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

Ludwig
Maximilians–
Universität
München
LMU

Fuzzy Time Intervals and Relations

The FuTIRe Library

Hans Jürgen Ohlbach

Technical Report, Institute for Computer Science, Munich, Germany
http://www.pms.informatik.uni-muenchen.de/publikationen
Forschungsbericht/Research Report PMS-FB-2004-4, 2 2004

Fuzzy Time Intervals and Relations – The FuTIRe Library

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

March 9, 2004

Abstract

The FuTIRe library is a collection of classes and methods for representing and manipulating fuzzy time intervals and relations between them. Time intervals like 'tonight', which are usually not very precise, can be modeled as fuzzy sets. But this causes the problem that the relations between points and intervals and between two intervals, which are usually very trivial, become very complex when the intervals are fuzzy sets. Moreover, there are many different possibilities to define such relations. In FuTIRe it is not only possible to represent fuzzy time intervals, but one can define customized fuzzy point-interval and interval-interval relations. These relations can even yield fuzzy values when the intervals are in fact crisp. As an example for an application, consider a database with, say, a cinema timetable, and you query the timetable "give me all performances ending before midnight". The usual 'before' relation will exclude the performances ending a second after midnight. With the fuzzy before relation in FuTIRe you can get instead of a sharp drop to 0 at midnight decreasing fuzzy values after midnight, and these can be used to order the results of the query. FuTIRe is an open source C++ library.

Contents

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 I describe the FuTIRe-system (Fuzzy Temporal Intervals and Relations), a library of datastructures and algorithms for representing and manipulating fuzzy temporal notions.

In FuTIRe one can

- represent finite and infinite fuzzy time intervals;
- combine and manipulate these intervals in various ways;
- define customized point-interval relations like 'before', 'starts', 'during' etc. between time points and crisp as well as fuzzy intervals in a fuzzy way;
- define customized interval-interval relations similar to Allen's interval relations, but between fuzzy intervals, and with fuzzy values as result.

In the first part of the paper I describe the components of the FuTIRe-system in a purely mathematical way, without any commitment to concrete datastructures and algorithms. In the second part I present a representation of the fuzzy intervals as polygons with integer coordinates. All algorithms in FuTIRe work on these polygons. Finally the concrete interface to the library is listed and explained. The library is a component of the WebCal-system [2], a program for evaluating temporal expressions like 'three weeks after Easter'. WebCal is currently under development.

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

2 The Mathematics of Fuzzy Time Intervals and Relations

The mathematics of general fuzzy sets [8] has been investigated in great depth. The particular fuzzy sets in FuTIRe, 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. [3]).

Since FuTIRe 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 organize 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.

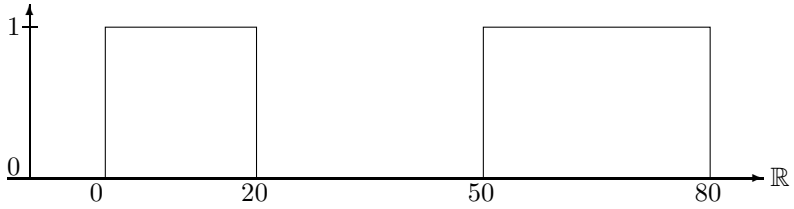
The base set for fuzzy time intervals is the time axis, in FuTIRe represented by the set $\mathbb{R}$ of real numbers. Real numbers allow us to model the continuous time flow which we perceive in our life. A fuzzy time interval in FuTIRe is now a fuzzy subset of the real numbers.

Definition 2.2 (Fuzzy Time Intervals) *A* fuzzy membership function *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*

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

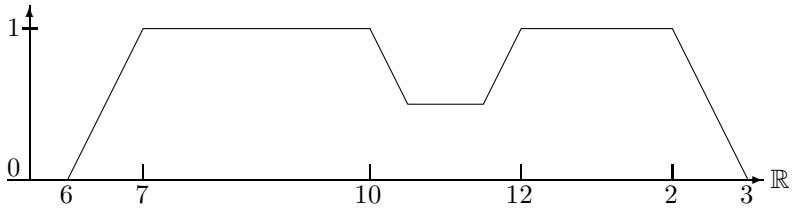
Given a fuzzy interval i *we usually write* $i(x)$ *to indicate the corresponding membership function.*
Let $F_{\mathbb{R}}$ *be the set of fuzzy time intervals.* ■

This definition comprises single or multiple crisp intervals like this:



Crisp Fuzzy Intervals [0,20],[50,80]

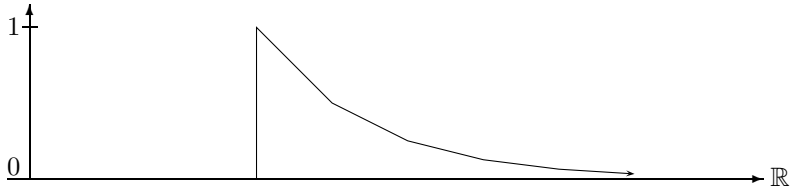
It also comprises finite fuzzy intervals like this one:



Party Time

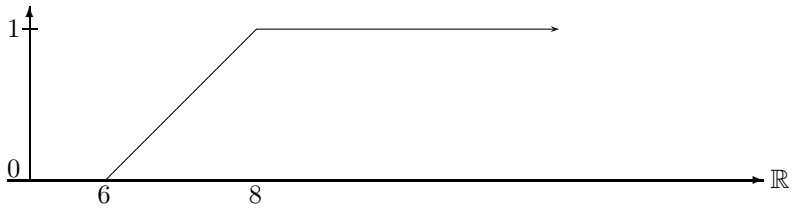
This set may represent a particular party time, where the first guests arrive at 6 pm. At 7 pm all guests are there. Half of them disappear between 10 and 12 pm (because they go to the pub next door to watch an important soccer game). Between 12 pm and 2 am all of them are back. At 2 am the first ones 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.



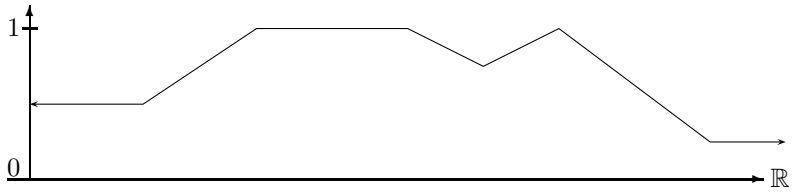
Dangour Time: Caused by Radioactive Decay

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 value which rises from fuzzy value 0 at 6 pm until fuzzy value 1 at 8 pm and then remains 1 ad infinitum.



After Tonight

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* $\cap$ *(union),* $\cup$ *(intersection) and* $\setminus$ *(set difference) to fuzzy intervals.*

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

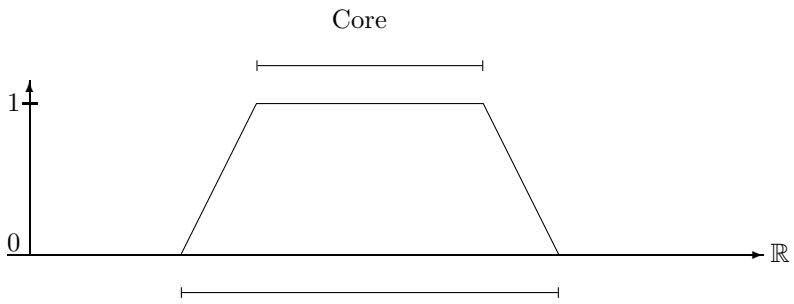
Definition 2.4 (Core, Support and Kernel)
The core $C(i)$ *of a fuzzy set* i *is the subset of* $\mathbb{R}$ *where the membership function is 1:*

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

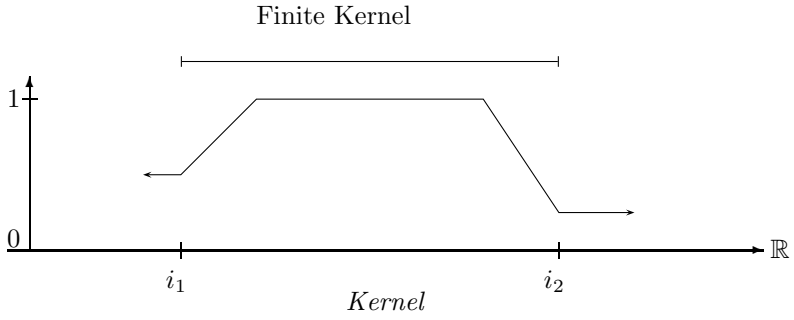
The core of i *can be empty even if* i *itself is not empty.*
The support $S(i)$ *of* i *is the subset of* $\mathbb{R}$ *where the membership function is nonzero:*

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

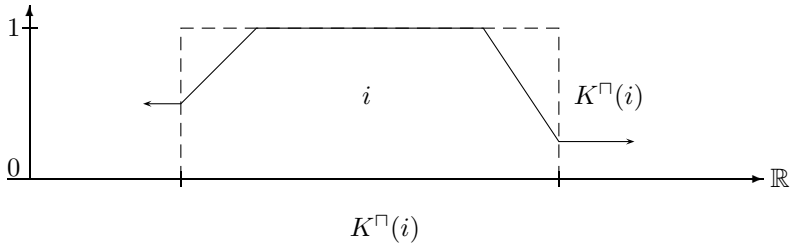
If $S(i) = \emptyset$ *then* $i = \emptyset$.
The kernel $K(i)$ *of* i *is the smallest interval* $[a,b] \subseteq \mathbb{R}^+$ *such that there are* $i_1 \in [0,1]$ *and* $i_2 \in [0,1]$ *with* $i(x) = i_1$ *for all* $x < a$ *and* $i(x) = i_2$ *for all* $x > b$.
$K(i)$ *can be empty, finite or infinite. If* $K(i) = \emptyset$ *then* i *is either empty or infinite and crisp.*
For $O \in \{C, S, K\}$ *let* $O^{\sqcap}(i)$ *be the (crisp) fuzzy interval such that* $C(O^{\sqcap}(i)) = S(O^{\sqcap}(i)) = K(O^{\sqcap}(i)) = O^{\sqcap}(i)$. ∎



Core and Support



The next picture shows the kernel of the same interval i as crisp set $K^{\sqcap}(i)$.



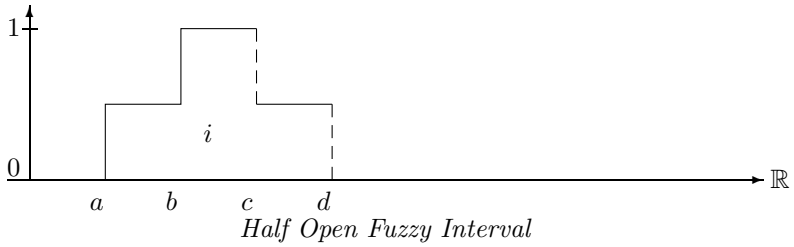
Fuzzy time intervals with finite kernel are of particular interest because although they may be infinite, they can easily be implemented with finite datastructures. Therefore we give them an extra name.

Definition 2.5 (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. 2.4).* ■

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

Definition 2.6 (Crisp Interval) *A* crisp interval *is a fuzzy interval i (Def. 2.2) such that $C(i) = S(i)$ (Def. 2.4).* ■

Remark 2.7 (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 2.8 (Size) *For $a, b \subseteq \mathbb{R}^+$ and a fuzzy time interval i let*

$$|i|_a^b \stackrel{\text{def}}{=} \int_a^b i(x)\ dx. \qquad |i| \stackrel{\text{def}}{=} |i|_{-\infty}^{+\infty}$$

is the size *of i.* ■

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

Definition 2.9 (First and Last Points) *For an operator $O \in \{C, S, K\}$ and a fuzzy time interval i with $O(i) \neq \emptyset$ let*

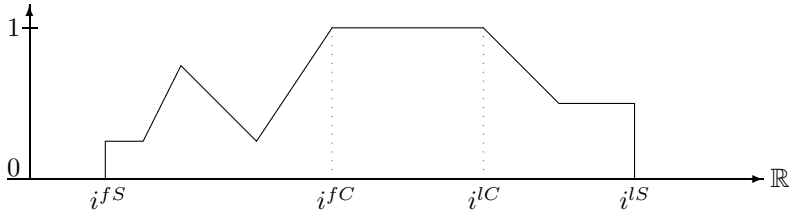
$$i^{fO} \stackrel{\text{def}}{=} \inf(O(i))$$

be the first O-point *of i and let*

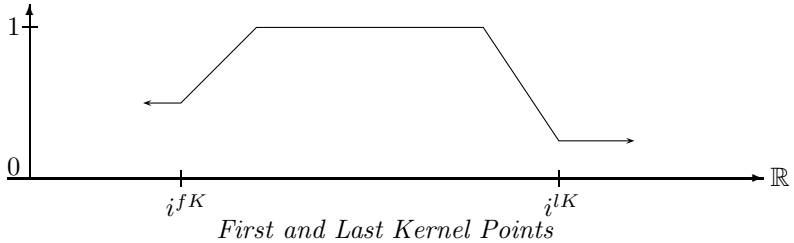
$$i^{lO} \stackrel{\text{def}}{=} \sup(O(i))$$

be the last O-point *of i.* ■

i^{fC} is the first core point, i^{fS} is the first support point, and i^{fK} is the first kernel point. i^{lC} is the last core point, i^{lS} is the last support point, and i^{lK} is the last kernel point.



First and Last Core and Support Points



First and Last Kernel Points

Definition 2.10 (Height of a Fuzzy Interval) *For a fuzzy interval $i \in F_{\mathbb{R}}$ let*

$$i\hat{} \stackrel{\text{def}}{=} \sup_x \{i(x)\}$$

be the height *of i.* ■

$i\hat{}$ is usually, but not necessarily, 1 for nonempty fuzzy time intervals. If $i\hat{} = 0$ then, however, i must be empty.

Definition 2.11 (First and Last Maximal Point) *If $i \in F_{\mathbb{R}}$ let*

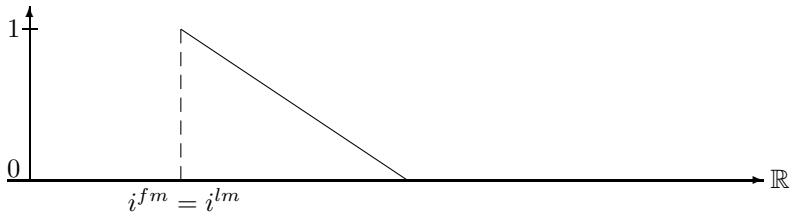
$$i^{fm} \stackrel{\text{def}}{=} \begin{cases} \sup\{x \mid \forall y < x : i(y) < i\hat{}\} & \textit{if the supremum exists} \\ -\infty & \textit{otherwise} \end{cases}$$

be the first maximum*, and*

$$i^{lm} \stackrel{\text{def}}{=} \begin{cases} \inf\{x \mid \forall y > x : i(y) < i\hat{}\} & \textit{if the infimum exists} \\ +\infty & \textit{otherwise} \end{cases}$$

be the last maximum*.* ■

The next picture shows an example where $i^{fm} = i^{lm}$ and where $i\hat{}$ is really the supremum, and not the maximum because $i(i^{fm}) = 0$.



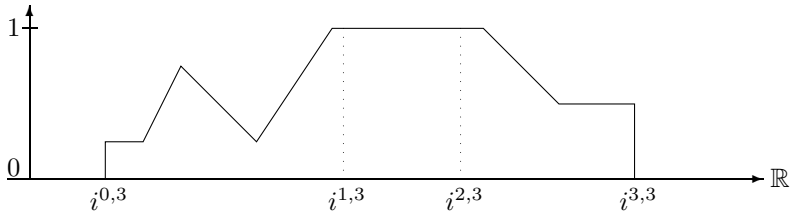
First and Last Maximal Points

If $i\hat{} = 1$ then $i^{fm} = i^{fC}$ and $i^{lm} = i^{lC}$ (Def. 2.9). If, however, $i\hat{} < 1$ then i^{fm} and i^{lm} have nothing to do with the core of i.

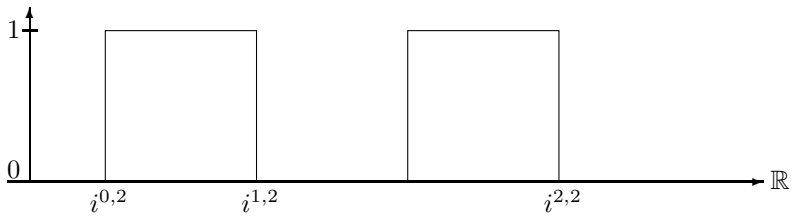
Center Points
The n, m-*center points* defined below are used to express temporal notions like 'the first half of the year', or 'the second quarter of the year' or more exotic expressions like 'the 25th 49th of the weekend' etc. The notion of n, m-center points makes only sense for finite intervals.

Definition 2.12 (n, m-Center 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-center points
$i^{n,m} \stackrel{\text{def}}{=} x_n$ *where x_n is a minimal $\mathbb{R}$-value in a sequence $i^{fS} = x_0, \ldots, x_m = i^{lS}$ with $|i|_{x_0}^{x_1} = |i|_{x_1}^{x_2} = \ldots = \ldots |i|_{x_{m-1}}^{x_m} = |i|/m$.* ∎

Examples: $i^{1,2}$ splits i in two halfs 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.



$n, 3$-*Center Points*



$n, 2$-*Center Points*

Middle Points:
The middle point between the center 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 $i^{3,4}$.

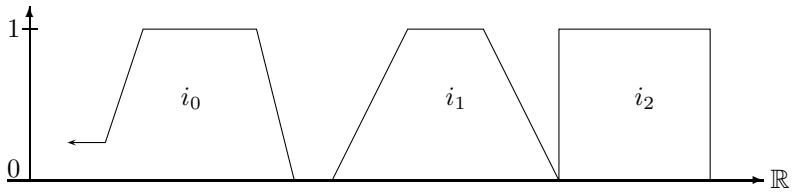
Components
Fuzzy time intervals can consist of several different components. We define a split of a time interval i into its components $i_0, \ldots, i_n$.

Definition 2.13 (Components) *Let $i \in F_{\mathbb{R}}$. The* components $i_0, \ldots, i_n$ *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 $k \in \{1, \ldots, n-1\}$:* $(\lim_{x \to i_k^{fs}} i(x) = 0$ *or* $\lim_{i_k^{fs} \leftarrow x} i(x) = 0)$ *and* $(\lim_{x \to i_k^{ls}} i(x) = 0$ *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:



Components

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 2.14 (Y-Functions)
Y-FCT$^n \stackrel{\text{def}}{=} \{f : [0,1]^n \mapsto [0,1]\}$ *is the set of n-place* y-functions.
They map fuzzy values to fuzzy values.
Y-FCT$\stackrel{\text{def}}{=} \bigcup_{n\geq 0}$ *Y-FCT*^{n}. ■

Definition 2.15 (Interval Operators)
I-OPs$^n \stackrel{\text{def}}{=} \{g : {F_{\mathbb{R}}}^n \mapsto F_{\mathbb{R}}\}$ *is the set of n-place* interval operators.
They map fuzzy intervals to fuzzy intervals.
I-OPs$\stackrel{\text{def}}{=} \bigcup_{n\geq 0}$ *I-OPs*^{n}. ■

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 2.16 (Associated Interval Operators) *If $f \in$ Y-FCT^n is a y-function then $g_f \in$ I-OPs^n defined by $g_f(i_1,\ldots,i_n)(x) \stackrel{\text{def}}{=} f(i_1(x)),\ldots,i_n(x))$ 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 of the polygons need to be transformed.

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

Definition 2.17 (Linear Y-Function) *A y-function $f \in$ Y-FCT^n is linear if the mapping $f'((x,y_1),\ldots,(x,y_n)) \stackrel{\text{def}}{=} (x, f(y_1,\ldots,y_n))$ maps non-intersecting line segments (x_1,z_{11}) – (x_2,z_{12}), …, (x_1,z_{n1}) – (x_2,z_{n2}) to a line segment $(x_1, f(z_{11},\ldots,z_{n1}))$ – $(x_2, f(z_{12},\ldots,z_{n2}))$.* ■

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

Proposition 2.18 (Characterization 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

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

The other direction of the proof is trivial. ■

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

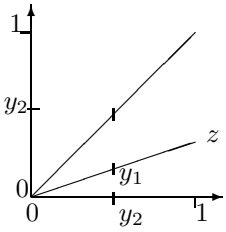
The characterization of two-place linear y-functions is a bit trickier.

Proposition 2.19 (Characterization of Two-Place Linear y-Functions) *A two-place y-function f is linear if and only if the following condition holds:*

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

Proof: Suppose f is linear.

We consider the case $y_1 \le y_2$ 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 $y_1 \le y_2$. 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 $z=y_1/y_2$.

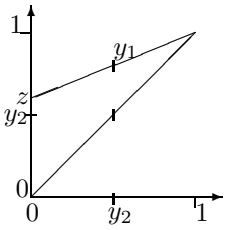


Since f is linear we have

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

Now consider the case $y_1 \ge y_2$.

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

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

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 function. 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 on 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 2.20 (Negation Function) *A function $n \in$ Y-FCT[1] satisfying the conditions*

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

is called a negation function*.*

Let NF be the set of all negation functions*.* ∎

Example 2.21 (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. 2.20.

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

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

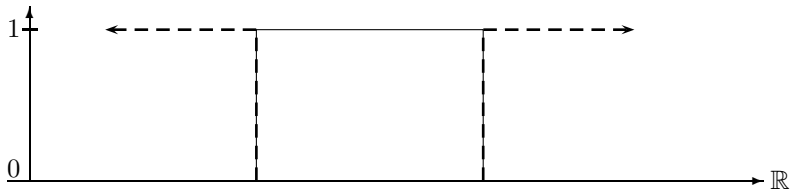
If i is a crisp interval then $N(i)=N_\lambda(i)$ ∎

Proposition 2.22 (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: $n_\lambda(n_\lambda(y)) = y$.*

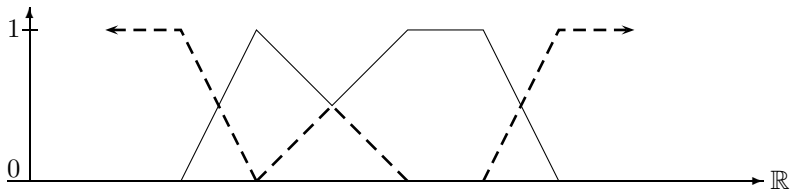
The proof is straightforward. ∎

This property need not hold for other negation functions.

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

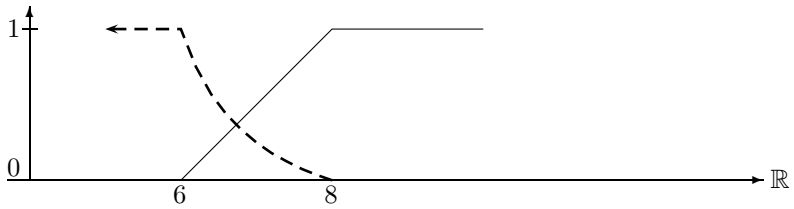


Standard Complement and λ-Complement for a Crisp Interval



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:

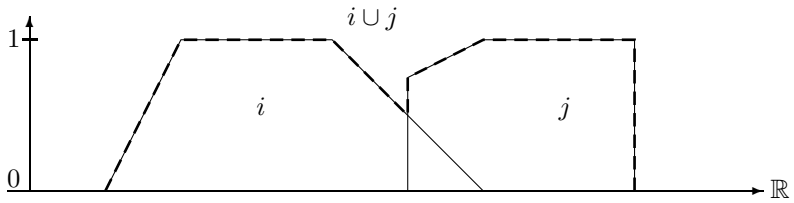


λ-Complement for $\lambda = 2$

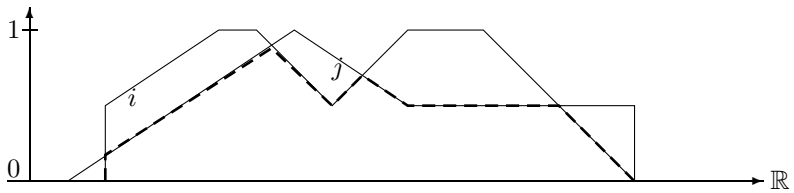
If λ is increased then the descend from 6 pm till 8 pm becomes steeper. A suitable λ could then in fact mean 'long before tonight'.

Intersection and Union of Fuzzy Time Intervals

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



Standard Union of Fuzzy Sets



Standard Intersection of Fuzzy Sets

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 conorms, must obey certain axioms in order to satisfy our intuition about union and intersection.

Definition 2.23 (Triangular Norms and Conorms) *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 conorm *or* t-conorm *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*:* $\forall x,y,u,v \in [0,1]\ x \leq u, y \leq v :$
T4: $T(x,y) \leq T(u,v),$
S4: $S(x,y) \leq S(u,v)$

Triangular norms and conorms are y-functions in Y-FCT^2 *(Def. 2.14).*

Let $TNorm$ *be the set of triangular norms and*
let $TCoNorm$ *be the set of triangular conorms.* ■

The triangular norms and conorms are now turned into interval operators $\cap_T$ and $\cup_S$:

Definition 2.24 (Intersection and Union) *Let* $i,j \in F_{\mathbb{R}}$ *be two fuzzy intervals. If* T *is a triangular norm and* S *a triangular conorm (Def. 2.23) then*

$$(i \cap_T j)(x) \stackrel{\text{def}}{=} T(i(x), j(x))$$
$$(i \cup_S j)(x) \stackrel{\text{def}}{=} S(i(x), j(x))$$

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

Example 2.25 (Standard Fuzzy Intersection and Union) *The function* min *is a triangular norm and the function* max *is a triangular conorm. Therefore*

$$(i \cap_{\min} j)(x) \stackrel{\text{def}}{=} \min(i(x), j(x))$$
$$(i \cup_{\max} j)(x) \stackrel{\text{def}}{=} \max(i(x), j(x))$$

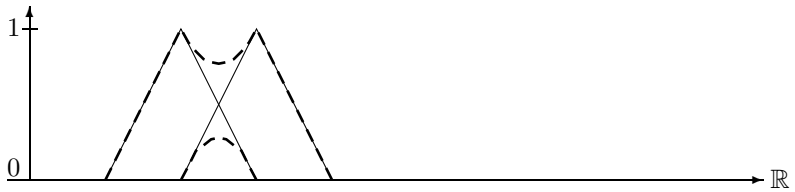
are the standard fuzzy intersection and union operators. ■

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

Example 2.26 (Hamacher Family) *The Hamacher family consists of the following parameterized families of triangular norms and conorms, and negation functions (λ-complement):*

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

■



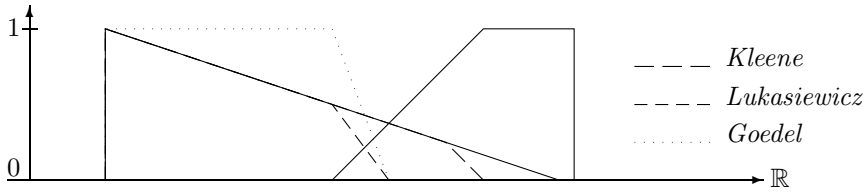
Hamacher Intersection and Union with $\beta = \gamma = 0.5$

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 2.27 (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 2.28 (Set Difference) *The following picture shows the difference between the three versions of set difference.*



Set Difference

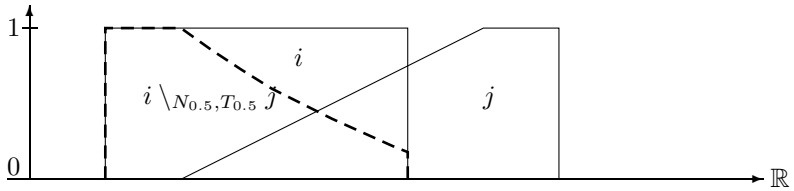
■

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 generalized by replacing $\cap$ with $\cap_T$ and j^c with a complement operator.

Definition 2.29 (Generalized 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*

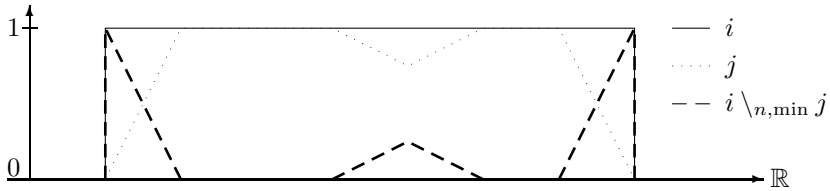
$$(i \setminus_{N,T} j) \stackrel{\text{def}}{=} i \cap_T N(j)$$

■



Fuzzy Set Difference $i \setminus_{N_{0.5},T_{0.5}} j$

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.

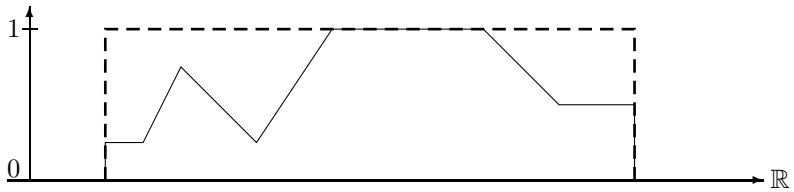


Set Difference Splits into Three Polygons

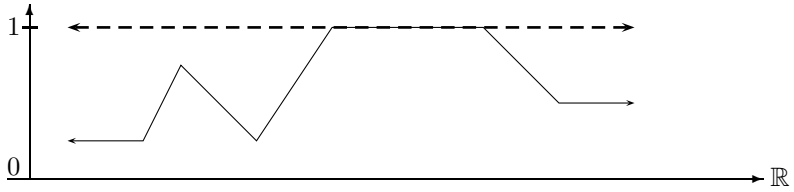
2.5 Hull Operators on Fuzzy Intervals

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* looses most information about the interval, whereas the last notion, the *monotone hull* looses the least information. All these notions of a hull coincide for crisp intervals.

Definition 2.30 (Crisp Hull) *For an interval $i \in F_{\mathbb{R}}$ let $CrH(i)$ be the smallest crisp interval containing i.* ■

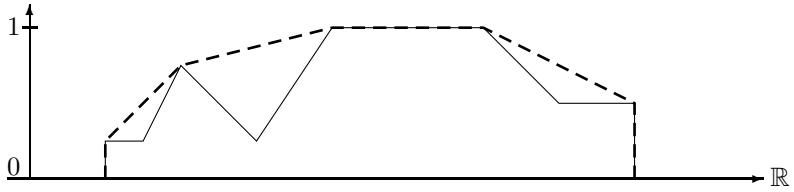


Crisp Hull of a Finite Interval

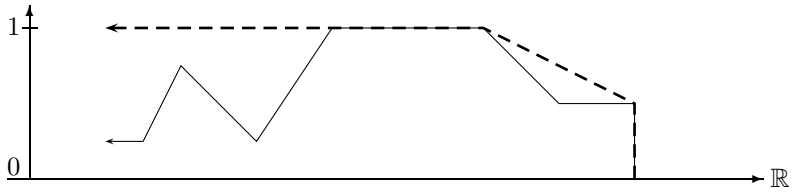


Crisp Hull of an Infinite Interval

Definition 2.31 (Convex Hull) *The* convex hull $CoH(i)$ *of a fuzzy set i is the smallest convex set containing i.* ■



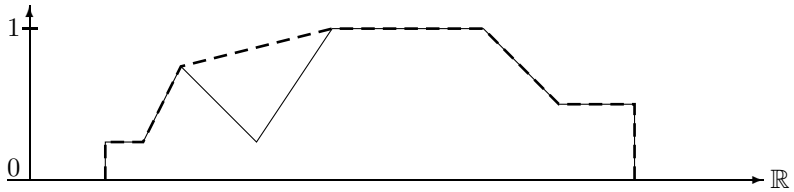
Convex Hull of a Finite Set



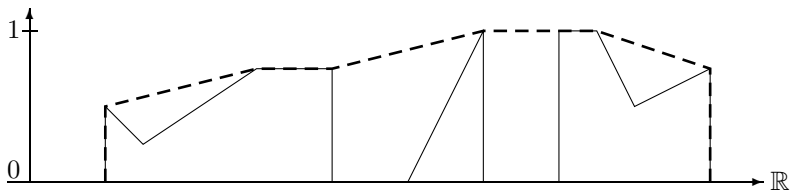
Convex Hull of an Infinite Set

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

Definition 2.32 (Monotone Hull) *The* monotone hull $MoH(i)$ *of a fuzzy set i is the smallest* monotone *fuzzy interval containing i.* Monotone *means that from left to right the fuzzy values $MoH(i)(x)$ are rising monotonically to $i\hat{}$, and then falling monotonically again.* ■



Monotone Hull



Monotone Hull of a Fuzzy Interval with Three Components

2.6 Basic Unary Transformations

The main purpose of FuTIRe is to provide fuzzy point-interval and interval-interval relations. As we shall see, these relations can also be parameterized, this time with interval operators. Therefore we first introduce a little library of interval operators, which can be used to build the point-interval and interval-interval relations.

Definition 2.33 (Basic Unary Transformations) *Let $i \in F_{\mathbb{R}}$ be a fuzzy interval. We define the following (parameterized) interval operators:*

$$identity(i) \stackrel{\text{def}}{=} i$$

$$extend^{+}(i)(x) \stackrel{\text{def}}{=} \begin{cases} i(x) & \text{if } x \leq i^{fm} \\ 1 & \text{otherwise} \end{cases}$$

$$extend^{-}(i)(x) \stackrel{\text{def}}{=} \begin{cases} i(x) & \text{if } x \geq i^{lm} \\ 1 & \text{otherwise} \end{cases}$$

$$scaleup(i)(x) \stackrel{\text{def}}{=} \begin{cases} i(x)/i\hat{} & \text{if } i\hat{} \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$cut_{x_1,x_2}(i)(x) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } x < x_1 \text{ or } x \geq x_2 \\ i(x) & \text{otherwise} \end{cases}$$

$$cut_{x_1,+}(i)(x) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } x < x_1 \\ i(x) & \text{otherwise} \end{cases}$$

$$cut_{x_1,-}(i)(x) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } x \geq x_1 \\ i(x) & \text{otherwise} \end{cases}$$

$$shift_n(i)(x) \stackrel{\text{def}}{=} i(x-n)$$

$$times_a(i)(x) \stackrel{\text{def}}{=} \min(1, a \cdot i(x)) \qquad a \geq 0$$

$$exp_e(i)(x) \stackrel{\text{def}}{=} i(x)^e \qquad e \geq 0$$

$$integrate^{+}(i)(x) \stackrel{\text{def}}{=} \lim_{a \mapsto \infty} \frac{\int_{-a}^{x} i(y)dy}{\int_{-a}^{+a} i(y)dy}$$

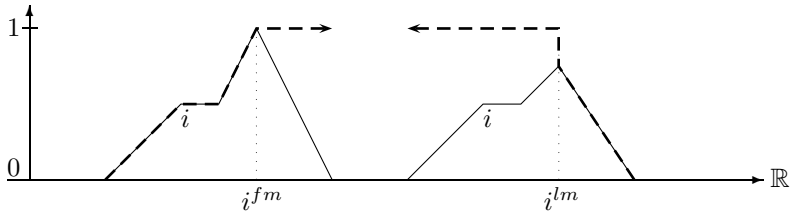
$$integrate^{-}(i)(x) \stackrel{\text{def}}{=} \lim_{a \mapsto \infty} \frac{\int_{x}^{+a} i(y)dy}{\int_{-a}^{+a} i(y)dy}$$

$$invert(i)(x) \stackrel{\text{def}}{=} \begin{cases} 1 - i(x) & \text{if } i_k^{fm} \leq x < i_{k+1}^{lm} \\ & \text{where } i_0, \ldots i_m \text{ are the components of } i \\ 0 & \text{otherwise.} \end{cases}$$

■

extend

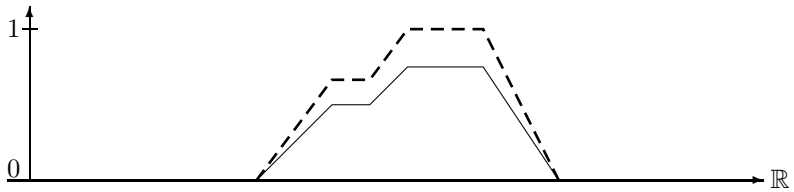
$extend^{+}(i)$ follows the left part of the interval until the left maximum i^{lm} is reached and then stays at fuzzy value 1. $extend^{-}(i)$ is the symmetric version of $extend^{+}(i)$.



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

scaleup

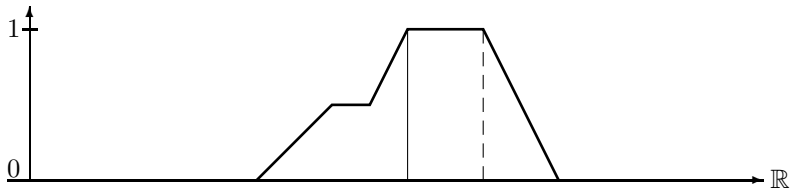
The *scaleup*-function is different to the *identity* function only if the hight $i\hat{}$ is not 1. In this case it scales the membership function up such that $scaleup(i)\hat{} = 1$.



scaleup

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.

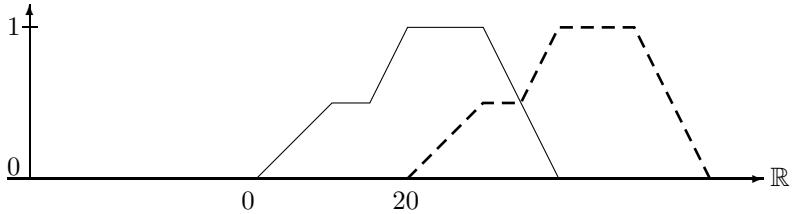


cut_{x_1,x_2}

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

shift

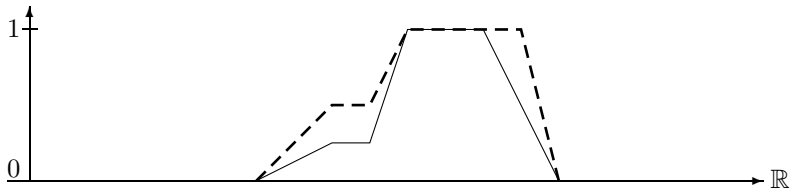
$shift_n$ just moves the interval by n time units.



$shift_{20}$

times

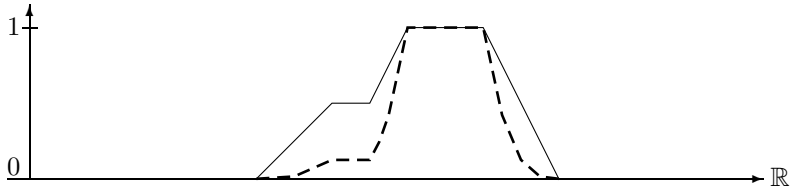
$times_a$ multiplies the membership function by a, but keeps the result smaller or equal 1. $times_a$ has no effect on crisp intervals.



$times_2$

exp

exp_e takes the membership function to the exponent e. It can be used to damp increases or decreases. exp_e has also no effect on crisp intervals.



exp_3

integrate

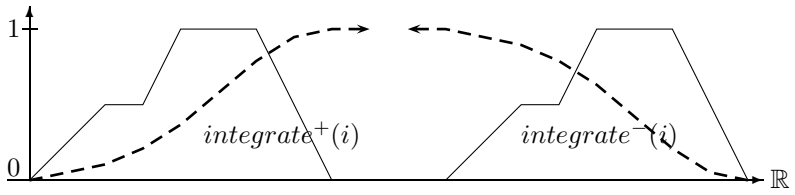
This operator integrates over the membership function and normalizes the integral to values $\leq$ 1. The two integration operators $integrate^+$ and $integrate^-$ can be simplified for finite fuzzy time intervals.

Proposition 2.34 (Integration for Finite Intervals) *If the fuzzy interval i is finite then*

$$integrate^+(i)(x) = \frac{\int_{-\infty}^{x} i(y)dy}{|i|} \qquad and \qquad integrate^-(i)(x) = \frac{\int_{x}^{+\infty} i(y)dy}{|i|}$$

The proofs are straightforward. ∎

Example for $integrate^+$ and $integrate^-$:



$integrate^+$ *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 2.35 (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*

$$integrate^+(i)(x) = \frac{i_1}{i_1 + i_2} \qquad and \qquad integrate^-(i)(x) = \frac{i_2}{i_1 + i_2}$$

Proof:

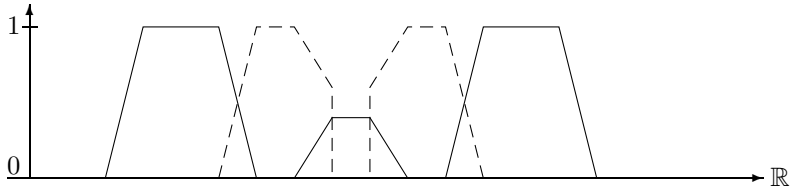
$$\begin{array}{lcl}
integrate^+(i)(x) & = & \lim_{a\mapsto\infty} \frac{\int_{-a}^{x} i(y)dy}{\int_{-a}^{+a} i(y)dy} \\
& = & \lim_{a\mapsto\infty} \frac{|i|_{-a}^{i^{fK}} + |i|_{i^{fK}}^{x}}{|i|_{-a}^{i^{fK}} + |i|_{i^{fK}}^{i^{lK}} + |i|_{i^{lK}}^{a}} \\
& = & \lim_{a\mapsto\infty} \frac{|i|_{-a}^{i^{fK}}}{|i|_{-a}^{i^{fK}} + |i|_{i^{lK}}^{a}} \\
& = & \lim_{a\mapsto\infty} \frac{(i^{fK}+a)\cdot i_1}{(i^{fK}+a)\cdot i_1 + (a-i^{lK})\cdot i_2} \\
& = & \lim_{a\mapsto\infty} \frac{a\cdot i_1}{a\cdot i_1 + a\cdot i_2} \\
& = & \frac{i_1}{i_1+i_2}
\end{array}$$

$$\begin{array}{lcl}
integrate^-(i)(x) & = & \lim_{a\mapsto\infty} \frac{\int_{x}^{+a} i(y)dy}{\int_{-a}^{+a} i(y)dy} \\
& = & \lim_{a\mapsto\infty} \frac{|i|_{x}^{i^{lK}} + |i|_{i^{lK}}^{a}}{|i|_{-a}^{i^{fK}} + |i|_{i^{fK}}^{i^{lK}} + |i|_{i^{lK}}^{a}} \\
& = & \lim_{a\mapsto\infty} \frac{|i|_{i^{lK}}^{a}}{|i|_{-a}^{i^{fK}} + |i|_{i^{lK}}^{a}} \\
& = & \lim_{a\mapsto\infty} \frac{(a-i^{lK})\cdot i_2}{(i^{fK}+a)\cdot i_1 + (a-i^{lK})\cdot i_2} \\
& = & \lim_{a\mapsto\infty} \frac{a\cdot i_2}{a\cdot i_1 + a\cdot i_2} \\
& = & \frac{i_2}{i_1+i_2}
\end{array}$$

∎

invert

The *invert* function is almost like the standard negation function, except that $invert(i)$ 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 $invert(i)$ drops down to 0 between the first and last maximum of the middle component. *invert* is needed for the *in_the_gap*-operator.



$invert(i)$

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 FuTIRe 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 2.36 (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:*

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

If the offset is nonzero we have

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

The 'back' linear fuzzification function is:

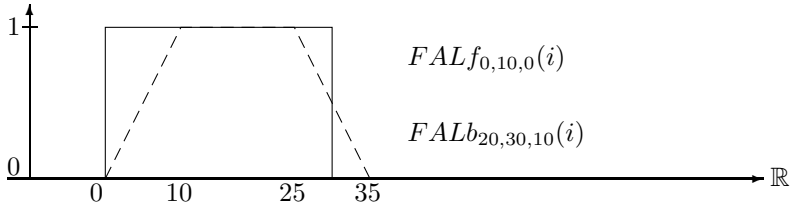
$$FALb_{x_1,x_2,0} \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } x \geq x_2 \\ i(x) & \text{if } x < x_1 i(x)\frac{x_2-x}{x_2-x_1} \quad \text{otherwise} \end{cases}$$

If the offset is nonzero we have

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

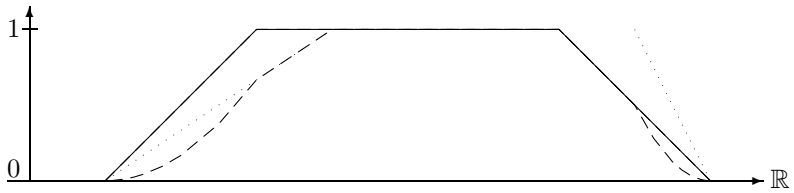
■

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.



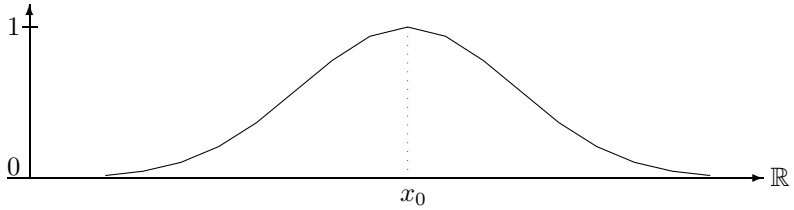
Linear Fuzzification:

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.



Linear Fuzzification of an Already Fuzzy Interval

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



Gaussian Shape

The Gaussian fuzzification function is determined by the parameters x_0 and x_h. x_h is the x-coordinate where $e^{-((x_h-x_0)/\sigma)^2} = 0.5$. 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 2.37 (Gaussian Fuzzification Function) *Let $i \in F_{\mathbb{R}}$, x_h, x_0 and $offset$ be x-coordinates. We define the 'front' Gaussian fuzzification function with zero offset first:*

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

If the offset is nonzero we have

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

The 'back' Gaussian fuzzification function is:

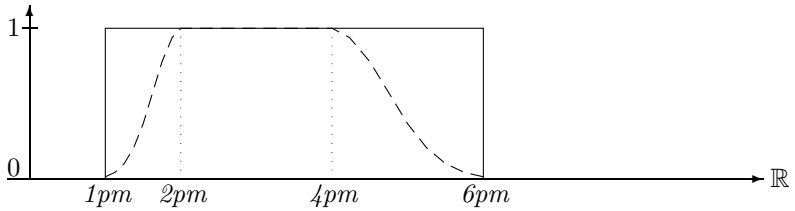
$$FAGb_{x_1,x_2,0} \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } x > 3x_h - 2x_0 \\ i(x) & \text{if } x < x_0 i(x)e^{-((x-x_0)/\sigma)^2} \quad \text{otherwise} \end{cases}$$

If the offset is nonzero we have

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

■

Example 2.38 *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

■

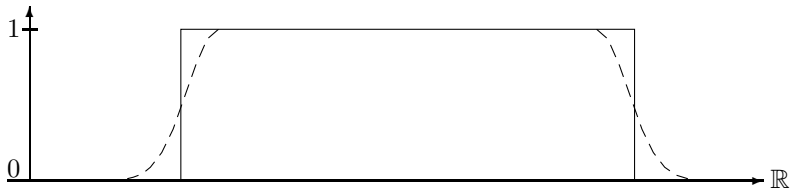
Fuzzification functions with absolute coordinates are not that useful because usually one does not know the coordinates in advance. Therefore FuTIRe also provides fuzzification functions where the parameters are percentage values. $FRLf_{10,5}$ for example means linear fuzzification where the linear increase is 10% of the kernel size and the offset is 5% of the kernel size (cf. Ex. 2.50). $FRLf_{10}$ 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 i_0 and the offset is such that x_h coincides with i^{fK}.

Definition 2.39 (Fuzzification with Relative Coordinates) *For an interval i, percentage numbers r and o between 0 and 100 we define the relative fuzzification functions.*

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

$$\begin{array}{lcl} FRLf_{r,o}(i) & \stackrel{\text{def}}{=} & FALf_{i^{fK},i^{fK}+d\cdot r,i^{fK}-d\cdot o}(i) \\ FRLb_{r,o}(i) & \stackrel{\text{def}}{=} & FALf_{i^{lK}-d\cdot r,i^{lK},i^{lK}+d\cdot o}(i) \\ FRGf_{r}(i) & \stackrel{\text{def}}{=} & FAGf_{i^{fK}+d\cdot r,i^{fK}+1/2\cdot d\cdot r,2/3\cdot d\cdot r}(i) \\ FRGb_{r}(i) & \stackrel{\text{def}}{=} & FAGf_{i^{lK}-d\cdot r,i^{lK}-1/2\cdot d\cdot r,2/3\cdot d\cdot r}(i) \end{array}$$

■



Relative Gaussian Fuzzification $FRGf_{20} \circ FRGb_{20}$

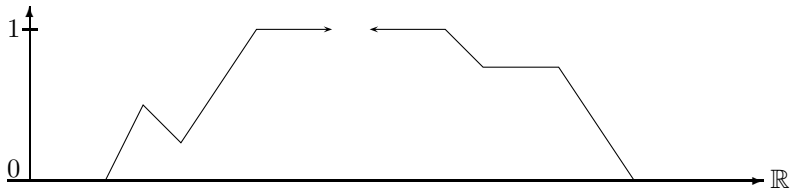
Classification of Interval Operators

Interval operators which turn fuzzy time intervals into infinite intervals which rise until 1 and then stay constant are of particular interest for the definition of certain point-interval relations (Sect. 2.7 below). Therefore we define rising (and falling) fuzzy time intervals and interval operators.

Definition 2.40 (Rising and Falling Fuzzy Intervals and Interval Operators) *A Fuzzy set i is* rising *iff for its membership function $i(x) = 1$ for all $x > i^{fm}$. i is* falling *iff for its membership function $i(x) = 1$ for all $x < i^{lm}$.*

An interval operator f is rising iff $f(i_1, \ldots, i_n)$ is rising *for all tuples $i_1, \ldots, i_n$ of intervals. f is falling iff $f(i_1, \ldots, i_n)$ is* falling *for all tuples $i_1, \ldots, i_n$ of intervals.* ∎

A rising fuzzy interval need not really be monotonically rising until $i\hat{}$. The membership function can move up and down until it reaches 1. Only after this point it must not fall again.



Rising and Falling Intervals

Proposition 2.41 *The basic unary transformations $extend^+$ and int^+ are rising interval operators and the unary transformations $extend^-$ and int^- are falling interval operators.*

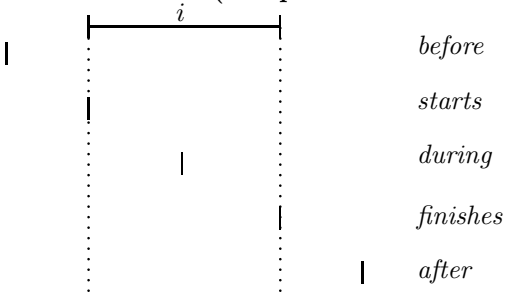
Any composition $f_1 \circ \ldots \circ f_n \circ f$ where f is a rising (falling) interval operator is again a rising (falling) interval operator.

The proofs are straightforward. ∎

2.7 Point-Interval Relations

There are five basic relations between a time point and a *crisp* interval i:

Definition 2.42 (Crisp Point-Interval Relations)



∎

If the intervals possess a metric, which is the case for time intervals over the real numbers, there are infinitely many more point-interval relations. Examples are 'during the first half' or 'in the middle of the third quarter'. For lists of intervals there are even more point-interval relations, for example 'between the intervals' or 'between the second and third interval' etc.

If we want to 'fuzzify' such relations we face two problems:
1. How should a relation like 'before' work for a fuzzy time interval? For example, what does 'before early afternoon' mean, where 'early afternoon' is represented by a fuzzy set?
2. How can we fuzzify a point-interval relation even for crisp intervals. Can we for example obtain a non-zero fuzzy value for the expression 'the party starts *after* midnight' even if the party starts, say, one minute before midnight?

There are no general solutions to these problems. Therefore FuTIRe provides a framework where application specific versions of such relations can be defined and used. Point-interval relations are in this framework represented by (parameterized) interval operators. This may be a bit surprising because relations are in general not functions. The relations are therefore represented indirectly as functions. Consider a point p and an interval i. As the result of the relation p *before* i we want a fuzzy value. To get this fuzzy value, we can turn i into another interval *before*(i) such that *before*$(i)(x)$ is the desired fuzzy value.

Since *before*$(\ldots)$ can operate on any fuzzy time interval we have also solved the problem to get a *before* relation between points and *fuzzy* time intervals.

Unfortunately there is no unique definition for *before*(i). One way is to parameterize *before* and all the other relations such that application specific relations can be defined very easily. The parameters are again interval operators. Other schemes are also known [6].

The general definitions are presented first, and then explained in detail.

Definition 2.43 (Point-Interval Relations as Unary Transformations)
Let $i \in F_{\mathbb{R}}$ be a fuzzy interval, and let T be a triangular norm (Def. 2.23). n, m, k are non-negative integers.

We define the following point-interval relation operators:

$$\text{before}_{N,E^+}(i) \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } i = \emptyset \\ N(E^+(i)) & \text{otherwise} \end{cases}$$
where N is a complement-operator and E^+ a rising operator (Def. 2.40).

$$\text{after}_{N,E^-}(i) \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } i = \emptyset \\ N(E^-(i)) & \text{otherwise} \end{cases}$$
where N is a complement-operator and E^- a falling operator (Def. 2.40).

$$\text{starts}_{E^+,B,T}(i) \stackrel{\text{def}}{=} scaleup(E^+(i) \cap_T B(i))$$
where E^+ is a rising operator and B a before*-operator*

$$\text{finishes}_{E^-,A,T}(i) \stackrel{\text{def}}{=} scaleup(E^-(i) \cap_T A(i))$$
where E^- is a falling operator and A an after*-operator*

$$\text{during}_{U,O}(i) \stackrel{\text{def}}{=} U_{l \in Cmp(i)} O(l)$$
where U is a union operator and O any unary transformation

$$\text{during}_{D,n,m}(i) \stackrel{\text{def}}{=} D(cut_{i^{n,m}, i^{n+1,m}}(i))$$
where D is a during*-operator*

$$\text{in\_the\_middle}_{D,k,n,m}(i) \stackrel{\text{def}}{=} D(cut_{i^{2^k(2n+1)-1,2^k 2m}, i^{2^k(2n+1)+1,2^k 2m}}(i))$$
where D is a during*-operator*

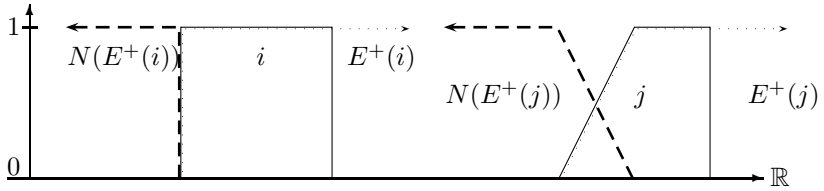
$$\text{in\_the\_gap}_D(i) \stackrel{\text{def}}{=} D(invert(i))$$
where D is a during*-operator*

$$\text{in\_the\_}k^{th}\text{gap}_{D,k}(i) \stackrel{\text{def}}{=} D(Component(invert(i), k))$$
where D is a during*-operator*

For a particular point x one can get the fuzzy value of the relation xri by applying the corresponding relation operator O_r to i and then calculating $O_r(i)(x)$. ∎

Before and **After**
The definition of *before* is $before_{n,E^+}(i) \stackrel{\text{def}}{=} N(E^+(i))$, where E^+ is a rising interval operator (Def. 2.40) and N is a negation interval operator. E^+ projects the rising part of i out and hides all the rest of i. N complements the rising part.

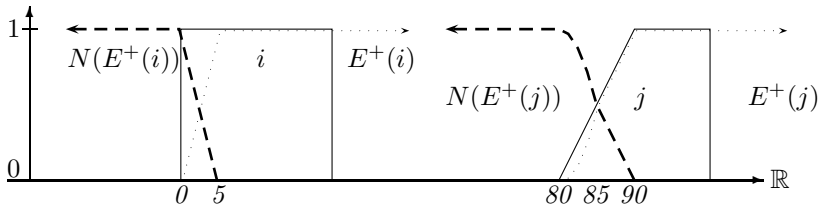
Example 2.44 (*before* with Standard Negation and $E^+ = extend^+$) *The first example gives us the standard* before*-relation. For crisp sets it is the usual before relation. For fuzzy sets it is essentially complements the front part of the interval.*



before *with Standard Negation and $extend^+$*

∎

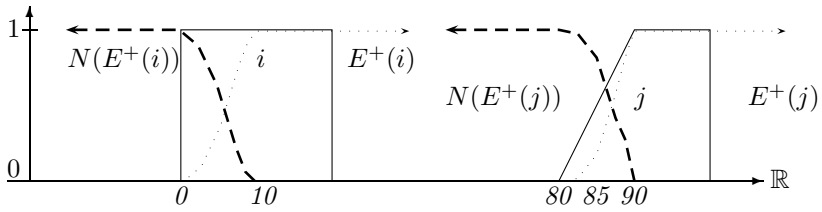
Example 2.45 (More Fuzzy *before*) *In this example we want to make the* before*-relation a bit more fuzzy, such that points after the start of the interval i get a non-zero fuzzy value. To this end we distort the interval i with linear fuzzification functions $F = FALf_{0,5,0}$ (left graph) and $F = FALf_{80,85,0}$ (right graph). $E^+ = F \circ extend^+$ and N is standard complement.*



before *with Linear Fuzzification*

The fuzzification function has not much of an effect on the second interval because the linear increase ends already in the middle of the ascend from 0 at 80 to 1 at 90. ■

Example 2.46 (Gaussian *before*) *This example is similar to the previous one, but instead of a linear fuzzification function we use Gaussian fuzzification functions $F = FAGf_{5,10,0}$ and $F = FAGf_{85,90,0}$ to fuzzify the before relation. $E^+ = F \circ extend^+$ and N is the standard complement.*



before *with Gaussian Fuzzification Function*

■

The *after*-relation is just the symmetric variant of the *before*-relation. Therefore no further explanation is necessary.

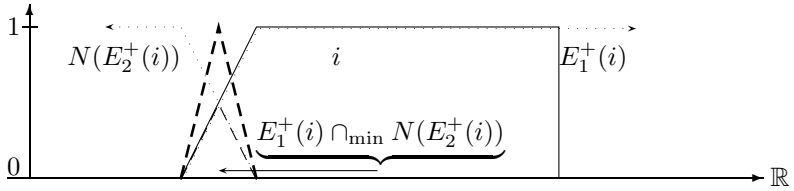
Starts and Finishes:
The *finishes* relation is the symmetric variant of the *starts*-relation. Therefore it is sufficient to discuss the *starts*-relation.

The standard *starts*-relation x *starts* i for crisp intervals i is 1 if x is the starting point of i, and 0 everywhere else. This corresponds to an almost empty fuzzy set, with a single peak right at the start of i. The basic idea of a fuzzy *starts*-relation is to widen this single peak a little bit. For crisp intervals there are some minor technical problems here. Therefore we first explain it with a fuzzy interval.

The definition is $starts_{E_1^+,E_2^+,T}(i) \stackrel{\text{def}}{=} scaleup(E_1^+(i) \cap_T N(E_2^+(i)))$.

The rising function E_1^+ extracts the front part of i. $N(E_2^+(i))$ extracts the same or another front part of i and complements it with the complement function N. The first extracted front part and the complemented extracted front part are then intersected. Since the intersection may yield a fuzzy set with maximum fuzzy value < 1 it is scaled up to 1.

Example 2.47 (Simple *starts*-Relation) *In the first example we take $E_1^+ = E_2^+ = extend^+$, the standard complement for N, and* min *for the t-norm T.*

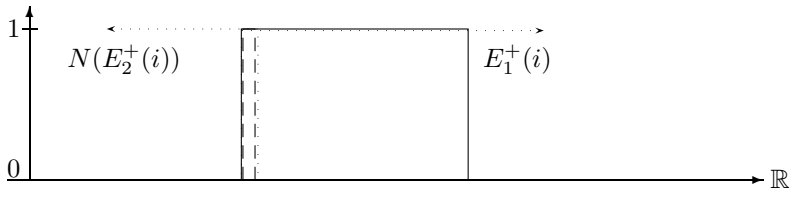


starts

The dashed line indicates the scaled up intersection. ■

It is now easy to imagine that the steeper the ascend of the front part of I is, the narrower the peak of the *starts*-relation will be. Unfortunately, in the extreme case where i is a crisp interval, there is no singular *starts*-peak anymore, but the result is just the empty set. If $E_1^+ = E_2^+ = extend^+$ then $E_1^+(i)$ and $N(E_2^+(i))$ are complementary sets, and their intersection is empty. To overcome this problem, FuTIRejust shifts $E_1^+(i)$ by a small amount.

Example 2.48 (*starts*-Relation for a Crisp Interval) *The dashed line indicates the* starts*-peak.*

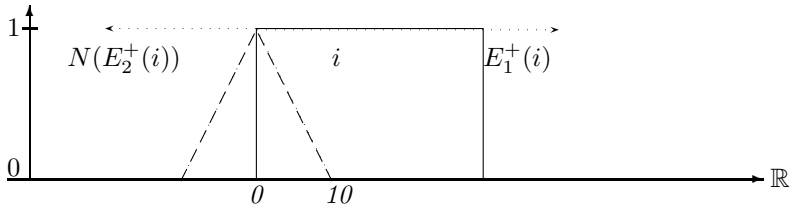


starts *for a Crisp Interval*

■

We can now fuzzify the *starts*-relation for crisp intervals by widening the *starts*-peak again.

Example 2.49 (*starts*-relation for a Crisp Interval) *We choose* $E_1^+ \stackrel{\text{def}}{=} extend^+ \circ FALf_{0,10,-10}$ *and* $E_2^+ \stackrel{\text{def}}{=} extend^+ \circ FALf_{0,10,0}$.

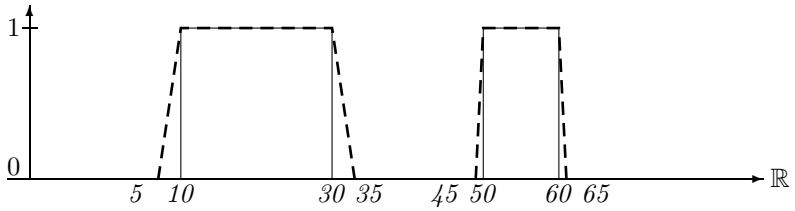


fuzzy starts *for a Crisp Interval*

■

During:
The simplest version of the *during* operator is just the identity: x *during* $i = identity(i)(x) = i(x)$. This returns 0 for all x outside i and the fuzzy membership value for all x inside i. In particular for crisp intervals this is a very strict interpretation of 'during'. We can weaken this strict interpretation by fuzzifying the interval i and taking the membership function of the fuzzified interval. If i consists of one single component, we get then $during(i) \stackrel{\text{def}}{=} O(i)$ where O fuzzifies i in some way. If i consists of several components, we must fuzzify each component separately and then take the union of all fuzzified components. The final definition is then $during_O(i) \stackrel{\text{def}}{=} \bigcup_{l \in Cmp(i)} O(l)$.

Example 2.50 (*during* for Crisp Intervals) *We choose* $O = FRLf_{10,10} \circ FRLb_{10,10}$

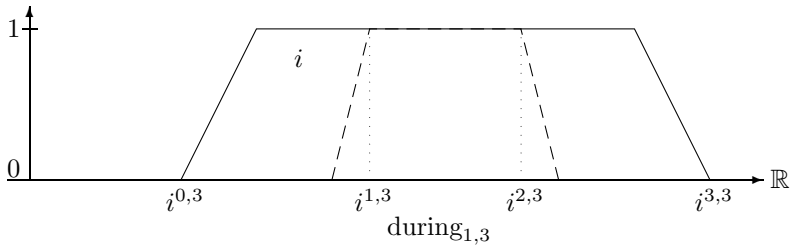


during *for a Crisp Interval with two Components*

■

During$_{n,m}$:
The relation 'during the first half' or 'during the second third' or in general 'during the n^{th} m^{th}' is very similar to the basic *during*-relation. The only difference is that the corresponding part – the first half or the second third etc. – has to be cut out of the fuzzy interval i before the basic *during*-operator is applied. The definition is therefore just $during_{O,n,m}(i) \stackrel{\text{def}}{=} during_O(cut_{i^{n,m},i^{n+1,m}})$.

Example 2.51 (*during*$_{1,3}$) *We choose again* $O = FRLf_{10,10} \circ FRLb_{10,10}$.



$\text{during}_{1,3}$

The dashed line indicates $\text{during}_{1,3}(i)$.

■

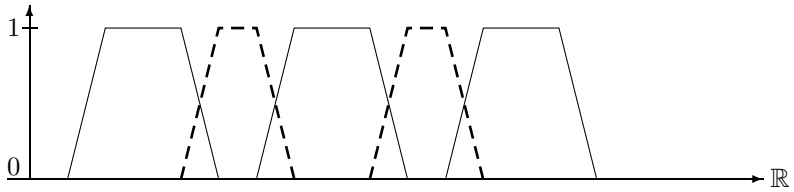
In the middle:
This relation focuses on the middle point of the interval between $i^{n,m}$ and $i^{n+1,m}$. The middle point is $i^{2n+1,2m}$. The idea is to cut a slice around this middle point out of i and apply a *during*-operator to this slice. The size of the slice is determined by the parameter k. The larger k the smaller the slice. This yields the definition

$$in\_the\_middle_{D,k,n,m}(i) \stackrel{\text{def}}{=} D(cut_{i^{2^k(2n+1)-1,2^k 2m}, i^{2^k(2n+1)+1,2^k 2m}}(i))$$

In the gap:
The *in_the_gap*-operator is sensitive to the gaps between different components of the fuzzy interval i. For example if i is the time of a soccer game, then *in_the_gap* detects the break between the two halfs of the game. The operator is quite simple: the first step is to invert the interval i in order to make the gaps visible. Then we apply a *during*-operator to the inverted i.

$$in\_the\_gap_D(i) \stackrel{\text{def}}{=} D(invert(i))$$

Example 2.52 (*in_the_gap*) *with* $D = identity$



in_the_gap_{id}

■

The *in_the_gap*-operator does not distinguish between the different gaps in the fuzzy interval i. Therefore we introduced the operator $in\_the\_k^{th}gap_{D,k}$ with the extra parameter k to select the particular gap. $in\_the\_k^{th}gap_{D,0}$ chooses the first gap, $in\_the\_k^{th}gap_{D,1}$ chooses the second gap etc.

The definition is $in\_the\_k^{th}gap_{D,k}(i) \stackrel{\text{def}}{=} D(Component(invert(i), k))$.

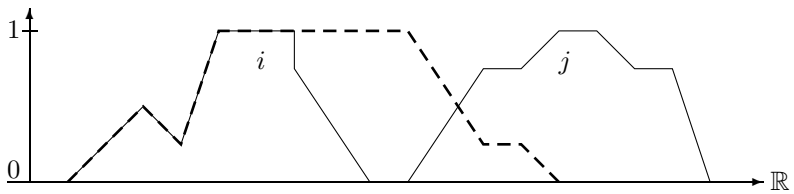
Until

The 'Until' operator is known from temporal logics [4] φ *Until* ψ in a temporal logic usually means 'eventually ψ holds and φ holds *until* this time point. FuTIRe also provides an 'Until' operator, but φ and ψ are not formulae but fuzzy time intervals. With FuTIRe's Until operator we can model expressions like 'from early morning until late night', where 'early morning' and 'late night' are concrete fuzzy intervals. An expression like this is ambiguous. It can be interpreted as 'from the beginning of early morning until the end of late night' or 'from the beginning of early morning until the beginning of late night', and there are two more possibilities. FuTIRe provides all four combinations.

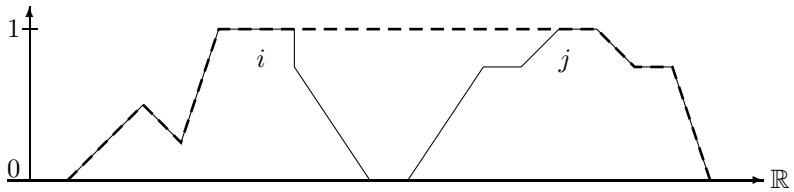
Definition 2.53 (Until) *Let E^+ be a rising function (Def. 2.40), E^- a falling function, N a complement operator, and T a t-norm (Def. 2.23). For two fuzzy sets i and j we define*

$$\begin{aligned}
Until^{bb}_{E^+,T,N}(i,j) &\stackrel{\text{def}}{=} E^+(i) \cap_T N(E^+(j)) \\
Until^{be}_{E^+,E^-,T}(i,j) &\stackrel{\text{def}}{=} E^+(i) \cap_T E^-(j) \\
Until^{eb}_{E^+,E^-,T,N}(i,j) &\stackrel{\text{def}}{=} N(E^-(i)) \cap_T N(E^+(j)) \\
Until^{ee}_{E^-,T,N}(i,j) &\stackrel{\text{def}}{=} N(E^-(i)) \cap_T E^-(j).
\end{aligned}$$

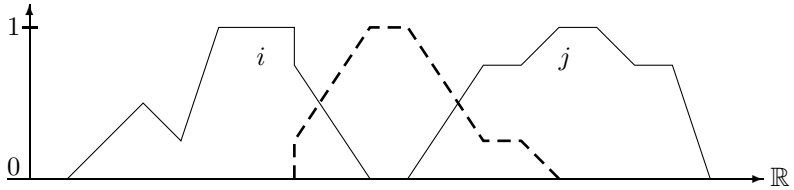
The next four figures show the four cases for the *Until* operator when two single non-overlapping fuzzy intervals are involved. We choose $E^+ = extend^+$, $E^- = extend^-$, N is the standard complement and $T = \min$.



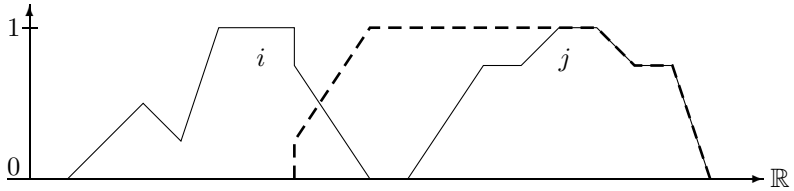
$Until^{bb}_{E^+,T,N}(i,j)$



$Until^{be}_{E^+,E^-,T}(i,j)$

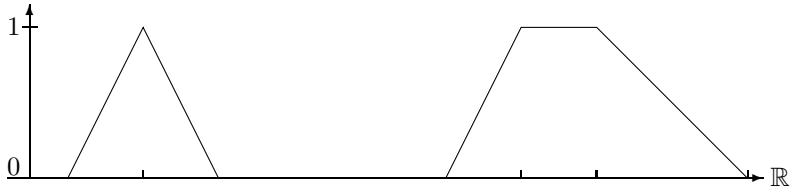


$Until^{eb}_{E^+,E^-,T,N}(i,j)$



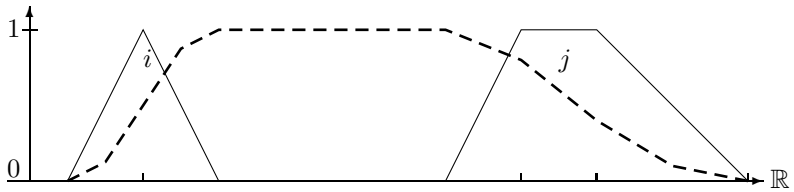
$Until^{ee}_{E^-,T,N}(i,j)$

Example 2.54 (Birthday Party Time) *The Until operator can be used in more sophisticated ways. Consider a database about, say, the institute's birthday parties. It may contain the entry that the birthday party for the director took place 'from around noon until early evening' of 20/7/2003. 'Around noon' is a fuzzy notion and 'early evening' is a fuzzy notion. Suppose, we have a formalization of 'around noon' and 'early evening' as the following fuzzy sets:*



Around Noon and Early Evening

What is now the duration of the birthday party? It must obviously also be a fuzzy set. The fuzzy value of the birthday party duration at a time point x is 1 if the probability that the party started before x is 1 and the probability that the party ended after x is also 1. Therefore the fuzzy value at point x is computed by integrating over the probabilities of the start points and the end points. Therefore we choose $E^+ = integrate^+$ and $E^- = integrate^-$ (Def. 2.33). The resulting fuzzy set is:



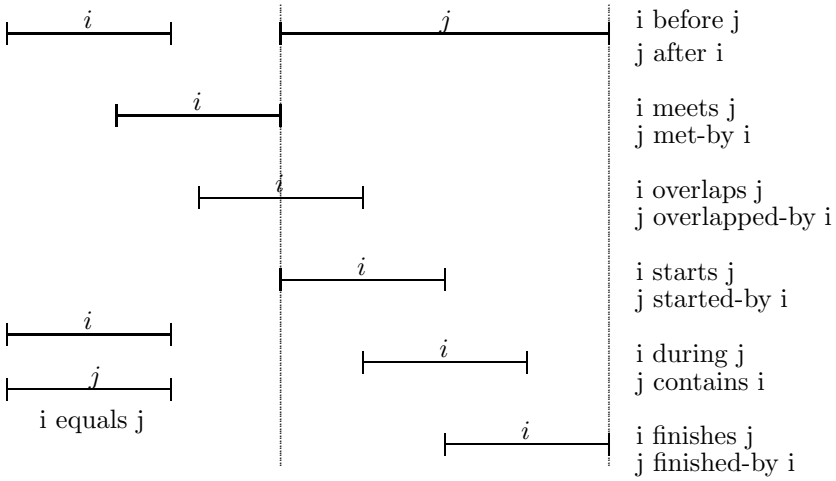
Birthday Party Time: $Until^{ee}_{integrate^+,integrate^-,N}(i,j)$

The dashed curve may for example represent the percentage of people at the party at a give time. ■

2.8 Interval-Interval Relations

Allen's seven interval relations [1] are the basic relations between two crisp intervals.

Definition 2.55 (Allen's Interval-Interval Relations)



■

We want to generalize these relations in two ways:

1. even for two crisp intervals we want a fuzzy value as result. The result should of course be 1 if the classical relation yields true, but it should not necessarily jump to 0 when the classical relation yields 0;
2. the relations should work for fuzzy time intervals regardless if they consist of one or more components or if they are finite or infinite.

The basic idea for the generalized relations is very simple: since we have the point-interval relations, we can extend the point to an interval in the relation by integrating over the interval's membership function. Take for example the point-interval relation *before*. We get the fuzzy value for a point x and an interval j by evaluating $\mathit{before}(j)(x)$. If we extend the point x to an interval i, we can obtain an average value for i *before* j by integrating over i and normalizing the result with $|i|$: $\mathit{before}(i,j) = \int i(x) \cdot \mathit{before}(j)(x)\ dx/|i|$.

The exact definition below is a bit more complicated because it takes into account that i may be empty or infinite.

We summerize the interval-interval relations in the definition below and then explain them in detail one by one.

Definition 2.56 (Interval-Interval Relations)
Let i and j be two fuzzy time intervals. We need the following point-interval operators:
B is a Point-Interval before*-operator (Def. 2.43),*
E^+ is a rising transformation (like extend$^+$) (Def. 2.40),
D is a Interval-Interval during *operation (defined below)*
D_p is a point-interval during*-operator (Def. 2.43),*
S, S_1 and S_2 are point-interval starts*-operators (Def. 2.43),*
F, F_1 and F_2 are point-interval finishes*-operators (Def. 2.43).*
For all 6 starts *and* finishes*-operators O it is assumed that $|O(i)| < \infty$ for all fuzzy time intervals i.*
The interval-interval relations are defined as follows:

$$\text{before}_B(i,j) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } i \text{ is empty or positive infinite or } j \text{ is empty} \\ 1 & \text{if } i \text{ is negative infinite and } i \cap_{min} j = \emptyset \\ \int (i \cap_{min} j)(x) \cdot B(j)(x)\ dx/|i \cap_{min} j| & \text{if } i \text{ is negative infinite} \\ \int i(x) \cdot B(j)(x)\ dx/|i| & \text{otherwise} \end{cases}$$

$$\text{meets}_{F,S}(i,j) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } i \text{ or } j \text{ are empty or } i \text{ is positive infinite or } j \text{ is negative infinite} \\ \int F(i)(x) \cdot S(j)(x)\ dx/N(F(i),S(j)) & \text{otherwise} \end{cases}$$

$$\text{during}_{D_p}(i,j)$$
$$\stackrel{\text{def}}{=} \begin{cases} 1 & \textit{if } i \textit{ is empty} \\ 0 & \textit{if } j \textit{ is empty} \\ 0 & \textit{if } i \textit{ is infinite and } i(-\infty) > j(-\infty) \textit{ or } i(+\infty) > j(+\infty) \\ \int i'(x), D_p(j')(x)\ dx/|i'| & \textit{if } i \textit{ is infinite} \\ \quad \textit{where } i' = cut_{min(i^{fK}, j^{fK}), max(i^{lK}, j^{lK})}(i) \textit{ and } j' = cut_{min(i^{fK}, j^{fK}), max(i^{lK}, j^{lK})}(j) & \\ \int i(x) \cdot D_p(j)(x)\ dx/|i| & \textit{otherwise} \end{cases}$$

$$\text{overlaps}_{E+,D}(i,j) \stackrel{\text{def}}{=} \begin{cases} 0 \quad \textit{if } i \textit{ or } j \textit{ is empty or } j \textit{ is negative infinite} & \\ |i \cap_{min} j|/|cut_{j^{fK}, j^{lK}}(j)| & \textit{if } i \textit{ is negative infinite} \\ (1 - D(i', E^+(j'))) \cdot D(i', j')/N'(i', j') & \textit{if } i \textit{ or } j \textit{ are positive infinite} \\ \qquad \textit{where } i' = cut_{i^{fS}, max(i^{lK}, j^{lK})}(i) \textit{ and } j' = cut_{j^{fS}, max(i^{lK}, j^{lK})}(j) & \\ (1 - D(i, E^+(j))) \cdot D(i,j)/N'(i,j) & \textit{otherwise} \end{cases}$$

$$\textit{where } N'(i,j) \stackrel{\text{def}}{=} \max_a((1 - D(shift_a(i), E^+(j))) \cdot D(shift_a(i), j))$$

$$\text{starts}_{S_1,S_2,D}(i,j) \stackrel{\text{def}}{=} \begin{cases} 0 \quad \textit{if } i \textit{ or } j \textit{ are empty or one of } i \textit{ and } j \textit{ is negative infinite} \\ D(i,j) \quad \textit{if both } i \textit{ and } j \textit{ are negative infinite} \\ \frac{\int S_1(i)(x) \cdot S_2(j)(x)\ dx}{N(S_1(i), S_2(j))} \cdot D(i,j) \quad \textit{otherwise} \end{cases}$$

$$\text{finishes}_{F_1,F_2,D}(i,j) \stackrel{\text{def}}{=} \begin{cases} 0 \quad \textit{if } i \textit{ or } j \textit{ are empty or one of } i \textit{ or } j \textit{ is positive infinite} \\ D(i,j) \quad \textit{if both } i \textit{ and } j \textit{ are positive infinite} \\ \frac{\int F_1(i)(x) \cdot F_2(j)(x)\ dx}{N(F_1(i), F_2(j))} \cdot D(i,j) \quad \textit{otherwise} \end{cases}$$

$$\text{equals}_D(i,j) \stackrel{\text{def}}{=} D(i,j) \cdot D(j,i)$$

where the normalization factor is $N(i,j) \stackrel{\text{def}}{=} min(|i|, |j|)$ *or* $N(i,j) \stackrel{\text{def}}{=} \max_a \int i(x-a) \cdot j(x)\ dx$

Notice that the first factor in the integrals above always denotes a finite interval. This guarantees that the integrals are finite, even if the second term denotes an infinite interval. ∎

The Normalization Factor

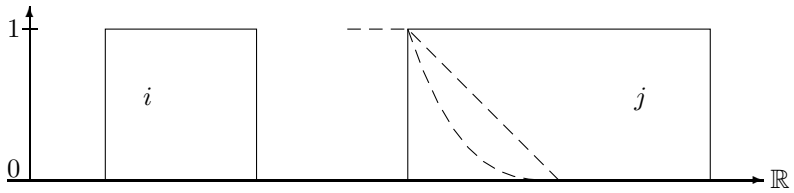
The integrals for relations *meets*, *starts* and *finishes* are normalization by a factor $N(i,j)$. The optimal choice is $N(i,j) \stackrel{\text{def}}{=} \max_a \int i(x-a) \cdot j(x)\ dx$ because this normalizes the integral such that the value 1 is a possible result. The intuition behind this is that for *meets*(i,j) there should be a position of i relative to j where *meets*$(i,j) = 1$. For *starts*(i,j) and *finishes*(i,j) there should be a position of i where at least the first factor which determines whether the left/right end of i and j coincide, is 1. $\max_a \int i(x-a) \cdot j(x)\ dx$ as normalization factor guarantees this. Unfortunately this causes a nontrivial search problem (Sec. 2.9). Therefore there is a second choice for the normalization factor: $N(i,j) \stackrel{\text{def}}{=} min(|i|, |j|)$. This avoids the search problem, but it means that the value of *meets*(i,j) etc. may be always smaller than 1.

We examine the interval-interval relations now in detail.

Before for Finite Intervals

The definition for finite intervals is: $before_B(i,j) \stackrel{\text{def}}{=} \int i(x) \cdot B(j)(x)\ dx/|i|$ where B is a point-interval *before*-relation. The idea of this definition is to average the point-interval *before*-relation over the interval i. The normalization factor is just $|i|$. The rationale behind this is the following: if j is finite then $B(j)(x) = 1$ if x is small enough. If we move the interval i into the area where $B(j)(x) = 1$ then the integral becomes $\int i(x) \cdot B(j)(x)\ dx = \int i(x) \cdot 1 dx = |i|$, such that *before*$(i,j) = 1$. Thus, $|i|$ as normalization factor yields the right result.

Example 2.57 (*before*) *The next picture illustrates the fuzzy* before *relation for two finite crisp intervals. B is the standard point-interval* before *operator of Example 2.44. For this particular operator B and crisp intervals i and j* $\text{before}_B(i,j)$ *yields the percentage of points in i which are before j (in the usual sense). The upper dashed line indicates the fuzzy value at position x if the end of interval i is at this position. The fuzzy value has dropped to 0 only if the interval i is moved completely into j. A steeper decrease can be enforced if the result of* $\text{before}_B(i,j)$ *is for example exponentiated with an exponent > 1. The lower dashed line in the figure therefore indicates* $\text{before}_B(i,j)^3$.

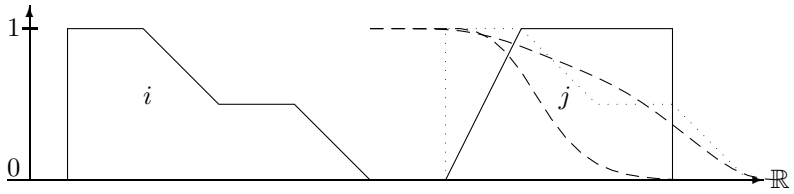


$\text{before}_B(i,j)$ *and* $\text{before}_B(i,j)^3$

■

One may argue whether it is desirable to have a 'before' relation where the standard parameters force the fuzzy value down to 0 only when the interval i is completely contained in j. The counter argument is that a smooth decrease can reveal more information about the structure of i and j than a steep drop. A steep drop to 0 can always be achieved by exponentiating the result with a large exponent.

Example 2.58 (*before* for Fuzzy Intervals) *The next picture shows the* before*-relation for real fuzzy intervals. The upper dashed line indicates the result of the* before*-relation at position x if the positive end of the interval i is moved to x. The lower dashed line is the upper dashed line exponentiated with the exponent 10. The dotted line represents the position of the interval i when the result value is dropped to 0.*



before *for Fuzzy Intervals*

■

Before for Infinite Intervals

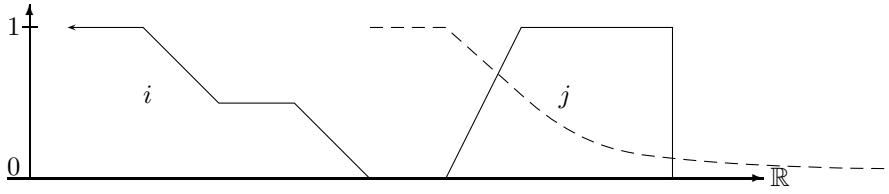
If the interval i is positive infinite, then nothing can be after i. Therefore the relation $\textit{before}(i,j)$ must yield 0. If the interval j is negative infinite, then nothing can be before j. Therefore the relation $\textit{before}(i,j)$ must also yield 0.

If i is negative infinite and j is finite or positive infinite then $\textit{before}(i,j)$ may well be not false. The problem is how to measure the degree of 'beforeness' in this case. Since i is infinite, $\int i \cdot B(j)\ dx$ will always be infinite. An alternative is to take instead of i only the intersection between i and j and to measure the degree of 'beforeness' of $i \cap j$. Since j is not negative infinite, $i \cap j$ is finite. The formula is then

$$\textit{before}_B(i,j) \stackrel{\text{def}}{=} \int (i \cap j)(x) \cdot B(j)\ dx / |i \cap j|$$

.

Example 2.59 (*before* for Infinite Intervals) *This picture shows the development of* $\text{before}_B(i,j)$ *when i is negative infinite and its positive end is moved along the x-axis. The fuzzy value drops down, but not to 0. It remains constant after a while.*



before *for Infinite Intervals*

■

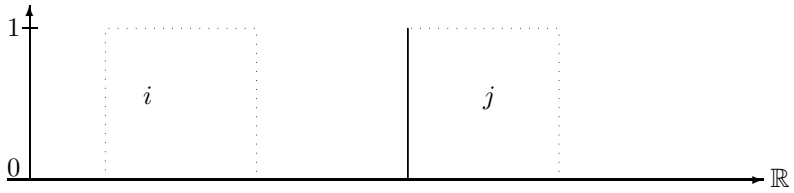
Meets

The classical 'meets' relation yields 'true' if the end of the first interval i touches the beginning of the

second interval j. The back end of i and the front end of j are therefore relevant for evaluating *meets*(i, j). We can get the back end of i in our fuzzy setting with the point-interval *finishes*-operator, and the front end of j with the point-interval *starts*-operator (cf. Example 2.47). The fuzzy *meets*-relation measures how many points in the back end $F(i)$ of i are in the front end $S(j)$ of j and normalizes the value with the maximum possible overlap between $F(i)$ and $S(j)$. Notice that this works only if $|F(i)|$ and $|S(j)|$ are finite. The definition is therefore $meets_{F,S}(i,j) \stackrel{\text{def}}{=} \int_{-\infty}^{+\infty} F(i)(x) \cdot S(j)(x)\ dx / N(F(i), S(j))$.

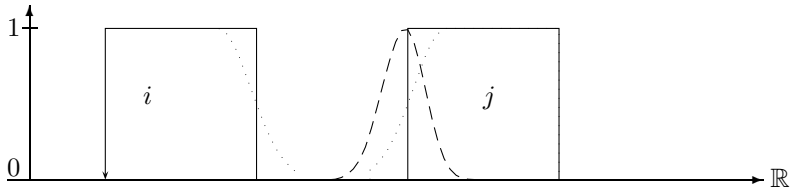
The normalization factor $N(F(i), S(j)) \stackrel{\text{def}}{=} \max_a \int F(i)(x-a) \cdot S(j)(x)\ dx$ amounts to a search problem where $F(i)$ is moved along the x-axis to find the position for i where the integral becomes maximal. This guarantees that there is a position for i where *meets*$(i, j) = 1$.

Example 2.60 (*meets* for Crisp Intervals)
The first picture shows the meets*-relation where for crisp intervals the operators F and S have a singular peek. Consequently the* meets*-relation has also a singular peek when the interval i meets j in the crisp sense.*



Crisp meets *for Crisp Intervals*

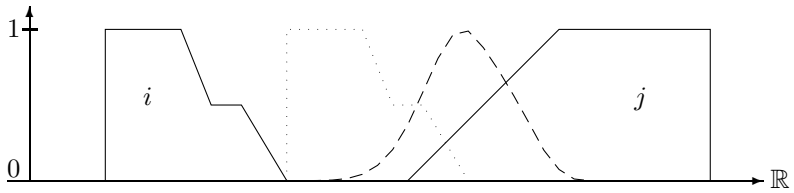
The next picture shows the result of the meets*-relation when the* finishes- *and* starts*-operators fuzzify the crisp sets. The dotted lines show the fuzzified crisp sets (with Gaussian fuzzification). The dashed line is again the result of* meets *when the endpoint of i is moved along the x-axis.*



Fuzzy meets *for Crisp Intervals*

■

Example 2.61 (*meets* for Fuzzy Intervals) *We illustrate the fuzzy* meets*-relation with two fuzzy time intervals and the simple point-interval* finishes *and* starts*-operators of Example 2.47. The dashed line shows the results of the* meets*-relation when the interval i is moved along the x-axis. The dotted figure is the position of i where* meets(i, j) *is maximal.*



meets *for Fuzzy Intervals*

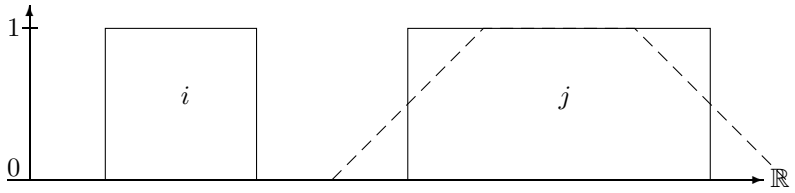
■

During for Finite Intervals
The interval-interval *during*-relation averages the point-interval *during* relation D_p by integrating over the interval i. The basic formula is therefore

$$during_{D_p}(i,j) \stackrel{\text{def}}{=} \frac{\int i(x) \cdot D_p(j)(x)\ dx}{|i|}$$

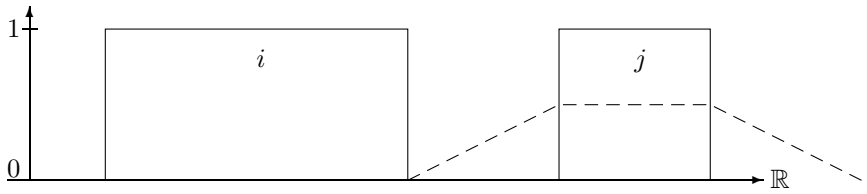
during(i, j) measures to what degree i is contained in j. The normalization factor is $|i|$ because if i is larger than j then *during*(i, j) should definitely be smaller than 1.

Example 2.62 (*during* for Crisp Intervals)
The dashed line in the picture below shows the result of the during*-relation for a coordinate x when the* middle point *of the interval i is moved to x.*



during *for Crisp Intervals*

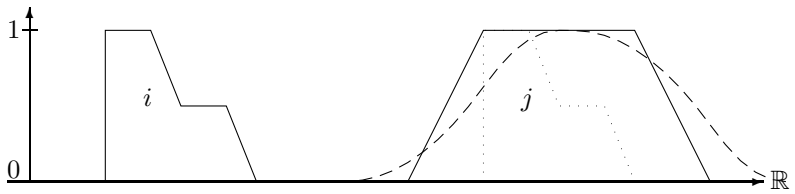
The interval i in the next picture is larger than j such that before(i, j) *never rises to 1.*



during *for Crisp Intervals*

■

Example 2.63 (*during* for Fuzzy Intervals)
The dashed line shows again the result of the during*-relation when the middle point of i is moved along the x-axis. The dotted figure indicates the position of i where* during(i, j) *is maximal.*



during *for Fuzzy Intervals*

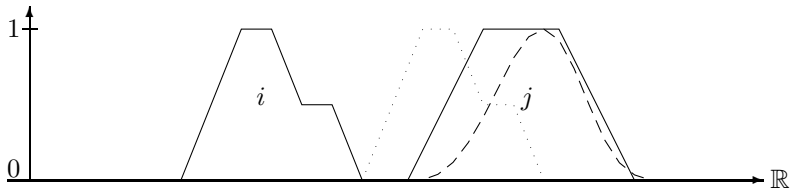
■

During for Infinite Intervals
If i is infinite or $i(-\infty) > j(-\infty)$ and $i(+\infty) > j(+\infty)$ then i is infinitely larger than j. Therefore *during*(i, j) must be 0. If i is infinite and j is infinite in the same way, one can compare the finite kernels of i and j and compute the *during*(i', j')-relation for them. This is the fourth clause in the definition of *during*.

Overlaps for Finite Intervals
The classical relation i*overlaps* j has two requirements:
1. a non-empty part i_1 of i must lie before j, and
2. another non-empty part i_2 of i must lie inside j.

The first condition is encoded in the factor $1 - D(i, E^+(j))$ where D is a *during*-operator. $E^+(j)$ extends the rising part of j to infinity. Therefore $D(i, E^+(j))$ measures the part of i which is after the front part of j. $1 - D(i, E^+(j))$ then measures the part of i which is before the front part of j. This factor is multiplied with $D(i, j)$ which corresponds to the second condition. It measures to which degree i is contained in j. The product is normalized with $\max_a((1 - D(shift_a(i), E^+(j))) \cdot D(shift_a(i), j))$ which corresponds to the maximal possible overlap when i is shifted along the x-axis. This guarantees that there is a position for i where *overlaps*$(i, j) = 1$.

Example 2.64 (*overlaps* for Fuzzy Intervals) *This example shows the result of the* overlaps *relation where the standard* during*-operator is used (with the identity function as point-interval* during*-operator).*



Example: Overlaps Relation

The dashed line represents the result of the overlaps relation for an x-coordinate x where the positive end of the interval i is moved to x. The dotted figure indicates the interval i moved to the position where overlaps(i, j) *becomes maximal.*

■

The normalization factor $\max_a((1 - D(shift_a(i), E^+(j))) \cdot D(shift_a(i), j))$ causes again a search problem. As one can see in the above example the search space is usually very simple (if the intervals are not too exotic). There is only one global maximum and no local maxima. Therefore standard hill climbing is an efficient search method in this case.

Overlaps for Infinite Intervals
If the interval j is negative infinite then there cannot be anything before j. Therefore $overlaps(i, j) = 0$. If i is negative infinite and j is not negative infinite, then there is definitely a part of i before j. It is, however, impossible to measure to what degree i is contained in j. In order to get any number which has something to do with i being contained in j I choose $|i \cap j|/|cut_{j\urcorner K, j\urcorner K}(j)|$.

If i or j are positive infinite, it has in addition to be checked whether some part of i is before j. To this end, $before(i', j')$ is computed, where i' is i restricted the finite kernel and j' is j restricted the finite kernel.

Starts and **Finishes**
The fuzzy *finishes*-relation is the symmetric variant of the fuzzy *starts*-relation. Therefore we need to consider only the *starts*-relation. The crisp i *starts* j-relation has two conditions:

1. the start point of i and the start point of j are identical, and
2. i is a subset of j.

This led to the basic definition of the fuzzy *starts*-relation as a product of the overlap between the two starting sections of i and j, and the $during(i, j)$-relation:

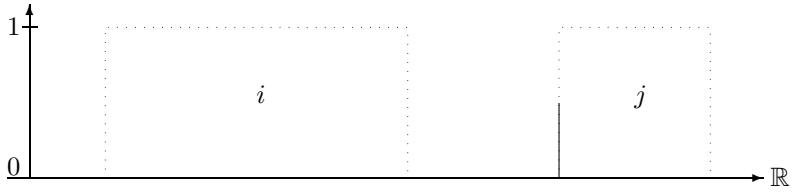
$$starts(i, j) \stackrel{\text{def}}{=} \frac{\int S_1(i)(x) \cdot S_2(j)(x)\ dx}{N(S_1(i), S_2(j))} \cdot D(i, j)$$

The first factor checks the first condition: the starting part $S_1(i)$ of i should coincide with the starting part $S_2(j)$ of j. This value is normalized to the maximal possible overlap of the starting parts. The assumption here is that if I move i along the $\mathbb{R}$-axis, there should be a position of i where i definitely starts j, and where therefore the fuzzy value should be 1. The second factor checks whether i is a subset of j. This factor need not be 1 if i is larger than j. Therefore the result of $starts(i, j)$ can be < 1 regardless of the position of i.

The extreme cases are:

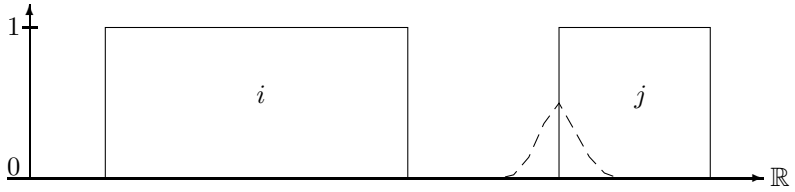
- if i or j are empty then $starts(i, j)$ must be 0;
- if one of i and j is negative infinite then they can't have the same starting point. Therefore $starts(i, j)$ must be 0 again;
- if both are negative infinite then we can assume that they have the same starting point, and therefore only the second condition $during(i, j)$ must be checked;
- it does not matter whether i or j are positive infinite because only the finite starting sections of i and j count.

Example 2.65 starts *for Crisp Intervals*
The first picture shows the case where a during *operator D is used with $D_d = identity$. If the back end of i is moved along the x-axis we get a single peak when it meets j. The peak, however, is only 0.5 high because i is twice as large as j.*



Crisp starts-*Relation*

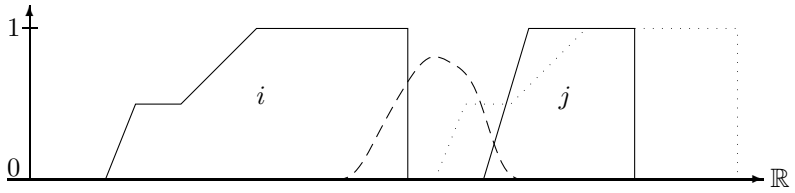
The next picture shows a fuzzified starts-*relation. The dashed line shows the value of* starts(i, j) *for a position x where the* front end *of i is moved to x. The crisp intervals are fuzzified in the same way as in Example 2.60. The peak is broader, but the maximum is still at 0.5 because i is twice as large as j.*



Fuzzified starts-*relation*

■

Example 2.66 (*starts*-Relation for Fuzzy Intervals) *The next figure shows the application of the same* starts-*relation as in the first picture of the above example to fuzzy intervals. The dashed line is again the result of the* starts-*relation. The dotted figure shows the position of the interval i where* starts(i, j) *is maximal.*



Example: Starts Relation

■

Equals:
An interval i equals an interval j if i is a subset of j and vice versa. Therefore we get

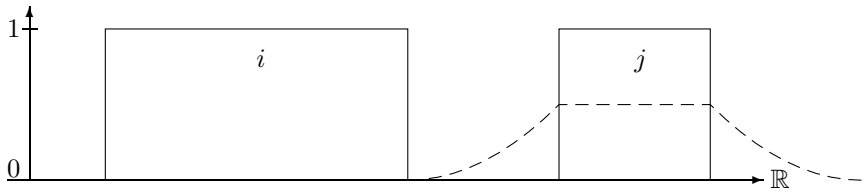
$$equals_D(i, j) \stackrel{\text{def}}{=} D(i, j) \cdot D(j, i)$$

where D is an interval-interval *during*-operator.

Example 2.67 (*equals* for Crisp Intervals)
The first picture shows the equals-*relation for similar intervals. If i is moved on top of j then* equals$(i, j) =$ 1

equals *for Similar Intervals*

i and j in the next figure are not equal. Therefore equals(i, j) *never rises to 1.*

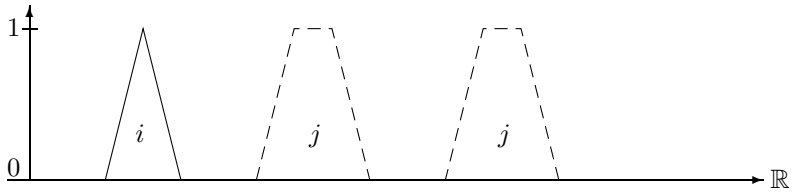


equals *for Different Intervals*

■

2.9 The Search Problem

The normalization factor $N(i, j) = \max_a \int i(x - a) \cdot j(x)\ dx$, where i is finite, amounts in general to a nontrivial search problem with unpredictable solutions. Consider the following example:



Maximizing 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 maximized 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.

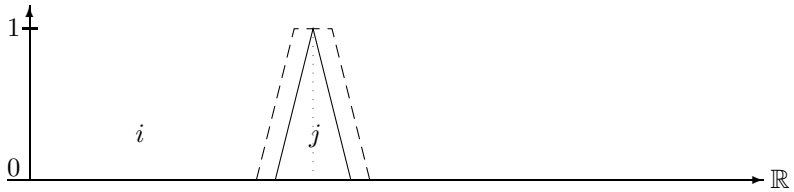
The first class is when j is infinite and $j(-\infty) = 1$ or $j(+\infty) = 1$, and, of course j has a finite kernel. in this case one can move i to the infinite part where j is constant 1. In this case $\int i(x - a)j(x)\ dx = |i|$, i.e. $\max_a \int i(x - a) \cdot j(x)\ dx = |i|$,

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

Definition 2.68 (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.

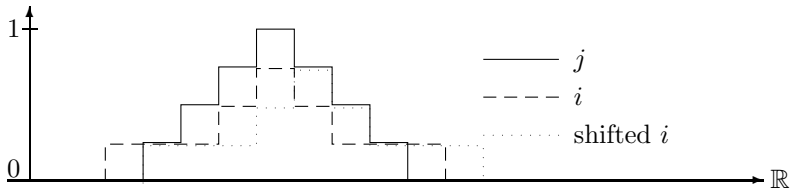


Maximal Overlap

Proposition 2.69 *If i and j are two monotone and symmetric fuzzy intervals then $\int_{-\infty}^{+\infty} i(x)j(x)\ dx$ 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 discretesized into step functions with finite step size. The limit 'step size $\mapsto$ 0' is then the original problem. The discretesized integral then becomes a sum $stepsize \cdot \Sigma_k i_k \cdot j_k$

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

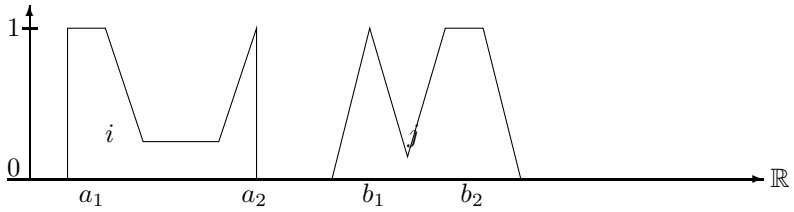


Discretesized Maximization Problem

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 looses 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 $\int_{-\infty}^{+\infty} i(x-a)j(x)\ dx$ 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 FuTIRe is a combination of an iterated binary local search with a randomized global search. It is optimized 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.



Starting Points for the Local Search

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 2.70 (The Search for Maximizing the Overlap) *Let i and j be two* finite *fuzzy intervals. We define a local search function and then a global search procedure for maximizing the integral*

$$Int(a) \stackrel{\text{def}}{=} \int_{-\infty}^{+\infty} i(x-a)j(x)\ dx.$$

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. ∎

3 Datastructures and Algorithms

The algorithms presented in this document need to deal with five basic datatypes: time points, fuzzy values, fuzzy temporal intervals, y-functions and interval-operators.

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 the two lowest hyperfine levels of the Cesium 133 atom. Therefore the most precise time measurement available at all depends on counting integers (cycles of light).

Therefore the FuTIRe-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 Cesium 133 light.

Fuzzy Values
Fuzzy values usually are 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 I decided again to use 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 [7, 5]; 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) *FuTIRe may use 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.*

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 modularized and easy to understand implementations, but it also makes the mathematical notation clearer. Therefore we shall use the notion $o.v$ and $o.m(p_1, \ldots, p_n)$ where o is an object, v is an instance variable, and m is a method (function) with arguments $p_1, \ldots, p_n$.

The expression $\min(i \geq 0 \mid \varphi(i))$ denotes a loop: starting with $i = 0$, increase i by 1 until $\varphi(i)$ becomes true. In this case return the i with $\varphi(i) = \mathit{true}$. Notice that the loop may in general not terminate. We use similar expressions with the obvious meaning.

The expression

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

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 $\varphi_1, \ldots, \varphi_{n-1}$ 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

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

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*'.

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

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 [7]. 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 $p_1.colinear(p_2, p_3)$ 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.* ■

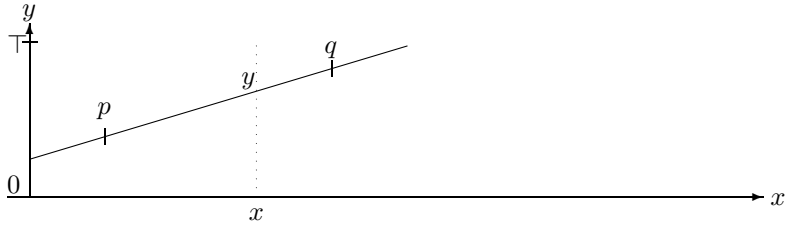
The $leftturn$ method computes the doubled area of the triangle p_1, p_2 and p_3 and checks its sign. Left turns and right turns yield opposite signs.

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

Definition 3.5 (IntersectsProper) *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.



$p.LineY(q, x)$

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

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

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.*

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

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 $p.Area2(q)$ 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 $p.Area2(q, x_1, x_2)$ 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.*

$p.Area2(q) \stackrel{\text{def}}{=} (q.x - p.x) \cdot (q.y + p.y)$

The result is an x-integer.

$$p.Area2(q,x_1,x_2) \stackrel{\text{def}}{=} \begin{cases} undefined & \text{if } p.x = q.x \text{ and } p.x \neq x \\ 0 & \text{if } p.x = q.x = x \\ (x_2 - x_1)\cdot(p.LineY(q,x_2) - p.LineY(q,x_1)) & \text{otherwise} \end{cases}$$

The result is a floating point number. ∎

The next method, $p.Area2X(q,a)$ 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.*

$$p.Area2X(q,a) \stackrel{\text{def}}{=} \begin{cases} undefined & \text{if } p.x = q.x \\ & \text{or } p.y = q.y = 0 \text{ and } a > 0 \\ & \text{or } p.y^2 < -slope \cdot a \\ p.x & \text{if } p.y = q.y = 0 \text{ and } a = 0 \\ p.x + roundX(\frac{a}{2p.y}) & \text{if } p.y = q.y \\ p.x + roundX(\frac{\sqrt{p.y^2 + slope\cdot a} - p.y}{slope}) & \text{otherwise} \\ \quad \text{where } slope = \frac{q.y-p.y}{q.x-p.x} & \end{cases}$$

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 $p.Area2X(q,a)$ 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

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

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

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
$(x - p.x) \cdot 2p.y+ = a$ with solution
$x = p.x + \frac{a}{2p.y}$.
Case 3: $slope \neq 0$:
The equation is normalized to
$slope \cdot (x - p.x)^2 + 2p.y(x - p.x) - a = 0$ with solution
$(x - p.x) = \frac{-2p.y \pm \sqrt{4p.y^2 + 4slope \cdot a}}{2slope}$
$x = p.x + \frac{-p.y + \sqrt{p.y^2 + slope \cdot a}}{slope}$
The $-\sqrt{\ldots}$-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
The Interval-Interval relations (Section 3.2.7) are defined as an integral over two multiplied polygons. 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.*

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

where $a \stackrel{\text{def}}{=} p_1.y - m_1 p_1.x$, $b \stackrel{\text{def}}{=} q_1.y - m_2 q_2.x$,

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

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:
$l_1(x) \stackrel{\text{def}}{=} p_1.y + m_1(x - p_1.x)$
$l_2(x) \stackrel{\text{def}}{=} q_1.y + m_2(x - q_1.x)$
where $m_1 \stackrel{\text{def}}{=} \frac{p_2.y - p_1.y}{p_2.x - p_1.x}$ and $m_2 \stackrel{\text{def}}{=} \frac{q_2.y - q_1.y}{q_2.x - q_1.x}$.
$\int_{x_1}^{x_2} l_1(x) \cdot l_2(x)\ dx$
$= \int_{x_1}^{x_2} (p_1.y + m_1(x - p_1.x))(q_1.y + m_2(x - q_1.x))\ dx$
$= [(p_1.y - m_1 p_1.x)(q_1.y - m_2 q_2.x) + (m_2(p_1.y - m_1 p_1.x) + m_1(q_1.y - m_2 q_2.x))x + m_1 m_2 x^2]_{x_1}^{x_2}$
$= [ab + (m_2 a + m_1 b)x + m_1 m_2 x^2]_{x_1}^{x_2}$
$= ab(x_2 - x_1) + (m_2 a + m_1 b)(x_2^2 - x_1^2)/2 + m_1 m_2 (x_2^3 - x_1^3)/3$
where $a \stackrel{\text{def}}{=} p_1.y - m_1 p_1.x$ and $b \stackrel{\text{def}}{=} q_1.y - m_2 q_2.x$. ∎

3.2 Fuzzy Time Intervals

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

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 the largest representable x-integer, and $-\infty$ stands for the smallest representable x-integer.* ∎

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* $p_0, \ldots, p_n$ *such that* $p_i.x \leq p_{i+1}.x$ *holds for all* i.

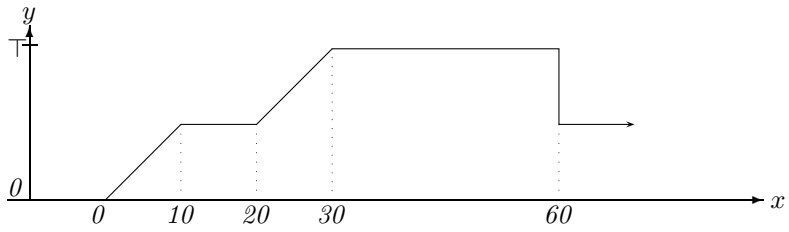
In most cases we can assume that the coordinates in I *are not redundant, i.e. there are no collinear triples* (p_i, p_{i+1}, p_{i+2}) *of points.*

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

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

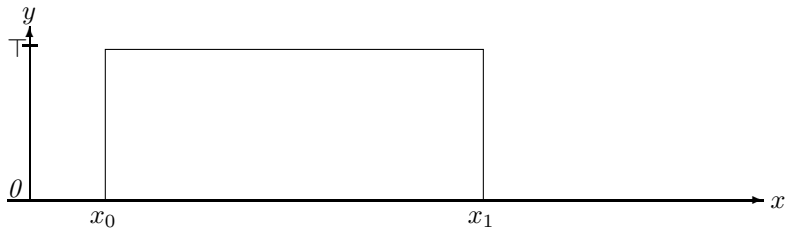
$$I = (0,0)(10,500)(20,500)(30,1000)(60,1000)(60,500).$$

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



The Envelope Polygon

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



Finite Crisp Interval

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

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* $I = (p_0, \ldots, p_n)$ *be an envelope polygon and* p *a new point.*

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

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

The method *I.close*() defined next 'closes' a polygon in the sense that it assumes that all points have been added with the *push_back*-method and some further redundancies can be removed. For some of the other algorithms it is important that these redundancies are in fact removed.

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

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

■

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 $I = (p_0, \ldots, p_n)$ 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.

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

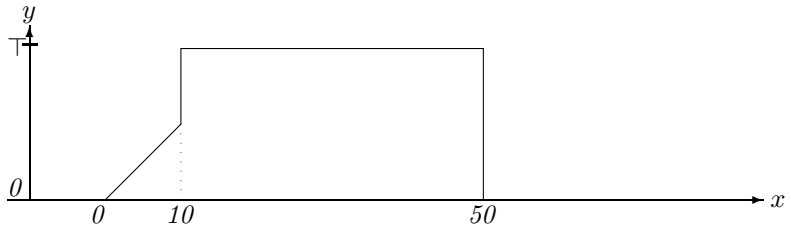
■

$IndexMax$ 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) $I = (p_0, \ldots, p_n)$ the* Member function *is defined:*

$$I.Member(x) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } I = () \\ p_0.y & \text{if } x < x_0 \\ p_n.y & \text{if } x \geq x_n \\ p_i.y & \text{if } p.x = p_i.x \\ p_i.lineY(p_{i+1}, x) & \text{otherwise} \\ \quad \text{where } i = I.Index(x) & \end{cases}$$

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

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

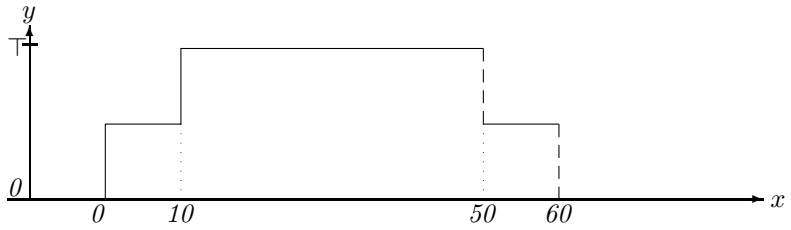
Remark 3.23 (Extrapolation and Infinite Intervals) *The Member method extrapolates the membership function to x-coordinates below $p_0.x$ and above $p_n.x$. The y-value for x-coordinates below $p_0.x$ is constant $p_0.y$. The y-value for x-coordinates above $p_n.x$ is constant $p_n.y$. Therefore envelope polygons always represent fuzzy intervals* with finite kernel *(Def. 2.4).* ■

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:

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



Half-Open Interval

We have $I.Member(0) = 500$, $I.Member(10) = 1000$, $I.Member(50) = 500$, $I.Member(60) = 0$. ■

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;*
- $I = ((a, y))$ *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*

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

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

■

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* $I = (p_0, \ldots, p_n)$ *be an envelope polygon.*

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

■

Using the $IndexMax$-method (Def. 3.20) we can define $I.Max()$ for computing the height $I\hat{}$ (Def. 2.10) of the fuzzy interval. We also define $MaxX(front)$ to compute the x-coordinate of the first maximal point i^{fm} (Def. 2.11) if $front = true$ and $MaxX(false)$ which computes the last maximum i^{lm} (Def. 2.11).

Definition 3.27 (Max Values) *Let* $I = (p_0, \ldots, p_n)$ *be an envelope polygon.*

$$I.Max() \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } I = () \\ p_{I.MaxIndex(true)}.y & \text{otherwise} \end{cases}$$

$$I.MaxX(true) \stackrel{\text{def}}{=} \begin{cases} -\infty & \text{if } I = () \text{ or } I.IndexMax(true) = 0 \text{ and } p_0.y > 0 \\ p_{I.IndexMax(true)}.x & \text{otherwise} \end{cases}$$

$$I.MaxX(false) \stackrel{\text{def}}{=} \begin{cases} +\infty & \text{if } I = () \text{ or } I.IndexMax(false) = n \text{ and } p_n.y > 0 \\ p_{I.IndexMax(false)}.x & \text{otherwise} \end{cases}$$

The result of Max is an y-integer value and the result of $MaxX$ is an x-integer value. ■

The complexity of Max and $MaxX$ is in general linear because $IndexMax$ requires linear search. It is constant for crisp intervals.

Proposition 3.28 (Soundness of Max and MaxX) *For an envelop polygon I we have $I.Max()/\top = I\hat{}$, $I.MaxX(true) = I^{fm}$ and $I.MaxX(false) = I^{lm}$.*

The proofs are straightforward. ∎

Size of Fuzzy Intervals

The *size* of a fuzzy interval is the integral over its membership functions (Def. 2.8). We define now three methods for computing the (doubled) size of a fuzzy interval. $Size2()$ computes the overall size, i.e. $I.Size()/\top = 2|I|$. $I.Size2(k,l)$ 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 3.29 (Size) *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon. Let k and l be two indices.*

$$I.Size2I(k,l) \stackrel{\text{def}}{=} \begin{cases} undefined & \text{if } k < 0 \text{ or } l > n \\ -I.Size2(l,k) & \text{if } l < k \\ 0 & \text{if } k = l \text{ or } I = () \\ \Sigma_{m=k}^{l-1} p_m.Area2(p_{m+1}) & \text{otherwise} \quad (Def.3.9) \end{cases}$$

$$I.Size2() \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } I = () \\ +\infty & \text{if } p_0.y > 0 \text{ or } p_n.y > 0 \\ I.Size2(0,n) & \text{otherwise} \end{cases}$$

Both versions of $Size2$ return x-integers.

Now let a and b be two x-coordinates:

$$I.Size2(a,b) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } I = () \text{or } a = b \\ -I.Size2(b,a) & \text{if } b < a \\ 2 \cdot (b-a) \cdot p_n.y & \text{if } a \geq p_n.x \\ 2 \cdot (b-a) \cdot p_0.y & \text{if } b \leq p_0.x \\ (b-a) \cdot (p_i.LineY(p_{i+1}, a) + p_i.LineY(p_{i+1}, b)) & \text{if } p_{i-1}.x \leq a \leq b \leq p_i.x \\ \quad \text{where } i = I.Index(a) & \\ head + middle + tail & \text{otherwise} \end{cases}$$

where

$$head \stackrel{\text{def}}{=} \begin{cases} 2 \cdot (p_0.x - a) \cdot p_0.y & \text{if } a \leq p_0.x \\ 0 & \text{if } p_i.x = a \\ p_i.Area2(p_{i+1}, a, p_{i+1}.x) & \text{otherwise} \\ \quad \text{where } i = I.Index(a) \text{ and} & \end{cases}$$

$$middle \stackrel{\text{def}}{=} I.Size2(I.Index(a), I.Index(b)) \text{ and}$$

$$tail \stackrel{\text{def}}{=} \begin{cases} 2 \cdot (b - p_n.x) \cdot p_n.y & \text{if } b \geq p_n.x \\ 0 & \text{if } p_i.x = b \\ p_i.Area2(p_{i+1}, p_i.x, b) & \text{otherwise} \\ \quad \text{where } i = I.Index(b) & \end{cases}$$

The method returns a floating point value. ∎

The next two methods compute the center and middle points for a fuzzy interval (Def. 2.12).

Definition 3.30 (Center Points) *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon.*

$$I.CenterPoint(k,m) \stackrel{\text{def}}{=} \begin{cases} undefined & \text{if } I = () \text{ or } I.isInfinite() \\ p_0.x & \text{if } k = 0 \\ p_n.x & \text{if } k = m \\ p_{i-1}.Area2X(p_i, \frac{s2 \cdot k}{m} - I.Size2(0, i-1)) & \text{otherwise} \\ \quad \text{where } s2 \stackrel{\text{def}}{=} I.Size2(0,n) \text{ and} & \\ \quad i = \min(i \mid m \cdot I.Size2(0,i) > s2 \cdot k) & \end{cases}$$

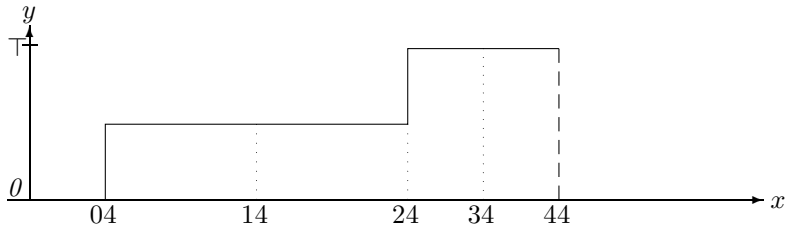
$I.MiddlePoint(n,m) \stackrel{\text{def}}{=} I.CenterPoint(2n+1, 2m)$.

Both functions return a (rounded) x-integer.

The search for the index i in $CenterPoint$ causes linear complexity for both methods. ∎

The CenterPoint 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 3.31 (Center Point Computation)
Let $I = \underbrace{(0,0)}_{p_0}\underbrace{(0,500)}_{p_1}\underbrace{(4,500)}_{p_2}\underbrace{(4,1000)}_{p_3}\underbrace{(6,1000)}_{p_4}\underbrace{(6,0)}_{p_5}$



Center Point Computation

We have $|I| = 4000$*, i.e.* $s2 = 8000$*, and we want to compute* $CenterPoint(1,4)$*. 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 3.32 (Number of Components) *Let* $I = (p_0, \ldots, p_n)$ *be an envelope polygon.*

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

■

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

Definition 3.33 (Component) *Let* $I = (p_0, \ldots, p_n)$ *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 $l \stackrel{\text{def}}{=} 0$*.*
$For_{i=1}^{n}$ $\{if(p_i.y = 0 \text{ and } p_{i-1}.y > 0)\ l = l + 1;$ // *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*

■

Core, Support and Kernel (Def. 2.4)
The FuTIRe-library provides for each of these three concepts the following 5 methods:

1. $Size()$ measures the size in x-coordinates;
2. $List()$ yields an ordered list $((i_0, j_0), \ldots)$ of indices of start and endpoints of the components;

3. $Crisp()$ yields a new envelope polygon with the crisp versions of the components.

4. $First()$ returns the first x-coordinate of the concept

5. $Last()$ returns the last x-coordinate of the concept.

$Size()$, $First()$ and $Last()$ return x-coordinates, $Crisp()$ returns an envelope polygon and $List()$ returns a list of index pairs.

Definition 3.34 (Algorithms for the Core) *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon.*

$$I.CSize() \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } I = () \\ +\infty & \text{if } p_0.y = \top \text{ or } p_n.y = \top \\ \Sigma_{i=0}^{n-1} \begin{cases} p_{i+1}.x - p_i.x & \text{if } p_i.y = p_{i+1}.y = \top \\ 0 & \text{otherwise} \end{cases} & \text{otherwise} \end{cases}$$

$$I.CList() \stackrel{\text{def}}{=} \begin{cases} () & \text{if } I = () \\ V_{i=0}^{n} \begin{cases} ((0,0)) & \text{if } i = 0 \text{ and } p_0.y = \top \\ ((n,n)) & \text{if } i = n \text{ and } p_n.y = \top \\ (i, i+1) & \text{if } i < n \text{ and } p_i.y = p_{i+1}.y = \top \\ () & \text{otherwise} \end{cases} & \text{otherwise} \end{cases}$$

$$I.CCrisp() \stackrel{\text{def}}{=} \begin{cases} () & \text{if } I = () \\ V_{i=0}^{n} \begin{cases} ((p_0.x, \top)(p_0.x, 0)) & \text{if } i = 0 \text{ and } p_0.y = \top \\ ((p_n.x, 0)(p_n.x, \top)) & \text{if } i = n \text{ and } p_n.y = \top \\ ((p_i.x, 0)(p_i.x, \top)(p_{i+1}.x, \top)(p_{i+1}.x, 0)) & \text{if } i < n \text{ and } p_i.y = p_{i+1}.y = \top \\ () & \text{otherwise} \end{cases} \end{cases}$$

$$I.CFirst() \stackrel{\text{def}}{=} \begin{cases} undefined & \text{if } I = () \\ -\infty & \text{if } p_0.y = \top \\ \min(p_i.x \mid p_i.y = \top) & \text{if this is defined} \\ undefined & \text{otherwise} \end{cases}$$

$$I.CLast() \stackrel{\text{def}}{=} \begin{cases} undefined & \text{if } I = () \\ +\infty & \text{if } p_n.y = \top \\ \max(p_i.x \mid p_i.y = \top) & \text{if this is defined} \\ undefined & \text{otherwise} \end{cases}$$

■

Definition 3.35 (Algorithms for the Support) *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon.*

$$I.SSize() \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } I = () \\ +\infty & \text{if } I.isInfinite() \\ \Sigma_{i=0}^{n-1}(p_{i+1}.x - p_i.x) & \text{if } p_i.y > 0 \text{ or } p_{i+1}.y > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$I.SList() \stackrel{\text{def}}{=} \begin{cases} () & \text{if } I = () \\ (V_{i=1}^{n-1} \begin{cases} ((s,i)) \text{ and } s := i+1 & \text{if } p_i.y = p_{i+1}.y = 0 \text{ and } p_{i-1}.y > 0 \\ () & \text{otherwise} \end{cases} & \\ (s,n)) & \text{otherwise} \\ \text{where } s := 0 \text{ initially} & \end{cases}$$

$$I.SCrisp() \stackrel{\text{def}}{=} \begin{cases} () & \text{if } I = () \\ \left(\begin{array}{l} \begin{cases} ((p_0.x,0)(p_0.x,\top)) & \text{if } p_0.y = 0 \\ ((p_0.x,\top)) & \text{otherwise} \end{cases}, \\ V_{i=1}^{n-1} \begin{cases} ((p_i.x,\top)(p_i.x,0)) & \text{if } p_i.y = p_{i+1}.y = 0 \text{ and } p_{i-1}.y > 0 \\ ((p_i.x,0)(p_i.x,\top)) & \text{if } p_i.y = p_{i-1}.y = 0 \text{ and } p_{i+1}.y > 0 \\ () & \text{otherwise} \end{cases}, \\ \begin{cases} ((p_n.x,\top)(p_n.x,0)) & \text{if } p_n.y = 0 \\ ((p_n.x,\top)) & \text{otherwise} \end{cases} \end{array} \right) \\ \text{otherwise} \end{cases}$$

$$I.SFirst() \stackrel{\text{def}}{=} \begin{cases} undefined & \text{if } I = () \\ -\infty & \text{if } p_0.y > 0 \\ p_0.x & \text{otherwise} \end{cases}$$

$$I.SLast() \stackrel{\text{def}}{=} \begin{cases} undefined & \text{if } I = () \\ +\infty & \text{if } p_n.y > 0 \\ p_n.x & \text{otherwise} \end{cases}$$

■

Definition 3.36 (Algorithms for the Kernel) *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon.*

$$I.KSize() \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } I = () \\ p_n.x - p_0.x & \text{otherwise} \end{cases}$$

$$I.KList() \stackrel{\text{def}}{=} \begin{cases} () & \text{if } I = () \text{ or } p_0.x = p_n.x \\ (0, n) & \text{otherwise} \end{cases}$$

$$I.KCrisp() \stackrel{\text{def}}{=} \begin{cases} () & \text{if } I = () \text{ or } p_0.x = p_n.x; \\ ((p_0.x, 0)(p_0.x, \top)(p_n.x, \top)(p_n.x, 0)) & \text{otherwise} \end{cases}$$

$$I.KFirst() \stackrel{\text{def}}{=} \begin{cases} undefined & \text{if } I = () \\ p_0.x & \text{otherwise} \end{cases}$$

$$I.KLast() \stackrel{\text{def}}{=} \begin{cases} undefined & \text{if } I = () \\ p_n.x & \text{otherwise} \end{cases}$$

■

3.2.3 Hull Operators

The method $I.CrispHull()$ implements the $CrH()$-function (Def. 2.30).

Definition 3.37 (Crisp Hull) *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon.*

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

■

The method $I.MonotoneHull()$ implements the $MoH()$-function (Def. 2.32). 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 3.38 (Monotone Hull) *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon. We describe the algorithm $I.MonotoneHull()$ procedurally:*

If $I = ()$ return ();
Let $newI_1 \stackrel{\text{def}}{=} (p_0, V_{i=0}^{I.FirstMax()} \begin{cases} (p_i) & \text{if } p_i.y \geq q_m.y \\ () & \text{otherwise} \end{cases})$
where q_m is always the current last element of $newI_1$:
Let $newI_2 \stackrel{\text{def}}{=} (p_n, V_{i=n}^{I.LastMax()} \begin{cases} (p_i) & \text{if } p_i.y \geq q_m.y \\ () & \text{otherwise} \end{cases})$
where and q_m is the current last element of $newI_2$.
Let $newI_2 = (q_0, \ldots, q_m)$;
For$_{i=m}^{0}$ $newI_1.push\_back(q_i)$.
return $newI_1$.

■

Finally we implement the convex hull function CoH (Def. 2.31). The algorithm 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 3.39 (Convex Hull) *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon. We describe the algorithm $I.ConvexHull()$ procedurally:*

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

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

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

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

For $^{i_l}_{i=i_f}$ $\quad while(m \geq 1 \text{ and } q_{m-1}.lefttturn(q_m, p_i))\ newI.pop\_back();$
$\quad newI.push\_back(p_i);$

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

return $newI$. ∎

3.2.4 Basic Unary Transformations

A number of basic unary transformations (Def. 2.33) can be implemented by just manipulating the vertices of the envelope polygons.

Definition 3.40 (Extend, Scaleup, Shift) *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon.*

$$I.Extend(true) \stackrel{\text{def}}{=} \begin{cases} () & \text{if } I = () \\ (V_{i=0}^{j}(p_i), (p_j.x, \top)) & \text{otherwise} \\ & \text{where } j = I.IndexMax(true) \end{cases}$$

$$I.Extend(false) \stackrel{\text{def}}{=} \begin{cases} () & \text{if } I = () \\ ((p_j.x, \top), V_{i=j}^{n}(p_i)) & \text{otherwise} \\ & \text{where } j = I.IndexMax(false) \end{cases}$$

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

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

∎

$Extend(true)$ implements $extend^+$, $Extend(false)$ implements $extend^-$, $ScaleUp$ implements *scaleup* and $Shift$ implements *shift* (Def. 2.33). $ScaleUpD$ is a destructive version of $ScaleUp$ and $ShiftD$ is a destructive version of $Shift$.

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 3.41 (Cut) *Let $I = (p_0, \ldots, p_n)$ 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.

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

$$I.Cut(x, true) \stackrel{\text{def}}{=} ((x, 0), (x, roundY(I.Member(x))), V_{i=I.Index(x)}^{n} p_i)$$

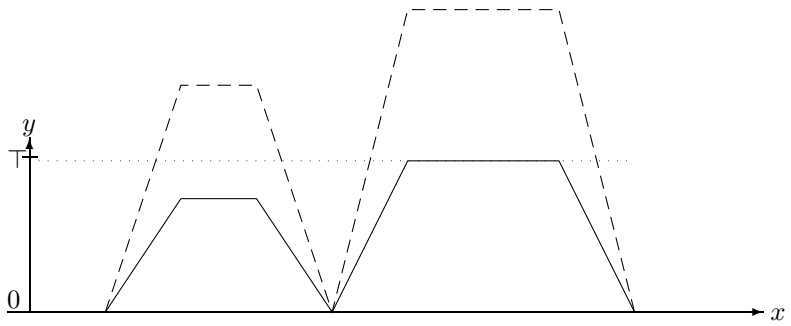
$$I.Cut(x, false) \stackrel{\text{def}}{=} (V_0^{i=I.Index(x)} p_i, (x, roundY(I.Member(x))), (x, 0))$$

∎

Times

The *times*-operator, which multiplies the membership function with a constant, is not so easy to imple-

ment. Since $y \cdot a > \top$ is possible, one has to cut the multiplied envelope polygon at $y = \top$. The picture below illustrates the problem.



Times

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 3.42 (Times) *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon and a a non-negative floating point number.*

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

■

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.

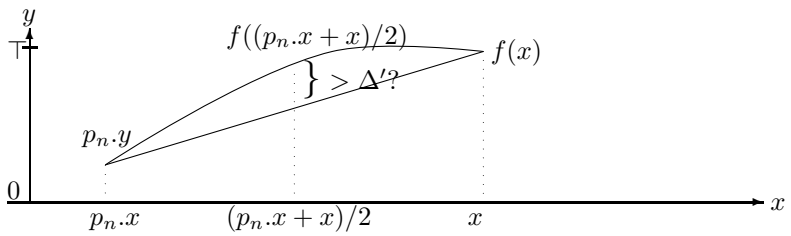
Definition 3.43 (Interpolation) *Let $I = (p_0, \ldots, p_n)$ be a non-empty envelope polygon, x an x-coordinate, f a function from x-coordinate $\mapsto$ y-coordinate and Δ a threshold value (e.g. $\Delta = 0.1$).*

$$I.Interpolate(x, f, \Delta) \stackrel{\text{def}}{=}$$

$$\begin{cases} I & \text{if } x \leq p_n.x \\ I.Interpolate(roundX((p_n.x + x)/2), f, \Delta).Interpolate(x, f, \Delta) & \\ \quad \text{if } |2y_1 - y_2| > \Delta' y_2 & \\ \quad \text{where } y_1 = f(roundX((p_n.x + x)/2)) \text{ and } y_2 = p_n.y + f(x) \text{ and } \Delta' = \Delta/(1 + 2y_2/\top) & \\ I.push\_back((x, f(x))) & \text{otherwise} \end{cases}$$

■

The $Interpolate$-method starts with an envelope polygon $I = ((x_0, y_0))$ and fills up I with interpolated values. Suppose $I = (p_0, \ldots, p_n)$. 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.



Interpolation

Integration
The $integrate^+$-function (Def. 2.33) 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 3.44 (Integration) *Let $I = (p_0, \ldots, p_n)$ 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 $newI = (q_0, \ldots, q_m)$
  return $newI$.

$I.Integrate(false)$:
  if $I = ()$ then return ()
  Let $newI \stackrel{\text{def}}{=} (p_n.x, 0)$
  $For_{i=n}^{1}$ $newI.Interpolate(p_{i-1}.x, \lambda(x)(q_m.y + p_{i-1}.Area2(p_i, x, false)/2), \Delta)$
  return $newI$ reversed. ■

3.2.5 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 3.45 (Unary Transformation) *Let $I = (p_0, \ldots, p_n)$ 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 $newI \stackrel{\text{def}}{=} (p_i.x, f(p_i.y))$;
  $For_{i=1}^{n}$ $newI.Interpolate(p_i.x, \lambda(x) f(p_{i-1}.LineY(p_i, x)), \Delta)$;
  return $newI$; ■

Exponentiation
The exponentiation operator $exp_e(i)$ (Def. 2.33) is the first non-linear transformation we consider here.

Definition 3.46 (Exponentiation) *Let $I = (p_0, \ldots, p_n)$ 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 3.47 (Complement) *Let $I = (p_0, \ldots, p_n)$ be an envelope polygon, n a negation function (Def. 2.20). and Δ the threshold.*

$I.Complement(n, \Delta) \stackrel{\text{def}}{=} I.UnaryTransformation(n, \Delta)$. ∎

3.2.6 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 my 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 $((p_0, q_0), \ldots)$ 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 3.48 (Intersection Points) *Let $I_1 = (p_0, \ldots, p_{n+1})$ and $I_2 = (q_0, \ldots, q_{m+1})$ 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 $((p_0, q_0), \ldots)$.*

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 ≤ 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, roundY(q_j.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, roundY(q_j.LineY(q_{j+1}, x))));

    if(i < n){
        if(j < m){
            if(p_i.intersectsProper(p_{i+1}, q_j, q_{j+1})){
                xint := p_i.intersection(p_{i+1}, q_j, q_{j+1});
                IntP.push_back((xint, roundY(p_i.LineY(p_{i+1}, xint))),
                               (xint, roundY(q_j.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 parameterized with two line segments instead of one.

Definition 3.49 (Binary Transformation) *Let $I_1 = (p_0, \ldots, p_{n_1})$ and $I_2 = (q_0, \ldots, q_{n_2})$ be envelope polygons. Let f be a binary y-function and Δ a threshold value.*
We describe the method $I_1.BinaryTransformation(I_2, f, \Delta)$ procedurally:
Let $I \stackrel{\text{def}}{=} ((p_0, q_0), \ldots) = I_1.IntersectionPoints(I_2)$
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 $newI \stackrel{\text{def}}{=} (p_0.x, f(p_0.y, q_0.y))$;
For$_{i=1}^{n}$ $newI.Interpolate(p_i.x, \lambda(x) f(p_{i-1}.y.LineY((p_i.y, x), q_{i-1}.x.LineY((q_i.x, x)), \Delta)$;
return $newI$; ∎

3.2.7 Interval-Interval Relations

Most formulae for interval-interval relations contain integrals over products of membership functions. The corresponding algorithms are listed in this section. All other algorithms for interval-interval relations have already been explained or they are trivial.

Definition 3.50 (Integration over Multiplied Intervals) *Let $I = (p_0, \ldots, p_n)$ and $J = (q_0, \ldots, q_m)$ be envelope polygons. The integral $I.Integrate(J, a) \stackrel{\text{def}}{=} \int_{-\infty}^{+\infty} I(x-a)J(x)\,dx$ 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 $p'_{i+1} \stackrel{\text{def}}{=} (p_{i+1}.x + a, p_{i+1}.y)$
$x = \min(p_{i+1}.x + a, q_{j+1}.x)$
$if(x = p_{i+1}.x + a)\ i = i + 1;$
$if(x = q_{j+1}.x)\ j = j + 1;$

$if(p_n.x + a \leq q_m.x) Int = Int + p_n.y \cdot J.Size2(p_n.x + a, q_m.x)/2$
$else \qquad Int = Int + q_m.y \cdot I.Size2(q_m.x - a, p_n.x)/2$
$return\ Int;$

The next method calls Integrate and normalizes the result.

$$I.IntegrateAsymmetric(J) \stackrel{\text{def}}{=} roundY(\frac{2 \cdot I.Integrate(J, 0)}{I.Size2()})$$

∎

Definition 3.51 (Symmetric Integration) *Let I and J be two envelope polygons.*
The function $I.IntegrateSymmetric(J, simple)$ computes $\int_{-\infty}^{+\infty} I(x) \cdot J(x)\,dx/N$
where $N = \begin{cases} \min(|I|, |J|) & \text{if } simple = true \\ \max_a(\int_{-\infty}^{+\infty} I(x-a) \cdot J(x)\,dx) & \text{otherwise} \end{cases}$

$I.IntegrateSymmetric(J, simple)$
$\stackrel{\text{def}}{=} \begin{cases} undefined & \text{if } I.isInfinite() \text{ or } J.isInfinite() \\ roundY(\frac{2 \cdot I.Integrate(J, 0)}{\min(I.Size2(), J.Size2)}) & \text{if } simple = true \\ roundY(\frac{\top \cdot I.Integrate(J, 0)}{maximizeOverlap(I, J)} & \text{otherwise (Def. 2.70)} \end{cases}$

∎

4 The FuTIRe Interface

The FuTIRe interface consists of the classes Point, Interval, YFunction, and lots of subclasses of the class Operation. For each of these 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 (for example long long 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. FuTIRe 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.

4.1 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 $p \to q \to r$ is a left turn or colinear. (Def. 3.4)

`bool p.leftturnProper(Point q, Point r)`
: true if $p \to q \to r$ is a proper left turn.

`bool p.rightturn(Point q, Point r)`
: true if $p \to q \to r$ is a right turn or colinear.

`bool p.rightturnProper(Point q, Point r)`
: true if $p \to q \to r$ is a proper right turn.

`bool p.colinear(Point q, Point r)`
: true if $p \to q \to r$ is colinear (Def. 3.3)

`bool p.colinear(Point q, Point r, Point s)`
: true if the line segment (p, q) is colinear 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 d)`
: 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 x_1 until x_2. It throws an error of 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)

4.2 Intervals

The Intervals class manages an 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(vector<Point> points)`
: constructs an interval with a vector of points.

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

`void close()`
: declares the polygon as finished and removes some redundancies. (Def. 3.19)

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.

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

`bool I.isMonotone()`
returns true if the polygon is monotone. (Def. 2.68)

`bool I.isSymmetric()`
returns true if the polygon is symmetric. (Def. 2.68)

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

`int I.Index(CX cx)`
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.Max()`
returns the hight $I\hat{}$ of the polygon. (Def. 3.27)

`CX I.MaxX(bool front)`
returns the x-coordinate of the leftmost / rightmost point with maximal y-value. It throws an error if the polygon is empty (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 * the area below the polygon from vertex k to vertex l. (Def. 3.29)

`CX I.Size2()`
returns 2 * the area below the polygon. (Def. 3.29)

`CX I.Size2(CX a, CX b)`
returns 2* the area below the polygon from x-coordinate a to x-coordinate b. (Def. 3.29)

`CX I.CenterPoint(int k, int m)`
returns the x-coordinates of the k, m-center point. (Def. 3.30)

`int I.nComponents()`
returns the number of components of the interval. (Def. 3.32)

`Interval I.Component(int k)`
returns the k^{th} component of I. It throws an error if $k < 0$. (Def. 3.33)

The Core of the Interval (Def. 3.34)

`CX I.CSize()`
returns the size of the core.

`vector<<int,int>> I.CList()`
returns the list of indices of the core boundaries.

`Interval I.CCrisp()`
returns the core as crisp interval.

`CX I.CFirst()`
returns the x-coordinate of the leftmost point of the core.

`CX I.CLast()`
returns the x-coordinate of the rightmost point of the core.

The Support of the Interval (Def. 3.35)

`CX I.SSize()`
: returns the size of the support.

`vector<<int,int>> I.SList()`
: returns the list of indices of the support boundaries.

`Interval I.SCrisp()`
: returns the support as crisp interval.

`CX I.SFirst()`
: returns the x-coordinate of the leftmost point of the support. It throws an error if the polygon is empty.

`CX I.SLast()`
: returns the x-coordinate of the rightmost point of the support. It throws an error if the polygon is empty.

The Kernel of the Interval (Def. 3.36)

`CX I.KSize()`
: returns the size of the kernel.

`vector<<int,int>> I.KList()`
: returns the list of indices of the kernel boundaries.

`Interval I.KCrisp()`
: returns the kernel as crisp interval.

`CX I.KFirst()`
: returns the x-coordinate of the leftmost point of the kernel. It throws an error if the polygon is empty.

`CX I.KLast()`
: returns the x-coordinate of the rightmost point of the kernel. It throws an error if the polygon is empty.

Hull Calculations

`Interval I.CrispHull()`
: returns the crisp hull of I. (Def. 3.37)

`Interval I.MonotoneHull()`
: returns the monotone hull of I. (Def. 3.38)

`Interval I.ConvexHull()`
: returns the convex hull of I. (Def. 3.39)

Basic Unary Transformations (Def. 2.33)

`Interval I.Extend(true)`
: returns the rising part of I. (Def. 3.40)

`Interval I.Extend(false)`
: returns the falling part of I. (Def. 3.40)

`Interval I.ScaleUp()`
: scales the y-values of the interval up to $\top$. (Def. 3.40)

`Interval I.ScaleUpD()`
: is the destructive version of ScaleUp.

`Interval I.Shift(CX a)`
: shifts the interval by a units. (Def. 3.40)

`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. 3.41)

`Interval I.CutI(int i1, int i2)`
: cuts the part of the interval between the points with index $i1$ and $i2$. (Def. 3.41)

`Interval I.Times(float a)`
: multiplies the y-values of the interval by a. (Def. 3.42)

`Interval I.Exp(float e)`
exponentiates the y-values of the interval with e. (Def. 3.46)

`Interval I.Integrate(true)`
computes $\int_{-\infty}^{x} I(y)dy/|I|$. I may be infinite. (Def. 3.44)

`Interval I.Integrate(false)`
computes $\int_{x}^{+\infty} I(y)dy/|I|$. I may be infinite. (Def. 3.44)

`Interval I.Invert()`
inverts the y-values. (Def. 2.33)

`CY I.IntegrateAsymmetric(Interval J)`
computes $\int I(x) \cdot J(x)\ dx/|I|$. I and J may be infinite. (Def. 3.50)

`CY I.IntegrateSymmetric(Interval J,bool simple)`
computes $\int (x) \cdot J(x)\ dx/N(I, J)$. It throws an error if both I and J are infinite. (Def. 3.51)

Fuzzification

`Interval I.FuzzifyLinear(bool front, CX x1, CX x2, CX offset)`
linear fuzzification of the front/end part of the interval with absolute coordinates. (Def. 2.36)

`Interval I.FuzzifyLinear(bool front, float percent, float offset)`
linear fuzzification of the front/end part of the interval with relative coordinates. (Def. 2.39)

`Interval I.FuzzifyGaussian(bool front, CX xh, CX x0, CX offset)`
Gaussian fuzzification of the front/end part of the interval with absolute coordinates. (Def. 2.37)

`Interval I.FuzzifyGaussian(bool front, float percent)`
Gaussian fuzzification of the front/end part of the interval with relative coordinates. (Def. 2.39)

General Transformations

`Interval I.UnaryTransformation(UnaryYFunction f)`
applies the unary y-function f to I. (Def. 3.45)

`Interval I.BinaryTransformation(Interval J, BinaryYFunction f)`
applies the binary y-function f to I and J. (Def. 3.49)

4.3 Y-Functions

The unary and binary transformation methods (Def. 3.45, 3.49) 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. 2.21) $n_\lambda(y) \stackrel{\text{def}}{=} \frac{1-y}{1+\lambda y}$. 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.

FuTIRe realizes y-functions with the class hierarchy in Fig. 1.

The top class, 'Operator', 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 `Operator::getByName(string name)`. The name is optional. Instances without names are not accessible via getByName.

Constructors

`NegationYFunction(string name)`
constructs the standard negation function $\lambda(y)(1 - y)$. (Def. 2.21)

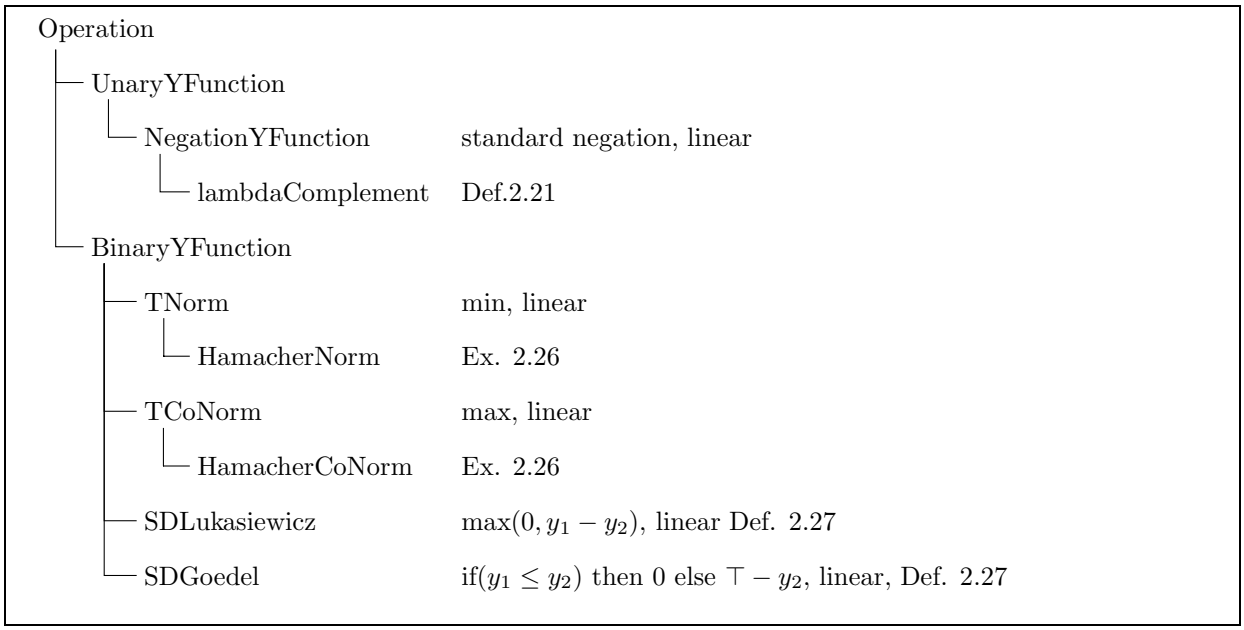


Figure 1: Class Hierarchy for Y-Functions

`lambdaComplement(float lambda, string name)`
constructs the lambda complement $\lambda(y)\frac{1-y}{1+\lambda y}$. (Def. 2.21)

`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. 2.26)

`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. 2.26)

Parameter Modification
The parameters lambda, gamma, beta are public instance variables and can therefore be changed by direct assignment.

4.4 Interval Operators

The interval operators transform one or two given intervals into a new interval. Some of them are implemented as methods of the Interval class. Since methods are in most programming languages not first-class objects, this has some disadvantages. The main disadvantages are that it is not possible to have methods as parameters of other methods, and to construct new methods at run time by composing existing ones. In FuTIRe interval operators are therefore represented as instances of corresponding classes. This way it is no problem to have an interval operator which has another interval operator as parameter, and to combine them in various ways. The class 'IntervalOperator' is a subclass of the class 'Operator' which we have met already. The full class hierarchy is presented in Fig. 2.

Methods
All the classes below have the following methods:

`operator()(Interval I)`
If f is an instance of one of the classes and I an interval then $f(I)$ yields the transformed interval.

`apply(Interval I, CX x, float e)`
If f is an instance of one of the classes below and I and interval then $f.apply(I, x, e)$ yields $f(I)(x)^e$.

Parameters
All input parameters of the constructors below are public instance variables and can be changed by direct assignment.

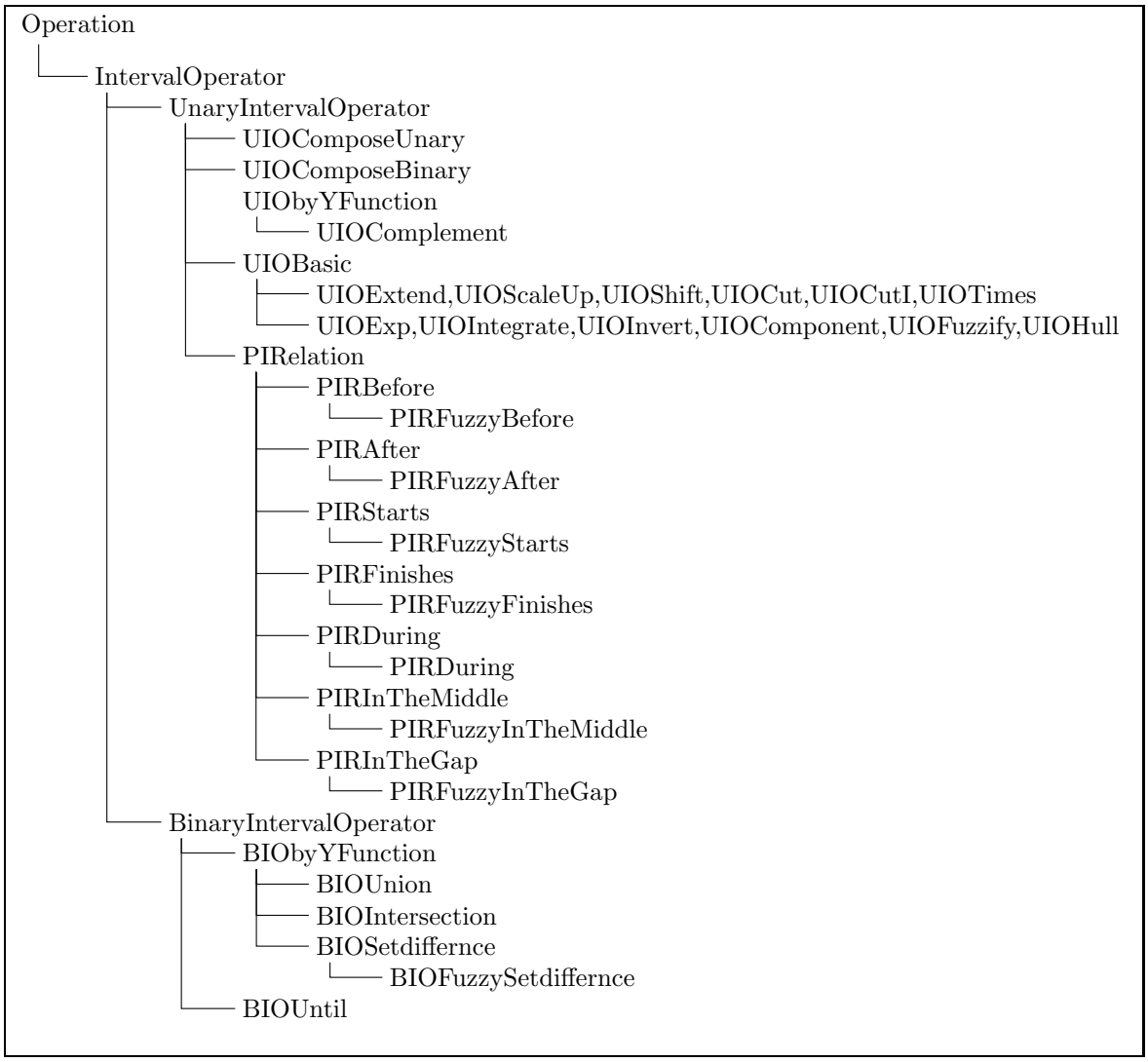


Figure 2: Class Hierarchy for Interval Operators

Constructors
All constructors below have an optional 'string name' argument as last argument.

`UIOComposeUnary(UnaryIntervalOperator U1, UnaryIntervalOperator U2)`
realizes the composition $U1 \circ U2$ of two unary interval operators.

`UIOComposeBinary(UnaryIntervalOperator U1, UnaryIntervalOperator U2, BinaryIntervalOperator B)`
realizes the composition $B(U1(I), U2(I))$ of two unary and one binary interval operator.

`UIObyYFunction(UnaryYFunction F)`
generates an operator which just calls $I.UnaryTransformation(F)$. (Def. 3.45)

`BIObyYFunction(BinaryYFunction F)`
generates an operator which just calls $I.UnaryTransformation(J, F)$. (Def. 3.49)

`UIOComplement()`
Generates the standard Complement Operator $\lambda(y)(1-x)$.

`UIOComplement(float lambda)`
Generates the λ-Complement Operator $\lambda(y)\frac{1-y}{1+\lambda y}$. (Def. 2.21)

`UIOExtend(bool rising)`
generates an operator which returns the rising part of an interval, if $rising = true$, otherwise it generates the falling part (see $I.Extend(rising)$). (Def. 3.40)

`UIOScaleUp()`
generates an operator which scales the y-values of the interval up to $\top$. (Def. 3.40)

`UIOShift(CX a)`
generates an operator which shifts the interval by a units (see $I.Shift(a)$.) (Def. 3.40)

`UIOCut(CX x1, CX x2)`
generates an operator which cuts the part of the interval between $x1$ and $x2$ (see $I.Cut(x1, x2)$). (Def. 3.41)

`UIOCut(CX x1, bool positive)`
generates a cut operator which, if positive = true, works like UIOCut(x1,$+\infty$), otherwise works like UIOCut($-\infty$,x1) (Def. 3.41)

`UIOCutI(int i1, int i2)`
generates an operator which cuts the part of the interval between the points with index $i1$ and $i2$ (see $I.CutI(i1, i2)$. (Def. 3.41)

`UIOTimes(float a)`
generates an operator which multiplies the y-values of the interval by a (see $I.Times(a)$). (Def. 3.42)

`UIOExp(float e)`
generates an operator which exponentiates the y-values of the interval with e (see $I.Exp(e)$). (Def. 3.42)

`UIOIntegrate(bool front)`
generates an operator which integrates the membership function, from $-\infty$, if $front = true$, otherwise from $+\infty$ (see $I.Integrate(front)$). (Def. 3.44)

`UIOInvert()`
generates an operator which inverts the y-values (see $I.Invert()$) (Def. 2.33)

`UIOComponent(int k)`
generates an operator which extracts the k^{th} component from the interval (see $I.Component(k)$.) (Def. 3.33)

`UIOFuzzify(true, bool front, CX x1, CX x2, CX offset)`
generates a linear fuzzify-operator with absolute coordinates (Def. 2.36)

`UIOFuzzify(true, bool front, flaot percent, float offset)`
generates a linear fuzzify-operator with relative coordinates (Def. 2.39)

`UIOFuzzify(false, bool front, CX x1, CX x2, CX offset)`
generates a Gaussian fuzzify-operator with absolute coordinates (Def. 2.37)

`UIOFuzzify(false, bool front, flaot percent)`
generates a Gaussian fuzzify-operator with relative coordinates (Def. 2.39)

The name of the first parameter in the UIOFuzzify-method is 'linear'.

`UIOHull(version)`
generates an operator which generates the corresponding hull. The following versions are avaiable: crisp (Def. 3.37), monotone (Def. 3.38), convex (Def. 3.39), core (Def. 3.34), support (Def. 3.35) and kernel (Def. 3.36).

`BIOUnion()`
generates an operator which computes the union of two fuzzy intervals using the max-norm (Def. 2.24)

`BIOUnion(TCoNorm S)`
generates an operator which computes the union of two fuzzy intervals using the t-conorm S. (Def. 2.24)

`BIOIntersection()`
generates an operator which computes the intersection of two fuzzy intervals using the min-norm (Def. 2.24)

`BIOIntersection(TNorm T)`
generates an operator which computes the union of two fuzzy intervals using the t-norm T. (Def. 2.24)

`BIOSetdifference(Version)`
generates an operator which computes the Kleene, Lukasiewicz, Goedel set difference of two fuzzy sets. (Def. 2.29)

`BIOFuzzySetdifference(BIOIntersection T, BIOComplement C)`
generates an operator which computes the generalized Kleen set difference of two fuzzy sets: $I \setminus J \stackrel{\text{def}}{=} T(I, C(J))$. (Def. 2.27)

`BIOUntil(bool begin, bool end)`
generates an *Until*-operator with $E^+ = UIOExtend(begin)$, $E^- = UIOExtend(end)$, standard intersection and standard complement. (Def. 2.53)

`BIOUntil(bool begin, bool end, UnaryIntervalOperator* E1, UnaryIntervalOperator* E2, BIOIntersection* Ints, UIOComplement* Cmpl)`
generates a general *Until*-operator (Def. 2.53)

Constructors for the Point-Interval Relation Operators
The point-interval relation operators come in two versions, for example PIRBefore and PIRFuzzyBefore. The first version works like the corresponding standard crisp relations. As boundaries of the intervals one can choose either the support, the core or the kernel. PIRFuzzyBefore is the operator version $N(E^+(I))$.

CrVersion is an enumeration type with values (Support,Core,Kernel)

`PIRBefore(CrVersion Version)`
generates a crisp *before*-operator x *before* I iff $x < I^{fV}$ where V is determined by the $Version$ parameter. (Def. 2.42)

`PIRFuzzyBefore()`
generates a *before*-operator $N(E^+(I))$ with $E^+ = UIOExtend(true)$ and standard complement N. (Def. 2.43)

`PIRFuzzyBefore(UnaryIntervalOperator Ep, Complement N)`
generates a *before*-operator $N(Ep(I))$. (Def. 2.43)

`PIRAfter(CrVersion Version)`
generates a crisp *after*-operator x *after* I iff $x > I^{lV}$ where V is determined by the $Version$ parameter. (Def. 2.42)

`PIRFuzzyAfter()`
generates a *after*-operator $N(E^-(I))$ with $E^- = UIOExtend(false)$ and standard complement N. (Def. 2.43)

`PIRFuzzyAfter(UnaryIntervalOperator Em, Complement N)`
generates a *after*-operator $N(Em(I))$. (Def. 2.43)

`PIRStarts(CrVersion Version)`
generates a *starts*-operator x *starts* I iff $x = I^{fV}$ where V is determined by the $Version$ parameter. (Def. 2.42)

`PIRFuzzyStarts()`
generates a *starts*-operator $scaleup(Ep(I)\ T\ B(I)$ where $Ep = UIOExtend(true)$, $T = BIOIntersection()$ and $B = PIRFuzzyBefore()$. (Def. 2.43)

`PIRFuzzyStarts(UnaryIntervalOperator Ep, PIRBefore B, BIOIntersection T)`
generates a *starts*-operator $scaleup(Ep(I)\ T\ B(I)$. (Def. 2.43)

`PIRFinishes(CrVersion Version)`
generates a *finishes*-operator x *finishes* I iff $x = I^{lV}$ where V is determined by the $Version$ parameter. (Def. 2.42)

`PIRFuzzyFinishes()`
generates a *finishes*-operator $scaleup(Em(I)\ T\ A(I)$ where $Em = UIOExtend(false)$, $T = BIOIntersection()$ and $A = PIRAfter()$. (Def. 2.43)

`PIRFuzzyFinishes(UnaryIntervalOperator Em, UIOAfter A, Intersection T)`
generates a *finishes*-operator $scaleup(Ep(I)\ T\ A(I)$. (Def. 2.43)

`PIRDuring(CrVersion Version)`
generates a *during*-operator x *during* I iff $x \in V(I)$ where V is determined by the $Version$ parameter. (Def. 2.42)

`PIRDuring(CrVersion Version, int n, int m)`
generates a $during_{n,m}$-operator x *during* I iff $x \in V(I.cut_{I^{n,m},I^{n+1,m}})$ where V is determined by the $Version$ parameter.

`PIRFuzzyDuring()`
generates the identity *during*-operator.

`PIRFuzzyDuring(UnaryTransformation O)`
generates the *during*-operator $U_{l \in CMP(I)}O(I)$. (Def. 2.43)

`PIRFuzzyDuring(PIRDuring D, int n, int m)`
generates the *during*-operator $D(cut_{i^{n,m},i^{n+1,m}})$. (Def. 2.43)

`PIRInTheMiddle(int n, int m)`
generates an *in_the_middle*-operator $xin\_the\_middle_{n,m}(I)$ iff $x = I^{2n+1,2m}$

`PIRFuzzyTheMiddle(PIRDuring D, int n, int m, int k)`
generates the *in_the_middle*-operator $D(cut_{i^{2^k(2n+1)-1,2^k 2m},i^{2^k(2n+1)+1,2^k 2m}}(I))$. (Def. 2.43)

`PIRInTheGap(CrVersion Version)`
generates an *in_the_gap*-operator $xin\_the\_gap(I)$ iff $x \in V(Invert(I))$ where V is determined by the $Version$ parameter.

`PIRInTheGap(CrVersion Version, int k)`
generates an *in_the_*k^{th}*gap*-operator $xin\_the\_k^{th}gap(I)$ iff $x \in V(Component(Invert(I), k))$ where V is determined by the $Version$ parameter.

`PIRFuzzyInTheGap(PIRDuring D)`
generates an *in_the_gap*-operator $D(invert(I))$ (Def. 2.43)

`PIRFuzzyInTheGap(PIRDuring D, int k)`
generates an *in_the_*k^{th}*gap*-operator $D(Component(invert(I), k))$ (Def. 2.43)

4.5 Interval-Interval Relations

The interval-interval relation operators are also implemented as subclasses of the class Operations. They come in two versions. For example $IIRBefore(Version)$ generates a crisp relation where either the support, the core or the kernel of the two intervals is compared. The subclass $IIRFuzzyBefore(\ldots)$ generates the operator version of the *before*-relation (Def. 2.56).

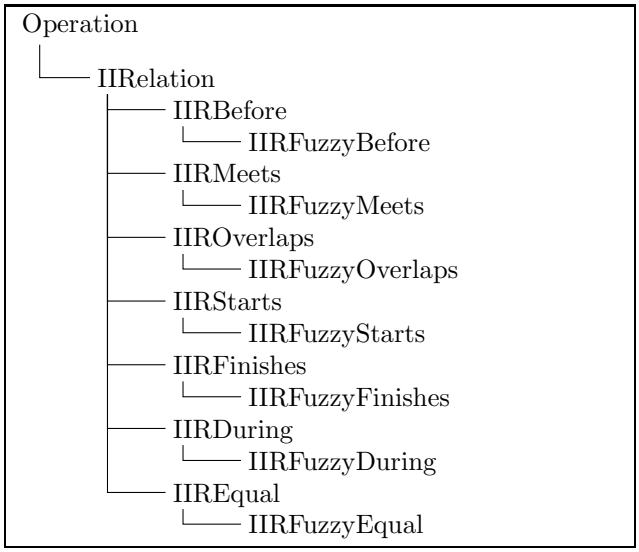


Figure 3: Class Hierarchy for Interval-Interval Relations

Methods

`operator()(Interval I, Interval J, float e=1)`
If for example r is an instance of one of the IIRelation classes and I and J are intervals then $r(I, J)^e$ yields the resulting fuzzy value.

Constructors for the Interval-Interval Relation Operators
All constructors have an optional 'string name' parameter as last argument.

`IIRBefore(CrVersion Version)`
generates a *before*-operator I *before* J iff $I^{lV} < J^{fV}$ where V is determined by the $Version$-parameter. (Def. 2.55)

`IIRFuzzyBefore(PIRBefore B)`
generates a *before*-operator which is essentially $\int I(x) \cdot B(J)(x)\ dx/|i|$ (Def. 2.56)

`IIRFuzzyBefore()`
generates a *before*-operator with $B = newPIRFuzzyBefore()$

`IIRMeets(CrVersion Version)`
generates a *meets*-operator I *meetsJ* iff $I^{lV} = J^{fV}$ where V is determined by the $Version$-parameter. (Def. 2.55)

`IIRFuzzyMeets(PIRFinishes F, PIRStarts S, bool simple)`
generates a *meets*-operator which is essentially $\int F(I)(x) \cdot S(I)(x)\ dx/N(F(I), S(J))$ where $N(I, J) = min(|I|, |J|)$ if $simple = true$ and $N = \max_a \int I(x - a) \cdot J(x)\ dx$ otherwise (Def. 2.56)

`IIRFuzzyMeets(bool simple)`
generates the *meets*-operator with $F = newPIRFinishes()$ and $S = newPIRFuzzyStarts()$

`IIROverlaps(CrVersion Version)`
generates a *overlaps*-operator I *overlaps* J iff some part $I_1 \subseteq V(I)$ is before J^{fV} and for the rest $I_2 = V(I) \setminus I_1$: $I_2 \subseteq V(J)$ holds. V is determined by the $Version$-parameter. (Def. 2.55)

`IIRFuzzyOverlaps(UnaryIntervalOperator* Ep, IIRDuring D, bool simple)`
generates an *overlaps*-operator which is essitially $(1 - D(I, E^+(J))) * D(I, J)$. If simple = false then this result is nomalized with $\max_a(1 - D(I.shiftD(a), E^+(J))) * D(I.shiftD(a), J)$. (Def. 2.56)

`IIRFuzzyOverlaps()`
generates an *overlaps*-operator with $Ep = newUIOExtend(true)$ and $D = newIIRFuzzyDuring()$

`IIRStarts(CrVersion Version)`
generates a *starts*-operator I *starts* J iff $I^{fV} = J^{fV}$ and $V(I) \subseteq V(J)$. V is determined by the $Version$-parameter. (Def. 2.55)

`IIRFuzzyStarts(PIRStarts S1, PIRStarts S2, PIRDuring D, bool simple)`
generates a *starts*-operator which is essentially $\int S_1(I)(x) \cdot S_2(I)(x)\ dx/N(S_1(I), S_2(I)) \cdot D(I, J)$ where $N(I, J) \stackrel{\text{def}}{=} min(|I|, |J|)$ if $simple = true$ and $N(I, J) \stackrel{\text{def}}{=} \max_a \int I(x - a) \cdot J(x)\ dx$ otherwise. (Def. 2.56)

`IIRFuzzyStarts(bool simple)`
generates a *starts*-operator with $S1 = S2 = newPIRFuzzyStarts()$ and $D = newIIRFuzzyDuring()$

`IIRFinishes(CrVersion Version)`
generates a *finishes*-operator I *finishes* J iff $I^{lV} = J^{lV}$ and $V(I) \subseteq V(J)$. V is determined by the $Version$-parameter. (Def. 2.55)

`IIRFuzzyFinishes(PIRFinishes F1, PIRFinishes F2, PIRDuring D, bool simple)`
generates an *finishes*-operator which is essentially $\int F_1(I)(x) \cdot F_2(I)(x)\ dx/N(F_1(I), F_2(I)) \cdot D(I, J)$ where $N(I, J) \stackrel{\text{def}}{=} min(|I|, |J|)$ if $simple = true$ and $N(I, J) \stackrel{\text{def}}{=} \max_a \int I(x - a) \cdot J(x)\ dx$ otherwise. (Def. 2.56)

`IIRFuzzyFinishes(bool simple)`
generates a *finishes*-operator with $F1 = F2 = newPIRFuzzyFinishes()$ and $D = newIIRFuzzyDuring()$

`IIRDuring(CrVersion Version)`
generates a *during*-operator I *during* J iff $V(I) \subseteq V(J)$. V is determined by the $Version$-parameter. (Def. 2.55)

`IIRFuzzyDuring(PIRDuring D)`
generates a *during*-operator which is essentially $\int I(x) \cdot D(J)(x)\ dx|I|$ (Def. 2.56)

`IIRFuzzyDuring()`
generates an *during*-operator with $D = newPIRFuzzyDuring()$.

`IIREquals(CrVersion Version)`
generates a *equals*-operator I *equals* J iff $V(I) = V(J)$. V is determined by the $Version$-parameter. (Def. 2.55)

`IIRFuzzyEquals(IIRDuring D)`
generates an *equals*-operator: $D(i, j) \cdot D(j, i)$ (Def. 2.56)

`IIRFuzzyEquals()`
generates an *equals*-operator with $D = newIIRFuzzyDuring()$.

5 Summary

This report is a detailed description of the FuTIRe-library. So far this library is a C++-package for representing and manipulating fuzzy time intervals and for generating customized point-interval and interval-interval relations for fuzzy intervals (a Java version is planned). The mathematical background, the concrete datastructures and algorithms, and the interface to the library are described.

There are quite different schemes for fuzzy point-interval and interval-interval relations. Besides a crisp approximation for these relations I proposed and implemented in FuTIRe an operator-based scheme which is very flexible and easy to adapt to the needs of concrete applications. The architecture of FuTIRe is open to add other schemes, and to use them together with the crisp and operator-based schemes.

References

[1] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[2] François Bry, Bernhard Lorenz, Hans Jürgen Ohlbach, and Stephanie Spranger. On reasoning on time and location on the web. In N. Henze F. Bry and J. Malusyński, editors, *Principles and Practice of Semantic Web Reasoning*, volume 2901 of *LNCS*, pages 69–83. Springer Verlag, 2003.

[3] Didier Dubois and Henri Prade, editors. *Fundamentals of Fuzzy Sets*. Kluwer Academic Publisher, 2000.

[4] D. M. Gabbay, I. Hodkinson, and M Reynolds, editors. *Temporal Logic: Mathematical Foundations and Computational Aspects*, volume 1. Oxford: Clarendon Press, 1994.

[5] Jacob E. Goodman and Joseph O'Rourke, editors. *Handbook of Discrete and Computational Geometry*. CRC Press, 1997.

[6] Gábor Nagypál and Boris Motik. A fuzzy model for representing uncertain, subjective and vague temporal knowledge in ontologies. In *Proceedings of the International Conference on Ontologies, Databases and Applications of Semantics, (ODBASE)*, volume 2888 of *LNCS*. Springer-Verlag, 2003.

[7] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, 1998.

[8] L. A. Zadeh. Fuzzy sets. *Information & Control*, 8:338–353, 1965.