Fuzzy Reasoning with Continuous Piecewise Linear Membership Functions

P.M. van den Broek
Department of Computer Science, Twente University of Technology,
P.O.Box 217, 7500 AE Enschede, the Netherlands
email : pimvdb@cs.utwente.nl

Abstract

It is shown that, for some intersection and implication functions, the complexity of the computation of inference results with generalised modus ponens can be reduced considerably when membership functions are restricted to functions which are continuous and piecewise linear. Algorithms for computing inference results are given in the functional language Miranda.

1. Introduction

In fuzzy reasoning with generalised modus ponens:

Premise 1	If X=A then	Y=B
Premise 2	X=A'	
---	---	---
Conclusion		Y=B'

one calculates the fuzzy set B' from the fuzzy sets A, A' and B with

$$B'(y) = \sup_x (I(A'(x), J(A(x), B(y)))) \quad (1)$$

where I is some intersection function and J is some implication function.

When the domains of the fuzzy sets are finite, the computation of B' presents no difficulties. Here we are interested in the case where the domains are finite intervals of the real numbers. In this case one cannot compute B' in general; instead one computes B'(y) for a finite number of values of y. The computation of a single value B'(y) involves functions over a continuous interval, and usually involves the application of numerical approximation techniques.

In this paper we will show that, for some intersection and implication functions, the complexity of the computation can be reduced considerably when only membership functions are considered which are continuous and piecewise linear. A function is said to be piecewise linear if it is linear in all but a finite number of points. We observe from the literature that for membership functions one often uses piecewise linear functions. The reduction of the complexity arises from the fact that no discretization is necessary.

We will consider of course only those intersection functions and implication functions which have the property that when A, A' and B are continuous and piecewise linear, also B' will be continuous and piecewise linear. Klir and Yuan [1] list four frequently used intersections: standard intersection, algebraic product, bounded difference and drastic intersection. From these we only consider the standard intersection and the bounded difference, since the inference result for the algebraic product will not in general be piecewise linear, and the inference result for the drastic intersection will not in general be continuous. From the list of implications given by Klir and Yuan, we only consider, for the same reason, the Lukasiewicz implication, the Kleene-Dienes implication, the Early-Zadeh implication and the Willmott implication. For J we will also consider the minimum function, also known as the Mamdani implication, because it is widely used in the literature, although it is not an implication.

Due to space limitations this paper only treats the standard intersection; the full version of this paper can be obtained via http://wwwtrese.cs.utwente.nl/~pimvdb/.

For each of the 5 implications we will present an algorithm for the computation of the continuous piecewise linear function B' from continuous piecewise linear functions A, A' and B with equation (1). The problem to be solved in each case is the computation of the supremum of a continuum of continuous piecewise linear functions. We will derive our results in a rather informal way; rigorous proofs of our results are however straightforward, and therefore not given in this paper. On the other hand, complete implementations of the algorithms will be given in the functional programming language Miranda (Turner [2]), since Miranda provides an excellent formalism for the notation of algorithms, and this notation is executable.

A second advantage, besides its efficiency, of an inference system based on our algorithms, is that the users

are able to approximate continuous input functions by continous piecewise linear functions themselves, instead of relying on discretization procedures incorporated in the system. The inference system then gives exact results for the approximated inputs.

A possible extension of this work would be to drop the condition that the membership functions are continuous. This would imply that also the drastic intersection might be taken into consideration, as well as the Gaines-Resher implication, the Gödel implication, and the Wu implication. However, the algebraic product intersection, as well as the Goguen implication, the Reichenbach implication and the Yager implication still could not be treated.

2. Preliminaries

A continuous piecewise linear function (plf, for short) will be represented as a list of tuples of numbers. The list $[(x_0,y_0),(x_1,y_1),...(x_n,y_n)]$ with $x_0<x_1<...<x_n$ will represent the plf f on the closed interval $[x_0,x_n]$ which is linear on all intervals $[x_i,x_{i+1}]$ and whose values are determined by $f(x_i)=y_i$.

Let apply be the function which takes a plf g and a number x as arguments, and returns g(x). In order to solve equation (1), we calculate the plf $g_{A,A'}$ which satisfies

$$\text{apply } g_{A,A'}\ z = \sup_x (I(A'(x),J(A(x),z))) \qquad (2)$$

for all z in the interval [0,1].

The solution of equation (1) is the function `fuzzy_reasoning`, which has four parameters: a solution $g_{A,A'}$ of equation (2), and the plfs A, B and A'; its result is the plf B'. The function `fuzzy_reasoning` is the composition of the plfs $g_{A,A'}$ and B.
In Miranda:

```
plf == [(num,num)]
fuzzy_reasoning ::
       (plf->plf->plf)->plf->plf->plf->plf
fuzzy_reasoning g a b a'
                      = compose (g a a') b
```

The function `compose`, which composes plfs, is given in the appendix.

As an example, consider the case where I is the standard intersection and J is the Lukasiewicz implication. The function which solves equation (2) in this case (and which is determined in the next section) is `lukasiewicz`. If we have the rule

If X = [(0,0),(2,1),(4,0),(5,0)] then Y = [(5,0),(6,1),(7,0)]

and the fact

X = [(0,0),(1,0),(3,1),(5,0)],

then we evaluate the expression

```
fuzzy_reasoning lukasiewicz
   [(0,0),(2,1),(4,0),(5,0)]
   [(5,0),(6,1),(7,0)]
   [(0,0),(1,0),(3,1),(5,0)]
```

to obtain [(5,0.75),(5.5,1),(6.5,1),(7,0.5)] as the inferred value of Y.

In several (but not all) cases we will use the fact that A and A' are plf's by solving equation (2) for the special case where both A and A' are linear. The domain of A and A' is subdivided into a finite number of intervals where both A and A' are linear. Let A_i and A'_i be the restrictions of A and A' to the i^{th} interval. Then equation (2) may be written as

$$\text{apply } g_{A,A'}\ z = \max_i \{\sup_x I(A'_i(x),J(A_i(x),z))\} \qquad (3)$$

If $g'_{A,A'}$ is the solution of equation (2) for the case where both A and A' are linear, the solution for the general case follows from equation (3):

$$\text{apply } g_{A,A'}\ z = \max_i \{\text{ apply } g'_{A_i,A'_i}\ z\}$$

In the appendix we give a Miranda function `all_intervals`, which determines the general solution when given a solution for the case of linear functions; for instance , the function `lukasiewicz`, mentioned above, can be defined by

```
lukasiewicz = all_intervals lukasiewicz'
```

where `lukasiewicz'` solves equation (2) in the case where I is the standard intersection, J is the Lukasiewicz implication and both A and A' are linear functions.

In all cases which we consider, the expression I(A'(x),J(A(x),z)) is a plf of the argument z, which means that it can be written as apply $f_{A,A',x}$ z. Equation (2) then becomes

$$\text{apply } g_{A,A'}\ z = \sup_x (\text{apply } f_{A,A',x}\ z) \qquad (4)$$

In the subsequent sections of this paper we will, for each implication J mentioned in the introduction, determine the plf's $f_{A,A',x}$ for each x in the domain $[x_0,x_1]$ of A and A' from

$$\text{apply } f_{A,A',x}\ z = \min(A'(x),J(A(x),z)), \qquad (5)$$

and find a solution $g_{A,A'}$ of equation (4), when needed only for the special case where A and A' are linear.

3. Lukasiewicz implication

In this case J(a,b) = min(1,1-a+b); so equation (5) reads

apply $f_{A,A',x}$ z = min(A'(x),1-A(x)+z) (6)

We assume that A and A' are linear on their domain $[x_0,x_1]$. The graphs of $f_{A,A',x0}$ and $f_{A,A',x1}$ are given in figure 1 (dashed resp. dash-dot-dotted).

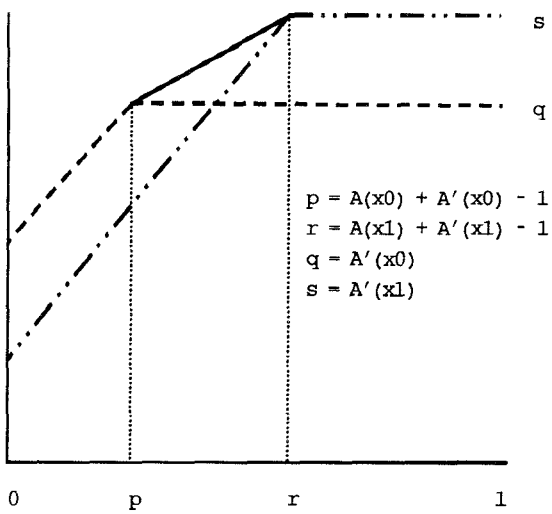

Figure 1

These graphs are non-linear in (p,q) and (r,s) respectively (p,q,r,and s are defined in the figure). We consider here the case where p≤r. Note that both p and r may be negative; in order to be able to treat all intervals in the same way we let z range over the interval [-1,1].The graphs of the functions $f_{A,A',x}$ with x in $[x_0,x_1]$, all have a point where they are non-linear; these points are situated on the (bold) straight line from (p,q) to (r,s) in figure 1, due to the linearity of A and A' on $[x_0,x_1]$.It is now easy to see how the function $g_{A,A'}$ from equation (4) is obtained. For z≤p and z≥r it is just the maximum of $f_{A,A',x0}$ and $f_{A,A',x1}$. For p<z<r it is the maximum of $f_{A,A',x0}$, $f_{A,A',x1}$ and the plf [(p,q),(r,s)].This leads to the following implementation of the function `lukasiewicz'`:

```
lukasiewicz' :: plf -> plf -> plf
lukasiewicz' [(x0,a0),(x1,a1)]
             [(x0,a0'),(x1,a1')]
  = normalize
      (from0 (maxplf f1 (maxplf f2 f3))
    where

    [(p,q),(r,s)]
     = sort [(a0+a0'-1,a0'),(a1+a1'-1,a1')]
    f1 = [(-1,q-p-1),(p,q),(r,s),(1,s)]
    f2 = [(-1,q-p-1),(p,q),(1,q)]
    f3 = [(-1,s-r-1),(r,s),(1,s)]
    from0 f = (0,apply f 0) :
              dropwhile ((<=0).fst) f
```

Here `from0` is the function which restricts its argument to the domain [0,1]. The functions `normalize` (which removes redundant entries from the representation of a plf), `apply` (introduced in section 2), and `maxplf` (which computes the maximum of two plf's), are given in the appendix.

As explained in the previous section, the function `lukasiewicz` is defined by

```
lukasiewicz :: plf -> plf -> plf
lukasiewicz = all_intervals lukasiewicz'
```

4. Kleene-Dienes implication

In this case J(a,b) = max(1-a,b); so equation (5) reads

apply $f_{A,A',x}$ z = min(A'(x),max(1-A(x),z)) (7)

The graph of $f_{A,A',x}$ is given in figure 2.

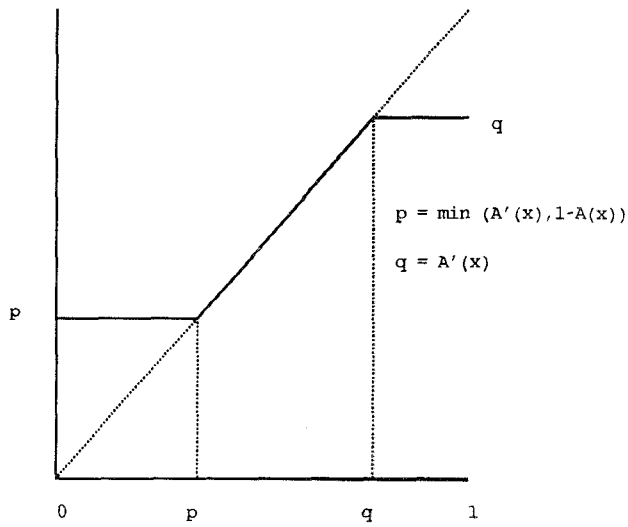

Figure 2

The graph of $g_{A,A'}$ has the same form as the graph in figure 2. So $g_{A,A'}$ can be constructed from $g_{A,A'}(0)$ and $g_{A,A'}(1)$, which are given by

$g_{A,A'}(0) = \sup_x (\min(A'(x),1-A(x)))$

and

$g_{A,A'}(1) = \sup_x A'(x)$,

respectively. Note that we did not need the assumption here that A and A' are linear. This leads to the following implementation of the function `kleene_dienes`:

```
kleene_dienes :: plf -> plf -> plf
kleene_dienes a a'
  = normalize [(0,p),(p,p),(q,q),(1,q)]
    where
    p = sup (minplf a' (complement a))
    q = sup a'
```

The functions `normalize`, `sup` (which computes the maximum value of a plf), `minplf` (which computes the minumum of two plf's), and `complement` (which changes the values y of a plf into 1-y), are given in the appendix.

5. Early-Zadeh implication

In this case J(a,b) = max (1-a,min(a,b)); so equation (5) reads

$$\text{apply } f_{A,A',x}\ z = \min(A'(x),\max(1-A(x),\min(A(x),z))) \quad (8)$$

The graph of $f_{A,A',x}$ is given in figure 3.

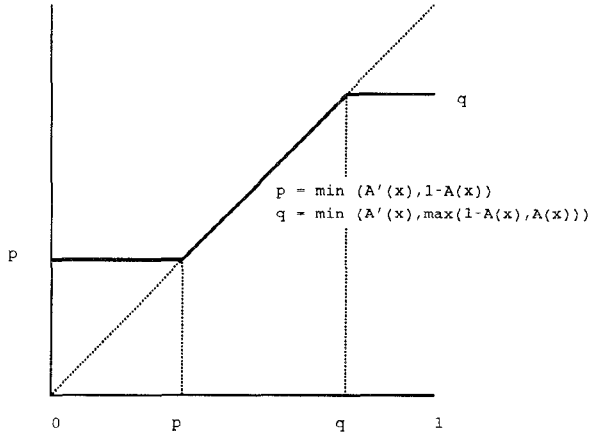

Figure 3

The graph of $g_{A,A'}$ has the same form as the graph in figure 3. So $g_{A,A'}$ can be constructed from $g_{A,A'}(0)$ and $g_{A,A'}(1)$, which are given by

$$g_{A,A'}(0) = \sup_x (\min(A'(x),1-A(x)))$$

and

$$g_{A,A'}(1) = \sup_x (\min(A'(x),\max(1-A(x),A(x))))$$

respectively. This leads to the following implementation of the function `early_zadeh`:

```
early_zadeh a a'
   = normalize [(0,p),(p,p),(q,q),(1,q)]
     where
     p = sup (minplf a' (complement a))
     q = sup (minplf a'
              (maxplf (complement a) a))
```

The functions `normalize`, `sup`, `minplf`, `complement` and `maxplf` are given in the appendix.

6. Willmott implication

In this case J(a,b) = min (max(1-a,b),max(a,1-a),max (b,1-b)); so equation (5) reads

$$\text{apply } f_{A,A',x}\ z = \min(A'(x),\min(\max(1-A(x),z),\max(A(x),1-A(x)),\max(z,1-z))) \quad (9)$$

It is straightforward to verify that equation (9) can be written as

$$\text{apply } f_{A,A',x}\ z = \min(\max(z,1-z),\min(A'(x),\max(1-A(x),\min(A(x),z)))) \quad (10)$$

Comparing this equation with equation (8) shows that an implementation of `willmott` is obtained directly from the implementation of of `early_zadeh`:

```
willmott a a'
     = normalize (minplf (early_zadeh a a')
                         [(0,1),(0.5,0.5),(1,1)])
```

The functions `normalize` and `minplf` are given in the appendix.

7. Mamdani implication

In this case J(a,b) = min(a,b); so equation (5) reads

$$\text{apply } f_{A,A',x}\ z = \min(A'(x),A(x),z) \quad (11)$$

The graph of $f_{A,A',x}$ is given in figure 4. The graph of $g_{A,A'}$ has the same form as the graph in figure 4. So $g_{A,A'}$ can be constructed from $g_{A,A'}(1)$, which is given by

$$g_{A,A'}(1) = \sup_x (\min(A'(x),A(x)))$$

This leads to the following implementation of the function `mamdani`:

```
mamdani a a'
   = normalize [(0,0),(q,q),(1,q)]
     where
     q = sup (minplf a a')
```

The functions `sup` and `minplf` are given in the appendix.

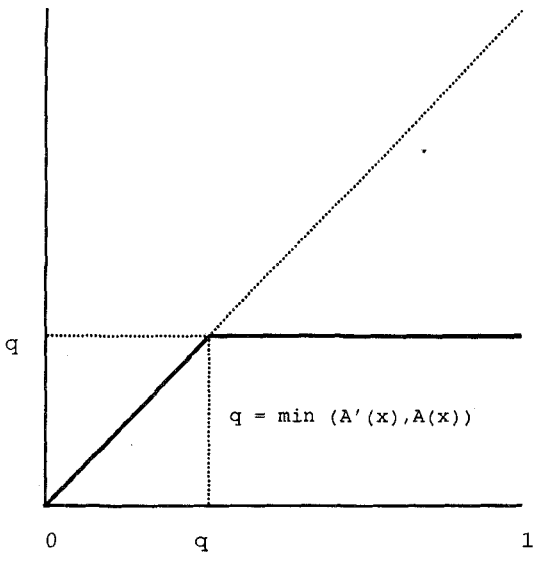

Figure 4

References

[1] G.J. Klir and B. Yuan, *Fuzzy sets and fuzzy logic, theory and applications* (Prentice-Hall) 1995

[2] D. Turner, *Miranda: a non-strict functional language with polymorphic types*, in: Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science Vol. 201, ed. J.-P. Jouannaud, (Springer-Verlag) 1-16, 1985

Appendix

This appendix contains the Miranda functions for handling plfs.

The function `apply` takes a plf A and a number x as arguments, and returns A(x).

```
apply :: plf -> num -> num
apply ((x0,y0):(x1,y1):zs) x
   = apply ((x1,y1):zs) x,
           equal x0 x1 \/ greater x x1
   = y0 + (x-x0)*(y1-y0)/(x1-x0)   ,
           otherwise
```

The function `solve` takes a plf A and a number x as arguments, and returns the list of all y which satisfy A(y)=x. If A(y) is equal to x on an interval, the list contains only the boundaries of the interval.

```
solve :: plf -> num -> [num]
solve [(a,b)] c = [a], equal b c
                = [], otherwise
solve ((a1,b1):(a2,b2):abs) c
  = a1 : solve ((a2,b2):abs) c, equal b1 c
  = a3 : solve ((a2,b2):abs) c,
                    greater 0 ((b2-c)*(b1-c))
  = solve ((a2,b2):abs) c, otherwise
    where a3 = a1 + (a2-a1)*(c-b1)/(b2-b1)
```

The function `complement` takes a plf A as argument and returns the plf A' with A'(x) = 1-A(x).

```
complement :: plf -> plf
complement a = [(x,1-y)|(x,y)<-a]
```

The function `dac` is a general recursion scheme to compute recursively functions of type `plf -> plf -> plf`, such as `minplf`, `maxplf` and `all_intervals`. The second argument of `dac` is the requested function for the special case where both plf's are linear. The first argument of `dac` is the function which returns the final result, given the result for an initial segment of the domain where both plf's are linear, and the result for the remainder of the domain.

```
dac :: (plf->plf->plf)->
       (plf->plf->plf)->plf->plf->plf
dac g f plf1 plf2 = f plf1 plf2,
                          #plf1 = 2 & #plf2 = 2
dac g f ((x0,a0):(x1,a1):xs)
        ((y0,b0):(y1,b1):ys)
= g (f [(x0,a0),(x1,a1)] [(y0,b0),(y1,b1)])
  (dac g f ((x1,a1):xs) ((y1,b1):ys)),
                                  equal x1 y1
= g (f [(x0,a0),(x1,a1)] [(y0,b0),(x1,b2)])
  (dac g f ((x1,a1):xs)
  ((x1,b2):(y1,b1):ys)),    greater y1 x1
= g (f [(x0,a0),(y1,a2)] [(y0,b0),(y1,b1)])
    (dac g f ((y1,a2):(x1,a1):xs)
             ((y1,b1):ys))   , otherwise
  where b2 = apply [(y0,b0),(y1,b1)] x1
        a2 = apply [(x0,a0),(x1,a1)] y1
```

The function `crossings` computes the list of cross-points of two linear plf's.

```
crossings :: plf -> plf -> [(num,num)]
crossings [(x0,a0),(x1,a1)]
          [(x0,b0),(x1,b1)]
  = [], equal (a1-a0) (b1-b0) \/
       greater_or_equal x0 y \/
       greater_or_equal y x1
  = [(y,b)], otherwise
    where y = x0 + (x1-x0)*(a0-b0)/
                           (a0-b0+b1-a1)
          b = a0 + (a1-a0)*(y-x0)/(x1-x0)
```

The function `compose` takes two plfs A and B as arguments, and returns the plf C with C(x) = A(B(x))

```
compose :: plf -> plf -> plf
compose a b
  = [(x,apply a (apply b x))
                |x<-sort (mkset (xs++ys))]
    where
    xs = map fst b
    ys = concat [solve b kr|kr<- map fst a]
```

The function sup takes a plf A as argument and returns $\sup_x A(x)$

```
sup :: plf -> num
sup = max.map snd
```

The function normalize takes a plf A as argument, and returns the same A with a "normalised" representation: duplicate points and points where A happens to be linear are removed

```
normalize :: plf -> plf
normalize
 = normalize2.normalize1
   where
   normalize1 ((x0,y0):(x1,y1):xys)
   = normalize1 ((x1,y1):xys), equal x0 x1
   = (x0,y0):normalize1 ((x1,y1):xys),
                                  otherwise
   normalize1 x = x
   normalize2 ((x0,y0):(x1,y1):(x2,y2):xys)
   = normalize2 ((x0,y0):(x2,y2):xys),
              equal ((y1-y0)/(x1-x0))
                    ((y2-y1)/(x2-x1))
   = (x0,y0):normalize2
          ((x1,y1):(x2,y2):xys), otherwise
   normalize2 x = x
```

The function maxplf takes two plfs A and B as arguments, and returns the plf C with C(x) = max (A(x),B(x))

```
maxplf :: plf -> plf -> plf
maxplf a a' =
 = dac g f (normalize a) (normalize a')
   where
   g xs ys = xs ++ tl ys
   f [(x0,a0),(x1,a1)] [(y0,b0),(y1,b1)]
    = (x0,max[a0,b0]) : crossings
      [(x0,a0),(x1,a1)] [(y0,b0),(y1,b1)]
      ++ [(x1,max[a1,b1])]
```

The function minplf takes two plfs A and B as arguments, and returns the plf C with C(x) = min (A(x),B(x))

```
minplf :: plf -> plf -> plf
minplf  a a'
 = dac g f (normalize a) (normalize a')
   where
   g xs ys = xs ++ tl ys
   f [(x0,a0),(x1,a1)] [(y0,b0),(y1,b1)]
    = (x0,min[a0,b0]) : crossings
      [(x0,a0),(x1,a1)] [(y0,b0),(y1,b1)]
      ++ [(x1,min[a1,b1])]
```

The function all_intervals is explained in section 3.

```
all_intervals :: (plf -> plf -> plf)
              -> plf -> plf -> plf
all_intervals f a a'
 = normalize (dac maxplf f a a')
```

Finally some functions for handling rounding errors in floating point arithmetic:

```
equal :: num->num->bool
equal n m = abs(n-m)<0.000000000001

greater_or_equal :: num->num->bool
greater_or_equal n m = n>m-0.000000000001

greater :: num->num->bool
greater n m = n>m & ~equal n m
```