# A Guide to the FuzzyNumbers 0.03-devel Package for R


Marek Gagolewski[1,2]

[1] Systems Research Institute, Polish Academy of Sciences
ul. Newelska 6, 01-447 Warsaw, Poland

[2] Rexamine, Email: `gagolews@rexamine.com`
www.rexamine.com/resources/fuzzynumbers/


June 2, 2013

*The package, as well as this tutorial, is still in its early days – any suggestions and contributions are welcome!*

---

## Contents

# 1 Getting Started

Fuzzy set theory lets us effectively and quite intuitively represent imprecise or vague information. Fuzzy numbers (FNs), introduced by Dubois and Prade in [7], form a particular subclass of fuzzy sets of the real line. Formally, a fuzzy set $A$ with membership function $\mu_A : \mathbb{R} \to [0,1]$ is a fuzzy number, if it possess at least the three following properties:

(i) it is a normalized fuzzy set, i.e. $\mu_A(x_0) = 1$ for some $x_0 \in \mathbb{R}$,

(ii) it is fuzzy convex, i.e. for any $x_1, x_2 \in \mathbb{R}$ and $\lambda \in [0,1]$ it holds $\mu_A(\lambda x_1 + (1-\lambda)x_2) \geq \mu_A(x_1) \wedge \mu_A(x_2)$,

(iii) the support of $A$ is bounded, where $\mathrm{supp}(A) = \mathrm{cl}(\{x \in \mathbb{R} : \mu_A(x) > 0\})$.

Fuzzy numbers play a significant role in many practical applications (cf. [13]) since we often describe our knowledge about objects through numbers, e.g. "I'm about 180 cm tall" or "The rocket was launched between 2 and 3 p.m.".

FuzzyNumbers is an Open Source (licensed under GNU LGPL 3) package for R– a free, open sourced software environment for statistical computing and graphics, which includes an implementation of a very powerful and quite popular high-level language called S. and runs on all major operating systems, i.e. Windows, Linux, and MacOS X[1].

FuzzyNumbers has been created in order to deal with fuzzy numbers conveniently and effectively. To install latest "official" release of the package available on *CRAN* we type:

```
install.packages('FuzzyNumbers')
```

Alternatively, we may fetch its current development snapshot from *GitHub*.

Each session with FuzzyNumbers should be preceded by a call to:

```
library('FuzzyNumbers') # Load the package
```

To view the main page of the manual we type:

```
library(help='FuzzyNumbers')
```

For more information please visit the package's homepage [9]. In case of any problems, comments, or suggestions feel free to contact the author. Good luck!

[1]To install R and/or find some information on the S language please visit R Project's Homepage at www.R-project.org. Perhaps you may also wish to install RStudio, a convenient development environment for R. It is available at www.rsudio.com/ide.

# 2 How to Create Instances of Fuzzy Numbers

## 2.1 Arbitrary Fuzzy Numbers

A fuzzy number $A$ may be defined by specifying its core, support, and either its left/right side functions or lower/upper $\alpha$-cut bounds. Please note that many algorithms that deal with FNs assume we provide at least the latter, i.e. $\alpha$-cuts.

### 2.1.1 Definition by Side Functions

A fuzzy number $A$ specified by side functions[2] has a membership function of the form:

$$\mu_A(x) = \begin{cases} 0 & \text{if} \quad x < \mathtt{a1}, \\ \mathtt{left}\left(\frac{x-\mathtt{a1}}{\mathtt{a2}-\mathtt{a1}}\right) & \text{if } \mathtt{a1} \le x < \mathtt{a2}, \\ 1 & \text{if } \mathtt{a2} \le x \le \mathtt{a3}, \\ \mathtt{right}\left(\frac{x-\mathtt{a3}}{\mathtt{a4}-\mathtt{a3}}\right) & \text{if } \mathtt{a3} < x \le \mathtt{a4}, \\ 0 & \text{if } \mathtt{a4} < x, \end{cases} \tag{1}$$

where $\mathtt{a1}, \mathtt{a2}, \mathtt{a3}, \mathtt{a4} \in \mathbb{R}$, $\mathtt{a1} \le \mathtt{a2} \le \mathtt{a3} \le \mathtt{a4}$, $\mathtt{left} : [0,1] \to [0,1]$ is a nondecreasing function (called *left side generator of* $A$), and $\mathtt{right} : [0,1] \to [0,1]$ is a nonincreasing function (*right side generator of* $A$). In our package, it is assumed that these functions fulfill the conditions $\mathtt{left}(0) \ge 0$, $\mathtt{left}(1) \le 1$, $\mathtt{right}(0) \le 1$, and $\mathtt{right}(1) \ge 0$.

Please note that by using side generating functions defined on $[0,1]$ we really make (in author's humble opinion) the process of generating examples for our publications much easier. A similar concept was used e.g. in [14] (LR-fuzzy numbers).

An example: a fuzzy number $A_1$ with linear sides (a trapezoidal fuzzy number, see also Sec. 2.4).

```
A1 <- FuzzyNumber(1, 2, 4, 7,
    left=function(x) x,
   right=function(x) 1-x
)
```

This object is an instance of the following R class:

```
class(A1)

## [1] "FuzzyNumber"
## attr(,"package")
## [1] "FuzzyNumbers"
```

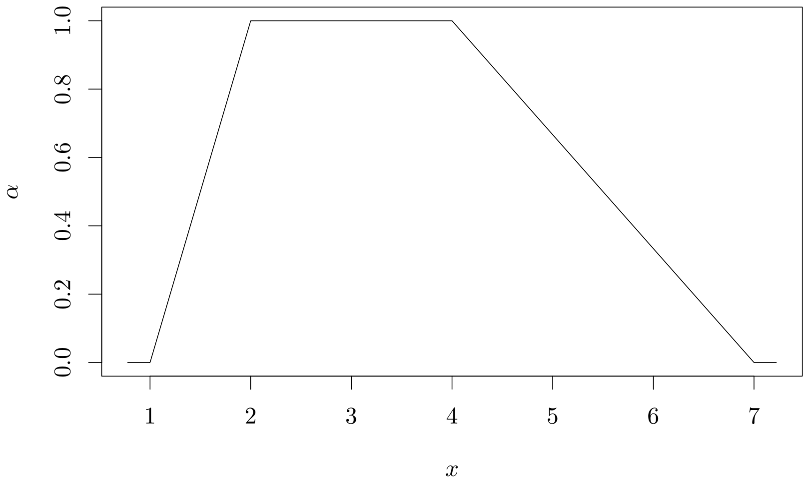
We may print some basic information on $A_1$ by calling `print(A1)` or simply by typing:

```
A1

## Fuzzy number with:
##    support=[1,7],
##       core=[2,4].
```

To depict $A_1$ we call:

```
plot(A1)
```

[2]Side functions are sometimes called branches or shape functions in the literature.




If we would like to generate figures for our publications, then we will be interested in storing them as PDF files. This may be done by calling:

```
pdf('figure1.pdf', width=8, height=5) # create file
plot(A1)
dev.off() # close graphical device and save the file
```

Postscript (PS) files are generated by substituting the call to `pdf()` for the call to the `postcript()` function.

**Remark.** Assume we are given two fancy side functions $f : [a_1, a_2] = [-4, -2] \to [0, 1]$, and $g : [a_3, a_4] = [-1, 10] \to [1, 0]$, for example:

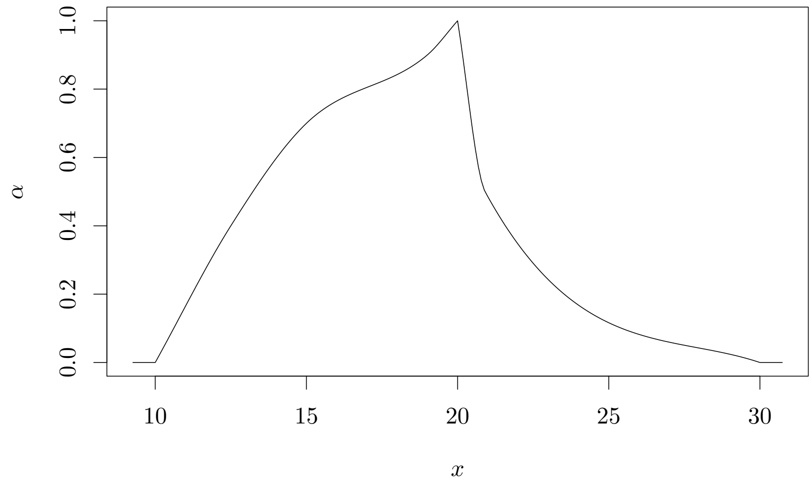
```
f <- splinefun(c(-4,-3.5,-3,-2.2,-2), c(0,0.4,0.7,0.9,1), method='monoH.FC')
g <- splinefun(c(-1,0,10), c(1,0.5,0), method='monoH.FC')
```

Let us convert them to side *generating* functions, which shall be defined on the interval $[0, 1]$. This may easily be done with the `convert.side()` function. It returns a new function that calls the original one with linearly transformed input.

```
convert.side(f, -4, -2)(c(0,1))
## [1] 0 1
convert.side(g, -1, 10)(c(0,1))
## [1] 1 0
convert.side(g, 10, -1)(c(0,1)) # interesting!
## [1] 0 1
```

These functions may be used to define a fuzzy number, now with arbitrary support and core.

```
B <- FuzzyNumber(10,20,20,30,
    left=convert.side(f, -4, -2),
   right=convert.side(g, -1, 10)
)
plot(B, xlab='$x$', ylab='$\\alpha$')
```




### 2.1.2 Definition by $\alpha$-cut Bounds

Alternatively, a fuzzy number $A$ may be defined by specifying its $\alpha$-cuts. We have (for $\alpha \in (0,1)$ and $\mathtt{a1} \le \mathtt{a2} \le \mathtt{a3} \le \mathtt{a4}$):

$$
\begin{aligned}
A_\alpha &:= [A_L(\alpha), A_U(\alpha)] &(2)\\
&= \big[\mathtt{a1} + (\mathtt{a2} - \mathtt{a1}) \cdot \mathtt{lower}(\alpha), \mathtt{a3} + (\mathtt{a4} - \mathtt{a3}) \cdot \mathtt{upper}(\alpha)\big], &(3)
\end{aligned}
$$

where $\mathtt{lower} : [0,1] \to [0,1]$ is a nondecreasing function (called *lower $\alpha$-cut bound generator of $A$*), and $\mathtt{upper} : [0,1] \to [0,1]$ is a nonincreasing function (*upper bound generator*). In our package, we assumed that $\mathtt{lower}(0) = 0$, $\mathtt{lower}(1) = 1$, $\mathtt{upper}(0) = 1$, and $\mathtt{upper}(1) = 0$.

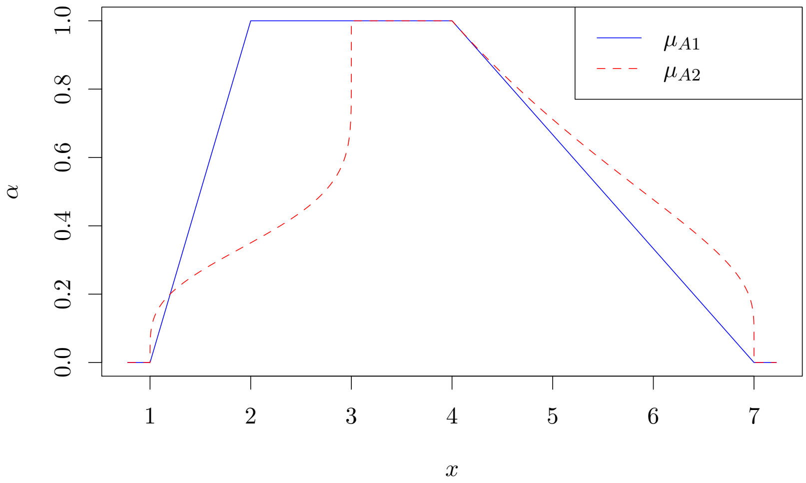
It is easily seen that for $\alpha \in (0,1)$ we have the following relationship between generating functions:

$$
\begin{aligned}
\mathtt{lower}(\alpha) &= \inf\{x : \mathtt{left}(x) \ge \alpha\}, &(4)\\
\mathtt{upper}(\alpha) &= \sup\{x : \mathtt{right}(x) \ge \alpha\}. &(5)
\end{aligned}
$$

Moreover, if side generating functions are continuous and strictly monotonic, then $\alpha$-cut bound generators are their inverses.

An example:

```
A1 <- FuzzyNumber(1, 2, 4, 7,
    left=function(x) x,
   right=function(x) 1-x
)
A2 <- FuzzyNumber(1, 3, 4, 7,
   lower=function(alpha) pbeta(alpha, 5, 9), # CDF of a beta distr.
   upper=function(alpha) pexp(1/alpha-1) # transformed CDF of an exp. distr.
)
plot(A1, col='blue')
plot(A2, col='red', lty=2, add=TRUE)
legend('topright', c(expression(mu[A1]), expression(mu[A2])),
   col=c('blue', 'red'), lty=c(1,2))
```




**Remark.** The **convert.alpha()** function works similarly to **convert.side()**. This tool, however, scales the output values of a given function, thus it may be used to create an alpha-cut generator conveniently.

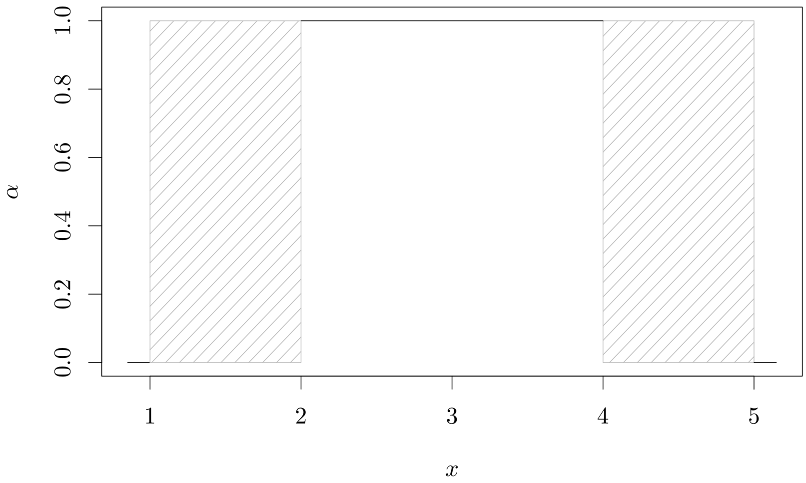
### 2.1.3 Definition with Generating Functions Omitted: Shadowed Sets

Please note that in the above examples we passed to the constructor of each `FuzzyNumber` class instance either side generating functions or $\alpha$-cut generators. Let us study what happens, if we omit both of them.

```
A3 <- FuzzyNumber(1, 2, 4, 5)
A3

## Fuzzy number with:
##    support=[1,5],
##       core=[2,4].
```

The object seems to be defined correctly: R does not make any complaints. However…

```
plot(A3)
```




It turns out that we have obtained a *shadowed set*! Indeed, this behavior is quite reasonable: we have provided no information on the "partial knowledge" part of our fuzzy number. In fact, the object has been initialized with generating functions always returning `NA` (*Not-Available* or *any* value). Does it mean that when we define a FN solely by side generators, we cannot compute its $\alpha$-cuts? Indeed!

```
alphacut(A2, 0.5) # A2 has alpha-cut generators defined
## [1] 2.733154 5.896362
alphacut(A1, 0.5) # A1 hasn't got them
## [1] NA NA
```

Another example: evaluation of the membership function.

```
evaluate(A1, 6.5) # A1 has side generators defined
## [1] 0.1666667
evaluate(A2, 6.5) # A2 hasn't got them
## [1] NA
```

### 2.2 Using Numeric Approximations of $\alpha$-cut or Side Generators

The reason for setting by default `NA`s[3] as return values of generators (when omitted) is simple. Finding a function inverse numerically requires lengthy computations and is always done locally (for a given point, not for "whole" the function at once). R is not a symbolic mathematical solver. If we had defined such procedures (it is really easy to do by using the `uniroot()` function), then an inexperienced user would have used it in his/her algorithms and wondered why everything runs so slow. To get more insight, let us look at the internals of `A2`:

```
A2['lower']
## function(alpha) pbeta(alpha, 5, 9)
A2['upper']
## function(alpha) pexp(1/alpha-1)
```

[3]To be precise, it's `NA_real_`.

```
A2['left']
## function (x)
## rep(NA_real_, length(x))
## <environment: 0x39bcc80>
A2['right']
## function (x)
## rep(NA_real_, length(x))
## <environment: 0x39bcc80>
```

Note that all generators are properly vectorized (for input vectors of length $n$ they always give output of the same length). Thus, general rules are as follows. If you want $\alpha$-cuts (e.g. for finding trapezoidal approximations of FNs), specify them. If you would like to access side functions (by the way, the `plot()` function automatically detects what kind of knowledge we have), assure they are provided.

However, we we provide some convenient short-cut methods to *interpolate* generating functions of one type to get some crude numeric approximations of their inverses. These are simple wrappers to R's `approxfun()` (piecewise linear interpolation, the `'linear'` `method`) and `splinefun()` (monotonic splines: `method`s `'hyman'` and `'monoH.FC'`; the latter is default and recommended). They are available as the `approx.invert()` function[4], and may of course be used on results returned by `convert.alpha()` and `convert.side()`.

```
l <- function(x) pbeta(x, 1, 2)
r <- function(x) 1-pbeta(x, 1, 0.1)
A4 <- FuzzyNumber(-2, 0, 0, 2,
   left  = l,
   right = r,
   lower = approx.invert(l),
   upper = approx.invert(r)
)

x <- seq(0,1,length.out=1e5)
max(abs(qbeta(x, 1, 2) - A4['lower'](x)))     # sup-error
## [1] 0.0001389811
max(abs(qbeta(1-x, 1, 0.1) - A4['upper'](x))) # sup-error
## [1] 0.0008607773
```

### 2.3 Fuzzy Numbers with Discontinuities

... TO BE DONE ....

```
A1 <- FuzzyNumber(0,1,1,1,
         lower=function(a) floor(3*a)/3,
         upper=function(a) 1-a
) # no info on discontinuities

A2 <- DiscontinuousFuzzyNumber(0,1,1,1,
         lower=function(a) floor(3*a)/3,
         upper=function(a) 1-a,
```

[4]The `n` argument, which sets the number of interpolation points, controls the trade-off between accuracy and computation speed. Well, world's not ideal, remember that "any" is better than "nothing" sometimes.

```
        discontinuities.lower=c(0, 1/3, 2/3, 1),
        discontinuities.upper=numeric(0)
) # discontinuities info included
```

... TO BE DONE ....

### 2.4 Trapezoidal Fuzzy Numbers

A trapezoidal fuzzy number (TFN) is a FN which has linear side generators and linear $\alpha$-cut bound generators. To create a trapezoidal fuzzy number $T_1$ with, for example, $\mathrm{core}(T_1) = [1.5, 4]$ and $\mathrm{supp}(T_1) = [1, 7]$ we call:

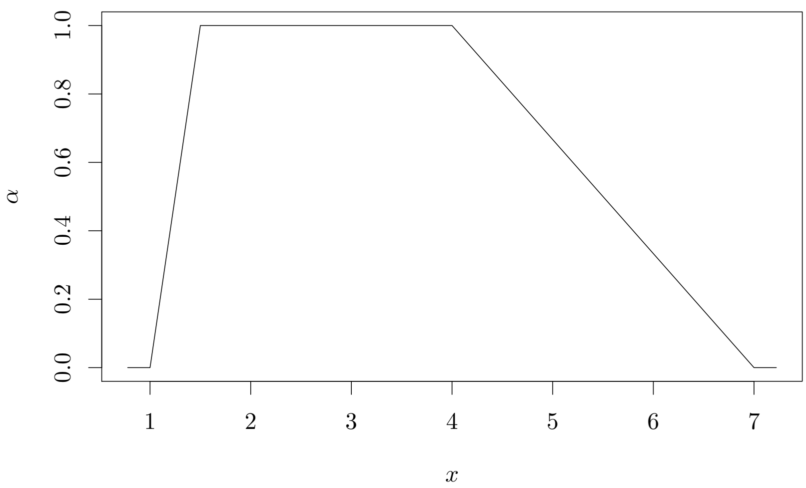
```
T1 <- TrapezoidalFuzzyNumber(1,1.5,4,7)
```

This object is an instance of the following R class:

```
class(T1)

## [1] "TrapezoidalFuzzyNumber"
## attr(,"package")
## [1] "FuzzyNumbers"
```

To depict $T_1$ we call:

```
plot(T1)
```




$T_1$ is (roughly) equivalent to the trapezoidal fuzzy number $A_1$ defined in the previous subsection. The `TrapezoidalFuzzyNumber` class inherits all the goodies from the `FuzzyNumber` class, but is more specific (guarantees faster computations, contains more detailed information, etc.). Of course, in this case the generating functions are known *a priori* ($A_1$ had no $\alpha$-cut generators) so there is no need to provide them manually (what is more, this has been disallowed for safety reasons). Thus, is we wanted to define a trapezoidal FN next time, we would rather not do it like with $A_1$ but as with $T_1$.

```
T1['lower']

## function (alpha)
## alpha
```

```
## <bytecode: 0x31a88d8>
## <environment: namespace:FuzzyNumbers>
T1['upper']
## function (alpha)
## 1 - alpha
## <bytecode: 0x31a87f8>
## <environment: namespace:FuzzyNumbers>
T1['left']
## function (x)
## x
## <bytecode: 0x31a8b40>
## <environment: namespace:FuzzyNumbers>
T1['right']
## function (x)
## 1 - x
## <bytecode: 0x31a8a60>
## <environment: namespace:FuzzyNumbers>
```

Thus, we have:

$$\mu_{T_1}(x) = \begin{cases} 0 & \text{for} \quad x \in (-\infty, 1), \\ (x-1)/0.5 & \text{for} \quad x \in [1, 1.5), \\ 1 & \text{for} \quad x \in [1.5, 4], \\ (7-x)/3 & \text{for} \quad x \in (4, 7], \\ 0 & \text{for} \quad x \in (7, +\infty). \end{cases}$$

$$T_{1\alpha} = [1 + 0.5\,\alpha, 7 - 3\,\alpha].$$

Note that the above equations have been atomatically generated by knitr and LaTeX by calling `cat(as.character(T1, toLaTeX=TRUE, varnameLaTeX='T_1'))`, see Sec. 3.

Trapezoidal fuzzy numbers are among the simplest FNs. Despite their simplicity, however, they include triangular FNs, “crisp” real intervals, and “crisp” reals. Please note that currently no separate classes for these particular TFNs types are implemented in the package.

```
TrapezoidalFuzzyNumber(1,2,2,3)  # triangular FN
## Trapezoidal fuzzy number with:
##    support=[1,3],
##       core=[2,2].
TrapezoidalFuzzyNumber(2,2,3,3)  # `crisp' interval
## Trapezoidal fuzzy number with:
##    support=[2,3],
##       core=[2,3].
TrapezoidalFuzzyNumber(5,5,5,5)  # `crisp' real
## Trapezoidal fuzzy number with:
##    support=[5,5],
##       core=[5,5].
```

## 2.5 Piecewise Linear Fuzzy Numbers

Trapezoidal fuzzy numbers are generalized by piecewise linear FNs (PLFNs), i.e. fuzzy numbers which side generating functions and $\alpha$-cut generators are piecewise linear functions. Each PLFN is given by:

- four coefficients $\mathtt{a1} \le \mathtt{a2} \le \mathtt{a3} \le \mathtt{a4}$ defining its support and core,
- the number of "knots", `knot.n`$\ge 0$,
- a vector of $\alpha$-cut coordinates, `knot.alpha`, consisting of `knot.n` elements $\in [0, 1]$,
- a nondecreasingly sorted vector `knot.left` consisting of `knot.n` elements $\in [\mathtt{a1}, \mathtt{a2}]$, defining interpolation points for the left side function, and
- a nondecreasingly sorted vector `knot.right` consisting of `knot.n` elements $\in [\mathtt{a2}, \mathtt{a3}]$, defining interpolation points for the right side function.

If `knot.n`$\ge 1$, then the membership function of a piecewise linear fuzzy number $P$ is defined as:

$$\mu_P(x) = \begin{cases} 0 & \text{if} \quad x < \mathtt{a1}, \\ \alpha_i + (\alpha_{i+1} - \alpha_i)\left(\frac{x-l_i}{l_{i+1}-l_i}\right) & \text{if} \ \ l_i \le x < l_{i+1} \\ & \ \text{for some } i \in \{1, \dots, n+1\}, \\ 1 & \text{if } \mathtt{a2} \le x \le \mathtt{a3}, \\ \alpha_{n-i+2} + (\alpha_{n-i+3} - \alpha_{n-i+2})\left(1 - \frac{x-r_i}{r_{i+1}-r_i}\right) & \text{if } \ r_i < x \le r_{i+1} \\ & \ \text{for some } i \in \{1, \dots, n+1\}, \\ 0 & \text{if } \mathtt{a4} < x, \end{cases} \tag{6}$$

and its $\alpha$-cuts for $\alpha \in [\alpha_i, \alpha_{i+1}]$ (for some $i \in \{1, \dots, n+1\}$) are given by:

$$P_L(\alpha) = l_i + (l_{i+1} - l_i)\left(\frac{\alpha - \alpha_i}{\alpha_{i+1} - \alpha_i}\right), \tag{7}$$

$$P_U,(\alpha) = r_{n-i+2} + (r_{n-i+3} - r_{n-i+2})\left(1 - \frac{\alpha - \alpha_i}{\alpha_{i+1} - \alpha_i}\right), \tag{8}$$

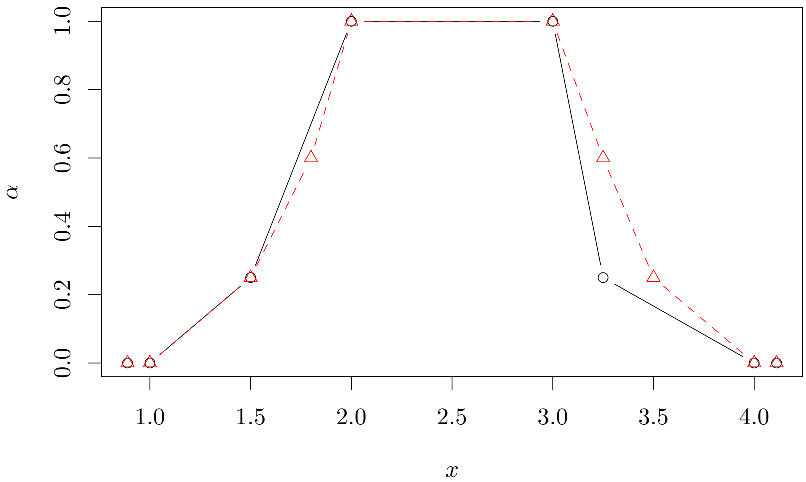
where $n = \mathtt{knot.n}$, $(l_1, \dots, l_{n+2}) = (\mathtt{a1}, \mathtt{knot.left}, \mathtt{a2})$, $(r_1, \dots, r_{n+2}) = (\mathtt{a3}, \mathtt{knot.right}, \mathtt{a4})$, and $(\alpha_1, \dots, \alpha_{n+2}) = (0, \mathtt{knot.alpha}, 1)$.

PLFNs in our package are represented by the `PiecewiseLinearFuzzyNumber` class.

```
P1 <- PiecewiseLinearFuzzyNumber(1, 2, 3, 4,
   knot.n=1, knot.alpha=0.25, knot.left=1.5, knot.right=3.25)
class(P1)

## [1] "PiecewiseLinearFuzzyNumber"
## attr(,"package")
## [1] "FuzzyNumbers"

P1

## Piecewise linear fuzzy number with 1 knot(s),
##    support=[1,4],
##       core=[2,3].

P2 <- PiecewiseLinearFuzzyNumber(1, 2, 3, 4,
   knot.n=2, knot.alpha=c(0.25,0.6),
   knot.left=c(1.5,1.8), knot.right=c(3.25, 3.5))
P2
```

```
## Piecewise linear fuzzy number with 2 knot(s),
##     support=[1,4],
##        core=[2,3].
```

```r
plot(P1, type='b')
plot(P2, type='b', col=2, lty=2, pch=2, add=TRUE)
```




The following operators return matrices with all knots of a PLFN. Each of them have three columns, in order: $\alpha$-cuts, left side coordinates, and right side coordinates.

```r
P1['knots']
```

```
##        alpha left right
## knot_1  0.25  1.5  3.25
```

```r
P1['allknots'] # including a1,a2,a3,a4
```

```
##        alpha left right
## supp    0.00  1.0  4.00
## knot_1  0.25  1.5  3.25
## core    1.00  2.0  3.00
```

We have, for example:

$$\mu_{P_1}(x) = \begin{cases} 0 & \text{for} \quad x \in (-\infty, 1), \\ 0 + 0.25\,(x+1)/0.5 & \text{for} \quad x \in [1, 1.5), \\ 0.25 + 0.75\,(x+1.5)/0.5 & \text{for} \quad x \in [1.5, 2), \\ 1 & \text{for} \quad x \in [2, 3], \\ 0.25 + 0.75\,(3.25-x)/0.25 & \text{for} \quad x \in [3, 3.25), \\ 0 + 0.25\,(4-x)/0.75 & \text{for} \quad x \in [3.25, 4), \\ 0 & \text{for} \quad x \in (4, +\infty). \end{cases}$$

$$P_{1\alpha} = [P_{1L}(\alpha), P_{1U}(\alpha)],$$

where

$$P_{1L}(\alpha) = \begin{cases} 1 + 0.5\,(\alpha - 0)/0.25 & \text{for} \quad \alpha \in [0, 0.25], \\ 1.5 + 0.5\,(\alpha - 0.25)/0.75 & \text{for} \quad \alpha \in [0.25, 1], \end{cases}$$

$$P_{1U}(\alpha) = \begin{cases} 3.25 + 0.75\,(0.25 - \alpha)/0.25 & \text{for} \quad \alpha \in [0, 0.25], \\ 3 + 0.25\,(1 - \alpha)/0.75 & \text{for} \quad \alpha \in [0.25, 1]. \end{cases}$$

If you want to obtain a PLFN with equally distributed knots, then you may use the more convenient version of the `PiecewiseLinearFuzzyNumber()` function.

```
PiecewiseLinearFuzzyNumber(knot.left=c(0,0.5,0.7,1),
                           knot.right=c(2,2.2,2.4,3))['allknots']
```

```
##              alpha left right
## supp    0.0000000  0.0   3.0
## knot_1  0.3333333  0.5   2.4
## knot_2  0.6666667  0.7   2.2
## core    1.0000000  1.0   2.0
```

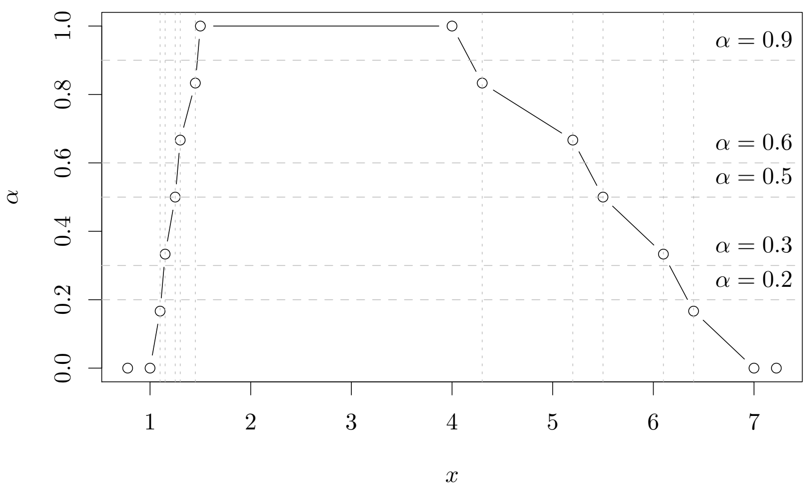
Note that if $a_1, \ldots, a_4$ are omitted, then they are taken from `knot.left` and `knot.right` (their lengths should then be equal to `knot.n`+2).

If `knot.n` is equal to 0 or all left and right knots lie on common lines, then a PLFN reduces to a TFN. Please note that, however, the `TrapezoidalFuzzyNumber` class does not inherit from `PiecewiseLinearFuzzyNumber` for efficiency reasons. If, however, we wanted to convert an object of the first mentioned class to the other, we would do that by calling:

```
alpha <- c(0.2, 0.3, 0.5, 0.6, 0.9)
P3 <- as.PiecewiseLinearFuzzyNumber(T1,
         knot.n=5, knot.alpha=alpha)
P3
```

```
## Piecewise linear fuzzy number with 5 knot(s),
##    support=[1,7],
##       core=[1.5,4].
```

```
plot(P3)
abline(h=alpha, col='gray', lty=2)
abline(v=P3['knot.left'], col='gray', lty=3)
abline(v=P3['knot.right'], col='gray', lty=3)
text(7, alpha, sprintf('a=%g', alpha), pos=3)
```



More generally, each PLFN or TFN may be converted to a direct `FuzzyNumber` class instance if needed (hope we will never not).

```
(as.FuzzyNumber(P3))

## Fuzzy number with:
##    support=[1,7],
##       core=[1.5,4].
```

On the other hand, to "convert" (with possible information loss) more general FNs to TFNs or PLFNs, we may use the approximation procedures described in Sec. 6.

### 2.6 Fuzzy Numbers with Sides Given by Power Functions

Fuzzy numbers which sides are given by power functions are defined with four coefficients $\mathtt{a1} \le \mathtt{a2} \le \mathtt{a3} \le \mathtt{a4}$, and parameters $\mathtt{p.left}, \mathtt{p.right} > 0$ which determine exponets for the side functions:

$$
\begin{aligned}
\mathtt{left}(x) &= x^{\mathtt{p.left}}, && (9)\\
\mathtt{right}(x) &= (1-x)^{\mathtt{p.right}}. && (10)
\end{aligned}
$$

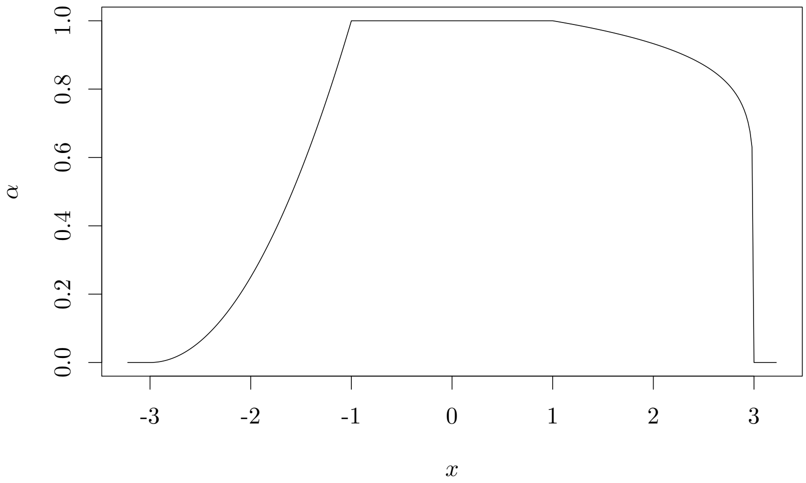
We also have:

$$
\begin{aligned}
\mathtt{lower}(\alpha) &= \sqrt[\mathrm{p.left}]{\alpha}, && (11)\\
\mathtt{upper}(\alpha) &= 1-\sqrt[\mathrm{p.right}]{\alpha}. && (12)
\end{aligned}
$$

These fuzzy numbers are another natural generalization of trapezoidal FNs.

An example:

```
X <- PowerFuzzyNumber(-3, -1, 1, 3, p.left=2, p.right=0.1)
class(X)

## [1] "PowerFuzzyNumber"
## attr(,"package")
## [1] "FuzzyNumbers"

X

## Fuzzy number given by power functions, and:
##    support=[-3,3],
##       core=[-1,1].

plot(X)
```




We have:

$$\mu_X(x) = \begin{cases} 0 & \text{for} \quad x \in (-\infty, -3), \\ ((x+3)/2)^2 & \text{for} \quad x \in [-3, -1), \\ 1 & \text{for} \quad x \in [-1, 1], \\ ((3-x)/2)^{0.1} & \text{for} \quad x \in (1, 3], \\ 0 & \text{for} \quad x \in (3, +\infty), \end{cases}$$

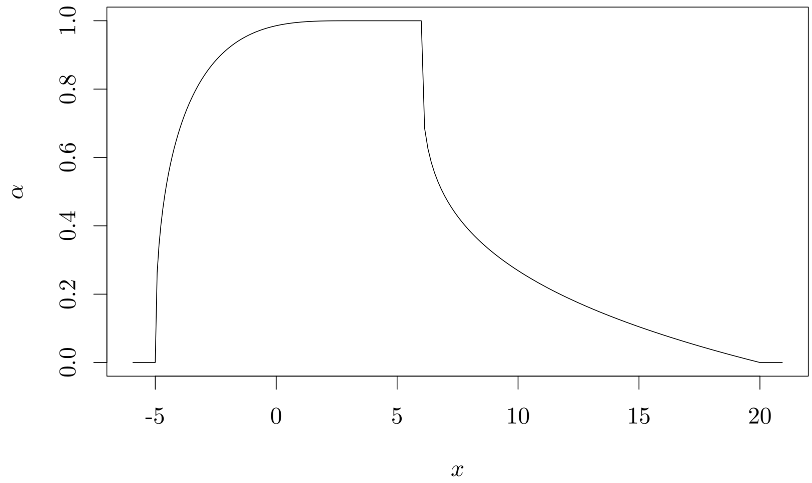
$$X_\alpha = [-3 + 2\,\alpha^{0.5}, 1 + 2\,(1 - \alpha^{10})].$$

# 3 Depicting Fuzzy Numbers

To depict FNs we use the `plot()` method, which uses similar parameters as the R-built-in `curve()` function. If you are new to R, you may wish to read the manual on the most popular graphical routines by calling `?plot`, `?plot.default`, `?curve`, `?abline`, `?par`, `?lines`, `?points`, `?legend`, `?text` (some of these functions have already been called in this tutorial).

Let us consider the following FN:

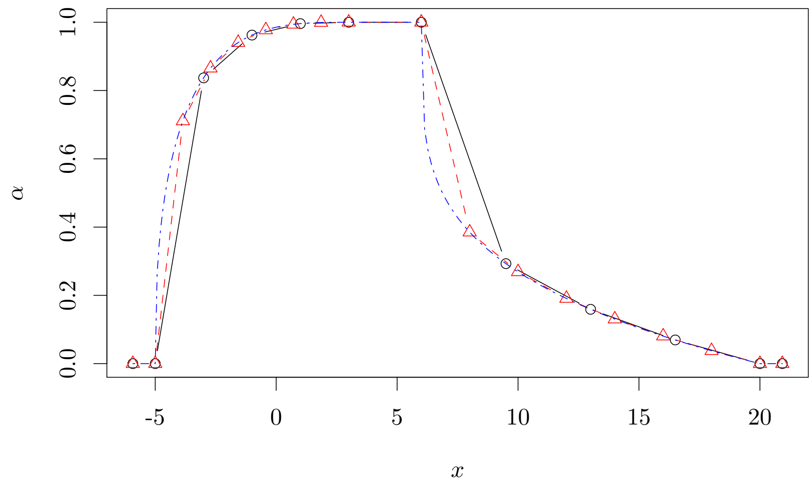
```
A <- FuzzyNumber(-5, 3, 6, 20,
    left=function(x) pbeta(x,0.4,3),
   right=function(x) 1-x^(1/4),
   lower=function(alpha) qbeta(alpha,0.4,3),
   upper=function(alpha) (1-alpha)^4
)
plot(A)
```




**Plotting issues: discretization.** Side functions or $\alpha$-cut bounds of objects of the `FuzzyNumber` class (not including its derivatives) when plotted are naively approximated by piecewise linear functions with equidistant knots at one of the axes. Therefore, if we probe them at too few points, we may obtain very rough graphical representations. To control the number of points at which the interpolation takes place, we use the `n` parameter (which defaults to `101` = quite accurate).

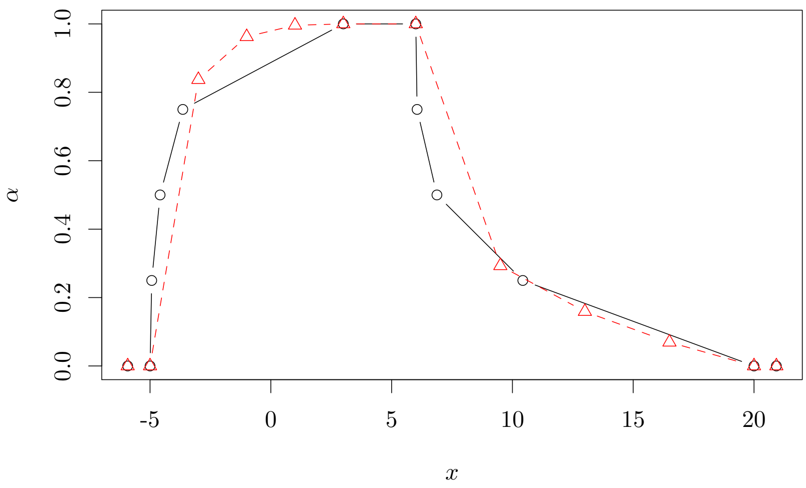
All three calls to the `plot()` method below depict the membership function of the same fuzzy number, but with different accuracy.

```
plot(A, n=3, type='b')
plot(A, n=6, add=TRUE,  lty=2, col=2, type='b', pch=2)
plot(A, n=101, add=TRUE, lty=4, col=4) # default n
```



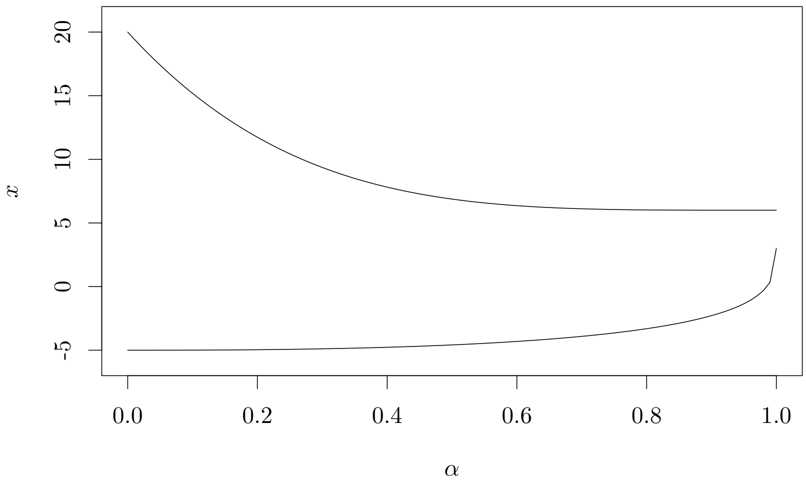

**Making use of different generating functions' types.** Please note (if you have not already) that to draw the membership function we do not need to provide necessarily the FN with side generators: the $\alpha$-cuts will also suffice. The function is smart enough to detect the internal representation of the FN and use the kind representation it has. It both types of generators are given then side functions are used. If we want to, for some reasons, use $\alpha$-cuts, then we may do as follows:

```
plot(A, n=3, at.alpha=numeric(0), type='b') # use alpha-cuts
plot(A, n=3, type='b', col=2, lty=2, pch=2, add=TRUE) # use sides
```




We may also illustrate an $\alpha$-cut representation of a fuzzy number:

```
plot(A, draw.alphacuts=TRUE)
```




**Conversion to LaTeX.** Another way to depict a FN is to... give a mathematical expression which defines it.

```
cat(as.character(A, toLaTeX=TRUE, varnameLaTeX='A'))
```

This gives the following LaTeX code...

```
\[
\mu_{A}(x) = \left\{
\begin{array}{lll}
0       & \text{for} & x\in(-\infty,-5), \\
l_{A}(x) & \text{for} & x\in[-5,3), \\
1       & \text{for} & x\in[3,6], \\
r_{A}(x) & \text{for} & x\in(6,20], \\
0       & \text{for} & x\in(20,+\infty), \\
\end{array}
\right.
\]
where $l_{A}=\mathtt{left}_A((x+5)/8)$,
$r_{A}=\mathtt{right}_A((x-6)/14)$.

\[
{A}_\alpha = [{A}_L(\alpha), {A}_U(\alpha)],
\]
where ${A}_L(\alpha)=-5+8\,\mathtt{lower}_{A}(\alpha)$,
${A}_U(\alpha)=6+14\,\mathtt{upper}_{A}(\alpha)$.
```

. . . and, after compiling:

$$\mu_{A}(x) = \left\{
\begin{array}{lll}
0       & \text{for} & x\in(-\infty,-5), \\
l_{A}(x) & \text{for} & x\in[-5,3), \\
1       & \text{for} & x\in[3,6], \\
r_{A}(x) & \text{for} & x\in(6,20], \\
0       & \text{for} & x\in(20,+\infty), \\
\end{array}
\right.$$

where $l_{A}=\mathtt{left}_A((x+5)/8)$, $r_{A}=\mathtt{right}_A((x-6)/14)$.

$$A_\alpha = [A_L(\alpha), A_U(\alpha)],$$

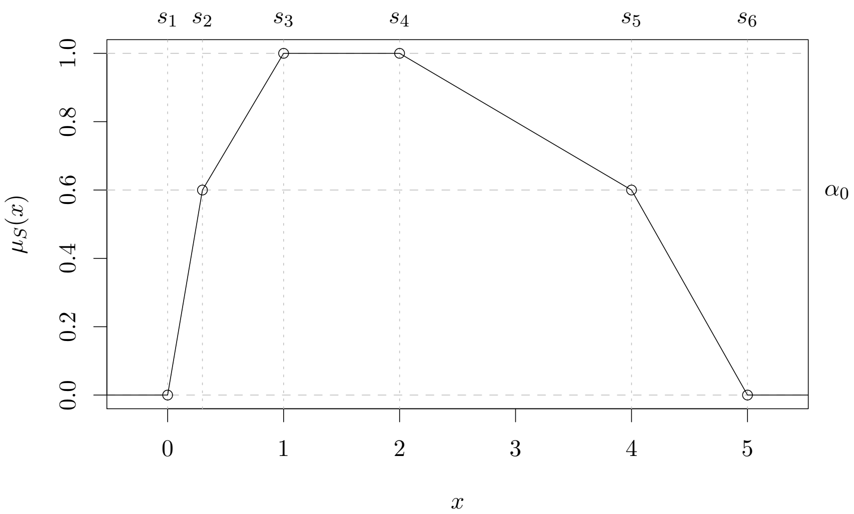
where $A_L(\alpha)=-5+8\,\mathtt{lower}_{A}(\alpha)$, $A_U(\alpha)=6+14\,\mathtt{upper}_{A}(\alpha)$.

The code may of course be modified to suit your needs.

**Tuning your figures.** Finally, we leave you with a quite complex example from one of our papers:

```
X <- PiecewiseLinearFuzzyNumber(0, 1, 2, 5, knot.n=1,
   knot.alpha=0.6, knot.left=0.3, knot.right=4)

plot.default(NA, xlab='$x$', ylab='$\\mu_S(x)$',
   xlim=c(-0.3,5.3), ylim=c(0,1)) # empty window

xpos <- c(X['a1'], X['knot.left'],  X['a2'],
          X['a3'], X['knot.right'], X['a4'])
xlab <- c('$s_1$', '$s_2$', '$s_3$',
          '$s_4$', '$s_5$', '$s_6$')
```

```
abline(v=xpos, col='gray', lty=3)
text(xpos, 1.05, xlab, pos=3, xpd=TRUE)

abline(h=c(0, X['knot.alpha'], 1), col='gray', lty=2)
text(5.55, X['knot.alpha'], sprintf('$\\alpha_0$'), pos=4, xpd=TRUE)

plot(X, add=TRUE, type='l', from=-1, to=6)
plot(X, add=TRUE, type='p', from=-1, to=6)
```




Please note that we use TEX commands in plot labels. They are interpreted by the `tikzDevice` package for R to generate beautiful figures, but setting this all up requires higher level of skills. . . and patience.

# 4 Basic Computations on and Characteristics of Fuzzy Numbers

In this section we consider the following FN:

```
A <- FuzzyNumber(-5, 3, 6, 20,
    left=function(x) pbeta(x,0.4,3),
   right=function(x) 1-x^(1/4),
   lower=function(alpha) qbeta(alpha,0.4,3),
   upper=function(alpha) (1-alpha)^4
)
```

## 4.1 Support and Core, and Other $\alpha$-cuts

The support of $A$, i.e. $\mathrm{supp}(A) = [\texttt{a1}, \texttt{a4}]$, may be obtained by calling:

```
supp(A)
## [1] -5 20
```

We get the core of $A$, i.e. $\mathrm{core}(A) = [\texttt{a2}, \texttt{a3}]$, with:

```
core(A)
## [1] 3 6
```

To compute arbitrary $\alpha$-cuts we use:

```
alphacut(A, 0) # same as supp(A) (if alpha-cut generators are defined)
## [1] -5 20
alphacut(A, 1) # same as core(A)
## [1] 3 6
alphacut(A, c(0,0.5,1))
##           [,1]   [,2]
## [1,] -5.000000 20.000
## [2,] -4.583591  6.875
## [3,]  3.000000  6.000
```

Note that if we request to compute more than one $\alpha$-cut at once, then a matrix with 2 columns (instead of a numeric vector of length 2) is returned. The `alphacut()` method may only be used when $\alpha$-cut generators are provided by the user during the declaration of $A$, even for $\alpha = 0$ or $\alpha = 1$.

### 4.2 Evaluation of the Membership Function

If side generators are defined, we may calculate the values of the membership function at different points by calling:

```
evaluate(A, 1)
## [1] 0.9960291
evaluate(A, c(-3,0,3))
## [1] 0.8371139 0.9855322 1.0000000
evaluate(A, seq(-1, 2, by=0.5))
## [1] 0.9624800 0.9760168 0.9855322 0.9919531 0.9960291 0.9983815 0.9995357
```

### 4.3 "Typical" Value

Let us first introduce the notion of the *expected interval* of $A$ [8].

$$\begin{aligned} \mathrm{EI}(A) &:= [\mathrm{EI}_L(A), \mathrm{EI}_U(A)] \qquad (13)\\ &= \left[\int_0^1 A_L(\alpha)\,d\alpha, \int_0^1 A_U(\alpha)\,d\alpha\right]. \qquad (14)\end{aligned}$$

To compute the expected interval of $A$ we call:

```
expectedInterval(A)
## [1] -4.058824  8.800000
```

Please note that in case of objects of the `FuzzyNumber` class the expected interval is approximated by numerical integration. This method calls the `integrate()` function and its accuracy (quite fine by default) may be controlled by the `subdivisions`, `rel.tol`, and `abs.tol` parameters (call `?integrate` for more details). On the other hand, for TFNs and PLFs this method returns exact results.

The midpoint of the expected interval is called the *expected value* of a fuzzy number. It is given by:

$$\mathrm{EV}(A) := \frac{\mathrm{EI}_L(A) + \mathrm{EI}_U(A)}{2}. \qquad (15)$$

Let us calculate $\mathrm{EV}(A)$.

```
expectedValue(A)
## [1] 2.370588
```

Note that this method uses a call to `expectedInterval(A)`, thus in case of `FuzzyNumber` class instances it also uses numerical approximation.

Sometimes a generalization of the expected value, called *weighted expected value*, is useful. For given $w \in [0,1]$ it is defined as:

$$\mathrm{EV}_w(A) := (1-w)\mathrm{EI}_L(A) + w\mathrm{EI}_U(A). \tag{16}$$

It is easily seen that $\mathrm{EV}_{0.5}(A) = \mathrm{EV}(A)$.
Some examples:

```
weightedExpectedValue(A, 0.5) # equivalent to expectedValue(A)
## [1] 2.370588
weightedExpectedValue(A, 0.25)
## [1] -0.8441176
```

The *value* of $A$ [5] is defined by:

$$\mathrm{val}(A) := \int_0^1 \alpha \left(A_L(\alpha) + A_U(\alpha)\right) \, d\alpha. \tag{17}$$

It may be calculated by calling:

```
value(A)
## [1] 1.736177
```

Please note that the expected value or value may be used for example to "defuzzify" $A$.

## 4.4 Measures of "Nonspecificity"

The *width* of $A$ [3] is defined as:

$$\mathrm{width}(A) := \mathrm{EI}_U(A) - \mathrm{EI}_L(A). \tag{18}$$

An example:

```
width(A)
## [1] 12.85882
```

The *ambiguity* of $A$ [5] is defined as:

$$\mathrm{amb}(A) := \int_0^1 \alpha \left(A_U(\alpha) - A_L(\alpha)\right) \, d\alpha. \tag{19}$$

```
ambiguity(A)
## [1] 5.197157
```

Additionally, to express "nonspecificity" of a fuzzy number we may use e.g. the width of its support:

```r
diff(supp(A))
## [1] 25
```

# 5 Operations on Fuzzy Numbers

## 5.1 Arithmetic Operations

The basic binary arithmetic operations for FNs are often defined by means of the so-called extension principle (see [13]) and interval arithmetic. For each $\alpha \in [0,1]$:

$$(A \circledast B)_\alpha = A_\alpha \circledast B_\alpha,$$

where $\circledast = +, -, *$ or $/$, and $A, B$ are arbitrary FNs.

For example, we define the sum $A + B$ for every $\alpha \in [0,1]$ as:

$$(A+B)_\alpha = A_\alpha + B_\alpha = \left[A_L(\alpha) + B_L(\alpha), A_U(\alpha) + B_U(\alpha)\right],$$

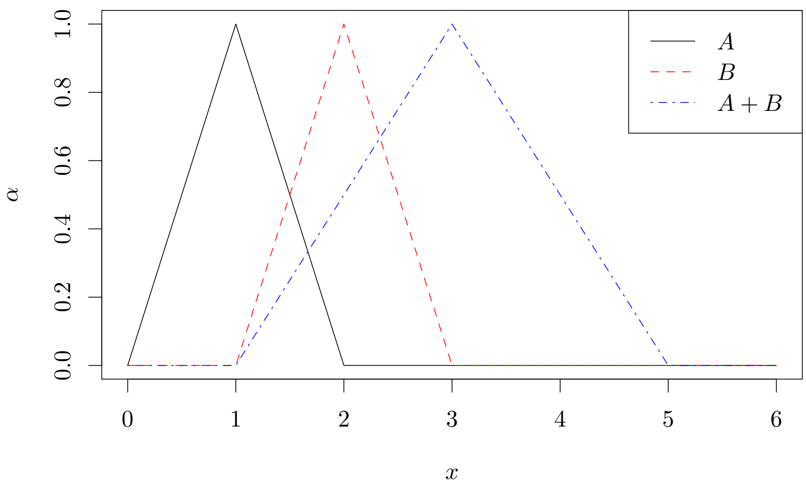
see [7, 6]. Moreover, for $\lambda \in \mathbb{R}$, the scalar multiplication is given by:

$$(\lambda \cdot A)_\alpha = \lambda A_\alpha = \begin{cases} \left[\lambda A_L(\alpha), \lambda A_U(\alpha)\right], & \text{if } \lambda \geq 0, \\ \left[\lambda A_U(\alpha), \lambda A_L(\alpha)\right], & \text{if } \lambda < 0, \end{cases}$$

for each $\alpha \in [0,1]$.

In the FuzzyNumbers package we have defined the +, -, * and / operators, which implements the basic arithmetic operations as defined in [13].

```r
A <- TrapezoidalFuzzyNumber(0, 1, 1, 2)
B <- TrapezoidalFuzzyNumber(1, 2, 2, 3)
plot(A, xlim=c(0,6), xlab='$x$', ylab='$\\alpha$')
plot(B, add=TRUE, col=2, lty=2)
plot(A+B, add=TRUE, col=4, lty=4)
legend('topright', c('$A$', '$B$', '$A+B$'), lty=c(1,2,4), col=c(1,2,4))
```




Currently all the operations are available for piecewise linear FNs only, and addition and scalar multiplication is additionally implemented for trapezoidal FNs. Note that the computer arithmetic has anyway a discrete nature, and a PLFN with large number of knots often approximates (cf. Sec. 6) an arbitrary FN sufficiently well. The computations are always exact (well, up to the computer floating-point arithmetic errors) at knots.

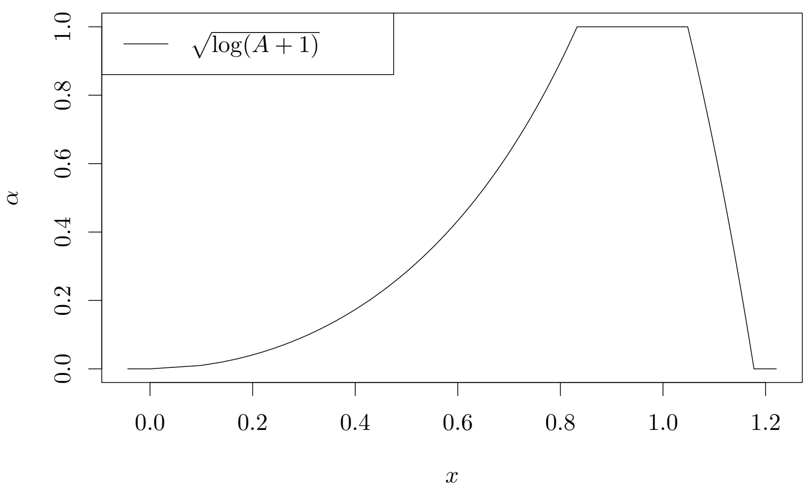
In theory the class of PLFNs is not closed under the operations `*` and `/`. However, if you operate on a large number of knots, the results should be satisfactory.

```
A <- piecewiseLinearApproximation(PowerFuzzyNumber(1,2,3,4,p.left=2,p.right=0.5),
   method="Naive", knot.n=20)
B <- piecewiseLinearApproximation(PowerFuzzyNumber(2,3,4,5,p.left=0.1,p.right=3),
   method="Naive", knot.n=40)
A+A # the same as 2*A

## Piecewise linear fuzzy number with 20 knot(s),
##    support=[2,8],
##       core=[4,6].

A+B # note the number of knots has increased

## Piecewise linear fuzzy number with 60 knot(s),
##    support=[3,9],
##       core=[5,7].
```

## 5.2 Applying Functions

To apply a monotonic transformation on a piecewise linear fuzzy number (using the extension principle) we call **`fapply()`**.

```
A <- as.PiecewiseLinearFuzzyNumber(TrapezoidalFuzzyNumber(0,1,2,3), knot.n=100)
plot(fapply(A, function(x) log(x+1)^0.5), xlab='$x$', ylab='$\\alpha$')
legend('topleft', '$\\sqrt{\\log(A+1)}$', lty=1)
```




The operation being applied should be a properly vectorized R function object.

# 6 Approximation of Fuzzy Numbers

## 6.1 Metrics in the Space of Fuzzy Numbers

It seems that the most suitable metric for approximation problems is an extension of the Euclidean ($L_2$) distance (cf. [10]), $d$, defined by the equation:

$$d^2(A, B) = \int_0^1 (A_L(\alpha) - B_L(\alpha))^2 \, d\alpha + \int_0^1 (A_U(\alpha) - B_U(\alpha))^2 \, d\alpha. \tag{20}$$

```
T1 <- TrapezoidalFuzzyNumber(-5, 3, 6, 20)
T2 <- TrapezoidalFuzzyNumber(-4, 4, 7, 21)
distance(T1, T2, type='Euclidean') # L2 distance /default/

## [1] 1.414214

distance(T1, T2, type='EuclideanSquared') # Squared L2 distance

## [1] 2
```

Types available (argument `type`): `Euclidean` (default), `EuclideanSquared`...

## 6.2 Approximation by Trapezoidal Fuzzy Numbers

TO BE DONE... Problem statement... Given a fuzzy number $A$ we seek for a trapezoidal fuzzy number $\mathcal{T}(A)$....

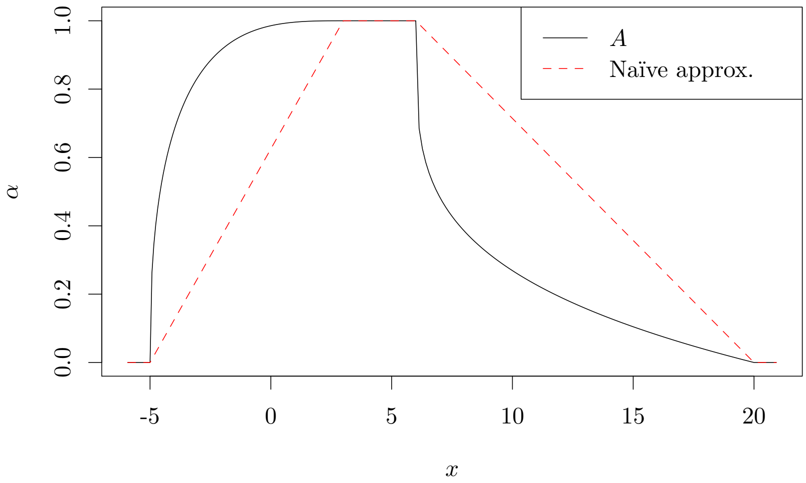
In this subsection we will use the following fuzzy number $A$ for the sake of illustration:

```
A <- FuzzyNumber(-5, 3, 6, 20,
   left=function(x) pbeta(x,0.4,3),
   right=function(x) 1-x^(1/4),
   lower=function(alpha) qbeta(alpha,0.4,3),
   upper=function(alpha) (1-alpha)^4
)
```

### 6.2.1 Naíve Approximation

The `'Naive'` method generates a trapezoidal FN with the same core and support as $A$.

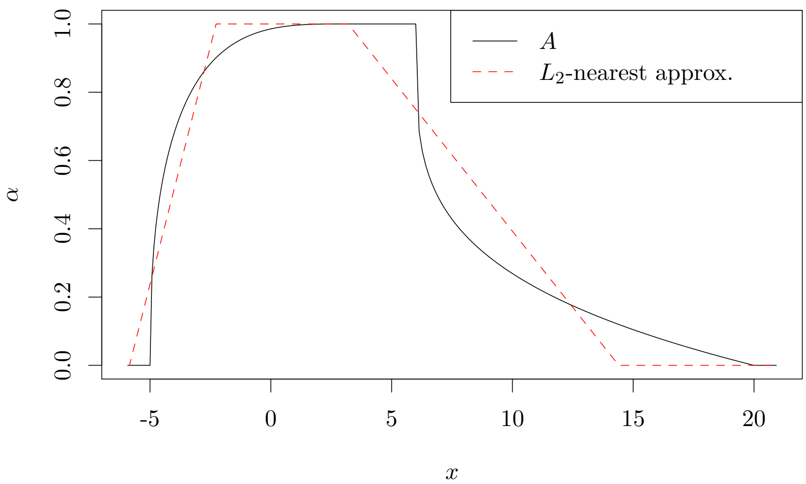
```
(T1 <- trapezoidalApproximation(A, method='Naive'))

## Trapezoidal fuzzy number with:
##    support=[-5,20],
##       core=[3,6].

distance(A, T1)

## [1] 5.761482
```




### 6.2.2 $L_2$-nearest Approximation

The 'NearestEuclidean' method gives the nearest $L_2$-approximation of $A$ [2, Corollary 8].

```
(T2 <- trapezoidalApproximation(A, method='NearestEuclidean'))
## Trapezoidal fuzzy number with:
##    support=[-5.85235,14.4],
##       core=[-2.26529,3.2].
distance(A, T2)
## [1] 1.98043
```



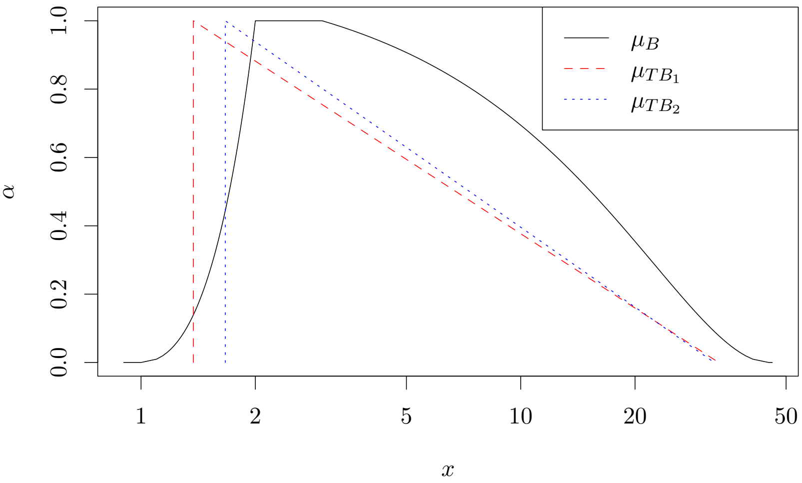

### 6.2.3 Expected Interval Preserving Approximation

The 'ExpectedIntervalPreserving' method gives the nearest $L_2$-approximation of $A$ preserving the expected interval [1, 11, 15]. Note that if $\mathrm{amb}(A) \geq \mathrm{width}(A)/3$, then we get the same result as in the 'NearestEuclidean' method.

```
(T3 <- trapezoidalApproximation(A, method='ExpectedIntervalPreserving'))

## Trapezoidal fuzzy number with:
##    support=[-5.85235,14.4],
##       core=[-2.26529,3.2].

distance(A, T3)

## [1] 1.98043

expectedInterval(A)

## [1] -4.058824  8.800000

expectedInterval(T3)

## [1] -4.058824  8.800000
```

Unfortunately, for highly skewed membership functions this method (as well as the previous one) reveals sometimes quite unfavorable behavior. E.g. if $B$ is a FN such that $\mathrm{Val}(B) < \mathrm{EV}_{1/3}(B)$ or $\mathrm{Val}(B) > \mathrm{EV}_{2/3}(B)$, then it may happen that the core of the output and the core of the original fuzzy number $B$ are disjoint, cf. [12].

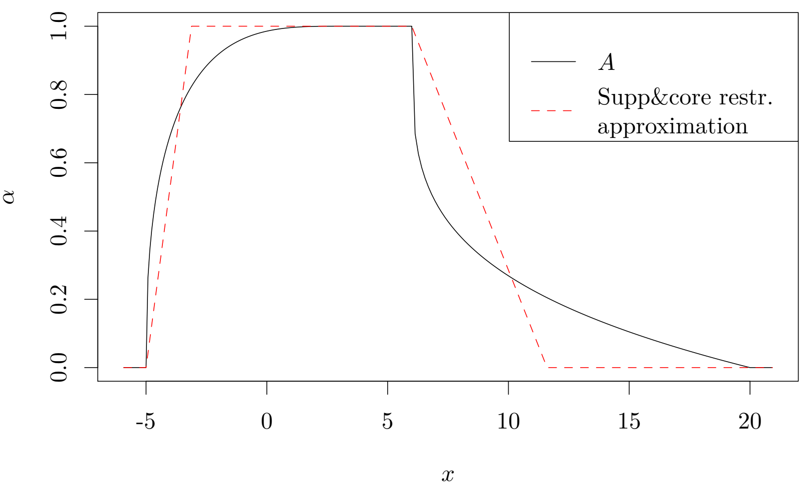
```
(B  <- FuzzyNumber(1, 2, 3, 45,
   lower=function(x) sqrt(x),
   upper=function(x) 1-sqrt(x)))

## Fuzzy number with:
##    support=[1,45],
##       core=[2,3].

(TB1 <- trapezoidalApproximation(B, 'NearestEuclidean'))

## Trapezoidal fuzzy number with:
##    support=[1.37333,33.2133],
##       core=[1.37333,1.37333].

(TB2 <- trapezoidalApproximation(B, 'ExpectedIntervalPreserving'))

## Trapezoidal fuzzy number with:
##    support=[1.66667,32.3333],
##       core=[1.66667,1.66667].

distance(B, TB1)

## [1] 2.098994

distance(B, TB2)

## [1] 2.166239
```




### 6.2.4 Approximation with Restrictions on Support and Core

The `'SupportCoreRestricted'` method was proposed in [12]. It gives the $L_2$-nearest trapezoidal approximation with constraints $\mathrm{core}(A) \subseteq \mathrm{core}(\mathcal{T}(A))$ and $\mathrm{supp}(\mathcal{T}(A)) \subseteq \mathrm{supp}(A)$, i.e. for which each point that surely belongs to $A$ also belongs to $\mathcal{T}(A)$, and each point that surely does not belong to $A$ also does not belong to $\mathcal{T}(A)$.

```
(T4 <- trapezoidalApproximation(A, method='SupportCoreRestricted'))

## Trapezoidal fuzzy number with:
##    support=[-5,11.6],
##       core=[-3.11765,6].

distance(A, T4)

## [1] 2.603383
```



## 6.3 Approximation by Piecewise Linear Fuzzy Numbers

TO BE DONE... Problem statement... Given a fuzzy number $A$ we seek for a piecewise linear fuzzy number $\mathcal{P}(A)$....

Currently only fixed `knot.alpha` allowed.... TO BE DONE....

In this subsection we will use the following fuzzy number $A$ for the sake of illustration:

```
A <- FuzzyNumber(-5, 3, 6, 20,
   left=function(x) pbeta(x,0.4,3),
   right=function(x) 1-x^(1/4),
   lower=function(alpha) qbeta(alpha,0.4,3),
   upper=function(alpha) (1-alpha)^4
)
```

### 6.3.1 Naíve Approximation

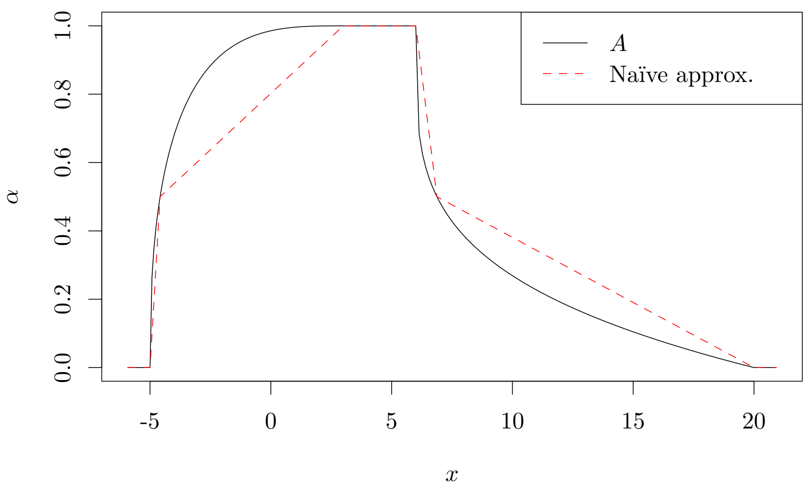
The `'Naive'` method generates a PLFN with the same core and support as $A$ and with sides interpolating the membership function of $A$ at given $\alpha$-cuts.

```
P1 <- piecewiseLinearApproximation(A, method='Naive',
         knot.n=1, knot.alpha=0.5)
P1['allknots']
```

```
##        alpha      left  right
## supp     0.0 -5.000000 20.000
## knot_1   0.5 -4.583591  6.875
## core     1.0  3.000000  6.000
```

```
print(distance(A, P1), 8)
```

```
## [1] 2.4753305
```




### 6.3.2 $L_2$-nearest Approximation

**Exact algorithm for fixed `knot.alpha`.** TO BE DONE...

For `knot.n==1` the method proposed in [4] is used.

```
system.time(P2 <- piecewiseLinearApproximation(A,
   method='NearestEuclidean', knot.n=1, knot.alpha=0.5))
```

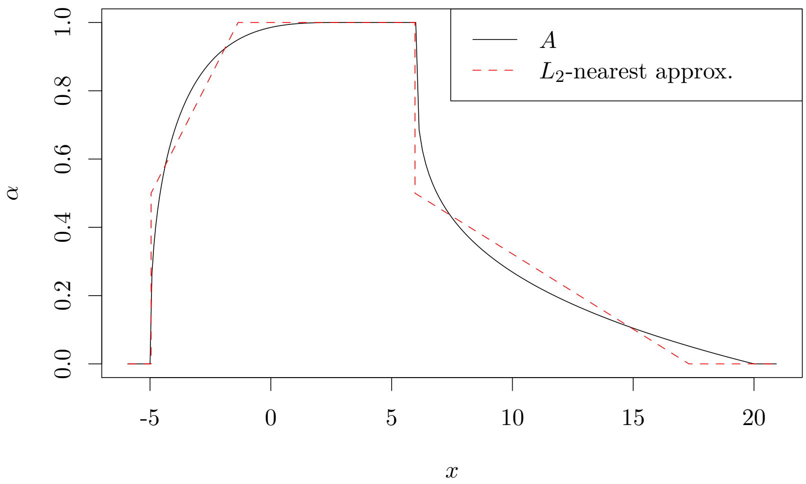
```
##    user  system elapsed
##   0.013   0.001   0.114
```

```
print(P2['allknots'], 6)
```

```
##         alpha     left  right
## supp      0.0 -4.95920 17.305
## knot_1    0.5 -4.95920  5.965
## core      1.0 -1.35769  5.965
```

```
print(distance(A, P2), 12)
```

```
## [1] 0.837361269482
```

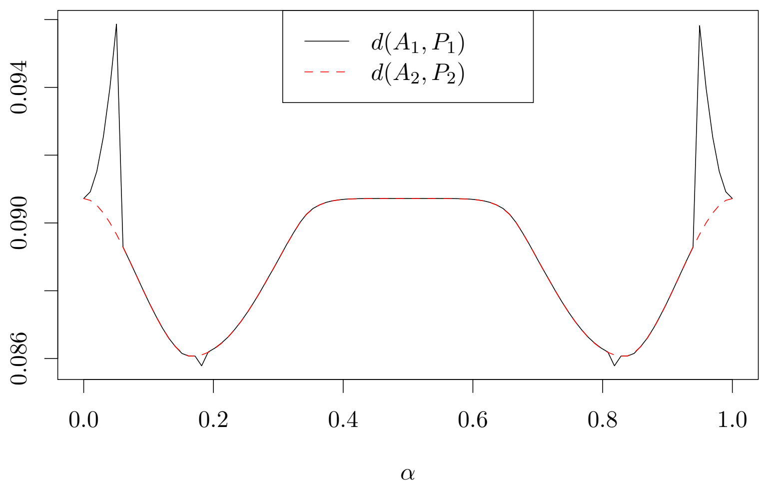



Beware of numerical error in integration e.g. due to discontinuity in $\alpha$-cuts........ ..... TO BE DONE....

```
A1 <- FuzzyNumber(0,1,1,1,
         lower=function(a) floor(3*a)/3,
         upper=function(a) 1-a
) # no info on discontinuities

A2 <- DiscontinuousFuzzyNumber(0,1,1,1,
         lower=function(a) floor(3*a)/3,
         upper=function(a) 1-a,
         discontinuities.lower=c(0, 1/3, 2/3, 1),
         discontinuities.upper=numeric(0)
) # discontinuities info included

a <- seq(1e-9, 1-1e-9, length.out=100) # many alphas from (0,1)
d1 <- numeric(length(a)) # distances #1 (to be calculated)
d2 <- numeric(length(a)) # distances #2 (to be calculated)
for (i in 1:length(a))
{
   P1 <- piecewiseLinearApproximation(A1, method='NearestEuclidean',
```

```
            knot.n=1, knot.alpha=a[i])
   P2 <- piecewiseLinearApproximation(A2, method='NearestEuclidean',
            knot.n=1, knot.alpha=a[i])

   d1[i] <- distance(A1, P1)
   d2[i] <- distance(A2, P2)
}
```

We note that in the first case the distance for $\alpha = 0$ (trapezoidal approximation) is smaller than e.g. for $\alpha \simeq 0.05$, which, theoretically, is not possible. Moreover, the distance is not continuous at some $\alpha$ (but it is in theory).

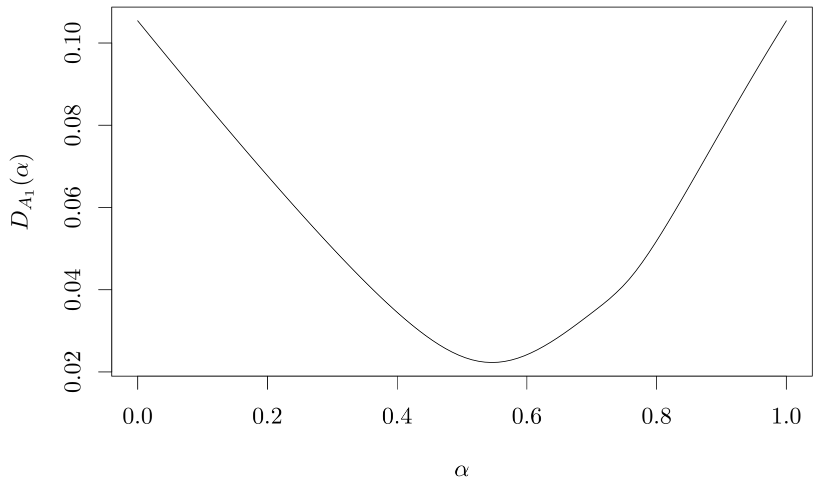



**Finding best `knot.alpha` numerically.** Consider the following fuzzy number $A_1$:

```
A1 <- FuzzyNumber(0,0,0,1,
   lower=function(a) a,
   upper=function(a) (1-a)^2)
```

Let us depict the $L_2$ distance $D_{A_1}(\alpha)$..........

```
a <- seq(1e-9, 1-1e-9, length.out=100) # many alphas from (0,1)
d <- numeric(length(a)) # distances (to be calculated)
for (i in 1:length(a))
{
   P1 <- piecewiseLinearApproximation(A1, method='NearestEuclidean',
            knot.n=1, knot.alpha=a[i])
   d[i] <- distance(A1, P1)
}
## Warning:  max(abs(d[K]))==7.14806e-08
## Warning:  max(abs(d[K]))==1.19959e-07
## Warning:  max(abs(d[K]))==3.57378e-07
## Warning:  max(abs(d[K]))==1.19209e-07
## Warning:  max(abs(d[K]))==2.38169e-07
plot(a, d, type='l', xlab=expression(alpha), ylab=expression(D[A](alpha)))
```




We may find best `knot.alpha` using numerical optimization. We only know that the distance function is continuous.

```
for (i in 1:5) # 5 iterations
{
   a0 <- runif(1,0,1) # random starting point
   optim(a0,
      function(a)
      {
         P1 <- piecewiseLinearApproximation(A1, method='NearestEuclidean',
                                             knot.n=1, knot.alpha=a)

         distance(A1, P1)
      }, method='L-BFGS-B', lower=1e-9, upper=1-1e-9) -> res
   cat(sprintf('%.9f %6g %.9f %.9f\n', a0, res$counts[1], res$par, res$value))
}
```

```
## 0.819360481     60 0.545856269 0.022293594
## 0.326543073     17 0.546145664 0.022294007
## 0.102972513     35 0.545856297 0.022293594
## 0.129096587      8 0.546262450 0.022294032
## 0.877980422     14 0.546147795 0.022294007
```

**Approximate algorithm for fixed `knot.alpha`.** This method uses a constrained version of the Nelder-Mead algorithm. The procedure minimizes the target function numerically by calling the `optim()` function. There is thus no guarantee that it will find to the global minimum (it may fall into a neighborhood of a local minimum or even fail to converge). However, this approach may be used for any number of knots.

TO BE DONE... WORK ON AN EXACT ALGORITHM IS IN PROGRESS.....

```
system.time(P3 <- piecewiseLinearApproximation(A,
   method='ApproximateNearestEuclidean', knot.n=1, knot.alpha=0.5))
```

```
##    user  system elapsed
##   1.355   0.001   1.385
```

```
print(P3['allknots'], 6)
```

```
##        alpha     left    right
## supp      0.0 -4.95921 17.30510
## knot_1    0.5 -4.95921  5.96493
## core      1.0 -1.35761  5.96493
```

```
print(distance(A, P3), digits=12)
```

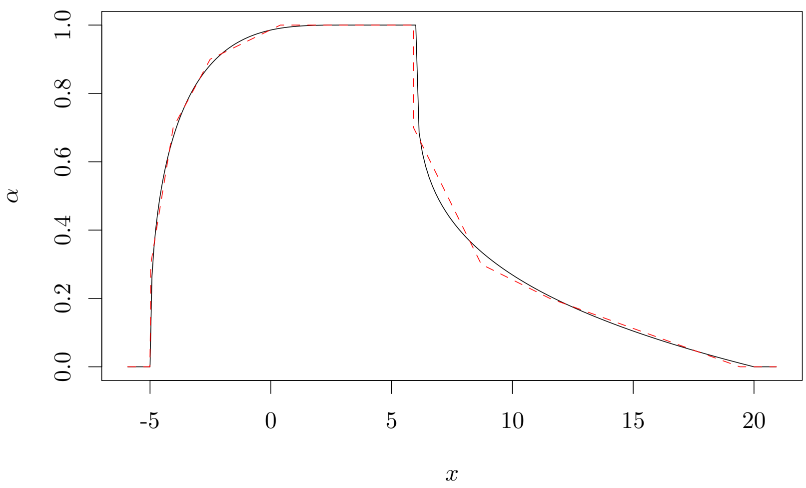
```
## [1] 0.837361274848
```

Compare with exact solution...... Please note that a call to this method may be time-consuming.

Another example:

```
system.time(P4 <- piecewiseLinearApproximation(A,
   method='ApproximateNearestEuclidean', knot.n=4,
   knot.alpha=c(0.2, 0.3, 0.7, 0.9), verbose=TRUE))
```

```
## Pass 1a,1b,DONE.
##    user  system elapsed
##  36.487   0.064  37.362
```




```
## [1] 0.283699551483
```

If the method fails to converge, you may try to call it e.g. with the `optim.control=list(maxit=5000)` parameter to allow for greater number of iterations.

# 7 NEWS/CHANGELOG

```
FuzzyNumbers Package NEWS

****************************************************************************

0.03 /under development/

* Added a call to  setGeneric("plot", function(x, y, ...) ...
  to avoid a warning on install

* New methods: as.character(); also used by show().
```

```
  This function also allows to generate LaTeX code defining the FN
  (toLaTeX arg thanks to Jan Caha).

* New binary arithmetic operators, especially
   for PiecewiseLinearFuzzyNumbers: +, -, *, /

* piecewiseLinearApproximation() and as.PiecewiseLinearFuzzyNumber()
  argument `knot.alpha` now defaults to equally distributed knots
  (via given `knot.n`)

* PiecewiseLinearFuzzyNumber() now accepts missing `a1`, `a2`, `a3`, `a4`,
  and `knot.left`, `knot.right` of length `knot.n`+2. Moreover, if `knot.n`
  is not given, then it is guessed from length(knot.left).
  If `knot.alpha` is missing, then the knots will be equally distributed
  on the interval [0,1].

* New function: fapply() - applies a function on a PLFN
  using the extension principle

* as.PiecewiseLinearFuzzyNumber() is now an S4 method,
  and can be called on objects of type numeric, as well as on
  various FNs

* ...

* The FuzzyNumbers Tutorial has been properly included
  as the package's vignette

* Man pages update & cleanup

***************************************************************************

0.02 /2012-12-27/

* approx.invert(): a new function to find the numerical
  inverse of a given side/alpha-cut generating function
  (by default via Hermite monotonic spline interpolation)

* convert.side(), convert.alpha():
  new functions to convert sides and alpha cuts
  to side generating funs and alpha cut generators

* FuzzyNumber class validity check for lower, upper, left, right:
   * checks whether each function is properly vectorized
     and gives numeric results
   * does not check for the number of formal arguments,
     but just uses the first from the list

* suggests `testthat`

* each object has been documented
```

```
* first CRAN release

*************************************************************************

0.01 /2012-07-01/

* initial release
```

**Acknowledgments.** This document has been generated with LaTeX, knitr and the tikzDevice package for R. Their authors' wonderful work is fully appreciated. Many thanks also to Przemysław Grzegorzewski, Lucian Coroianu, Jan Caha, and Pablo Villacorta Iglesias for stimulating discussion.

The contribution of Marek Gagolewski was partially supported by the European Union from resources of the European Social Fund, Project PO KL "Information technologies: Research and their interdisciplinary applications", agreement UDA-POKL.04.01.01-00-051/10-00 (March-June 2013), and by FNP START Scholarship from the Foundation for Polish Science (2013).

# Bibliography

[1] Ban A.I., Approximation of fuzzy numbers by trapezoidal fuzzy numbers preserving the expected interval, *Fuzzy Sets and Systems* **159**, 2008, pp. 1327–1344.

[2] Ban A.I., On the nearest parametric approximation of a fuzzy number – Revisited, *Fuzzy Sets and Systems* **160**, 2009, pp. 3027–3047.

[3] Chanas S., On the interval approximation of a fuzzy number, *Fuzzy Sets and Systems* **122**, 2001, pp. 353–356.

[4] Coroianu L., Gagolewski M., Grzegorzewski P., *Nearest Piecewise Linear Approximation of Fuzzy Numbers*, *Fuzzy Sets and Systems*, 2013, doi:10.1016/j.fss.2013.02.005.

[5] Delgado M., Vila M.A., Voxman W., On a canonical representation of a fuzzy number, *Fuzzy Sets and Systems* **93**, 1998, pp. 125–135.

[6] P. Diamond, P. Kloeden, Metric Spaces of Fuzzy Sets. Theory and Applications. World Scientific, Singapore, 1994.

[7] Dubois D., Prade H., Operations on fuzzy numbers, *Int. J. Syst. Sci.* **9**, 1978, pp. 613–626.

[8] Dubois D., Prade H., The mean value of a fuzzy number, *Fuzzy Sets and Systems* **24**, 1987, pp. 279–300.

[9] Gagolewski M., FuzzyNumbers: Tools to deal with fuzzy numbers in R, www.rexamine.com/resources/fuzzynumbers/, 2013.

[10] Grzegorzewski P., Metrics and orders in space of fuzzy numbers, *Fuzzy Sets and Systems* **97**, 1998, pp. 83–94.

[11] Grzegorzewski P., *Algorithms for trapezoidal approximations of fuzzy numbers preserving the expected interval*, In: Bouchon-Meunier B. et al (Eds.), *Foundations of Reasoning Under Uncertainty*, Springer, 2010, pp. 85–98.

[12] Grzegorzewski P., Pasternak-Winiarska K., *Trapezoidal approximations of fuzzy numbers with restrictions on the support and core*, In: *Proc. EUSFLAT/LFA 2011*, Atlantic Press, 2011, pp. 749–756.

[13] Klir G.J., Yuan B., *Fuzzy sets and fuzzy logic. Theory and applications*, Prentice Hall, New Jersey, 1995.

[14] Stefanini L., Sorini L., *Fuzzy arithmetic with parametric LR fuzzy numbers*, In: *Proc. IFSA/EUSFLAT 2009*, pp. 600–605.

[15] Yeh C.-T., Trapezoidal and triangular approximations preserving the expected interval, *Fuzzy Sets and Systems* **159**, 2008, pp. 1345–1353.