

INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen
Oettingenstraße 67, D-80538 München

————— **LMU**
Ludwig—————
Maximilians—
Universität___
München_____

Fuzzy Time Intervals – The FuTI-Library

Hans Jürgen Ohlbach

Technical Report, Computer Science Institute, Munich, Germany
<http://www.pms.informatik.uni-muenchen.de/publikationen>
Forschungsbericht/Research Report PMS-FB-2005-26, April 2005

Fuzzy Time Intervals – The FuTI–Library

Hans Jürgen Ohlbach
Institut für Informatik, Universität München
email: ohlbach@lmu.de

January 10, 2006

Abstract

The FuTI–library is a collection of classes and methods for representing and manipulating fuzzy time intervals. The report consists of three parts. The mathematical theory is described in the first part. The concrete representation of fuzzy time intervals as polygons, together with the algorithms for operating on these polygons is presented in the second part. Finally, the third part contains a short description of the application interface. FuTI is an open source C++ library [5].

Contents

| | | |
|----------|--|-----------|
| 1 | Motivation and Introduction | 1 |
| 2 | The Mathematics of Fuzzy Time Intervals | 2 |
| 2.1 | Fuzzy Time Intervals | 2 |
| 2.2 | Scalar Properties of Fuzzy Time Intervals | 6 |
| 2.3 | Functions Operating on Fuzzy Time Intervals | 7 |
| 2.4 | Set Operators for Fuzzy Intervals | 9 |
| 2.5 | Hull Operators for Fuzzy Intervals | 12 |
| 2.6 | Basic Unary Transformations | 13 |
| 3 | Data Structures and Algorithms | 19 |
| 3.1 | Points | 21 |
| 3.2 | Fuzzy Time Intervals | 23 |
| 3.2.1 | Representation and Construction | 24 |
| 3.2.2 | Basic Features of Fuzzy Intervals | 26 |
| 3.2.3 | Regions | 28 |
| 3.2.4 | Point–Interval and Interval–Interval Relations | 29 |
| 3.2.5 | Hull Operators | 30 |
| 3.2.6 | Basic Unary Transformations | 31 |
| 3.2.7 | Y-Function Based Unary Transformations | 33 |
| 3.2.8 | Y-Function Based Binary Transformations | 34 |
| 3.3 | Integration over Multiplied Intervals | 35 |
| 4 | Summary | 38 |
| A | The FuTI-interface | 40 |
| A.1 | Points | 40 |
| A.2 | Intervals | 41 |
| A.3 | Y-Functions | 45 |

1 Motivation and Introduction

Many temporal notions used in everyday life have a deliberate imprecise meaning. For example, if I say in the morning “tonight I’ll go to the disco”, and somebody asks me “will you go to the disco at 8 pm?” I may neither want to say “yes” nor may I want to say “no”. One may argue whether in this case any precise mathematical model of “tonight” is useful at all. There are other cases, however, where a fuzzy

logic model of imprecise notions is definitely helpful. Consider, for example, a database with, say, a cinema timetable. If you query the timetable “give me all performances ending *before* midnight”, do you really want to exclude a performance ending just one minute after midnight? I think, not. One could solve this problem by giving the ‘before’ relation a fuzzy meaning, such that performances ending before midnight get a fuzzy value 1, and performances ending after midnight get a fuzzy value which decreases the later the performance ends. The fuzzy value could then be used to order the answers to the query such that the performances ending after midnight come late in the list.

In this paper the FuTI-library (Fuzzy Temporal Intervals) of data structures and algorithms for representing and manipulating fuzzy temporal notions is described. In the first part the components of the FuTI-library are described in a purely mathematical way, without any commitment to concrete data structures and algorithms. A representation of the fuzzy intervals as polygons with integer coordinates is explained in the second part. All algorithms in FuTI work on these polygons. Finally the concrete interface to the library is listed and explained.

The library is a component of the CTTN-system (Computational Treatment of Temporal Notions) [4], a program for evaluating temporal expressions like ‘three weeks after Easter’. CTTN is currently under development. CTTN contains in particular the specification language GeTS for specifying and working with temporal notions [6]. Many of the language primitives in GeTS are the operations of the FuTI-library. Other language primitives in GeTS use the PartLib-library for representing periodical temporal notions [7]. GeTS is in particular suitable for specifying fuzzy relations between fuzzy time intervals. Therefore FuTI is only one piece in a bigger mosaic. Some of the design decisions in FuTI are motivated by the needs of the GeTS language.

The fuzzy intervals in FuTI are fuzzy subsets of the real values. Therefore they can represent all kinds of things. The main motivation for most of the operations in FuTI, however, comes from their interpretation as *temporal* intervals and relations; and this is the reason for the ‘T’ in FuTI.

2 The Mathematics of Fuzzy Time Intervals

The mathematics of general fuzzy sets [10] has been investigated in great depth. The particular fuzzy sets in FuTI are subsets of the real numbers. On the one hand, this makes things easier. On the other hand, however, it offers a very rich algebraic structure with many different operations and relations. Therefore it is useful to start with an overview of the basic ideas and definitions about fuzzy sets. Some, but not all of them can be found in textbooks about fuzzy sets (see e.g. [2]).

Since FuTI is designed as a library to be used in many different applications, we need to provide a broad spectrum of quite different concepts and operations. I tried to organise them in a meaningful way and to motivate them with temporal notions and operations.

Definition 2.1 (Basic Notions and Notations) *We define some notions and notations about numbers and intervals.*

\mathbb{N} are the integers, \mathbb{R} are the real numbers and $\mathbb{R}^+ \stackrel{\text{def}}{=} \mathbb{R} \cup \{-\infty, +\infty\}$.

$\sup\{s\}$ is the supremum of the set $s \subseteq \mathbb{R}$ and $\inf\{s\}$ is the infimum of the set $s \subseteq \mathbb{R}$.

For an interval $I = [a, b] \subseteq \mathbb{R}$ let $|I| \stackrel{\text{def}}{=} b - a$ be the length of I . If I consists of several subintervals let $|I|$ be the sum of the length of the subintervals. The same definitions apply if I consists of open or half-open intervals. ■

2.1 Fuzzy Time Intervals

Fuzzy Intervals are usually defined through their membership functions. A membership function maps a base set to a real number between 0 and 1. This “fuzzy value” denotes a kind of degree of membership to a fuzzy set S . For example, the base set may consist of all people on earth, and S may be the set of ‘large persons’. If for the person John the fuzzy value for ‘large persons’ is 1 then John is definitely a large person. If the fuzzy value is 0 then John is definitely not a large person. If, instead, the fuzzy value is, for example, 0.8, then John is quite tall, but not as tall as really large persons.

The base set for fuzzy time intervals is the time axis. In FuTI it is represented by the set \mathbb{R} of real numbers. Real numbers allow us to model the continuous time flow which we perceive in our life. A fuzzy time interval in FuTI is now a fuzzy subset of the real numbers.

Definition 2.2 (Fuzzy Time Intervals) *A fuzzy membership function in FuTI is a total function $f : \mathbb{R} \mapsto [0, 1]$ which need not be continuous, but it must be integratable.*

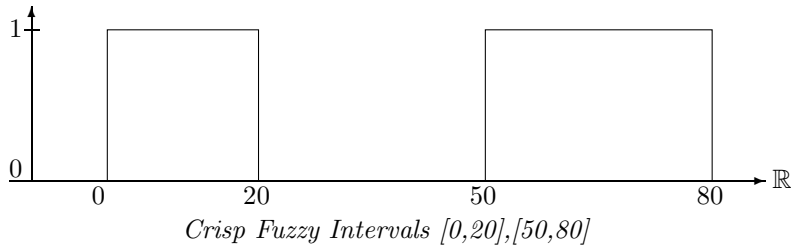
The fuzzy interval I_f that corresponds to a fuzzy membership function f is

$$I_f \stackrel{\text{def}}{=} \{(x, y) \subseteq \mathbb{R} \times [0, 1] \mid y \leq f(x)\}.$$

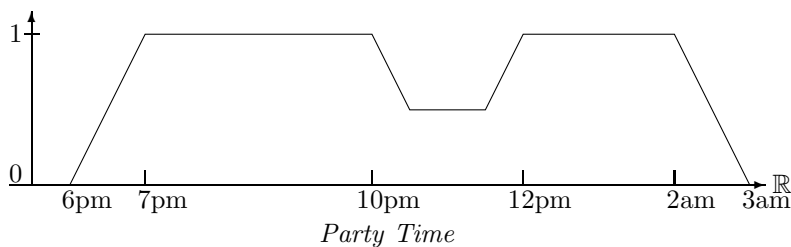
Given a fuzzy interval I we usually write $I(x)$ to indicate the value of the corresponding membership function at point x .

Let $F_{\mathbb{R}}$ be the set of fuzzy time intervals. ■

This definition comprises single or multiple crisp intervals like this:

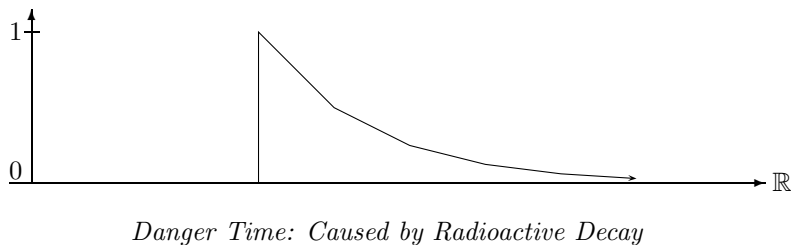


It also comprises finite fuzzy intervals like this one:

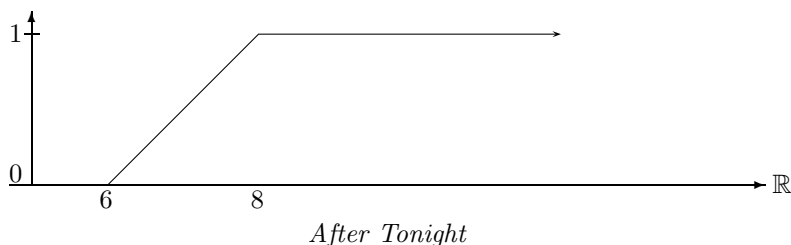


This set may represent a particular party time, where the first guests arrive at 6 pm. At 7 pm all guests are there. Half of them disappear between 10 and 12 pm (because they go to the pub next door to watch an important soccer game). Between 12 pm and 2 am all of them are back. At 2 am the first ones go home, and finally at 3 am all are gone. The fuzzy value indicates in this case the number of people at the party.

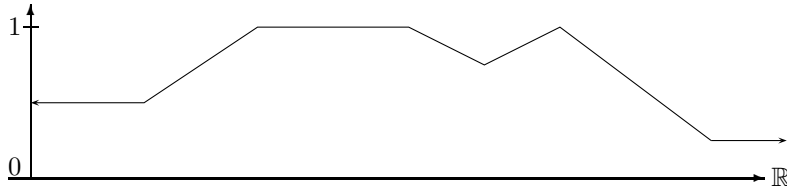
Fuzzy intervals may also be infinite.



More realistic examples of infinite fuzzy time intervals are intervals where the fuzzy value remains constant after a while. For example, the term ‘after tonight’ may be represented by a fuzzy interval which rises from fuzzy value 0 at 6 pm until fuzzy value 1 at 8 pm and then remains 1 ad infinitum.



The more general case are infinite fuzzy intervals which may be infinite at one or two sides, but where the membership function becomes constant, but not necessarily 1, after a while.



Infinite Fuzzy Interval with Mostly Constant Membership Function

Remark 2.3 The representation of a fuzzy interval as a subset of $\mathbb{R} \times [0, 1]$ means that one can apply the standard set operators \cup (union), \cap (intersection) and \setminus (set difference) to fuzzy intervals. The standard set theoretic definition of intersection and set difference, however, need not yield fuzzy intervals any more. Therefore there are other, more appropriate definitions of the set operations on fuzzy intervals (see Section 2.4).

Definition 2.4 (Height of a Fuzzy Interval (sup, inf)) For a fuzzy interval $I \in F_{\mathbb{R}}$ let

$$\sup(I) \stackrel{\text{def}}{=} \sup\{I(x) \mid x \in \mathbb{R}\}$$

be the height (largest fuzzy value, supremum) of I and let

$$\inf(I) \stackrel{\text{def}}{=} \inf\{I(x) \mid x \in \mathbb{R}\}$$

be the smallest fuzzy value, (infimum) of I ■

$\sup(I)$ is usually, but not necessarily, 1 for nonempty fuzzy time intervals. If $\sup(I) = 0$ then, however, I must be empty.

Fuzzy time intervals may be quite complex structures with many different characteristic features. The simplest ones are *core* and *support*. The core is the subset of \mathbb{R} where the fuzzy value is 1, and the support is the subset of \mathbb{R} where the fuzzy value is non-zero. In addition, we define the *kernel* as the subset of \mathbb{R} where the fuzzy value is *not* constant ad infinitum. Finally, *maxRegion* is the interval between the first and last point where the fuzzy value is maximal.

Definition 2.5 (Core, Support, Kernel and MaxRegion)

The core $C(I)$ of a fuzzy set I is the subset of \mathbb{R} where the membership function is 1:

$$C(I) \stackrel{\text{def}}{=} \{x \in \mathbb{R} \mid I(x) = 1\}.$$

The core of I can be empty even if I itself is not empty.

The support $S(I)$ of I is the subset of \mathbb{R} where the membership function is nonzero:

$$S(I) \stackrel{\text{def}}{=} \{x \in \mathbb{R} \mid I(x) \neq 0\}.$$

If $S(I) = \emptyset$ then $I = \emptyset$.

The kernel $K(I)$ of I is the smallest interval $[a, b] \subseteq \mathbb{R}^+$ such that there are $I_1 \in [0, 1]$ and $I_2 \in [0, 1]$ with $I(x) = I_1$ for all $x < a$ and $I(x) = I_2$ for all $x > b$.

$K(I)$ can be empty, finite or infinite. If $K(I) = \emptyset$ then I is either empty or infinite with at most two different fuzzy values.

The maxRegion $M(I)$ of I is the interval between the first and last maximal points, i.e.

$$M(I) \stackrel{\text{def}}{=} \begin{cases} [\inf\{x \mid I(x) = \sup(I)\}, \sup\{x \mid I(x) = \sup(I)\}] & \text{if } \sup(I) \neq 0 \\ \emptyset & \text{otherwise} \end{cases}$$

For $O \in \{C, S, K, M\}$ let $O^{\square}(I)$ be the (crisp) fuzzy interval such that $C(O^{\square}(I)) = S(O^{\square}(I)) = O^{\square}(I)$.

For $O \in \{C, S, K, M\}$ let

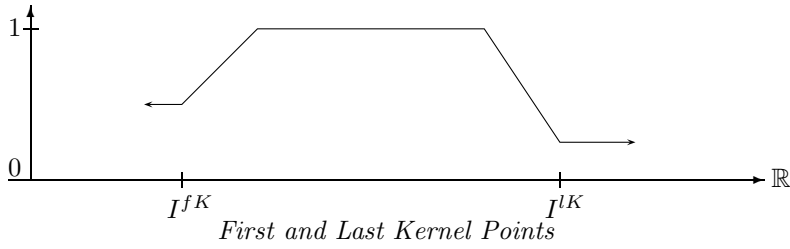
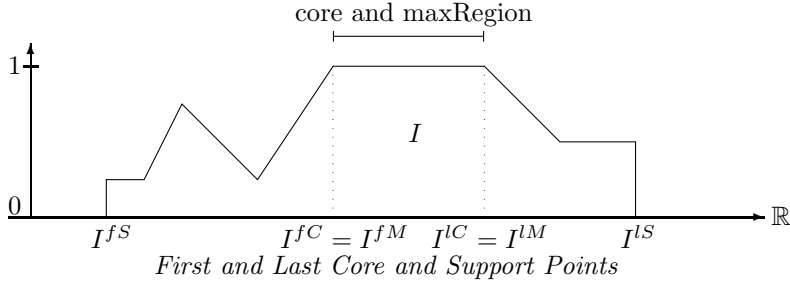
$$I^f O \stackrel{\text{def}}{=} \begin{cases} \inf\{x \mid O(I)(x) \neq 0\} & \text{if } O(I) \neq \emptyset \\ +\infty & \text{otherwise} \end{cases}$$

be the first O -point of I and let

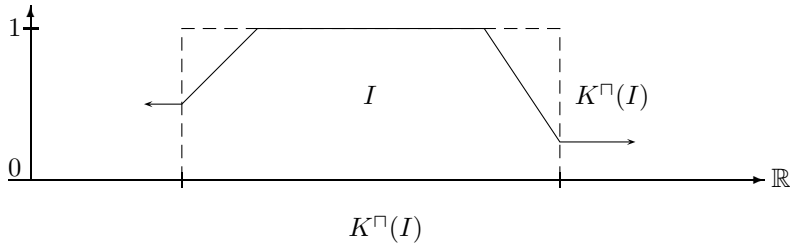
$$I^{lO} \stackrel{\text{def}}{=} \begin{cases} \sup\{x \mid O(I)(x) \neq 0\} & \text{if } O(I) \neq \emptyset \\ +\infty & \text{otherwise} \end{cases}$$

be the last O -point of I . ■

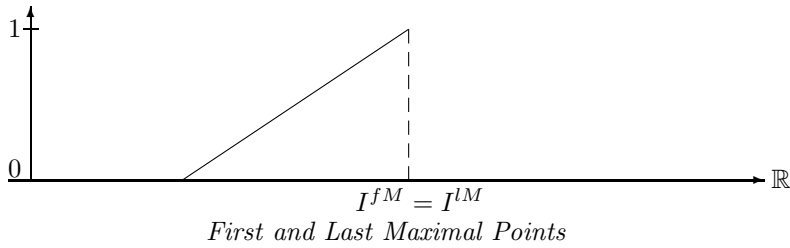
I^{fC} and I^{lC} are the first/last core points.
 I^{fS} and I^{lS} are the first/last support points.
 I^{fK} and I^{lK} are the first/last kernel points.
 I^{fM} and I^{lM} are the first/last maxRegion points.



The next picture shows the kernel of the same interval I as crisp interval $K^\square(I)$.



The next picture shows an example where $I^{fM} = I^{lM}$ and where $\sup(I)$ is really the supremum, and not the maximum because $I(I^{fM}) = 0$.



If $\sup(I) = 1$ then $I^{fM} = I^{fC}$ and $I^{lM} = I^{lC}$. If, however, $\sup(I) < 1$ then I^{fM} and I^{lM} have nothing to do with the core of I .

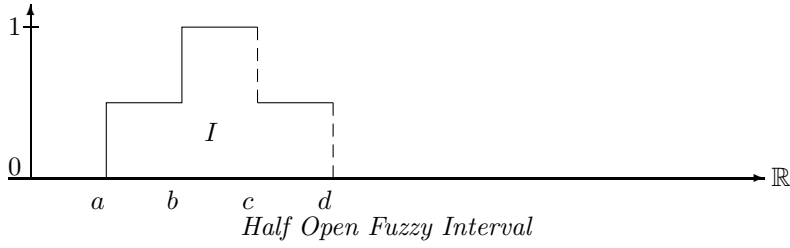
Fuzzy time intervals with finite kernel are of particular interest because although they may be infinite, they can easily be implemented with finite data structures. Therefore we give them an extra name.

Definition 2.6 (Fuzzy Time Intervals with Finite Kernel) Let $F_{\mathbb{R}}^f$ be the set of fuzzy time intervals (Def. 2.2) with finite kernel (Def. 2.5). ■

Fuzzy time intervals which are in fact crisp intervals can now be characterised very easily as intervals where core and support are the same.

Definition 2.7 (Crisp Interval) A crisp interval is a fuzzy interval I (Def. 2.2) such that $C(I) = S(I)$ (Def. 2.5). ■

Remark 2.8 (Openness and Closedness) Ordinary intervals can be open or closed. A similar distinction can also be made for fuzzy intervals. As an example, consider the following fuzzy interval I :



If we have $I(a) = 0.5$, $I(b) = 1$, but $I(c) = 0.5$ and $I(d) = 0$ then I is closed at a and b and open at c and d .

We sometimes indicate the open sides of fuzzy intervals with dashed lines. ■

Half-open intervals are of particular interest for time intervals. Consider, for example, the two intervals ‘this week’s Monday’ and ‘this week’s Tuesday’. If both intervals are represented as closed intervals then midnight belongs to Monday and Tuesday. This is not what we usually want. Therefore it is more realistic to represent the intervals as half-open intervals such that midnight belongs to either Monday or Tuesday, but not to both days. As a convention, we assume that (finite) time intervals are half open at the positive side: their structure is $[a, b[$. Midnight would then belong to Tuesday. This has some consequences for the algorithms (cf. Remark 3.24).

2.2 Scalar Properties of Fuzzy Time Intervals

Fuzzy time intervals can be measured in various ways. Besides the size, which is the integral over the membership function, one can locate the position of the core, support and kernel. One can also measure the maximal fuzzy value. This should, but need not be 1. Furthermore, one can split the interval into parts of equal size (the first half and the second half etc.), and locate their boundaries. Let us start with the size of the interval.

Definition 2.9 (Size) For $a, b \subseteq \mathbb{R}^+$ and a fuzzy time interval I let

$$|I|_a^b \stackrel{\text{def}}{=} \int_a^b I(x) dx. \quad |I| \stackrel{\text{def}}{=} |I|_{-\infty}^{+\infty} \text{ is the size of } I.$$

■

If I is a crisp interval then $|I|$ yields the length of I in the usual sense.

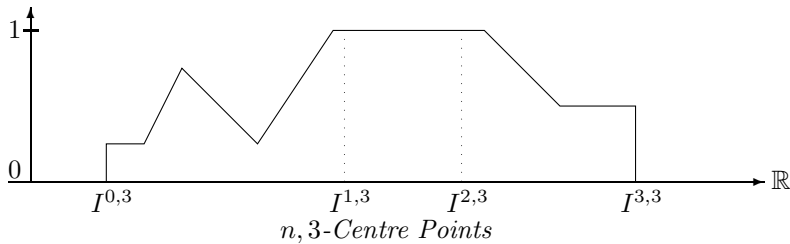
Centre Points

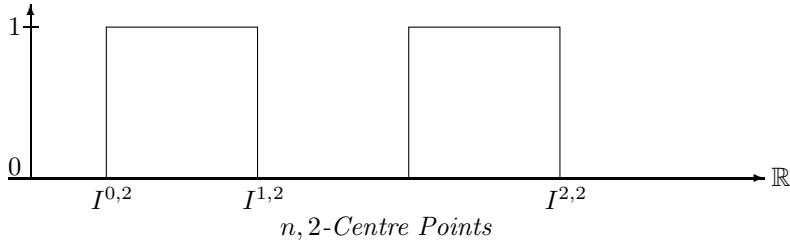
The n, m -centre points defined below are used to express temporal notions like ‘the first half of the year’, or ‘the second quarter of the year’ or more exotic expressions like ‘the 25th 49th of the weekend’ etc. The notion of n, m -centre points makes only sense for finite intervals.

Definition 2.10 (n, m -Centre Points) Let $I \in F_{\mathbb{R}}$ with $|I| < \infty$. For two integers $m > 1$ and $0 \leq n \leq m$ we define the n, m -centre points

$I^{n,m} \stackrel{\text{def}}{=} x_n$ where x_n is a minimal \mathbb{R} -value in a sequence $I^{fS} = x_0, \dots, x_m = I^{lS}$ with $|I|_{x_0}^{x_1} = |I|_{x_1}^{x_2} = \dots = |I|_{x_{m-1}}^{x_m} = |I|/m$. ■

Examples: $I^{1,2}$ splits I in two halves of the same size. $I^{1,3}$ indicates a split of I into three parts of the same size. $I^{1,3}$ is the boundary of the first third, $I^{2,3}$ is the boundary of the second third.





Middle Points:

The middle point between the centre points $I^{n,m}$ and $I^{n+1,m}$ is just $I^{2n+1,2m}$. For example, the middle point in the first half of I is $I^{1,4}$ and the middle point in the second half is $I^{3,4}$.

Components

Fuzzy time intervals can consist of several different components. A component is a sub-interval of a fuzzy interval such that the left and right end is either the infinity, or the membership function drops down to 0. Let $Cmp(I)$ be the list of components of I . $nComponents(I)$ is the number of components of I . $component(I, k)$ is the k^{th} component of I .

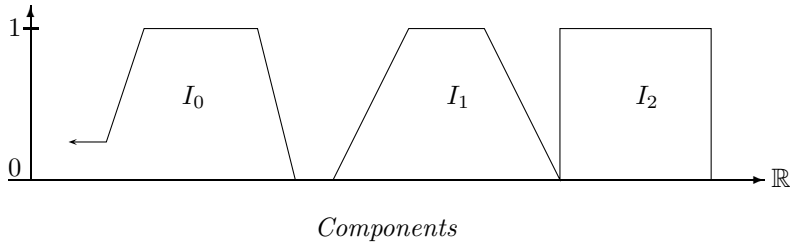
Definition 2.11 (Components) Let $I \in F_{\mathbb{R}}$. The components I_0, \dots, I_n of I are fuzzy time intervals such that: (i) $I_k(x) = I(x)$ for all $x \in S(I_k)$ and $0 \leq k \leq n$, and (ii) for all $k \in \{1, \dots, n-1\}$: $(\lim_{x \rightarrow I_k^{fs}} I(x) = 0 \text{ or } \lim_{I_k^{fs} \leftarrow x} I(x) = 0)$ and $(\lim_{x \rightarrow I_k^{ls}} I(x) = 0 \text{ or } \lim_{I_k^{ls} \leftarrow x} I(x) = 0)$.

Let $nComponents(I)$ be the number of components of I .

Let $component(I, k)$ be the k^{th} component of I . ■

The definition is quite complicated because we want to count as separate components parts of fuzzy time intervals where the membership function drops down to 0 at just one single point.

Example:



2.3 Functions Operating on Fuzzy Time Intervals

Time intervals usually don't appear from nowhere, but they are constructed from other time intervals. We distinguish two ways of constructing new fuzzy time intervals, first by means of *y-functions* and then by means of *interval operators*. Y-functions map fuzzy values to fuzzy values. They can therefore be used to construct a new interval from a given one by applying the y-function point by point to the membership function values.

Interval operators are more general construction functions. They take one or more fuzzy time intervals and construct a new one out of them.

Definition 2.12 (Y-Functions)

$Y-FCT^{n, \text{def}} \{f : [0, 1]^n \mapsto [0, 1]\}$ is the set of n -place y-functions.

They map fuzzy values to fuzzy values.

$$Y-FCT^{\text{def}} \equiv \bigcup_{n \geq 0} Y-FCT^n. \quad \blacksquare$$

Definition 2.13 (Interval Operators)

$I-OPS^{n, \text{def}} \{g : F_{\mathbb{R}}^n \mapsto F_{\mathbb{R}}\}$ is the set of n -place interval operators.

They map fuzzy intervals to fuzzy intervals.

$$I-OPS^{\text{def}} \equiv \bigcup_{n \geq 0} I-OPS^n. \quad \blacksquare$$

Every y-function can be used to construct a new fuzzy time interval from given ones by applying the y-function to the fuzzy values.

Definition 2.14 (Associated Interval Operators) If $f \in Y-FCT^n$ is a y -function then $g_f \in I-OPs^n$ defined by $g_f(I_1, \dots, I_n)(x) \stackrel{\text{def}}{=} f(I_1(x), \dots, I_n(x))$ is the associated interval operator. ■

Linear Y-Functions

A small, but important class of y -functions are *linear* y -functions. They are important firstly because very natural operators, like standard complement, intersection and union of fuzzy time intervals can be described with linear y -functions. Secondly they are important because they allow us to transform intervals represented by polygons in a very efficient way: only the vertices and intersection points of the polygons need to be transformed.

The main characterisation of linear y -functions is therefore that they map non-intersecting straight line segments to straight line segments, and not to curves.

Definition 2.15 (Linear Y-Function) A y -function $f \in Y-FCT^n$ is linear if the mapping $f'((x, y_1), \dots, (x, y_n)) \stackrel{\text{def}}{=} (x, f(y_1, \dots, y_n))$ maps non-intersecting line segments $(x_1, z_{11}) - (x_2, z_{12}), \dots, (x_1, z_{n1}) - (x_2, z_{n2})$ to a line segment $(x_1, f(z_{11}, \dots, z_{n1})) - (x_2, f(z_{12}, \dots, z_{n2}))$. ■

One-place linear y -functions can be characterised in the following way:

Proposition 2.16 (Characterisation of One-Place Linear y -Functions) A one-place y -function f is linear if and only if $f(y) = f(0) + (f(1) - f(0)) \cdot y$ holds.

Proof: Suppose f is linear. We take the straight line segment between $(0, 0)$ and $(1, 1)$. The mapping $f'(x, y) \stackrel{\text{def}}{=} (x, f(y))$ maps this line segment to a line segment between $(0, f(0))$ and $(1, f(1))$. Therefore

$$\begin{aligned} f(y) &= f(0) + \frac{f(1) - f(0)}{1 - 0} \cdot (y - 0) \quad (\text{line equation}) \\ &= f(0) + (f(1) - f(0)) \cdot y \end{aligned}$$

The other direction of the proof is trivial. ■

An example for a one-place linear y -function is the standard negation $n(y) = 1 - y$.

The characterisation of two-place linear y -functions is a bit trickier.

Proposition 2.17 (Characterisation of Two-Place Linear y -Functions) A two-place y -function f is linear if and only if the following condition holds:

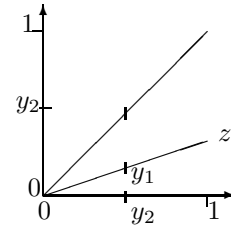
$$f(y_1, y_2) = \begin{cases} f(0, 0) + (f(y_1/y_2, 1) - f(0, 0)) \cdot y_2 & \text{if } y_1 \leq y_2 \\ f(0, (y_1 - y_2)/(1 - y_2)) + (f(1, 1) - f(0, (y_1 - y_2)/(1 - y_2))) \cdot y_2 & \text{otherwise} \end{cases}$$

Proof: Suppose f is linear.

We consider the case $y_1 \leq y_2$ first. To this end we take the straight line segment between $(0, 0)$ and $(1, 1)$. The line equation for this line is just $y = x$. Now take an arbitrary $y_2 \in [0, 1]$ and an arbitrary $y_1 \leq y_2$. The line equation for the line segment starting at $(0, 0)$ and crossing (y_2, y_1) is $y = (y_1 - 0)/(y_2 - 0) \cdot x$. For $x = 1$ we get $z = y_1/y_2$.

Since f is linear we have

$$\begin{aligned} f(y_1, y_2) &= f(0, 0) + \frac{f(z, 1) - f(0, 0)}{1 - 0} \cdot y_2 \\ &= f(0, 0) + (f(\frac{y_1}{y_2}, 1) - f(0, 0)) \cdot y_2 \end{aligned}$$

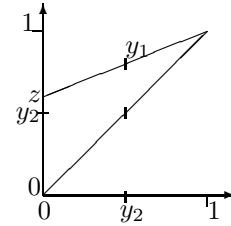


Now consider the case $y_1 \geq y_2$.

The line starting at $(1, 1)$ and crossing (y_2, y_1) crosses the y -axis at $z = (y_1 - y_2)/(1 - y_2)$.

Since f is linear we have

$$\begin{aligned} f(y_1, y_2) &= f(0, z) + \frac{f(1, 1) - f(0, z)}{1 - 0} \cdot y_2 \\ &= f(0, \frac{y_1 - y_2}{1 - y_2}) + (f(1, 1) - f(0, \frac{y_1 - y_2}{1 - y_2})) \cdot y_2 \end{aligned}$$



The other direction, showing that the two conditions imply linearity, is again straightforward. ■

Simple examples for linear two-place y -functions are the minimum and maximum functions. The minimum function is used to realize standard intersection of two fuzzy time intervals, and the maximum function is used to realize standard union of two fuzzy time intervals.

2.4 Set Operators for Fuzzy Intervals

For ordinary intervals there are the standard Boolean set operators: complement, intersection, union etc. These are uniquely defined. There is no choice. Unfortunately, or fortunately, because it gives you more flexibility, there are no such uniquely defined set operators for fuzzy intervals. Set operators are essentially transformations of the membership functions, and there are lots of different ones. One has tried to classify them such that essential properties of the Boolean set operators are preserved.

Complement of Fuzzy Time Intervals

The complement operator for fuzzy time intervals is to be understood in the following sense: if for a particular point x the probability to belong to a set S is y then the probability to belong to the complement of S is $n(y)$ where n is a so called *negation function*.

Definition 2.18 (Negation Function) A function $n \in Y\text{-FCT}^1$ satisfying the conditions

- $n(0) = 1$ and $n(1) = 0$;
- n is non-increasing, i.e. $\forall x, y \in [0, 1] : x \leq y \Rightarrow n(x) \geq n(y)$

is called a negation function.

Let NF be the set of all negation functions. ■

Example 2.19 (Standard Negation and λ -Complement) The function

$$n(y) \stackrel{\text{def}}{=} 1 - y$$

is the standard fuzzy negation.

For any $\lambda > -1$ the so called λ -complement is the function

$$n_\lambda(y) \stackrel{\text{def}}{=} \frac{1 - y}{1 + \lambda y}.$$

Both functions n and n_λ are negation functions in the sense of Def. 2.18.

$N(I)(x) \stackrel{\text{def}}{=} n(I(x))$ is the standard complement operator.

$N_\lambda(I)(x) \stackrel{\text{def}}{=} n_\lambda(I(x))$ is the λ -complement operator.

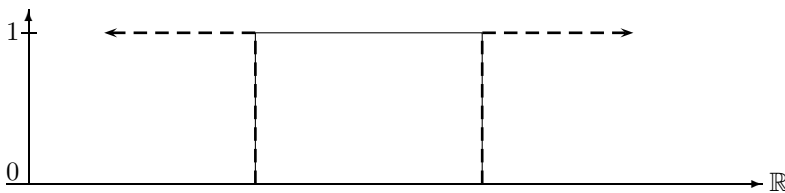
If I is a crisp interval then $N(I) = N_\lambda(I)$ ■

Proposition 2.20 (Idempotency of the negation functions) For every $y \in [0, 1]$ we have for the standard negation $n(n(y)) = y$ and for the λ -complement: $n_\lambda(n_\lambda(y)) = y$.

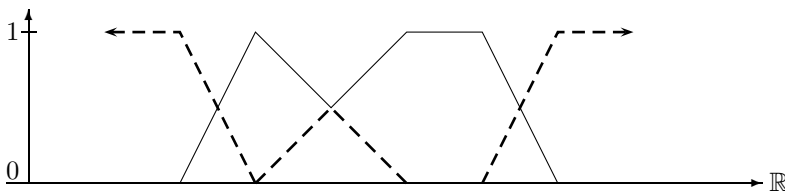
The proof is straightforward. ■

This property need not hold for other negation functions.

We give some examples for standard and λ -complement. The dashed lines indicate the complement.

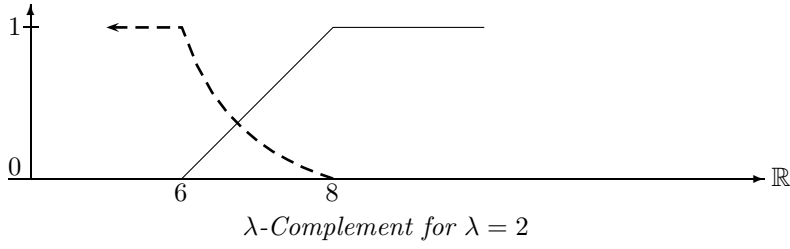


Standard Complement and λ -Complement for a Crisp Interval



Standard Complement for a Fuzzy Interval

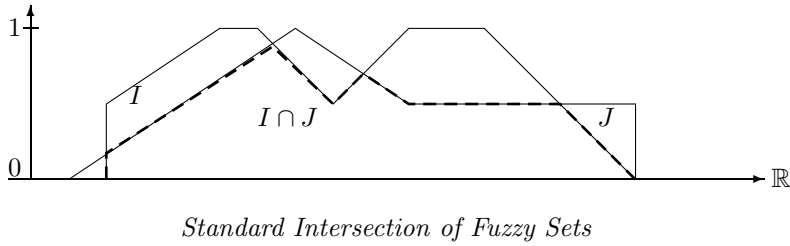
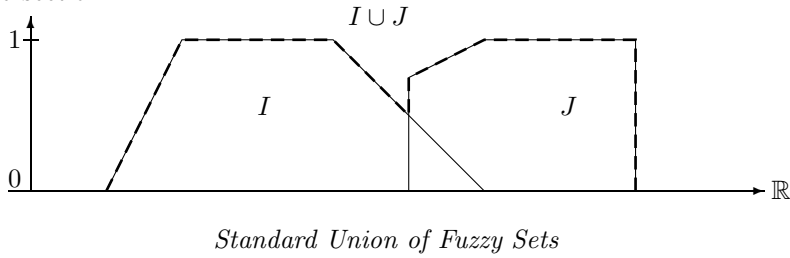
If we define ‘tonight’ as a fuzzy interval, rising from 0 at 6pm to 1 at 8pm, we could use the standard complement for ‘before tonight’. The term ‘long before tonight’ must of course be represented differently to ‘before tonight’. A λ -complement version with $\lambda = 2$ looks as follows:



If λ is increased then the descend from 6 pm till 8 pm becomes steeper. A suitable λ could then in fact mean ‘long before tonight’.

Intersection and Union of Fuzzy Time Intervals

The two figures below show standard union and intersection of fuzzy intervals. Union is obtained by taking the maximum of the two member functions. The minimum of the two member functions yields intersection.



Standard union and intersection, however, is only one particular form of union and intersection. Instead of minimum and maximum one could think of other functions for computing union and intersection. These other functions, so called triangular norms and co-norms, must fulfil certain axioms in order to satisfy our intuition about union and intersection.

Definition 2.21 (Triangular Norms and Co-norms) A function $T : [0, 1]^2 \mapsto [0, 1]$ is called a triangular norm, or t-norm for short, iff it satisfies the laws T1-T4 below. A function $S : [0, 1]^2 \mapsto [0, 1]$ is called a triangular co-norm, or t-co-norm for short, iff it satisfies the laws S1-S4 below.

- | | | |
|--|------------|--|
| Identity law: | T1: | $\forall x \quad T(x, 1) = x,$ |
| | S1: | $\forall x \quad S(x, 0) = x$ |
| Commutativity: | T2: | $\forall x, y \quad T(x, y) = T(y, x),$ |
| | S2: | $\forall x, y \quad S(x, y) = T(y, x)$ |
| Associativity: | T3: | $\forall x, y, z \quad T(x, T(y, z)) = T(T(x, y), z),$ |
| | S3: | $\forall x, y, z \quad S(x, S(y, z)) = S(S(x, y), z)$ |
| Monotonicity: $\forall x, y, u, v \in [0, 1] \quad x \leq u, y \leq v :$ | T4: | $T(x, y) \leq T(u, v),$ |
| | S4: | $S(x, y) \leq S(u, v)$ |

Triangular norms and co-norms are γ -functions in $Y\text{-FCT}^2$ (Def. 2.12).

Let $TNorm$ be the set of triangular norms and let $TCoNorm$ be the set of triangular co-norms. ■

The triangular norms and co-norms are now turned into interval operators \cap_T and \cup_S :

Definition 2.22 (Intersection and Union) Let $I, J \in F_{\mathbb{R}}$ be two fuzzy intervals. If T is a triangular norm and S a triangular co-norm (Def. 2.21) then

$$(I \cap_T J)(x) \stackrel{\text{def}}{=} T(I(x), J(x)) \quad \text{and} \quad (I \cup_S J)(x) \stackrel{\text{def}}{=} S(I(x), J(x))$$

are the intersection and union operators on the fuzzy intervals. ■

Example 2.23 (Standard Fuzzy Intersection and Union) The function \min is a triangular norm and the function \max is a triangular co-norm. Therefore

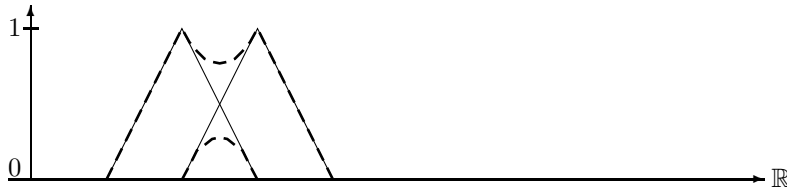
$$(I \cap_{\min} J)(x) \stackrel{\text{def}}{=} \min(I(x), J(x)) \quad \text{and} \quad (I \cup_{\max} J)(x) \stackrel{\text{def}}{=} \max(I(x), J(x))$$

are the standard fuzzy intersection and union operators. ■

A particular class of triangular norms and co-norms, together with a negation function, is the *Hamacher family*.

Example 2.24 (Hamacher Family) The Hamacher family consists of the following parameterised families of triangular norms and co-norms, and negation functions (λ -complement):

$$\begin{aligned} T_{\gamma}(x, y) &\stackrel{\text{def}}{=} \frac{xy}{\gamma + (1 - \gamma)(x + y - xy)} && \gamma \geq 0 \\ S_{\beta}(x, y) &\stackrel{\text{def}}{=} \frac{x + y + (\beta - 1)xy}{1 + \beta xy} && \beta \geq -1 \\ n_{\lambda}(x) &\stackrel{\text{def}}{=} \frac{1 - x}{1 + \lambda x} && \lambda > -1 \end{aligned}$$



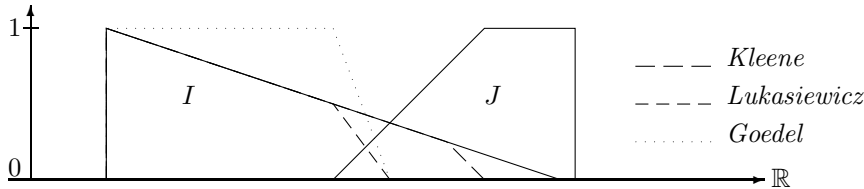
Hamacher Intersection and Union with $\beta = \gamma = 0.5$

Set Difference of Fuzzy Time Intervals Set difference $I \setminus J$ can also be defined by means of y -functions. The following versions are derived from corresponding implication functions:

Definition 2.25 (Set Difference)

- Kleene: $(I \setminus J)(x) \stackrel{\text{def}}{=} \min(I(x), 1 - J(x))$
- Lukasiewicz: $(I \setminus J)(x) \stackrel{\text{def}}{=} \max(0, I(x) - J(x))$
- Goedel: $(I \setminus J)(x) \stackrel{\text{def}}{=} 0$ if $I(x) \leq J(x)$ and $1 - J(x)$ otherwise

Example 2.26 (Set Difference) The following picture shows the difference between the three versions of set difference.

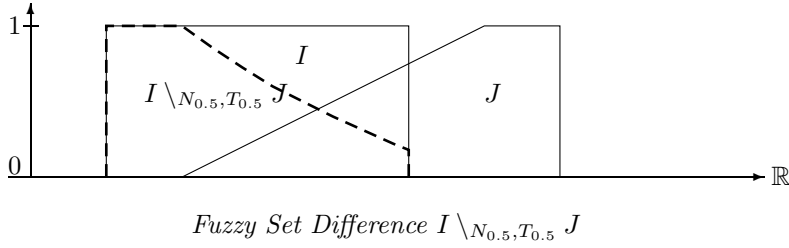


Set Difference $I \setminus J$

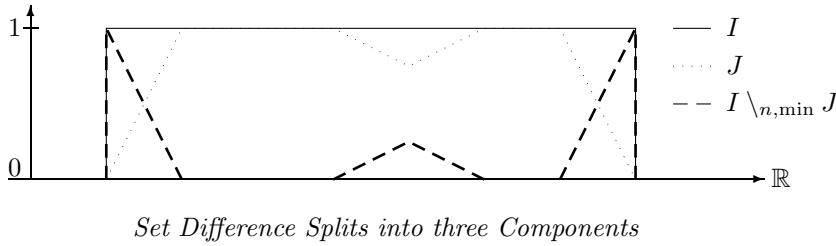
The Kleene version corresponds to the crisp definition of set difference: $I \setminus J = I \cap J^c$ where J^c is the complement of J . This can be generalised by replacing \cap with \cap_T and J^c with a complement operator. ■

Definition 2.27 (Generalised Set Difference) Let N be a complement function and T a t -norm. We define the set difference operator $\setminus_{N,T}$ between two fuzzy intervals I and J as

$$(I \setminus_{N,T} J) \stackrel{\text{def}}{=} I \cap_T N(J)$$



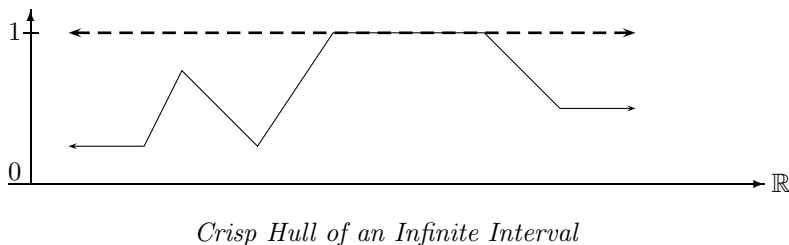
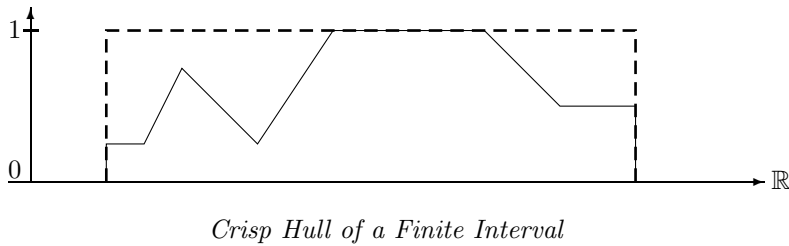
Splitting an interval into two intervals is the worst that can happen for the set difference of two crisp intervals. In the case of fuzzy intervals, the set difference operator can produce arbitrary many disjoint intervals, as the next figure shows.



2.5 Hull Operators for Fuzzy Intervals

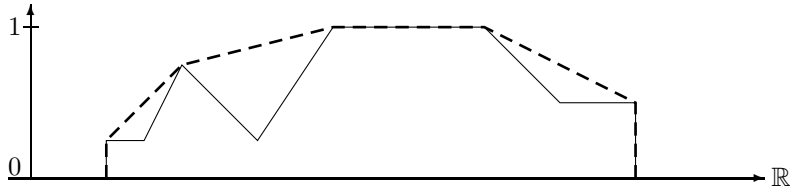
Except for the closed hull of an open interval there is no meaningful notion of a ‘hull’ for a single crisp time interval. It turns out, however, that there are various *hulls* for fuzzy intervals. We define them in the order of information loss. The first notion of a hull, the *crisp hull* loses most information about the interval, whereas the last notion, the *monotone hull* loses the least information. All these notions of a hull coincide for crisp intervals.

Definition 2.28 (Crisp Hull) For an interval $I \in F_{\mathbb{R}}$ let $\text{crispHull}(I)$ be the smallest convex crisp interval containing I . ■

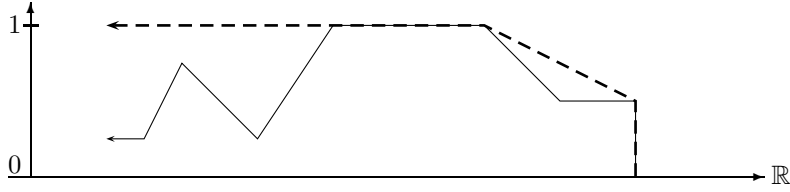


Notice that $\text{crispHull}(I) \neq S^{\square}(I)$ if I consists of several unconnected components.

Definition 2.29 (Convex Hull) The convex hull $\text{convexHull}(I)$ of a fuzzy set I is the smallest convex set containing I . ■



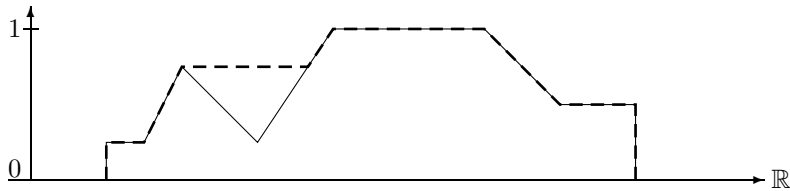
Convex Hull of a Finite Set



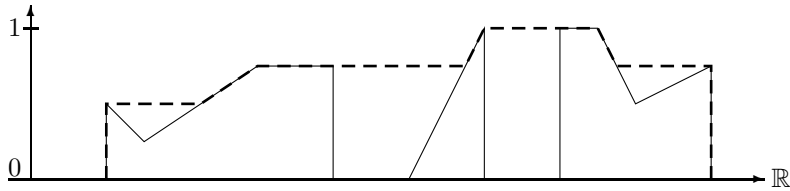
Convex Hull of an Infinite Set

Finally we define the *monotone hull* which loses the least of the structural information about the interval.

Definition 2.30 (Monotone Hull) *The monotone hull $\text{monotoneHull}(I)$ of a fuzzy set I is the smallest monotone fuzzy interval containing I . Monotone means that from left to right the fuzzy values $\text{monotoneHull}(I)(x)$ are rising monotonically to $\text{sup}(I)$, and then falling monotonically again. ■*



Monotone Hull



Monotone Hull of a Fuzzy Interval with Three Components

2.6 Basic Unary Transformations

We now introduce a little library of interval operators. They are used in the GeTS specification language [6] as building blocks, for example, to define fuzzy point–interval and fuzzy interval–interval relations [8].

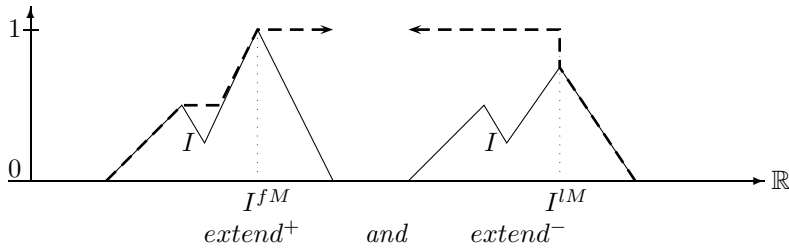
Definition 2.31 (Basic Unary Transformations) *Let $I \in F_{\mathbb{R}}$ be a fuzzy interval. We define the following (parameterised) interval operators:*

$$\begin{aligned}
\text{identity}(I) &\stackrel{\text{def}}{=} I \\
\text{extend}^+(I)(x) &\stackrel{\text{def}}{=} \begin{cases} \sup\{I(y) \mid y \leq x\} & \text{if } x \leq I^{fM} \\ 1 & \text{otherwise} \end{cases} \\
\text{extend}^-(I)(x) &\stackrel{\text{def}}{=} \begin{cases} \sup\{I(y) \mid y \geq x\} & \text{if } x \geq I^{lM} \\ 1 & \text{otherwise} \end{cases} \\
\text{scaleup}(I)(x) &\stackrel{\text{def}}{=} \begin{cases} I(x)/\sup(I) & \text{if } \sup(I) \neq 0 \\ 0 & \text{otherwise} \end{cases} \\
\text{cut}_{x_1, x_2}(I)(x) &\stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } x < x_1 \text{ or } x \geq x_2 \\ I(x) & \text{otherwise} \end{cases} \\
\text{cut}_{x_1, +}(I)(x) &\stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } x < x_1 \\ I(x) & \text{otherwise} \end{cases} \\
\text{cut}_{x_1, -}(I)(x) &\stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } x \geq x_1 \\ I(x) & \text{otherwise} \end{cases} \\
\text{shift}_n(I)(x) &\stackrel{\text{def}}{=} I(x - n) \\
\text{times}_a(I)(x) &\stackrel{\text{def}}{=} \min(1, a \cdot I(x)) \quad a \geq 0 \\
\text{exponentiate}_e(I)(x) &\stackrel{\text{def}}{=} I(x)^e \quad e \geq 0 \\
\text{integrate}^+(I)(x) &\stackrel{\text{def}}{=} \lim_{a \rightarrow \infty} \frac{\int_{-a}^x I(y) dy}{\int_{-a}^{+a} I(y) dy} \\
\text{integrate}^-(I)(x) &\stackrel{\text{def}}{=} \lim_{a \rightarrow \infty} \frac{\int_x^{+a} I(y) dy}{\int_{-a}^{+a} I(y) dy} \\
\text{negate}_{\text{offset}}(I)(x) &\stackrel{\text{def}}{=} 1 - I(x - \text{offset}) \\
\text{invert}(I)(x) &\stackrel{\text{def}}{=} \begin{cases} 1 - I(x) & \text{if } I_k^{fM} \leq x < I_{k+1}^{lM} \\ & \text{where } I_0, \dots, I_m \text{ are the components of } I \\ 0 & \text{otherwise.} \end{cases}
\end{aligned}$$

■

extend

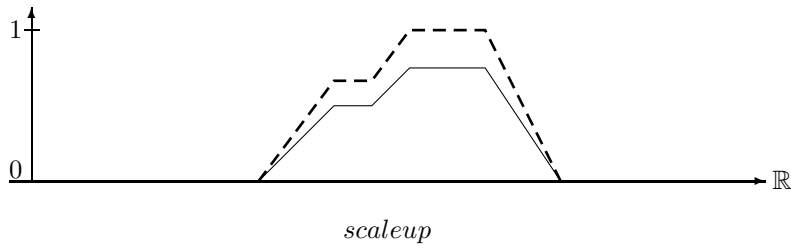
$\text{extend}^+(I)$ follows the left part of the monotone hull of the interval until the left maximum I^{lM} is reached and then stays at fuzzy value 1. $\text{extend}^-(I)$ is the symmetric version of $\text{extend}^+(I)$.



$\text{extend}^+(I)$ is useful for implementing a fuzzy ‘before’-relation because only the left part of I is relevant for evaluating ‘before’. $\text{extend}^-(I)$, on the other hand, can be used for an ‘after’-relation.

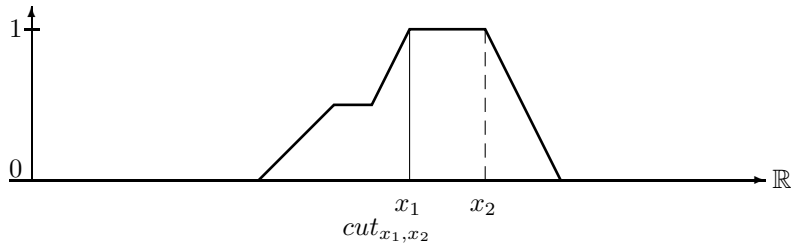
scaleup

The scaleup -function is different to the identity function only if the high $\sup(I)$ is not 1. In this case it scales the membership function up such that $\sup(\text{scaleup}(I)) = 1$.



cut

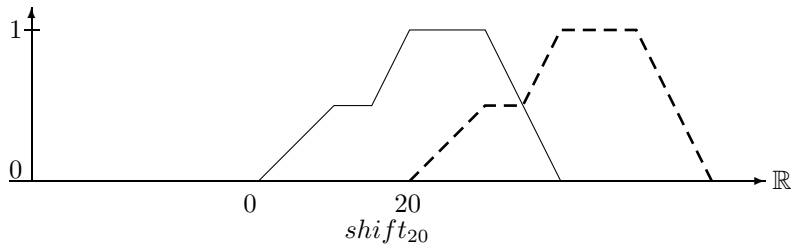
$cut_{x_1, x_2}(I)$ just cuts the piece between x_1 and x_2 out of the interval I . The resulting interval is closed at x_1 and half open at x_2 .



$cut_{x_1, +}(I)$ cuts the part out of I before x_1 whereas $cut_{x_1, -}(I)$ cuts the part out of I after x_1 .

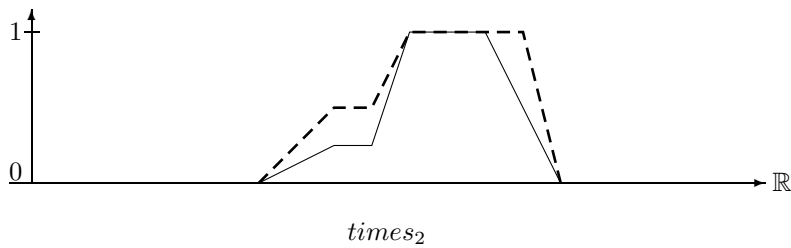
shift

$shift_n$ just moves the interval by n time units.



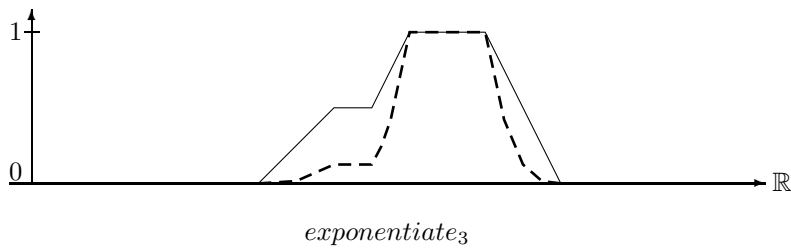
times

$times_a$ multiplies the membership function by a , but keeps the result smaller or equal 1. $times_a$ has no effect on crisp intervals.



exponentiate

$exponentiate_e$ takes the membership function to the exponent e . It can be used to damp increases or decreases. $exponentiate_e$ has also no effect on crisp intervals. $exponentiate_e$ is non-linear in the sense that straight lines are turned into curved lines.



integrate

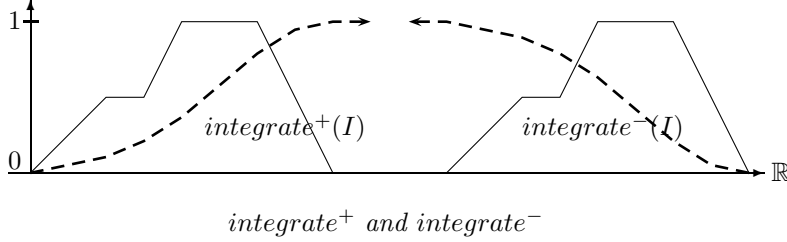
This operator integrates over the membership function and normalises the integral to values ≤ 1 . The two integration operators $integrate^+$ and $integrate^-$ can be simplified for finite fuzzy time intervals.

Proposition 2.32 (Integration for Finite Intervals) *If the fuzzy interval I is finite then*

$$\text{integrate}^+(I)(x) = \frac{\int_{-\infty}^x I(y)dy}{|I|} \quad \text{and} \quad \text{integrate}^-(I)(x) = \frac{\int_x^{+\infty} I(y)dy}{|I|}$$

The proofs are straightforward. ■

Example for integrate^+ and integrate^- :



The integration operator for infinite intervals I with finite kernel turns the interval into a constant function which does no longer depend on the finite part of I .

Proposition 2.33 (Integration for Intervals with Finite Kernel) *If the infinite fuzzy interval I has a finite kernel with $I_1 \stackrel{\text{def}}{=} I(-\infty)$ and $I_2 \stackrel{\text{def}}{=} I(+\infty)$ then*

$$\text{integrate}^+(I)(x) = \frac{I_1}{I_1 + I_2} \quad \text{and} \quad \text{integrate}^-(I)(x) = \frac{I_2}{I_1 + I_2}$$

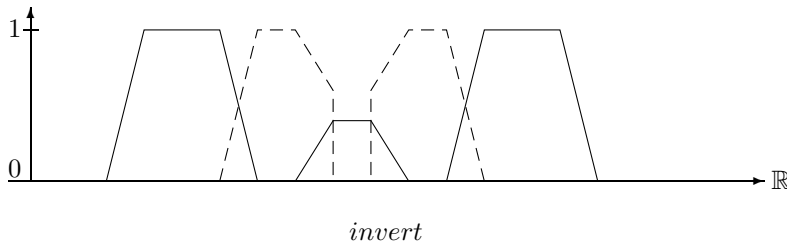
Proof:

$$\begin{aligned} \text{integrate}^+(I)(x) &= \lim_{a \rightarrow \infty} \frac{\int_{-a}^x I(y)dy}{\int_{-a}^x I(y)dy} & \text{integrate}^-(I)(x) &= \lim_{a \rightarrow \infty} \frac{\int_x^{+a} I(y)dy}{\int_x^{+a} I(y)dy} \\ &= \lim_{a \rightarrow \infty} \frac{|I|_{-a}^{IJK} + |I|_{IJK}^x}{|I|_{-a}^{IJK} + |I|_{IJK}^{IJK} + |I|_{IJK}^a} & &= \lim_{a \rightarrow \infty} \frac{|I|_x^{IJK} + |I|_{IJK}^a}{|I|_{-a}^{IJK} + |I|_{IJK}^{IJK} + |I|_{IJK}^a} \\ &= \lim_{a \rightarrow \infty} \frac{|I|_{-a}^{IJK}}{|I|_{-a}^{IJK} + |I|_{IJK}^a} & &= \lim_{a \rightarrow \infty} \frac{|I|_{IJK}^a}{|I|_{-a}^{IJK} + |I|_{IJK}^a} \\ &= \lim_{a \rightarrow \infty} \frac{(I^{JK} + a) \cdot I_1}{(I^{JK} + a) \cdot I_1 + (a - I^{JK}) \cdot I_2} & &= \lim_{a \rightarrow \infty} \frac{(a - I^{JK}) \cdot I_2}{(I^{JK} + a) \cdot I_1 + (a - I^{JK}) \cdot I_2} \\ &= \lim_{a \rightarrow \infty} \frac{a \cdot I_1}{a \cdot I_1 + a \cdot I_2} & &= \lim_{a \rightarrow \infty} \frac{a \cdot I_2}{a \cdot I_1 + a \cdot I_2} \\ &= \frac{I_1}{I_1 + I_2} & &= \frac{I_2}{I_1 + I_2} \end{aligned}$$

■

invert

The *invert* function is almost like the standard negation function, except that $\text{invert}(I)$ is nonzero only in the gaps between the components of I . The interval I in the next picture consists of three components. The maximal fuzzy value of the middle component is not 1. Nevertheless $\text{invert}(I)$ drops down to 0 between the first and last maximum of the middle component. *invert* is needed for an *in_the_gap* operator.



Fuzzification

Fuzzy time intervals could be defined by specifying the shape of the membership function in some way. This is in general very inconvenient. Therefore FuTI provides an alternative. The idea is to take a crisp interval and to ‘fuzzify’ the front and back end in a certain way. For example, one may specify ‘early afternoon’ by taking the interval between 1 and 6 pm and imposing, for example, a linear or a Gaussian shape increase from 1 to 2 pm, and a linear or a Gaussian shape decrease from 4 to 6 pm. Technically this means multiplying a linear or Gaussian function with the membership values.

The fuzzification functions can be defined with absolute coordinates and with relative coordinates. We define the absolute version first.

Definition 2.34 (Linear Fuzzification Function) Let $I \in F_{\mathbb{R}}$, x_1 , x_2 and *offset* be x -coordinates. We define the ‘front’ linear fuzzification function with zero offset first:

$$FALf_{x_1, x_2, 0}(I)(x) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } x < x_1 \\ I(x) & \text{if } x \geq x_2 \\ I(x) \frac{x-x_1}{x_2-x_1} & \text{otherwise} \end{cases}$$

If the *offset* is nonzero we have

$$FALf_{x_1, x_2, \text{offset}}(I)(x) \stackrel{\text{def}}{=} \begin{cases} FALf_{x_1, x_2, 0}(x + \text{offset}) & \text{if } x < x_2 - \text{offset} \\ FALf_{x_1, x_2, 0}(x_2) & \text{if } x_2 - \text{offset} \leq x < x_2 \\ I(x) & \text{otherwise} \end{cases}$$

The ‘back’ linear fuzzification function is:

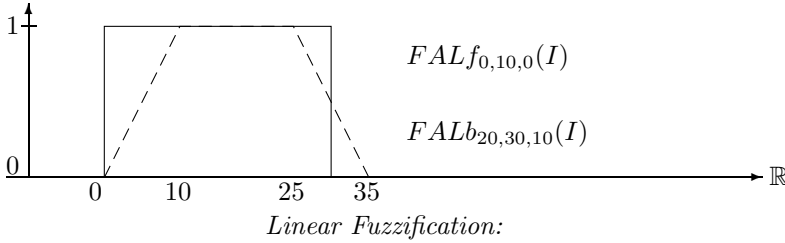
$$FALb_{x_1, x_2, 0}(I)(x) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } x \geq x_2 \\ I(x) & \text{if } x < x_1 \\ I(x) \frac{x_2-x}{x_2-x_1} & \text{otherwise} \end{cases}$$

If the *offset* is nonzero we have

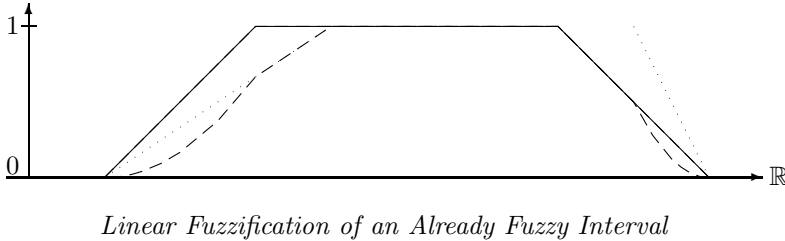
$$FALb_{x_1, x_2, \text{offset}}(I)(x) \stackrel{\text{def}}{=} \begin{cases} FALb_{x_1, x_2, 0}(x - \text{offset}) & \text{if } x \geq x_1 + \text{offset} \\ FALb_{x_1, x_2, 0}(x_2) & \text{if } x_1 \leq x \leq x_1 + \text{offset} \\ I(x) & \text{otherwise} \end{cases}$$

■

In the picture below we fuzzify a crisp interval with a linear increase from 0 – 10, and a linear decrease from 20 – 30, which is shifted by an offset of 10.

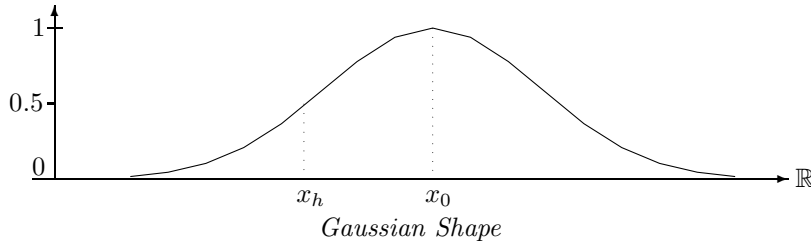


The next example shows the linear fuzzification of an already fuzzy interval. The dotted lines show the linear increase and decrease. The dashed line is the result of the fuzzification operator. Since the two polygons are multiplied, we get quadratic curves.



Gaussian Fuzzification

Besides linear fuzzification, FuTI offers the fuzzification with a Gaussian shape. The Gaussian function is $e^{-\frac{(x-x_0)^2}{\sigma}}$. x_0 is the symmetry point and σ determines the increase and decrease.



The Gaussian fuzzification function is determined by the parameters x_0 and x_h . x_h is the x -coordinate where $e^{-\frac{(x_h-x_0)^2}{\sigma}} = 0.5$. This condition determines $\sigma = \sqrt{(-1/\ln(0.5))} \cdot (x_h - x_0)$.

Since the Gauss function does not become 0, we must cut it off at some x -coordinate. The heuristic is to cut it off at a distance $3(x_0 - x_h)$ from x_0 .

Definition 2.35 (Gaussian Fuzzification Function) Let $I \in F_{\mathbb{R}}$, x_h , x_0 and *offset* be x -coordinates. Let $\sigma \stackrel{\text{def}}{=} \sqrt{(-1/\ln(0.5))} \cdot (x_h - x_0)$.

We define the ‘front’ Gaussian fuzzification function with zero ‘offset’ first:

$$FAGf_{x_h, x_0, 0}(I)(x) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } x < 3x_h - 2x_0 \\ I(x) & \text{if } x \geq x_0 \\ I(x)e^{-((x-x_0)/\sigma)^2} & \text{otherwise} \end{cases}$$

If the *offset* is nonzero we have

$$FAGf_{x_h, x_0, \text{offset}}(I)(x) \stackrel{\text{def}}{=} \begin{cases} FAGf_{x_h, x_0, 0}(x + \text{offset}) & \text{if } x < x_0 - \text{offset} \\ FAGf_{x_h, x_0, 0}(x_0) & \text{if } x_0 - \text{offset} \leq x < x_0 \\ I(x) & \text{otherwise} \end{cases}$$

The ‘back’ Gaussian fuzzification function is:

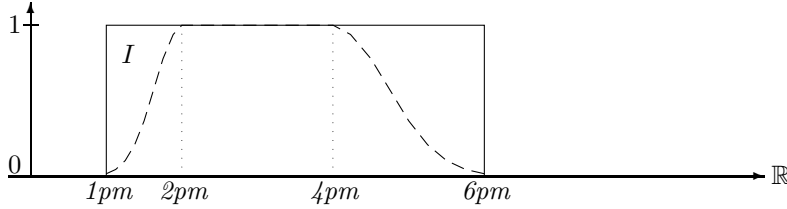
$$FAGb_{x_h, x_0, 0}(I)(x) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } x > 3x_h - 2x_0 \\ I(x) & \text{if } x < x_0 \\ I(x)e^{-((x-x_0)/\sigma)^2} & \text{otherwise} \end{cases}$$

If the *offset* is nonzero we have

$$FAGb_{x_h, x_0, \text{offset}}(I)(x) \stackrel{\text{def}}{=} \begin{cases} FAGb_{x_h, x_0, 0}(x - \text{offset}) & \text{if } x \geq x_0 + \text{offset} \\ FAGb_{x_h, x_0, 0}(x_2) & \text{if } x_0 \leq x \leq x_0 + \text{offset} \\ I(x) & \text{otherwise} \end{cases}$$

■

Example 2.36 We fuzzify ‘early afternoon’ by taking the interval between 1pm and 6pm, imposing a Gaussian rise between 1pm and 2pm and a Gaussian decrease between 4 and 6pm.



Early Afternoon: $FAGf_{1.5, 2, 0}(I)$ and $FAGb_{5, 4, 0}(I)$

■

Fuzzification functions with absolute coordinates are not that useful because usually one does not know the coordinates in advance. Therefore FuTI also provides fuzzification functions where the parameters are percentage values. $FRLf_{10, 5}$, for example, means linear fuzzification where the linear increase is 10% of the kernel size and the offset is 5% of the kernel size. $FRGf_{10, 0}$ means a Gaussian increase where x_0 is 10% of the kernel size past I^{fK} , x_h is 1/2 the distance between I^{fK} and x_0 , and the offset is such that x_h coincides with I^{fK} .

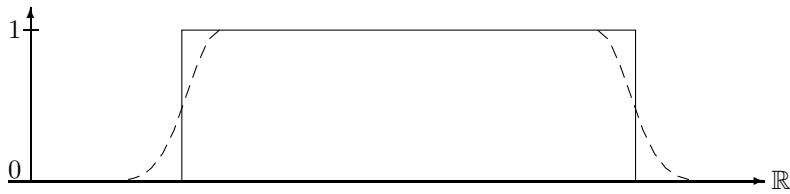
Definition 2.37 (Fuzzification with Relative Coordinates) For an interval I , percentage numbers r and o between 0 and 100 we define the relative fuzzification functions.

Let $d = (I^{lK} - I^{fK})/100$.

$$\begin{aligned} FRLf_{r, o}(I) &\stackrel{\text{def}}{=} FALf_{I^{fK}, I^{fK}+d \cdot r, I^{fK}-d \cdot o}(I) \\ FRLb_{r, o}(I) &\stackrel{\text{def}}{=} FALb_{I^{lK}-d \cdot r, I^{lK}, I^{lK}+d \cdot o}(I) \\ FRL_{r, o}(I) &\stackrel{\text{def}}{=} FALb_{I^{lK}-d \cdot r, I^{lK}, I^{lK}+d \cdot o}(FALf_{I^{fK}, I^{fK}+d \cdot r, I^{fK}-d \cdot o}(I)) \\ FRGf_r(I) &\stackrel{\text{def}}{=} FAGf_{I^{fK}+d \cdot r, I^{fK}+1/2 \cdot d \cdot r, 2/3 \cdot d \cdot r}(I) \\ FRGb_r(I) &\stackrel{\text{def}}{=} FAGb_{I^{lK}-d \cdot r, I^{lK}-1/2 \cdot d \cdot r, 2/3 \cdot d \cdot r}(I) \\ FRG_r(I) &\stackrel{\text{def}}{=} FAGb_{I^{lK}-d \cdot r, I^{lK}-1/2 \cdot d \cdot r, 2/3 \cdot d \cdot r}(FAGf_{I^{fK}+d \cdot r, I^{fK}+1/2 \cdot d \cdot r, 2/3 \cdot d \cdot r}(I)) \end{aligned}$$

■

The functions FRL and FRG fuzzify an interval at both sides. A simple composition of $FRLf$ and $FRLb$, for example, yields an un-symmetric result because the fuzzification at one end first changes the kernel size. The relative fuzzification of the other side of the changed literal therefore uses the data of the changed interval for computing x_h and x_0 . FRL and FRG avoid this by computing the absolute coordinates first and using them for both sides.



Relative Gaussian Fuzzification FRG_{20}

3 Data Structures and Algorithms

The main data structures and algorithms of the FuTI-library are presented in this section. The actual implementation contains a few more functions. Since their implementation is more or less straight forward, they are not mentioned explicitly here. There are four basic datatypes: time points, fuzzy values, fuzzy temporal intervals and y -functions.

Time Points

The time points are points on the \mathbb{R} -axis. Arbitrary real numbers cannot be represented on computers. The choice is therefore between floating point numbers and integers as representation of time points. The range of floating point numbers is much higher than the range of integers. Unfortunately, algorithms operating on floating point numbers are prone to uncontrollable rounding errors. Another argument for using integers instead of floating point numbers is that the real time measurements on earth give you always integers. The very definition of exact time measurement already uses integers: in 1967 one second was defined as 9.192.631.770 cycles of the light emitted when an electron jumps between the two lowest hyperfine levels of the Caesium 133 atom. Thus, the most precise time measurement available at all depends on counting integers (cycles of light).

Therefore the FuTI-library *represents time with integer coordinates*. There is no assumption about the meaning of these integers. They may be years, seconds, picoseconds or even cycles of the Caesium 133 light.

Fuzzy Values

Fuzzy values are usually real numbers between 0 and 1. A first choice would therefore be to use floating point numbers for the fuzzy values. Again, floating point numbers are prone to rounding errors. Moreover, computation with floating point numbers is more expensive than computation with integers. Therefore FuTI uses again integers instead of floating point numbers. This means of course that one cannot represent the fuzzy value 1 as the integer 1. We could then use just 0 and 1 and no other fuzzy value. Instead one better represents the fuzzy value 1 as a suitable unsigned integer of a certain bit size. Since fuzzy values are estimates only anyway, 16 bit unsigned integer (unsigned short int in C) are precise enough for fuzzy values.

Definition 3.1 (Largest Fuzzy Value) *Let \top be the maximal fuzzy value in the implementation.* ■

To make the examples more easy to understand, we use $\top = 1000$ in this paper. \top is a compiler option in the actual implementation and can be changed easily.

Fuzzy Time Intervals

Fuzzy intervals are usually implemented by a representation of their membership functions. Arbitrary membership functions are almost impossible to represent precisely on a computer. A natural choice for realizing approximated fuzzy time intervals over integer time and integer fuzzy values is the representation with *envelope polygons* over integer coordinates. This has a number of advantages: the representation is compact and can nevertheless approximate the membership functions very well; simple structures, like crisp intervals, have a simple representation; we can use ideas and algorithms from Computational Geometry [9, 3]; there are very efficient algorithms for most of the problems, and it is clear where rounding errors can occur, and where not.

Coordinates and Integer Datatypes

The implemented fuzzy intervals are independent of their interpretation as fuzzy time intervals. Therefore we shall speak of the x -axis instead of the time axis and of the y -axis instead of the fuzzy value axis.



The Used Coordinate System

Definition 3.2 (x-Integers and y-Integers) *FuTI uses integers of different size for the x-coordinates and the y-coordinates. Therefore we shall speak of the x-integers and of the y-integers. The default for x-integers is 64 bit long long integers, and the default for y-integers is 16 bit short integers. The library has also been tested with multiple-precision x-integers. ■*

Notation for Algorithms

We shall write most algorithms in a functional notation which is as mathematical as possible, but still concrete enough that they can be implemented straight away. It turned out that the object oriented paradigm is not only very good for getting modularised and easy to understand implementations, but it also makes the mathematical notation clearer. Therefore we shall use the notation $o.v$ and $o.m(p_1, \dots, p_n)$ where o is an object, v is an instance variable, and m is a method (function) with arguments p_1, \dots, p_n .

The expression

$$a \stackrel{\text{def}}{=} \begin{cases} s_1 & \text{if } \varphi_1 \\ s_2 & \text{if } \varphi_2 \\ \dots & \dots \\ s_n & \text{otherwise} \end{cases}$$

is a case analysis. It means:

$a \stackrel{\text{def}}{=} s_1$ if φ_1 is true

$a \stackrel{\text{def}}{=} s_2$ if φ_1 is false and φ_2 is true

...

$a \stackrel{\text{def}}{=} s_n$ if $\varphi_1, \dots, \varphi_{n-1}$ are all false.

The notation $\Sigma_{n=0}^m s(n)$ is well known in mathematics. In the same style we define a notation $V_{n=0}^m s(n)$. The V operator causes the values $s(n)$ to be collected in a list. For example,

$$V_{n=0}^{20} \begin{cases} (n) & \text{if } n \text{ is a prime number} \\ () & \text{otherwise} \end{cases}$$

yields the list (1,3,5,7,11,13,17,19).

We may also use the keyword *break* to stop the V -loop. For example,

$$V_{n>0} \begin{cases} break & \text{if } n > 20 \\ (n) & \text{if } n \text{ is a prime number} \\ () & \text{otherwise} \end{cases}$$

yields the same list (1,3,5,7,11,13,17,19).

Sometimes it is necessary to include a value in a list and then stop the loop. We specify this with an expression ' s and *break*'.

$$V_{n>0} \begin{cases} (n) \text{ and } break & \text{if } n > 20 \\ (n) & \text{if } n \text{ is a prime number} \\ () & \text{otherwise} \end{cases}$$

yields the list (1,3,5,7,11,13,17,19,21).

Partial Functions and Error Handling

Most of the functions defined in this chapter are partial functions. Therefore the preconditions the arguments must meet when these functions are called need to be stated very clearly. This means for

an implementation that the functions should only be called when the preconditions are guaranteed. An error handling mechanism treats the cases where the preconditions are not met.

Special Functions

We use the following functions:

$roundX(a)$ rounds the floating point number a to the closest x -integer (time value).

$roundY(a)$ rounds the floating point number a to the closest y -integer (fuzzy value).

The two functions are almost identical. The only difference is the bit length of the resulting integer values.

3.1 Points

We need 2-dimensional points with coordinates (x, y) as the representation of points on the envelope polygon. The x -coordinate is the time coordinate and the y -coordinate is the fuzzy value coordinate. x -coordinates are represented with x -integers and y -coordinates are represented with y -integers (Def. 3.2).

Notation

If $p = (x, y)$ is a point then $p.x$ denotes the x -coordinate (time coordinate) of p and $p.y$ denotes the y -coordinate (fuzzy coordinate) of p .

Collinearity

The collinearity check for three points p_1, p_2 and p_3 is a standard method from Computational Geometry [9]. The doubled area of the triangle p_1, p_2 and p_3 is computed. With integer coordinates this can be done without any error at all. If the doubled area is 0 then the three points are collinear.

Definition 3.3 (Collinear) *The method $p_1.collinear(p_2, p_3)$ returns true if the three points p_1, p_2 and p_3 are collinear.* ■

Left turn

Another important operator is the ‘left turn test’.

Definition 3.4 (Left Turn) *The method $p_1.leftturn(p_2, p_3)$ returns true if the three points p_1, p_2 and p_3 make a left turn.* ■

The $leftturn$ method computes the doubled area of the triangle p_1, p_2 and p_3 and checks its sign. Left turns and right turns yield opposite signs.

Intersection

Testing whether line segments intersect and computing the intersection point are also standard methods from Computational Geometry.

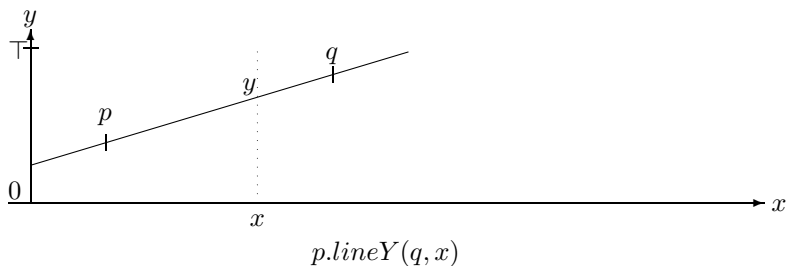
Definition 3.5 (Intersects and IntersectsProper) *The method $p_1.intersects(p_2, q_1, q_2)$ returns true if the line segment (p_1, p_2) intersects or touches the line segment (q_1, q_2) .*

The method $p_1.intersectsProper(p_2, q_1, q_2)$ returns true if the line segment (p_1, p_2) intersects properly, and not only touches the line segment (q_1, q_2) . ■

Definition 3.6 (Intersection) *The method $p_1.intersection(p_2, q_1, q_2)$ returns the rounded x -coordinate of the intersection point of the two intersecting line segments (p_1, p_2) and (q_1, q_2) .* ■

lineY

The function $p.lineY(q, x)$ considers the line crossing the points p and q , and computes for a given x -value the corresponding y value at the line.



Definition 3.7 (lineY) Let p and q be the two points which define a line, and let x be an x -coordinate.

$$p.\text{lineY}(q, x) \stackrel{\text{def}}{=} \begin{cases} \text{undefined} & \text{if } p.x = q.x \\ p.y + \frac{(q.y-p.y) \cdot (x-p.x)}{q.x-p.x} & \text{otherwise} \end{cases}$$

The result is floating point number. ■

lineX

This method computes for a line and a y -value the corresponding x -value.

Definition 3.8 (lineX) Let p and q be the two points which define a line, and let y be a y -coordinate.

$$p.\text{lineX}(q, y) \stackrel{\text{def}}{=} \begin{cases} \text{undefined} & \text{if } p.y = q.y \\ p.x + \text{roundX}\left(\frac{(q.x-p.x) \cdot (y-p.y)}{q.y-p.y}\right) & \text{otherwise} \end{cases}$$

The result is an x -coordinate. ■

Area

We provide two methods for computing the area between a line and the x -axis. The first function $p.\text{area2}(q)$ computes for two points p and q twice the area below the line segment between p and q . When p and q are points with integer coordinates then twice the area yields also an integer, and no rounding is necessary.

The second method $p.\text{area2}(q, x_1, x_2)$ computes twice the area between x_1 and x_2 below the line segment between p and q .

Definition 3.9 (area2) Let p and q be the two points which define a line, let x_1 and x_2 x -coordinates.

$$p.\text{area2}(q) \stackrel{\text{def}}{=} (q.x - p.x) \cdot (q.y + p.y)$$

The result is an x -integer.

$$p.\text{area2}(q, x_1, x_2) \stackrel{\text{def}}{=} \begin{cases} \text{undefined} & \text{if } p.x = q.x \text{ and } p.x \neq x \\ 0 & \text{if } p.x = q.x = x \\ (x_2 - x_1) \cdot (p.\text{lineY}(q, x_2) - p.\text{lineY}(q, x_1)) & \text{otherwise} \end{cases}$$

The result is a floating point number. ■

The next method, $p.\text{area2X}(q, a)$ computes for two points p and q and for a doubled area a the x -coordinate x such that twice the area below the line segment between p and q from $p.x$ till x is a . The function is undefined if the line is vertical, or the line is just the coordinate axis and $a > 0$, or the slope of the line is negative and there is not enough area available between $p.x$ and the point where the line crosses the coordinate axis.

Definition 3.10 (area2X) Let p and q be the two points which define a line. Let $a \geq 0$ be an integer or floating point number.

$$p.\text{area2X}(q, a) \stackrel{\text{def}}{=} \begin{cases} \text{undefined} & \text{if } p.x = q.x \\ & \text{or } p.y = q.y = 0 \text{ and } a > 0 \\ & \text{or } p.y^2 < -\text{slope} \cdot a \\ p.x & \text{if } p.y = q.y = 0 \text{ and } a = 0 \\ p.x + \text{roundX}\left(\frac{a}{2p.y}\right) & \text{if } p.y = q.y \\ p.x + \text{roundX}\left(\frac{\sqrt{p.y^2 + \text{slope} \cdot a} - p.y}{\text{slope}}\right) & \text{otherwise} \\ \text{where } \text{slope} \stackrel{\text{def}}{=} \frac{q.y-p.y}{q.x-p.x} \end{cases}$$

The result is a rounded x -integer. ■

Proposition 3.11 (Soundness of area2X) Let p and q be two points and a a doubled area (non-negative number). Then $p.\text{area2X}(q, a)$ returns the (rounded) x -coordinate x such that the doubled area below the line crossing p and q and between $p.x$ and x equals a .

Proof: The doubled area below the line crossing p and q and between $p.x$ and x is

$$(x - p.x) \cdot (p.y + (p.y + \text{slope} \cdot (x - p.x))) = a$$

where $\text{slope} = \frac{q.y-p.y}{q.x-p.x}$

Case 1: $q.x - p.x = 0$, i.e. $p.x = q.x$.

The equation is not solvable in this case.

Case 2: $slope = 0$, i.e. $p.y = q.y$:

Case 2a: $p.y = 0$: the equation is only solvable for $a = 0$, in which case $p.x$ is a solution.

Case 2b: $p.y > 0$: The equation simplifies in this case to

$(x - p.x) \cdot 2p.y + a = 0$ with solution

$$x = p.x + \frac{a}{2p.y}.$$

Case 3: $slope \neq 0$:

The equation is normalised to

$slope \cdot (x - p.x)^2 + 2p.y(x - p.x) - a = 0$ with solution

$$(x - p.x) = \frac{-2p.y \pm \sqrt{4p.y^2 + 4slope \cdot a}}{2slope}$$

$$x = p.x + \frac{-p.y \pm \sqrt{p.y^2 + slope \cdot a}}{slope}$$

The $-\sqrt{\dots}$ -case yields a point left of $p.x$, which is not what we want. The square root has a real number solution only if $p.y^2 + slope \cdot a \geq 0$. Otherwise the function is undefined. ■

Integration

Some interval–interval relations are defined as an integral over two multiplied polygons (Section 3.3). A building block for the integration algorithm is a method which integrates the product of two lines.

Definition 3.12 (Integration of Multiplied Lines) Let p_1, p_2 and q_1, q_2 be the two pairs of points which define two lines. Let x_1 and x_2 be two x -coordinates.

$$p_1.integrate(p_2, q_1, q_2, x_1, x_2) \stackrel{\text{def}}{=} \begin{cases} \text{undefined} & \text{if } (p_1.x = p_2.x \text{ or } q_1.x = q_2.x) \text{ and } x_1 \neq x_2 \\ 0 & \text{if } x_1 = x_2 \\ a \cdot b \cdot (x_2 - x_1) + (m_2a + m_1b) \cdot (x_2^2 - x_1^2)/2 + m_1 \cdot m_2 \cdot (x_2^3 - x_1^3)/3 & \text{otherwise} \end{cases}$$

where $a \stackrel{\text{def}}{=} p_1.y - m_1p_1.x$, $b \stackrel{\text{def}}{=} q_1.y - m_2q_2.x$,

$$m_1 \stackrel{\text{def}}{=} \frac{p_2.y - p_1.y}{p_2.x - p_1.x} \text{ and } m_2 \stackrel{\text{def}}{=} \frac{q_2.y - q_1.y}{q_2.x - q_1.x}.$$

The result is a floating point number. ■

Proposition 3.13 (Soundness of Integration of Multiplied Lines) Let p_1, p_2 and q_1, q_2 be the two pairs of points which define two lines. Let x_1 and x_2 be two x -coordinates. Then

$$p_1.integrate(p_2, q_1, q_2, x, y) = \int_{x_1}^{x_2} l_1(x) \cdot l_2(x) dx$$

where l_1 is the line crossing p_1 and p_2 and l_2 is the line crossing q_1 and q_2 .

Proof: Let $l_1(x) \stackrel{\text{def}}{=} p_1.y + m_1(x - p_1.x)$ and $l_2(x) \stackrel{\text{def}}{=} q_1.y + m_2(x - q_1.x)$

where $m_1 \stackrel{\text{def}}{=} \frac{p_2.y - p_1.y}{p_2.x - p_1.x}$ and $m_2 \stackrel{\text{def}}{=} \frac{q_2.y - q_1.y}{q_2.x - q_1.x}$.

$$\begin{aligned} & \int_{x_1}^{x_2} l_1(x) \cdot l_2(x) dx \\ &= \int_{x_1}^{x_2} (p_1.y + m_1(x - p_1.x))(q_1.y + m_2(x - q_1.x)) dx \\ &= [(p_1.y - m_1p_1.x)(q_1.y - m_2q_2.x) + (m_2(p_1.y - m_1p_1.x) + m_1(q_1.y - m_2q_2.x))x + m_1m_2x^2]_{x_1}^{x_2} \\ &= [ab + (m_2a + m_1b)x + m_1m_2x^2]_{x_1}^{x_2} \\ &= ab(x_2 - x_1) + (m_2a + m_1b)(x_2^2 - x_1^2)/2 + m_1m_2(x_2^3 - x_1^3)/3 \end{aligned}$$

where $a \stackrel{\text{def}}{=} p_1.y - m_1p_1.x$ and $b \stackrel{\text{def}}{=} q_1.y - m_2q_2.x$. ■

3.2 Fuzzy Time Intervals

In this section we introduce a concrete representation of fuzzy time intervals and present the algorithms implemented in FuTI.

Definition 3.14 (Infinity) We use $+\infty$ and $-\infty$ with the same meaning as before. However, since infinity cannot be represented properly on a computer, $+\infty$ stands in fact for a sufficiently large positive representable x -integer, and $-\infty$ stands for a sufficiently large negative representable x -integer. If the bit size of the integers is fixed, these can be the largest representable integers at all. For multiple-precision integers one can choose an arbitrary very large number. ■

The finite representation of $+\infty$ and $-\infty$ could in principle cause errors if the time values become extremely large. Therefore one has to check in the application how large the numbers could become and then choose a large enough x -integer datatype.

3.2.1 Representation and Construction

Fuzzy intervals are represented by their *envelope polygons*. These polygons represent the membership functions.

Definition 3.15 (Envelope Polygon) The envelope polygon I of a fuzzy time interval is a finite sequence of points p_0, \dots, p_n such that $p_i.x \leq p_{i+1}.x$ holds for all i .

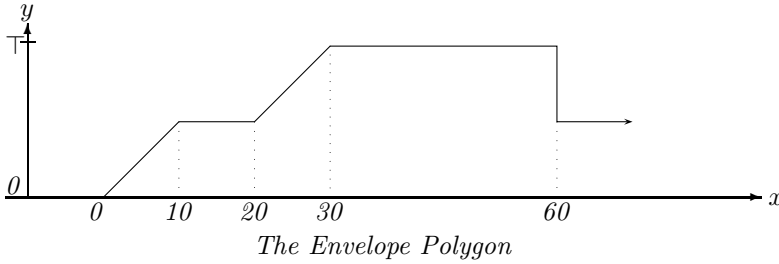
The envelope polygons in FuTI are constructed that there are no redundant points. That means in particular that there are no collinear triples (p_i, p_{i+1}, p_{i+2}) of points.

We usually identify the envelope polygon with the fuzzy interval itself. ■

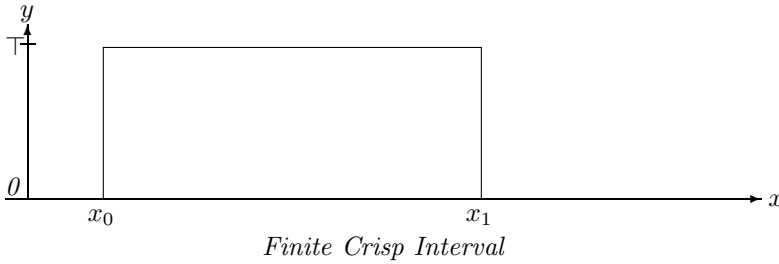
Example 3.16 (Envelope Polygon) The picture below shows the envelope polygon

$$I = (0, 0)(10, 500)(20, 500)(30, 1000)(60, 1000)(60, 500).$$

Since $p_5.y = 500 > 0$ it represents a positive infinite fuzzy interval.



Example 3.17 (Crisp Intervals) The representation of finite crisp intervals consists always of four points: $I = ((x_0, 0)(x_0, T)(x_1, T)(x_1, 0))$.



The envelope polygon representation of fuzzy intervals leaves it open whether they are open or closed intervals. This decision is left to the membership function (see Remark 3.24 below).

Infinite crisp intervals can of course also be represented. For example, $[10, +\infty[$ can be represented by $(10, 0)(10, T)$. $[-\infty, 10[$ can be represented by $(10, T)(10, 0)$. ■

An envelope polygon is constructed from the empty list of points by adding new points to the back of the list. The *push_back* method defined below ensures that the condition $p_i.x \leq p_{i+1}.x$ holds and that collinear triples of points are avoided.

Definition 3.18 (push_back and pop_back) Let $I = (p_0, \dots, p_n)$ be an envelope polygon and p a new point.

$$I.push_back(p) \stackrel{\text{def}}{=} \begin{cases} \text{undefined} & \text{if } I \neq () \text{ and } p.x < p_n.x \\ (p) & \text{if } I = () \text{ or } I = (p_0) \text{ and } p.y = p_0.y \\ (p_0, p) & \text{if } I = (p_0) \\ (p_1, \dots, p_{n-1}, p) & \text{if } p.collinear(p_{n-1}, p_n) = \text{true (Def. 3.3)} \\ (p_1, \dots, p_n, p) & \text{otherwise} \end{cases}$$

$I.pop_back()$ removes the last element. ■

The *push_back* method alone does not guarantee that there are no redundant points in an envelope polygon. The method $I.close()$ defined next removes all remaining redundancies. It is automatically called before the other algorithms use the envelope polygon.

Definition 3.19 (Close) Let $I = (p_0, \dots, p_n)$ be an envelope polygon.

$$I.close() \stackrel{\text{def}}{=} \begin{cases} () & \text{if } I = ((x, 0)) \\ (p_1, \dots, p_n).close() & \text{if } p_0.y = p_1.y \\ (p_0, \dots, p_{n-1}) & \text{if } p_{n-1}.y = p_n.y \\ I & \text{otherwise} \end{cases}$$

■

The method *index* defined below can be used to locate for a given x -coordinate x and an envelope polygon I the line segment which is above x . *indexMax(true)* locates the index of the leftmost polygon point with maximum y -value (I^{fM}), whereas *indexMax(false)* locates the index of I^{lM} .

Definition 3.20 (index and indexMax) For an envelope polygon $I = (p_0, \dots, p_n)$ let

$$I.index(x) \stackrel{\text{def}}{=} \begin{cases} -1 & \text{if } I = () \text{ or } x < p_0.x \\ \max\{k \leq n \mid x_k \leq x\} & \text{otherwise} \end{cases}$$

be the index of the rightmost polygon point that is left of x . The index is actually obtained with binary search in $O(\log_2(n))$ time.

$$I.indexMax(front) \stackrel{\text{def}}{=} \begin{cases} -1 & \text{if } I = () \\ \min\{i \geq 0 \mid p_i.y = \top \text{ or } \forall j : 0 \leq j < i : p_j.y < p_i.y\} & \text{if } front = true \\ \max\{i \leq n \mid p_i.y = \top \text{ or } \forall j : i < j \leq n : p_j.y < p_i.y\} & \text{if } front = false \end{cases}$$

■

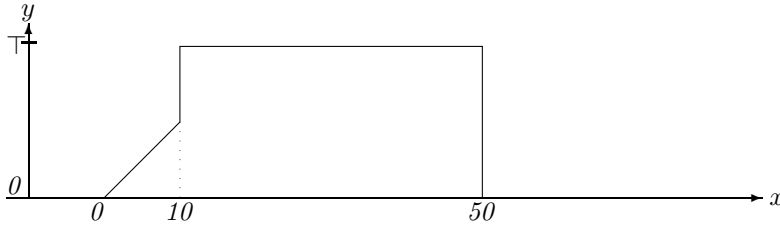
indexMax requires linear search. Fortunately the search can be stopped as soon as a point p_i is reached with $p_i.y = \top$. Therefore for the important case of crisp polygons, the search stops always at the second point.

Example 3.21 (index and indexMax) For the envelope polygon

$$I = \underbrace{(0, 0)}_{p_0} \underbrace{(10, 500)}_{p_1} \underbrace{(10, 1000)}_{p_2} \underbrace{(50, 1000)}_{p_3} \underbrace{(50, 0)}_{p_4}$$

we have

$$I.index(0) = 0, I.index(9) = 0, I.index(10) = 2, I.index(11) = 2, I.index(50) = 4, \\ I.indexMax(true) = 2, I.indexMax(false) = 3.$$



index and indexMax

■

The envelope polygon contains only the vertices of a piecewise linear membership function. Therefore we need a *member* method which interpolates for a given x the corresponding y -value of the membership function.

Definition 3.22 (Member Function) Given a fuzzy interval (envelope polygon) $I = (p_0, \dots, p_n)$ the Member function is defined:

$$I.member(x) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } I = () \\ p_0.y & \text{if } x < p_0.x \\ p_n.y & \text{if } x \geq p_n.x \\ p_i.y & \text{if } x = p_i.x \\ p_i.lineY(p_{i+1}, x) & \text{otherwise} \\ \text{where } i = I.index(x) \end{cases}$$

The result is converted to a floating point number, if necessary.

The usual membership function (Def. 2.2) is then $I(x) = I.member(I, x)/\top$

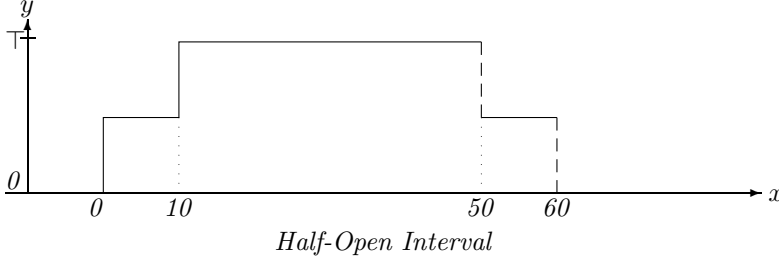
■

Remark 3.23 (Extrapolation and Infinite Intervals) *The member method extrapolates the membership function to x -coordinates below $p_0.x$ and above $p_n.x$. The y -value for x -coordinates below $p_0.x$ is constant $p_0.y$. The y -value for x -coordinates above $p_n.x$ is constant $p_n.y$. Therefore envelope polygons always represent fuzzy intervals with finite kernel (Def. 2.5). ■*

Remark 3.24 (Half-open Intervals) *The index method (Def. 3.20) which is used in the member method returns for a given x the largest index i such that $p_i.x \leq x$. This causes that the envelope function is interpreted as a half-open interval which is closed at the left hand side and open at the right hand side.*

To see this, consider the following example:

$$I = \underbrace{(0, 0)}_{p_0} \underbrace{(0, 500)}_{p_1} \underbrace{(10, 500)}_{p_2} \underbrace{(10, 1000)}_{p_3} \underbrace{(50, 1000)}_{p_4} \underbrace{(50, 500)}_{p_5} \underbrace{(60, 500)}_{p_6} \underbrace{(60, 0)}_{p_7}$$



We have $I.member(0) = 500$, $I.member(10) = 1000$, $I.member(50) = 500$, $I.member(60) = 0$ because $I.index(50) = 5$ and $I.index(60) = 7$. ■

Remark 3.25 (Extreme Cases) *There are a number of extreme cases of envelope polygons I :*

- $I = ()$ represents the empty set;
- $I = ((a, 0))$ also represents the empty set (which gets normalised to $()$);
- $I = ((a, y))$ with $y > 0$ represents the infinite fuzzy interval with constant membership function $I(x) = y$;
- $I = ((a, y_1)(a, y_2))$ represents the fuzzy interval with membership function

$$I(x) = \begin{cases} y_1 & \text{for } x < a \\ y_2 & \text{for } x \geq a. \end{cases}$$

- $((0, 0)(0, \top))$ represents $[0, +\infty[$
- $((0, \top)(0, 0))$ represents $] - \infty, 0[$

3.2.2 Basic Features of Fuzzy Intervals

We start with some simple predicates for checking whether the intervals are infinite.

Definition 3.26 (Infinity Predicates) *Let $I = (p_0, \dots, p_n)$ be an envelope polygon.*

$$\begin{aligned} I.isNegInfinite() &\stackrel{\text{def}}{=} I \neq () \text{ and } p_0.y > 0 \\ I.isPosInfinite() &\stackrel{\text{def}}{=} I \neq () \text{ and } p_n.y > 0 \\ I.isInfinite() &\stackrel{\text{def}}{=} I \neq () \text{ and } p_0.y > 0 \text{ or } p_n.y > 0 \end{aligned}$$

Using the *indexMax*-method (Def. 3.20) we can define $I.sup()$ for computing the height $\text{sup}(I)$ (Def. 2.4) of the fuzzy interval.

Definition 3.27 (sup and inf Values) *Let $I = (p_0, \dots, p_n)$ be an envelope polygon.*

$$\begin{aligned} I.inf() &\stackrel{\text{def}}{=} \min\{p_i.y \mid 0 \leq i \leq n\} \\ I.sup() &\stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } I = () \\ p_{I.indexMax(true)}.y & \text{otherwise} \end{cases} \end{aligned}$$

The result of *sup* and *inf* are y -integer values. ■

The complexity of *sup* and *inf* are in general linear because *indexMax* requires linear search. It is constant for crisp intervals.

Size of Fuzzy Intervals

The *size* of a fuzzy interval is the integral over its membership functions (Def. 2.9). We define now three methods for computing the (doubled) size of a fuzzy interval. *size2()* computes the overall size, i.e. $I.size2()/\top = 2|I|$. *I.size2(k,l)* computes the size between two vertices of the envelope polygon, i.e. $I.size2(k,l)/\top = 2|I|_{p_k.x}^{p_l.x}$. Finally *I.size2(a,b)* computes the size between two arbitrary *x*-coordinates *a* and *b*: $I.size2(a,b)/\top = 2|I|_a^b$.

Definition 3.28 (Size) Let $I = (p_0, \dots, p_n)$ be an envelope polygon. Let *k* and *l* be two indices.

$$I.size2I(k, l) \stackrel{\text{def}}{=} \begin{cases} \text{undefined} & \text{if } k < 0 \text{ or } l > n \\ -I.size2(l, k) & \text{if } l < k \\ 0 & \text{if } k = l \text{ or } I = () \\ \sum_{m=k}^{l-1} p_m.area2(p_{m+1}) & \text{otherwise} \quad (\text{Def.3.9}) \end{cases}$$

$$I.size2() \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } I = () \\ +\infty & \text{if } p_0.y > 0 \text{ or } p_n.y > 0 \\ I.size2(0, n) & \text{otherwise} \end{cases}$$

Both versions of *size2* return *x*-integers.

Now let *a* and *b* be two *x*-coordinates:

$$I.size2(a, b) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } I = () \text{ or } a = b \\ -I.size2(b, a) & \text{if } b < a \\ 2 \cdot (b - a) \cdot p_n.y & \text{if } a \geq p_n.x \\ 2 \cdot (b - a) \cdot p_0.y & \text{if } b \leq p_0.x \\ (b - a) \cdot (p_i.lineY(p_{i+1}, a) + p_i.lineY(p_{i+1}, b)) & \text{if } p_{i-1}.x \leq a \leq b \leq p_i.x \\ & \text{where } i = I.index(a) \\ \text{head} + \text{middle} + \text{tail} & \text{otherwise} \end{cases}$$

where

$$\text{head} \stackrel{\text{def}}{=} \begin{cases} 2 \cdot (p_0.x - a) \cdot p_0.y & \text{if } a \leq p_0.x \\ 0 & \text{if } p_i.x = a \\ p_i.area2(p_{i+1}, a, p_{i+1}.x) & \text{otherwise} \\ & \text{where } i = I.index(a) \text{ and} \end{cases}$$

middle $\stackrel{\text{def}}{=} I.size2(I.index(a), I.index(b))$ and

$$\text{tail} \stackrel{\text{def}}{=} \begin{cases} 2 \cdot (b - p_n.x) \cdot p_n.y & \text{if } b \geq p_n.x \\ 0 & \text{if } p_i.x = b \\ p_i.area2(p_{i+1}, p_i.x, b) & \text{otherwise} \\ & \text{where } i = I.index(b) \end{cases}$$

The method returns a floating point value. ■

The next two methods compute the centre and middle points for a fuzzy interval (Def. 2.10).

Definition 3.29 (Centre Points) Let $I = (p_0, \dots, p_n)$ be an envelope polygon.

$$I.centrePoint(k, m) \stackrel{\text{def}}{=} \begin{cases} \text{undefined} & \text{if } I = () \text{ or } I.isInfinite() \\ p_0.x & \text{if } k = 0 \\ p_n.x & \text{if } k = m \\ p_{i-1}.area2X(p_i, \frac{s \cdot k}{m} - I.size2(0, i - 1)) & \text{otherwise} \\ & \text{where } s \stackrel{\text{def}}{=} I.size2(0, n) \text{ and} \\ & i = \min\{i \mid m \cdot I.size2(0, i) > s2 \cdot k\} \end{cases}$$

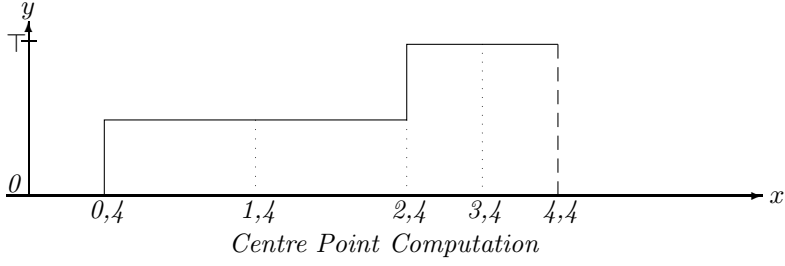
centrePoint returns a (rounded) *x*-integer.

The search for the index *i* in *centrePoint* causes linear complexity for both methods. ■

The *centrePoint* method needs to locate the *x*-coordinate such that $|I|_{-\infty}^x = \frac{k}{m}|I|$. To this end it first locates the index *i* with $p_{i-1} \leq x \leq p_i$. Then it calls the *area2X*-method to calculate the *x*-coordinate *x* with $|I|_{-\infty}^{p_{i-1}.x} + |I|_{p_{i-1}.x}^x = \frac{k}{m}|I|$.

Example 3.30 (Centre Point Computation)

Let $I = (0, 0) (0, 500) (4, 500) (4, 1000) (6, 1000) (6, 0)$



We have $|I| = 4000$, i.e. $s_2 = 8000$, and we want to compute $centrePoint(1, 4)$. The search for $i = \min\{i \mid 4 \cdot I.size2(0, i) > 8000 \cdot 1\}$ yields $i = 2$ because $4 \cdot 4000 > 8000 \cdot 1$.

Since $|I|_{-\infty}^{p_1.x} = 0$ there is still an area the size of 2000 to be covered by $|I|_{p_1.x}^x$.

The call to $p_1.area2X(p_2, \frac{8000 \cdot 1}{4} - 0) = p_1.area2X(p_2, 2000)$ yields 2, such that $x = 2$ is in fact the correct result for $I^{1,4}$. ■

Components of Fuzzy Intervals

The $nComponents$ -method can be used to count the number of components of an interval. It counts the number of times the envelope polygon drops down to an y -value 0 and adds 1 if it is positively infinite.

Definition 3.31 (Number of Components) Let $I = (p_0, \dots, p_n)$ be an envelope polygon.

$$I.nComponents() \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } I = () \text{ or } n = 0 \text{ and } p_0.y = 0 \\ 1 & \text{if } n = 0 \text{ and } p_0.y > 0 \\ \sum_{i=1}^n \begin{cases} 1 & \text{if } p_i.y = 0 \text{ and } p_{i-1}.y > 0 \\ 0 & \text{otherwise} \end{cases} + \begin{cases} 1 & \text{if } p_n.y > 0 \\ 0 & \text{otherwise} \end{cases} \end{cases}$$

The method $component(k)$ below extracts from an envelope polygon the k^{th} component as a new envelope polygon.

Definition 3.32 (component) Let $I = (p_0, \dots, p_n)$ be an envelope polygon.

$$I.component(k) \stackrel{\text{def}}{=} \begin{cases} () & \text{if } I = () \\ V_{i=I.skipComponent(k-1)}^n \begin{cases} (p_i) \text{ and break} & \text{if } p_i.y = 0 \text{ and } p_{i-1}.y > 0 \\ (p_i) & \text{otherwise} \end{cases} \end{cases}$$

where $I.skipComponent(k)$ returns the first index of the $k + 1^{st}$ component.

It is described procedurally.

If $k = 0$ return 0.

If $n = 0$ return 1.

Let $l \stackrel{\text{def}}{=} 0$.

For $i=1$ to n { if $(p_i.y = 0 \text{ and } p_{i-1}.y > 0)$ $l = l + 1$; // next component

if $(l = k)$ { if $(i = n)$ return $n + 1$ // last component skipped

if $(p_{i+1}.y > 0)$ return i // the two components meet at $p_i.x$

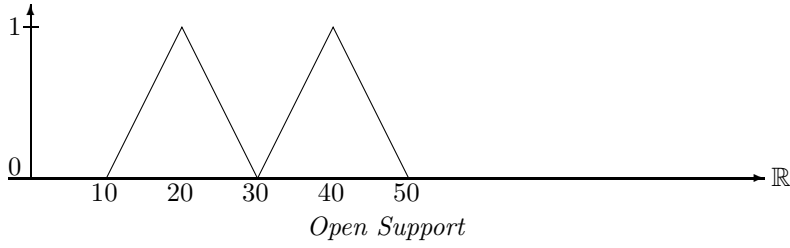
else return $i + 1$. }

return $n + 1$

// last component skipped

3.2.3 Regions

A basic design decision for the algorithms in the FuTI-library is to have only half open crisp intervals of the form $[a, b[$. Unfortunately there are fuzzy intervals with open intervals as support (Def. 2.5). Consider the following interval:



Its support is the open interval $]10, 30[,]30, 50[$. If this is again represented as a fuzzy interval in the FuTI style, it corresponds to the half open interval $[10, 50[$. This is a fundamental weakness of the FuTI-library, which, hopefully, has no practical relevance.

The FuTI-library has therefore the concept of a *region* of a fuzzy interval, which approximates the real regions as half-open intervals.

Definition 3.33 (Region) *We define an enumeration type `Region` with values `core`, `support`, `kernel` and `maximum`. These values indicate the smallest, possibly non-convex, half open interval which contain the corresponding regions of Def. 2.5.* ■

This means that if one of the subsequent functions with parameter `region` computes the corresponding region, the smallest half open interval which contain the corresponding regions of Def. 2.5 is meant.

The components of the regions of an interval can be enumerated with the method `nextComponent`.

Definition 3.34 (nextComponent) *The method `bool I.nextComponent(x1, x2, i1, i2, region)` computes the components of the given region (`core`, `support`, `kernel` or `maximum`) of the interval I one by one. The first component is computed when the variable `i1` has the value `-1`. In this case $x1$ and $x2$ are bound to the x -coordinates of the endpoints of the component. $x1$ and $x2$ may be the infinity. `i1` and `i2` are bound to the indices of the endpoints of the component in the envelope polygon. Successive calls to `nextComponent` yield the next components. The Boolean result of `nextComponent` is true as long as there is still a component which has just been computed.* ■

Notice that `nextComponent` computes for the above example with support $]10, 30[,]30, 50[$ indeed the two components $[10, 30[$ and $[30, 50[$, and not $[10, 50[$.

The FuTI-library provides three more methods for extracting information about the regions of an interval, `size`, `crisp` and `side`:

Definition 3.35 (size) *The method `I.size(region)` measures the size of the given region (`core`, `support`, `kernel` or `maximum`) of the interval I in x -coordinates.* ■

The algorithm uses `I.nextComponent(x1, x2, i1, i2, region)` (Def. 3.34) to enumerate the components of I 's region, and adds up the differences $x2 - x1$. If one of these values is the infinity then the infinity is returned.

Definition 3.36 (crisp) *The method `I.crisp(region)` turns the given region (`core`, `support`, `kernel` or `maximum`) into a - possibly non-convex - crisp interval.* ■

The algorithm uses `I.nextComponent(x1, x2, i1, i2, region)` (Def. 3.34) to enumerate the components of I 's region, and constructs from $x1$ and $x2$ crisp subintervals which are joined to a new interval.

Definition 3.37 (side) *The method `I.side(region, front)` computes the left (if `front = true`) or right (if `front = false`) side of the region (`core`, `support`, `kernel` or `maximum`) of the interval I .*

The result may be the infinity. ■

3.2.4 Point-Interval and Interval-Interval Relations

Point-interval and interval-interval relations for fuzzy intervals should not yield boolean, but fuzzy results. Unfortunately the situation here is even worse than with the set operations for fuzzy intervals. There are no natural and obvious definitions for such relations. There are different and equally plausible versions. Therefore there are no fuzzy point-interval and interval-interval relations within the FuTI-library. Different versions of such relations have been specified in the GeTS language [8].

The FuTI-library contains instead crisp point-interval and interval-interval relations between the regions of the intervals. Since the regions are crisp intervals, there are natural definitions for these relations.

Point–Interval Relations for Regions

The relations between a single time point and a, possibly non-convex, (crisp) region of a fuzzy interval are: *before*, *starts*, *during*, *finishes*, *after* and *between*. The point–interval relations *before* and *after* respect that the regions are treated as half open intervals $[a\dots b[$. That means t *before* $[a\dots b[$ is only true if $t < a$. t *after* $[a\dots b[$ is true if $t \geq b$.

The *between*–relation makes sense for non-convex regions of intervals. t *between* I means that t is in a gap between the components of the corresponding region of I . Notice that for the above interval I with support $]10, 30[,]30, 50[$ we get $(30 \text{ between } I) = \text{false}$ because the **support** region (Def. 3.33) is $[10, 50[$ in this case.

Interval–Interval Relations

For crisp intervals there is the standard set of Allen’s interval–interval relations (Fig. 1) [1].

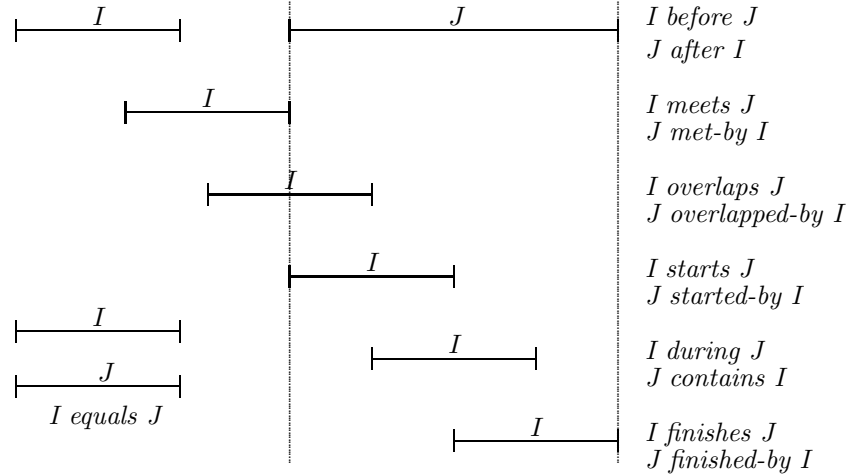


Figure 1: Interval–Interval Relations

These relations are defined for convex crisp intervals. Since the FuTI–library has to deal with non-convex crisp intervals as well, it has to generalise these relations to the non-convex case. The generalisation is straightforward for *before*, *meets*, *during* and *after*. The condition for *I overlaps J* is that a part of I is before J and the rest of I is during J , and this version is implemented in FuTI. The condition for *I starts J* is that the left ends of I and J coincide and I is during J , and this works also for non-convex intervals. The analogous conditions hold for the *finishes* relation.

There is a further subtlety to be considered when implementing interval–interval relations. Allen’s interval–interval relations come with a constraint calculus which relies heavily on the condition that the relations are disjoint. That means, two intervals I and J stand in exactly one of the relations to each other. For example, $[0, 10[$ meets $[10, 20[$, and therefore $[0, 10[$ before $[10, 20[$ must be false. The disjointness of the relations is necessary for the constraint calculus. For other applications it not very intuitive. Therefore the interval–interval relations in the FuTI–library are defined in the more intuitive way where both $[0, 10[$ meets $[10, 20[$, and $[0, 10[$ before $[10, 20[$ are true. By similar reasons, the *during* relation is defined such that $\text{starts} \subseteq \text{during}$ and $\text{finishes} \subseteq \text{during}$.

The FuTI–library provides two sets of interval–interval relations. One set is between ordinary intervals $[a, b[$ and regions of fuzzy intervals. The second set is between the corresponding regions of two fuzzy intervals. The algorithms have either constant or at worst linear complexity. In particular the algorithm for *during* uses a sweep line technique for going only once through the two envelope polygons.

3.2.5 Hull Operators

The method $I.\text{crispHull}()$ implements the $\text{crispHull}()$ -function (Def. 2.28).

Definition 3.38 (Crisp Hull) Let $I = (p_0, \dots, p_n)$ be an envelope polygon.

$$I.\text{crispHull}() \stackrel{\text{def}}{=} \begin{cases} () & \text{if } I = () \\ \left(\begin{cases} ((p_0.x, \top)) & \text{if } p_0.y > 0 \\ ((p_0.x, 0)(p_0.x, \top)) & \text{otherwise} \end{cases} \right) & \text{if } p_n.y > 0 \\ \left(\begin{cases} ((p_n.x, \top)) & \text{if } p_n.y > 0 \\ ((p_n.x, \top)(p_n.x, 0)) & \text{otherwise} \end{cases} \right) & \text{otherwise} \end{cases}$$

■

The method $I.monotoneHull()$ implements the $monotoneHull()$ -function (Def. 2.30). The algorithm scans the envelope polygon first from 0 to the first maximal element and skips all vertices which destroy monotonicity. Then it scans the envelope polygon from the last element to the last maximal element and skips again all vertices which destroy monotonicity. Finally it appends the first lists with the reversed second list.

Definition 3.39 (Monotone Hull) *Let $I = (p_0, \dots, p_n)$ be an envelope polygon. We describe the algorithm $I.monotoneHull()$ procedurally:*

If $I = ()$ return $()$;

$$\text{Let } newI_1 \stackrel{\text{def}}{=} (p_0, V_{i=0}^{I.indexMax(true)}) \begin{cases} ((p_{i-1}.lineX(p_i, max), max), p_i) & \text{if } i > 0 \text{ and } p_i.y \geq max \text{ and } p_{i-1}.y < max \\ (p_i) & \text{if } p_i.y \geq max \\ () & \text{otherwise} \end{cases}$$

where max is the current largest y -coordinate in $newI_1$:

$$\text{Let } newI_2 \stackrel{\text{def}}{=} (p_n, V_{i=n}^{I.indexMax(false)}) \begin{cases} ((p_{i-1}.lineX(p_i, max), max), p_i) & \text{if } i < n \text{ and } p_i.y \geq max \text{ and } p_{i-1}.y < max \\ (p_i) & \text{if } p_i.y \geq max \\ () & \text{otherwise} \end{cases}$$

where max is now the current largest y -coordinate in $newI_2$:

Let $newI_2 = (q_0, \dots, q_m)$;

For $i=m$ to 0 $newI_1.push_back(q_i)$.

return $newI_1$.

■

The algorithm for the convex hull function $convexHull$ (Def. 2.29) is a special version of the *Graham Scan* algorithm for arbitrary polygons. It goes from left to right through the envelope polygon and pushes all candidates for the convex hull on a stack. Wrong candidates are later popped from the stack. Since the points are already sorted, its complexity is linear.

Definition 3.40 (Convex Hull) *Let $I = (p_0, \dots, p_n)$ be an envelope polygon. We describe the algorithm $I.convexHull()$ procedurally:*

If $I = ()$ return $()$.

$$\text{Let } f \stackrel{\text{def}}{=} \begin{cases} I.indexMax(true) & \text{if } p_0.y > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\text{Let } l \stackrel{\text{def}}{=} \begin{cases} I.indexMax(false) & \text{if } p_n.y > 0 \\ n & \text{otherwise} \end{cases}$$

Let $newI \stackrel{\text{def}}{=} (p_{i_f})$.

*For $i=f$ to l while $(m \geq 1 \text{ and } q_{m-1}.leftturn(q_m, p_i))$ $newI.pop_back()$;
 $newI.push_back(p_i)$;*

where q_m is the current last element of $newI$.

return $newI$.

■

3.2.6 Basic Unary Transformations

A number of basic unary transformations (Def. 2.31) can be implemented by just manipulating the vertices of the envelope polygons.

Definition 3.41 (Extend, Scaleup, Shift) *Let $I = (p_0, \dots, p_n)$ be an envelope polygon. The three functions return $()$ if $I = ()$. Let $M \stackrel{\text{def}}{=} (q_0, \dots, q_k) \stackrel{\text{def}}{=} I.monotoneHull()$.*

$$I.extend(true) \stackrel{\text{def}}{=} (V_{i=0}^j(q_i), (q_j.x, \top)) \\ \text{where } j = M.indexMax(true)$$

$$I.extend(false) \stackrel{\text{def}}{=} ((q_j.x, \top), V_{i=j}^k(q_i)) \\ \text{where } j = M.indexMax(false)$$

$$I.scaleUp() \stackrel{\text{def}}{=} V_{i=0}^n(p_i.x, \text{roundY}((p_i.y \cdot \top / I.sup())))$$

$$I.shift(a) \stackrel{\text{def}}{=} V_{i=0}^n(p_i.x + a, p_i.y)$$

■

$extend(true)$ implements $extend^+$, $extend(false)$ implements $extend^-$, (Def. 2.31).

cut

We provide three *cut*-methods. The first one cuts an envelope polygon between two given x -coordinates x_1 and x_2 . The second one cuts it between the x -coordinates of two given vertices. The third one cuts the interval after or before an x -coordinate.

Definition 3.42 (cut) Let $I = (p_0, \dots, p_n)$ be an envelope polygon. x , x_1 and x_2 are x -coordinates.

$$I.cut(x_1, x_2) \stackrel{\text{def}}{=} \begin{cases} () & \text{if } x_2 \leq x_1 \\ ((x_1, 0), (x_1, I.member(x_1)), (V_{i=I.index(x_1)}^{I.index(x_2)} p_i), (x_2, I.member(x_2)), (x_2, 0)) & \text{otherwise} \end{cases}$$

where the list is formed with the `push_back` operator (Def. 3.18). This removes certain redundancies.

Let i_1 and i_2 be two indices.

$$I.cutI(i_1, i_2) \stackrel{\text{def}}{=} \begin{cases} () & \text{if } i_2 \leq i_1 \\ V_{i=i_1}^{i_2}(p_i) & \text{otherwise.} \end{cases}$$

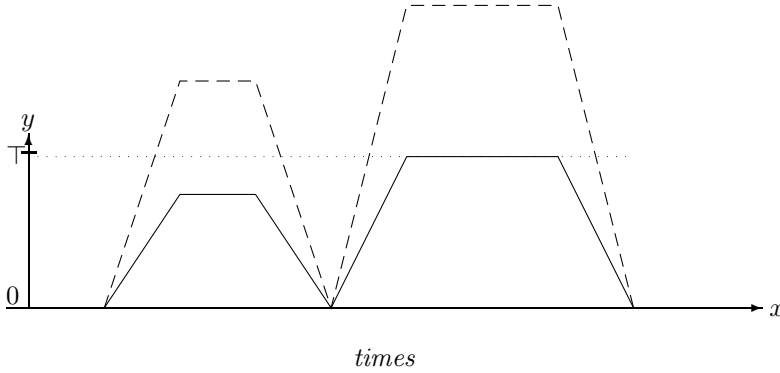
$$I.cut(x, true) \stackrel{\text{def}}{=} ((x, 0), (x, roundY(I.member(x))), V_{i=I.index(x)}^n p_i)$$

$$I.cut(x, false) \stackrel{\text{def}}{=} (V_0^{i=I.index(x)} p_i, (x, roundY(I.member(x))), (x, 0))$$

■

times

The *times* operator, which multiplies the membership function with a constant, is not so easy to implement. Since $y \cdot a > \top$ is possible, one has to cut the multiplied envelope polygon at $y = \top$. The picture below illustrates the problem.



In order to cut the multiplied polygon at $y = \top$ the intersection points between the dotted and dashed lines have to be computed. The *times* function defined below follows the line segments of the envelope polygon I and checks whether the multiplied line segments cross the $y = \top$ line. In this case the intersection points are computed and inserted into the transformed polygon.

Definition 3.43 (times) Let $I = (p_0, \dots, p_n)$ be an envelope polygon and a a non-negative floating point number.

$$I.times(a) \stackrel{\text{def}}{=} \begin{cases} () & \text{if } I = () \\ \left((p_0.x, \min(1, p_0.y \cdot a)), \right. & \\ \left. V_{i=1}^n \left\{ \begin{array}{l} () \\ ((x, \top)) \\ ((x, \top), (p_i.x, p_i.y \cdot a)) \\ ((p_i.x, p_i.y \cdot a)) \end{array} \right. \right) & \begin{array}{l} \text{if } p_{i-1}.y \cdot a \geq \top \text{ and } p_i.y \cdot a > \top \\ \text{if } p_i.y \cdot a < \top \text{ and } p_{i-1}.y \cdot a > \top \\ \text{if } p_i.y \cdot a > \top \text{ and } p_{i-1}.y \cdot a < \top \\ \text{where } x = p_{i-1}.lineX(p_i, \top) \\ \text{otherwise} \end{array} \end{cases}$$

■

Interpolation

Some of the transformations of fuzzy time intervals are non-linear in the sense that they transform straight lines into curved lines. These transformations cannot be implemented by simply transforming the vertices

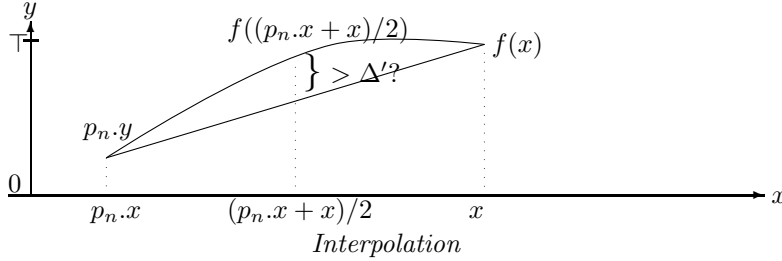
of the envelope polygons. Since the result of the transformations must be envelope polygons, we need to approximate curved lines by polygons. To this end we define a method *interpolate* which interpolates curved lines between vertices of polygons.

Definition 3.44 (Interpolation) Let $I = (p_0, \dots, p_n)$ be a non-empty envelope polygon, x an x -coordinate, f a function from x -coordinate \mapsto y -coordinate and Δ a threshold value (e.g. $\Delta = 0.1$).

$$I.interpolate(x, f, \Delta) \stackrel{\text{def}}{=} \begin{cases} I & \text{if } x \leq p_n.x \\ I.interpolate(\text{round}X((p_n.x + x)/2), f, \Delta).interpolate(x, f, \Delta) & \text{if } |2y_1 - y_2| > \Delta'y_2 \\ & \text{where } y_1 = f(\text{round}X((p_n.x + x)/2)) \text{ and } y_2 = p_n.y + f(x) \text{ and } \Delta' = \Delta/(1 + 2y_2/\top) \\ I.push_back((x, f(x))) & \text{otherwise} \end{cases}$$

■

The *interpolate*-method starts with an envelope polygon $I = ((x_0, y_0))$ and fills up I with interpolated values. Suppose $I = (p_0, \dots, p_n)$. For a given $x > p_n.x$ it checks whether the relative difference between the middle point $(p_n.x + x)/2$ of the straight line between p_n and $(x, f(x))$, and $f((p_n.x + x)/2)$ is larger than Δ . If this is not the case then the approximation is good enough and the point $(x, f(x))$ is pushed onto I . If this is the case, better interpolation is necessary. Therefore it calls itself recursively with $x =$ middle point to fill up I until the middle point, and then with x itself to fill up I from the middle point until the actual x . The threshold Δ is only a basic threshold for very small y -values. The threshold Δ' causes that the interpolation becomes denser for larger y -values.



Integration

The *integrate*⁺-function (Def. 2.31) is implemented by the *integrate(true)*-method below and the *integrate*⁻-function is implemented by the *integrate(false)*-method. *integrate(true)* goes from left to right through the envelope polygon I and calls for each line segment the *area2*-function for points (Def. 3.9). *integrate(false)* goes from right to left through the polygon. Therefore the resulting list has to be reversed. Since line segments are linear, their integration yields a quadratic curve. Therefore interpolation is necessary.

Definition 3.45 (Integration) Let $I = (p_0, \dots, p_n)$ be an envelope polygon and Δ the threshold. We write the function again in a procedural style.

I.integrate(true):

if $I = ()$ then return $()$

Let $newI \stackrel{\text{def}}{=} (p_0.x, 0)$

For _{$i=1$} ^{n} $newI.interpolate(p_i.x, \lambda(x)(q_m.y + p_{i-1}.area2(p_i, x, true)/2), \Delta)$
where $newI = (q_0, \dots, q_m)$

return $newI$.

I.integrate(false):

if $I = ()$ then return $()$

Let $newI \stackrel{\text{def}}{=} (p_n.x, 0)$

For _{$i=n$} ^{1} $newI.interpolate(p_{i-1}.x, \lambda(x)(q_m.y + p_{i-1}.area2(p_i, x, false)/2), \Delta)$
where $newI = (q_0, \dots, q_m)$

return $newI$ reversed.

■

3.2.7 Y-Function Based Unary Transformations

For unary transformations of fuzzy intervals which can be generated by applying a y -function to the membership values, there is a simple algorithm scheme: if the y -function is linear, apply it to the y -coordinates of the envelope polygon; if the function is not linear, use the *interpolate* method.

Definition 3.46 (Unary Transformation) Let $I = (p_0, \dots, p_n)$ be an envelope polygon and let f be a unary y -function and Δ a threshold value. We describe the method $I.unaryTransformation(f, \Delta)$ procedurally:

If $I = ()$ return $()$;
 If f is linear then return $V_{i=0}^n(p_i.x, f(p_i.y))$.
 Otherwise:
 Let $newI \stackrel{\text{def}}{=} (p_0.x, f(p_0.y))$;
 For $i=1$ to n $newI.interpolate(p_i.x, \lambda(x)f(p_{i-1}.lineY(p_i, x)), \Delta)$;
 return $newI$; ■

Exponentiation

The exponentiation operator $exponentiate_e(i)$ (Def. 2.31) is the first non-linear transformation we consider here.

Definition 3.47 (Exponentiation) Let $I = (p_0, \dots, p_n)$ be an envelope polygon, e a non-negative number (the exponent) and Δ the threshold.

$I.Exp(e) \stackrel{\text{def}}{=} I.unaryTransformation(\lambda(y)y^e, \Delta)$.
 $\lambda(y)y^e$ is not linear. ■

Complement Operator

Another point-based transformation is the complement operator.

Definition 3.48 (complement) Let $I = (p_0, \dots, p_n)$ be an envelope polygon, n a negation function (Def. 2.18) and Δ the threshold.

$I.complement(n) \stackrel{\text{def}}{=} I.unaryTransformation(n, \Delta)$. ■

3.2.8 Y-Function Based Binary Transformations

Y-Function based binary transformations of fuzzy intervals are more complicated to implement because besides the vertices of the two envelope polygons their intersection points are relevant for the transformation. The intersection points may become vertices of the transformed envelope polygons. Therefore the first thing the binary transformation algorithm must do is to compute the intersection points of the two polygons. Fortunately, since the two polygons are unimonotone, this can be done with a sweep line algorithm in linear time. The result of the *IntersectionPoints*-algorithm defined below is a list $((p_0, q_0), \dots)$ of pairs of points. The p_i are the vertices of I_1 and the intersection points between I_1 and I_2 . The q_i are the vertices of I_2 and also the intersection points between I_1 and I_2 . $p_i.x = q_i.x$ holds for all i .

In order to simplify the presentation of the algorithm a little bit we assume that I_1 and I_2 start at the same x -coordinates, and that both polygons have a redundant extra point p_{n+1} and q_{m+1} at the end. This saves some case distinctions at the beginning and at the end of the sweep.

Definition 3.49 (Intersection Points) Let $I_1 = (p_0, \dots, p_{n+1})$ and $I_2 = (q_0, \dots, q_{m+1})$ be two envelope polygons such that $p_0.x = q_0.x$ and $p_n.y = p_{n+1}.y$ and $q_m.y = q_{m+1}.y$. We define the method $I_1.IntersectionPoints(I_2)$. It returns a list of pairs $((p_0, q_0), \dots)$.

Let $IntP \stackrel{\text{def}}{=} ()$.
 Let $i \stackrel{\text{def}}{=} 0$ and $j \stackrel{\text{def}}{=} 0$
 Let $x \stackrel{\text{def}}{=} p_0.x$ (x is the position of the sweep line).

```

while( $x \leq \max(p_n.x, q_m.x)$ ){
  if( $i < n$  and  $p_i.x = p_{i+1}.x$ )
    if( $x = q_j.x$ ){
      IntP.push_back( $p_i, q_j$ );  $i := i + 1$ ; }
      if( $j < m$  and  $q_j.x = q_{j+1}.x$ )  $j := j + 1$ ; }
    else{IntP.push_back( $p_i, (x, \text{roundY}(q_j.\text{lineY}(q_{j+1}, x)))$ );  $i := i + 1$ ; }
    continue; }

if( $j < m$  and  $q_j.x = q_{j+1}.x$ ){
  IntP.push_back( $p_i, q_j$ );
  IntP.push_back( $p_i, q_{j+1}$ );  $j := j + 1$ 
  continue; }

if( $x = q_j.x$ )IntP.push_back( $p_i, p_j$ )
else IntP.push_back( $p_i, (x, \text{roundY}(q_j.\text{lineY}(q_{j+1}, x)))$ );

if( $i < n$ ){
  if( $j < m$ ){
    if( $p_i.\text{intersectsProper}(p_{i+1}, q_j, q_{j+1})$ ){
       $xint := p_i.\text{intersection}(p_{i+1}, q_j, q_{j+1})$ ;
      IntP.push_back( $(xint, \text{roundY}(p_i.\text{lineY}(p_{i+1}, xint)))$ ,
        ( $xint, \text{roundY}(q_j.\text{lineY}(q_{j+1}, xint)))$ ); }

      if( $p_i.x < q_j.x$ ){ $x = p_{i+1}.x$ ;  $i := i + 1$ ; continue; }
      if( $p_i.x = q_j.x$ ){ $x = p_{i+1}.x$ ;  $j := j + 1$ ; continue; }
       $x = q_{j+1}.x$ ; continue; }
     $x = p_{i+1}.x$ ; continue; }

  if( $j < m$ ){ $x := q_{j+1}.x$ ; continue; }
   $x := x + 1$ ; }

return IntP.

```

■

We can now define the *binaryTransformation*-method. It works much like the *unaryTransformation*-method. The differences are that *binaryTransformation* first needs to compute the intersection points, and that the call to the *interpolate*-method gets as input a function which is parameterised with two line segments instead of one.

Definition 3.50 (Binary Transformation) Let $I_1 = (p_0, \dots, p_{n_1})$ and $I_2 = (q_0, \dots, q_{n_2})$ be envelope polygons. Let f be a binary y -function and Δ a threshold value.

We describe the method $I_1.\text{binaryTransformation}(I_2, f, \Delta)$ procedurally:

Let $I \stackrel{\text{def}}{=} ((p_0, q_0), \dots, (p_n, q_n)) = I_1.\text{IntersectionPoints}(I_2)$

If $I = ()$ return $()$;

If f is linear then return $V_{i=0}^n(p_i.x, f(p_i.y, q_i.y))$.

Otherwise:

Let $\text{newI} \stackrel{\text{def}}{=} (p_0.x, f(p_0.y, q_0.y))$;

For $i=1$ to n $\text{newI}.\text{interpolate}(p_i.x, \lambda(x)f(p_{i-1}.\text{lineY}((p_i, x), q_{i-1}.\text{lineY}((q_i, x)), \Delta))$;

return newI ;

■

3.3 Integration over Multiplied Intervals

The motivation for the operators in this section come from certain fuzzy relations between fuzzy intervals. There is no unique generalisation of interval–interval relations like ‘before’ to fuzzy intervals. One idea for the generalisation works in two steps. The first step is to define a point–interval ‘before’-relation: $PI\text{before}(x, I)$. This can also be done in different ways. Regardless how the concrete definition is, it is always possible to define this as an operator which maps an interval to an interval: $PI\text{before}'(I)(x) \stackrel{\text{def}}{=} PI\text{before}(x, I)$. We can now generalise the point–interval ‘before’ relation to an interval–interval ‘before’-relation $II\text{before}(I, J)$ by averaging the point–interval ‘before’ relation over J : $II\text{before}(I, J) \stackrel{\text{def}}{=} \int I(x) \cdot PI\text{before}'(J)(x) dx / N(I, J)$. $N(I, J)$ is a normalisation factor which forces the result to be a fuzzy value between 0 and 1. More details about fuzzy point–interval and interval–interval relations can be found in [8].

FuTI provides two different integration operations which can be used for these purposes. We start with an auxiliary definition, a parameterised integration over multiplied membership functions.

Definition 3.51 (Integration over Multiplied Intervals) Let $I = (p_0, \dots, p_n)$ and $J = (q_0, \dots, q_m)$ be envelope polygons. Let a be an x -coordinate. The integral $I.integrate(J, a) \stackrel{\text{def}}{=} \int I(x - a)J(x) dx$ is computed as follows:

```

If  $I = ()$  or  $J = ()$  then return 0;
If ( $I.isInfinite()$  and  $J.isInfinite()$ ) then undefined;
Let  $Int = 0$ ;
If  $(p_0.x + a \leq q_0.x)$  then  $\{j = 0; i = I.index(q_0.x - a); x = q_0.x;$ 
     $Int = q_0.y \cdot I.size2(p_0.x, q_0.x - a)/2;\}$ 
else  $\{i = 0; j = J.index(p_0.x); x = p_0.x + a;$ 
     $Int = p_0.y \cdot J.size2(q_0.x, p_0.x + a)/2;\}$ 
while ( $i < n$  and  $j < m$ ) {
     $Int = Int + p'_i.integrate(p'_{i+1}, q_j, q_{j+1}, x, \min(p_{i+1}.x + a, q_{j+1}.x));$  //Def.3.12
    where  $p'_i \stackrel{\text{def}}{=} (p_i.x + a, p_i.y)$  and  $p'_{i+1} \stackrel{\text{def}}{=} (p_{i+1}.x + a, p_{i+1}.y)$ 
     $x = \min(p_{i+1}.x + a, q_{j+1}.x);$ 
    if  $(x = p_{i+1}.x + a)$   $i = i + 1;$ 
    if  $(x = q_{j+1}.x)$   $j = j + 1;$ 
}
if  $(p_n.x + a \leq q_m.x)$   $Int = Int + p_n.y \cdot J.size2(p_n.x + a, q_m.x)/2;$ 
else  $Int = Int + q_m.y \cdot I.size2(q_m.x - a, p_n.x)/2;$ 
return  $Int$ ;

```

Asymmetric integration integrates over the multiplied membership functions of I and J and normalises the result with the size of I .

Definition 3.52 (Asymmetric Integration) Let I and J be two fuzzy polygons. I must be finite. The method

$$I.integrateAsymmetric(J) \stackrel{\text{def}}{=} \text{roundY}\left(\frac{2 \cdot I.integrate(J, 0)}{I.size2()}\right)$$

computes $\int I(x) \cdot J(x) dx / |I|$.

The ‘symmetric integration’ over multiplied envelope polygons differs from the asymmetric integration by the normalisation factor. The normalisation factor $maximizeOverlap(I, J)$ below causes that there is a position of the interval I relative to the interval J such that the value of the normalised integral is 1. This is a useful operation for defining a fuzzy interval–interval ‘meets’ relation. It guarantees that if (a finite) I is shifted along the time axis, eventually it meets (a finite) J with resulting fuzzy value 1.

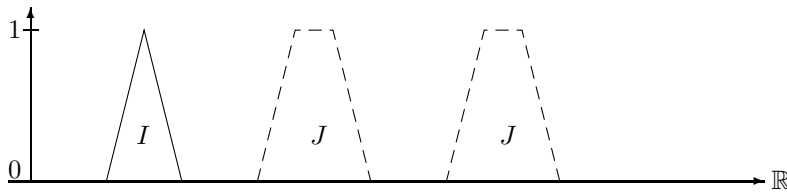
Definition 3.53 (Symmetric Integration) Let I and J be two envelope polygons.

The function $I.integrateSymmetric(J, simple)$ computes $\int I(x) \cdot J(x) dx / N$

where $N = \begin{cases} \min(|I|, |J|) & \text{if } simple = true \\ \max\{a \mid \int I(x - a) \cdot J(x) dx\} & \text{otherwise} \end{cases}$

$$I.integrateSymmetric(J, simple) \stackrel{\text{def}}{=} \begin{cases} \text{undefined} & \text{if } I.isInfinite() \text{ or } J.isInfinite() \\ \text{roundY}\left(\frac{2 \cdot I.integrate(J, 0)}{\min(I.size2(), J.size2())}\right) & \text{if } simple = true \\ \text{roundY}\left(\frac{\top \cdot I.integrate(J, 0)}{maximizeOverlap(I, J)}\right) & \text{otherwise (Def. 3.56 below)} \end{cases}$$

The normalisation factor $maximizeOverlap(I, J) = \max\{a \mid \int I(x - a) \cdot J(x) dx\}$, where I is finite, amounts in general to a nontrivial search problem with unpredictable solutions. Consider the following example:



Maximising the Overlap

If we move I into the left component of J we get maximal overlap as long as I is completely contained in this part of J . The same holds for the right part of J .

For the parameter a to be maximised in the integral we get two plateaux as solutions.

There seems to be no easy analytical solution to this problem. Fortunately there are important classes of fuzzy time intervals, where this problem is extremely easy to solve.

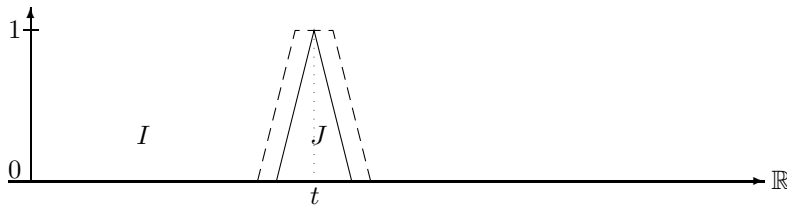
The first class is when J is infinite and $J(-\infty) = \top$ or $J(+\infty) = \top$, and, of course J has a finite kernel. In this case one can move I to the infinite part where J is constant 1. $\int I(x-a)J(x) dx = |I|$ in this case, i.e. $\max_a \int I(x-a) \cdot J(x) dx = |I|$,

The other class are the the *symmetric* and *monotone* fuzzy intervals.

Definition 3.54 (Symmetric and Monotone Intervals) A fuzzy time interval I is symmetric if there is a time point t such that $I(t-x) = I(t+x)$ for all x holds. t is the symmetry axis.

A fuzzy time interval I is monotone if with increasing time coordinate x , $I(x)$ is monotonically increasing until a maximal value and then it is monotonically decreasing again. ■

Crisp intervals are in particular monotone and symmetric. Maximal overlap is achieved for monotone and symmetric intervals if the symmetry axes of both intervals coincide.

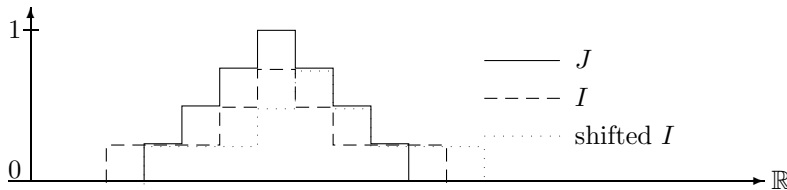


Maximal Overlap

Proposition 3.55 If I and J are two monotone and symmetric fuzzy intervals then $\int I(x)J(x) dx$ is maximal if the symmetry axis of I and J coincide. ■

The proof is very technical. We therefore sketch only the basic idea. First I and J are discretised into step functions with finite step size. The limit ‘step size $\mapsto 0$ ’ is then the original problem. The discretised integral then becomes a sum $stepsize \cdot \sum_k I_k \cdot J_k$

One must show that moving the interval I away from the position where the two symmetry axes coincide, decreases the sum.



Discretised Maximisation Problem

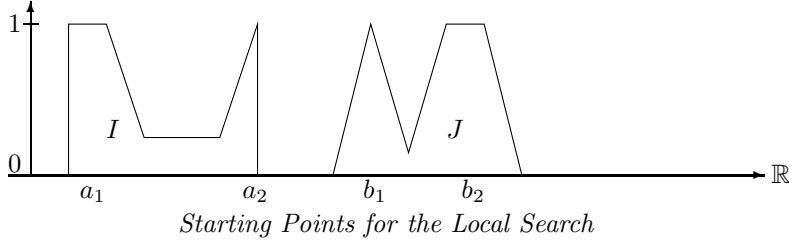
As one can see in this picture, shifting I to the right hand side, decreases the parts of the sum $I_k \cdot J_k$ on the left side of the symmetry axis of J , and increases the parts of the sum on the right side of the symmetry axis. The important observation is, that because J is monotone falling at the right hand side, the parts I_k on the right side, which cause the sum to increase again, are multiplied with smaller J_k than the corresponding parts on the left hand side. Therefore the sum gains less on the right hand side than it loses on the left hand side. The overall sum therefore decreases or remains constant.

A General Search Procedure

We want to find a value for a such that $\int I(x-a)J(x) dx$ is maximal. If I or J are not monotone and

symmetric a general search procedure has to be applied. The search procedure which is implemented in FuTI is a combination of an iterated binary local search with a randomised global search. It is optimised for search spaces with little structure and terminates quickly. 100% success, however, is not guaranteed.

The first problem to be solved is to find good starting points for the search. Reasonable choices are the middle points of the local maxima of I and J . For the examples in the picture below the search starts by matching the four combinations of a_k with b_l .



Since all these combinations may miss the global maximum, random start points are also generated.

The second problem is to choose an initial step size for the search. The initial step size is $\Delta = \min(J^{LS} - b_0, b_0 - J^{fS})/2$, i.e. half way between the start point b_0 of the search in the interval J and the closest end of J .

If, for example, the value for the integral increases for $a_0 + \Delta$ then the local search procedure is called recursively for the initial value $a_0 + \Delta$ and step size $\Delta/2$. The other cases are similar. This way Δ is decreased exponentially until it reaches a certain threshold. The new value for a is now the start point of another local search with the same Δ as before. This is iterated until the changes in the integral falls under another threshold (1% seemed to be a good choice).

Definition 3.56 (The Search for Maximising the Overlap) *Let I and J be two finite fuzzy intervals. We define a local search function and then a global search procedure for maximising the integral*

$$Int(a) \stackrel{\text{def}}{=} \int I(x - a)J(x) dx.$$

Let a be the start value for the search and Δ the step size. ‘threshold’ is threshold for Δ .

$localSearch(a, \Delta)$

$$\stackrel{\text{def}}{=} \begin{cases} (a, Int(a)) & \text{if } \Delta \leq \text{threshold} \\ localSearch(a + \Delta, \Delta/2) & \text{if } Int(a + \Delta) > Int(a) \text{ and } Int(a + \Delta) \geq Int(a - \Delta) \\ localSearch(a - \Delta, \Delta/2) & \text{if } Int(a - \Delta) > Int(a) \text{ and } Int(a - \Delta) \geq Int(a + \Delta) \\ localSearch(a, \Delta/2) & \text{otherwise} \end{cases}$$

$iteratedLocalSearch(a, \Delta)$: *iterate $(a, Int) := localSearch(a, \Delta)$ until the changes in Int falls under a threshold. return Int .*

The global search procedure $maximizeOverlap(I, J)$ is described procedurally:

For all combinations m_i and n_j of middle points of local maxima of I and J :

let $\Delta = \min(J^{LS} - n_j, n_j - J^{fS})/2$, call $Int = iteratedLocalSearch(n_j - m_i, \Delta)$ and choose the maximal Int -value.

Repeat this k times with randomly chosen m_i and n_j and choose again the maximal Int -value. ($k = 5$ seemed to be enough.)

return the maximal Int -value. ■

4 Summary

This report is a detailed description of the FuTI-library. This library is a C++-package for representing and manipulating fuzzy time intervals. The mathematical background, the concrete data structures and algorithms, and the interface to the library are described. The FuTI-library is used in the GeTS language [6]. This language in turn is then used to define point-interval and interval-interval relations for fuzzy intervals [8].

References

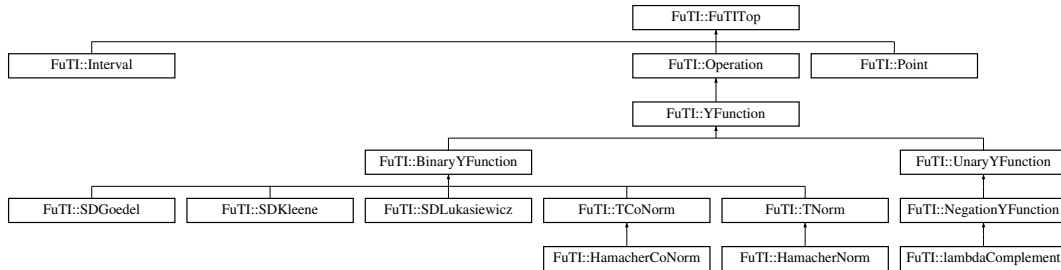
- [1] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.
- [2] Didier Dubois and Henri Prade, editors. *Fundamentals of Fuzzy Sets*. Kluwer Academic Publisher, 2000.
- [3] Jacob E. Goodman and Joseph O’Rourke, editors. *Handbook of Discrete and Computational Geometry*. CRC Press, 1997.
- [4] Hans Jürgen Ohlbach. Computational treatment of temporal notions – the CTTN-system. Research Report PMS-FB-2005-30, Inst. für Informatik, LFE PMS, University of Munich, June 2005. URL: <http://www.pms.ifl.lmu.de/publikationen/#PMS-FB-2005-30>.
- [5] Hans Jürgen Ohlbach. Fuzzy time intervals – the FuTI-library. Research Report PMS-FB-2005-26, Inst. für Informatik, LFE PMS, University of Munich, June 2005. URL: <http://www.pms.ifl.lmu.de/publikationen/#PMS-FB-2005-26>.
- [6] Hans Jürgen Ohlbach. GeTS – a specification language for geo-temporal notions. Research Report PMS-FB-2005-29, Inst. für Informatik, LFE PMS, University of Munich, June 2005. URL: <http://www.pms.ifl.lmu.de/publikationen/#PMS-FB-2005-29>.
- [7] Hans Jürgen Ohlbach. Modelling periodic temporal notions by labelled partitionings of the real numbers – the PartLib library. Research Report PMS-FB-2005-28, Inst. für Informatik, LFE PMS, University of Munich, June 2005. URL: <http://www.pms.ifl.lmu.de/publikationen/#PMS-FB-2005-28>.
- [8] Hans Jürgen Ohlbach. Relations between fuzzy time intervals. Research Report PMS-FB-2005-27, Inst. für Informatik, LFE PMS, University of Munich, June 2005. URL: <http://www.pms.ifl.lmu.de/publikationen/#PMS-FB-2005-27>.
- [9] Joseph O’Rourke. *Computational Geometry in C*. Cambridge University Press, 1998.
- [10] L. A. Zadeh. Fuzzy sets. *Information & Control*, 8:338–353, 1965.

Appendix

A The FuTI-interface

This is a very brief overview of the FuTI-interface. A much more detailed documentation is available as doxygen generated html and latex files.

The FuTI-interface consists of the main classes *Point*, *Interval*, *Operation* and *YFunction*. Additionally there are a number of auxiliary classes. The top class for all classes in the CTTN system is *FuTITop*. In addition there is a namespace *Service* which is imported from the CTTN system. This class contains all the general purpose functions which cannot be associated to a particular component system. The *YFunction* has also a number of subclasses. The complete class hierarchy is as follows:



For the main classes we list the constructor methods, the main public methods and explain briefly what they do. The syntax we use in this section is simplified C++ or Java. The precise syntax is of course in the corresponding header or class files.

CX is the datatype of the x -coordinates (long long integers or multiple precision integers) and CY is the datatype of the y -coordinates (typically unsigned short integers). CX and CY are compiler options.

Many of the methods represent partial functions. FuTI has a DEBUG mode (compiler option) where the necessary preconditions are checked and an error is thrown when the preconditions are not met. If the DEBUG mode is turned off, only the errors which can be caused by data and not by program errors are still caught.

A.1 Points

This class represents 2-dimensional points with coordinates of type CX and CY.

Constructors

`Point(CX x, CY y)`

constructs a point from x and y -coordinates.

`Point(string s)`

reconstructs a point from a string representation " x,y ".

Predicates

`bool p.leftturn(Point q, Point r)`

true if $p \rightarrow q \rightarrow r$ is a left turn or collinear.

(Def. 3.4)

`bool p.leftturnProper(Point q, Point r)`

true if $p \rightarrow q \rightarrow r$ is a proper left turn.

`bool p.rightturn(Point q, Point r)`

true if $p \rightarrow q \rightarrow r$ is a right turn or collinear.

`bool p.rightturnProper(Point q, Point r)`

true if $p \rightarrow q \rightarrow r$ is a proper right turn.

`bool p.collinear(Point q, Point r)`

true if $p \rightarrow q \rightarrow r$ is collinear

(Def. 3.3)

`bool p.collinear(Point q, Point r, Point s)`

true if the line segment (p, q) is collinear with the line segment (r, s) .

`bool p.between(Point q, Point r)`

true if p is between q and r .

bool p.betweenProper(Point q, Point r)
true if p is between q and r, but different to q and r.

bool p.intersects(Point q, Point r, Point s)
true if the line (p, q) intersects the line (r, s).

bool p.intersectsProper(Point q, Point r, Point s)
true if the line (p, q) intersects the line (r, s), but does not only touch it. (Def. 3.5)

Computations

CX p.intersection(Point q, Point r, Point s)
computes the intersection point for the line segments (p, q) and (r, s). An error is thrown if the line segments do not intersect! (Def. 3.6)

float p.lineY(Point q, CX x)
computes for the line crossing p and q the y -value at point x. An error is thrown if the line is vertical. (Def. 3.7)

CX p.lineX(Point q, CY y)
computes for the line crossing p and q the x -value at point y. An error is thrown if the line is horizontal. (Def. 3.8)

CX p.area2(Point q)
computes the area below the line segment (p, q). (Def. 3.9)

float p.area2(Point q, CX x1, CX x2)
computes the area below the line segment (p, q) from x1 until x2. It throws an error if the line is vertical and $x1 \neq x2$ (Def. 3.9)

CX p.area2X(Point q, float a)
computes the x -coordinate x such that the area below the line segment (p, q) from p until x is just a. An error is thrown if there is not enough area below the line segment. (Def. 3.10)

float p1.integrate(Point p2, Point q1, Point q2, CX x1, CX x2)
computes $\int_{x1}^{x2} l_1(x) \cdot l_2(x) dx$ where l_1 is the line crossing p1 and p2 and l_2 is the line crossing q1 and q2. It throws an error if one of the lines is vertical. (Def. 3.12)

A.2 Intervals

The Interval class manages and manipulates fuzzy temporal intervals (Sec. 3.2). The intervals are represented by their envelope polygons (Def. 3.15).

Constructors

Interval()
constructs an empty interval.

Interval(Point p)
constructs an interval with a single point p.

Interval(CX x, CY y)
constructs an interval with a single point (x, y).

Interval(CX a, CX b)
constructs a crisp interval [a, b[.

Interval(vector<Point> points)
constructs an interval with a vector of points.

Interval(string s)
constructs an interval from a string representation $[x_1, y_1 x_2, y_2 \dots[$.

Adding and Removing Points.

void I.push_back(Point p)
adds the point p to the end of the polygon. It throws an error if p.x is before the last point in the polygon. (Def. 3.18)

void I.push_back(CX x, CY y)
adds the point x, y to the end of the polygon. It throws an error if x is before the last point in the polygon. (Def. 3.18)

`void I.pop_back()`
removes the last point from the polygon. It does nothing on empty polygons. (Def. 3.18)

Simple Properties of the Intervals

`Point I.front()`
returns the leftmost point and throws an error if $I = ()$.

`Point I.back()`
returns the rightmost point and throws an error if $I = ()$.

`CX I.frontX()`
returns the leftmost x -coordinate and throws an error if $I = ()$.

`CX I.backX()`
returns the rightmost x -coordinate and throws an error if $I = ()$.

`CY I.frontY()`
returns the leftmost y -coordinate and throws an error if $I = ()$.

`CY I.backY()`
returns the rightmost y -coordinate and throws an error if $I = ()$.

`bool I.isNegInfinite()`
returns true if the interval is negative infinite.

`bool I.isPosInfinite()`
returns true if the interval is positive infinite.

`bool I.isInfinite()`
returns true if the interval is infinite.

`bool I.isEmpty()`
return true if the polygon is empty.

`bool I.isNonempty()`
returns true if the polygon is not empty.

`int I.nPoints()`
returns the number of points in the polygon.

`int I.isCrisp()`
returns true if the polygon is non-empty, finite and crisp.

`int I.isCrisp(bool front)`
checks for crispness at the left/right side of the polygon.

`int I.isSingleCrisp()`
returns true if the polygon is a non empty convex crisp interval.

`bool I.isConvex()`
returns true if the polygon is convex.

`bool I.isMonotone()`
returns true if the polygon is monotone. (Def. 3.54)

`bool I.isSymmetric()`
returns true if the polygon is symmetric. (Def. 3.54)

`CX I.SymmetryAxis2()`
returns twice the x -coordinate of the symmetry axis and throws an error if I is not symmetric.

`int I.index(CX x)`
returns the index of the rightmost polygon point that is left of x , or -1 if there is no such point. (Def. 3.20)

`int I.indexMax(bool front)`
if `front = true` it returns the index of the leftmost point with maximal y -value, otherwise it returns the index of the rightmost point with maximal y -value. If the polygon is empty it returns -1. (Def. 3.20)

`CY I.inf()`
returns the smallest y -value of the polygon. (Def. 3.27)

`CY I.sup()`
returns the hight `sup(I)` of the polygon. (Def. 3.27)

float I.member(CX x)
returns the membership value for the x -coordinate x . (Def. 3.22)

CX I.size2I(int k, int l)
returns $2 * \text{the area below the polygon from vertex } k \text{ to vertex } l$. (Def. 3.28)

CX I.size2()
returns $2 * \text{the area below the polygon}$. (Def. 3.28)

CX I.size2(CX a, CX b)
returns $2 * \text{the area below the polygon from } x\text{-coordinate } a \text{ to } x\text{-coordinate } b$. (Def. 3.28)

CX I.centrePoint(int k, int m)
returns the x -coordinates of the k, m -center point. (Def. 3.29)

int I.nComponents()
returns the number of components of the interval. (Def. 3.31)

Interval I.component(int k)
returns the k^{th} component of I. It throws an error if $k < 0$. (Def. 3.32)

Region is an enumeration type with values support, core, kernel, maximum.

bool I.nextComponent(CX& x1, CX& x2, int& i1, int& i2, Region region)
enumerates the components of the corresponding region (Def. 3.34)

CX I.size(Region r)
returns the size of the core/support/kernel/maxRegion. (Def. 3.35)

Interval I.crisp(Region r)
returns the core/support/kernel/maxRegion as crisp interval. (Def. 3.36)

CX I.side(Region r, bool front)
returns the x -coordinate of the left/rightmost point of the core/support/kernel/maxRegion. (Def. 3.37)

Hull Calculations

Interval I.crispHull()
returns the crisp hull of I. (Def. 3.38)

Interval I.monotoneHull()
returns the monotone hull of I. (Def. 3.39)

Interval I.convexHull()
returns the convex hull of I. (Def. 3.40)

Point–Interval Relations for Regions (Sec. 3.2.3)

bool I.before(CX t, Region region)
returns true if t is before the corresponding region of I.

bool I.starts(CX t, Region region)
returns true if t equals the left endpoint of the corresponding region of I.

bool I.during(CX t, Region region)
returns true if t is in the corresponding region of I.

bool I.finishes(CX t, Region region)
returns true if t equals the right endpoint of the corresponding region of I.

bool I.after(CX t, Region region)
returns true if t is after the corresponding region of I.

bool I.between(CX t, Region region)
returns true if t is in a gap between the corresponding region of I.

Relations Between Crisp Intervals and Regions of Fuzzy Intervals (Sec. 3.2.3)

bool I.before(CX t1, CX t2, Region region)
returns true if $[t1, t2[$ is before the corresponding region of I.

bool I.meets(CX t1, CX t2, Region region)
returns true if $[t1, t2[$ meets the corresponding region of I.

bool I.overlaps(CX t1, CX t2, Region region)
 returns true if [t1,t2[overlaps the corresponding region of I.

bool I.starts(CX t1, CX t2, Region region)
 returns true if [t1,t2[starts the corresponding region of I.

bool I.during(CX t1, CX t2, Region region)
 returns true if [t1,t2[is during the corresponding region of I.

bool I.finishes(CX t1, CX t2, Region region)
 returns true if [t1,t2[finishes the corresponding region of I.

bool I.after(CX t1, CX t2, Region region)
 returns true if [t1,t2[is after the corresponding region of I.

bool I.between(CX t1, CX t2, Region region)
 returns true if [t1,t2[is during a gap of the corresponding region of I.

bool I.disjoint(CX t1, CX t2, Region region)
 returns true if [t1,t2[is disjoint with the corresponding region of I.

CX I.partInside(CX t1, CX t2, Region region)
 returns the size of the part of [t1,t2[which is inside the corresponding region of I.

Relations Between the Regions of two Fuzzy Intervals (Sec. 3.2.3)

bool I.before(Interval J, Region region)
 returns true if the corresponding region of I is before the corresponding region of J.

bool I.meets(Interval J, Region region)
 returns true if the corresponding region of I meets the corresponding region of J.

bool I.overlaps(Interval J, Region region)
 returns true if the corresponding region of I overlaps with the corresponding region of J.

bool I.starts(Interval J, Region region)
 returns true if the corresponding region of I starts the corresponding region of J.

bool I.during(Interval J, Region region)
 returns true if the corresponding region of I is during the corresponding region of J.

bool I.finishes(Interval J, Region region)
 returns true if the corresponding region of I finishes the corresponding region of J.

bool I.equals(Interval J, Region region)
 returns true if the corresponding region of I equals the corresponding region of J.

bool I.disjoint(Interval J, Region region)
 returns true if the corresponding region of I is disjoint with the corresponding region of J.

CX I.partInside(Interval J, Region region)
 yields the size of the part of the corresponding region of I which is inside the corresponding region of J.

Basic Unary Transformations (Def. 2.31)

Interval I.extend(true)
 returns the rising part of I. (Def. 3.41)

Interval I.extend(false)
 returns the falling part of I. (Def. 3.41)

Interval I.scaleUp()
 scales the y -values of the interval up to T . (Def. 3.41)

Interval I.scaleUpD()
 is the destructive version of ScaleUp.

Interval I.shift(CX a)
 shifts the interval by a units. (Def. 3.41)

Interval I.shiftD(CX a)
 is the destructive version of Shift.

Interval I.cut(CX x1, CX x2)
 cuts the part of the interval between $x1$ and $x2$. (Def. 3.42)

| | | |
|---|---|-------------|
| <code>Interval I.cutI(int i1, int i2)</code> | cuts the part of the interval between the points with index <code>i1</code> and <code>i2</code> . | (Def. 3.42) |
| <code>Interval I.times(float a)</code> | multiplies the y -values of the interval by <code>a</code> . | (Def. 3.43) |
| <code>Interval I.exponentiate(float e)</code> | exponentiates the y -values of the interval with <code>e</code> . | (Def. 3.47) |
| <code>Interval I.integrate(true)</code> | computes $J(x) \stackrel{\text{def}}{=} \int_{-\infty}^x I(y)dy/ I $. <code>I</code> may be infinite. | (Def. 3.45) |
| <code>Interval I.integrate(false)</code> | computes $J(x) \stackrel{\text{def}}{=} \int_x^{+\infty} I(y)dy/ I $. <code>I</code> may be infinite. | (Def. 3.45) |
| <code>Interval I.negate()</code> | inverts the y -values. | (Def. 2.31) |
| <code>Interval I.invert()</code> | inverts the y -values of the gaps in the interval. | (Def. 2.31) |
| <code>CY I.integrateAsymmetric(Interval J)</code> | computes $\int I(x) \cdot J(x) dx/ I $. <code>I</code> and <code>J</code> may be infinite. | (Def. 3.51) |
| <code>CY I.integrateSymmetric(Interval J, bool simple)</code> | computes $\int I(x) \cdot J(x) dx/N(I, J)$. It throws an error if both <code>I</code> and <code>J</code> are infinite. | (Def. 3.53) |

Fuzzification

| | | |
|--|---|-------------|
| <code>Interval I.fuzzifyLinear(bool front, CX x1, CX x2, CX offset)</code> | linear fuzzification of the front/end part of the interval with absolute coordinates. | (Def. 2.34) |
| <code>Interval I.fuzzifyLinear(bool front, float percent, float offset)</code> | linear fuzzification of the front/end part of the interval with relative coordinates. | (Def. 2.37) |
| <code>Interval I.fuzzifyLinear(float percent, float offset)</code> | linear fuzzification of both sides of the interval with relative coordinates. | |
| <code>Interval I.fuzzifyGaussian(bool front, CX xh, CX x0, CX offset)</code> | Gaussian fuzzification of the front/end part of the interval with absolute coordinates. | (Def. 2.35) |
| <code>Interval I.fuzzifyGaussian(bool front, float percent, float offset)</code> | Gaussian fuzzification of the front/end part of the interval with relative coordinates. | (Def. 2.37) |
| <code>Interval I.fuzzifyGaussian(float percent, float offset)</code> | Gaussian fuzzification of both sides of the interval with relative coordinates. | |

General Transformations

| | | |
|---|--|-------------|
| <code>Interval I.unaryTransformation(UnaryYFunction f)</code> | applies the unary y -function <code>f</code> to <code>I</code> . | (Def. 3.46) |
| <code>Interval I.binaryTransformation(Interval J, BinaryYFunction f)</code> | applies the binary y -function <code>f</code> to <code>I</code> and <code>J</code> . | (Def. 3.50) |

A.3 Y-Functions

The unary and binary transformation methods (Def. 3.46, 3.50) expect a function f which is to be applied to one or two y -coordinates. Some of these functions, however, depend on extra parameters. For example the λ -complement (Def. 2.19) $n_\lambda(y) \stackrel{\text{def}}{=} \frac{1-y}{1+\lambda y}$ depends on the parameter λ . This would not be a problem in most functional programming languages. One can define $n(\lambda, x)$ and then get n_λ through currying. The solution in object oriented languages is a bit different. One defines a class “lambdaComplement” with instance variable “lambda”. The class can be instantiated with a corresponding value for lambda. This instance can now be used like any other data object in the language. The trick which allows one to use the instance like a function depends on the programming language. In C++ one can define a `()` operator for this class, which realizes the function application. If the instance is bound to the variable f , and x is another variable then $f(x)$ is now a legal expression and yields the function value. In Java one would define an apply-method and write $f.apply(x)$. The class-approach has many advantages: the parameters can be changed at any time, which is not so easy for curried functions; a class hierarchy can structure the functions according to their semantics, and not their types; further methods can be defined which do other kinds of computations and return meta-information, for example whether the function is linear.

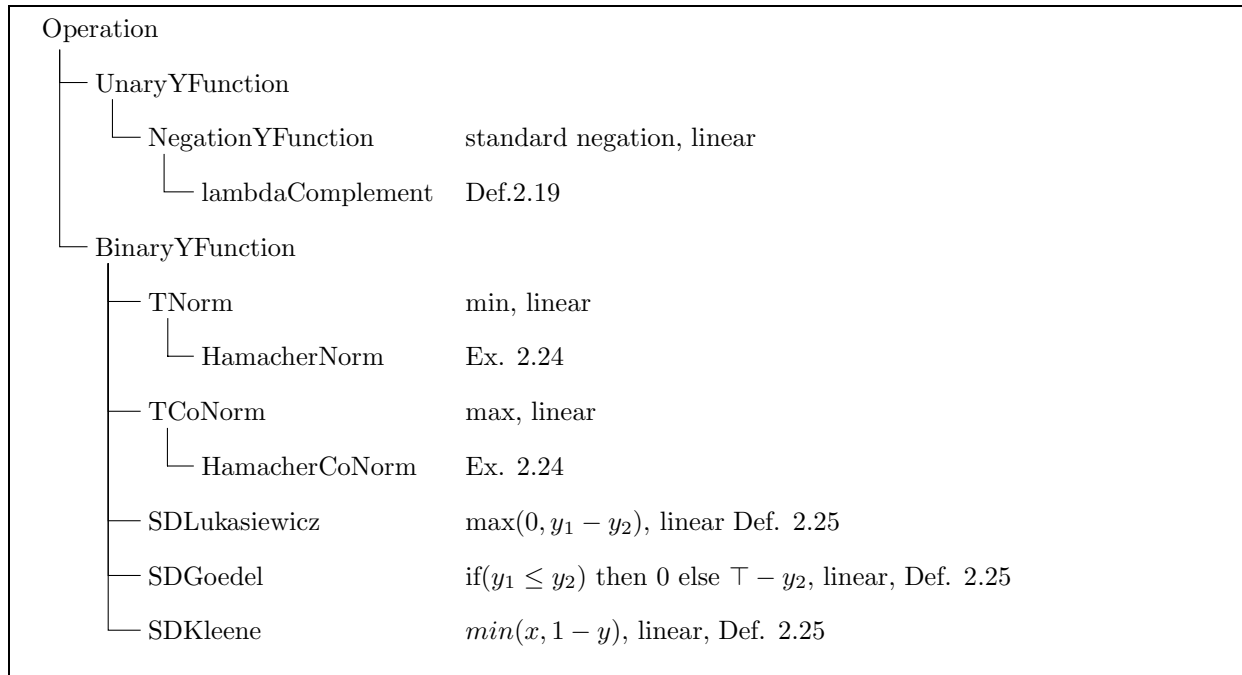


Figure 2: Class Hierarchy for Y-Functions

FuTI realizes y-functions with the class hierarchy in Fig. 2.

The top class, ‘Operation’, manages the mapping of function names to the functions (instances of the other classes). Each instance can get a name, for example ‘myFavoriteLambdaComplement’, and one can retrieve the corresponding instance with the method `Operation::getOperation(string name)`. The name is optional. Instances without names are not accessible via `getOperation`.

Constructors

`NegationYFunction(string name)`
 constructs the standard negation function $\lambda(y)(1 - y)$. (Def. 2.19)

`lambdaComplement(float lambda, string name)`
 constructs the lambda complement $\lambda(y) \frac{1-y}{1+\lambda y}$. (Def. 2.19)

`TNorm(string name)`
 Constructs the min t-norm.

`HamacherNorm(float gamma, string name)`
 Constructs the Hamacher t-norm $\lambda(x, y) \frac{xy}{\gamma + (1-\gamma)(x+y-xy)}$. (Def. 2.24)

`TCoNorm(string name)`
 Constructs the max t-conorm

`HamacherCoNorm(float beta, string name)`
 Constructs the Hamacher t-conorm $\lambda(x, y) \frac{x+y+(\beta-1)xy}{1+\beta xy}$. (Def. 2.24)

Parameter Modification

The parameters `lambda`, `gamma`, `beta` can be changed with `setParameter` and read with `getParameter`.

Acknowledgements

This research has been funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REVERSE number 506779 (cf. <http://reverse.net>).