Stable Numerics Subroutine Library

Jan Adelsbach

July 9, 2026

Contents

1 About this Guide

1.1 Legal Information

1.2 Feedback and Contact

For feedback on this document, please use the following email address:
techpubs@adelsbach-research.eu

Please include the page number or a link to the page.

For general contact details, please visit https://adelsbach-research.eu/contact.

2 Overview

2.1 Introduction

The *Adelsbach Stable Numerics Subroutine Library* is a function library for integer, real and complex interval arithmetic written in C++.

2.2 Thread Safety

All routines in the library are globally thread safe, however they are not safe against multiple threads calling non-const routines on the same object instance.

3 Mathematical Definitions

3.1 Real Interval Arithmetic $\mathbb{IR}$

Interval arithmetic consists of a tuple of two numbers representing an infinimum and supremum of an interval. By definition an interval is in $\mathbb{R}^2$ or $\mathbb{Z}^2$ we will however for distinction represent interval numbers as $\mathbb{IR}$ (interval real space) and $\mathbb{IZ}$ (interval integer space) as they are used in a distinct manner differing from standard $\mathbb{R}^2$ or $\mathbb{Z}^2$ arithmetic. The definition of an interval is:

$$\mathbb{IR} = \{[\underline{x}, \overline{x}] | \underline{x}, \overline{x} \in \mathbb{R}, \underline{x} \leq \overline{x}\}$$

The two elements in an interval $x \in \mathbb{IR}$ or $\mathbb{Z}^2$ are called *infinimum* $\underline{x}$ and a *supremum* $\overline{x}$ where for a normalized interval $\underline{x} \leq \overline{x}$. The two latter are thereby representing the respective infinimum and supremum of an interval hence by definition an interval $X \in \mathbb{IR}$ actually defines a set of values:

$$X = \{\forall a \in \mathbb{R} : \underline{x} \leq a \leq \overline{x}\}$$

This makes it distinct from $\mathbb{R}^2$ where a number in the latter would correspond to a fixed defined point.

3.1.1 Singleton Interval

The map for a real number $r \in \mathbb{R}$ in interval space, $\mathbb{X} \longrightarrow \mathbb{IR}$ would hence correspond to

$$X = \{r \in \mathbb{R} : \underline{x} = \overline{x} = r\} = [r, r]$$

This is called a *singleton* interval, where $\underline{r} = \overline{r}$. The reverse map on a singleton interval is defined as being $r = \underline{x} = \overline{x}$. For non-singleton intervals the map $\mathbb{IR} \longrightarrow \mathbb{R}$ is generally defined as the midpoint between the interval limits, however multiple further definitions exist - see the chapter on *Interval Intrinsic Functions* for a listing on the supported methods.

3.1.2 Empty Interval

An empty interval $X = \varnothing$ for $X \in \mathbb{IR}$ is defined as a singleton where

$$X = \{\underline{x} = \overline{x} = \varnothing\} = \varnothing$$

where $\varnothing \notin \mathbb{R}$. Hence no arithmetic can be performed on an empty interval. On a computer $\varnothing$ is represented by `qNaNs` (Quiet-Not-a-Number) hence causing any arithmetic to yield further `qNaNs`.

3.1.3 Sign of an Interval

The sign of an interval is only deductible if both infinimum and supremum are of the same sign, this can be represented by the following: Formally for an interval $X \in \mathbb{IR}$ that is:

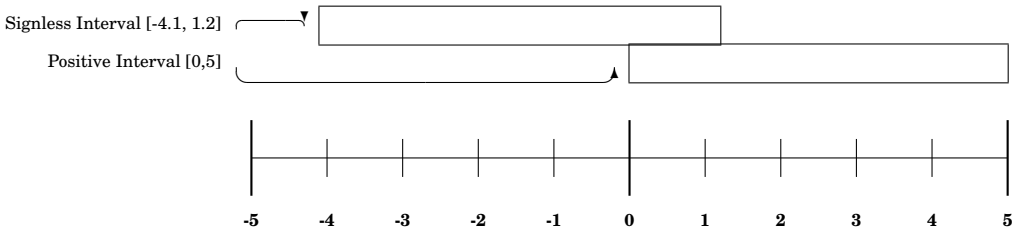

Figure 1: Example of an unsigned and signed interval

$$\text{sgn(X)} = \text{sgn}(\underline{\text{x}}) \Leftrightarrow \text{sgn}(\underline{\text{x}}) = \text{sgn}(\overline{\text{x}})$$

Otherwise if the sign of the infinimum and supremum are not equal, the interval X is considered *signless*.

3.1.4 Bounded and Whole Interval

An interval $X \in \mathbb{IR}$ is considered to be *bounded* if $\overline{x}, \underline{x} \in \mathbb{R}$ and are finite. If either but not both of $\underline{x}$ or $\overline{x}$ are $\pm\infty$ with the respective sign the interval is considered to be a *lower bounded* or *upper unbounded* interval.

If $\underline{x} = -\infty$ and $\overline{x} = \infty$ the interval spans the whole $x \in \mathbb{R}$ space, this is called a *whole* interval.

A special case is $\underline{x} = \overline{x} = \infty$ which is by definition an empty interval, as there will be no elements in the resulting interval set.

3.1.5 Normalized Interval

An interval $X \in \mathbb{IR}$ is considered to be *normalized* if either $\underline{x} \leq \overline{x}$ or a singleton $\underline{x} = \overline{x} = \varnothing$ otherwise the interval is considered to be *denormalized* in the first case, in the second case if one of but not both of $\underline{x}$ or $\overline{x}$ are $\varnothing$ the interval is considered *improper*. Singleton intervals are always considered *normalized*.

3.2 Circular Complex Interval Arithmetic $\mathbb{KC}$

Circular complex arithmetic $\mathbb{KC}$ consists of a complex number $\mathbb{C}$, acting as a center point and a real number $\mathbb{R}$ acting as a radius of a circle in a two dimensional Cartesian space. As such any number that is within radius of the pivot is contained in the interval. The mathematical definition a complex circular interval $X \in \mathbb{KC}$ is as follows:

$$X = \{a \in \mathbb{C} : (\Re(a) - \Re(x_p))^2 + (\Im(a) - \Im(x_p))^2 - k_r^2 \leq 0\}$$

Where $x_r \in \{r \in \mathbb{R} : r \geq 0\} \equiv x_r \in \mathbb{R}^+$ as the radius of the circle and $x_p \in \mathbb{C}$ as the center point in a two dimensional Cartesian coordinate system.

3.2.1 Singleton Interval

In $\mathbb{KC}$ arithmetic a singleton is an interval, where the radius $k_r = 0$ and as such the interval only refers to a single complex number.

3.2.2 Empty Interval

An empty interval $\varnothing$ in $X \in \mathbb{KC}$ arithmetic is represented by the pivot $x_p \in \mathbb{C}$ or radius $x_r \in \mathbb{R}$ being a value $r \notin \mathbb{R}$ such that no arithmetic can be performed.

3.2.3 Whole Interval

In $X \in \mathbb{KC}$ arithmetic a whole interval contains the full complex space, such that a is $\forall a \in \mathbb{C}$. This is represented by the radius being $k_r = \infty$.

3.2.4 Normalized Interval

A circular complex interval $X \in \mathbb{KC}$ is considered *normalized* if

$$(x_p \in \mathbb{C}) \wedge (x_r \geq 0)$$

3.3 Rectangular Complex Interval Arithmetic $\mathbb{RC}$

Rectangular complex arithmetic $\mathbb{RC}$ consists of two intervals $\mathbb{IR}$ representing the real $\Re$ and imaginary $\Im$ component of a complex number $\mathbb{C}$. Geometrically a rectangular complex interval is a box in Cartesian two-dimensional space. Given a rectangular complex interval $k \in \mathbb{RC}$ where the extraction operators $\Re_i(k), \Im_i(k) \in \mathbb{IR}$ are used to extract the real and imaginary components a rectangular complex consists of a set of all complex numbers $v \in \mathbb{C}$ that uphold:

$$\mathbb{RC} = \{(\underline{\Re_i(k)} \leq \Re(v) \leq \overline{\Re_i(k)} \wedge (\underline{\Im_i(k)} \leq \Im(v) \leq \overline{\Im_i(k)}\}$$

3.3.1 Singleton Interval

In $\mathbb{RC}$ arithmetic a singleton interval $k \in \mathbb{RC}$ is defined where the infinimum and supremum of both the real and imaginary parts are equal, that is where:

$$\underline{\Re_i(k)} = \overline{\Re_i(k)} \wedge \underline{\Im_i(k)} = \overline{\Im_i(k)}$$

3.3.2 Empty Interval

In $\mathbb{RC}$ arithmetic an empty interval has both the real and imaginary sub-intervals as empty intervals, that is:

$$\Re_i(k) = \varnothing \wedge \Im_i(k) = \varnothing$$

3.3.3 Whole Interval

In $\mathbb{RC}$ arithmetic a whole interval has both the real and imaginary sub-intervals as whole intervals, that is:

$$\underline{\Re_i(k)} = \underline{\Im_i(k)} = -\infty \wedge \overline{\Re_i(k)} = \overline{\Im_i(k)} = \infty$$

3.4 Kaucher Interval Arithmetic $\mathbb{KR}$

Generalized interval arithmetic, also known as *Kaucher Interval Arithmetic* ($\mathbb{KR}$), extends the classical interval arithmetic framework ($\mathbb{IR}$) by relaxing the ordering constraint between the bounds. In $\mathbb{KR}$, the *infinimum* $\underline{x}$ and *supremum* $\overline{x}$ are not required to satisfy $\underline{x} \leq \overline{x}$. This generalization allows for the representation of intervals where the bounds may be reversed, enabling a broader class of mathematical operations and applications.

The set of Kaucher intervals is formally defined as:

$$\mathbb{KR} = \{[\underline{x}, \overline{x}] \mid \underline{x}, \overline{x} \in \mathbb{R}\}.$$

Unlike classical intervals, Kaucher intervals can model directed quantities, such as oriented distances or differences, where the sign of the interval carries meaningful information.

3.4.1 Empty Interval

An empty Kaucher interval $X = \varnothing$ for $X \in \mathbb{KR}$ is defined as an interval where at least one of the bounds is undefined:

$$X = \{[\underline{x}, \overline{x}] \mid \underline{x} = \varnothing \vee \overline{x} = \varnothing\} = \varnothing.$$

Here, $\varnothing \notin \mathbb{R}$, meaning the empty interval is not a real-valued interval. Consequently, no arithmetic operations can be performed on an empty interval.

In computational implementations, the empty interval is typically represented using `qNaN` (Quiet-Not-a-Number) values. Any arithmetic operation involving a `qNaN` will propagate the `qNaN`, ensuring that invalid or undefined operations do not yield misleading results.

3.5 Real Triplex Arithmetic $\mathbb{TR}$

Triplex arithmetic $\mathbb{TR}$ extends the concept of interval arithmetic $\mathbb{IR}$ by incorporating not only the lower bound $\underline{x}$ and upper bound $\overline{x}$, but also an explicit midpoint value x. This midpoint represents the default floating-point value obtained through standard nearest rounding, thereby capturing the central tendency of the interval.

For any triplex number $X \in \mathbb{TR}$, the relationship between its components is defined as:

$$\underline{x} \leq x \leq \overline{x} \quad \forall X \in \mathbb{TR}.$$

This structure allows for a more precise representation of uncertainty or variability, as it explicitly includes the most likely or rounded value alongside the bounds.

3.5.1 Empty Triplex

An empty triplex value $T = \varnothing$ for $T \in \mathbb{TR}$ is defined as a degenerate case where all components are undefined or null:

$$T = \{[\underline{x}, x, \overline{x}] \mid \underline{x} = x = \overline{x} = \varnothing\} = \varnothing.$$

This represents the absence of a valid triplex value, analogous to the empty set in interval arithmetic.

In computational implementations, the empty triplex is typically represented using `qNaN` (Quiet-Not-a-Number) values. Any arithmetic operation involving a `qNaN` will propagate the `qNaN`, ensuring that invalid or undefined operations do not yield misleading results.

3.6 Pair Arithmetic

Pair arithmetic is a numerical representation method designed to extend the precision of floating-point computations by splitting a value into a *main part* and a *remainder part*. This approach mitigates rounding errors that arise in standard floating-point arithmetic, particularly for operations involving numbers of vastly different magnitudes.

A *pair* $\langle x, y \rangle$ represents a real number z as the sum of two floating-point values:

$$z = x + y,$$

where:

- x is the *main value*, a floating-point number capturing the significant digits of z.
- y is the *remainder*, a floating-point number representing the residual part of z that is too small to be accurately represented in x alone.

The pair $\langle x, y \rangle$ satisfies the condition:

$$|y| \leq \frac{1}{2}\text{ulp}(x),$$

where $\text{ulp}(x)$ (unit in the last place) denotes the spacing between x and the next representable floating-point value. This ensures y is negligible relative to x but still contributes to the overall precision of z.

3.6.1 Not-A-Number Pair

A pair is considered a *Not-a-Number* (`NaN`) if either of its components, x or y, is `NaN`. This aligns with the IEEE 754 standard for floating-point arithmetic, where any operation involving `NaN` propagates the `NaN` value. Specifically:

$$\langle x, y \rangle = \texttt{NaN} \quad \text{if} \quad x = \texttt{NaN} \vee y = \texttt{NaN}.$$

In such cases, the pair cannot represent a valid numerical value, and any arithmetic operation involving a `NaN` pair will result in another `NaN` pair.

4 String Representation

4.1 String representation of $\mathbb{IR}$

For intervals in the $\mathbb{IR}$ space there are two representation variants, for proper intervals and for singleton intervals.

Singleton Intervals
For a singleton interval, where $\underline{x} = \overline{x}$ a single value is represented in brackets. Examples include `[8.3]`, `[8]`.

Normal Intervals
For normal intervals, the infinimum and supremum are written in brackets, separated by a comma. Examples include `[8.3,12.2]`, `[8,12.6]`.

Empty Intervals
Empty intervals can be specified as both a singleton or a normal interval with instead of numbers the string `nan`. Examples include: `[nan]` and `[nan,nan]`.

Whole and Unbound Intervals
Whole or unbound intervals can be specified with the corresponding element matching the string `inf`. Examples include: `[-inf, 3]` and `[-inf,inf]`. Note that `[inf]` will result in an empty interval.

4.2 String representation of $\mathbb{KC}$

For intervals in the $\mathbb{KC}$ space there are two representation variants, for proper intervals and for singleton intervals.

Singleton Intervals
For a singleton interval, where $\underline{x} = \overline{x}$ a single value is represented in brackets or with the radius given as 0 separated by a comma. Examples include `[(8,4),0]`, `[(8,4)]`.

Normal Intervals
For normal intervals, the complex number and it's radius are given in backets separated by a comma. Examples include `[(8,4),12.2]`, `[(8,4),5]`.

4.3 String representation of $\mathbb{RC}$

Intervals of $\mathbb{RC}$ are represented as two intervals $\mathbb{IR}$ for the interval of the real and imaginary parts. See **??** for $\mathbb{IR}$ string representation details.

4.4 String representation of $\mathbb{KR}$

Kaucher interval arithmetic uses the same string definitions as for real intervals. Please refer to **??**.

4.5 String representation of $\mathbb{TR}$

Triplex arithmetic uses a similar string defintion as normal intervals, see **??**, but extended to three values. These are the infinimum, value and supremum.

Singleton Triplexes
For a singleton triplex, where $\underline{x} = x = \overline{x}$ a single value is represented in brackets. Examples include `[8.3]`, `[8]`.

Normal Triplexes
For normal triplexes, the infinimum, value and supremum are written in brackets, separated by a comma. Examples include `[8.3,10.1,12.2]`.

Empty Triplexes
Empty intervals can be specified as both a singleton or a normal triplex with instead of numbers the string `nan`. Examples include: `[nan]` and `[nan,nan,nan]`.

5 Real Intervals $\mathbb{IR}$

5.1 General Description

For the C++ programming language family the package is mainly implemented as a templated class, with the exception of comparison functions which are implemented as procedural function declarations.

5.1.1 Header Files

All function prototypes and data structures are declared in the `interval.hpp` header file, which in itself includes a couple further header files that contain all subsystems of the interval package.

5.2 `interval<`*T*`>` - Class Reference

The `interval<`*T*`>` class represents an interval number of the given template type *T*. The class checks that the latter type *T* matches one of the following types by a `static_assert`, this is to ensure that the class only gets instantiated for types that are actually implemented. The following types are supported:

- `float`
- `double`
- `long double`

5.2.1 Public Member Variables

`inf`
: Infinimum of the interval $\underline{x}$.

`sup`
: Supremum of the interval $\overline{x}$.

5.2.2 Constructors

```
constexpr interval() = default;
constexpr interval(const T& a, const T& b);
constexpr interval(const T a[2]);
constexpr interval(const T v);
constexpr interval(const interval<T>& o) = default;
constexpr interval(const std::pair<T,T>& p);
```

These are the constructors of the `interval<T>` class. All constructors with the `constexpr` declaration specifier are implemented in inline.

`interval()`
: Default constructor, leaves infinimum and supremum uninitialized.

`interval(const interval<T>&)`
: Default copy constructor.

`interval(const T&, const T&)` **and** `interval(const T v[2])`
: Assignment constructor, assigns infinimum and supremum to the first and second argument respectively.

`interval(const T&)`
: Assignment constructor, initializes the interval as a singleton of the given value.

`interval(std::pair<T,T> &)`
: Assignment constructor, initializes the infinimum and supremum to the `first` and `second` value of the `std::pair`, respectively.

5.2.3 Assignment

```
constexpr interval<T>& operator=(const interval<T>&) = default;
interval<T>& operator=(const std::pair<T,T>& o);

interval<T>& set_midpoint(const T& mid, const T& error);
interval<T>& set_range(const T& i, const T& s);
interval<T>& set_range(const std::pair<T,T>& p);
```

These member functions assign the infinimum and supremum of the interval.

`operator=(const interval<T>&)`
: Standard one-to-one assignment.

`operator=(const std::pair<T,T>&)` **and** `set_range(const std::pair<T,T>& p)`
: Assigns the infinimum to the first element, and the supremum to the second element of the `std::pair`.

`set_midpoint(const T& mid, const T& error)`
: Assign the infinimum to `mid-|error|` the supremum to `mid+|error|`.

`set_range(const T& i, const T& s)`
: Assign the infinimum and supremum from the given first and second argument, respectively.

5.2.4 Special Assignment

```
interval<T>& set_whole();
interval<T>& set_empty();
interval<T>& set_lunbound();
interval<T>& set_runbound();
```

These assign special values to the given interval object.

`set_whole()`
: Set the current interval $A \in \mathbb{IR}$ to an unbounded state $\underline{a} = -\infty$ and $\overline{a} = \infty$. Returns a reference to the current object.

`set_empty()`
: Set the current interval $A \in \mathbb{IR}$ to $A = \varnothing$. Returns a reference to the current object.

`set_lunbound()`
: Set the infinimum of the current interval as unbounded. Returns a reference to the current object.

`set_runbound()`
: Set the supremum of the current interval as unbounded. Returns a reference to the current object.

5.2.5 Comparison operators

```
bool operator==(const interval<T>& o) const;
bool operator!=(const interval<T>& o) const;
bool operator==(const std::pair<T,T>& o) const;
bool operator!=(const std::pair<T,T>& o) const;
```

These two operators check the equality or inequality of the infinimum and supremum element-wise. Let $T, O \in \mathbb{IR}$ with T being the current instance, `this` and O being the given other interval to compare to, that is `o`:

`operator==(const interval<T>&)` **and** `operator==(const std::pair<T,T>&)`
: Set equality operation, as defined as $(\underline{t} = \underline{o}) \wedge (\overline{t} = \overline{o})$. With either an other interval or a `std::pair` whose first element is interpreted as $\underline{o}$ and second element $\overline{o}$.

`operator!=(const interval<T>&)` **and** `operator!=(const std::pair<T,T>&)`
: Set inequality operation, as defined as $(\underline{t} \neq \underline{o}) \vee (\overline{t} \neq \overline{o})$. With either an other interval or a `std::pair` whose first element is interpreted as $\underline{o}$ and second element $\overline{o}$.

5.2.6 Conversion functions

```
std::pair<T,T> as_pair() const;
```

These member functions convert the current instance of the `interval<T>` class into other formats.

`as_pair()`
: Convert the current interval to a `std::pair` with the `first` and `second` members set to the infinimum and supremum, respectively.

5.2.7 Arithmetic Operators

```
interval<T> operator+(const interval<T>& o) const;
interval<T> operator-(const interval<T>& o) const;
interval<T> operator*(const interval<T>& o) const;
interval<T> operator/(const interval<T>& o) const;
interval<T> operator-() const;
interval<T> operator+() const;

interval<T>& operator+=(const interval<T>& o);
interval<T>& operator-=(const interval<T>& o);
interval<T>& operator*=(const interval<T>& o);
interval<T>& operator/=(const interval<T>& o);

interval<T> operator+(const T& v) const;
interval<T> operator-(const T& v) const;
interval<T> operator*(const T& v) const;
interval<T> operator/(const T& v) const;

interval<T>& operator+(const T& v);
interval<T>& operator-(const T& v);
interval<T>& operator*(const T& v);
interval<T>& operator/(const T& v);
```

These operators execute the respective equivalent C++ standard arithmetic operation on the `interval<T>`.

5.2.8 Status functions

```
bool is_empty() const;
bool is_whole() const;
bool is_signed() const;
bool is_bounded() const;
bool is_singleton() const;
bool is_okay() const;

bool has_zero() const;
```

These member functions inquire the current status of the interval object.

`is_empty()`
: Whether the interval is empty, that is whether $A \in \mathbb{IR}$ is $\underline{a} = \overline{a} = \varnothing$.

`is_whole()`
: Whether the interval spans the whole $\mathbb{R}$, that is whether for $A \in \mathbb{IR}$ is $\underline{a} = -\infty \wedge \overline{a} = \infty$.

`is_signed()`
: Whether the interval contains a sign, that is for $A \in \mathbb{IR}$ defined as $\text{sgn}(\underline{a}) = \text{sgn}(\overline{a})$. It is of note that for this definition it is considered that $-0 \neq 0$.

`is_singleton()`
: Whether the interval represents a single value in $\mathbb{R}$, that is for $A \in \mathbb{IR}$ defined as $\underline{a} = \overline{a} \Leftrightarrow \underline{a}, \overline{a} \in \mathbb{R}$.

`is_okay()`
: Determines whether the interval is normalized, if `false` the interval must be normalized with the respective functions.

`has_zero()`
: Determines whether the interval contains a zero, i.e. whether for a given interval $A \in \mathbb{IR}$ it is true that $\pm 0 \in A$, where this function will return `true` if either or both of -0 and 0 are in A.

5.2.9 Sign

```
int sign() const;
```

This function returns the sign of the interval, if the latter exists. The following return values may be occurred:

1
: Positive sign.

0
: A sign cannot be determined.

-1
: Negative sign.

5.2.10 Normalization

```
interval<T> normalized() const;
interval<T>& normalize();
```

These functions normalize the interval, that is ensure that for an interval $A \in \mathbb{IR}$ it is guaranteed that $\underline{a} \leq \overline{a}$ and that if any of the two is $\varnothing$ so is the other.

`normalized()`
: Returns a copy of the current interval object but in a normalized state.

`normalize()`
: Normalizes the current interval object and returns a reference to it.

5.2.11 Set comparisons

```
bool contains(const T& v) const;
bool contains(const interval<T>& o) const;
bool overlaps(const interval<T>& o) const;
```

These functions make set assertions by comparison to an other $\mathbb{IR}$ or $\mathbb{R}$ value.

`contains(const T&)`
: Determines whether the given $\mathbb{R}$ value is contained in the current interval.

`contains(const interval<T>&)`
: Determines whether the given interval is fully inside and contained in the current interval.

`overlaps(const interval<T>&)`
: Determines whether the given interval overlaps with the current interval but is not fully contained in the latter.

5.2.12 Concatenation and Convex Hull

```
interval<T> concatenated(const interval<T>& o) const;
interval<T>& concatenate(const interval<T>& o);
```

`concatenated(const interval<T>&)`
: Return a new interval, that contains both the current and the argument supplied interval.

`concatenate(const interval<T>&)`
: Expand the current interval such that it also contains the supplied interval.

5.2.13 Bisection and Outside

```
std::pair<interval<T>,interval<T>> outside() const;
std::pair<interval<T>,interval<T>> bisect() const;
std::pair<interval<T>,interval<T>> bisect(const T& atloc) const;
```

`outside()`
: Returns two intervals representing the outside of the current interval. If only one interval is needed to represent the outside of the current interval, which is the case if the current interval is unbounded in one direction or $\varnothing$, the single outside interval will be in the `first` element of the `std::pair`, with the other being set to $\varnothing$.

`bisect()`
: Split the current interval into two halves if applicable, that is the current interval must be either whole or unbounded.

`bisect(const T&)`
: Split the current interval into two halves with a separation number at which the interval will be split.

5.2.14 Metrics and $\mathbb{R}$ conversion

```
T width() const;
T midpoint() const;
T magnitude() const;
T mignitude() const;
T pivot() const;
T size() const;
T radius() const;
```

These functions convert the interval into a $\mathbb{R}$ space representation. Given the current interval as $A \in \mathbb{IR}$ these metrics are defined as:

`width()`
: Width of the interval $\overline{a} - \underline{a}$.

`midpoint()`
: Midpoint of the interval $\frac{1}{2}\left(\overline{a} + \underline{a}\right)$.

`magnitude()`
: Magnitude of the interval $\max(|\underline{a}|, |\overline{a}|)$.

`mignitude()`
: Mignitude of the interval $\min(|\underline{a}|, |\overline{a}|)$.

`pivot()`
: Pivot of the interval $\sqrt{\max(|\underline{a}|, |\overline{a}|)\min(|\underline{a}|, |\overline{a}|)}$.

`size()`
: Size of the interval $\frac{1}{2}\left(|\underline{a}| + |\overline{a}|\right)$.

`radius()`
: Radius of the interval $\max\left(\frac{\overline{a}-\underline{a}}{2} - \underline{a}, \overline{a} - \frac{\overline{a}-\underline{a}}{2}\right)$.

5.2.15 Input/Output Functions

```
std::string as_string() const;
bool from_string(const std::string& str);
```

These functions handle conversion between strings, see **??** for string representations.

`as_string()`
: Returns the interval object as a string.

`from_string(const std::string& str)`
: Attempts to parse the given string and set the infinimum and supremum of the current object. Returns whether the parsing was successful.

5.3 Hausdorff distance

```
T hausdorff_distance(const interval<T>& a, const interval<T>& b);
T hausdorff_distance_error(const interval<T>& a, const interval<T>& b);
```

`hausdorff_distance(const interval<T>&, const interval<T>&)`
Computes the hausdorff distance between the two given intervals $A,B \in \mathbb{IR}$, as defined as:

$$d_H(A,B) = \max\{|\underline{b} - \underline{a}|, |\overline{b} - \overline{a}|\}$$

`hausdorff_distance_error(const interval<T>&, const interval<T>&)`
Computes the hausdorff distance error between the two given intervals $A,B \in \mathbb{IR}$, as defined as:

$$d_{He} = \frac{d_H(A,B)}{\overline{a} - \underline{a}}$$

5.4 Set operations

5.4.1 Set assertions

```
bool disjoint(const interval<T>& a, const interval<T>& b);
bool proper_subset(const interval<T>& a, const interval<T>& b);
bool subset(const interval<T>& a, const interval<T>& b);
bool proper_superset(const interval<T>& a, const interval<T>& b);
bool superset(const interval<T>& a, const interval<T>& b);
bool interior(const interval<T>& a, const interval<T>& b);
bool overlap(const interval<T>& a, const interval<T>& b);

bool inside(const interval<T>& a, const T& v);
```

`disjoint(const interval<T>& a, const interval<T>& b)`
: Determines whether the two given intervals are disjoint to each other, that is whether they do not have any common set.

`proper_subset(const interval<T>& a, const interval<T>& b)`
: Determines whether A is a proper subset of B, that is $A \subset B$.

`subset(const interval<T>& a, const interval<T>& b)`
: Determines whether A is a subset of B, that is $A \subseteq B$.

`proper_superset(const interval<T>& a, const interval<T>& b)`
: Determines whether A is a proper superset of B, that is $A \supset B$.

`superset(const interval<T>& a, const interval<T>& b)`
: Determines whether A is a superset of B, that is $A \supseteq B$.

`interior(const interval<T>& a, const interval<T>& b)`
: Determines whether A is in inside of the convex hull of B, that is $A \in B^\circ$

`overlap(const interval<T>& a, const interval<T>& b)`
: Determines whether the ranges of A and B overlap to any degree.

`inside(const interval<T>& a, const T& v)`
: Determines whether the given value v is contained inside A, that is whether $v \in A$.

5.4.2 Set operations

```
interval<T> hull(const interval<T>& a, const interval<T>& b);
interval<T> intersection(const interval<T>& a, const interval<T>& b);
interval<T> between(const interval<T>& a, const interval<T>& b);
```

`hull(const interval<T>& a, const interval<T>& b)`
: Returns an interval that contains both A and B.

`intersection(const interval<T>& a, const interval<T>& b)`
: Returns an interval that is the intersection of A and B.

`between(const interval<T>& a, const interval<T>& b)`
: Return an interval, that is between the two given intervals. If the given intervals overlap or are unbounded the resulting interval will be $\varnothing$.

5.5 Comparison operators

5.5.1 Set comparison

```
bool seq(const interval<T>&, const interval<T>&);
bool sne(const interval<T>&, const interval<T>&);
bool sle(const interval<T>&, const interval<T>&);
bool sge(const interval<T>&, const interval<T>&);
bool slt(const interval<T>&, const interval<T>&);
bool sgt(const interval<T>&, const interval<T>&);
```

Set comparison operators treat the given interval numbers as sets and provide *exact* comparisons, empty intervals of $\varnothing$ are explicitly considered valid. That is for $X, Y \in \mathbb{IR}$ and $\square \in \{=, <, \leq, >, \geq\}$:

$$X \square Y \equiv (\forall a \in X, \exists b \in Y : a \square b) \wedge (\exists a \in X, \forall b \in Y : a \neq b)$$

A special case is for inequality $\neq$ which is the same as above but using a disjunction instead of a conjunction, that is:

$$X \neq Y \equiv (\forall a \in X, \exists b \in Y : a \neq b) \vee (\exists a \in X, \forall b \in Y : a \neq b)$$

5.5.2 Value comparison

```
bool veq(const interval<T>&, const interval<T>&);
bool vne(const interval<T>&, const interval<T>&);
bool vle(const interval<T>&, const interval<T>&);
bool vge(const interval<T>&, const interval<T>&);
bool vlt(const interval<T>&, const interval<T>&);
bool vgt(const interval<T>&, const interval<T>&);
```

Value comparison operators treat the given intervals as if they were a single number. Empty intervals of $\varnothing$ are explicitly considered valid. That is for $X, Y \in \mathbb{IR}$ and $\square \in \{=, <, \leq, >, \geq\}$:

$$X \square Y \equiv (\forall a \in X, \exists b \in Y : a \square b)$$

A special case is for inequality $\neq$ which is the same as above but using a disjunction instead of a conjunction, that is:

$$X \neq Y \equiv (\forall a \in X, \exists b \in Y : a \neq b)$$

5.5.3 Possible comparison

```
bool peq(const interval<T>&, const interval<T>&);
bool pne(const interval<T>&, const interval<T>&);
bool ple(const interval<T>&, const interval<T>&);
bool pge(const interval<T>&, const interval<T>&);
bool plt(const interval<T>&, const interval<T>&);
bool pgt(const interval<T>&, const interval<T>&);
```

Possible comparison operators compare two intervals without checking for overlap between the intervals, that is possible comparison operators only check the appropriate extremas for matching and don't consider whether they overlap. As follows for a logical comparison operator $\square \in \{=, \neq, <, \leq, >, \geq\}$ for $X, Y \in \mathbb{IR}$ possible comparison operators are defined as:

$$X \square Y \equiv (X, Y \neq \varnothing) \wedge (\exists a \in X, \exists b \in Y : a \square b)$$

5.5.4 Certain comparison

```
bool ceq(const interval<T>&, const interval<T>&);
bool cne(const interval<T>&, const interval<T>&);
bool cle(const interval<T>&, const interval<T>&);
bool cge(const interval<T>&, const interval<T>&);
bool clt(const interval<T>&, const interval<T>&);
bool cgt(const interval<T>&, const interval<T>&);
```

Certain comparison operators check that the intervals do not overlap for their respective comparison operation. As follows for a logical comparison operator $\square \in \{=, \neq, <, \leq, >, \geq\}$ for $X, Y \in \mathbb{IR}$ certain comparison operators are defined as:

$$X \square Y \equiv (X, Y \neq \varnothing) \wedge (\forall a \in X, \forall b \in Y : a \square b)$$

5.6 Math Functions

```
interval<T> exp(const interval<T>& v);
interval<T> exp2(const interval<T>& v);
interval<T> log(const interval<T>& v);
interval<T> log2(const interval<T>& v);
interval<T> log10(const interval<T>& v);
interval<T> sqrt(const interval<T>& v);
interval<T> cbrt(const interval<T>& v);
interval<T> sin(const interval<T>& v);
interval<T> cos(const interval<T>& v);
interval<T> tan(const interval<T>& v);
interval<T> asin(const interval<T>& v);
interval<T> acos(const interval<T>& v);
interval<T> atan(const interval<T>& v);
interval<T> sinh(const interval<T>& v);
interval<T> cosh(const interval<T>& v);
interval<T> tanh(const interval<T>& v);
interval<T> asinh(const interval<T>& v);
interval<T> acosh(const interval<T>& v);
interval<T> atanh(const interval<T>& v);
interval<T> erf(const interval<T>& v);
interval<T> erfc(const interval<T>& v);
interval<T> hypot(const interval<T>& a, const interval<T>& b);
interval<T> atan2(const interval<T>& a, const interval<T>& b);
```

These execute the respective standard math function on an interval. The resulting interval is always normalized.

5.7 Specializations

5.7.1 `std::hash` - Standard hashing

The standard hash function `std::hash` is implemented for `interval<T>`.

5.7.2 `std::operator«` - Output stream

The output stream operator `std::operator«` is implemented by default for `interval<T>`.

5.7.3 `std::operator»` - Input stream

The input stream operator `std::operator»` is implemented by default for `interval<T>`. If an error occurs during parsing the `failbit` of the `std::istream` will be set, which may throw the `std::ios_base::failure` exception if it has been enabled.

6 Complex Intervals $\mathbb{KC}$ and $\mathbb{RC}$

6.1 General Information

6.1.1 Header Files

All function prototypes and data structures are declared in the `complex_interval.hpp` header file.

6.2 `kc_interval<T>` - Class Reference

The `kc_interval<T>` class represents a circular complex interval number of the given template type T. The class checks that the latter type T matches one of the following types by a `static_assert`, this is to ensure that the class only gets instantiated for types that are actually implemented. The following types are supported:

- `float`
- `double`
- `long double`

6.2.1 Public member variables

`mid`
: Complex number of the interval $k_p \in \mathbb{C}$.

`rad`
: Radius of the interval $k_r \in \mathbb{R}$.

Where $K \in \mathbb{KC}$.

6.2.2 Constructors

```
constexpr kc_interval() = default;
constexpr kc_interval(const kc_interval<T>&) = default;
constexpr kc_interval(const std::complex<T>& m, const T& r);
constexpr kc_interval(const std::pair<std::complex<T>,T>& p);
```

These are the constructors of the `kc_interval<T>` class. All constructors with the `constexpr` declaration specifier are implemented in inline.

`kc_interval()`
: Default constructor, leaves values uninitialized.

`kc_interval(const kc_interval<T>&)`
: Copy constructor.

`kc_interval(const std::complex<T>&, const T&)`
: Assignment constructor

`kc_interval(const std::pair<std::complex<T>,T>&)`
: Assignment constructor from a `std::pair`.

6.2.3 Assignment

```
constexpr kc_interval<T>& operator=(const kc_interval<T>&) = default;
constexpr kc_interval<T>& operator=(const std::pair<std::complex<T>,T>&)
```

The following assignment operators are implemented:

`operator=(const kc_interval<T>&)`
: Standard assignment operator.

`operator=(const std::pair<std::complex<T>,T>&)`
: Assignment operator from a `std::pair`.

6.2.4 Special Assignment

```
kc_interval<T>& set_empty();
kc_interval<T>& set_whole();
```

These functions assign special states to the current interval $k \in \mathbb{KC}$:

`set_empty()`
: Set the interval empty, such that $K = \varnothing$.

`set_whole()`
: Set the interval to span the whole complex space, $k_r = \infty$.

6.2.5 Comparison Operators

```
bool operator==(const kc_interval<T>& o) const;
bool operator!=(const kc_interval<T>& o) const;
```

These operators check for equality or inequality to another `kc_interval<T>` instance. The equality and inequality for the circular intervals $a, b \in \mathbb{KC}$ are defined for an operator $\square \in \{=, \neq\}$ as:

$$a \square b \equiv a_r \square b_r \wedge a_p \square b_p$$

6.2.6 Conversion functions

```
std::pair<std::complex<T>,T> as_pair() const;
```

This function converts the current interval into a `std::pair`.

6.2.7 Arithmetic Operators

```
kc_interval<T> operator+(const kc_interval<T>& o) const;
kc_interval<T> operator-(const kc_interval<T>& o) const;
kc_interval<T> operator*(const kc_interval<T>& o) const;
kc_interval<T> operator/(const kc_interval<T>& o) const;
kc_interval<T> operator+() const;
kc_interval<T> operator-() const;

kc_interval<T> operator+(const std::complex<T>& o) const;
kc_interval<T> operator-(const std::complex<T>& o) const;
kc_interval<T> operator*(const std::complex<T>& o) const;
kc_interval<T> operator/(const std::complex<T>& o) const;

kc_interval<T>& operator+=(const kc_interval<T>& o);
kc_interval<T>& operator-=(const kc_interval<T>& o);
kc_interval<T>& operator*=(const kc_interval<T>& o);
kc_interval<T>& operator/=(const kc_interval<T>& o);

kc_interval<T>& operator+=(const std::complex<T>& o);
kc_interval<T>& operator-=(const std::complex<T>& o);
kc_interval<T>& operator*=(const std::complex<T>& o);
kc_interval<T>& operator/=(const std::complex<T>& o);
```

These member functions execute the respective C++ arithmetic operation on the `kc_interval<T>` object.

6.2.8 Status functions

```
bool is_empty() const;
bool is_whole() const;
bool is_singleton() const;
bool is_okay() const;
```

These functions return the status of the current interval $k \in \mathbb{KC}$:

`is_empty()`
: Whether the interval is empty $k = \varnothing$.

`is_whole()`
: Whether the interval spans the whole complex space $k \neq \varnothing \wedge k_r = \infty$.

`is_whole()`
: Whether the interval refers to a single complex number. $k \neq \varnothing \wedge k_r = 0$.

`is_okay()`
: Whether the interval is properly defined, i.e. whether $k_r \geq 0$.

6.2.9 Normalization

```
kc_interval<T> normalized() const;
kc_interval<T>& normalize();
```

These functions ensure that the interval is normalized, such that `is_okay()` returns true.

`normalized()`
: Returns a copy of the current interval object but in a normalized state.

`normalize()`
: Normalizes the current interval object in-place and returns a reference to it.

6.2.10 Set comparison

```
bool contains(const std::complex<T>& v) const;
bool contains(const kc_interval<T>& o) const;
bool overlaps(const kc_interval<T>& o) const;
```

`contains(const std::complex<T>& v)`
: Determines whether the given $v \in \mathbb{C}$ is in the current interval $X \in \mathbb{KC}$, that is $v \in X$.

`contains(const kc_interval<T>& o)`
: Determines whether the given $O \in \mathbb{KC}$ is contained in the current interval $X \in \mathbb{KC}$, that is $O \in X$.

`overlaps(const kc_interval<T>& o)`
: Determines whether the given $O \in \mathbb{KC}$ overlaps with the current interval $X \in \mathbb{KC}$, that is $\exists v \in R$ such that $v \in O \wedge v \in X$.

6.2.11 Concatenation and convex hull

```
kc_interval<T> concatenated(const kc_interval<T>& o) const;
kc_interval<T>& concatenate(const kc_interval<T>& o);

kc_interval<T> concatenated_radius(const kc_interval<T>& o) const;
kc_interval<T>& concatenate_radius(const kc_interval<T>& o);
```

These functions compute the hull of the current and a further interval, $A, B \in \mathbb{KC}$ such that $A, B \in X$ for the resulting interval $X \in \mathbb{KC}$. This is done in one of two ways:

- By adjusting the center point to the middle of A and B and computing a radius that encompasses both.
- By leaving the center point of A in place and just expanding the radius to encompass B also.

`concatenated(const kc_interval<T>&)`
: Return a new interval, that contains the hull of both the current and the argument supplied interval.

`concatenate(const kc_interval<T>&)`
: Expand the current interval such that it also contains the supplied interval. This operation will change the center point.

`concatenated_radius(const kc_interval<T>&)`
: Return a new interval with the radius expanded to encompass B. The center point will be that of the current interval.

`concatenate_radius(const kc_interval<T>&)`
: Expand the radius of the current interval to encompass B. The center point will remain unchanged.

6.2.12 Metrics and $\mathbb{R}$ conversion

```
T magnitude() const;
T mignitude() const;
T width() const;
```

`magnitude()`
: Magnitude of the interval $|k_p| + k_r$.

`mignitude()`
: Mignitude of the interval $|k_p| - k_r$.

`width()`
: Width of the interval $2k_r$.

6.2.13 Complex Conjugation

```
kc_interval<T> conjugated() const;
kc_interval<T>& conjugate();
```

Complex conjugate functions for the $\mathbb{KC}$ interval.

`conjugated()`
: Returns a conjugated version of the interval.

`conjugate()`
: Sets the current interval to it's complex conjugate and return a reference to it.

6.2.14 Input/Output Functions

```
std::string as_string() const;
bool from_string(const std::string& str);
```

These functions handle conversion between strings, see **??** for string representations.

`as_string()`
: Returns the interval object as a string.

`from_string(const std::string& str)`
: Attempts to parse the given string and set the infinimum and supremum of the current object. Returns whether the parsing was successful.

6.3 rc_interval<T> - Class Reference

The `rc_interval<T>` class represents a rectangular complex interval number of the given template type T. The class checks that the latter type T matches one of the following types by a `static_assert`, this is to ensure that the class only gets instantiated for types that are actually implemented. The following types are supported:

- `float`
- `double`
- `long double`

6.3.1 Public member variables

`real`
: An interval representing the real part of the complex number $\Re_i(k) \in \mathbb{IR}$.

`imag`
: An interval representing the imaginary part of the complex number $\Im_i(k) \in \mathbb{IR}$.

Where $k \in \mathbb{RC}$.

6.3.2 Constructors

```
constexpr rc_interval() = default;
constexpr rc_interval(const kc_interval<T>&) = default;

constexpr rc_interval(const T& ri, const T& rs,
                      const T& ii, const T& is);
constexpr rc_interval(const interval<T>& r, const interval<T>& i);
constexpr rc_interval(const std::complex<T>& s);
```

These are the constructors of the `kc_interval<T>` class. All constructors with the `constexpr` declaration specifier are implemented in inline.

`rc_interval()`
: Default constructor, leaves values uninitialized.

`rc_interval(const rc_interval<T>&)`
: Copy constructor.

`rc_interval(const T& ri, const T& rs, const T& ii, const T& is)`
: Assignment constructor where `ri` and `rs` are the real part infinimum and supremum and `ii` and `is` are the imaginary part infinimum and supremum.

`rc_interval(const interval<T>& r, const interval<T>& i)`
: Assignment constructor, where `r` is the real part interval and `i` is the imaginary part interval.

`rc_interval(const std::complex<T>& s)`
: Assignment constructor, assign this interval as a singleton $s \in \mathbb{C}$.

6.3.3 Assignment

```
constexpr rc_interval<T>& operator=(const kc_interval<T>&) = default;
constexpr rc_interval<T>& operator=(const std::pair<interval<T>, interval<T>>& o);
```

The following assignment operators are implemented:

`operator=(const rc_interval<T>&)`
: Standard assignment operator.

`operator=(const std::pair<interval<T>, interval<T»& o)`
: Assign from a `std::pair` of normal intervals, where the first interval is the interval for the real part and the second interval is the interval for the imaginary component.

6.3.4 Special Assignment

```
rc_interval<T>& set_empty();
rc_interval<T>& set_whole();
```

These functions assign special states to the current interval $K \in \mathbb{RC}$:

`set_empty()`
: Set the interval empty, such that $K = \varnothing$.

`set_whole()`
: Set the interval to span the whole complex space, such that $K = \mathbb{C}$.

6.3.5 Comparison Operators

```
bool operator==(const rc_interval<T>& o) const;
bool operator!=(const rc_interval<T>& o) const;
```

These operators check for equality or inequality to another `rc_interval<T>` instance. This is done by set comparison of the real and imaginary component intervals.

6.3.6 Conversion functions

```
std::pair<interval<T>,interval<T>> as_pair() const;
```

This function converts the current interval into a `std::pair`, where the `first` element is the real component interval, and the `second` element is the imaginary component.

6.3.7 Arithmetic Operators

```
rc_interval<T> operator+(const rc_interval<T>& o) const;
rc_interval<T> operator-(const rc_interval<T>& o) const;
rc_interval<T> operator*(const rc_interval<T>& o) const;
rc_interval<T> operator/(const rc_interval<T>& o) const;
rc_interval<T> operator+() const;
rc_interval<T> operator-() const;

rc_interval<T> operator+(const std::complex<T>& o) const;
rc_interval<T> operator-(const std::complex<T>& o) const;
rc_interval<T> operator*(const std::complex<T>& o) const;
rc_interval<T> operator/(const std::complex<T>& o) const;

rc_interval<T>& operator+=(const rc_interval<T>& o);
rc_interval<T>& operator-=(const rc_interval<T>& o);
rc_interval<T>& operator*=(const rc_interval<T>& o);
rc_interval<T>& operator/=(const rc_interval<T>& o);

rc_interval<T>& operator+=(const std::complex<T>& o);
rc_interval<T>& operator-=(const std::complex<T>& o);
rc_interval<T>& operator*=(const std::complex<T>& o);
rc_interval<T>& operator/=(const std::complex<T>& o);
```

These member functions execute the respective C++ arithmetic operation on the `rc_interval<T>` object.

6.3.8 Status functions

```
bool is_empty() const;
bool is_whole() const;
bool is_singleton() const;
bool is_okay() const;
```

These functions return the status of the current interval k, is properly $k \in \mathbb{RC}$.

`is_empty()`
: Whether the interval is empty i.e. $\Re_i, \Im_i \neq \varnothing$.

`is_whole()`
: Whether the interval spans the whole complex space $(\Re_i, \Im_i \neq \varnothing) \wedge (|\underline{\Re_i}, \overline{\Re_i}, \underline{\Im_i}, \overline{\Im_i}| = \infty)$.

`is_singleton()`
: Whether the interval refers to a single complex number. $(\Re_i, \Im_i \neq \varnothing) \wedge (\underline{\Re_i} = \overline{\Re_i}) \wedge (\underline{\Im_i} = \overline{\Im_i})$.

`is_okay()`
: Whether the interval is properly defined.

6.3.9 Normalization

```
rc_interval<T> normalized() const;
rc_interval<T>& normalize();
```

These functions ensure that the interval is normalized, such that `is_okay()` returns true.

`normalized()`
: Returns a copy of the current interval object but in a normalized state.

`normalize()`
: Normalizes the current interval object in-place and returns a reference to it.

6.3.10 Set comparison

```
bool contains(const std::complex<T>& v) const;
bool contains(const rc_interval<T>& o) const;
bool overlaps(const rc_interval<T>& o) const;
```

`contains(const std::complex<T>& v)`
: Determines whether the given $v \in \mathbb{C}$ is in the current interval $X \in \mathbb{RC}$, that is $v \in X$.

`contains(const kc_interval<T>& o)`
: Determines whether the given $O \in \mathbb{RC}$ is contained in the current interval $X \in \mathbb{RC}$, that is $O \in X$.

`overlaps(const kc_interval<T>& o)`
: Determines whether the given $O \in \mathbb{RC}$ overlaps with the current interval $X \in \mathbb{RC}$, that is $\exists v \in R$ such that $v \in X \wedge v \in O$.

6.3.11 Concatenation and convex hull

```
rc_interval<T> concatenated(const rc_interval<T>& o) const;
rc_interval<T>& concatenate(const rc_interval<T>& o);
```

These functions compute the hull of the current and a further interval, $A,B \in \mathbb{RC}$ such that $A,B \in X$ for the resulting interval $X \in \mathbb{RC}$.

`concatenated(const rc_interval<T>&)`
: Return a new interval, that contains the hull of both the current and the argument supplied interval.

`concatenate(const rc_interval<T>&)`
: Expand the current interval such that it also contains the supplied interval.

6.3.12 Metric and $\mathbb{C}$ conversion

```
std::complex<T> width() const;
std::complex<T> midpoint() const;
std::complex<T> magnitute() const;
std::complex<T> mignitute() const;
std::complex<T> pivot() const;
std::complex<T> size() const;
std::complex<T> radius() const;
```

These functions provide metrics for the $\mathbb{RC}$ interval. These use the metrics provided in the `interval<T>` for both the real and imaginary parts.

`width()`
: Width of the interval.

`midpoint()`
: Midpoint of the interval.

`magnitute()`
: Magnitute of the interval.

`mignitute()`
: Mignitute of the interval.

`pivot()`
: Pivot of the interval.

`size()`
: Size of the interval.

`radius()`
: Radius of the interval.

6.3.13 Complex Conjugation

```
rc_interval<T> conjugated() const;
rc_interval<T>& conjugate();
```

Complex conjugate functions for the $\mathbb{RC}$ interval.

`conjugated()`
: Returns a conjugated version of the interval.

`conjugate()`
: Sets the current interval to it's complex conjugate and return a reference to it.

6.3.14 Riemann Projection

```
rc_interval<T> projected() const;
rc_interval<T>& project();
```

These functions project a $\mathbb{RC}$ interval onto a Riemann sphere.

`projected()`
: Returns a Riemann sphere projected version of the interval.

`project()`
: Sets the current interval to it's Riemann sphere projection and return a reference to it.

6.3.15 Input/Output Functions

```
std::string as_string() const;
bool from_string(const std::string& str);
```

These functions handle conversion between strings, see **??** for string representations.

`as_string()`
: Returns the interval object as a string.

`from_string(const std::string& str)`
: Attempts to parse the given string and set the infinimum and supremum of the current object. Returns whether the parsing was successful.

6.4 Set operations

6.4.1 Set Assertions $\mathbb{KC}$

```
bool disjoint(const kc_interval<T>& a, const kc_interval<T>& b);
bool proper_subset(const kc_interval<T>& a, const kc_interval<T>& b);
bool subset(const kc_interval<T>& a, const kc_interval<T>& b);
bool proper_superset(const kc_interval<T>& a, const kc_interval<T>& b);
bool superset(const kc_interval<T>& a, const kc_interval<T>& b);
bool overlap(const kc_interval<T>& a, const kc_interval<T>& b);

bool inside(const kc_interval<T>& a, const std::complex<T>& v);
```

These functions assert set properties between two given circular intervals $A, B \in \mathbb{KC}$.

`disjoint(const kc_interval<T>& a, const kc_interval<T>& b)`
: Whether the two intervals are disjoint to each other.

`proper_subset(const kc_interval<T>& a, const kc_interval<T>& b);`
: Determines whether A is a proper subset of B, that is $A \subset B$.

`subset(const kc_interval<T>& a, const kc_interval<T>& b);`
: Determines whether A is a subset of B, that is $A \subseteq B$.

`proper_superset(const kc_interval<T>& a, const kc_interval<T>& b);`
: Determines whether A is a proper superset of B, that is $A \supset B$.

`superset(const kc_interval<T>& a, const kc_interval<T>& b);`
: Determines whether A is a superset of B, that is $A \supseteq B$.

`overlap(const kc_interval<T>& a, const kc_interval<T>& b);`
: Determines whether A intersects or overlaps with B.

`inside(const kc_interval<T>& a, const std::complex<T>& v);`
: Whether the complex number $v \in \mathbb{C}$ is within the span of the given circular interval A.

6.4.2 Set Assertions $\mathbb{RC}$

```
bool disjoint(const rc_interval<T>& a, const rc_interval<T>& b);
bool proper_subset(const rc_interval<T>& a, const rc_interval<T>& b);
bool subset(const rc_interval<T>& a, const rc_interval<T>& b);
bool proper_superset(const rc_interval<T>& a, const rc_interval<T>& b);
bool superset(const rc_interval<T>& a, const rc_interval<T>& b);
bool overlap(const rc_interval<T>& a, const rc_interval<T>& b);

bool inside(const rc_interval<T>& a, const std::complex<T>& v);
```

These functions assert set properties between two given rectangular intervals $A, B \in \mathbb{RC}$.

`disjoint(const rc_interval<T>& a, const rc_interval<T>& b)`
: Whether the two intervals are disjoint to each other.

`proper_subset(const rc_interval<T>& a, const rc_interval<T>& b);`
: Determines whether A is a proper subset of B, that is $A \subset B$.

`subset(const rc_interval<T>& a, const rc_interval<T>& b);`
: Determines whether A is a subset of B, that is $A \subseteq B$.

`proper_superset(const rc_interval<T>& a, const rc_interval<T>& b);`
: Determines whether A is a proper superset of B, that is $A \supset B$.

`superset(const rc_interval<T>& a, const rc_interval<T>& b);`
: Determines whether A is a superset of B, that is $A \supseteq B$.

`overlap(const rc_interval<T>& a, const rc_interval<T>& b);`
: Determines whether A intersects or overlaps with B.

`inside(const rc_interval<T>& a, const std::complex<T>& v);`

Whether the complex number $v \in \mathbb{C}$ is within the span of the given rectangular complex interval A.

6.4.3 Set Operations $\mathbb{KC}$

```
kc_interval<T> hull(const kc_interval<T>& a, const kc_interval<T>& b);
kc_interval<T> hull_radius(const kc_interval<T>& a, const kc_interval<T>& b);
```

These functions do set manipulations on $\mathbb{KC}$ intervals.

`hull(const kc_interval<T>& a, const kc_interval<T>& b)`
: Return an interval $X \in \mathbb{KC}$ that encloses both $A,B \in \mathbb{KC}$, that is $A,B \in X$.

`hull_radius(const kc_interval<T>& a, const kc_interval<T>& b)`
: Return an interval $X \in \mathbb{KC}$ that encloses both $A,B \in \mathbb{KC}$, that is $A,B \in X$ by expanding the radius of the first interval A to enclose B.

6.4.4 Set Operations $\mathbb{RC}$

```
rc_interval<T> hull(const rc_interval<T>& a, const rc_interval<T>& b);
rc_interval<T> intersection(const rc_interval<T>& a, const rc_interval<T>& b);
```

Set operations given two rectangular complex intervals, $A,B \in \mathbb{RC}$:

`hull(const rc_interval<T>& a, const rc_interval<T>& b)`
: Returns an interval that contains both A and B.

`intersection(const rc_interval<T>& a, const rc_interval<T>& b)`
: Returns an interval that is the intersection of A and B.

6.5 Complex Hausdorff Distance ℝℂ

```
std::complex<T> hausdorff_distance(const rc_interval<T>& a, const rc_interval<T>& b);
std::complex<T> hausdorff_distance_error(const rc_interval<T>& a, const rc_interval<T>& b);
```

The complex Hausdorff distance is the normal Hausdorff distance applied to the real and imaginary components individually.

`hausdorff_distance(const rc_interval<T>&, const rc_interval<T>&)`
Computes the hausdorff distance between the two given intervals $A, B \in \mathbb{IR}$, as defined as:

$$d_H(A,B) = \max\{|\underline{b} - \underline{a}|, |\overline{b} - \overline{a}|\}$$

For $X, Y \in \mathbb{RC}$ this becomes:

$$d_H(X,Y) = d_H(\Re_i(X), \Re_i(Y)) + i d_H(\Im_i(X), \Im_i(Y))$$

`hausdorff_distance_error(const rc_interval<T>&, const rc_interval<T>&)`
Computes the hausdorff distance error between the two given intervals $A, B \in \mathbb{IR}$, as defined as:

$$d_{He} = \frac{d_H(A,B)}{\overline{a} - \underline{a}}$$

So in the case of $X, Y \in \mathbb{RC}$:

$$d_{He}(X,Y) = d_{He}(\Re_i(X), \Re_i(Y)) + i d_{He}(\Im_i(X), \Im_i(Y))$$

6.6 Specializations

6.6.1 `std::hash` - Standard hashing

The standard hash function `std::hash` is implemented for both `kc_interval<T>` and `rc_interval<T>`.

6.6.2 `std::operator«` - Output stream

The output stream operator `std::operator«` is implemented by default for `kc_interval<T>` and `rc_interval`.

6.6.3 `std::operator»` - Input stream

The input stream operator `std::operator»` is implemented by default for `kc_interval<T>` and `rc_interval`. If an error occurs during parsing the `failbit` of the `std::istream` will be set, which may throw the `std::ios_base::failure` exception if it has been enabled.

7 Kaucher Intervals 𝕂ℝ

7.1 General Description

For the C++ programming language family the package is mainly implemented as a templated class, with the exception of comparison functions which are implemented as procedural function declarations.

7.1.1 Header Files

All function prototypes and data structures are declared in the `kaucher_interval.hpp` header file, which in itself includes a couple further header files that contain all subsystems of the interval package.

7.2 `kaucher_interval<`T`>` - Class Reference

The `kaucher_interval<`T`>` class represents a Kaucher interval number of the given template type T. The class checks that the latter type T matches one of the following types by a `static_assert`, this is to ensure that the class only gets instantiated for types that are actually implemented. The following types are supported:

- `float`
- `double`
- `long double`

7.2.1 Public Member Variables

`inf`
: Infinimum of the interval $\underline{x}$.

`sup`
: Supremum of the interval $\overline{x}$.

7.2.2 Constructors

```
constexpr kaucher_interval() = default;
constexpr kaucher_interval(const T& a, const T& b);
constexpr kaucher_interval(const T a[2]);
constexpr kaucher_interval(const T v);
constexpr kaucher_interval(const kaucher_interval<T>& o) = default;
constexpr kaucher_interval(const std::pair<T,T>& p);
constexpr kaucher_interval(const interval<T>& o);
```

These are the constructors of the `kaucher_interval<T>` class. All constructors with the `constexpr` declaration specifier are implemented in inline.

`kaucher_interval()`
: Default constructor, leaves infinimum and supremum uninitialized.

`kaucher_interval(const kaucher_interval<T>&)`
: Default copy constructor.

`kaucher_interval(const T&, const T&)` **and** `kaucher_interval(const T a[2])`
: Assignment constructor, assigns infinimum and supremum to the first and second argument respectively.

`kaucher_interval(const T&)`
: Assignment constructor, initializes the interval as a singleton of the given value.

`kaucher_interval(std::pair<T,T> &)`
: Assignment constructor, initializes the infinimum and supremum to the `first` and `second` value of the `std::pair`, respectively.

`kaucher_interval(const interval<T>&)`
: Assignment constructor, initializes a Kaucher interval from a classic interval.

7.2.3 Assignment

```
constexpr kaucher_interval<T>& operator=(const kaucher_interval<T>&) = default;
kaucher_interval<T>& operator=(const std::pair<T,T>& o);
kaucher_interval<T>& operator=(const interval<T>& o);

kaucher_interval<T>& set_midpoint(const T& mid, const T& error);
kaucher_interval<T>& set_range(const T& i, const T& s);
kaucher_interval<T>& set_range(const std::pair<T,T>& p);
```

These member functions assign the infinimum and supremum of the interval.

`operator=(const kaucher_interval<T>&)`
: Standard one-to-one assignment.

`operator=(const interval<T>&)`
: Assign values form a standard `interval<T>`.

`operator=(const std::pair<T,T>&)` **and** `set_range(const std::pair<T,T>& p)`
: Assigns the infinimum to the first element, and the supremum to the second element of the `std::pair`.

`set_midpoint(const T& mid, const T& error)`
: Assign the infinimum to `mid-|error|` the supremum to `mid+|error|`.

`set_range(const T& i, const T& s)`
: Assign the infinimum and supremum from the given first and second argument, respectively.

7.2.4 Special Assignment

```
kaucher_interval<T>& set_empty();
```

These functions set the Kaucher interval to special values.

`set_empty()`
: Set the interval to an empty state $K = \varnothing$ for $K \in \mathbb{KR}$.

7.2.5 Comparison operators

```
bool operator==(const kaucher_interval<T>& o) const;
bool operator!=(const kaucher_interval<T>& o) const;
bool operator==(const std::pair<T,T>& o) const;
bool operator!=(const std::pair<T,T>& o) const;
```

These two operators check the equality or inequality of the infinimum and supremum element-wise. Let $T, O \in \mathbb{KR}$ with T being the current instance, `this` and O being the given other interval to compare to, that is `o`:

`operator==(const kaucher_interval<T>&)` **and** `operator==(const std::pair<T,T>&)`
: Set equality operation, as defined as $(\underline{t} = \underline{o}) \wedge (\overline{t} = \overline{o})$. With either an other interval or a `std::pair` whose first element is interpreted as $\underline{o}$ and second element $\overline{o}$.

`operator!=(const kaucher_interval<T>&)` **and** `operator!=(const std::pair<T,T>&)`
: Set inequality operation, as defined as $(\underline{t} \neq \underline{o}) \vee (\overline{t} \neq \overline{o})$. With either an other interval or a `std::pair` whose first element is interpreted as $\underline{o}$ and second element $\overline{o}$.

7.2.6 Conversion functions

```
std::pair<T,T> as_pair() const;
interval<T> as_interval() const;
```

These member functions convert the current instance of the `kaucher_interval<T>` class into other formats.

`as_pair()`
: Convert the current interval to a `std::pair` with the `first` and `second` members set to the infinimum and supremum, respectively.

`as_interval()`
: Convert the Kaucher interval $\mathbb{KR}$ to a normal *interval<T>*, $\mathbb{IR}$.

7.2.7 Arithmetic Operators

```
kaucher_interval<T> operator+(const kaucher_interval<T>& o) const;
kaucher_interval<T> operator-(const kaucher_interval<T>& o) const;
kaucher_interval<T> operator*(const kaucher_interval<T>& o) const;
kaucher_interval<T> operator/(const kaucher_interval<T>& o) const;
kaucher_interval<T> operator-() const;

kaucher_interval<T>& operator+=(const kaucher_interval<T>& o);
kaucher_interval<T>& operator-=(const kaucher_interval<T>& o);
kaucher_interval<T>& operator*=(const kaucher_interval<T>& o);
kaucher_interval<T>& operator/=(const kaucher_interval<T>& o);
```

These operators execute the respective equivalent C++ standard arithmetic operation on the `kaucher_interval<T>`.

7.2.8 Normalization

```
interval<T> normalized() const;
interval<T>& normalize();
```

These functions normalize the interval, that is ensure that for an interval $A \in \mathbb{KR}$ it is guaranteed that if any of $\overline{a}$ or $\underline{a}$ is $\varnothing$ so is the other.

`normalized()`
: Returns a copy of the current interval object but in a normalized state.

`normalize()`
: Normalizes the current interval object and returns a reference to it.

7.2.9 Set comparisons

```
bool contains(const T& v) const;
bool contains(const interval<T>& o) const;
bool overlaps(const interval<T>& o) const;
```

These functions make set assertions by comparison to an other $\mathbb{KR}$ or $\mathbb{R}$ value.

`contains(const T&)`
: Determines whether the given $\mathbb{R}$ value is contained in the current interval.

`contains(const kaucher_interval<T>&)`
: Determines whether the given interval is fully inside and contained in the current interval.

`overlaps(const kaucher_interval<T>&)`
: Determines whether the given interval overlaps with the current interval but is not fully contained in the latter.

7.2.10 Status Functions

```
bool is_empty() const;
```

These functions check whether the Kaucher interval has a special value.

`is_empty()`
: Checks whether the interval is in an empty state $K = \varnothing$ for $K \in \mathbb{KR}$.

7.2.11 Metrics and $\mathbb{R}$ conversion

```
T width() const;
T midpoint() const;
T magnitude() const;
T mignitude() const;
T pivot() const;
T size() const;
T radius() const;
```

These functions convert the interval into a $\mathbb{R}$ space representation. Given the current interval as $A \in \mathbb{KR}$ these metrics are defined as:

`width()`
: Width of the interval $\overline{a} - \underline{a}$.

`midpoint()`
: Midpoint of the interval $\frac{1}{2}\left(\overline{a} + \underline{a}\right)$.

`magnitude()`
: Magnitude of the interval $\max(|\underline{a}|, |\overline{a}|)$.

`mignitude()`
: Mignitude of the interval $\min(|\underline{a}|, |\overline{a}|)$.

`pivot()`
: Pivot of the interval $\sqrt{\max(|\underline{a}|, |\overline{a}|)\min(|\underline{a}|, |\overline{a}|)}$.

`size()`
: Size of the interval $\frac{1}{2}\left(|\underline{a}| + |\overline{a}|\right)$.

`radius()`
: Radius of the interval $\max\left(\frac{\overline{a}-\underline{a}}{2} - \underline{a}, \overline{a} - \frac{\overline{a}-\underline{a}}{2}\right)$.

7.3 Hausdorff distance

```
T hausdorff_distance(const kaucher_interval<T>& a, const kaucher_interval<T>& b);
T hausdorff_distance_error(const kaucher_interval<T>& a, const kaucher_interval<T>& b);
```

`hausdorff_distance(const kaucher_interval<T>&, const kaucher_interval<T>&)`
Computes the hausdorff distance between the two given Kaucher intervals $A,B \in \mathbb{KR}$, as defined as:

$$d_H(A,B) = \max\{|\underline{b} - \underline{a}|, |\overline{b} - \overline{a}|\}$$

`hausdorff_distance_error(const kaucher_interval<T>&, const kaucher_interval<T>&)`
Computes the hausdorff distance error between the two given Kaucher intervals $A,B \in \mathbb{KR}$, as defined as:

$$d_{He} = \frac{d_H(A,B)}{\overline{a} - \underline{a}}$$

7.4 Set operations

7.4.1 Set assertions

```
bool disjoint(const kaucher_interval<T>& a, const kaucher_interval<T>& b);
bool proper_subset(const kaucher_interval<T>& a, const kaucher_interval<T>& b);
bool subset(const kaucher_interval<T>& a, const kaucher_interval<T>& b);
bool proper_superset(const kaucher_interval<T>& a, const kaucher_interval<T>& b);
bool superset(const kaucher_interval<T>& a, const kaucher_interval<T>& b);
bool overlap(const kaucher_interval<T>& a, const kaucher_interval<T>& b);
```

`disjoint(const kaucher_interval<T>& a, const kaucher_interval<T>& b)`
: Determines whether the two given intervals are disjoint to each other, that is whether they do not have any common set.

`proper_subset(const kaucher_interval<T>& a, const kaucher_interval<T>& b)`
: Determines whether A is a proper subset of B, that is $A \subset B$.

`subset(const kaucher_interval<T>& a, const kaucher_interval<T>& b)`
: Determines whether A is a subset of B, that is $A \subseteq B$.

`proper_superset(const kaucher_interval<T>& a, const kaucher_interval<T>& b)`
: Determines whether A is a proper superset of B, that is $A \supset B$.

`superset(const kaucher_interval<T>& a, const kaucher_interval<T>& b)`
: Determines whether A is a superset of B, that is $A \supseteq B$.

`overlap(const kaucher_interval<T>& a, const kaucher_interval<T>& b)`
: Determines whether the ranges of A and B overlap to any degree.

7.5 Specializations

7.5.1 `std::hash` - Standard hashing

The standard hash function `std::hash` is implemented for `kaucher_interval<T>`.

7.5.2 `std::operator«` - Output stream

The output stream operator `std::operator«` is implemented by default for `kaucher_interval<T>`.

7.5.3 `std::operator»` - Input stream

The input stream operator `std::operator»` is implemented by default for `kaucher_interval<T>`. If an error occurs during parsing the `failbit` of the `std::istream` will be set, which may throw the `std::ios_base::failure` exception if it has been enabled.

8 Triplex $\mathbb{TR}$

8.1 General Description

For the C++ programming language family the package is mainly implemented as a templated class, with the exception of comparison functions which are implemented as procedural function declarations.

8.1.1 Header Files

All function prototypes and data structures are declared in the `triplex.hpp` header file, which in itself includes a couple further header files that contain all subsystems of the interval package.

8.2 `triplex<`*T*`>` - Class Reference

The `triplex<`*T*`>` class represents a triplex number of the given template type *T*. The class checks that the latter type *T* matches one of the following types by a `static_assert`, this is to ensure that the class only gets instantiated for types that are actually implemented. The following types are supported:

- `float`
- `double`
- `long double`

8.2.1 Public Member Variables

`inf`
: Infinimum of the interval $\underline{x}$.

`val`
: Machine precision value of the interval x.

`sup`
: Supremum of the interval $\overline{x}$.

8.2.2 Constructors

```
constexpr triplex() = default;
constexpr triplex(const triplex<T>&) = default;
constexpr triplex(const T& v);
constexpr triplex(const T& a, const T& b, const T& c);
constexpr triplex(const interval<T>& o);
constexpr triplex(const interval<T>& o, const T& m);
constexpr triplex(const kaucher_interval<T>& o);
constexpr triplex(const kaucher_interval<T>& o, const T& m);
constexpr triplex(const std::pair<T,T>& p, const T& m);
constexpr triplex(const T v[3]);
```

These are the constructors of the `triplex<T>` class. All constructors with the `constexpr` declaration specifier are implemented in inline.

`triplex()`
: Default constructor, leaves infinimum, value and supremum uninitialized.

`triplex(const triplex<T>&)`
: Default copy constructor.

`triplex(const T&, const T&, const T&)` **and** `triplex(const T a[3])`
: Assignment constructor, assigns infinimum, value and supremum to the first and second argument respectively.

`triplex(const T&)`
: Assignment constructor, initializes the interval as a singleton of the given value.

`triplex(std::pair<T,T> &, const T&)`
: Assignment constructor, initializes the infinimum and supremum to the `first` and `second` value of the `std::pair`, respectively and sets the value to the given scalar.

`triplex(const interval<T>&)` **and** `triplex(const interval<T>&, const T&)`
: Assignment constructor, initializes a triplex from a classic interval, the value is either set as the middle of the interval or explicitly by the given scalar.

`triplex(const kaucher_interval<T>&)` **and** `triplex(const kaucher_interval<T>&, const T&)`
: Assignment constructor, initializes a triplex from a kaucher interval, the value is either set as the middle of the interval or explicitly by the given scalar.

8.2.3 Assignment

```
constexpr triplex<T>& operator=(const triplex<T>&) = default;
triplex<T>& set(const T&, const T&, const T&);
```

These member functions assign the infinimum, value and supremum of the interval.

`operator=(const triplex<T>&)`
: Standard one-to-one assignment.

`set(const T&, const T&, const T&)`
: Set the infinimum $\underline{x}$, value x and supremum $\overline{x}$ of the triplex.

8.2.4 Conversion functions

```
std::pair<T,T> as_pair() const;
interval<T> as_interval() const;
kaucher_interval<T> as_kaucher_interval() const;
```

These member functions convert the current instance of the `triplex<T>` class into other formats.

`as_pair()`
: Convert the current interval to a `std::pair` with the `first` and `second` members set to the infinimum and supremum, respectively.

`as_interval()`
: Convert the triplex to a classic interval.

`as_kaucher_interval()`
: Convert the triplex to a kaucher interval.

8.2.5 Comparison Operators

```
bool operator==(const triplex<T>& o) const;
bool operator!=(const triplex<T>& o) const;
```

These operators check the equality or inequality of the current triplex object to another triplex object.

8.2.6 Normalization

```
triplex<T> normalized() const;
triplex<T>& normalize();
```

These functions normalize the triplex, that is ensure that for a triplex $T \in \mathbb{TR}$ it is guaranteed that if any of $\overline{t}$, t or $\underline{t}$ is $\varnothing$ so are the others.

`normalized()`
: Returns a copy of the current triplex object but in a normalized state.

`normalize()`
: Normalizes the current triplex object and returns a reference to it.

8.2.7 Status functions

```
bool is_empty() const;
```

These member functions provide information over the current state of the triplex.

`is_empty()`
: Whether the triplex is empty $T = \varnothing$ for $T \in \mathbb{TR}$.

8.2.8 Special Assignment

```
triplex<T>& set_empty() const;
```

These functions set the triplex to special values.

`set_empty()`
: Sets the current triplex to an empty state $T = \varnothing$ for $T \in \mathbb{TR}$.

8.2.9 Arithmetic Operators

```
triplex<T> operator+(const triplex<T>& o) const;
triplex<T> operator-(const triplex<T>& o) const;
triplex<T> operator*(const triplex<T>& o) const;
triplex<T> operator/(const triplex<T>& o) const;
triplex<T> operator-() const;

triplex<T>& operator+=(const triplex<T>& o);
triplex<T>& operator-=(const triplex<T>& o);
triplex<T>& operator*=(const triplex<T>& o);
triplex<T>& operator/=(const triplex<T>& o);
```

These operators execute the respective equivalent C++ standard arithmetic operation on the `triplex<T>`.

8.2.10 Set comparisons

```
bool contains(const T& v) const;
bool contains(const triplex<T>& o) const;
bool overlaps(const triplex<T>& o) const;
```

These functions make set assertions by comparison to an other $\mathbb{TR}$ or $\mathbb{R}$ value.

`contains(const T&)`
: Determines whether the given $\mathbb{R}$ value is contained in the current triplex.

`contains(const triplex<T>&)`
: Determines whether the given triplex is fully inside and contained in the current triplex.

`overlaps(const triplex<T>&)`
: Determines whether the given triplex overlaps with the current triplex but is not fully contained in the latter.

8.2.11 Metrics and $\mathbb{R}$ conversion

```
T width() const;
T midpoint() const;
T magnitude() const;
T mignitude() const;
T pivot() const;
T size() const;
T radius() const;
```

These functions convert the triplex into a $\mathbb{R}$ space representation. Given the current triplex as $A \in \mathbb{TR}$ these metrics are defined as:

`width()`
: Width of the triplex $\overline{a} - \underline{a}$.

`midpoint()`
: Midpoint of the triplex $\frac{1}{2}\left(\overline{a} + \underline{a}\right)$.

`magnitude()`
: Magnitude of the triplex $\max(|\underline{a}|, |\overline{a}|)$.

`mignitude()`
: Mignitude of the triplex $\min(|\underline{a}|, |\overline{a}|)$.

`pivot()`
: Pivot of the triplex $\sqrt{\max(|\underline{a}|, |\overline{a}|)\min(|\underline{a}|, |\overline{a}|)}$.

`size()`
: Size of the triplex $\frac{1}{2}\left(|\underline{a}| + |\overline{a}|\right)$.

`radius()`
: Radius of the triplex $\max\left(\frac{\overline{a}-\underline{a}}{2} - \underline{a}, \overline{a} - \frac{\overline{a}-\underline{a}}{2}\right)$.

8.3 Hausdorff distance

```
T hausdorff_distance(const triplex<T>& a, const triplex<T>& b);
T hausdorff_distance_error(const triplex<T>& a, const triplex<T>& b);
```

`hausdorff_distance(const triplex<T>&, const triplex<T>&)`
Computes the hausdorff distance between the two given Triplexes $A,B \in \mathbb{KR}$, as defined as:

$$d_H(A,B) = \max\{|\underline{b} - \underline{a}|, |\overline{b} - \overline{a}|\}$$

`hausdorff_distance_error(const triplex<T>&, const triplex<T>&)`
Computes the hausdorff distance error between the two given Triplexes $A,B \in \mathbb{KR}$, as defined as:

$$d_{He} = \frac{d_H(A,B)}{\overline{a} - \underline{a}}$$

8.4 Set operations

8.4.1 Set assertions

```
bool disjoint(const triplex<T>& a, const triplex<T>& b);
bool proper_subset(const triplex<T>& a, const triplex<T>& b);
bool subset(const triplex<T>& a, const triplex<T>& b);
bool proper_superset(const triplex<T>& a, const triplex<T>& b);
bool superset(const triplex<T>& a, const triplex<T>& b);
bool overlap(const triplex<T>& a, const triplex<T>& b);
```

`disjoint(const triplex<T>& a, const triplex<T>& b)`
: Determines whether the two given triplexes are disjoint to each other, that is whether they do not have any common set.

`proper_subset(const triplex<T>& a, const triplex<T>& b)`
: Determines whether A is a proper subset of B, that is $A \subset B$.

`subset(const triplex<T>& a, const triplex<T>& b)`
: Determines whether A is a subset of B, that is $A \subseteq B$.

`proper_superset(const triplex<T>& a, const triplex<T>& b)`
: Determines whether A is a proper superset of B, that is $A \supset B$.

`superset(const triplex<T>& a, const triplex<T>& b)`
: Determines whether A is a superset of B, that is $A \supseteq B$.

`overlap(const triplex<T>& a, const triplex<T>& b)`
: Determines whether the ranges of A and B overlap to any degree.

8.5 Specializations

8.5.1 `std::hash` - Standard hashing

The standard hash function `std::hash` is implemented for `triplex<T>`.

8.5.2 `std::operator«` - Output stream

The output stream operator `std::operator«` is implemented by default for `triplex<T>`.

8.5.3 `std::operator»` - Input stream

The input stream operator `std::operator»` is implemented by default for `triplex<T>`. If an error occurs during parsing the `failbit` of the `std::istream` will be set, which may throw the `std::ios_base::failure` exception if it has been enabled.

9 Pair Arithmetic

9.1 General Description

For the C++ programming language family the package is mainly implemented as a templated class, with the exception of comparison functions which are implemented as procedural function declarations.

9.1.1 Header Files

All function prototypes and data structures are declared in the `pair.hpp` header file, which in itself includes a couple further header files that contain all subsystems of the interval package.

9.2 `pair<`*T*`>` - Class Reference

The `pair<`*T*`>` class represents a pair arithmetic number of the given template type *T*. The class checks that the latter type *T* matches one of the following types by a `static_assert`, this is to ensure that the class only gets instantiated for types that are actually implemented. The following types are supported:

- `float`
- `double`
- `long double`

9.2.1 Public Member Variables

`low`
: Main value of the pair.

`high`
: Remainder value of the pair.

9.2.2 Constructors

```
constexpr pair() = default;
constexpr pair(const pair<T>&) = default;
constexpr pair(const std::pair<T,T>& o);
constexpr pair(const T& a);
constexpr pair(const T& a, const T& b);
constexpr pair(const T v[2]);
```

These are the constructors of the `pair<T>` class. All constructors with the `constexpr` declaration specifier are implemented in inline.

`pair()`
: Default constructor, leaves all values uninitialized.

`pair(const pair<T>&)`
: Default copy constructor.

`pair(const T&, const T&)` **and** `pair(const T a[2])`
: Assignment constructor, assigns main value and remainder value, respectively.

`pair(const T&)`
: Assignment constructor, initializes the pair as a singleton of the given value.

`pair(std::pair<T,T> &)`
: Assignment constructor, initializes the main and remainder to the `first` and `second` value of the `std::pair`, respectively.

9.2.3 Assignment

```
constexpr pair<T>& operator=(const pair<T>&) = default;
```

These member functions assign the values of the pair.

`operator=(const pair<T>&)`
: Standard one-to-one assignment.

9.2.4 Conversion functions

```
std::pair<T,T> as_pair() const;
interval<T> as_interval() const;
kaucher_interval<T> as_kaucher_interval() const;
```

These member functions convert the current instance of the `pair<T>` class into other formats.

`as_pair()`
: Convert the current interval to a `std::pair` with the `first` and `second` members set to the main- and remainder value, respectively.

`as_interval()`
: Convert the pair to a classic interval.

`as_kaucher_interval()`
: Convert the pair to a kaucher interval.

9.2.5 Comparison Operators

```
bool operator==(const pair<T>& o) const;
bool operator!=(const pair<T>& o) const;
bool operator==(const std::pair<T,T>& o) const;
bool operator!=(const std::pair<T,T>& o) const;
```

These operators check the equality or inequality of the current pair object to another pair object.

The version of the `std::pair` assumes the `first` element contains the main part and `second` part contains the remainder.

9.2.6 Normalization

```
pair<T> normalized() const;
pair<T>& normalize();
```

These functions normalize the pair, that is ensure that for a pair the *main part* contains the significant value and the *remainder part* contains the insignificant remainder.

`normalized()`
: Returns a copy of the current pair object but in a normalized state.

`normalize()`
: Normalizes the current pair object and returns a reference to it.

9.2.7 Status functions

```
bool is_nan() const;
```

These member functions provide information over the current state of the pair.

`is_nan()`
: Whether the pair is not-a-number.

9.2.8 Arithmetic Operators

```
pair<T> operator-() const;

pair<T> operator+(const pair<T>& o) const;
pair<T> operator-(const pair<T>& o) const;
pair<T> operator*(const pair<T>& o) const;
pair<T> operator/(const pair<T>& o) const;

pair<T>& operator+=(const pair<T>& o);
pair<T>& operator-=(const pair<T>& o);
pair<T>& operator*=(const pair<T>& o);
pair<T>& operator/=(const pair<T>& o);

pair<T> operator+(const T o) const;
pair<T> operator-(const T o) const;
pair<T> operator*(const T o) const;
pair<T> operator/(const T o) const;

pair<T>& operator+=(const T o);
pair<T>& operator-=(const T o);
pair<T>& operator*=(const T o);
pair<T>& operator/=(const T o);
```

These operators execute the respective equivalent C++ standard arithmetic operation on the `pair<T>`.

9.2.9 Evaluation

```
constexpr T eval() const;
```

This function re-evaluates the pair as a single number, by attempting to add the remainder part to the main part and returning the result.

9.3 Specializations

9.3.1 `std::operator«` - Output stream

The output stream operator `std::operator«` is implemented by default for `pair<T>`.

9.3.2 `std::operator»` - Input stream

The input stream operator `std::operator»` is implemented by default for `pair<T>`. If an error occurs during parsing the `failbit` of the `std::istream` will be set, which may throw the `std::ios_base::failure` exception if it has been enabled.

10 Mathematical Constants

10.1 Varadic templates

The following constants are defined as `interval<T>` types in the `constants.hpp` header file for all supported floating point types.

`pi_v`
: π constant as a varadic template.

`invpi_v`
: $1/\pi$ constant as a varadic template.

`e_v`
: e constant as a varadic template.

`log2e_v`
: $\log_2(e)$ constant as a varadic template.

`log10e_v`
: $\log_{10}(e)$ constant as a varadic template.

`ln2e_v`
: $\ln_2(e)$ constant as a varadic template.

`ln2e_v`
: $\ln_{10}(e)$ constant as a varadic template.

`sqrt2_v`
: $\sqrt{2}$ constant as a varadic template.

`sqrt3_v`
: $\sqrt{3}$ constant as a varadic template.

10.2 `numbers<T>` - Class Reference

The following class provides read-only constants similar to `std::numbers`. It supports all `T` types that are also supported by the `interval<T>` class.

10.2.1 Public Member Variables

`pi`
: π constant.

`invpi`
: $1/\pi$ constant.

`e`
: e constant.

`log2e`
: $\log_2(e)$ constant.

`log10e`
: $\log_{10}(e)$ constant.

`ln2e`
: $\ln_2(e)$ constant.

`ln2e`
: $\ln_{10}(e)$ constant.

`sqrt2`
: $\sqrt{2}$ constant.

`sqrt3`
: $\sqrt{3}$ constant.

11 Floating Point Utilities

11.1 Header File

All functions and classes defined are contains in the header file `fpu.hpp`.

11.2 `fpuenv` - Class Reference

The class `fpuenv` allows for manipulating the floating point environment.

11.2.1 Member Types

```
enum round_mode {
  round_down       = 1,
  round_up         = 2,
  round_tonearest  = 3,
  round_towardzero = 4,
  round_error      = 99
};
```

`round_down`
: Round downwards.

`round_up`
: Round upwards.

`round_tonearest`
: Round toward the nearest number.

`round_towardzero`
: Round toward towards zero.

`round_error`
: Error if the rounding type is unknown or an error occurred.

11.2.2 Member Functions

```
virtual void set_round(enum round_mode md) = 0;
virtual enum round_mode get_round() const = 0;
```

These functions get and set the rounding mode.

`set_round(enum round_mode)`
: Set the rounding mode to the specified mode.

`get_round()`
: Get the current active rounding mode.

11.2.3 Implementations

The following two implementations are provided:

`fpuenv_real` This implementation manipulates the floating point units in the processor.

`fpuenv_dummy` This implementation does nothing.

11.3 scoped_rounding_mode<T> - Class Reference

This class captures the current rounding mode upon initialization and restores it in the destructor.

11.3.1 Constructors

`scoped_rounding_mode()`

Capture the current rounding mode. If the given template type of the class `T` is $\mathbb{R}$, the implementation will use `fpuenv_real`, otherwise the dummy `fpuenv_dummy` will be used see **??**.

11.3.2 Member Functions

```
void restore();

void round_down();
void round_up();
void round_tonearest();
void round_towardzero();
```

The following functions manipulate the floating point rounding mode.

`restore()`
: Restores the rounding mode to the initial mode set in the constructor.

`round_down()`
: Set rounding mode to round down.

`round_up()`
: Set rounding mode to round up.

`round_tonearest()`
: Set rounding mode to nearest.

`round_towardzero()`
: Set rounding mode towards zero.