Why3 を用いた区間べき関数のプログラム検証

村上 椋星　薮 智仁　石井 大輔

区間計算は，数値の代わりに真解を含んだ区間値を扱うことで，数値計算を精度保証付きで行う手法である．区間計算の実装は特殊値や区間端点の組み合わせなどにより複雑化することが多い．そのため実装の検証が高信頼な数値計算のために重要である．一方で，検証ツールによる浮動小数点数サポートなどにより，数値計算プログラムの自動検証が可能になりつつある．そこで本研究では，区間計算プログラムの検証を行い，高信頼な実装を提供することを目標とする．繰り返し処理による区間べき関数（累乗）の実装を対象とし，検証ツール Why3 と SMT ソルバー Alt-Ergo を用いて検証を実施する．検証では，まず対象プログラムと満たすべき性質を Why3 の記述言語で記述し，つぎにツールで検証条件を生成・証明する．区間計算プログラムに対して有用なループ不変条件や補助定理等を明らかにすることを目指す．

Interval computation is a rigorous numerical method that computes intervals instead of numerical values; a resulting interval encloses a true solution and also guarantees the precision of the numerical computation. Implementations of interval computation often become complicated by considering e.g. combinations of the special values and the bounds of argument intervals. Therefore, verification of implementations is important for the reliable numerical computation. On the other hand, with the progress of verification tools, e.g., a direct support of floating-point numbers, automated verification of numerical programs becomes possible. In this research, we verify programs that implements interval computation to achieve reliability. More specifically, we apply the Why3 tool and the Alt-Ergo SMT solver to verify a program that implements an interval power function. In the verification, we first specify the target program annotated with expected properties with Why3's specification language; then, we prove the verification conditions using the tools. We aim to develop a scheme that identifies loop invariants and auxiliary lemmas that are useful in the interval computation domain.

1　はじめに

区間計算（2 節）とは，数値の代わりに真解を含んだ区間値を扱うことにより，数値計算を精度保証付きで行う手法である．区間計算では数学的な実数関数について区間拡張を考える．関数の区間拡張は引数として区間を取り，計算結果も区間となる．結果の区間は，引数区間中の各実数に当該関数を適用した結果をすべて含む．区間の端点は浮動小数点数であるため，区間の端点に対して適切な方向に丸め処理を施すことで丸め誤差を含めて計算を行う．また，各数学関数について，区間幅の増大を抑えた区間拡張の方法が提案されている．

区間計算の実装は区間の端点の組み合わせなどにより複雑化することが多く，バグの混入により計算結果の信頼性が損なわれてしまう．区間計算では，各端点が 0 や無限大の値をとった場合を個別に考慮する必要があり，多数の場合分けが生じてしまう．また，ある実装が複数の引数の各端点について正しく動作するかどうかは自明でないことが多い．

本研究では，区間計算プログラムの検証を行い，高信頼な実装を提供することを目標とする．区間の整数べきを計算する関数を検証対象とし，健全性（計算結果が真解を含んだ区間となっていること）を検証す

Program Verification of Interval Power Function with Why3.

Ryosei Murakami, 福井大学工学研究科情報・メディア工学専攻, Dept. of Information Science, Fukui University.

Tomohito Yabu, 株式会社コロプラ, COLOPL, Inc.

Daisuke Ishii, 福井大学, Fukui University.

る（4 節）．

検証にはプログラム検証ツール Why3（3.1 節）を用いて行う．Why3 は，プログラムを記述するためと，仕様をアノテーションするための 2 種類の専用言語を提供する．記述した仕様付きプログラムから検証条件を自動生成し，それらをバックエンド証明器群（3.2 節）により証明し，半自動で検証を行うことができる．最近の証明器の発展により，数値計算プログラムの検証が可能となりつつある．

本研究ではまず，べき関数とその健全性について Why3 の基本的な検証プロセスを適用する（4.1–4.3 節）．しかし，いくつかの検証条件の自動証明がうまくいかない結果となってしまう．次に，対象プログラムに補助的なアノテーションを施し，自動証明器の推論機構を活用する戦略を取る（4.4 節）．プログラムの途中状態や，べき関数に関する定理のアノテーションを検討する．この戦略の結果，すべての検証条件の証明に成功し，区間の整数べき関数の検証が可能であることを確かめる（4.5 節）．

2 区間計算

区間計算は，数値の代わりに真解を含んだ区間値を扱い，高信頼に数値計算を行うための手法である．実数の集合を $\mathbb{R}$，浮動小数点数の集合を $\mathbb{F}$ とする．
以下は，区間の表記と定義である:

$$\boldsymbol{x} = [\underline{x}, \overline{x}] := \{x \in \mathbb{R} | \underline{x}, \overline{x} \in \mathbb{F}, \underline{x} \leq x \leq \overline{x}\} \quad (1)$$

ここで，$\underline{x}$ と $\overline{x}$ それぞれを下限，上限と呼ぶ．ただし，$\underline{x}, \overline{x} \neq \mathrm{NaN}$, $\underline{x} \neq +\infty$, $\overline{x} \neq -\infty$ とする．本稿では，上記に沿った区間のみを妥当（valid）な区間と考える．実数の関数 $f : \mathbb{R}^m \to \mathbb{R}^n$ について，区間拡張 $\boldsymbol{f} : \mathbb{I}^m \to \mathbb{I}^n$ を考える．このとき，$\forall x \in \boldsymbol{x}, f(x) \in \boldsymbol{f}(\boldsymbol{x})$ が成り立つ必要がある（健全性）．区間の両端は浮動小数点数のため，下限値，上限値を計算する際に丸め誤差による影響が生じる．健全性を満たすため，下限値を計算するときは下向きの丸め，上限値を計算をするときは上向きの丸めで計算する．しかし，外側に丸めて計算するため，理論値（当該関数 f の真値の集合）より広い区間になる性質がある．また，区間拡張は理論値を包む最小幅の区間を返すのが望ましい．

区間 $\boldsymbol{x}, \boldsymbol{y}$ と，四則演算 $\odot \in \{+, -, *, /\}$ について，以下のような区間拡張を考えることができる:

$$\begin{aligned} \boldsymbol{x} + \boldsymbol{y} &:= [\bigtriangledown(\underline{x} + \underline{y}), \bigtriangleup(\overline{x} + \overline{y})], \\ \boldsymbol{x} - \boldsymbol{y} &:= [\bigtriangledown(\underline{x} - \overline{y}), \bigtriangleup(\overline{x} - \underline{y})], \\ \boldsymbol{x} * \boldsymbol{y} &:= [\bigtriangledown(\min S), \bigtriangleup(\max S)], \\ \boldsymbol{x} / \boldsymbol{y} &:= \boldsymbol{x} * [\bigtriangledown(1/\overline{y}), \bigtriangleup(1/\underline{y})] \end{aligned} \quad (2)$$

ただし，$\bigtriangledown$ と $\bigtriangleup$ は下向きと上向きの丸め演算子，$\odot = /$ のとき $0 \notin \boldsymbol{y}$，$S = \{\underline{x} * \underline{y}, \underline{x} * \overline{y}, \overline{x} * \underline{y}, \overline{x} * \overline{y}\}$ である．

3 プログラム検証ツール

3.1 Why3

Why3[†1] はホーア論理に基づく半自動的なプログラミング検証を効率よく実施するための支援ソフトウェアである．検証対象のプログラムを記述するための OCaml に似た WhyML 言語と，プログラムに仕様をアノテーションするための Why3 言語を備えている．Why3 を用いることで，WhyML 言語で記述したプログラムと Why3 言語で記述した仕様から，検証条件（Verification Condition, VC）を自動生成することができる．また，生成した検証条件は，バックエンド証明器としてプラグインされた SMT ソルバー（Alt-Ergo 等）や証明支援系（Coq[†2] 等）を用い，自動的あるいは対話的に証明することができる．

3.2 SMT ソルバー Alt-Ergo

Alt-Ergo はプログラム検証に応用する際に有用な SMT ソルバーの 1 つである．実数や浮動少数点数の背景理論に基づく述語論理式の妥当性を判定することができる．SMT ソルバーは判定結果として 4 種類の判定結果を返す．(1) 述語論理式が妥当である（Valid），(2) 妥当ではない（Invalid），(3) 制限時間以内に停止しなかった（Timeout），(4) 決定不能である（Unknown）のいずれかである．トリガー機構により，付加的に与えた公理や定理を利用した柔軟な自動推論が可能である．本研究では，SMT ソルバーのなかでもおもにこの Alt-Ergo を用いることで検証

†1 http://why3.lri.fr/
†2 https://coq.inria.fr/

条件の証明を行う．

3.3 Why3 を用いたプログラム検証作業

Why3 を用いたプログラム検証では，以下の作業を実施する．

1. WhyML 言語で検証したいプログラムを記述し，記述したプログラムに対し満たすべき仕様を Why3 言語で記述する（事前/事後条件など）．
2. Why3 により，記述したアノテーション付きの検証対象プログラムから検証条件（述語論理式）を自動生成する．
3. バックエンド証明器を用いて，生成された検証条件の妥当性を判定する．すべての検証条件が妥当と判定できれば，元の仕様付きプログラムの正しさが示せたことになる．

ステップ 3 の妥当性判定がうまくいかない場合，ステップ 1 へ戻り，仕様やプログラムの修正を行ったり，仕様の一部として補助定理を追加したりする．

4 区間のべき関数プログラムの検証

本節では，区間の整数べき（累乗）を計算する関数 `pown` の実装と検証について述べる．本研究では，$\texttt{pown}(\boldsymbol{x}, y)$ が $\forall x \in \boldsymbol{x}, x^y \in \texttt{pown}(\boldsymbol{x}, y)$ であることを検証する．関数 `pown` は，区間の底 $\boldsymbol{x}$ と正の整数のべき指数 y からべき関数の区間包囲を計算する．ここで，関数 `pown` は kv ライブラリ [†3] の実装を WhyML 言語に変換したものである．

4.1 モジュールシステム

WhyML 言語による記述は，モジュール単位で管理される．また，複数のモジュールをファイルにまとめるが，まとめたものをパッケージと呼ぶことにする．各モジュールには，型，変数，関数，述語などを定義し，他のモジュール定義からインポートすることで再利用することができる．本研究のおもな使用モジュール・パッケージは以下のとおりである．

- `int` パッケージ: `int` 型の定義と，`int` 型変数に対する算術演算や簡単な関数等を定義している．
- `real` パッケージ: `real` 型の定義と，`real` 型変数に対する算術演算や簡単な関数等を定義している．
- `ieee_float.Float64` モジュール: `Float64.t` 型の定義と，$\pm\infty$ 等を定義している．`Float64.t` 型変数に対する算術演算等も定義している．
- `interval.Interval` モジュール: `t` 型を 2 つ束ねたレコード型の `interval` 型 を定義している．演算時の区間端点の丸めや，区間の妥当性判定のための述語等も定義している．
- `mul.Mul` モジュール: 区間の乗算関数 `mul` を提供している．
- `PownTheory` モジュール: べき関数プログラムの検証補助している（4.4 節参照）．

4.2 べき関数の実装

図 1 は，WhyML 言語にて記述した区間のべき関数 `pown` である．引数として区間型（`interval`）の `x` と 整数型の `y` を受け取り区間型の値を結果として返している．今回，引数 `y` は正の値のみの場合を考えている．まずはじめに，引数 `y` が 0 であるかどうかに応じて場合分けを行う．`y` が 0 であったならば区間値 $[1, 1]$ を返す（14 行目）．0 でなかった場合，15 行目以下に進む．18 ～ 26 行目では計算を行う回数を保管している `a` と引数の `x` の値に応じて場合分けを行い，それぞれの場合に応じた計算を行う．`a` が偶数かつ 区間 `x` の間に 0 が入っている場合，18 ～ 20 行目，それ以外の場合は 22 ～ 26 行目である．各場合において，区間 `x` の端点の浮動少数点数の整数べきを計算する関数 `pown_point` を呼び出す．

浮動少数点数の整数べき関数 `pown_point` を図 2 に示す．引数として，浮動小数点数型（`Float64.t`）`x` と整数型 `y` を受け取り区間型の値を結果として返す．まずはじめに，特殊な場合（`y = 0`, `x` が $\pm\infty$）を計算する．特殊な場合でなかったならば，再帰関数 `aux` により整数べきを計算する．21 ～ 22 行目にて関数 `aux` に与える引数の初期区間値を定義し，23 行目で関数 `aux` を呼び出している．`r0` は $[1, 1]$，`xp0` は $[\texttt{x}, \texttt{x}]$ である．関数 `aux` の計算例を以下に示す．10 進数 5 桁の浮動小数点数による計算とする．

†3 `http://verifiedby.me/kv/simple/index-e.html`

```
1  module Pown
2    use mach.int.Int
3    use real.RealInfix
4    use real.PowerInt
5    use ieee_float.Float64
6    use interval.Float64Ex
7    use interval.Interval
8    use mul.Mul
9    use pown_theory.PownTheory
10   use pown_point.PownPoint
11
12   let pown (x: interval) (y: int) : interval
13    (* Specification *)
14   = if y = 0 then of_float oneF
15     else begin
16       let a = y in
17       let r =
18         if a % 2 = 0 && float_in zeroF x then begin
19           let r' = hull_with_float (pown_point (mag x) a) zeroF in
20           r'
21         end else begin
22           let ri = pown_point x.inf a in
23           let rs = pown_point x.sup a in
24           let r' = hull ri rs in
25           r'
26         end
27       in
28       r
29     end
30   end
31 end
```

図 1　整数べき関数の WhyML 言語による実装

```
1    let pown_point (x: Float64.t) (y: int) : interval
2     (* Specification 1 *)
3    = if y = 0 then of_float oneF
4      else if x .= infinity then of_float x
5      else if x .= neg_infinity then begin
6        if y % 2 = 0 then of_float (neg x) else of_float x
7      end
8      else begin
9        let rec aux (r xp: interval) (n: int) : interval
10        (* Specification 2 *)
11       = if n = 0 then r
12         else begin
13           let r' =
14             if n % 2 <> 0 then begin
15               mul r xp
16             end else r
17           in
18           aux r' (mul xp xp) (n / 2)
19         end
20       in
21       let r0 = { inf = oneF; sup = oneF } in
22       let xp0 = of_float x in
23       aux r0 xp0 y
24     end
```

図 2　浮動小数点数の整数べき関数の WhyML 言語による実装

xp0 $= [3.1415, 3.1415]$，$n = 6$ の場合:

1. 23 行目にて関数を評価

   `aux [1, 1] [3.1415, 3.1415] 6`

2. $n = 6$ で偶数のため 16 行目へ

   `r' = [1, 1]`

3. 18 行目にて自身 aux を再帰呼出し

   `aux [1, 1] [9.8690, 9.8691] 3`

4. $n = 3$ で奇数のため 15 行目へ

   `r' = mul [1, 1] [9.8690, 9.8691]`

```
1   requires { valid x }
2   requires { y >= 0 }
3   ensures { valid result }
4   ensures { forall t. real_in t x
5     -> real_in (power t y) result }
```

図 3 関数 pown に **Why3** 言語にて付与した事前/事後条件

```
1   requires { is_not_nan x }
2   requires { 0 <= y }
3   ensures { valid result }
4   ensures { t'isFinite x
5     -> real_in
6       (power (t'real x) y) result }
7   ensures { y = 0
8     -> real_in 1. result }
9   ensures { is_plus_infinity x
10    -> y <> 0
11      -> is_plus_infinity
12        result.sup }
13  ensures { is_minus_infinity x
14    -> y <> 0 -> mod y 2 = 0
15      -> is_plus_infinity
16        result.sup }
17  ensures { is_minus_infinity x
18    -> mod y 2 = 1
19      -> is_minus_infinity
20        result.inf }
```

図 4 関数 pown_point に **Why3** 言語にて付与した事前/事後条件

```
= [9.8690, 9.8691]
```

5. 18 行目にて自身 aux を再帰呼出し

```
aux [9.8690, 9.8691] [97.397, 97.400] 1
```

6. $n = 1$ で奇数のため 15 行目へ

```
r' = mul [9.8690, 9.8691] [97.397, 97.400]
= [961.21, 961.25]
```

7. 18 行目にて自身 aux を再帰呼出し

```
aux [961.21, 961.25] [9486.1, 9486.8] 0
```

8. $n = 0$ のため

```
r = [961.21, 961.25]
```

を返して終了

理論値は 961.219$\cdots$ のため，理論値を含んだ区間を計算できていることがわかる．

4.3 事前・事後条件のアノテーション

本研究では，前節の実装が「正しい区間として底を，正数として指数を与えられると，正しい区間としてべきの区間包囲を計算する」ことを検証する．検証にあたり，前節の実装に事前条件と事後条件をアノテーションする．関数 pown の事前条件には，区間の下限と上限の大小関係が正しい区間が与えられていること，及び指数の値が正の値となっていることを記述している．事後条件には，結果として正しい区間が得られているか，及び健全性を記述している．関数 pown に付与する Why3 言語にて記述した事前条件と事後条件を図 3 に示す．

```
1   requires { valid r }
2   requires { valid xp }
3   requires { 0 <= n <= y }
4   variant { n }
5   ensures { valid result }
6   ensures { t'isFinite x
7     -> real_in
8       (power (t'real x) y) result }
9   ensures { is_plus_infinity
10    x /\ y <> 0
11      -> is_plus_infinity
12        result.sup }
```

図 5 関数 aux に **Why3** 言語にて付与した事前/事後条件

また，関数 pown_point にも同様に事前条件と事後条件を付与する．Why3 言語にて記述した事前条件と事後条件をそれぞれ図 4，図 5 に示す．図 4 は関数 pown_point に付与したもの，図 5 は関数 aux に付与したものである．

関数 pown_point では，事前条件として正しい引数が与えられていることを記述し，事後条件として，得られた結果が正しい区間であること，及び健全性を記述している．関数 aux では，事前条件として正しい区間が与えられていることを記述し，事後条件として，得られた結果が正しい区間であること，及び健全性を記述し，また再帰処理の停止性として再帰処理ごとに減少する値を variant として記述している．

仕様付きプログラムが準備できたため，Why3 ツールでプログラムを読み込むことにより検証条件が自動生成される．検証条件は，定義した関数の数生成される．ここでは，関数 pown，関数 pown_point の検証条件 2 つが生成される．Why3 では，生成された検証条件を更に細かな検証条件へ分割する split 機能が備わっている．関数 pown の検証条件を分割した結果，13 個の検証条件に分割できた．分割した検証条件に対し Alt-Ergo を適用すると，12 個を証明でき，

残り 1 個は時間切れとなった．関数 pown_point の検証条件を分割した結果，23 個の検証条件に分割できた．分割した検証条件に対し Alt-Ergo を適用すると，22 個を証明でき，残り 1 個は時間切れとなった．

4.4 補助アノテーションの記述・証明

Why3 と Alt-Ergo を用いた検証では，補助定理が有用である．補助定理は，PownTheory モジュールとして別ファイルに定義し，インポートすることで利用する．追加した補助定理は 2 種類に分類できる．

1 種類目は，除算と剰余算に関する補助定理である．ここでは，追加した中の 1 つである Z_div_mod_eq について説明する．

```
1  lemma Z_div_mod_eq: forall a b.
2    b > 0
3      -> a = b * (div a b) + (mod a b)
```

この補助定理は，Coq の記述 †4 の Z_div_mod_eq を元に追加したものであり，除算と剰余算の性質を記述している．

2 種類目は，実数の整数べき関数に関する補助定理である．ここでは，追加した中の 1 つである Rpower_incr について説明する．

```
1  lemma Rpower_incr: forall x y n.
2    0. <=. x <=. y /\ n >= 0
3      -> power x n <=. power y n
```

この補助定理も，Z_div_mod_eq と同様 Coq の記述を元に追加したものであり，底における単調増加性を記述している．

他にも，プログラムの途中で成り立つべき条件を assert 式によりアノテーションすることにより，検証に必要な証明プロセスを補助することができる．以下は関数 pown_point（図 2） の 21 行目と 22 行目の間に追加した assert 式である．

```
1  assert { t'isFinite x <->
2    xp0.inf .= x /\ xp0.sup .= x };
```

このように，プログラムの途中で成り立つべき条件を記述することで，証明の補助を行う．

†4 https://coq.inria.fr/library/

表 1 追加した補助定理と assert の式

関数，モジュール	補助定理	assert 式
pown	0	8
pown_point	0	2
pown_theory	11	0

Why3 には，ゴースト変数という仕様記述用の変数がある．今回，関数 pown_point（図 2）内の関数 aux の引数 r と xp がそれぞれ x の何乗であるかを示すゴースト変数 log_r と log_xp を導入した．これにより，健全性を満たしつつ再帰処理を行っていることを仕様に記述できる．2 行目は，再帰関数 aux に引数として 2 つのゴースト変数 log_r と log_xp を追加している．3 ～ 6 行目は，ゴースト変数を追加したことにより増えた事前条件である．再帰関数の事前条件は，再帰呼出しされるたびに成り立つことを示すため，再帰処理されている間常に成り立つことを意味する．8 ～ 12 行目はゴースト変数の更新処理をしており，13 行目にて再帰呼び出ししている．

4.5 検証結果

4.4 節により，追加した補助定理，assert 式の数を，表 1 に示す． 検証は，Intel Xeon プロセッサ E5–2650v4 2.2GHz と 128GB のメモリを備える計算機上で行い，タイムアウト時間を 600s と設定した．その結果，表 2 のとおり検証結果が得られた．表 2 の 2 列目は，生成された検証条件を split 機能により分割した後の検証条件の数である．その検証条件に対し，Alt-Ergo にて証明できた検証条件の数を列 3，Coq にて証明できた検証条件の数を列 4 に示している．列 5 は Alt-Ergo にて行った各検証条件の証明に要した時間の合計である．

5 関連研究

5.1 区間四則演算の検証

Ayad ら [1] や薮ら [4] が区間四則演算のプログラム検証事例を報告している．前者は Frama-C，後者は Why3 を検証ツールとして利用した．四則演算を実装した繰り返し処理がなく分岐処理のみからなるプ

```
let rec aux (r xp: interval) (n: int)
            (ghost log_r log_xp: int) : interval
  requires { 0 <= log_r < log_xp }
  requires { t'isFinite x -> real_in (power (t'real x) log_r) r }
  requires { t'isFinite x -> real_in (power (t'real x) log_xp) xp }
  requires { y = n * log_xp + log_r }
  . . .
    let ghost log_r' =
      if n % 2 <> 0 then begin
        log_r + log_xp
      end else log_r
    in
    aux r' (mul xp xp) (n / 2) log_r' (log_xp * 2)
```

図 6　関数 aux に導入したゴースト変数

表 2　検証結果

関数，モジュール	VC	Alt-Ergo (2.2.0)	Coq (8.8.0)	CPU 時間
pown	21	21	0	600s
pown_point	37	37	0	26s
pown_theory	11	5	6	3.4s

ログラムに対し，本研究と同様に事前・事後条件と補助定理を付与し，SMT ソルバーや Coq を用いて検証を実施した．事後条件として，両者は計算結果が真解を含んでいる健全性を記述し，後者はさらに計算結果が引数区間の端点から計算できる幅が小さい区間であること (tight 性) を検証した．本研究では四則演算を用いた繰り返し処理によるべき関数を検証した．本研究は健全性のみを事後条件としたが，tight 性の検証は今後の課題である．

5.2 CRlibm

CRlibm [2] は効率的かつ丸めの正しさが証明された，浮動小数点数に基づく汎用の数学ライブラリである．CRlibm での正しさの証明は，定理証明器 Gappa などを用いて区間解析により数値誤差を評価し，演算結果が指定した丸め方向のもっとも近い値であるかどうか等の性質を検証している．本研究とは検証する性質が異なり，また本研究が検証ツールによる軽量な検証を指向しているのに対し，検証が複雑な証明プロセスを経ている点が異なる．

6 まとめと今後の課題

今回，区間の整数べきを計算する関数 pown の実装，及び検証実験結果を示した．以上の実験により，元の数学関数の定理が区間の関数でも有用であることがわかった．また，assert 式により途中式を記述することが，本研究においてとても有用であった．今後の課題として，結果として得られる区間の tight 性の検証が挙げられる．

謝辞　本研究の一部は科研費 J180000175, J180000528 の補助を得て行った．

参考文献

[1] Ali Ayad and Claude Marché: Multi-Prover Verification of Floating-Point Programs, *IJCAR*, LNAI 6173, pp. 127–141, 2010.

[2] C. Daramy-Loirat, D. Defour, F. de Dinechin, M. Gallet, N. Gast, C. Q. Lauter, and J.-M. Muller: CR-LIBM - A library of correctly rounded elementary functions in double-precision, Version 1.0 beta 5, 2016.

[3] Ramon E. Moore: Interval Analysis, 1966, *Prentic-Hall*.

[4] Tomohito Yabu, Daisuke Ishii. Machine-Aided Verification of Four Interval Arithmetic Operators. In International Symposium on Scientific Computing, Computer Arithmetic, and Verified Numerical Computations (SCAN), pages 176–177, 2018.