An Investigation into Fuzzy Systems

Tim Lawson

University of Bristol

tim.lawson@bristol.ac.uk

Q1 Fuzzy control and sound synthesis

1 Introduction

Natural and artificial systems are frequently complex, non-linear, and operate under variable conditions. Accordingly, approaches to modelling and controlling these systems must be designed to approximate non-linear functions and handle uncertainty. Fuzzy systems and neural networks are different, and sometimes complementary, approaches to modelling and control. The learned parameters of neural networks are often difficult to interpret, and their behaviours are difficult to explain. By contrast, fuzzy systems can learn linguistically interpretable rules that can be verified by humans, or explicitly encode human knowledge (Babuška and Verbruggen 1996, p. 1593). Interpretability and explainability are central concerns in artificial-intelligence research (e.g. Gilpin et al. 2018), particularly in its applications to risk-averse domains, such as healthcare, robotics, and industrial processes. Additionally, in creative applications, the ability to encode human knowledge and define novel behaviours is a desirable feature of fuzzy approaches. In this essay, I provide a brief overview of fuzzy control systems in section 2 and an example of its application to sound synthesis in section 3. Finally, I contrast this case study with neural-network approaches in section 4.

2 Fuzzy control

Since Zadeh's introduction of fuzzy sets (1965), fuzzy logic has been widely applied to control systems (Klir and Yuan 1995, p. 330). A basic feedback control system comprises a controlled object, sensors that measure the conditions of the object, and a controller that generates actions to apply to it (Doyle et al. 1990, p. 27). The controller applies inference rules to the conditions to generate actions. A schematic of this arrangement is given in fig. 1. Generally, sensors produce crisp measurement values and actuators apply actions that are defined by crisp values. Hence, to apply fuzzy inference rules, a fuzzy controller:

1. fuzzifies the conditions of the object, i.e., converts them to fuzzy sets;
2. applies fuzzy inference rules to the fuzzified conditions; and
3. defuzzifies the outputs of the inference rules, i.e., converts them to crisp values.

This procedure is depicted in fig. 2, after Klir and Yuan (1995, pp. 331–332).

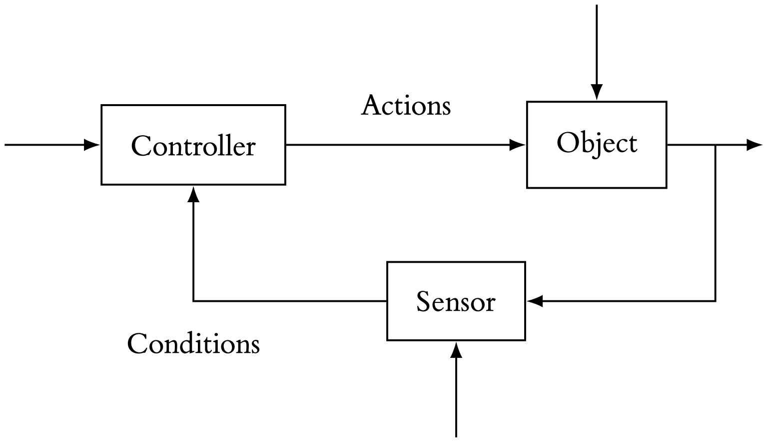

Figure 1: A basic feedback control system (Doyle et al. 1990, p. 27).

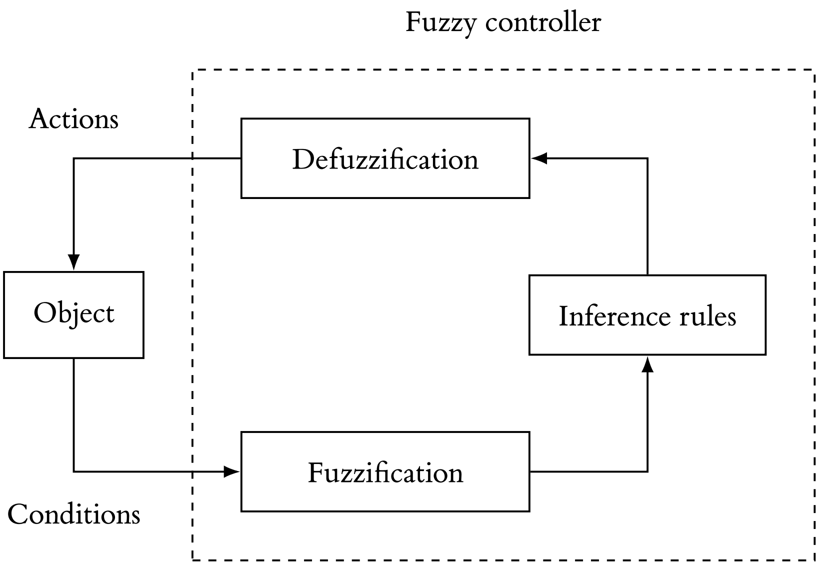

Figure 2: A fuzzy control system (Klir and Yuan 1995, pp. 331–332).

2.1 Fuzzification

In a fuzzy system, variables are described by fuzzy sets (Klir and Yuan 1995, p. 327). To convert a crisp value to a fuzzy set, the 'universe of discourse' of the variable, i.e., the range of values it takes, and the semantics of the fuzzy set that describes it, i.e., its membership function, must be defined (Sugeno 1985, p. 60). For example, instead of a crisp value of 60 dB, the amplitude of a sound may be expressed in terms of the membership values of the elements of a fuzzy set {low, medium, high}. In this way, a fuzzy representation of a variable can account for the inherent uncertainty of measurement values and the limited resolution of measurement instruments, and present a more intuitive description to human designers and operators.

2.2 Inference

Zadeh later presented an approach to fuzzy system modelling based on *linguistic variables*, i.e., variables whose values are expressions in a natural or artificial language (1975, p. 199). As above, the amplitude of a sound event could be described as 'low', 'medium', or 'high' instead of a crisp value in decibels. Mamdani and Assilian (1975) were the first to apply this approach to control systems. In this scheme, the inference rules of a control system are *if-then* statements whose premises and consequences are expressed in terms of linguistic variables (Nguyen, Taniguchi, et al. 2019, pp. 57–58). For example, the application of the rule '*if* the amplitude is low *then* increase the amplitude' to a sound event whose amplitude is 'low' may produce an event whose amplitude is 'medium'. An *if-then* statement is an example of *modus ponens*, an argumentative form in propositional logic – Zadeh's approach is thus sometimes called generalized modus ponens (e.g. Dubois and Prade 1984).

A key advantage of this type of system is that it is generally easier for humans to formulate inference rules in linguistic terms than in mathematical terms, which helps to encode expert knowledge in the system and to understand its behaviour. On the other hand, Mamdani controllers do not construct an explicit model of the controlled process, which makes it difficult to analyse its stability (Nguyen, Taniguchi, et al. 2019, p. 58). Stability analysis is an important aspect of the design of control systems, which are frequently used to maintain an equilibrium state (Doyle et al. 1990, p. 3).

However, not all fuzzy control systems are based on linguistic variables. Accordingly, Sugeno (1985) classifies fuzzy systems according to the form of the consequences of their inference rules (table 1). For example, Takagi-Sugeno controllers map input values to (usually) linear functions (Takagi and Sugeno 1985; Nguyen, Taniguchi, et al. 2019, pp. 58–59). This approach is particularly useful for modelling non-linear systems, but sacrifices the interpretability of linguistic variables in its outputs. Nguyen, Sugeno, et al. (2017) presents a third type of fuzzy system, in which the consequences of the inference rules are real numbers. These 'singleton' or piecewise multi-affine systems are computationally inexpensive and amenable to stability analysis (Nguyen, Taniguchi, et al. 2019, pp. 63–64).

2.3 Defuzzification

The output fuzzy sets of a fuzzy controller must be converted to crisp values to be applied to the controlled object. Generally, approaches to defuzzification are based on either the maxima of the membership function of the fuzzy set, its distribution, or the area under its curve, depending on the

Type	Consequences	Reference
Mamdani	fuzzy sets	Mamdani and Assilian (1975)
Takagi-Sugeno	(linear) functions	Takagi and Sugeno (1985)
Singleton	real numbers	Nguyen, Sugeno, et al. (2017)

Table 1: Types of fuzzy systems (Nguyen, Taniguchi, et al. 2019).

type of the variable (Leekwijck and Kerre 1999, pp. 166–172). The first two of these approaches are illustrated by definitions 1 and 2, in which $\tilde{A}$ is a fuzzy set with membership function $\chi_A : W \to [0, 1]$.

Definition 1. A *random choice of maxima* is the result of an experiment with the probability distribution:

$$P(x) = \begin{cases} |\mathrm{core}(\tilde{A})|^{-1} & \text{if } x \in \mathrm{core}(\tilde{A}) \\ 0 & \text{otherwise} \end{cases}, \quad \mathrm{core}(\tilde{A}) = \left\{ x \in W \mid \chi_A(x) = \max_{y \in W} \chi_A(y) \right\} \tag{1}$$

Definition 2. The *centre of gravity* of $\tilde{A}$ is:

$$\mathrm{cog}(\tilde{A}) = \frac{\sum_{x \in W} x \cdot \chi_A(x)}{\sum_{x \in W} \chi_A(x)}. \tag{2}$$

If χ_A is a probability distribution, then the centre of gravity is equivalent to the expected value of the random variable $X \in W$ where $P(x) = \chi_A(x)$.

3 Case study: sound synthesis

The opportunity for humans to define the inference rules that operate a control system has clear benefits for creative applications. As Cádiz (2020, pp. 1–2) explains, musical concepts are frequently imprecise, such as the directives of tempo and dynamics in a musical score. Furthermore, non-linearity and the complex interplay of components are desirable characteristics of music-making systems.[1] A promising creative application of fuzzy systems is sound synthesis, i.e., the computer-based generation of sound, which is surveyed by de Poli (1983). In general, a synthesis algorithm involves many parameters, whose values must be manually programmed or otherwise controlled to produce a desired sound. Parametric control is thus an important aspect of electronic composition, performance, and sound design.

Cádiz (2020) describes an application of fuzzy control to parametric control of granular synthesis. A granular synthesis algorithm generates sound from many very short sound events or 'grains' (Roads 1988), taking inspiration from physics (Gabor 1946). It is particularly amenable to novel control methods due to the large number of input parameters and the opacity of their relations to the output sound (Cádiz 2020, p. 11). In this instance, the controlled object is a `Max/MSP`[2] object with five parameters that are varied by the control system. The evolution of these parameters over time according to fuzzy inference rules generates complex time-series from a comparably small number of user inputs. Naturally, this methodology can be applied to control other synthesis algorithms (e.g. Cádiz and Inostroza 2018) and software environments.

[1] For example, the musician and synthesizer designer Vlad Kreimer cites these characteristics as key principles of his design philosophy (mylarmelodies 2023).

[2] See `https://cycling74.com/products/max` or Manzo (2011), for example.

4 Neural networks

Various authors have advocated for fuzzy set theory in the context of control systems and artificial intelligence, as opposed to probability and statistics, due to its ability to explicitly represent different kinds of uncertainty (Laviolette et al. 1995, p. 249). Neural networks are probabilistic models that are commonly used to approximate non-linear functions by learning from data. In contrast to section 3, for instance, Bitton et al. (2020) describe a granular sound synthesis technique based on a generative neural network. In this case, the properties of the grains are represented by a latent space, which is learned by a variational auto-encoder, and the network generates the waveform directly, instead of controlling a synthesis algorithm. This approach does not allow for the explicit definition of inference rules, but it can learn to generate waveforms that are similar to a corpus of audio. Closer parallels to the work of Cádiz (2020) are provided by Fiebrink et al. (2009), who presented a system to learn mappings between user inputs and algorithm parameters, and Jonason et al. (2020), who presented a similar 'control-synthesis' approach to transforming user inputs, based on a recurrent neural network. Thus, the principal difference between neural-network and fuzzy approaches to control systems in this context is whether the inference rules are explicitly defined by the user or learned from data. Generally, however, fuzzy inference rules can also be constructed from data, including by the use of a neural network (Klir and Yuan 1995, pp. 281, 295–296).

References

Babuška, R. and H.B. Verbruggen (1996). "An Overview of Fuzzy Modeling for Control". In: *Control Engineering Practice* 4.11, pp. 1593–1606.

Bitton, Adrien, Philippe Esling, and Tatsuya Harada (2020). "Neural Granular Sound Synthesis". In: *International Computer Music Conference*. Santiago, Chile.

Cádiz, Rodrigo F. (2020). "Creating Music With Fuzzy Logic". In: *Frontiers in Artificial Intelligence* 3.

Cádiz, Rodrigo F. and Marie Carmen González Inostroza (2018). "Fuzzy Logic Control Toolkit 2.0: Composing and Synthesis by Fuzzyfication." In: *Proceedings of the 2018 Conference on New Interfaces for Musical Expression (NIME 2018)*. Vol. 18. Blacksburg, VA, pp. 398–402.

de Poli, Giovanni (1983). "A Tutorial on Digital Sound Synthesis Techniques". In: *Computer Music Journal* 7.4, pp. 8–26.

Doyle, John C., Bruce A. Francis, and Allen Tannenbaum (1990). *Feedback Control Theory*. New York, NY: Macmillan Publishing Co.

Dubois, Didier and Henri Prade (1984). "Fuzzy Logics and the Generalized Modus Ponens Revisited". In: *Cybernetics and Systems* 15.3-4, pp. 293–331.

Fiebrink, Rebecca, Perry R. Cook, and Dan Trueman (2009). "Play-along Mapping of Musical Controllers". In: *Proceedings of the 2009 International Computer Music Conference, ICMC 2009*. International Computer Music Association, pp. 61–64.

Gabor, D. (1946). "Theory of Communication". In: *Journal of the Institution of Electrical Engineers – Part III: Radio and Communication Engineering* 93.26, pp. 429–457.

Gilpin, Leilani H. et al. (2018). “Explaining Explanations: An Overview of Interpretability of Machine Learning”. In: *2018 IEEE 5th International Conference on Data Science and Advanced Analytics (DSAA)*, pp. 80–89.

Jonason, Nicolas, Bob L. T. Sturm, and Carl Thome (2020). “The Control-Synthesis Approach for Making Expressive and Controllable Neural Music Synthesizers”. In:

Klir, George J. and Bo Yuan (1995). *Fuzzy Sets and Fuzzy Logic: Theory and Applications*. Upper Saddle River, NJ: Prentice Hall.

Laviolette, Michael et al. (1995). “A Probabilistic and Statistical View of Fuzzy Methods”. In: *Technometrics* 37.3, pp. 249–261.

Leekwijck, Werner Van and Etienne E. Kerre (1999). “Defuzzification: Criteria and Classification”. In: *Fuzzy Sets and Systems* 108.2, pp. 159–178.

Mamdani, E. H. and S. Assilian (1975). “An Experiment in Linguistic Synthesis with a Fuzzy Logic Controller”. In: *International Journal of Man-Machine Studies* 7.1, pp. 1–13.

Manzo, V. J. (2011). *Max/MSP/Jitter for Music: A Practical Guide to Developing Interactive Music Systems for Education and More*. New York, NY: Oxford University Press.

mylarmelodies (2023). *A synth design masterclass by SOMA SYNTHS Vlad Kreimer*. URL: `https://www.youtube.com/watch?v=WxdzM86Urhg`.

Nguyen, Anh-Tu, Michio Sugeno, et al. (2017). “LMI-Based Stability Analysis for Piecewise Multi-affine Systems”. In: *IEEE Transactions on Fuzzy Systems* 25.3, pp. 707–714.

Nguyen, Anh-Tu, Tadanari Taniguchi, et al. (2019). “Fuzzy Control Systems: Past, Present and Future”. In: *IEEE Computational Intelligence Magazine* 14.1, pp. 56–68.

Roads, Curtis (1988). “Introduction to Granular Synthesis”. In: *Computer Music Journal* 12.2, pp. 11–13.

Sugeno, Michio (1985). “An Introductory Survey of Fuzzy Control”. In: *Information Sciences* 36.1, pp. 59–83.

Takagi, Tomohiro and Michio Sugeno (1985). “Fuzzy Identification of Systems and Its Applications to Modeling and Control”. In: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-15.1, pp. 116–132.

Zadeh, L. A. (1965). “Fuzzy Sets”. In: *Information and Control* 8.3, pp. 338–353.

— (1975). “The Concept of a Linguistic Variable and its Application to Approximate Reasoning – I”. In: *Information Sciences* 8.3, pp. 199–249.

Q2 Fuzzy sets in Python

This section explains the Python code that I wrote to complete the questions posed, and specific test cases that I wrote to verify its correctness. The code is reproduced in appendix A and available at `tslwn/fuzzy-systems`. I wrote the code in Python 3.12, to take advantage of the type-annotation syntax for generic classes and functions.[3] I completed the majority of the questions by writing methods of a `FuzzySet` class, shown in lines 14–160, which implements the built-in abstract `Set` class.

Q2(a) From fuzzy sets to α-cuts

The aim of this question is to compute the α-cuts of a discrete fuzzy set with a finite number of elements. The corresponding Python code is shown in lines 46–82 of `main.py`. The `FuzzySet` method `alpha_cut` returns the set of elements of the fuzzy set whose membership values are greater than or equal to the given α-value. Then, the method `alpha_cuts` iterates over pairs of membership values, with the addition of zero, sorted in increasing order of value. It returns a dictionary whose keys are the sets of elements and whose values are the corresponding intervals of α-values. Two test cases that apply to fuzzy sets of at least ten elements with non-zero membership values are shown in lines 62–86 of `main_test.py`. The second test case applies to the fuzzy set shown in lines 17–30, whose α-cuts are shown in lines 76–86. The corresponding equations are:

$$\tilde{A} = 1/0.1 + 2/0.1 + 3/0.3 + 4/0.3 + 5/0.5 + 6/0.5 + 7/0.7 + 8/0.7 + 9/0.9 + 10/0.9 + 11/1 \tag{3}$$

$$\tilde{A}_\alpha = \begin{cases} \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\} : \alpha \in (0, 0.1] \\ \{3, 4, 5, 6, 7, 8, 9, 10, 11\} : \alpha \in (0.1, 0.3] \\ \{5, 6, 7, 8, 9, 10, 11\} : \alpha \in (0.3, 0.5] \\ \{7, 8, 9, 10, 11\} : \alpha \in (0.5, 0.7] \\ \{9, 10, 11\} : \alpha \in (0.7, 0.9] \\ \{11\} : \alpha \in (0.9, 1] \end{cases} \tag{4}$$

A third test case corresponds to Example 5.3.2 from the lecture notes, shown in lines 88–96.

Q2(b) From α-cuts to fuzzy sets

The aim of this question is to compute the discrete fuzzy set that corresponds to the given α-cuts. The corresponding Python code is shown in lines 84–103 of `main.py`. The static `FuzzySet` method `from_alpha_cuts` iterates over the given dictionary, whose keys are the sets of elements and whose values are the corresponding intervals of α-values, as in Q2(a). It returns a fuzzy set whose elements are the unique elements of the keys of the dictionary and whose membership values are the maximum α-values of the corresponding intervals. Its consistency with the code that corresponds to Q2(a) is verified by the test cases shown in lines 98–128 of `main_test.py`, which apply to the same fuzzy sets as the test cases shown in lines 62–86. The third case corresponds to Example 5.3.3 from the lecture notes, shown in lines 130–140.

[3] See `https://docs.python.org/3.12/whatsnew/3.12.html#pep-695-type-parameter-syntax`.

Q2(c) Functions of fuzzy sets

The aims of this question are to compute the set $\{f(x) : x \in A\}$, given a function $f : \mathbb{R} \to \mathbb{R}$ and a set of real numbers A, and to use this to compute $f(\tilde{A})$ by the α-cut method. The corresponding Python code for the first of these aims is shown in lines 192–214 of `main.py`. The function `apply_elementwise` returns a set whose elements are the results of applying the function f to the elements of the set A. Two test cases that apply this function with $f(x) = x^2$ (one-to-one) and $f(x) = \lfloor \frac{x}{2} \rfloor$ (many-to-one) are shown in lines 211–217 of `main_test.py`.

The corresponding Python code for the second of these aims is shown in lines 105–135 and 163–189 of `main.py`. The `FuzzySet` method `apply_elementwise` iterates over the α-cuts of the fuzzy set $\tilde{A}$, as in Q2(a), and applies the function f to the set of elements of each α-cut with the Python function `apply_elementwise`, described above. Then, the function `merge_alpha_cuts` merges the result by taking the union of the intervals of α-cuts that correspond to the same set of elements. Finally, the static method `from_alpha_cuts` is applied to the result to return the fuzzy set $f(\tilde{A})$. Two test cases that apply to the functions f defined above are shown in lines 142–185 of `main_test.py`. A third test case corresponds to Example 5.4.2 from the lecture notes, shown in lines 187–199.

Q2(d) Conditional probability distributions

The aim of this question is to compute the conditional probability distribution $P(w \mid \tilde{A})$, defined as follows. Let $P : 2^W \to [0, 1]$ be a probability distribution and $\tilde{A}$ be a fuzzy set characterized by a membership function $\chi_{\tilde{A}} : W \to [0, 1]$. Then, for all $w \in W$:

$$P(w \mid \tilde{A}) = \int_0^1 P(w \mid \tilde{A}_\alpha)\, \mathrm{d}\alpha \tag{5}$$

$$= \int_0^1 \frac{P(w \cap \tilde{A}_\alpha)}{P(\tilde{A}_\alpha)}\, \mathrm{d}\alpha\,, \quad P(w \cap \tilde{A}_\alpha) = \begin{cases} P(w) & \text{if } w \in \tilde{A}_\alpha \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

The corresponding Python code is shown in lines 215–291 and 137–160 of `main.py`. The integral is performed by the `FuzzySet` method `apply_numeric` (lines 137–160), which takes a function $f : 2^W \to \mathbb{R}$. It iterates over the α-cuts of the fuzzy set $\tilde{A}$, applies the function f to the set of elements of each α-cut, multiplies them by the size of the corresponding α-value interval, and sums the results. A test case that corresponds to Example 5.4.1 from the lecture notes is shown in lines 201–208 of `main_test.py`.

The function `fuzzy_cond_prob_dist` completes the computation. First, it defines a function that returns the conditional probability of a possible world given a proposition, i.e., eq. (6), by dividing the probability of the intersection of the proposition and the possible world by the probability of the proposition, shown in lines 265–287 of `main.py`. Then, for each possible world $w \in W$, it applies this function to the fuzzy proposition $\tilde{A}$ by the `apply_numeric` method, described above, to compute the conditional probability distribution $P(w \mid \tilde{A})$. This is shown in lines 289–291.

Test cases are shown in lines 220–315 of `main_test.py`. In order to correspond to eq. (5), P and $\tilde{A}$ must be defined for the same, non-empty set of possible worlds (lines 237–273), and P must be a probability distribution (lines 289–301). The other cases demonstrate the circumstances in which $P(w \mid \tilde{A})$ is not well-defined:

- If $\tilde{A}_1 = \emptyset$, i.e., the fuzzy proposition does not contain a possible world with membership value 1, then the result is not a probability distribution because it does not sum to 1 (lines 275–287).
- If $\exists\, \alpha : P(w \mid \tilde{A}_\alpha) = 0$, i.e., the sum of the probabilities of the possible worlds in an α-cut is zero, then the result is undefined because it contains a division by zero (lines 303–315).

In Dempster–Shafer theory, the posterior probability distribution $P(A \mid m)$ is defined as follows. Let $P : W \to [0,1]$ be a probability distribution, $m : 2^W \to [0,1]$ be a mass function, and A and B be propositions. Then, for all $A \subseteq W$:

$$P(A \mid m) = \sum_{B \subseteq W} P(A \mid B)\, m(B) \tag{7}$$

$$= \sum_{B \subseteq W} \frac{P(A \cap B)}{P(B)}\, m(B) \tag{8}$$

Hence, the conditional probability distribution $P(w \mid \tilde{A})$ is a special case of the posterior probability distribution $P(A \mid m)$, where A is a singleton set, and m is a mass function that assigns, for each α-cut of $\tilde{A}$, a mass of the size of the corresponding α-value interval to the set of elements of the α-cut, and zero mass to all other propositions $B \subseteq W$.

A Python code

main.py

```python
"""Implementation for 'An Investigation into Fuzzy Systems' Q2."""

from collections.abc import Hashable, Set
from itertools import pairwise
from typing import Callable, Iterator

# The alpha-cuts of a fuzzy set, represented as a map from sets of
# elements to intervals of alpha-values.
type AlphaCuts[Element: Hashable] = dict[
    Set[Element], tuple[float, float]
]


class FuzzySet[Element: Hashable](Set[Element], Hashable):
    """
    A discrete fuzzy set with a finite number of elements.

```

```
    Parameters
    ----------
    elements
        A set of elements.
    membership_function
        A map from elements to their membership values.
    """

    def __init__(
        self,
        elements: Set[Element],
        membership_function: dict[Element, float],
    ):
        self.elements = elements
        self.membership_function = membership_function

    def __contains__(self, element: Element) -> bool:
        return element in self.elements

    def __hash__(self) -> int:
        return hash(self.elements)

    def __iter__(self) -> Iterator[Element]:
        return iter(self.elements)

    def __len__(self) -> int:
        return len(self.elements)

    def alpha_cut(self, alpha: float) -> Set[Element]:
        """
        Q2(a). Given an alpha value, returns the alpha-cut of the fuzzy
        set.

        Parameters
        ----------
        alpha
            An alpha value.

        Return
        ----------
        alpha_cut
            The alpha-cut of the fuzzy set.
        """
        return frozenset(
            {
                element
```

```
                for element in self.elements
                if self.membership_function[element] >= alpha
            }
        )

    def alpha_cuts(self) -> AlphaCuts[Element]:
        """
        Q2(a). Returns the alpha-cuts of the fuzzy set.

        Return
        ----------
        alpha_cuts
            The alpha-cuts of the fuzzy set.
        """
        values = [0.0] + sorted(set(self.membership_function.values()))
        return {
            self.alpha_cut(upper): (lower, upper)
            for [lower, upper] in pairwise(values)
        }

    @staticmethod
    def from_alpha_cuts(alpha_cuts: AlphaCuts[Element]):
        """
        Q2(b). Given a set of alpha-cuts, returns the fuzzy set.

        Parameters
        ----------
        alpha_cuts
            A set of alpha-cuts.

        Return
        ----------
        from_alpha_cuts
            The fuzzy set.
        """
        membership_function = dict[Element, float]()
        for elements, (_lower, upper) in alpha_cuts.items():
            for element in elements:
                membership_function[element] = upper
        return FuzzySet(set(membership_function), membership_function)

    def apply_elementwise[
        Result: Hashable
    ](self, function: Callable[[Element], Result]):
        """
        Q2(c). Given an element-wise function f : W -> W, returns the
```

```
        fuzzy set of results of applying the function.

        Parameters
        ----------
        function
            An element-wise function f : W -> W.

        Return
        ----------
        apply_elementwise
            The fuzzy set of results of applying the function.
        """
        return FuzzySet.from_alpha_cuts(
            merge_alpha_cuts(
                {
                    apply_elementwise(elements, function): (
                        lower,
                        upper,
                    )
                    for elements, (
                        lower,
                        upper,
                    ) in self.alpha_cuts().items()
                }
            )
        )

    def apply_numeric(
        self, function: Callable[[Set[Element]], float]
    ) -> float:
        """
        Given a function f : 2^W -> R, returns the result of applying the
        function to the fuzzy set.

        Parameters
        ----------
        function
            A function f : 2^W -> R.

        Return
        ----------
        apply_numeric
            The result of applying the function to the fuzzy set.
        """
        return sum(
            (upper - lower) * function(elements)
```

```
                for elements, (
                    lower,
                    upper,
                ) in self.alpha_cuts().items()
            )


def merge_alpha_cuts[
    Element: Hashable
](alpha_cuts: AlphaCuts[Element]) -> AlphaCuts[Element]:
    """
    Given alpha-cuts with duplicate elements, returns the merged
    alpha-cuts.

    Parameters
    ----------
    alpha_cuts
        The alpha-cuts.

    Return
    ----------
    merge_alpha_cuts
        The merged alpha-cuts.
    """
    merged: AlphaCuts[Element] = {}
    for elements, (lower, upper) in alpha_cuts.items():
        if elements in merged:
            merged[elements] = (
                min(lower, merged[elements][0]),
                max(upper, merged[elements][1]),
            )
        else:
            merged[elements] = (lower, upper)
    return merged


def apply_elementwise[
    Argument: Hashable, Result: Hashable
](
    elements: Set[Argument],
    function: Callable[[Argument], Result],
) -> Set[Result]:
    """
    Q2(c). Given a set of elements and an element-wise function f : W -> W,
    returns the set of results of applying the function to the elements.

```

```
    Parameters
    ----------
    elements
        A set of elements.
    function
        An element-wise function f : W -> W.

    Return
    ----------
    apply_elementwise
        The set of results of applying the function to the elements.
    """
    return frozenset({function(element) for element in elements})


def fuzzy_cond_prob_dist[
    PossibleWorld: Hashable
](
    prob_dist: dict[PossibleWorld, float],
    fuzzy_prop: FuzzySet[PossibleWorld],
) -> dict[PossibleWorld, float]:
    """
    Q2(d). Given a probability distribution over a finite set of possible
    worlds and a fuzzy proposition over the same set of possible worlds,
    returns the conditional probability distribution given the fuzzy
    proposition.

    Parameters
    ----------
    prob_dist
        A probability distribution over a finite set of possible worlds.
    fuzzy_prop
        A fuzzy proposition over the same set of possible worlds.

    Return
    ----------
    fuzzy_cond_prob_dist
        The conditional probability distribution given the fuzzy
        proposition.
    """

    if not prob_dist or not fuzzy_prop:
        raise ValueError("prob_dist and fuzzy_prop must be non-empty.")

    if set(prob_dist.keys()) != fuzzy_prop.elements:
        raise ValueError(
```

```
            (
                "prob_dist and fuzzy_prop must be defined for the same "
                "possible worlds."
            )
        )

    if 1.0 not in fuzzy_prop.membership_function.values():
        raise ValueError(
            (
                "fuzzy_prop must contain a possible world with membership"
                "value 1."
            )
        )

    if sum(prob_dist.values()) != 1.0:
        raise ValueError("prob_dist must have total probability 1.")

    def sum_prob(ws: Set[PossibleWorld]) -> float:
        """
        Given a set of possible worlds, returns the sum of their
        probabilities.
        """
        return sum(prob_dist[w] for w in ws)

    def make_cond_prob(
        w: PossibleWorld,
    ) -> Callable[[Set[PossibleWorld]], float]:
        """
        Given a possible world, returns a function that returns its
        conditional probability given a crisp proposition.
        """

        def cond_prob(ws: Set[PossibleWorld]) -> float:
            if sum_prob(ws) == 0.0:
                raise ValueError(
                    f"{set(ws)} must have non-zero total probability."
                )
            return (prob_dist[w] if w in ws else 0.0) / sum_prob(ws)

        return cond_prob

    return {
        w: fuzzy_prop.apply_numeric(make_cond_prob(w)) for w in prob_dist
    }
```

main_test.py

```python
"""Test cases for 'An Investigation into Fuzzy Systems' Q2."""

# pylint: disable=missing-function-docstring, missing-class-docstring

from pytest import approx, raises  # type: ignore

from main import FuzzySet, apply_elementwise, fuzzy_cond_prob_dist

# A set of integer elements from 1 to 11.
test_elements = set(range(1, 12))

# A membership function that maps the above elements to unique values.
test_membership = {
    element: min(element / 10, 1.0) for element in range(1, 12)
}

# A membership function that maps the above elements to non-unique values.
test_membership_duplicates = {
    1: 0.1,
    2: 0.1,
    3: 0.3,
    4: 0.3,
    5: 0.5,
    6: 0.5,
    7: 0.7,
    8: 0.7,
    9: 0.9,
    10: 0.9,
    11: 1.0,
}


class TestFuzzySet:
    def test_alpha_cut(self):
        fuzzy_set = FuzzySet(test_elements, test_membership)
        assert fuzzy_set.alpha_cut(0.1) == set(range(1, 12))
        assert fuzzy_set.alpha_cut(0.2) == set(range(2, 12))
        assert fuzzy_set.alpha_cut(0.3) == set(range(3, 12))
        assert fuzzy_set.alpha_cut(0.4) == set(range(4, 12))
        assert fuzzy_set.alpha_cut(0.5) == set(range(5, 12))
        assert fuzzy_set.alpha_cut(0.6) == set(range(6, 12))
        assert fuzzy_set.alpha_cut(0.7) == set(range(7, 12))
        assert fuzzy_set.alpha_cut(0.8) == set(range(8, 12))
        assert fuzzy_set.alpha_cut(0.9) == set(range(9, 12))
        assert fuzzy_set.alpha_cut(1.0) == set(range(10, 12))
```

```python

    def test_alpha_cut_duplicates(self):
        fuzzy_set = FuzzySet(test_elements, test_membership_duplicates)
        assert fuzzy_set.alpha_cut(0.1) == set(range(1, 12))
        assert fuzzy_set.alpha_cut(0.3) == set(range(3, 12))
        assert fuzzy_set.alpha_cut(0.5) == set(range(5, 12))
        assert fuzzy_set.alpha_cut(0.7) == set(range(7, 12))
        assert fuzzy_set.alpha_cut(0.9) == set(range(9, 12))

    def test_alpha_cut_lecture_notes(self):
        """Example 5.3.2 from the lecture notes."""
        fuzzy_set = FuzzySet({4, 5, 6}, {4: 0.3, 5: 0.7, 6: 1.0})
        assert fuzzy_set.alpha_cut(0.3) == {4, 5, 6}
        assert fuzzy_set.alpha_cut(0.7) == {5, 6}
        assert fuzzy_set.alpha_cut(1.0) == {6}

    def test_alpha_cuts(self):
        assert FuzzySet(test_elements, test_membership).alpha_cuts() == {
            frozenset(range(1, 12)): (0.0, 0.1),
            frozenset(range(2, 12)): (0.1, 0.2),
            frozenset(range(3, 12)): (0.2, 0.3),
            frozenset(range(4, 12)): (0.3, 0.4),
            frozenset(range(5, 12)): (0.4, 0.5),
            frozenset(range(6, 12)): (0.5, 0.6),
            frozenset(range(7, 12)): (0.6, 0.7),
            frozenset(range(8, 12)): (0.7, 0.8),
            frozenset(range(9, 12)): (0.8, 0.9),
            frozenset(range(10, 12)): (0.9, 1.0),
        }

    def test_alpha_cuts_duplicates(self):
        assert FuzzySet(
            test_elements, test_membership_duplicates
        ).alpha_cuts() == {
            frozenset(range(1, 12)): (0.0, 0.1),
            frozenset(range(3, 12)): (0.1, 0.3),
            frozenset(range(5, 12)): (0.3, 0.5),
            frozenset(range(7, 12)): (0.5, 0.7),
            frozenset(range(9, 12)): (0.7, 0.9),
            frozenset(range(11, 12)): (0.9, 1.0),
        }

    def test_alpha_cuts_lecture_notes(self):
        """Example 5.3.2 from the lecture notes."""
        assert FuzzySet(
            {4, 5, 6}, {4: 0.3, 5: 0.7, 6: 1.0}
```

```python
        ).alpha_cuts() == {
            frozenset({4, 5, 6}): (0.0, 0.3),
            frozenset({5, 6}): (0.3, 0.7),
            frozenset({6}): (0.7, 1.0),
        }

    def test_from_alpha_cuts(self):
        fuzzy_set = FuzzySet.from_alpha_cuts(
            {
                frozenset(range(1, 12)): (0.0, 0.1),
                frozenset(range(2, 12)): (0.1, 0.2),
                frozenset(range(3, 12)): (0.2, 0.3),
                frozenset(range(4, 12)): (0.3, 0.4),
                frozenset(range(5, 12)): (0.4, 0.5),
                frozenset(range(6, 12)): (0.5, 0.6),
                frozenset(range(7, 12)): (0.6, 0.7),
                frozenset(range(8, 12)): (0.7, 0.8),
                frozenset(range(9, 12)): (0.8, 0.9),
                frozenset(range(10, 12)): (0.9, 1.0),
            }
        )
        assert fuzzy_set.elements == test_elements
        assert fuzzy_set.membership_function == test_membership

    def test_from_alpha_cuts_duplicates(self):
        fuzzy_set = FuzzySet.from_alpha_cuts(
            {
                frozenset(range(1, 12)): (0.0, 0.1),
                frozenset(range(3, 12)): (0.1, 0.3),
                frozenset(range(5, 12)): (0.3, 0.5),
                frozenset(range(7, 12)): (0.5, 0.7),
                frozenset(range(9, 12)): (0.7, 0.9),
                frozenset(range(11, 12)): (0.9, 1.0),
            }
        )
        assert fuzzy_set.elements == test_elements
        assert fuzzy_set.membership_function == test_membership_duplicates

    def test_from_alpha_cuts_lecture_notes(self):
        """Example 5.3.3 from the lecture notes."""
        fuzzy_set = FuzzySet.from_alpha_cuts(
            {
                frozenset({4, 5, 6}): (0.0, 0.3),
                frozenset({5, 6}): (0.3, 0.7),
                frozenset({6}): (0.7, 1.0),
            }
```

```
        )
        assert fuzzy_set.elements == {4, 5, 6}
        assert fuzzy_set.membership_function == {4: 0.3, 5: 0.7, 6: 1.0}

    def test_apply_elementwise_one_to_one(self):
        fuzzy_set = FuzzySet(
            test_elements, test_membership
        ).apply_elementwise(lambda element: element**2)
        assert fuzzy_set.elements == {
            1,
            4,
            9,
            16,
            25,
            36,
            49,
            64,
            81,
            100,
            121,
        }
        assert fuzzy_set.membership_function == {
            1: 0.1,
            4: 0.2,
            9: 0.3,
            16: 0.4,
            25: 0.5,
            36: 0.6,
            49: 0.7,
            64: 0.8,
            81: 0.9,
            100: 1.0,
            121: 1.0,
        }

    def test_apply_elementwise_many_to_one(self):
        fuzzy_set = FuzzySet(
            test_elements, test_membership
        ).apply_elementwise(lambda element: element // 2)
        assert fuzzy_set.elements == {0, 1, 2, 3, 4, 5}
        assert fuzzy_set.membership_function == {
            0: 0.1,
            1: 0.3,
            2: 0.5,
            3: 0.7,
            4: 0.9,
```

```
            5: 1.0,
        }

    def test_apply_elementwise_lecture_notes(self):
        """Example 5.4.2 from the lecture notes."""
        fuzzy_set = FuzzySet(
            {4, 5, 6}, {4: 0.3, 5: 0.7, 6: 1.0}
        ).apply_elementwise(
            lambda element: 6 if element == 1 else element - 1
        )
        assert fuzzy_set.elements == {3, 4, 5}
        assert fuzzy_set.membership_function == {
            3: 0.3,
            4: 0.7,
            5: 1.0,
        }

    def test_apply_numeric_lecture_notes(self):
        """Example 5.4.1 from the lecture notes."""
        assert (
            FuzzySet({4, 5, 6}, {4: 0.3, 5: 0.7, 6: 1.0}).apply_numeric(
                len
            )
            == 2.0
        )


def test_apply_elementwise():
    assert apply_elementwise(
        set(range(1, 11)), lambda element: element**2
    ) == {1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
    assert apply_elementwise(
        set(range(1, 11)), lambda element: element // 2
    ) == {0, 1, 2, 3, 4, 5}


class TestFuzzyConditional:
    def test_valid(self):
        p = fuzzy_cond_prob_dist(
            {1: 0.1, 2: 0.2, 3: 0.3, 4: 0.4},
            FuzzySet(
                {1, 2, 3, 4},
                {1: 1.0, 2: 0.9, 3: 0.8, 4: 0.7},
            ),
        )
        assert sum(p.values()) == 1.0
```

```
        assert approx(p) == {
            1: 0.22,
            2: 0.24,
            3: 0.26,
            4: 0.28,
        }

    def test_invalid_empty(self):
        with raises(ValueError) as excinfo:
            fuzzy_cond_prob_dist(
                dict[int, float](),
                FuzzySet(
                    {1, 2, 3, 4},
                    {1: 0.9, 2: 0.9, 3: 0.8, 4: 0.7},
                ),
            )
        assert (
            str(excinfo.value)
            == "prob_dist and fuzzy_prop must be non-empty."
        )

        with raises(ValueError) as excinfo:
            fuzzy_cond_prob_dist(
                {1: 0.1, 2: 0.2, 3: 0.3, 4: 0.4},
                FuzzySet(set[int](), dict[int, float]()),
            )
        assert (
            str(excinfo.value)
            == "prob_dist and fuzzy_prop must be non-empty."
        )

    def test_invalid_elements(self):
        with raises(ValueError) as excinfo:
            fuzzy_cond_prob_dist(
                {1: 0.1, 2: 0.2, 3: 0.3, 4: 0.4},
                FuzzySet(
                    {1, 2, 3},
                    {1: 0.9, 2: 0.9, 3: 0.8},
                ),
            )
        assert str(excinfo.value) == (
            "prob_dist and fuzzy_prop must be defined for the same "
            "possible worlds."
        )

    def test_invalid_membership(self):
```

```python
        with raises(ValueError) as excinfo:
            fuzzy_cond_prob_dist(
                {1: 0.1, 2: 0.2, 3: 0.3, 4: 0.4},
                FuzzySet(
                    {1, 2, 3, 4},
                    {1: 0.9, 2: 0.9, 3: 0.8, 4: 0.7},
                ),
            )
        assert str(excinfo.value) == (
            "fuzzy_prop must contain a possible world with membership"
            "value 1."
        )

    def test_invalid_prob_dist(self):
        with raises(ValueError) as excinfo:
            fuzzy_cond_prob_dist(
                {1: 0.1, 2: 0.2, 3: 0.3, 4: 0.3},
                FuzzySet(
                    {1, 2, 3, 4},
                    {1: 1.0, 2: 0.9, 3: 0.8, 4: 0.7},
                ),
            )
        assert (
            str(excinfo.value)
            == "prob_dist must have total probability 1."
        )

    def test_invalid_zero_probability(self):
        with raises(ValueError) as excinfo:
            fuzzy_cond_prob_dist(
                {1: 0.0, 2: 0.2, 3: 0.4, 4: 0.4},
                FuzzySet(
                    {1, 2, 3, 4},
                    {1: 1.0, 2: 0.9, 3: 0.8, 4: 0.7},
                ),
            )
        assert (
            str(excinfo.value)
            == "{1} must have non-zero total probability."
        )
```