

JavaScript Interval Library

(Interval Arithmetic & Decoration)

Version 1.5

By Henrik Vestermark (hve@hvks.com)

Contents

Introduction.....	6
Design Rationale: Improper Intervals.....	7
Why are improper intervals allowed.....	7
Interval Arithmetic vs IEEE-754 Floating-Point Arithmetic.....	8
IEEE-754 arithmetic	8
Interval arithmetic.....	8
Key differences	9
When to use interval arithmetic	9
Complementary, not competing.....	9
When Not to Use Interval Arithmetic.....	9
Performance-critical inner loops.....	9
Algorithms that rely on exact equality	10
Problems with the unbounded growth of uncertainty	10
Situations where domain violations are acceptable	10
Simple, well-understood numerical code.....	11
Combining Scalar and Interval Code Safely.....	11
General rule.....	11
Convert scalars explicitly to singleton or parsed intervals	12
Avoid mixing scalar operations inside interval algorithms	12
Preserve the interval’s semantic checks.....	12
Use containment tests instead of equality.....	13
Keep rounding and outward enclosure in one place	13
Decide whether the scalar is “exact input” or “approximate measurement”	13
Recommended practice	14
Interval	15
Constructor.....	15
Arguments.....	15
Returns	15
Internal Format of an Interval Object	15
The Empty Interval	16
Not an Interval (NaN).....	16
Interval Decorations.....	17
Hull and Union.....	17

Arithmetic Interval operations	18
Logical interval operations	18
Polyfills functions.....	18
Methods.....	18
Functions.....	19
Constants.....	20
Miscellaneous	20
API	20
Interval.abs()	21
Interval.add().....	21
Interval.acos()	22
Interval.acosh()	22
Interval.asin()	23
Interval.asinh()	23
Interval.atan().....	24
Interval.atan2().....	24
Interval.atanh().....	25
Interval.center().....	26
Interval.cos()	26
Interval.cosh()	27
Interval.decorationToString()	27
Interval.div()	28
Interval.E.....	28
Interval.equal().....	29
Interval.exp().....	30
Interval.greater ()	30
Interval.greaterequal().....	31
Interval.hull()	32
Interval.in()	33
Interval.inf()	33
Interval.interior().....	34
Interval.intersection().....	34
Interval.intervaldecoration()	35
Interval.intervaltype()	36

Interval.isEmpty()	37
Interval.isEntire()	37
Interval.isImproper()	38
Interval.isNaN()	38
Interval.isPoint()	39
Interval.isProper()	39
Interval.leftinterval()	40
Interval.less ()	40
Interval.lessequal()	41
Interval.LN2	42
Interval.LN10	42
Interval.log()	43
Interval.log10()	44
Interval.mag()	44
Interval.mig()	45
Interval.mul()	45
Interval.neg()	46
Interval.notequal()	46
Interval.one	47
Interval.PI	47
Interval.pow()	48
Interval.precedes()	49
Interval.pred()	49
Interval.radius()	50
Interval.rightinterval()	50
Interval.sin()	51
Interval.sinh()	52
Interval.sub()	52
Interval.sup()	53
Interval.SQRT2()	53
Interval.sqr()	54
Interval.sqrt()	54
Interval.succ()	55
Interval.tan()	55

Interval.tanh().....	56
Interval.toClose()	56
Interval.toExponential()	57
Interval.toFixed()	58
Interval.toOpen().....	59
Interval.toPrecision()	59
Interval.toString().....	60
Interval.union()	61
Interval.valueOf()	62
Interval.width()	62
Interval.zero	63
parseInterval().....	63
Example	65

Introduction

This library provides full support for interval arithmetic in JavaScript, including IEEE-1788–style interval decorations, directed rounding, and mathematically rigorous enclosure of real values. It is designed for users who require numerically reliable results, where rounding errors, representation limits, and domain restrictions must be handled explicitly rather than implicitly.

Unlike standard floating-point arithmetic, which represents a single approximate value, interval arithmetic represents a range of possible values that is guaranteed to contain the exact mathematical result. Each operation produces an interval that encloses all rounding and approximation effects introduced by finite-precision computation.

The library supports:

- Closed, open, and half-open intervals
- Proper, improper, empty, entire, and NaI intervals
- IEEE-1788 interval decorations (ILL, TRV, DEF, DAC, COM)
- Arithmetic, transcendental, and logical interval operations
- Exact parsing of decimal input via tight enclosures

The implementation follows these guiding principles:

1. Correct enclosure over performance
All operations are designed to produce correct enclosing intervals, even when this requires outward rounding or interval widening.
2. Explicit handling of undefined results
Invalid operations, such as division by an interval containing zero or logarithms outside their domain, result in well-defined NaI intervals rather than silent failures.
3. Separation of mathematical meaning and representation
Empty intervals, improper intervals, and NaI are distinct concepts. An empty interval is not represented by reversed bounds, and improper intervals are permitted internally.
4. IEEE-1788–inspired decoration propagation
Each interval carries a decoration that reflects the reliability and definedness of the result. Decorations are propagated conservatively through all operations.
5. Interoperability with JavaScript numbers
Interval objects interoperate naturally with JavaScript numbers while preserving interval semantics. Scalar inputs are converted to singleton intervals when required.

This library is intended for:

- Numerical analysis and scientific computing
- Verified and validated numerical algorithms

- Robust geometric and algebraic computations
- Educational use where floating-point behavior must be made explicit

The manual is organized as follows:

- Core interval concepts and representations
- Interval types, emptiness, and NaN
- Decorations and their propagation rules
- Arithmetic, transcendental, and logical operations
- Formatting, parsing, and utility methods
- Complete API reference with examples

Readers are expected to be familiar with basic concepts of floating-point arithmetic and numerical computation. Prior knowledge of interval arithmetic is helpful but not required.

Design Rationale: Improper Intervals

This library deliberately supports improper intervals, where the left endpoint may be greater than the right endpoint.

An improper interval is not the empty interval.

Why are improper intervals allowed?

Allowing improper intervals is a conscious design choice motivated by correctness, robustness, and alignment with advanced interval arithmetic techniques.

1. Separation of semantics from representation
An interval's *mathematical meaning* (empty, defined, undefined) is not inferred solely from endpoint ordering.
Emptiness is represented explicitly by a dedicated interval type, not by reversed bounds.
2. Correct handling of directed rounding
During intermediate computations, especially division and transcendental functions, temporary improper intervals may arise before outward rounding is applied. Treating such intermediates as empty would incorrectly discard valid enclosures.
3. Support for Kaucher-style arithmetic patterns
While this library primarily targets classical interval arithmetic, allowing improper intervals enables extensions such as:
 - Dual arithmetic
 - Certain constraint-solving techniques
 - Robust enclosure algorithms that temporarily invert or reorder bounds
4. Avoidance of false emptiness
Automatically classifying `left > right` as empty can introduce subtle bugs,

particularly in operations involving reciprocal intervals or sign-dependent rounding.

5. Explicit control over emptiness

An interval is empty only when explicitly constructed as such or when produced by an operation whose result is mathematically empty. This makes emptiness a semantic property rather than an incidental artifact of floating-point rounding.

Interval Arithmetic vs IEEE-754 Floating-Point Arithmetic

Standard IEEE-754 floating-point arithmetic represents single approximate values. Interval arithmetic represents ranges of values that are guaranteed to contain the exact mathematical result.

IEEE-754 arithmetic

In IEEE-754 arithmetic:

- Each operation returns a single rounded value
- Rounding errors are implicit and often invisible
- Domain errors may produce NaN or \pm Infinity
- No information is preserved about uncertainty or error bounds

Example:

```
0.1 + 0.2 === 0.30000000000000004
```

The result is a rounded approximation with no indication of error.

Interval arithmetic

In interval arithmetic:

- Each operation returns an interval enclosing the exact result
- Rounding errors are explicitly captured
- Domain violations produce NaN, not silent NaN propagation
- Results include both numeric bounds and a decoration describing reliability

Example:

```
Interval(0.1).add(Interval(0.2))  
→ [0.29999999999999993, 0.30000000000000004]
```

This interval provably contains the exact real result 0.3.

Key differences

Aspect	IEEE-754 scalar	Interval arithmetic
Result	Single value	Guaranteed enclosure
Rounding error	Implicit	Explicit
Error tracking	None	Built-in
Domain violations	NaN / \pm Infinity	NaN with decoration
Reliability	Assumed	Verified
Composition	Error-prone	Safe under composition

When to use interval arithmetic

Interval arithmetic is especially valuable when:

- Correctness matters more than raw speed
- Results must be provably reliable
- Algorithms are sensitive to rounding or cancellation
- Domain boundaries must be respected rigorously
- Debugging numerical instability is required

IEEE-754 arithmetic remains appropriate for performance-critical code where small rounding errors are acceptable and well understood.

Complementary, not competing

This library does not replace IEEE-754 arithmetic. Instead, it builds on it, using IEEE-754 operations with controlled outward rounding to produce mathematically rigorous results.

Intervals make floating-point behavior explicit, inspectable, and safe.

When Not to Use Interval Arithmetic

While interval arithmetic provides strong correctness guarantees, it is not always the appropriate tool. Users should be aware of its limitations and trade-offs.

Performance-critical inner loops

Interval arithmetic is inherently more expensive than scalar IEEE-754 arithmetic. Each operation involves multiple floating-point operations, outward rounding, and decoration handling.

Interval arithmetic is therefore not well-suited for:

- Tight inner loops with millions of iterations
- Real-time or latency-critical code paths
- Graphics, audio, or signal-processing pipelines where a small numerical error is acceptable

In such cases, standard floating-point arithmetic is usually preferable.

Algorithms that rely on exact equality

Interval arithmetic tracks uncertainty explicitly. As a result:

- Exact equality tests (`==`) are rarely meaningful
- Results that are mathematically equal may still differ in interval width or decoration

Algorithms that depend on strict equality comparisons or bit-wise reproducibility are often a poor fit for interval arithmetic.

Problems with the unbounded growth of uncertainty

Repeated interval operations can lead to overestimation, especially in long iterative computations or when variables are reused without contraction.

Examples include:

- Naive fixed-point iterations
- Poorly conditioned problems
- Algorithms that repeatedly widen intervals without refinement

In such cases, interval arithmetic may produce results that are technically correct but too wide to be useful.

Situations where domain violations are acceptable

Interval arithmetic treats domain violations conservatively. Operations outside their mathematical domain produce NaI rather than silently continuing.

If an application:

- Intentionally allows NaN or \pm Infinity
- Uses NaN as a control-flow mechanism
- Does not require rigorous domain enforcement

Then interval arithmetic may feel restrictive.

Simple, well-understood numerical code

For many applications:

- Inputs are well-conditioned
- Numerical behavior is well understood
- Small rounding errors are irrelevant

Using interval arithmetic in such cases may add complexity without meaningful benefit.

Interval arithmetic should be used when correctness, robustness, and verifiability are more important than raw performance or simplicity.

It is not a replacement for standard floating-point arithmetic, but a complementary tool for problems where numerical reliability must be explicit and guaranteed.

Combining Scalar and Interval Code Safely

In many applications, it is practical to mix standard IEEE-754 scalar computations with interval computations. This can be efficient and convenient, but it requires discipline to avoid accidentally losing the enclosure guarantees that interval arithmetic provides.

General rule

Use scalars for:

- control flow, indexing, UI, formatting
- cheap prechecks (sign tests, rough magnitude checks)
- performance-critical code where bounded error is acceptable

Use intervals for:

- final results that must be guaranteed correct
- computations near singularities or domain boundaries
- numerically sensitive operations (cancellation, division, transcendental chains)

Once an interval is introduced into a reliable computation path, keep subsequent steps in interval form until the end.

Convert scalars explicitly to singleton or parsed intervals

If a scalar x represents an exact binary64 value (for example, values already produced by computations), use a singleton interval:

```
const ix = new Interval(x, x);
```

If a scalar originates from a decimal literal or user input and should be treated as an exact real decimal, parse it as an enclosing interval:

```
const ix = parseInterval("0.1"); // tight enclosure of the real  
decimal 0.1
```

Do not assume that decimal literals such as 0.1 are exact in binary.

Avoid mixing scalar operations inside interval algorithms

A common mistake is to compute a scalar approximation inside an interval function and then wrap it back into an interval without outward rounding. This can lose the containment guarantee.

Bad pattern:

```
// produces an interval, but may be inward  
const y = new Interval(Math.sin(x.inf()), Math.sin(x.sup()));
```

Better pattern:

- use the interval implementation (`Interval.sin(x)`) so outward rounding and critical points are handled
- or explicitly widen outward if you must use scalars

Preserve the interval's semantic checks

Scalar code often checks domain conditions with single comparisons. With intervals, you must check the entire range.

Examples:

Logarithm

- scalar: `if (x <= 0) error`
- interval: `if sup(x) <= 0 undefined, if inf(x) <= 0 < sup(x) result may be unbounded and decoration degrades`

Division

- scalar: if $(b === 0)$ error
- interval: if $0 \in b$ result is unbounded or NaN, depending on your definition and decoration rules

Use containment tests instead of equality

When comparing interval results to scalar values, prefer containment:

```
iv.in(x)
```

or compare scalar to interval bounds:

```
iv.inf() <= x && x <= iv.sup()
```

Avoid `iv == x` style comparisons.

Keep rounding and outward enclosure in one place

If you do any mixed-mode arithmetic where the core computation is scalar but the result must be an interval, ensure the conversion step is conservative:

- compute scalar approximation q
- widen outward by at least one ulp on each side if exactness cannot be proven

Example pattern:

```
const q = a / b;  
const r = new Interval(Math.pred(q), Math.succ(q));
```

This is especially important for division, reciprocal, and transcendental functions.

Decide whether the scalar is “exact input” or “approximate measurement.”

A useful practical distinction:

1. Exact intended value (user typed “0.1”, constants, configuration)
Use `parseInterval()` to obtain a correct enclosure of the real decimal.
2. Computed value (already rounded IEEE-754 result)
Use a singleton interval $[x, x]$, because the exact value you intend to propagate is the binary64 value already stored in x .

This avoids double-counting uncertainty.

Recommended practice

- Keep public API tolerant: accept numbers, but convert internally to intervals immediately.
- Use `parseInterval()` at the boundary where human-readable decimals enter.
- Do not “unwrap” intervals to scalars mid-computation unless you accept losing guarantees.
- If you must unwrap for speed, document that the result is an approximation and may not be enclosed.

Interval

Support for Interval number arithmetic in JavaScript

Constructor

`new Interval(left,right,type)` // Invoke as a Constructor
`Interval(value)` // Invoke as a Conversion

Arguments

left The left part of an Interval number
right The optional right part of an Interval number
type The optional interval type.
decoration The interval decoration in accordance with IEEE1788

If the right argument is omitted, it is treated as the same as the left argument. (Singleton Interval)

If there are no arguments, it is treated as an Empty Interval

If an *Interval* is invoked as a conversion, the value parameter is converted to an *Interval number* and returned.

The interval type parameter specifies whether an interval is closed, open, or half-open.

Returns

Returns an Interval object initialized with the left and right values. If an *Interval* is invoked as a conversion, the *value* parameter is converted to an *Interval number* and returned; if *the value is another Interval number*, it is returned.

Internal Format of an Interval Object

The Interval is stored as an Object with the following fields:

Object.left Store the left end of the Interval.
Object.right Store the right end of the Interval.
Object.type Is indicating if the Interval is empty, closed, open, or semi-open intervals.
Object.decoration Is the current decoration status as per the IEEE1788 standard

A closed interval is written in the notation within [] brackets. Open with (). If only one side is open or closed, it is called a half-open interval.

<i>Notation</i>	<i>Type Parameter</i>	<i>Explanation</i>
-----------------	-----------------------	--------------------

(a,b)	$a < x < b$	"()"	<i>An Open interval</i>
$[a,b)$	$a \leq x < b$	"["	<i>Closed on the left, Open on the right, also named half open</i>
$(a,b]$	$a < x \leq b$	"]"	<i>Open on the left, closed on the right, also named half open</i>
$[a,b]$	$a \leq x \leq b$	"["	<i>A Closed interval</i>

The interval type is written as a two-character string when creating an interval. See above. If an Interval only contains a singleton value, by definition, the (a, a) , $(a, a]$, $[a, a)$ represent the empty set, where $[a, a]$ represents the set a . You can convert an open set to the closed form by calling `Interval.toClose()` or an interval to its open form by calling `Interval.toOpen()`. Note that an interval can also contain the *undefined* value or *NaN* for both the lower and the upper part of the Interval.

The Empty Interval

The empty interval represents the empty set and is a distinct interval type. Important notes:

- The empty interval is not represented by reversed bounds.
- Improper intervals (where `left > right`) are permitted internally and are not automatically empty.
- Emptiness is determined by the interval type, not by endpoint ordering.

The empty interval:

- Has type `EMPTY`
- Has decoration `TRV`
- Causes most arithmetic operations to return the empty interval
- Is distinct from `NaN`

This distinction is necessary to support proper IEEE 1788 semantics and directed rounding.

Not an Interval (NaN)

In accordance with IEEE 1788, the library supports a special state called NaN (Not an Interval).

NaN represents an ill-formed or undefined interval result.

A NaN interval is distinct from the empty interval.

Typical situations that produce NaN include:

- Division by an interval containing zero
- Applying a function outside its mathematical domain (e.g. $\log(x)$ for $x \leq 0$)

- Explicit construction via `Interval.NaI()`

Most interval operations propagate `NaI`:
If any operand is `NaI`, the result is `NaI`.

`NaI` intervals:

- Are not empty
- Have decoration `ILL`
- Cause predicates such as `in()`, `subset()`, etc. to return `false`

Interval Decorations

Each interval carries a decoration, following the IEEE 1788 standard:

- `ILL` – Ill-formed, Not an Interval (`NaI`)
- `TRV` – Trivial, no information
- `DEF` – Defined and non-empty
- `DAC` – Defined, continuous and non-empty
- `COM` – Common, bounded and non-empty

Decorations are propagated through operations using the minimum of the operand decorations, with additional degradation where required by the operation (for example, division over an interval spanning zero degrades to `TRV`).

The empty interval has decoration `TRV`.
`NaI` always has decoration `ILL`.

Hull and Union

The library distinguishes between **hull** and **union**, following IEEE 1788 semantics.

`Interval.hull(a, b)`

Returns the **convex hull** of two intervals.

The hull is the smallest interval that contains both operands.

- Always returns an interval (unless one operand is `NaI`)
- Does not require the intervals to overlap
- Equivalent to $[\min(\inf(a), \inf(b)), \max(\sup(a), \sup(b))]$

`Interval.union(a, b)`

Returns the set-theoretic union only if the result is connected.

- If the intervals overlap or touch, the result is an interval

- If the intervals are disjoint, the union is not representable as a single interval, and `NaN` is returned

Users should prefer `hull()` when a guaranteed interval result is required.

Arithmetic Interval operations

There is the usual interval arithmetic function, like adding, subtracting, multiplying, and dividing intervals. Notice that all arithmetic or Interval Math functions maintain the form type (Closed, Open, etc.).

Logical interval operations

There are also several logical operations, such as the union and intersection of intervals. Check for a subset, interior of intervals, if an interval precedes another interval or a number is within an interval range, and the six comparison types: `>`, `<`, `>=`, `<=`, `==` or `!=`

Polyfilla functions.

There is a need for certain functions commonly found in C++ libraries. These are:

```
Math.ipow()
Math.lDEXP()
Math.fREXP()
Math.nextafter()
Math.succ() // Shortcut for nextafter(a, Number.POSITIVE_INFINITY)
Math.pred() // Shortcut for nextafter(a, Number.NEGATIVE_INFINITY)
Math.fMA()
Number.EPSILON
```

They are all implemented in such a way that if JavaScript Math is later on defined, then it will default to the JavaScript Math library.

Methods

```
abs()          Return the absolute value of the Interval number
center()       Return the center (or midpoint) of the Interval number
in()           Test if a number is within the interval range. Return true or false.
inf()          Returns the infimum (lowest value) of the Interval.
intervaldecoration() Get or set the Interval's decorations as per IEEE1788
intervaltype() Get or set the Interval's type, representing different states like open,
closed, empty, etc.
isEmpty()      Test if the Interval is empty or not. Return true or false.
isEntire()     Return true if the Interval represents the entire range of real numbers.
```

isImproper()	Return true if the Interval is improper. ($\text{left} > \text{right}$).
isNaI	Return true if the Interval is not an interval, otherwise false.
isPoint()	Return true if the Interval is a point (singleton) interval.
isProper()	Return true if the Interval is proper ($\text{left} \leq \text{right}$).
leftinterval()	Return or set the left part of the Interval number.
mag()	Return the largest absolute value of the Interval's endpoints.
mig()	Return the smallest absolute value of the Interval's endpoints.
neg()	Return a new Negated Interval Object.
radius()	Return the radius (half-width) of the interval
rightinterval()	Return or set the right part of the Interval.
decorationToString()	Return a string representing the Interval's decoration status.
sup()	Return the supremum (highest value) of the Interval
toNumber	Convert an interval number to a floating point using the midpoint of the interval number. Same as the center() method.
toExponential()	Converts an Interval number to a string using exponential notation with the specified number of digits after the Decimal place.
toFixed()	Converts an Interval number to a string that contains a specified number of digits after the decimal place.
toPrecision()	Convert an Interval number to a string using the specified number of precision digits. Uses exponential or fixed-point notation depending on the size of the number and the number of significant digits specified.
toString()	Convert an Interval number to a string using a specified radix(base)
valueOf()	The primitive lower number value of this Interval number object.
width()	Return the width of the interval number

Functions

abs()	Return the magnitude of an Interval number
add()	Return the addition of two Interval numbers
acos()	Return arc cosine of the Interval number
acosh()	Return the arc cosine hyperbolic of the Interval number
asin()	Return the arcsine of the Interval number
asinh()	Return the arc sine hyperbolic of the Interval number
atan()	Return the arc tangent of the Interval number
atan2()	Return the arc tangent of the quotient of its arguments
atanh()	Return the arctangent hyperbolic of the Interval number
cos()	Return the cosine of the Interval number
cosh()	Return the cosine hyperbolic of the Interval number
div()	Return the division of two Interval numbers.
E	Return $\exp(1)$ as a constant interval.
equal()	Return the Boolean value (true, false) of the equality of two Interval numbers.
exp()	Return the Interval power of e.
greater()	Return the Boolean value (true, false) of the interval comparison
greaterequal()	Return the Boolean value (true, false) of the interval comparison
hull()	Return the convex hull of the two intervals.

interior()	Return true if an interval is located within another interval; otherwise, false.
intersection()	Return the Intersection of two interval numbers. (and)
less()	Return the Boolean value (true, false) of the interval comparison
lessequal()	Return the Boolean value (true, false) of the interval comparison
LN2	Return Log(2) as a constant interval.
LN10	Return Log(10) as a constant interval.
log()	Return the Interval natural logarithm.
log10()	Return the Interval base ten logarithms.
mod()	Return the modulus of two interval numbers.
mul()	Return the product of two Interval numbers.
notequal()	Return the Boolean value (true, false) of the inequality of two Interval numbers.
PI	Return π as a constant interval.
pow()	Return the Interval power of x^y .
precedes ()	Return true if an interval precedes another interval. Otherwise, false
pred()	Return the predecessor number to a floating-point number
sin()	Return the sine of the Interval number
sinh()	Return the sine hyperbolic of the Interval number
SQRT2	Return the constant interval for the square root of two
sqr()	Return the square of an interval number.
sqrt()	Return the Interval square root.
sub()	Return the difference between two Interval numbers.
subset()	Return true if an interval is a subset of another interval. Otherwise, false
succ()	Return the successor number to a floating-point number
tan()	Return the tangent of the Interval number
tanh()	Return the tangent hyperbolic of the Interval number
toClose()	Convert and return a new closed interval object
toOpen()	Convert an return a new open interval object
union()	Return the union (or) of the two intervals

Constants

Interval.NaI	Return the 'Not an interval' object
Interval.zero	Return a new Interval(0,0) object
Interval.one	Return a new Interval(1,1) object

Miscellaneous

parseInterval()	Parse an Interval float number string
-----------------	---------------------------------------

API

Interval.abs()

Return the absolute value of the Interval number.

Synopsis

Interval object.abs()

Returns

The absolute value of the Interval number is returned.

Example

```
var z = new Interval(3,-4);  
z.abs()           // result [3,4]
```

See Also

Interval.negate()

Interval.add()

Add two Interval numbers.

Synopsis

Interval.add(a,b)

Arguments

a,b The interval numbers are to be added.

Returns

The result of the Interval addition operation.

Example

```
var z = new Interval(3,4);  
var y=new Interval(1,2);  
  
Interval.add(z,y)           // result [4:6]
```

See Also

Interval.div(), Interval.mul(), Interval.sub()

Interval.acos()

Return the arc cosine of the Interval number.

Synopsis

Interval.acos(a)

Returns

Return the arc cosine of the Interval number. If $a > 1$ or $a < -1$, then it returns `Interval(NaN)`.

Example

```
var z = new Interval(0.5,0.6);
Interval.acos(z)           // result [0.9272952180016133, 1.0471975511965967]
Interval.acos(Interval(0.5)); // result [1.0471975511965967, 1.0471975511965992]
```

See Also

Interval.asin(), Interval.atan()

Interval.acosh()

Return the arc cosine hyperbolic of the Interval number.

Synopsis

Interval.acosh(a)

Returns

Return the arc cosine hyperbolic of the Interval number.

Example

```
var z = new Interval(3,4);
```

```
Interval.acosh(z) // result  
[1.7627471740390848,[0.96242365011920661,0.96242365011920705]  
2.0634370688955634]  
Interval.acosh(Interval(1.5)); // result [0.9624236501192058, 0.9624236501192067]  
Interval.acosh(Interval(0.5)); // result [NaN,NaN]
```

See Also

Interval.asinh(), Interval.atanh()

Interval.asin()

Return the arcsine of the Interval number.

Synopsis

Interval.asin(a)

Returns

Return the arc sine of the Interval number. If $a > 1$ or $a < -1$, then it returns `Interval(NaN)`.

Example

```
var z = new Interval(0.5,0.6);  
Interval.asin(z) // result [0.5235987755982974, 0.6435011087932853]  
Interval.asin(Interval(0.5)); // result [0.5235987755982974, 0.5235987755982998]
```

See Also

Interval.acos(), Interval.atan()

Interval.asinh()

Return the arc sine hyperbolic of the Interval number.

Synopsis

Interval.asinh(a)

Returns

Return the arc sine hyperbolic of the Interval number.

Example

```
var z = new Interval(3,4);
Interval.asinh(z)           // result [1.8184464592320593, 2.0947125472611066]
Interval.asinh(Interval(0.5)); // result [0.4812118250596029, 0.4812118250596051]
```

See Also

Interval.acosh(), Interval.atanh()

Interval.atan()

Return the arctangent of the Interval number.

Synopsis

Interval.atan(a)

Returns

Return the arc tangent of the Interval number.

Example

```
var z = new Interval(3,4);
Interval.atan(z)           // result [1.2490457723982522,1.3258176636680343]
Interval.atan(Interval(0.5)); // result [0.4636476090008055,0.4636476090008068]
```

See Also

Interval.acos(), Interval.asin(), Interval.atan2()

Interval.atan2()

Calculate atan2() of the Interval numbers.

Synopsis

Interval.atan2(y,x)

Arguments

y The Interval number of the Y coordinate of the point
x The Interval number of the X coordinate of the point

Returns

The result of the `Interval.atan2()` which is an Interval number between $-\pi$ and $+\pi$

Description

The `Interval.atan2()` function computes the arc tangent of the ratio y/x . The *y* argument can be considered the Y coordinate of an Interval, and the *x* argument can be the X coordinate of the Interval. Note the unusual order of the arguments to this function. The Y coordinate is passed before the X coordinate.

Example

```
var x=new Interval(1), y=new Interval(2);  
Interval.atan2(y,x)            // result [1.1071487177940862,1.1071487177940942]
```

See Also

`BigFloat.acos()`, `BigFloat.atan()`, `BigFloat.asin()`

`Interval.atanh()`

Return the arc tangent hyperbolic of the Interval number.

Synopsis

`Interval.atanh(a)`

Returns

Return the arc tanh hyperbolic of the Interval number.

Example

```
var z = new Interval(0.3,0.4);  
Interval.atanh(z)            // result [0.30951960420311136, 0.42364893019360217]  
Interval.atanh(Interval(0.5)); // result [0.5493061443340536, 0.5493061443340561]
```

See Also

`Interval.acosh()`, `Interval.asinh()`

Interval.center()

Return the center or midpoint of the Interval number.

Synopsis

Interval Object.center()

Returns

Return the center of the Interval number.

Example

```
var z = new Interval(3,4);  
z.center()           // result 3.5
```

See Also

Interval.width(), Interval.radius()

Interval.cos()

Return the cosine of the Interval number.

Synopsis

Interval.cos(a)

Returns

Return the cosine of the Interval number.

Example

```
var z = new Interval(3,4);  
Interval.cos(z)           // result [-0.9899924966004484,-0.6536436208636089]  
Interval.cos(Interval(0.5)) // result [0.8775825618903724, 0.8775825618903731]
```

See Also

Interval.sin(), Interval.tan()

Interval.cosh()

Return the cosine hyperbolic of the Interval number.

Synopsis

Interval.cosh(a)

Returns

Return the cosine hyperbolic of the Interval number.

Example

```
var z = new Interval(3,4);
Interval.cosh(z)           // result [10.051926281038115, 27.3239685507563]
Interval.cos(Interval(0.5)) // result [1.1276259652063798,1.127625965206382]
```

See Also

Interval.sinh(), Interval.tanh()

Interval.decorationToString()

Return the decoration as a string.

Synopsis

Interval Object.decorationToString()

Returns

A string representation of the Interval decorations.

It can be any of the following:

- "ILL: Ill formed, Not An Interval (NAI)."
- "TRV: Trivial, no Information."
- "DEF: Defined and non Empty."
- "DAC: Defined, Continuous and non Empty."
- "COM: Common, Bounded and non Empty."
- "Unknown Decoration".

Example

```
var z = new Interval( 1,2 );
z.decorationToString(); // "COM: Common, Bounded and non Empty."
z = new Interval(); // Empty Interval
z.decorationToString(); // "ILL: Ill formed, Not An Interval (NAI)."
```

See Also

Interval.intervaldecoration()

Interval.div()

Divide two Interval numbers.

Synopsis

Interval.div(a,b)

Arguments

a,b The interval numbers are to be divided. Particular calculations are made to prevent intermediate results from overflowing.

Returns

The result of the Interval division a/b .

Example

```
var z = new Interval(3,4);
var y=new Interval(1,2);

Interval.div(z,y)            // result [1.5,4]
```

See Also

Interval.add(), Interval.mul(), Interval.sub()

Interval.E

Return Interval exp (1).

Synopsis

Interval.E

Returns

The Interval constant E [2.718281828459045,2.7182818284590455]. Using Interval.E instead of the call Interval.exp(1) is more accurate.

Example

```
var z = Interval.E;
```

See Also

Interval.LN10, Interval.LN2

Interval.equal()

Compare two Interval numbers for equality.

Synopsis

```
Interval.equal(a,b)
```

Arguments

a,b The Interval numbers to be compared for

Returns

The Boolean value of the equal comparison.

Example

```
var z = new Interval(3,4);  
var y=new Interval(1,2);
```

```
if( Interval.equal(z,y))...            // result false  
if( Interval.equal(z,z))...            // result true  
if( !Interval.equal(z,y))...          // result true ! to do a "notequal" comparison
```

See Also

Interval.notequal(), Interval.lessequal(), Interval.less(), Interval.greater(),
Interval.greaterequal()

Interval.exp()

Compute e^x for the Interval number.

Synopsis

Interval.exp(x)

Arguments

x A Interval numbers to be used as the exponent

Returns

e^x , e raised to the power of the specified exponent x , where e is the base of the natural logarithm, with a value of approximately 2.71828.

Example

```
var z = new Interval(3,4);
```

```
Interval.exp(z)...            // result [20.085536923187497,54.598150033144734]  
Interval.exp(Interval(2));   // result [7.389056098930634,7.389056098930683]
```

See Also

Interval.log(), Interval.log10(), Interval.pow()

,

Interval.greater ()

Compare two Interval numbers for greater.

Synopsis

Interval.greater(a,b)

Arguments

a,b The Interval numbers to be compared for

Returns

The Boolean value of the greater comparison.

Example

```
var z = new Interval(3,4);
var y=new Interval(1,2);

if( Interval.greater(z,y))...      // result true
if( Interval.greater(y,z))...      // result false
if( Interval.greater(z,z))...      // result false
```

See Also

Interval.notequal(), Interval.equal(), Interval.lessequal(), Interval.less(),
Interval.greaterequal()

Interval.greaterequal()

Compare two Interval numbers for greater or equal.

Synopsis

Interval.greaterequal(a,b)

Arguments

a,b The Interval numbers to be compared for

Returns

The Boolean value of the greater equal comparison.

Example

```
var z = new Interval(3,4);
var y=new Interval(1,2);

if( Interval.greaterequal(z,y))...      // result true
if( Interval.greaterequal(y,z))...      // result false
if( Interval.greaterequal(z,z))...      // result true
```

See Also

`Interval.notequal()`, `Interval.equal()`, `Interval.less()`, `Interval.greater()`, `Interval.lessequal()`

`Interval.hull()`

`Interval.hull(a, b)` returns the convex hull of two intervals. The convex hull is the smallest interval that contains all values of both operands. Unlike `union()`, the hull is always representable as a single interval, provided neither operand is `NaN`.

Synopsis

`Interval.hull(a,b)`

Arguments

a,b The Interval numbers for which the convex hull is computed. The result:

- Returns the smallest interval enclosing both *a* and *b*
- Preserves correct open or closed endpoint semantics
- Always returns an interval when both operands are valid
- Returns `NaN` if any operand is `NaN`

The hull operation does **not** require the intervals to overlap.

Returns

The convex hull of the two input intervals.

Example

```
var z = new Interval(3,4);
var y = new Interval(1,2);
var x = new Interval(2.5,3.5);
```

```
Interval.hull(z, y) // result [1,4]
Interval.hull(x, z) // result [2.5,4]
Interval.hull(z, z) // result [3,4]
```

Notes

`Interval.hull()` differs from `Interval.union()`:
`union()` is defined only when the result is connected
`hull()` always returns a single enclosing interval
`hull()` is equivalent to:

```
[min(inf(a), inf(b)), max(sup(a), sup(b))]
```

with proper handling of endpoint openness.

See Also

Interval.union(), Interval.intersection()

Interval.in()

Test if the argument is within the interval range and return the Boolean value.

Synopsis

Interval object.in(x)

Returns

Return true if x is within the Interval range; otherwise false.

Example

```
var z = new Interval(3,4);
z.in(2);           // return false
z.in(3);           // returns true
z.in(3.5);         // returns true
```

See Also

Interval.isEmpty()

Interval.inf()

Return the infimum (smallest value) of the Interval's endpoints.

Synopsis

Interval object.inf()

Returns

Return the smallest value of the Interval's endpoint.

Example

```
var z = new Interval(3,4);
z.inf();           // result 3
z = new Interval(4,3);
z.inf();           // result 3
```

See Also

Interval.sup()

Interval.interior()

Return true if a-interval is an interior of the b-interval. Otherwise false.

Synopsis

Interval.interior(a,b)

Arguments

a,b is the interval numbers.

Returns

The Boolean result if *a* is an interior of *b*. An interior is true if $b.low < a.low$ and $a.high < b.high$; otherwise false

Example

```
var z = new Interval(3,4);
var y=new Interval(2,4);
var x=new Interval(3.5,3.9);

Interval.interior(z,y) // result false
Interval.interior(x,z) // result true
Interval.interior(z,z) // result false
```

See Also

Interval.precedes()

Interval.intersection()

Interval is returned Interval.intersection(a, b) returns the intersection of two intervals.

Synopsis

`Interval.intersection(a,b)`

Arguments

a,b The Interval numbers returns the intersection of the two intervals. The result:

- Preserves correct open or closed endpoint semantics
- Returns the empty interval if there is no overlap
- Returns NaN if any operand is NaN

Endpoint openness is handled consistently, not merely by numeric `max` and `min`.

Returns

The result of the Interval intersection.

Example

```
var z = new Interval(3,4);  
var y=new Interval(1,2);  
var x=new Interval(2.5,3.5);
```

```
Interval.intersection(z,y)      // result empty interval []  
Interval.intersection(x,z)     // result [3,3.5]  
Interval.intersection(z,z)     // result [3,4]
```

See Also

`Interval.union()`, `Interval.hull()`

`Interval.intervaldecoration()`

Return or Set the interval decoration for the Interval number.

Synopsis

Interval object.`intervaldecoration(d)`

Arguments

d The optional interval decoration *d* when setting the interval decoration. If omitted, the call returns the actual interval decoration.

Returns

Return the interval decoration of the Interval.

Example

```
var z = new Interval(3,4);
z.lintervaldecoration(); // return the Interval._COM.
z.intervaldecoration(Interval._TRV); // set the intervaldecoration to the Trivial
```

See Also

Interval.intervaltype()

Return or Set the interval type of the Interval number.

Synopsis

Interval object.intervaltype(*t*)

Arguments

t The optional interval type value when setting the interval type. If omitted, the call returns the actual interval type.

Returns

Return the interval type of the Interval number.

Example

```
var z = new Interval(3,4);
z.lintervaltype(); // return the interval type (Interval._CLOSED).
z.intervaltype(Interval._EMPTY); // set the intervaltype to the empty interval
```

See Also

Interval.isEmpty()

Test if the Interval is empty and return the Boolean value.

Synopsis

Interval object.isEmpty()

Returns

Return true if the Interval is empty; otherwise, it is false.

Example

```
var z = new Interval(3,4);
z.isEmpty();           // return false
z= new Interval();
z.isEmpty();           // returns true
z=new Interval(3,"[]");
z.isEmpty();           // returns true since [3] is an empty interval
```

See Also

Interval.in(), Interval.isEntire(), Interval.isPoint(), Interval.isProper(),
Interval.isImproper()

Interval.isEntire()

Test if the Interval spans the entire real number range $[-\infty, +\infty]$ and return the Boolean value.

Synopsis

Interval object.isEntire()

Returns

Return true if the interval span of the entire real number range is false; otherwise, it is false.

Example

```
var z = new Interval(3,4);
z.isEntire();           // return false
z= new Interval(Number.NEGATIVE_INFINITY,Number.POSITIVE_INFINITY);
```

```
z.isEntire(); // returns true
```

See Also

Interval.in(), Interval.isEmpty(), Interval.isPoint(), Interval.isProper(), Interval.isImproper()

Interval.isImproper()

Test if the Interval is improper (left<right) and return the Boolean value.

Synopsis

Interval object.isImproper()

Returns

Return true if the Interval is improper (left<right); otherwise false.

Example

```
var z = new Interval(3,4);
z.isImproper(); // return false
z=new Interval(3);
z.isImproper(); // returns false
z=new Interval(4,3);
z.isImproper(); // returns true since left > right
```

See Also

Interval.in(), Interval.isEntire(), Interval.isPoint(), Interval.isProper(), Interval.isEmpty()

Interval.isNal()

Test if the Interval is flagged as not an interval and return the Boolean value.

Synopsis

Interval object.isNal()

Returns

Return true if the Interval is 'Not an Interval'; otherwise, false.

Example

```
var z = new Interval(3,4);
z.isNaN();           // return false
```

See Also

Interval.in(), Interval.isEntire(), Interval.isPoint(), Interval.isProper(), Interval.isEmpty()

Interval.isPoint()

Test if the Interval is a singleton interval and return the Boolean value.

Synopsis

Interval object.isPoint()

Returns

Return true if the Interval is a single point (singleton interval); otherwise, false.

Example

```
var z = new Interval(3,4);
z.isPoint();           // return false
z = new Interval(3);
z.isPoint();           // returns true
```

See Also

Interval.in(), Interval.isEntire(), Interval.isEmpty(), Interval.isProper(), Interval.isImproper()

Interval.isProper()

Test if the Interval is proper (left<=right) and return the Boolean value.

Synopsis

Interval object.isProper()

Returns

Return true if the Interval is proper (left<= right); otherwise, false.

Example

```
var z = new Interval(3,4);
z.isProper();           // return true
z= new Interval(3);
z.isProper();           // returns true
z=new Interval(4,3);
z.isProper();           // returns false left> right
```

See Also

Interval.in(), Interval.isEntire(), Interval.isPoint(), Interval.isEmpty(),
Interval.isImproper()

Interval.leftinterval()

Return or Set the left endpoint of the Interval number.

Synopsis

Interval object.leftinterval(i)

Arguments

i The optional left value when setting the left endpoint to a new value. If omitted, the call returns the actual left endpoint of the Interval number.

Returns

Return the left endpoint of the Interval number.

Example

```
var z = new Interval(3,4);
z.leftinterval();       // result 3
z.leftinterval(5);     // set left endpoint to 5 and return it. Z is now [5,4]
```

See Also

Interval.rightinterval()

Interval.less ()

Compare two Interval numbers for less.

Synopsis

Interval.less(a,b)

Arguments

a,b The Interval numbers to be compared for

Returns

The Boolean value of the less comparison.

Example

```
var z = new Interval(3,4);
var y=new Interval(1,2);

if( Interval.less(z,y))...           // result false
if( Interval.less(y,z))...           // result true
if( Interval.less(z,z))...           // result false
```

See Also

Interval.notequal(), Interval.equal(), Interval.lessequal(), Interval.greater(),
Interval.greaterequal()

Interval.lessequal()

Compare two Interval numbers for less or equal.

Synopsis

Interval.lessequal(a,b)

Arguments

a,b The Interval numbers to be compared for

Returns

The Boolean value of the less equal comparison.

Example

```
var z = new Interval(3,4);
var y=new Interval(1,2);

if( Interval.lessequal(z,y))... // result false
if( Interval.lessequal(y,z))... // result true
if( Interval.lessequal(z,z))... // result true
```

See Also

Interval.notequal(), Interval.equal(), Interval.less(), Interval.greater(),
Interval.greaterequal()

Interval.LN2

Return Interval log(2) of the natural logarithm.

Synopsis

Interval.LN2

Returns

The Interval constant LN2 [0.6931471805599453,0.6931471805599454]. It is more accurate to use Interval.LN2 instead of the call Interval.log(2)

Example

```
var z = Interval.LN2;
```

See Also

Interval.LN10, Interval.E

Interval.LN10

Return the Interval log(10) of the natural logarithm.

Synopsis

Interval.LN10

Returns

The Interval constant LN10 [2.3025850929940455,2.302585092994046]. It is more accurate to use Interval.LN10 instead of the call Interval.log(10)

Example

```
var z = Interval.LN10;
```

See Also

Interval.LN2, Interval.E

Interval.log()

Compute the natural logarithm of x.

Synopsis

Interval.log(x)

Arguments

x A Interval numbers greater than 0

Returns

Return log(x). If x is 0, then Interval(*-Infinity*) is returned. If x is less than 0, Interval(*NaN*) is returned.

Example

```
var z = new Interval(3,4);

Interval.log(z)...            // result [1.0986122886681071, 1.3862943611198921]
Interval.log(Interval(2));   // result [0.6931471805599435, 0.6931471805599461]
```

See Also

Interval.exp(), Interval.log10(),

Interval.log10()

Compute the base-10 logarithm of x .

Synopsis

Interval.log10(x)

Arguments

x A Interval numbers not equal to zero

Returns

Return $\log_{10}(x)$. If x is 0, then *−Infinity* is returned. If x is less than 0, then *NaN* is returned.

Example

```
var z = new Interval(3,4);

Interval.log10(z)...      // result [0.4771212547196612, 0.6020599913279631]
Interval.log(Interval(2)); // result [0.30102999566398037, 0.30102999566398153]
```

See Also

Interval.exp(), Interval.log(),

Interval.mag()

Return the largest absolute value of the Interval's endpoints.

Synopsis

Interval Objects.mag()

Returns

Return the largest absolute value of the Interval's endpoints.

Example

```
var z = new Interval(3,4);
```

```
z.mag()           // result 4
```

See Also

Interval.mig()

Interval.mig()

Return the smallest absolute value of the Interval's endpoints.

Synopsis

Interval Objects.mig()

Returns

Return the smallest absolute value of the Interval's endpoints.

Example

```
var z = new Interval(3,4);  
z.mig()           // result 3
```

See Also

Interval.mag()

Interval.mul()

Multiply two Interval numbers.

Synopsis

Interval.mul(a,b)

Arguments

a,b The Interval numbers to be multiplied.

Returns

The result of the Interval multiplication.

Example

```
var z = new Interval(3,4);
var y=new Interval(1,2);

Interval.mul(z,y)           // result [3,8]
Interval.mul(Interval(2.5),Interval(0.1)); // result [0.25,0.250000000000000006]
```

See Also

Interval.add(), Interval.div(), Interval.sub()

Interval.neg()

Return the negated Interval number.

Synopsis

Interval object.neg()

Returns

Return the negated Interval number.

Example

```
var z = new Interval(3,4);
z.neg()           // result [-4,-3]
```

See Also

Interval.notequal()

Compare two Interval numbers for inequality.

Synopsis

Interval.notequal(a,b)

Arguments

a,b The Interval numbers to be compared for

Returns

The Boolean value of the notequal comparison.

Example

```
var z = new Interval(3,4);  
var y=new Interval(1,2);
```

```
if( Interval.notequal(z,y))... // result true  
if( Interval.notequal(z,z))... // result false  
if( !Interval.noyequal(z,y))... // result false ! to do a "notequal" comparison
```

See Also

Interval.equal(), Interval.equal(), Interval.less(), Interval.lessequal(), Interval.greater(), Interval.greaterequal()

Interval.one

Return Interval one.

Synopsis

Interval.one

Returns

The Interval constant for one [1].

Example

```
var z = Interval.one; // z=[1,1]
```

See Also

Interval.zero

Interval.PI

Return Interval constant π .

Synopsis

Interval.PI

Returns

The Interval constant PI [3.141592653589793,3.1415926535897936].

Example

```
var z = Interval.PI;
```

See Also

Interval.LN10, Interval.LN2, Interval.E

Interval.pow()

Compute x^y

Synopsis

Interval.pow(x,y)

Arguments

x A Interval numbers to be raised to a power
y A Interval power that x is raised to

Returns

X to the power of y. x^y

Example

```
var x = new Interval(3,4);  
var y = new Interval(1,2);
```

```
Interval.pow(x,y)            // result [2.9999999999999787,16.000000000000167]  
Interval.power(Interval(3),Interval(2)); // [8.999999999999872,9.00000000000011]
```

See Also

Interval.exp()

Interval.precedes()

Return true if a-interval precedes b-interval; otherwise, it is false.

Synopsis

Interval.precedes(a,b)

Arguments

a,b is the interval numbers.

Returns

The Boolean result if *a* precedes *b*. Precedes return true if $a.high < b.low$; otherwise false.

Example

```
var z = new Interval(3,4);
var y=new Interval(1,2);
var x=new Interval(3.5,3.9);

Interval.precedes(z,y) // result false
Interval.precedes(y,z) // result true
Interval.precedes(z,z) // result false
```

See Also

Interval.interior()

Interval.pred()

Compute the previous representable number less than x

Synopsis

Interval.pred(x)

Arguments

x A floating point number.

Returns

The previous representable number is less than x.

Example

```
var z = 1.5
```

```
Interval.pred(z);           // result 1.4999999999999998
```

See Also

`Interval.succ()`

`Interval.radius()`

Return the half-width (radius) of the Interval number.

Synopsis

Interval Object.radius()

Returns

Return the half-width (radius) of the Interval number.

Example

```
var z = new Interval(3,4);  
z.radius(z)           // result 0.5
```

See Also

`Interval.center()`, `Interval.width()`

`Interval.rightinterval()`

Return or set the right endpoint for the interval number.

Synopsis

Interval object.rightinterval(*r*)

Arguments

r The optional upper value when setting the right endpoint to a new value. If omitted, the call returns the actual right endpoint of the Interval number.

Returns

Return the right endpoint of the Interval number.

Example

```
var z = new Interval(3,4);
z.rightinterval();           // result 4
z.rightinterval(5);         //set the right endpoint to 5 and return the result 5
```

See Also

Interval.leftinterval()

Interval.sin()

Return the sine of the Interval number.

Synopsis

Interval.sin(a)

Returns

Return the sine of the Interval number.

Example

```
var z = new Interval(3,4);
Interval.sin(z)           // result [-0.7568024953079279, 0.1411200080598619]
Interval.sin(Interval(2)); // result [0.9092974268256793, 0.909297426825684]
```

See Also

Interval.cos(), Interval.tan()

Interval.sinh()

Return the hyperbolic sine of the interval number.

Synopsis

```
Interval.sinh(a)
```

Returns

Return the sine hyperbolic of the Interval number.

Example

```
var z = new Interval(3,4);
Interval.sinh(z)           // result [10.017874927409816, 27.289917197128002]
Interval.sinh(Interval(2)); // result [3.6268604078470106, 3.626860407847036]
```

See Also

Interval.cosh(), Interval.tanh()

Interval.sub()

Subtract two Interval numbers.

Synopsis

```
Interval.sub(a,b)
```

Arguments

a,b The interval numbers are to be subtracted.

Returns

The result of the Interval subtraction.

Example

```
var z = new Interval(3,4);
var y=new Interval(1,2);

Interval.sub(z,y)           // result [1,3]
```

See Also

Interval.add(), Interval.div(), Interval.mul()

Interval.sup()

Return the supremum (largest value) of the Interval's endpoints.

Synopsis

Interval object.sup()

Returns

Return the largest value of the Interval's endpoint.

Example

```
var z = new Interval(3,4);
z.sup();           // result 4
z = new Interval(4,3);
z.sup();           // result 4
```

See Also

Interval.inf()

Interval.SQRT2()

Compute an Interval of square root of 2.

Synopsis

Interval.SQRT2()

Returns

The square root of 2.

Example

```
var z = Interval.SQRT2; // result [1.4142135623730947,1.4142135623730951)
```

See Also

Interval.sqrt()

Interval.sqr()

Compute the Interval square.

Synopsis

Interval.sqr(x)

Arguments

x A Interval numbers to be squared. Particular calculations are made to prevent intermediate results from overflowing.

Returns

The square of *x*. Notice that it provides tighter bounds for negative values than just multiplying *x* with itself.

Example

```
var z = new Interval(-2,1);  
Interval.sqr(z)...            // result [4, 1]
```

See Also

Interval.sqrt()

Compute an Interval square root.

Synopsis

Interval.sqrt(x)

Arguments

x A Interval numbers are to be square-rooted. Particular calculations are made to prevent intermediate results from overflowing.

Returns

The square root of x . if $x < 0$ then it return `Interval(NaN)`.

Example

```
var z = new Interval(3,4);
```

```
Interval.sqrt(z)...           // result [1.7320508075688774, 2]  
Interval.sqrt(Interval(2)); // result [1.414213562373095, 1.4142135623730951]
```

See Also

`Interval.SQRT2()`

`Interval.succ()`

Compute the following representable number bigger than x

Synopsis

`Interval.succ(x)`

Arguments

x A floating point number.

Returns

The following representable number that is bigger than x .

Example

```
var z = 1.5
```

```
Interval.succ(z);           // result 1.5000000000000002
```

See Also

`Interval.pred()`

`Interval.tan()`

Return the tangent of the Interval number.

Synopsis

Interval.tan(a)

Returns

Return the tangent of the Interval number. If $a = \pi/2$, then it returns Interval(*Infinity*). If $a = 3\pi/2$, then it returns Interval(*-Infinity*).

Example

```
var z = new Interval(3,4);
Interval.tan(z)           // result [-0.14254654307428194, 1.15782128234958]
Interval.tan(Interval(2)); // result [-2.185039863261552, -2.1850398632614847]
```

See Also

Interval.cos(), Interval.sin()

Interval.tanh()

Return the tangent hyperbolic of the Interval number.

Synopsis

Interval.tanh(a)

Returns

Return the tanh hyperbolic of the Interval number.

Example

```
var z = new Interval(3,4);
Interval.tanh(z)           // result [0.13495454840125132, 7.368313124558387]
Interval.tanh(interval(2)); // result [0.9640275800758038, 0.96402758007583]
```

See Also

Interval.cosh(), Interval.sinh()

Interval.toClose()

Convert the Interval to a close form of an interval.

Synopsis

Interval.toClose(a)

Returns

Return the Close interval form of a. If a is already in a closed form, then a is returned. If a is half open on the left side, then the closed form of [Interval.succ(a.inf()), a.sup()] is returned. If a is half open on the right side, then the closed form of [a.inf(), Interval.pred(a.sup())] is returned, and if in open form, then [Interval.succ(a.inf()), Interval.pred(a.sup())] is returned.

Example

```
var z = new Interval(1.5,2,"[]");
var x;
x=Interval.toClose(z)      // result [1.5000000000000002, 2]
Interval.toClose(x);      // result [1.5000000000000002, 2] or unchanged
                          // since it is already in closed form
```

See Also

Interval.toOpen()

Interval.toExponential()

Format a number using exponential notation.

Synopsis

Interval.toExponential(digits)

Arguments

Digits The number of digits that will appear after the decimal point. This may be a value between 0 and 20, inclusive. If this argument is omitted, as many digits as necessary will be used. An Interval number is always formatted as:

[left_part , right_part]

The left or right bracket can be rounded, representing an open or half-open interval.

Returns

A string representation of the Interval number, in exponential notation, with one digit before the decimal place and *digits* digits after the decimal place. The fractional part of the Interval number is rounded or padded with zeros, as necessary, so that it has the specified length.

Example

```
var z = new Interval( 12345.6789, 12345.6789 );
z.toExponential(1);      // result [1.2e+4,1.2e+4]
z.toExponential(5);     // result [1.23457e+4,1.23457e+4]
z.toExponential(10);    // result [1.23456789000e+4,1.23456789000e+4]
z.toExponential();      // result [1.23456789e+4,1.23456789e+4]
```

See Also

Interval.toFixed(), Interval.toPrecision(), Interval.toString()

Interval.toFixed()

Format a number using fixed-point notation.

Synopsis

Interval.toFixed(*digits*)

Arguments

Digits The number of digits that will appear after the decimal point. This may be a value between 0 and 20, inclusive. If this argument is omitted, it is treated as zero. An Interval number is always formatted as:

[left_part , right_part]

The left or right bracket can be rounded, representing an open or half-open interval.

Returns

A string representation of the interval number that does not use exponential notation and has exactly *digits* after the decimal point. The *Interval number* is rounded as necessary, and the fraction part is padded with zeros if required to have the specified length. If the *Interval number* is greater than 1e+21, this method calls *number.toString()* and returns a string in exponential notation.

Example

```
var z = new Interval( 12345.6789, 12345.6789,"[]" );
z.toFixed(5);           // result [12345.7,12345.7]
z.toFixed(6);           // result [12345.678900,12345.678900]
z.toFixed();            // result [12346,1234.6]
```

See Also

Interval.toExponential(), Interval.toPrecision(), Interval.toString()

Interval.toOpen()

Convert the Interval to an open form of an interval.

Synopsis

Interval.toOpen(a)

Returns

Return the Open interval form of a . If a is already in an open form, then a is returned. If a is half open on the left side, then the open form of $(a.inf(), Interval.pred(a.sup()))$ is returned. If a is half open on the right side, then the open form of $(Interval.pred(a.inf()), a.sup())$ is returned, and if in open form, then $(Interval.pred(a.inf()), Interval.succ(a.sup()))$ is returned.

Example

```
var z = new Interval(1.5,1.5,"[]");
var x;
x=Interval.toOpen(z)           // result (1.4999999999999998,1.5)
Interval.toOpen(x);           // result (1.4999999999999998,1.5) or unchanged
                               // since it is already in an open form
```

See Also

Interval.toClose()

Interval.toPrecision()

Format the significant digits of an Interval number.

Synopsis

Interval.toPrecision(digits)

Arguments

Digits The number of significant digits to appear in the returned string. This may be a value between 1 and 21, inclusive. If this argument is omitted, the `toString()` method is used instead to convert the Interval number to a base-10 value. An Interval number is always formatted as:

[left_part , right_part]

The left or right bracket can be rounded, representing an open or half-open interval.

Returns

A string representation of the *Interval number*, which contains *precisions* significant digits. If *precision* is large enough to include all the digits of the integer part of the number, the returned string uses fixed-point notation. Otherwise, exponential notation is used with one digit before the decimal place and *precision* – 1 digits after the decimal place. The number is rounded or padded with zeros as necessary.

Example

```
var z = new Interval( 12345.6789, 12345.6789 );
z.toPrecision(1);                    // result [1e+4,1e+4]
z.toPrecision(3);                   // result [1.23e+4,1.2e+4]
z.toPrecision(5);                   // result [12346,12346]
```

See Also

`Interval.toExponential()`, `Interval.toFixed()`, `Interval.toString()`

`Interval.toString()`

Format the significant digits of an Interval number.

Synopsis

Interval.toString(radix)

Arguments

Radix If omitted, the base ten will be used to convert the Interval number to a string. Otherwise, the radix will be used (2..36). An Interval number is always formatted as:

[left_part , right_part]

The left or right bracket can be rounded, representing an open or half-open interval.

Returns

A string representation of the *Interval number*, in the indicated radix.

Example

```
var z = new Interval( 12345.6789, 12345.6789 );
z.toString();           // result [1234.6789,1234.6789]
```

See Also

Interval.toExponential(), Interval.toFixed(), Interval.toPrecision()

Interval.union()

Create the union of two Interval numbers.

Synopsis

Interval.union(a,b)

Arguments

a,b The Interval numbers to be unionized or together. It creates an Interval from the lowest to the highest range. It is conceptually similar to or'ring two intervals.

Returns

The result of the Interval union.

Example

```
var z = new Interval(3,4);
var y=new Interval(1,2);
```

```
Interval.union(z,y) // result [1,4]
```

See Also

Interval.intersection(), Interval.hull()

Interval.valueOf()

Return the primitive number value.

Synopsis

Interval object.valueOf()

Returns

The primitive value of the *Interval number* is returned, the same as the *Interval number*.leftinterval().

Example

```
var z = new Interval(3,4);  
z.valueOf() // return 3
```

See Also

Interval.width()

Return the width of the Interval number.

Synopsis

Interval.width(a)

Returns

Return the width of the Interval number a.

Example

```
var z = new Interval(3,4);  
Interval.width(z) // result 1
```

See Also

`Interval.center()`, `Interval.radius()`

`Interval.zero`

Return Interval zero.

Synopsis

`Interval.zero`

Returns

The Interval constant zero `[0,0]`.

Example

```
var z = Interval.zero;
```

See Also

`Interval.one`

`parseInterval()`

Convert a string to an Interval number.

Synopsis

parseInterval(s)

Arguments

s The string to be parsed and converted to an *Interval number*.

Returns

`parseInterval()` parses and returns a new Interval number contained in *s*. `parseInterval()` returns an Interval NaN number if parsing fails. An Interval number can either be in the format:

[left_part , right_part]

Where either the left or right part can be missing, even simultaneously (Empty Interval), `parseInterval()` can also parse a string by omitting the leading and trailing parentheses or square brackets. Parsing decimal input

The function `parseInterval(string)` parses numeric and interval strings into interval objects. When parsing a decimal number such as "0.1":

- The value is treated as an exact real decimal
- The result is a tight enclosing interval, not a point interval
- This accounts for the fact that many decimals do not have exact binary representations

For example:

```
parseInterval("0.1")
```

returns a small interval enclosing the exact value 0.1.

Invalid input strings produce `NaN`.

Example

```
var z = parseInterval( "(1.2,3.4E-5)" ); // result (1.2,3.4E-5) Open
z = parseInterval( "[1.2]" ); // result [1.2 ,1.2] Closed
z = parseInterval( "(1,2]" ); // result (1 , 2] Half open
z = parseInterval( "[]" ); // result the Empty interval
```

See Also

Example

This is a simplified version of the `Interval.log()` function. Exponent handling and argument reduction have been removed to simplify the example.

```
Interval.log=function(t)
  {var low,high;
  function intervallog(x)
    {var zn,zsq,i,k,sum,delta;
    if(x<0) {return new Interval(NaN,NaN);}
    if(x==0) {return new Interval(-Infinity,-Infinity);}
    if(x==1) {return new Interval(0);}
    zn=Interval(x);
    // Taylor series of log(x)
    // log(x)=2( z + z^3/3 + z^5/5 ... )
    // where z=(x-1)/(x+1)
    // Initialize the iteration
    zn=Interval.div(Interval.sub(zn,Interval.one),Interval.add(zn,Interval.one));
    zsq=Interval.mul(zn,zn); sum=zn;
    // Iterate using Taylor series log(x) == 2( z + z^3/3 + z^5/5 ... )
    for(i=3;;i+=2)
      {
      zn=Interval.mul(zn,zsq);
      delta=Interval.div(zn,Interval(i));
      if(sum.center()+delta.center()==sum.center()) break;
      sum=Interval.add(sum,delta);
      }
    sum=Interval.mul(sum,Interval(2));
    return sum;
  }
  if(t.isPoint()) return intervallog(t.inf());
  low=intervallog(t.inf()).inf(); high=intervallog(t.sup()).sup();
  return new Interval(low,high);
}
```