline{\mathbf{I_y}}}, \frac{\overline{\mathbf{I_z}}}{\underline{\mathbf{I_y}}}]$	$I \cap [0, \frac{\overline{\mathbf{I_z}}}{\underline{\mathbf{I_y}}}]$	$\mathbf{I_x} \cap [\frac{\underline{\mathbf{I_z}}}{\overline{\mathbf{I_y}}}, \frac{\overline{\mathbf{I_z}}}{\underline{\mathbf{I_y}}}]$

4.2 Inverse Power Function

The inverse power function nth_root_rel() is needed to compute the domain of an unknown x in a relation of the form $x^n = y$. Our implementation restricts n to the set $\mathbb{N}$ of natural numbers. The precise definition is:

$$\text{nth\_root\_rel}(\mathbf{I_y}, n, \mathbf{I_x}) = \text{cch}\left(\{x \in \mathbf{I_x} \mid \exists y \in \mathbf{I_y} \colon y = x^n\}\right), \quad n \in \mathbb{N}.$$

Following Graham and others [Graham et al. 1994], the algorithm given in Table II assumes $0^0 = 1$.

The cases occurring in Function `nth_root_rel()` in Table II are justified graphically in Figures 2 and 3, which represent principal branches of the nth root for n odd and even.
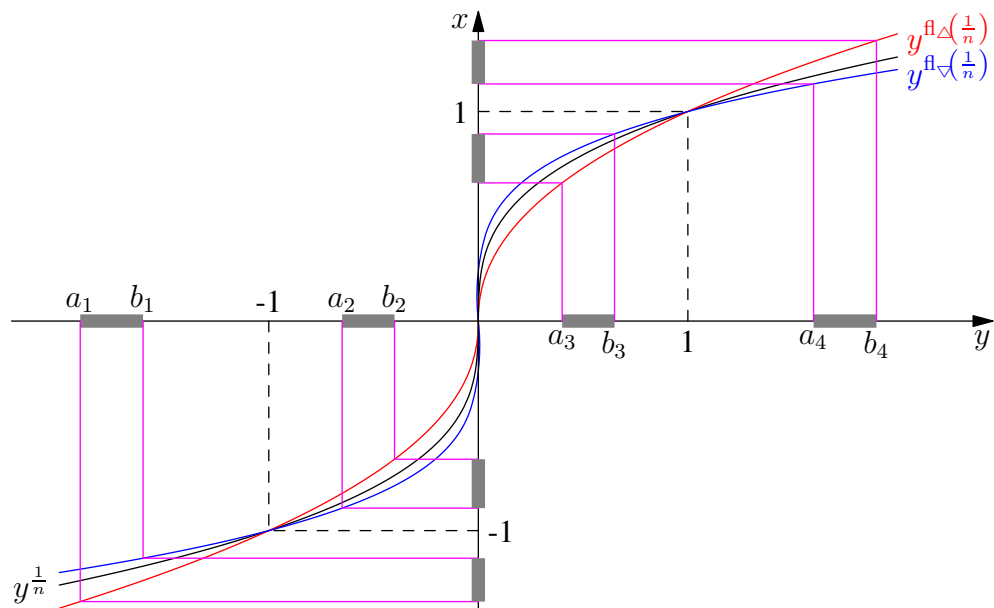

Fig. 2.   Inverse power of n for n odd

Many mathematical libraries do not handle well a call to `pow()` with negative arguments for non-integral exponents, as is the case on Lines 20, 22, 29, 31 in Table II. In that case, an easy workaround is to take the opposite twice and to round the result in the opposite direction as usual: For example, Line 20 would be replaced by "$l \leftarrow -\text{fl}_{\triangle}\left(\text{pow}(-\underline{\mathbf{I_y}}, \text{fl}_{\triangledown}(\tfrac{1}{n}))\right)$."

For n even, the intersection with $\mathbf{I_x}$ is performed separately for the negative side and the positive side (see Line 47) in order to compute the tightess domain possible.

4.3 Inverse Cosine

Given the relation $y = \cos x$, the inverse cosine is used to compute a new domain for x with respect to y. Due to the periodicity of the cosine function, the inverse

Table II.   Inverse N^{th} power

```
1  function nth_root_rel(I_y ∈ 𝕀, n ∈ ℕ, I_x ∈ 𝕀):
2  % Computes an enclosing interval for cch ({x ∈ I_x | ∃y ∈ I_y : y = x^n})
3      if n = 0:
4          if 1 ∈ I_y:
5              return I_x
6          else:
7              return ∅
8      elsif n = 1:
9          return I_x ∩ I_y
10     else:
11         if odd(n):
12             if I_x = ∅ ∨ I_y = ∅:
13                 return ∅
14             % Computing the left bound
15             if I_y ⩾ 1:
16                 l ← pow_dn(I_y, fl_▽(1/n)) % See "proxy functions" on Page 7
17             elsif I_y ⩾ 0:
18                 l ← pow_dn(I_y, fl_△(1/n))
19             elsif I_y ⩾ −1:
20                 l ← pow_dn(I_y, fl_▽(1/n))
21             else:
22                 l ← pow_dn(I_y, fl_△(1/n))
23             % Computing the right bound
24             if I_y ⩾ 1:
25                 l ← pow_up(I_y, fl_△(1/n)) % See "proxy functions" on Page 7
26             elsif I_y ⩾ 0:
27                 l ← pow_up(I_y, fl_▽(1/n))
28             elsif I_y ⩾ −1:
29                 l ← pow_up(I_y, fl_△(1/n))
30             else:
31                 l ← pow_up(I_y, fl_▽(1/n))
32             return [l, r] ∩ I_x
33         else: % even(n)
34             I_y' ← I_y ∩ [0, +∞]
35             if I_x = ∅ ∨ I_y' = ∅:
36                 return ∅
37             % Computing the left bound
38             if I_y ⩾ 1:
39                 l ← pow_dn(I_y, fl_▽(1/n))
40             else:
41                 l ← pow_dn(I_y, fl_△(1/n))
42             % Computing the right bound
43             if I_y ⩾ 1:
44                 l ← pow_up(I_y, fl_△(1/n))
45             else:
46                 l ← pow_up(I_y, fl_▽(1/n))
47             return cch (([l, r] ∩ I_x) ∪ ([−r, −l] ∩ I_x))
```
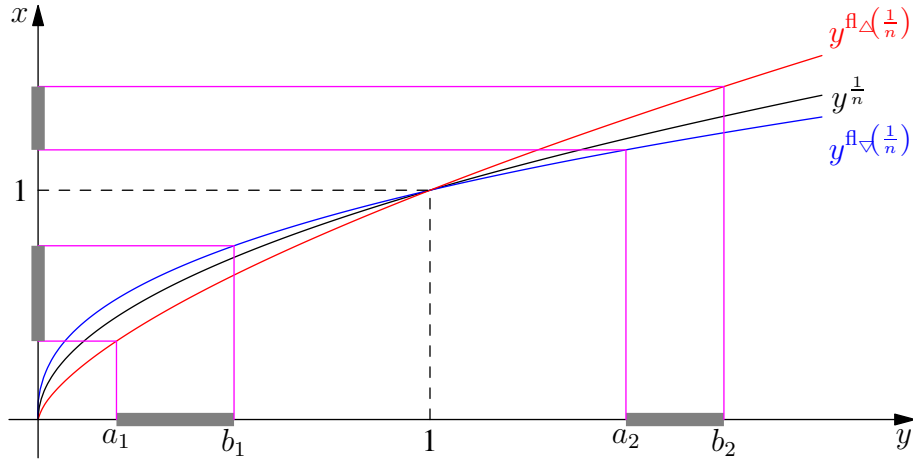
Fig. 3.   Inverse power of n for n even

cosine has to take into account the domain of x in addition to the one of y in order to compute the correct projection.

The algorithm "acos_rel" used to implement the inverse cosine is presented in Table III. Figure 4 on Page 14 graphically exemplifies the workings of the algorithm. In essence, we first compute the regular arccosine of the argument $\mathbf{I_y}$ in the domain $[0, \pi]$ (see Table IV), and we then translate by $k_l\pi$ and $k_r\pi$ the interval $\mathbf{acosI}_y$ obtained to compute new left and right bounds for $\mathbf{I_x}$ (see Table V). Integers k_l and k_r are computed by performing the first step of an argument reduction of $\mathbf{I_x}$'s bounds: For the left bound of $\mathbf{I_x}$, we compute an integer k_l such that $\underline{\mathbf{I_x}} = k_l\pi + \alpha$, with $\alpha < \pi$ (Lines 14–19 in Table III). Proposition 4.1 below ensures that, under reasonable conditions, the value k_l computed cannot be off by more than 1 from the true result.

PROPOSITION 4.1.   *Given $l \in \mathbb{F}$ a floating-point number, we have:*

$$\begin{cases} \frac{l}{\pi} - \left\lfloor \mathrm{fl}_\triangledown\!\left(\frac{l}{\mathrm{fl}_\triangledown(\pi)}\right) \right\rfloor \leqslant 1 \ \ \text{if} \ -2^{52} \leqslant l < 0 \\ \frac{l}{\pi} - \left\lfloor \mathrm{fl}_\triangledown\!\left(\frac{l}{\mathrm{fl}_\triangle(\pi)}\right) \right\rfloor \leqslant 1 \ \ \text{if} \ \ \ 0 \leqslant l \leqslant 2^{52} \end{cases}$$

where $\lfloor x \rfloor$ is the greatest integer smaller than x.

PROOF.   *We prove the result for $0 \leqslant l \leqslant 2^{52}$: Using a classical roundoff error modeling (see, e.g., Stoer and Bulirsch's book [Stoer and Bulirsch 1993]) for the* double *format in use within* gaol*, we have:*

$$\mathrm{fl}_\triangle(\pi) = \pi(1 + \varepsilon_1) \quad \text{with} \quad 0 < \varepsilon_1 \leqslant 2^{-52}.$$

The same holds for the quotient:

$$\mathrm{fl}_\triangledown\!\left(\frac{l}{\mathrm{fl}_\triangle(\pi)}\right) = \left(\frac{l}{\pi(1 + \varepsilon_1)}\right)(1 - \varepsilon_2) \quad \text{with} \quad 0 \leqslant \varepsilon_2 \leqslant 2^{-52}.$$

<div align="center">Table III.   Inverse cosine</div>

```
1  function acos_rel(I_y ∈ 𝕀, I_x ∈ 𝕀):
2     % Returns an enclosing interval for the preimage of I_y w.r.t. the cosine function and I_x:
3     % acos_rel(I_y, I_x) ⊇ cch ({x ∈ I_x | ∃y ∈ I_y : y = cos x})
4     if I_x = ∅ ∨ (I_y ∩ [−1, 1]) = ∅:
5        return ∅
6     if [−1, 1] ⊆ I_y:
7        return I_x
8     acosI_y ← acos(I_y)  % See Table IV for the definition of "acos"
9     % Checking whether the left bound is too large to perform a reliable range reduction
10    % That is, k_l would be off by more than one unit
11    if I_x ∉ [−2^52, 2^52]:
12       R_left ← I_x
13    else:
14       if I_x < 0:
15          k_l ← ⌊fl_▽( I_x / fl_▽(π) )⌋
16       elsif I_x > 0:
17          k_l ← ⌊fl_▽( I_x / fl_△(π) )⌋
18       else:
19          k_l ← 0
20       % From here, the k_l computed is at most off by 1 less than its exact value
21       R_left ← acos_k(k_l, acosI_y) ∩ I_x  % See Table V for the definition of "acos_k"
22       if R_left = ∅:
23          R_left ← acos_k(k_l + 1, acosI_y) ∩ I_x
24    % Checking whether the right bound is too large to perform a reliable range reduction
25    % That is, k_r would be off by more than one unit
26    if I_x ∉ [−2^52, 2^52]:
27       R_right ← I_x
28    else:
29       if I_x < 0:
30          k_r ← ⌊fl_△( I_x / fl_△(π) )⌋
31       elsif I_x > 0:
32          k_r ← ⌊fl_△( I_x / fl_▽(π) )⌋
33       else:
34          k_r ← 0
35       % From here, the k_r computed is at most off by 1 more than its exact value
36       if k_r = k_l:
37          R_right ← R_left
38       else:
39          R_right ← acos_k(k_r, acosI_y) ∩ I_x
40          if R_right = ∅:
41             R_right ← acos_k(k_r − 1, acosI_y) ∩ I_x
42    return [R_left, R_right]
```

We then have:

$$\frac{l}{\pi} - \text{fl}_\bigtriangledown\!\left(\frac{l}{\text{fl}_\bigtriangleup(\pi)}\right) = \frac{l}{\pi}\left(1 - \frac{1-\varepsilon_2}{1+\varepsilon_1}\right)$$
$$= \frac{l}{\pi}\frac{\varepsilon_1+\varepsilon_2}{1+\varepsilon_1}.$$

As a consequence, the quotient computed differs from the true quotient l/π by 1 or

less if and only if:

$$\frac{l}{\pi}\frac{\varepsilon_1 + \varepsilon_2}{1 + \varepsilon_1} \leqslant 1$$

that is:

$$l \leqslant \frac{1 + \varepsilon_1}{\varepsilon_1 + \varepsilon_2}\pi \qquad (3)$$

We then need to know the infimum of the right-hand side of Eq. (3) for $\varepsilon_1 \in (0, 2^{-52}]$ and $\varepsilon_2 \in [0, 2^{-52}]$. It is obtained for $\varepsilon_1 = \varepsilon_2 = 2^{-52}$, which leads to the upper bound for l :

$$l \leqslant (2^{51} + 2^{-1})\pi.$$

Since, we have:

$$2^{52} < (2^{51} + 2^{-1})\pi < 2^{53}$$

the proposition is proved for $0 \leqslant l \leqslant 2^{52}$. The proof for $-2^{52} \leqslant l < 0$ may be performed analogously and will therefore be omitted. □

Similar operations are used to compute a new bound from the right bound of $\mathbf{I_x}$.

Table IV.   Arccosine function acos: $[-1, 1] \to [0, \pi]$

```
1 function acos(I_y ∈ 𝕀):
2     I'_y ← I_y ∩ [−1, 1]
3     if I'_y = ∅:
4         return ∅
5     else:
6         return [acos_dn(Ī'_y), acos_up(I'_y)]  % See "proxy functions" on Page 7
```

Proposition 4.1 for computing the left bound—and the similar proposition for the right bound—implies that we have to compute new bounds for the inverse cosine only if the bounds of $\mathbf{I_x}$ are not too large (more precisely, they shall be smaller than 2^{52} in absolute value). Within the domain $[-2^{52}, 2^{52}]$, we know that the error made when computing k_l and k_r is less than 1 (more precisely, $0 < \lfloor l/\pi \rfloor - k_l \leqslant 1$, and $0 < k_r - \lfloor r/\pi \rfloor \leqslant 1$). As a consequence, if the intersection of $\mathbf{I_x}$ with the translated arccosine of $\mathbf{I_y}$ by $k_l\pi$ (resp. by $k_r\pi$) is empty—see Lines 21–23 and 39–41—, it suffices to translate the arccosine by one more period of π to the right for the left bound (resp., by one less period of π to the left for the right bound) to ensure correctness. Note that computing the intersection of $\mathbf{I_x}$ with the next period for the left bound, or the previous period for the right bound, would be sometimes necessary anyway, as exemplified in Figure 4: We have $\mathbf{I_x} = [a_0, b_0]$, $\mathbf{I_y} = [c, d]$, and the relation $y = \cos x$; here, k_r is equal to 2 but the intersection between $\mathbf{I_x}$ and $[a_3, b_3] = 2\mathbf{\Pi} + \text{acos}\,\mathbf{I_y}$—where $\mathbf{\Pi}$ is the smallest interval containing π—is empty. We have to consider the previous period to intersect $\mathbf{I_x}$ with $[a_2, b_2]$.

If either bound of $\mathbf{I_x}$ is outside the domain $[-2^{52}, 2^{52}]$, we do not attempt to modify it (see Lines 11–12 and Lines 26–27).

Possible values for k_l and k_r range from $\lfloor -2^{52}/\pi - 1 \rfloor$ to $\lfloor -2^{52}/\pi + 1 \rfloor$, while the typical range for signed integer numbers in two's complement is from -2^{31} to

Table V.   Arccosine function translated by $k\pi$

```
1 function acos_k(k ∈ ℤ, acosI_y ∈ 𝕀):
2   % Computes acosI_y translated to the kth period
3   % Π is the smallest floating-point interval containing π
4   if even(k):
5       return kΠ + acosI_y
6   else:
7       return (k + 1)Π − acosI_y
```
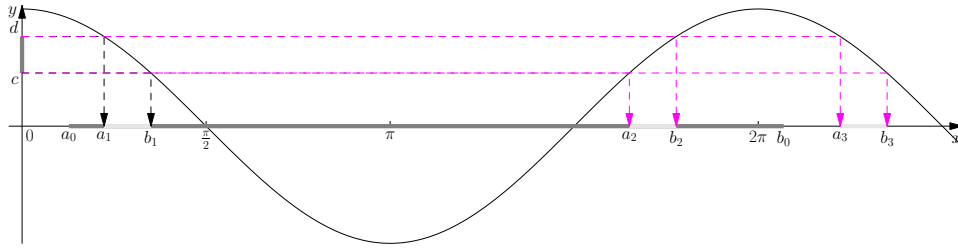

Fig. 4.   Computing the inverse cosine of $[c, d]$ with respect to $[a_0, b_0]$

$2^{31} - 1$ (for 32 bits integers). As a consequence, *gaol* uses floating-point numbers to represent k_l and k_r in the actual implementation of "acos_rel.[9]"

4.4   Inverse Sine

The inverse sine is required to compute a new domain for x with respect to y in the relation $y = \sin x$. It could be easily implemented in the same way as "acos_rel." However, it is even easier to define it in terms of "acos_rel" itself by using simple trigonometric identities, as presented in Table VI.

Table VI.   Inverse sine

```
1 function asin_rel(I_y ∈ 𝕀, I_x ∈ 𝕀):
2   % Computes a superset of cch ({x ∈ I_x | ∃y ∈ I_y : y = sin x})
3   % Π is the smallest floating-point interval containing π
4   return Π/2 + acos_rel(I_y, I_x − Π/2)
```

In all fairness, the advantages obtained from this approach ("asin_rel" may automatically benefit from future improvements to the "acos_rel" algorithm; it reduces the number of lines of code in the library, thereby decreasing the likelihood of bugs) are counterbalanced by the slight increase in the size of the results with respect to a direct implementation, which is introduced by the outward rounding occurring in the operations $\mathbf{I_x} - \mathbf{\Pi}/2$ and the addition of $\mathbf{\Pi}/2$ to the result of "acos_rel."

---

[9]Note that in the domain $[-2^{52}, 2^{52}]$, all integers have a `double` floating-point representation.

4.5 Inverse Tangent

As for the inverse cosine and the inverse sine, the inverse tangent is used to compute a new domain for x from the domain of y with the relation $y = \tan x$. The algorithm to implement "atan_rel" presented in Table VII goes along the same lines as the one for "acos_rel": we first compute the periods k_l and k_r in which fall the left bound and the right bound of $\mathbf{I_x}$ (Lines 8–13 and 21–26). We then compute the intersection of $\mathbf{I_x}$ with atan $\mathbf{I_y}$ translated by the relevant periods (Lines 14–16 and 30–32).

Table VII.   Inverse tangent

1 **function atan_rel**($\mathbf{I_y} \in \mathbb{I}, \mathbf{I_x} \in \mathbb{I}$)
2 *% Computes a superset of* cch $(\{x \in \mathbf{I_x} \mid \exists y \in \mathbf{I_y} : y = \tan x\})$
3   **if** $\mathbf{I_x} = \varnothing \vee \mathbf{I_y} = \varnothing$:
4   **return** $\varnothing$
5   **if** $\underline{\mathbf{I_x}} \notin [-2^{51}, 2^{51}]$:
6       $\mathbf{R}_{\text{left}} \leftarrow \mathbf{I_x}$
7   **else**:
8   **if** $\underline{\mathbf{I_x}} < 0$:
9         $k_l = \left\lfloor \left(\left\lfloor \text{fl}_{\triangledown}\left(\frac{\underline{\mathbf{I_x}}}{\text{fl}_{\triangleleft}(\pi \div 2)} \right) \right\rfloor + 1 \right) \div 2 \right\rfloor$
10   **elsif** $\underline{\mathbf{I_x}} > 0$:
11         $k_l = \left\lfloor \left(\left\lfloor \text{fl}_{\triangledown}\left(\frac{\underline{\mathbf{I_x}}}{\text{fl}_{\triangle}(\pi \div 2)} \right) \right\rfloor + 1 \right) \div 2 \right\rfloor$
12   **else**:
13         $k_l \leftarrow 0$
14       $\mathbf{R}_{\text{left}} \leftarrow (\textbf{atan}(\mathbf{I_y}) + k_l \mathbf{\Pi}) \cap \mathbf{I_x}$ *% See Table VIII for the definition of "atan"*
15   **if** $\mathbf{R}_{\text{left}} = \varnothing$:
16         $\mathbf{R}_{\text{left}} \leftarrow (\textbf{atan}(\mathbf{I_y}) + (k_l + 1)\mathbf{\Pi}) \cap \mathbf{I_x}$
17
18   **if** $\overline{\mathbf{I_x}} \notin [-2^{51}, 2^{51}]$:
19       $\mathbf{R}_{\text{right}} \leftarrow \mathbf{I_x}$
20   **else**:
21   **if** $\overline{\mathbf{I_x}} < 0$:
22         $k_r = \left\lfloor \left(\left\lfloor \text{fl}_{\triangle}\left(\frac{\overline{\mathbf{I_x}}}{\text{fl}_{\triangle}(\pi \div 2)} \right) \right\rfloor + 1 \right) \div 2 \right\rfloor$
23   **elsif** $\overline{\mathbf{I_x}} > 0$:
24         $k_l = \left\lfloor \left(\left\lfloor \text{fl}_{\triangle}\left(\frac{\overline{\mathbf{I_x}}}{\text{fl}_{\triangleleft}(\pi \div 2)} \right) \right\rfloor + 1 \right) \div 2 \right\rfloor$
25   **else**:
26         $k_r \leftarrow 0$
27   **if** $k_r = k_l$:
28         $\mathbf{R}_{\text{right}} \leftarrow \mathbf{R}_{\text{left}}$
29   **else**:
30         $\mathbf{R}_{\text{right}} \leftarrow (\textbf{atan}(\mathbf{I_y}) + k_r \mathbf{\Pi}) \cap \mathbf{I_x}$
31   **if** $\mathbf{R}_{\text{right}} = \varnothing$:
32           $\mathbf{R}_{\text{right}} \leftarrow (\textbf{atan}(\mathbf{I_y}) + (k_r - 1)\mathbf{\Pi}) \cap \mathbf{I_x}$
33   **return** $[\underline{\mathbf{R}_{\text{left}}}, \overline{\mathbf{R}_{\text{right}}}]$

The periods are counted as follows: Period 0 is from $-\pi/2$ to $\pi/2$, Period 1 is from $\pi/2$ to $3\pi/2$, Period -1 is from $-3\pi/2$ to $-\pi/2$, and so on. This is such that the tangent function be monotonic on each separate period.

As for "acos_rel," we use the following proposition to ensure that the computation of k_l is correctly performed (an analogous result holds for k_r):

PROPOSITION 4.2. *Given $l \in \mathbb{F}$ a floating-point number, we have:*

$$\begin{cases} \left\lfloor \left(\left\lfloor \frac{l}{\pi \div 2} \right\rfloor + 1 \right) \div 2 \right\rfloor - \left\lfloor \left(\left\lfloor \mathrm{fl}_{\bigtriangledown}\!\left(\frac{l}{\mathrm{fl}_{\bigtriangledown}(\pi \div 2)} \right) \right\rfloor + 1 \right) \div 2 \right\rfloor \leqslant 1 & \text{if } -2^{51} \leqslant l < 0 \\ \left\lfloor \left(\left\lfloor \frac{l}{\pi \div 2} \right\rfloor + 1 \right) \div 2 \right\rfloor - \left\lfloor \left(\left\lfloor \mathrm{fl}_{\bigtriangledown}\!\left(\frac{l}{\mathrm{fl}_{\bigtriangleup}(\pi \div 2)} \right) \right\rfloor + 1 \right) \div 2 \right\rfloor \leqslant 1 & \text{if } 0 \leqslant l \leqslant 2^{51} \end{cases}$$

PROOF. *We prove the result for $0 \leqslant l \leqslant 2^{51}$ (the case $-2^{51} \leqslant l < 0$ is proved in the same way, and is therefore omitted): First, note that the division by 2 does not introduce any rounding error if we use floating-point numbers conforming to the IEEE 754 standard. Consequently, we have $\mathrm{fl}_{\bigtriangleup}(\pi \div 2) = \mathrm{fl}_{\bigtriangleup}(\pi) \div 2$. We then only have to prove:*

$$\frac{l}{\pi \div 2} - \mathrm{fl}_{\bigtriangledown}\!\left(\frac{l}{\mathrm{fl}_{\bigtriangleup}(\pi) \div 2} \right) \leqslant 1 \quad \text{if} \quad 0 \leqslant l \leqslant 2^{51}.$$

Using the same modeling for rounding errors as in Proposition 4.1, we get:

$$\frac{l}{\pi \div 2} - \mathrm{fl}_{\bigtriangledown}\!\left(\frac{l}{\mathrm{fl}_{\bigtriangleup}(\pi) \div 2} \right) = \frac{l}{\pi \div 2} \left(1 - \frac{1 - \varepsilon_2}{1 + \varepsilon_1} \right)$$

with $0 < \varepsilon_1 \leqslant 2^{-52}$ and $0 \leqslant \varepsilon_2 \leqslant 2^{-52}$. Consequently:

$$\frac{l}{\pi \div 2} \left(1 - \frac{1 - \varepsilon_2}{1 + \varepsilon_1} \right) \leqslant 1 \iff l \leqslant \frac{1 + \varepsilon_1}{\varepsilon_1 + \varepsilon_2} \frac{\pi}{2}.$$

With the known bounds on ε_1 and ε_2, it comes that:

$$l \leqslant (2^{50} + 2^{-2})\pi$$

is a sufficient condition for:

$$\frac{l}{\pi \div 2} - \mathrm{fl}_{\bigtriangledown}\!\left(\frac{l}{\mathrm{fl}_{\bigtriangleup}(\pi) \div 2} \right) \leqslant 1.$$

Since $2^{51} < (2^{50} + 2^{-2})\pi < 2^{52}$, the proof is complete. $\square$

From Proposition 4.2 and the analogous proposition for the right bound, we may consider that the values computed for k_l and k_r will be off by 1 or less (by default for k_l and in excess for k_r) if the bounds of $\mathbf{I_x}$ lie in the domain $[-2^{51}, 2^{51}]$. Outside of this domain, the computation of k_l and k_r is not safe, and we therefore leave the corresponding bound of $\mathbf{I_x}$ unchanged if such situation arises.

The "atan" function used in Table VII is presented in Table VIII. It uses a floating-point arctangent function whose range is—as in most mathematical libraries—$[-\pi/2, \pi/2]$.

Table VIII.   Arctangent function

```
1 function atan(I_y ∈ 𝕀)
2 % Computes a superset of cch ({x ∈ [−π/2, π/2] | ∃y ∈ I_y : x = atan y})
3    return [atan_dn(I_y), atan_up(I_y)] % See "proxy functions" on Page 7
```

4.6   Inverse Hyperbolic Cosine

The inverse hyperbolic cosine is used to compute a new domain for x with respect to y in the relation $y = \cosh x$. Contrarily to the inverse hyperbolic sine and the inverse hyperbolic tangent, the inverse of the hyperbolic cosine requires a special algorithm, which is given in Table IX. It is quite simple and only has to take into account the evenness of the hyperbolic cosine function.

Table IX.   Inverse hyperbolic cosine

```
1  function acosh_rel(I_y ∈ I, I_x ∈ I):
2  % Computes a superset of cch ({x ∈ I_x | ∃y ∈ I_y : y = cosh x})
3     if I_x = ∅ ∨ I_y = ∅:
4        return ∅
5     acoshI_y ← acosh(I_y) % See Table X for the definition of "acosh"
6     if I_x ⩾ 0:
7        return I_x ∩ acoshI_y
8     elsif I_x ⩽ 0:
9        return I_x ∩ −acoshI_y
10    else:
11       return cch ((I_x ∩ acoshI_y) ∪ (I_x ∩ −acoshI_y))
```

The "acosh_rel" function uses the interval extension "acosh" of the hyperbolic arccosine presented in Table X. Since no floating-point "acosh" function is implemented in the IBM Accurate Portable Mathematical Library, we make do with the identity $\operatorname{acosh} x = \log(x + \sqrt{x^2 - 1})$.

Table X.   Hyperbolic arccosine

```
1  function acosh(I_y ∈ I):
2  % Computes a superset of cch ({x ∈ ℝ^+ | ∃y ∈ I_y : x = acosh y})
3     I'_y ← I_y ∩ [1, +∞]
4     if I'_y = ∅:
5        return ∅
6     else:
7        return log(I'_y + √(I'_y² − 1)) % See Table XI for the definition of log
```

Table XI.   Interval extension of the logarithm

```
1  function log(I_y ∈ I):
2  % Computes a superset of cch ({x ∈ ℝ^+ | ∃y ∈ I_y : x = log y})
3     I'_y ← I_y ∩ [0, +∞]
4     if I'_y = ∅:
5        return ∅
6     else:
7        return [log_dn(I'_y), log_up(I'_y)] % See "proxy functions" on Page 7
```

5.   EVALUATION OF THE ALGORITHMS

In order to assess the effectiveness of the algorithms presented, we tested them as follows: For each one of the relations $xy = z$, $y = \cos x$, $y = \sin x$, $y = \tan x$, $y = \cosh x$, and $y = x^n$, we randomly generated $10,000,000$ pairs of non-empty domains (or triplets of domains, for the relation $xy = z$), and we applied the corresponding direct operator and inverse operator on them in order to determine new domains for the variables, as presented in Eq. (1) on Page 3.

The results are presented in Table XII. We considered the time spent to perform the $10,000,000$ narrowings for two versions of *gaol*: The "RP" (*Rounding Preserved*) column corresponds to the version in which the rounding direction is set to "upward" once at the beginning of each call to any interval operation, and set back to its previous value at the end of the operation. The "RNP" (*Rounding Not Preserved*) column corresponds to the version in which the rounding direction is set to "upward" at the very beginning of the program, and is almost never modified afterwards.[10] Note that both versions ensure correctness of the computation; either one may be selected when configuring *gaol* prior to compiling it (see *gaol*'s manual). The "RP" version should be used when embedding *gaol* in a program that intends to manipulate the rounding direction itself, or that assumes some rounding direction different from the "upward" one; otherwise, the "RNP" version should be used, as it is much more efficient.

As a comparison, we also present the results obtained with *smath*. There is no entry for the relation "$y = \text{acosh}\,x$," as the hyperbolic cosine and its inverse are not supported by that library. There is no entry for the relation "$y = x^n$" either, as we could not reliably test the power function for odd values of n. Therefore, the last row of the table presents results for even values of n only, in order to ensure some point of comparison for such an important relation.

Besides speed, we also tested the amount of reduction performed by the operators (Column "Reduction"). For each relation, we give two percentages: the one at the top represents the average percentage of reduction on all variables and for all $10,000,000$ tests, excepting the cases where at least one domain was narrowed down to the empty set; the one at the bottom gives the number of reductions that led to failure (at least one variable in the relation was narrowed down to the empty set) as a percentage.

All the experiments were conducted on an Intel Pentium IV at 3.8 GHz with 2 GB of RAM, running Linux *Kubuntu Gutsy Gibbon*. This computer has a *Standard Unit Time* equal to $50.4\,s$ (This is defined as the time required for performing 10^8 evaluations of the function Shekel 5 [Dixon and Szegö 1978]). All tests were performed with *gaol* 3.1.0 and *smath* 2005-12-19. The $10,000,000$ non-empty intervals were generated by using the `drand48()` and `rand()` standard functions to create `double` bounds from a mantissa (obtained with `drand48()`), an exponent and a sign (obtained from `rand()`). The pseudo-random generators were initialized with the same fixed seed for both *gaol* and *smath*. For the purpose of testing the

---

[10]This is true only for the functions for which *correct rounding* is mandated by the standard. For all other functions (that is, mainly, cos, sin, tan, pow, cosh and their inverses), we have to switch the rounding to the nearest for each call in order to compute the correct result with the IBM Accurate Portable Mathematical Library.

Table XII.   Performances on $10,000,000$ tests per relation

Relation	Time (in s.)			Reduction		
	Gaol		smath	Gaol	smath	
	RP	RNP				
$xy = z$	29	3.7	5.4	55.5%	56.2%	%age reduc.
				19.6%	18.4%	%age fail.
$y = \cos x$	74.2	16	160.3	56.8%	56.8%	%age reduc.
				25.1%	25.2%	%age fail.
$y = \sin x$	103	37.6	177.2	56.8%	53%	%age reduc.
				25.1%	19.2%	%age fail.
$y = \tan x$	118.8	37.6	362.4	82.5%	53%	%age reduc.
				9.5%	19.2%	%age fail.
$y = \cosh x$	47.7	11	NA	30.5%	NA	%age reduc.
				62.8%	NA	%age fail.
$y = x^n$	56.1	12.6	NA	23.6%	NA	%age reduc.
				39%	NA	%age fail.
$y = x^m,$ $m \equiv 0 \pmod 2$	49.2	12.3	502.5	23%	23.1%	%age reduc.
				40.7%	40.7%	%age fail.

Time on an Intel Pentium IV at 3.8 GHz (rounded to nearest 10th sec.)

power function and its inverse, the values of the exponents n and m were chosen randomly between 0 and 99 inclusive.

As seen in Table XII, the "RNP" version of *gaol* is much faster than the "RP" version, and it outperforms *smath* on all operators. The "RP" version is still better than *smath* on almost all operators, except for the multiplication and its inverse. As for the quality of reduction, *gaol* and *smath* appear equivalent, except for the "$y = \tan x$" relation: *gaol* achieves an impressive 82.5% of reduction, versus 53% for *smath*. However these figures do not tell the whole story: indeed, *smath* is able to discover failure in 19.2% of the cases, versus 9.5% only for *gaol*. As a consequence, the results of *smath* and *gaol* for the tangent and the inverse tangent are not directly comparable. Lastly, we observe that the treatment of the cosine relation is significantly faster than the one of the sine relation, which sheds a new light on our choice of reusing code for the implementation of the inverse sine (and for the sine as well).

6.   CONCLUSION

The results given in Section 5 should be taken with a grain of salt (shouldn't they all?), as they are by no means indicative of the actual performances in real conditions of the libraries tested. They should only be considered as a benchmark for future works in the domain. However, from the return on experience by those who already used *gaol* for their project, we still are confident of its good standing among the interval libraries available, be it for the quality of its "functional" operators or

its "relational" ones.

The *gaol* library still has weaknesses that should be addressed. In particular, it is not possible to use bounds of varying precision as in *MPFI* [Revol and Rouillier 2005] and *Boost Interval* [Melquiond et al. 2007]. Even for fixed precision, relying on a library like *MPFR* [Hanrot et al. 2007], which implements correctly rounded floating-point functions, would allow us to compute the exact closed convex hull of all inverse operators supported.

REFERENCES

ABERTH, O. AND SCHAEFER, M. J. 1992. Precise computation using range arithmetic, via C++. *ACM Transactions on Mathematical Software 18,* 4, 481–491.

BENHAMOU, F. 1995. Interval constraint logic programming. In *Constraint programming: basics and trends: 1994 Châtillon Spring School, Châtillon-sur-Seine, France, May 16–20, 1994*, A. Podelski, Ed. Lecture notes in computer science, vol. 910. Springer-Verlag, 1–21.

BENHAMOU, F. 2001. Interval constraints, interval propagation. In *Encyclopedia of Optimization*, P. M. Pardalos and C. A. A. Floudas, Eds. Vol. 3. Kluwer Academic Publishers, Dordrecht, The Netherlands, 45–48.

BENHAMOU, F., GOUALARD, F., GRANVILLIERS, L., AND PUGET, J.-F. 1999. Revising hull and box consistency. In *Proceedings of the sixteenth International Conference on Logic Programming (ICLP'99)*. The MIT Press, Las Cruces, USA, 230–244.

BENHAMOU, F. AND TOURAÏVANE. 1995. Prolog IV: langage et algorithmes. In *JFPL'95: IVèmes Journées Francophones de Programmation en Logique*. Teknea, Dijon, France, 51–65.

BRÖNNIMANN, H., MELQUIOND, G., AND PION, S. 2006a. The design of the Boost interval arithmetic library. *Theor. Comput. Sci 351*, 111–118.

BRÖNNIMANN, H., MELQUIOND, G., AND PION, S. 2006b. A proposal to add interval arithmetic to the C++ standard library. Technical Report N2137=06-0207, rev. 2, CIS Polytechnic Univ., École Normale Suprieure de Lyon, and INRIA.

C99. 1999. ISO/IEC 9899 - programming languages - C. Draft of the standard available at `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf`.

CEBERIO, M. AND GRANVILLIERS, L. 2000. Solving nonlinear systems by constraint inversion and interval arithmetic. In *Proceedings of International Conference on Artificial Intelligence and Symbolic Computation*, J. A. Campbell and E. Roanes-Lozano, Eds. Lecture Notes in Artificial Intelligence, vol. 1930. Springer-Verlag, Madrid, Spain, 127–141.

CHEADLE, A. M., EL-SAKKOUT, H., SCHIMPF, J., SHEN, K., AND NOVELLO, S. 2007. ECLIPSe. Available at `http://sourceforge.net/projects/eclipse-clp`.

CHRISTIE, M., GRANVILLIERS, L., AND SORIN, V. 2005. Elisa: an open C++ library for constraint programming and constraint solving techniques. Available at `http://sourceforge.net/projects/elisa/`.

CLEARY, J. G. 1987. Logical arithmetic. *Future Computing Systems 2,* 2, 125–149.

CLOCKSIN, W. F. AND MELLISH, C. S. 1981. *Programming in Prolog.* Springer-Verlag, New York.

DAVIS, E. 1987. Constraint propagation with interval labels. *Artificial Intelligence 32*, 281–331.

DAWES, B., ABRAHAMS, D., AND RIVERA, R. 1998–2007. Boost C++ libraries. Web site at `http://www.boost.org/`.

DE KONINCK, L., SCHRIJVERS, T., AND DEMOEN, B. 2006. Inclp($\mathbb{R}$): Interval-based nonlinear constraint logic programming over the reals. In *Proceedings of the 20th Workshop on Logic Programming*, M. Fink, H. Tompits, and S. Woltran, Eds. INFSYS Research Report, vol. 1843-06-02. Technische Universität Wien, 91–100.

DECHTER, R. 2003. *Constraint Processing*, 1st ed. Morgan Kaufmann. With contributions by David Cohen, Peter Jeavons, and Francesca Rossi.

DIXON, L. C. W. AND SZEGÖ, G. P. 1978. The global optimization problem: an introduction. In *Towards Global Optimization 2*, L. C. W. Dixon and G. P. Szegö, Eds. North-Holland, 1–15.

FEYDY, T. AND STUCKEY, P. J. 2007. Propagating dense systems of integer linear equations. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*. ACM, New York, NY, USA, 306–310.

FRÄNZLE, M., HERDE, C., TEIGE, T., RATSCHAN, S., AND SCHUBERT, T. 2007. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation 1*, 209–236.

GOUALARD, F. 2000. Towards good C++ interval libraries: Tricks and traits. Accepted for publication at the 4th Asian Symposium on Computer Mathematics. Chiang Mai, Thailand, December 17–21, 2000.

GOUALARD, F. 2001–2007. GAOL: not Just Another Interval Library. Web site at: `http://sourceforge.net/projects/gaol/`.

GRAHAM, R. L., KNUTH, D. E., AND PATASHNIK, O. 1994. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

GRANVILLIERS, L. 2004. Realpaver. Available at `http://www.sciences.univ-nantes.fr/info/perso/permanents/granvil/realpaver/`.

GRANVILLIERS, L. AND BENHAMOU, F. 2001. Progress in the solving of a circuit design problem. *Journal of Global Optimization 20*, 155–168.

GRANVILLIERS, L. AND BENHAMOU, F. 2006. Algorithm 852: Realpaver: an interval solver using constraint satisfaction techniques. *ACM Trans. Math. Softw. 32,* 1, 138–156.

HANROT, G., LEFÈVRE, V., PÉLISSIER, P., THÉVENY, P., AND ZIMMERMANN, P. 2000–2007. The MPFR library. C library available at `http://www.mpfr.org/`.

HANSEN, E. R. AND SENGUPTA, S. 1981. Bounding solutions of systems of equations using interval analysis. *BIT 21*, 203–211.

HICKEY, T. 2000–2005. smath: a library of C routines for interval arithmetic and constraint narrowing. Software library available at `http://interval.sourceforge.net/interval/prolog/clip/clip/smath/README.html`.

HICKEY, T. AND JU, Q. 1996. Fast, sound, and precise narrowing of the exponential function. Technical report, Michtom School of Computer Science. Brandeis Univ. Mar.

HICKEY, T. AND JU, Q. 1997. Efficient implementation of interval arithmetic narrowing using IEEE arithmetic. Technical report, Michtom School of Computer Science. Brandeis Univ. Mar.

HICKEY, T. J., JU, Q., AND VAN EMDEN, M. H. 2001. Interval arithmetic: from principles to implementation. *Journal of the ACM 48,* 5 (Sept.), 1038–1068.

HYVÖNEN, E. 1989. Constraint reasoning based on interval arithmetic. In *Proceedings of IJCAI'89*. 1193–1198.

HYVÖNEN, E. 1995. LIA InC++: A local interval arithmetic library for discontinuous intervals. Research report, VTT Information Technology.

IEEE. 1985. IEEE standard for binary floating-point arithmetic. Tech. Rep. IEEE Std 754-1985, Institute of Electrical and Electronics Engineers. Reaffirmed 1990.

ILOG INC. 2007. Ilog CP: Ilog Solver. See `http://www.ilog.com/products/cp/`.

KEARFOTT, R. B., NAKAO, M. T., NEUMAIER, A., RUMP, S. M., SHARY, S. P., AND VAN HENTENRYCK, P. 2005. Standardized notation in interval analysis. In *Proc. XIII Baikal International School-seminar "Optimization methods and their applications"*. Vol. 4 "Interval analysis". Irkutsk: Institute of Energy Systems SB RAS, 106–113.

KNÜPPEL, O. 1994. PROFIL/BIAS—a fast interval library. *Computing 53*, 277–287.

KOWALSKI, R. 1974. Predicate logic as programming language. In *IFIP Congress*. 569–574. Reprinted in Computers for Artificial Intelligence Applications, (eds. Wah, B. and Li, G.-J.), IEEE Computer Society Press, Los Angeles, 1986, pp. 68–73.

LAMBOV, B. 2006. Interval arithmetic using SSE-2. In *Reliable Implementation of Real Number Algorithms: Theory and Practice*, P. Hertling, C. M. Hoffmann, W. Luther, and

N. Revol, Eds. Number 06021 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany.

LERCH, M., TISCHLER, G., GUDENBERG, J. W. V., HOFSCHUSTER, W., AND KRÄMER, W. 2006. Filib++, a fast interval library supporting containment computations. *ACM Trans. Math. Softw 32*, 299–324.

LHOMME, O. 1993. Consistency techniques for numeric CSPs. In *Proceedings of the 13th IJCAI*, R. Bajcsy, Ed. IEEE Computer Society Press, Chambéry, France, 232–238.

MACKWORTH, A. K. 1977. Consistency in networks of relations. *Artificial Intelligence 1,* 8, 99–118.

MALINS, E., SZULARZ, M., AND SCOTNEY, B. 2006. Vectorised/semi-parallel interval multiplica- tion. In *Procs. of 12th GAMM - IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics (SCAN 2006)*. Abstract.

MEJIAS, B., COLLET, R., DUCHIER, D., GROLAUX, D., KONVIČKA, F., SPIESSENS, F., GUTIER- REZ, G., GLYNN, K., ANDERS, T., AND JARADIN, Y. 2006. The Mozart programming system. Available at `http://www.mozart-oz.org/`.

MELQUIOND, G., PION, S., BRÖNNIMANN, H., YU, J., AND REVOL, N. 2007. Boost Interval: In- terval arithmetic for the C++ standard. C++ library available at `http://gforge.inria.fr/projects/std-interval/`.

MICROSYSTEMS, S. 2002. C++ interval arithmetic programming reference. Technical Report No. 816-2465-10, Rev. A, SUN Microsystems.

MOORE, R. E. 1966. *Interval Analysis*. Prentice-Hall, Englewood Cliffs, N. J.

OLDER, W. J. 1989. Interval arithmetic specification. Research Report 89032, Computer Research Laboratory, Bell-Northern Research. July.

OLDER, W. J. AND VELLINO, A. 1990. Extending prolog with constraint arithmetic on real intervals. In *Procs. of the IEEE Canadian Conference on Electrical and Computer Engineering*. IEEE Computer Society Press.

PUGET, J.-F. 1994. A C++ implementation of CLP. In *Proceedings of the Singapore Conference on Intelligent Systems (SPICIS'94)*. Singapore.

PYTHON SOFTWARE FOUNDATION. 2007. The Python programming language. Web site at `http://www.python.org/`.

REVOL, N. AND ROUILLIER, F. 2005. Motivations for an arbitrary precision interval arithmetic and the MPFI library. *Reliable Computing 11,* 4 (Aug.), 275–290.

RUMP, S. M. 1999. Intlab - interval laboratory. In *Developments in Reliable Computing*, T. Csendes, Ed. Kluwer Academic Publishers, 77–104.

STOER, J. AND BULIRSCH, R. 1993. *Introduction to Numerical Analysis*, second ed. Number 12 in Texts in Applied Mathematics. Springer, Heidelberg.

SUNAGA, T. 1958. Theory of an interval algebra and its application to numerical analysis. In *Research Association of Applied Geometry Memoirs of the unifying study of basic problems in engineering and physical sciences by means of geometry*, K. Kondo, Ed. Vol. 2. Gakujutsu Bunken Fukyu-Kai, Japan, Chapter Miscellaneous Subjects II, 547–564.

SUTHERLAND, I. E. 1963. Sketchpad: A man-machine graphical communication system. In *AFIPS Conference Proceedings*. 323–328. Reprinted as Technical Report UCAM-CL-TR-574, Cam- bridge University.

WARMUS, M. 1956. Calculus of approximations. *Bull. Acad. Polon. Sci., Cl. III 4,* 5, 253–259.

WIETHOFF, A. 1996. C-XSC: a C++ class library for extended scientific computing. Research Report 2/1996, Institut für Angewandte Mathematik. Universität Karlsruhe.

WOLFF VON GUDENBERG, J. 2002. Interval arithmetic on multimedia architectures. *Reliable Computing 8*, 307–312.

YOHE, J. M. 1979. Software for interval arithmetic: A reasonably portable package. *ACM Trans- actions on Mathematical Software 5,* 1 (Mar.), 50–63.

ZIMMER, L. 2004. The CO2 project : Purpose, results and future challenges. Presentation at the Franco-Japanese Workshop on Constraint Programming.

Ziv, A., Olshansky, M., Henis, E., and Reitman, A. 2001. Accurate portable mathematical library (IBM APMathLib). Available at `ftp://www-126.ibm.com/pub/mathlib/mathlib12.20.2001.tar.gz`.

Zoeteweij, P. and Arbab, F. 2004. A component-based parallel constraint solver. In *Procs. of Coordination Models and Languages (COORDINATION 2004)*, R. De Nicola, G. Ferrari, and G. Meredith, Eds. Number 2949 in LNCS. Springer-Verlag, 307–322.