ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

BÁNKI DONÁT GÉPÉSZ ÉS
BIZTONSÁGTECHNIKAI MÉRNÖKI KAR
Mechatronikai és Járműtechnikai Intézet

FUZZY HIBAFA ELJÁRÁS FEJLESZTÉSE
FUZZY FAULT TREE METHOD DEVELOPMENT

OE-BGK
2021.12.12

Hallgató neve: Végh Benjamin Bence
Hallgató törzskönyvi száma: T005735/FI12904/B

- REMOVED PAGE CONTAINING SIGNATURES -

- REMOVED PAGE CONTAINING SIGNATURES -

Table of Contents

1 INTRODUCTION

As self-driving vehicles become more widely adopted it is becoming increasingly important to do thorough risk analysis of not just individual hardware components but software components as well, neural network models especially to assess their overall performance in various traffic conditions and individual intersections.

Fault Tree Analysis has often been used in conjunction with other methods to assess component-level reliability and identify key components which contribute the most for the overall system failure. However it assumes that each component has a well defined failure probability which is often not the case and instead estimates have to be used, which introduces uncertainties into the system model.

Fuzzy Fault Tree Analysis was later developed to quantify these uncertainties by applying Fuzzy Logic to traditional Fault Tree Analysis, which has had promising results in various fields of study. It however has very few if any published research applying it to the field of self-driving vehicles.

Unfortunately there is a lack of freely available software in the field of Fuzzy Fault Tree Analysis to serve as a basis of research. The only such example the institution has access to is the TREEZZY2 software which does not have either documentation or source-code bundled with it and neither is it possible to use it in embedded applications easily, making it unsuitable to be applied to the field of self-driving vehicles.

For this reason in this thesis a new computer software was developed and documented to serve as a basis for further development and research under a permissive license, designed for embedded applications using the popular Python programming language.

This thesis also contains brief overviews of the areas of both Traditional and Fuzzy Fault Tree Analysis to introduce the theory and logic used, as well as an introduction and overview of the Python programming language and it's syntax to help provided a basis of understanding for the source code for when it is used, extended and improved.

2 FAULT TREE ANALYSIS

2.1 Traditional Fault Tree Analysis

The Fault Tree Analysis, or FTA, is a qualitative and quantitative method which depicts the logical relationships between basic events and a top event using logic gates. [1]

The top event is the currently analysed event, such as a system failure event or other undesired event occurring. The base event is the smallest analysed event that contribute to the top event in some meaningful way, such as a component failure in machinery. Finally, logic gates are used to establish the relationships between base events and the top event. A logic gate takes a number of events as input, uses some logic operation (AND, OR, NOT) and outputs an intermediate event, the last of such intermediate event is the top event. [1]

This method uses bivalent (or Boolean) logic and while the Tree can have many other types of elements the simplest form is made with Base Events, Logic Gates which output Intermediate Events and the Top Event. Due to the events being Bivalent, either occurring or not, the tree can be thought as a graphical representation of the underlying Boolean (or bivalent) logic, as such Boolean algebra can be applied to the Fault Tree Analysis. [1]

Figure 1: Symbols of the AND logic gate (a.) and the OR logic gate (b.)

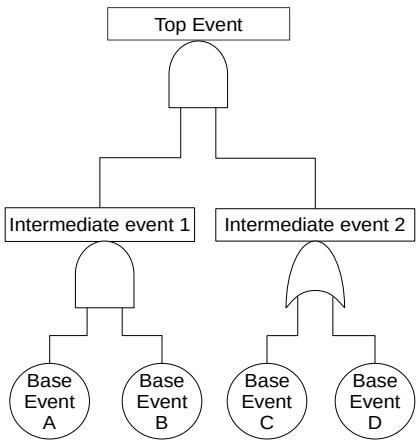

Figure 2: An example of how the Fault Tree is constructed

The qualitative analysis of Fault Trees is done by obtaining the Minimal Cut Sets, which is the smallest combination of Base Events that cause the Top Event to occur. Any Fault Tree has a finite number of these Minimal Cut Sets. [1]

Acquiring these Minimal Cut Sets is done by first translating the Fault Tree into it's equivalent Boolean Algebraic Expression, then using Boolean Algebra to get the Sum of Products (SOP) representation of the fault tree, where each product in the chain of sums is a single cut-set.

After the qualitative analysis was performed on a given Fault Tree, quantitative analysis can be performed. This type of analysis is done by assigning an estimated probability of occurrence to each base event. These can then be used to estimate the top event occurrence and the importance of each base event. [1]

2.2 Uncertainty in Fault Tree Analysis

Due to Traditional Fault Tree Analysis using bivalent logic, events need to be defined exactly. Failing to do so, or even working with rough estimates of ill-defined events will result in uncertainties in the analysis results. This is due to the fact that in quantitative analysis we estimate the probability distribution of Base Events occurring based on a large quantity of data gathered, some expert's opinions in the field, simulation or other methods. Each of these have their own uncertainties or subjectivity which will be propagated to the overall Top Event probability. This problem is exasperated in cases where data cannot be gathered in sufficient quantities, like a rare component failure in a factory for instance. [2][3]

To combat or at least quantify these uncertainties, several approaches were developed. One such approach is to incorporate Fuzzy Set Theory, often times with Linguistic Variables, to incorporate the inaccuracy and subjectivity in base event probability estimates, expert's opinions and the lack of sufficient data. This approach has been used with promising results in modelling our uncertainties in base event probabilities that can then be propagated to the Top Event probability for further analysis. [1][2][3]

3 SETS AND LOGIC

3.1 Traditional Sets and Logic

3.1.1 Sets

A set in the traditional sense is a collection of elements, which can be any kind of mathematical object or number, even other sets. Such elements that are inside a given set can be called members of that set.

Traditional sets use bivalent logic when determining set membership, that is a particular object or number is either a member of the set or not. [3]

A mathematical function that tests an object or number's membership to a particular set is called the set's membership function.

Sets have a few basic set operations for constructing new sets from existing ones, such as in example: Union (Figure 3.c) , Intersection (Figure 3.b) and Complement (Figure 3.a). These constitute the minimal set of operations to accomplish all traditional set logic.

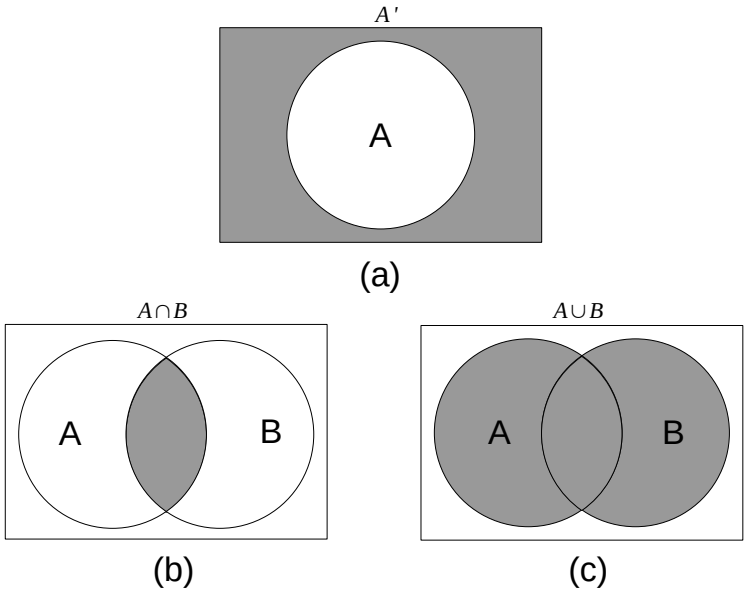

Figure 3: Venn-Diagrams of common Set Operations

3.1.2 Logic

"Logic studies the methods and principles of reasoning." [4]

Traditional Logic works with Propositions. A proposition can be either be True or False. This Propositional Logic describes ways to handle combinations of logical variables with a set of logical primitives, such a combination of logical primitives and variables is called a logic function. A set of logic primitives is complete if any such logic function can be composed by a finite number of these logic primitives. This propositional logic, provided that it is based on a finite set of logic variables, is isomorphic to a finite Boolean algebra, both of these systems are also isomorphic to a finite set theory. [4]

Propositional Logic		Boolean Algebra		Finite Set Theory	
Symbol	Name	Symbol	Name	Symbol	Name
∧	Conjuction	· or &	AND	∩	Intersection
∨	Disjunction	+ or ||	OR	U	Union
¬	Negation	◌̄ or !	NOT	◌'	Complement

Table 1: Equivalent logical operation symbols and their names for Propositional logic, Boolean algebra and Finite Set Theory

Propositional logic is said to be in normal form only if it contains statement (or logical) variables and its negations along with operators ∧ and ∨. These normal forms may be conjunctive or disjunctive and both of these can also be canonical. The disjunctive normal form (DNF) is a disjunction of general conjunction sub-expressions, while the conjunctive normal form (CNF) is a conjunction of general disjunction sub-expressions. Both CNF and DNF can be canonical, which means minimizing the count of sub-expressions while ensuring each sub-expression contains all statement variables. Most importantly, every proposition (or logical expression) has an equivalent conjunctive normal form and an equivalent disjunctive normal form. [5]

An example for disjunctive normal form (DNF) :

$$(A \wedge B) \vee (C \wedge \neg B) \vee (\neg A \wedge \neg C \wedge D) \tag{3.1}$$

An example for conjunctive normal form (CNF) :

$$(\neg C \vee A) \wedge (D \vee B) \wedge (\neg A \vee D) \tag{3.2}$$

In Boolean algebra the disjunctive normal form (DNF) is also referred to as sum-of-products (SOP) while the conjunctive normal form (CNF) is referred to as product-of-sums (POS). These two forms are useful for the simplification of boolean functions. As with propositional logic each boolean function has an equivalent POS and SOP representation.

To illustrate the the two forms in boolean algebra let us take the following boolean function:

$$A + (D \cdot (B+C)) \cdot (B + (D \cdot C)) \tag{3.3}$$

The equivalent simplified SOP representation of it is:

$$A + (B \cdot D) + (C \cdot D) \tag{3.4}$$

While the equivalent simplified POS representation of it is:

$$(A + B + C) \cdot (A + D) \tag{3.5}$$

3.2 Fuzzy Numbers

3.2.1 Fuzzy Sets

"A fuzzy set is a class of objects with a continuum of grades of membership. Such a set is characterized by it's membership (characteristic) function which assigns each object a grade of membership ranging between zero and one." [6]

In traditional sets bivalent, or two valued, logic is used to determine if an object or number is part of a set or not. As such membership functions of traditional sets gives a clear, True or False answer to the question 'is X a member of Y set', or 1 and 0 respectively.

In contrast to traditional sets, Fuzzy Sets describe membership functions by which each object or number can be tested for membership and given a degree of membership ' μ_A ' which is a real number in the interval $[0,1]$. [6]

Fuzzy Sets behave similarly to Traditional Sets in terms of set operations. Indeed L. A. Zadeh has shown that the basic identities of the Traditional Sets can be extended to Fuzzy Sets as well, such as De Morgan's laws and Distributive Laws. [6]

3.2.2 Fuzzy Membership Functions

Usually in fuzzy sets, membership functions are considered to be Triangular or Trapeziodal in shape, even if other shapes are used. [3]

Describing a Fuzzy Set by it's membership function is called the **Vertical Representation**.

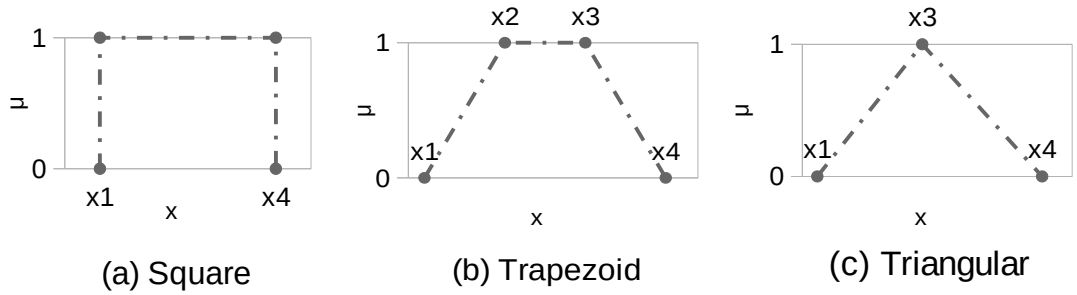

Figure 4: Vertical Representation of Common Fuzzy Sets

Let $x, x_1, x_2, x_3, x_4 \epsilon R$, a fuzzy number ' A ' can be described by it's membership function $f_A(x)=\mu_A$, $\mu_A: R \rightarrow [0,1]$.

In vertical representation the Fuzzy Set is described by it's membership function.

A Trapezoid fuzzy number ' A ' as seen in Figure 4.b can be described as follows :

$$f_A(x)=\begin{cases} \dfrac{x-x_1}{x_2-x_1} & for\ x_1 \le x \le x_2 \\ 1 & for\ x_2 \le x \le x_3 \\ \dfrac{x-x_4}{x_3-x_4} & for\ x_3 \le x \le x_4 \\ 0 & otherwise \end{cases} \tag{3.6}$$

Assuming $x_1 < x_2 < x_3 < x_4$ a Trapezoid fuzzy number A can be denoted by $A=[x_1; x_2; x_3; x_4]$.

A triangular fuzzy number, as seen in Figure 4.c , can be described by an equivalent Trapezoid fuzzy number where $x_2=x_3$ and thus we can simplify the previous formula as follows :

$$f_A(x)=\begin{cases} \dfrac{x-x_1}{x_2-x_1} & for\ x_1 \le x \le x_2 \\ \dfrac{x-x_4}{x_2-x_4} & for\ x_2 \le x \le x_4 \\ 0 & otherwise \end{cases} \tag{3.7}$$

assuming $x_1 < x_2 < x_4$ and $x_2=x_3$ a Triangular fuzzy number A can be denoted by $A=[x_1; x_2; x_4]$

A Traditional Set can be approximated by using a Square fuzzy number, as seen in Figure 4.a , can be described by an equivalent Trapezoid fuzzy number where $x_1=x_2$ and $x_3=x_4$, assuming $x_1 < x_4$ we can simplify it as follows :

$$f_A(x)=\begin{cases} 1 & for\ \ x_1 \le x \le x_4 \\ 0 & otherwise \end{cases} \tag{3.8}$$

3.2.3 Alpha Cuts

The alpha-cut method method, also known as resolution identity, can be applied to Fuzzy Sets when there is a need for a more crisp set in order to simplify the formulas and computation. Thusly, using the alpha-cut method we restrict a Fuzzy Set to discrete membership values. A Fuzzy Set defined in terms of it's Alpha-Cuts is called that set's **Horizontal Representation**.[4]

Let ' A ' be a Fuzzy Set with membership function $f_A(x)$, it's alpha-cut (or alpha-level set) is denoted by ' A_α ' where $\alpha \ \epsilon \ [0,1]$ is the cut's alpha-level, also called the level of confidence. The alpha-level set of ' A ' is defined as:

$$A_\alpha = \{ \ x \epsilon X \ | \ f_A(x) \geq \alpha \ \} \tag{3.9}$$

The strong alpha-cut (or strict alpha-level set) as:

$$A_\alpha = \{ \ x \epsilon X \ | \ f_A(x) > \alpha \ \} \tag{3.10}$$

Where ' X ' is the set of all values ' x ' can take.

In other terms, an alpha-cut takes a Fuzzy Set and creates a traditional set called that set's Alpha-Level Set, by restricting the values that the degree-of-membership ' μ ' can assume to a discrete bivalent set, which is written mathematically as: $\mu \epsilon \{0, \alpha\}$.

The membership function changes for an alpha-cut set to be:

$$A_\alpha = \begin{cases} \mu_{A_\alpha} = \alpha \ \ if \ \ f_A(x) \geq \alpha \\ \mu_{A_\alpha} = 0 \qquad otherwise \end{cases} \tag{3.11}$$

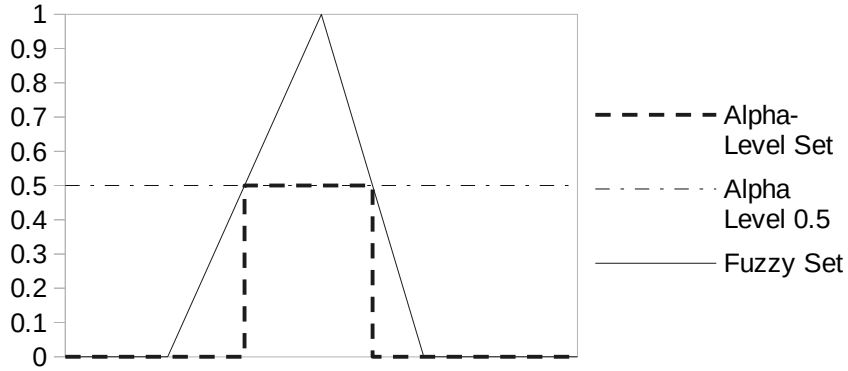

Figure 5: The Alpha-Level Set of a Triangular Fuzzy Set at the 0.5 Alpha Level

Describing a Fuzzy Set by it's Alpha-Cut's is done in the following way:

Let ' L ' be a set of alpha-levels the cuts will be made at so that $\alpha \epsilon L$. Let ' A ' be a Fuzzy Set to be described. Suppose that $X \epsilon R$, $x \epsilon X$ where X is the interval on which we work with the Fuzzy Set and ' x ' is a point within that interval. The Alpha-Level Set, if it is continuous at the chosen Alpha-Level can be written as $A_\alpha = [x_L \ , \ x_U]$, or if it is discontinuous at the given alpha-level $A_\alpha = [x_{L(1)} \ , \ x_{U(1)}] \cup [x_{L(2)} \ , \ x_{U(2)}] \ ... \ [x_{L(i-1)} \ , \ x_{U(i-1)}] \cup [x_{L(i)} \ , \ x_{U(i)}]$ where x_L , x_U are the lower and upper bounds of the interval where $f_A(x) \geq \alpha$.

In a concrete example, let's assume we describe the fuzzy number ' B ' and that $X = [0, 100]$, $L = [0, 0.5, 1]$. The Alpha-Level sets at each Alpha Level are as follows:

$$B_0 = [10, 80]$$
$$B_{0.5} = [10, 30] \cup [60, 80]$$
$$B_1 = \{15\} \cup [60, 70] \cup [75, 80]$$

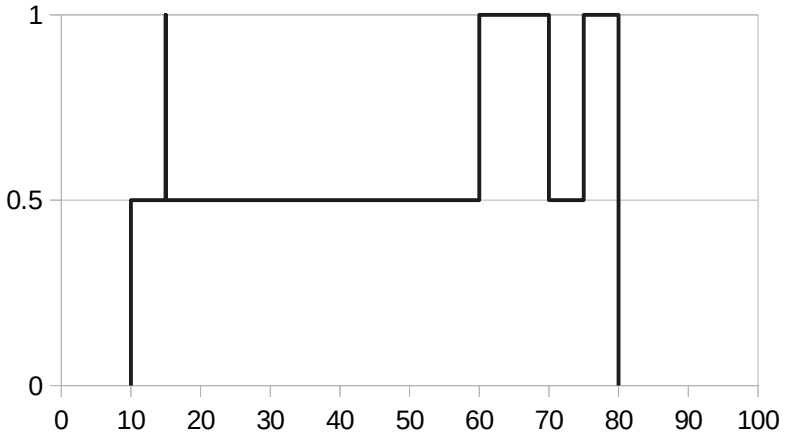

Figure 6: Upper Envelope of the Alpha-Level Sets of Fuzzy Set B

3.2.4 Fuzzy Logic

For Fuzzy Logic, the complete minimal set of logic operations as described by Zadeh is {AND, OR, NOT}. [4]

For the Vertical Representation that uses membership functions to describe Fuzzy Sets these basic logic operations can be described as follows:

Let ' A, B ' be a Fuzzy Number with membership functions $f_A(x), f_B(x)$ where $x \epsilon R$ and ' C ' is the resultant Fuzzy Number from the operations, the logic operations can be written as follows:

Complement (logical NOT): $C = \neg A = 1 - f_A(x)$ or $C = \neg B = 1 - f_B(x)$ (3.12)

Intersect (logical AND): $C = A \cap B = min(\, f_A(x)\, ,\, f_B(x)\,)$ (3.13)

Union (logical OR): $C = A \cup B = max(\, f_A(x)\, ,\, f_B(x)\,)$ (3.14)

For the Horizontal Representation where we describe a Fuzzy Set by it's Alpha-Level Sets it is much simpler as those are just unions of intervals at each Alpha-Level. As such traditional set logic can be used on them, however this has to be done at each Alpha-Level. As such the logic operations are as follows:

Complement (logical NOT): $C_\alpha = \neg A_\alpha$ (3.15)

for each Alpha-Level α of ' A '

Intersect (logical AND): $C_\alpha = A_\alpha \cap B_\alpha$ (3.16)

for each Alpha-Level α of ' A, B '

Union (logical OR): $C_\alpha = A_\alpha \cup B_\alpha$ (3.17)

for each Alpha-Level α of ' A, B '

4 FUZZY FAULT TREE ANALYSIS

4.1 Fuzzy Set Theory in Fault Tree Analysis

In Fuzzy Fault Tree Analysis the probabilities of Events are modeled with a Fuzzy Number. Using the Fuzzy Set theory we can use similar Logic Gates as in Traditional Fault Tree Analysis to build up the Tree. The Intermediate Events and Top Event will also be Fuzzy Numbers with the uncertainties in Base Event probabilities propagated up the tree to the Top Event. Thus we can not only estimate the Top Event probability but also assess our uncertainty in that estimate.

Acquiring point-wise probability estimates of the Top Event is straight forward with all kinds of Fuzzy Number shapes. However, to perform meaningful analysis some assumptions have to be made to simplify the underlying calculations and algebraic formulas.

4.2 Membership Functions and Linguistic Variables

The membership functions for the Fuzzy Sets are uniform in shape for all Events, usually Trapezoidal or Triangular, which simplifies combining them with the Logic Gates and reduces the complexity of calculating each event's Alpha-Cuts.

The Alpha-Level set, or alpha-cut, can be calculated for a Trapezoidal Fuzzy Number as a closed interval with a lower and upper bound defined as:

$$x_1 \leq L \leq x_2 \ , \ x_3 \leq U \leq x4 \tag{4.1}$$

$$\begin{aligned} L &= x_1 + \alpha * (x_2 - x_1) \\ U &= x_4 - \alpha * (x_4 - x_3) \end{aligned} \tag{4.2}$$

Where ' L , U ' are the lower and upper bounds of the Alpha-Level Set, x_1 , x_2 , x_3 , x_4 are the key points of the Trapezoidal fuzzy number and ' α ' is the Alpha-Level.

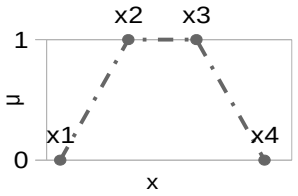

Figure 7: Membership Function of a Trapezoidal Fuzzy Number

Fuzzy Number's with Triangular shape can be calculated using the same formulas as a Trapezoidal one with the exception that $x_2 = x_3$.

However, defining the membership function for each Base Event can be time consuming and require the knowledge and understanding of Fuzzy Set Theory. Thus when working with raw numbers it can be hard to visualize the difference between one set of values over another set of values. For that reason

Linguistic Values are used, where a set of Fuzzy Numbers get assigned human readable names for classification (in example: High, Medium, Low) by experts in the field. This enables us to quickly and effectively classify Base Event probabilities by expert's opinions with an acceptable level of uncertainty or subjectivity. [3]

4.3 Algebraic Equations for Fuzzy Probabilities

During the quantitative analysis of fault trees, each base event gets assigned a probability of it occurring. In traditional FTA this is a point-value (also called a crisp value).

Propagating these probabilities through the fault tree logic gates is done using probability theory and is also described in [1] with [2] applying this approach to fuzzy probability numbers.

To propagate event probabilities through a Fault Tree we can describe the three most common gates as follows:

Let A , B be two events with known probabilities F_A , F_B , assuming non-negative probabilities the logical AND gate operator can be written as:

$$F_{A\cap B}=F_A\times F_B \quad \text{or more generalized} \quad F^{\text{AND}} = \prod_{i=1}^{n} F_i \tag{4.3}$$

The NOT gate, or complement, operator takes only a single probability as input and is written as:

$$F^{NOT} = 1-F \tag{4.4}$$

Finally, the OR gate operator described in [1] is written in the maxterm notation, the one described in [2] however is a much simpler minterm notation and as they are both equivalent the simpler minterm notation is picked and it's generalized form is written as:

$$F^{\text{OR}} = 1 - \prod_{i=1}^{n} (1-F_i) \tag{4.5}$$

These formulas are however, used for point-wise probability numbers, while in fuzzy FTA these are instead fuzzy numbers, as such the formulas by themselves cannot be used.

However, fuzzy numbers with simple shapes, such as trapezoidal or triangular, are represented by their key points, which are themselves crisp probability numbers. Even alpha-cuts of fuzzy probability numbers are represented by two key points (the lower and upper bound) which are also crisp probability numbers. In essence, fuzzy numbers that are represented by key points can be written as an n-element vector, then by using element-wise operations (addition, subtraction, multiplication) with the above formulas we can apply the normal probability formulas to fuzzy numbers.

In example, let the inputs be trapezoidal fuzzy probability numbers, which have 4 key points x, y, z, k , the logic gate operations can be expanded and written as follows:

$$F^{\text{AND}} = \prod_{i=1}^{n} F_i = \prod_{i=1}^{n} \begin{bmatrix} x_i \\ y_i \\ z_i \\ k_i \end{bmatrix} = \begin{bmatrix} \prod_{i=1}^{n} x_i \\ \prod_{i=1}^{n} y_i \\ \prod_{i=1}^{n} z_i \\ \prod_{i=1}^{n} k_i \end{bmatrix} \tag{4.6}$$

$$F^{NOT} = 1 - F = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} x \\ y \\ z \\ k \end{bmatrix} = \begin{bmatrix} 1-x \\ 1-y \\ 1-z \\ 1-k \end{bmatrix} \tag{4.7}$$

$$F^{\text{OR}} = 1 - \prod_{i=1}^{n} (1 - F_i) = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} - \prod_{i=1}^{n} \begin{bmatrix} 1-x_i \\ 1-y_i \\ 1-z_i \\ 1-k_i \end{bmatrix} = \begin{bmatrix} 1 - \prod_{i=1}^{n} (1-x_i) \\ 1 - \prod_{i=1}^{n} (1-y_i) \\ 1 - \prod_{i=1}^{n} (1-z_i) \\ 1 - \prod_{i=1}^{n} (1-k_i) \end{bmatrix} \tag{4.8}$$

4.4 Fuzzy Importance and Uncertainty Measures

An important use of Fault Tree Analysis is to identify critical events that contribute the most to the top event occurrence, this is essential when it is used for safety analysis. In [7][13] two different importance measures were used, the Fuzzy Importance Measure (FIM) and Fuzzy Uncertainty Importance Measure (FUIM).

The Fuzzy Importance Measure is an extension of Birnbaum importance used in traditional FTA to be used in Fuzzy FTA. It is used to identify critical base events that contribute the most to the top event occurrence. [7][13]

Assume we have a Fuzzy Fault Tree containing ' n ' base events and we want to calculate the FIM of base event ' i ' with ' q_i ' being the fuzzy number representing it's probability of occurrence. The top event probability ' Q ' is calculated by some function that represents propagating the base events through the Fault Tree as follows:

$$Q = f(q_1, q_2, \dots, q_{i-1}, q_i, q_{i+1}, \dots, q_n) \tag{4.9}$$

The FIM can be calculated by assuming the base event ' i ' to be have occurred ($q_i = 1$) and calculating the top event with this assumption ($Q_{q_i=1}$), then assuming the base event has not occurred ($q_i = 0$) and calculating the top event with this assumption as well ($Q_{q_i=0}$). The difference between the two top events is the Birnbaum importance. [7][13]

However the top event is also a fuzzy number so in [7][13] the authors proposed a simple approach to calculate the difference using euclidean distances. The Fuzzy Importance Measure can be calculated using their approach with:

$$FIM_i = ED[Q_{q_i=1}, Q_{q_i=0}] \tag{4.10}$$

where ' $ED[A, B]$ ' is the euclidean distance between fuzzy sets ' A, B ' and is defined as:

$$ED[A, B] = \sum_{\alpha} \sqrt{(A_\alpha^L - B_\alpha^L)^2 + (A_\alpha^U - B_\alpha^U)^2} \tag{4.11}$$

where ' A_α^L ' and ' A_α^U ' are the lower and upper bounds respectively of the fuzzy set ' A ' at alpha-level ' α ', for each alpha level.

The FUIM represents the contribution of the uncertainty in base event ' i ' to the overall top event uncertainty, this can be used to identify which base event needs to have more data gathered about it to lower the overall system uncertainty by the largest amount.

The FUIM can be calculated in a similar manner to FIM except we use the assumption that the base event ' i ' has a crisp value for it's probability instead of a fuzzy number, this can be a point-wise or an interval depending on what strategy is used to acquire this crisp value, then calculating the top event probability ' Q_i '. This then compared to the regular top event probability ' Q ' without this assumption with the formula:

$$FUIM_i = ED[Q\ ,\ Q_i] \tag{4.12}$$

5 TREEZZY2 COMPUTER PROGRAM

The **TREEZZY2** compute software was developed by the University of Palermo Department of Nuclear Engineering in Italy, the work was presented around the year 2004. It is a computer software with a graphical interface to accomplish fuzzy fault tree or fuzzy event tree analysis. It can approximate top events or use the alpha-cut method to get a more accurate result at the cost of longer computation time. Further features include minimum cut set calculation, which is required for it to accomplish the most important calculations, the FIM and FUIM with rankings. [8]

Unfortunately, no information was garnered from the paper [8] as it merely presents their work, nor do they present the source code, further complication is the fact that it is in Italian with no option that was found to switch it to english, necessitating translation programs and observation to garner it's inner workings.

Furthermore it is written in Visual Basic, and requires **Microsoft Visual Basic 6 Service Pack 6** to be installed on a windows machine to work. The dependency installation can fail, in which case an external program called **7zip** needs to be used to extract the installer, the resulting **.dll** and **.ocx** files need to be manually registered from the windows command line (ensuring that the command prompt is opened with administrator privileges) using the **regsvr32** command in order to be able to use **TREEZZY2**.

There are relatively few options in the software, mainly how many digit precision to use, the base event shape (trapezoidal or triangular) and finally the notation which can either be decimal notation (0.15) or scientific notation (1.5e-1).

A curious note on the operation of the software is the top event calculation is done normally, while FIM and FUIM requires calculating minimum cut sets. A warning message will pop up in case the minimum cut sets were not yet calculated when the importance measure calculation was selected.

A shortcoming of the software is that there is no obvious way to use the same base event in multiple places, one has to declare it multiple times under different names, which makes the FIM and FUIM measure be unusable in that case.

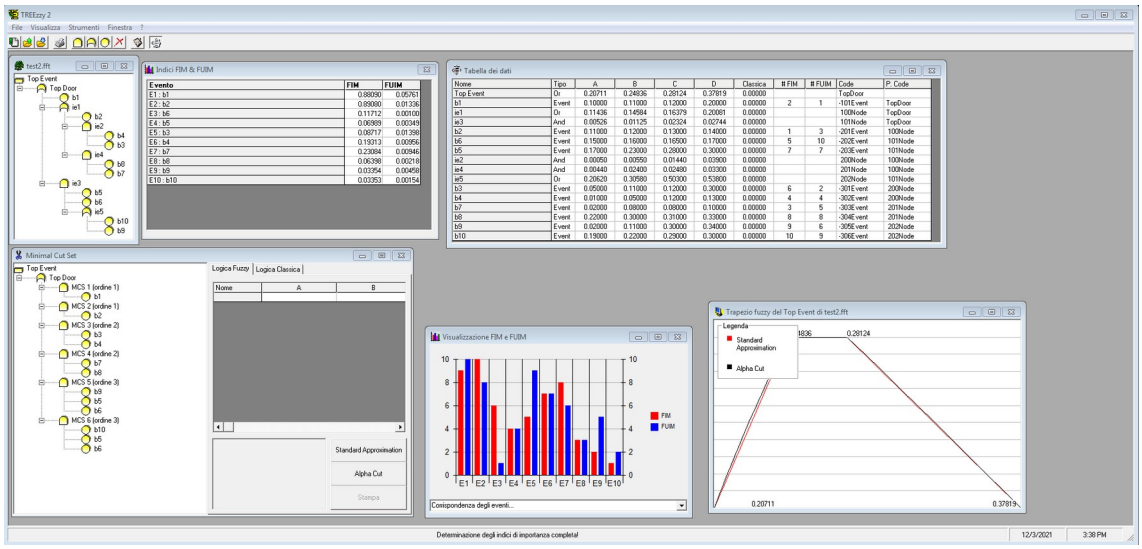

Figure 8: TREEZZY2 computer software graphical interface

6 FUZZYFTAPY SOFTWARE DEVELOPMENT

6.1 Python Language Overview

Python is a high-level, object-oriented, interpreted language with dynamic semantics and dynamic typing. It focuses on easily readable syntax and supports packages and modules to make modular applications. [10]

It is popular in the scientific and engineering fields because of it's readability and large community of developers and scientists that develop popular frameworks and libraries.

This section is a general overview of basic programming syntax with examples of the things used throughout the developed software. Tricks and advanced syntax used during each software section will be explained in chapter 6.4 where it first occurs in the source code.

6.1.1 Variables

In Python variables are declared using the following syntax:

```
varName = 2
varNameAswell = varName

anotherVar: dict = {'a': 1}
```

Snippet 1: Example in declaring variables

The variables are declared in this way are put in the name space they are declared in, in the above example both variables enter the global name space.

Python is a dynamically typed language, meaning that variable type declarations are unnecessary as they are inferred when variables are assigned. [10]

It is important to keep in mind that in Python everything is an object, this includes numbers, lists, functions definitions and class definitions, variables are merely a named pointer to the object. This has the side-effect that, using the above example, when we change the value of **varName** then **varNameAswell** will change along with it in cases where the variable points to a complex object like a dictionary, in that case manual call of the object's **copy()** or **deepcopy()** method is necessary to create an independent copy of the object. [10]

Lastly Python allows "type hinting" using the syntax in the above **anotherVar** variable, which is useful during development as modern code editors will show type hints during function calls to help developers decide what arguments to use and also can be used with static type checking software. It is a general good practice to type hint function arguments during declaration as it acts as another layer of documentation. [10]

6.1.2 Functions

In Python delimiting code blocks (loops, function declarations, etc.) is done by white-space indentation instead of using braces. This in turn makes the program code's visual structure more accurately represent it's semantic structure. The following example:

```
def some_function(positionalArgument: int, keyWordArgument: bool = True):
        aVar = 2
        print("Hello world!")
        print("This is a function")
        print(positionalArgument)
        return 42

some_function(3)
some_function(3, keyWordArgument = False)
```

Snippet 2: Example function declaration showing how indentation is used to separate code blocks

In the above example a function is declared with two different types of arguments, positional and key-word.

Positional arguments are required and failure to supply them will cause an error, their order matters and can be passed in when calling the function as seen above with the syntax " **some_function(2)** ". [10]

Key-word arguments strictly come after positional arguments during function declaration, they are declared by assigning some default value to it the same way variables are declared, using the above example the syntax for adding key-word argument is "**keyWordArgument = True**", this sets the default value when it is not supplied and to supply it during function calls is done with the syntax "**some_function(3, keyWordArgument = False)**". [10]

Finally functions always have a return value, if not explicitly given the function returns **None** , defining what the function returns is done by the **return** key-word followed by the object to be returned, note that this will immediately cause the function to exit. [10]

It is worth noting that functions can use the **yield** key-word instead to return a generator object, generators return a value and pause function execution at the exact line yield was

issued instead of exiting so it can be later resumed, we call these special functions generators. [10]

6.1.3 Classes

Defining new object classes in Python is done with the following syntax:

```
class NameOfClass:

        clsVar = 2

        def __init__(self, argument: int, kwArgument: bool = False):
                self.locVar = argument
                self.locVarKw = kwArgument

        def display_all(self):
                print(self.clsVar)
                print(self.locVar)
                print(self.locVarKw)

        @staticmethod
        def test():
                print("Hello World!")

class Inheritor(NameOfClass, OtherParent):
        pass

newInstance = NameOfClass(12)  # instancing
newInstance.display()          # calling a method
print(NameOfClass.clsVar)      # accessing an attribute
```

Snippet 3: Example class declaration showing initialization and inheritance

In the above example, the **NameOfClass** class does not inherit from other classes and so the parenthesis can be left out. Functions that are defined inside a class are called methods, the first argument automatically passed in is the object reference (upon which the method operates) and is denoted by **self** as a convention. [10]

Methods that start and end with double underscores (also referred to as dunder methods) are special (referred to as magic methods), there are a number of such methods classes can define to change their functionality but the most common one is **__init__** , which is called after the class has been instantiated to finish the class initialization. [10]

Variables attached to objects are called attributes, and can be accessed with the syntax **object.attribute** or in a concrete example **newInstance.locVar** . Objects can have more attributes added and their values can overridden at runtime like any variable. Methods are accessed the same way as attributes using the dotted syntax. [10]

Variables declared outside of **__init__** like the example **clsVar** are class-wide attributes that all instances have access to and can even be accessed without needing to instantiate the class. When an object tries to override it's class-wide attribute, it instead gets assigned a new attribute with the same name and because objects first search their local attributes before searching class-wide attributes they can be overridden on an instance basis if needed. [10]

Classes can inherit attributes and methods from other classes and Python supports multiple inheritance, meaning a single class can inherit from multiple classes as seen in the example class **Inheritor** .[10]

Lastly, the **@staticmethod** syntax is called a decorator. Decorators are outside the scope of this overview but it is worth noting that this particular one makes the method it is written above able to be called without having to instance a class, as it requires no object reference. [10]

6.1.4 Comments, Doc-Strings and F-Strings

Comments can be inserted by the # character, anything after it on the same line will be treated as comment. [10]

Multi-line comments can be made by triple quotes **"""Example comment"""** this is just a multi-line string that the Python interpreter will ignore if there are no other pieces of code on the same line as the beginning or the end of it. [10]

If the first line during the definition of a function, method or class is a string, we call it a doc-string and it acts as the documentation for the particular function, method or class which the code editors will often render out to help developers. An example in writing doc-strings is as follows for each use case: [10]

```
def func():
        """This is a function documentation"""

class TestClass:
        """This is a class documentation"""

        def test_method(self):
                """This is a method documentation"""
```

Snippet 4: Example Doc-String syntax

Finally f-strings are a special type of strings denoted by the syntax **f"..."** the letter **f** in-front of the string. F-strings allow embedding variables and function calls into strings and incorporating their return values into the text. Inside an f-string things between

curly braces, in-example **{varName}**, will be interpreted as code and executed, in the case of variables their values taken, while functions have their return values taken and converted into strings that are embedded into the f-string. [10]

An example of an f-string is:

```
f"Hello {name} ! Today is {get_day()} ."
```

Snippet 5: Example F-String syntax

6.1.5 Control Flow

In Python the **if, elif, else** keywords constitute the control flow, or branched execution of computer code. Best described by an example:

```
if value < 1:
        print("runs if the expression evaluates true")
elif value < 2:
        print("runs if the first expression evaluates false, but this one true")
else:
        print("runs if none of the if/elif branches run")
```

Snippet 6: Example if-else statement syntax

Using the above example, the **if** keyword creates the control flow, the expression after it must evaluate to a boolean (True of False) value. Multiple **if** statements can be bellow one another if they need to individually check and run.[10]

The **elif** statement works the same as **if** with the key exception that it must follow an **if** block and that it runs only if the expression after it is runs if it evaluates itself True only if the **if** block evaluates False, it is essentially an "else if" block.[10]

The **else** block must follow **if** or **elif** blocks and is the last block, running if all of them evaluated False, that is if they did not run.[10]

Another way to control the flow of program execution is with the keywords **try, except, else, finally**. This is beyond the scope of this thesis but it is worth noting that it is used for checking if a section of code executes successfully or it errors out, and catching or ignoring specific errors.[10]

6.1.6 Loops

In python, there are two types of loops, one is conditional the other is iterative. The **while** keyword loop runs while the conditional expression after it evaluates to True, testing it with each loop and exiting if it no longer does so.[10]

The **for** loop on the other hand works differently from other programming languages. Instead of the loop incrementing a counter, it consumes iterables returning each element

one element per loop and assigns that element to a variable to work with during the loop. An iterable is any object that is either a list, dictionary, string, generator, or any complex object class if it defines the magic methods to act like an iterable.[10]

Loops also have two keywords to further refine their operation. The **continue** keyword jumps to the next loop (jumps to the top), while the **break** keyword exits out of the loop.[10]

Lastly, loops can have an optional **else** block follow them, which executes the code block if and only if the loop did not exit because of a **break** keyword.[10]

```
x = 0

while x < 5:
        x += 1

for i in range(10):
        if i == 5:
                break
        elif i == 2:
                Continue

else:
        print("woops")
```

Snippet 7: Example for loop statements syntax

In the above example the **range(10)** built-in function creates a generator that returns exactly 10 integers starting from 0 incrementing by 1. Usually this exact syntax is used when there is a need for the loop to run a sent number of times.

6.1.7 Importing Modules

Importing is done via the **import** key word. Importing loads the specified python-file in the active directory, an installed or built-in module in its entirety.[10]

Python's import module is outside the scope of this thesis.

6.2 Software Design Considerations

The software name is **fuzzyftapy** , standing for Fuzzy- FaultTreeAnalysis-Python.

As previously discussed in 4.2 the most common fuzzy number's are trapezoidal or triangular in shape and triangular shapes can be described by special case trapezoidal ones, in this software implementation Trapezoidal fuzzy numbers will be used exclusively. Effort will have to be made to accommodate future expansion to other shapes as necessary.

Two logic gate types chosen to be implemented are AND gate, OR gate. The NOT gate is left out as it was not used in the papers or the software used to verify results. The gates will be used to propagate base events up to the top event, both in standard approximation and with alpha-cut methods.

To perform qualitative analysis on the fault tree, the minimum cut sets must be derived. This can be done if we assume the base events use bivalent logic, then use boolean algebra to get the sum-of-products (SOP) representation as mentioned in chapter 3.1.2 .

To accomplish quantitative analysis the fuzzy importance (FIM) and fuzzy uncertainty importance (FUIM) measures are implemented as described in chapter 4.4 .

To interface with the users the software will have a command-line interface to cut down on development time, later on a GUI can be added to make the software usable to others as well.

In order to load the fault tree to be used in the software, a simple approach of using text files will be used. This eliminates the need to program complex UI's for data loading as it can be written using a text editor, provided the file-structure is simple enough.

Finally, the results of the analysis will be output into a JSON text file for later analysis or directly printed to the console in JSON format, depending on user preference.

6.3 Save File Structure

In Python the dictionary is a native way to represent object relationships, and indeed objects themselves. To persist dictionaries on disk as save files, the JavaScript Object Notation (JSON) format is used, as Python gives tools to convert between dictionaries and JSON documents natively.

The dictionary type and JSON format stores data in key-value pairs. A key is unique and usually a string, it is by which we can retrieve data. The value can be any arbitrary data in a dictionary even other dictionaries, but in JSON it is limited to a couple of types, namely string, integer, float and other JSON documents. This latter functionality allows JSON documents to be embedded inside a JSON document under a key and allows it to represent complex data structures and object-relationships.

An important note in JSON structure is that key-value pairs and list entries are separated by a comma and most importantly, the last such element or key-value pair cannot have a trailing comma, otherwise an error will be produced.

```
{
        "key1" : {
                "key11" : 12,
                "key12" : "embedded"
        },
        "key2" : 3.14
}
```

Snippet 8: An example of the JSON format

Using the JSON format to represent the tree structure it needs to store meta-data, base events and the logic gates' inputs/outputs. The overall structure is as follows:

```
{
"metadata" : { ... },
"base-events" : { ... },
"logic-gates" : { ... }
}
```

Snippet 9: Overall structure of the input JSON file

The **"metadata"** section contains the software version, the tree type which determines the fuzzy number shapes of events.

The **"base-events"** section have each base event stored, the name's of which has to be unique as it is used as the key, while it's value will be a list of key points. An example as follows:

```
"base-events" : {
        "example trapezoidal base event" : [0.12 , 0.15 , 0.17 , 0.20],
        ...
}
```

Snippet 10: Base Event declaration in the input JSON file

The "logic-gates" section holds all logic gates used in the fault tree analysis. In this section, the keys are the names of the intermediate events (the output of the logic operation). Each logic gate holds another dictionary inside which has two keys "type" and "inputs", "type" defines the logic operation the gate performs while "inputs" lists out all base event or intermediate event's keys the gate uses as input.

An example of it's structure is as follows:

```
"logic-gates" : {
        "top-event" : {"type": "and", "inputs": ["base event 1", "intermediate 1"]},
          "intermediate 1": {"type": "or", "inputs": ["base event 2", "intermediate 2"]},
            "intermediate 2": {"type": "and", "inputs": ["base event 3", "base event 4"]}
}
```

Snippet 11: Logic Gate declaration in the input JSON file

The indentation in the above example serves no other purpose then to help visualize the tree structure.

A concrete example of a valid input file is as follows:

```
{
  "metadata": {
    "version": "0.0.1",
    "base-event-shape": "trapezoidal"
  },
  "base-events": {
    "b1": [0.1, 0.11, 0.12, 0.2],
    "b2": [0.21, 0.29, 0.31, 0.4],
    "b3": [0.05, 0.11, 0.12, 0.3],
    "b4": [0.3, 0.35, 0.48, 0.49],
    "b5": [0.17, 0.23, 0.28, 0.3],
    "b6": [0.22, 0.3, 0.3, 0.33]
  },
  "logic-gates": {
    "top-event": {"type": "and", "inputs": ["b1", "ie1", "ie3"]},
      "ie1": {"type": "or", "inputs": ["b2", "ie2"]},
        "ie2": {"type": "and", "inputs": ["b3", "b4"]},
      "ie3": {"type": "and", "inputs": ["b5", "b6"]}
  }
}
```

Snippet 12: Concrete example of an accepted input JSON file

Which is the representation of the following fault tree:

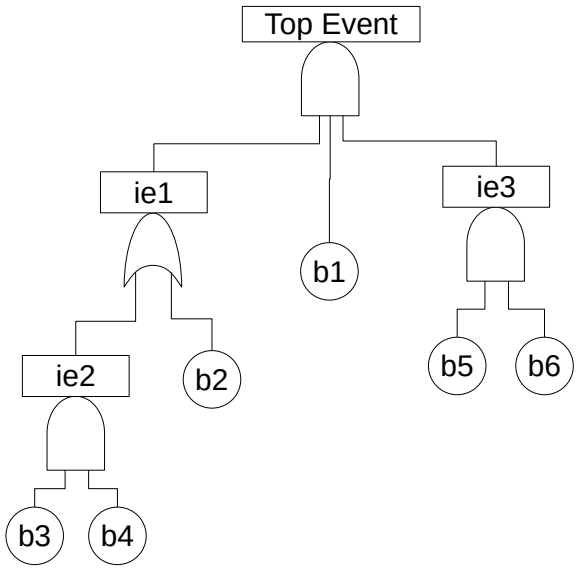

Figure 9: Fault Tree representation of example input JSON file shown in Snippet 12

6.4 Software Code and Documentation

6.4.1 Dependencies and Imports

```
import os
import itertools
import json
import math
```

Snippet 13: Import statements

The software itself was written on Python 3.9.7 specifically, later versions should work without issue and earlier versions from Python 3.6+ should work as well.

The only optional external dependency required is **matplotlib** (version 3.4.3 but other versions should work without issue) and is used for graph drawing and visualization. If those functions are unused (in example when the software is embedded) then it is not required to install.

6.4.2 Module-Level variables

```
__PRECISION__ = 12
__version__ = "0.0.1"
__MINIMUM_VERSION__ = "0.0.1"
```

Snippet 14: Module-level variables and their values at the time of writing.

When declaring variables in a Python file, not inside any function or class definition, we are putting it inside the Module's global name space, which is the same as the global name space only if we run the Python file directly and do not import it.

__PRECISION__ is an integer number that defines the precision of numbers (how many fractional digits to use). All calculations use the **round(x , __PRECISION__)** function to avoid floating point precision errors during comparisons and logic operations, where **x** is the number being rounded to **__PRECISION__** number of digits.

__version__ is a string that contains the current version number of the software. The version is built up from integer numbers separated by dots (in example: "0.1.1"), during version checking the leftmost number is the most significant, while right most number is least-significant (in example, "0.1.1" is less than "1.0.0"). It is used to determine save file compatibility.

__MINIMUM_VERSION__ is the same as **__version__** however it is used for denoting the minimum accepted version of fault tree save files. During file loads if the file has lesser version then the minimal allowed it will raise an error and prevent loading the file.

__licence__ holds the copyright notice for the Python code.

6.4.3 Exception classes

```
class FaultTreeLoadError(Exception):
    """An issue has occurred during file loading."""

class FaultTreeError(Exception):
    """An issue has occoured during some operation of the Fault Tree."""

class ComputationOrderingError(Exception):
    """An issue has occurred during the calculation of computation order."""

class VersionError(Exception):
    """An issue has occurred during version checking."""

class FuzzyNumberError(Exception):
    """An issue has occoured during the running of fuzzy number class operations."""
```

Snippet 15: Source code for Exception Classes

Exceptions defined in the software are simply an alias for Python's base **Exception** class, they have been made so that when errors occur their name is more descriptive and also easier to catch, separating built-in exceptions from those generated by the software itself. An example use case would be catching the exception generated from loading an invalid file and displaying an error message to the user, without terminating the entire application. Exceptions also accept an optional string to explain the error that has occurred. Exceptions can be sent to the Python interpreter using the **raise** keyword.

FaultTreeLoadError is raised when the fault tree fails to load from a file.

FaultTreeError is raised when an operation of the Fuzzy Fault tree fails for some reason.

ComputationOrderingError is raised when a **FuzzyFaultTree** object tries to calculate the order in which logic gates need to be calculated based on their input events, but fails to do so successfully.

VersionError is raised when the version check fails on fault tree loading from file.

FuzzyNumberError is raised when an error occurs with a fuzzy number class method.

6.4.4 ProbabilityTools class

```
class ProbabilityTools:

    @staticmethod
    def logical_and(probabilities: list[float]):
        return round(math.prod(probabilities), __PRECISION__)

    @staticmethod
    def logical_or(probabilities: list[float]):
        return round(1 - math.prod([1-p for p in probabilities]), __PRECISION__)

    @staticmethod
    def logical_not(probability):
        return round(1 - probability, __PRECISION__)
```

Snippet 16: Source code for the ProbabilityTools class, docstrings and error checking code has been omitted for the sake of brevity

This is a container class for holding functions that perform logical operations on a single, or a list of probability values, which are float types (fractional numbers) with values between $[0,1]$. The formulas used are explained in chapter 4.3 .

The used built-in function **math.product()** takes a list of values and multiplies each element together.

The code **[1-p for p in probabilities]** is a list comprehension that builds a new list by performing the operation $1-p$ on each element of the supplied list **probabilities** .

6.4.5 AlphaLevelInterval class

```
class AlphaLevelInterval:
    def __init__(self, lower: float, upper: float, alphaLevel: float):
        self.lower = lower
        self.upper = upper
        self.alphaLevel = alphaLevel

    def __repr__(self):
        return f"""AlphaLevelInterval(lower = {self.lower}, upper = {self.upper}, alphaLevel =
{self.alphaLevel})"""

    def to_list(self):
        return [self.lower, self.upper, self.alphaLevel]

    @staticmethod
    def logical_and(terms: list):
        lower = ProbabilityTools.logical_and([t.lower for t in terms])
        upper = ProbabilityTools.logical_and([t.upper for t in terms])
        return AlphaLevelInterval(lower, upper, terms[0].alphaLevel)

    @staticmethod
    def logical_or(terms: list):
        lower = ProbabilityTools.logical_or([t.lower for t in terms])
        upper = ProbabilityTools.logical_or([t.upper for t in terms])
        return AlphaLevelInterval(lower, upper, terms[0].alphaLevel)
```

Snippet 17: Source code for the AlphaLevelInterval class, docstrings and error checking code has been omitted for the sake of brevity

This is primarily a data container class for the results of alpha-cuts on fuzzy numbers, it also provides convenience functions for doing point-wise logical operations (and, or) on objects of the same class, using the formulas from chapter 4.3 .

6.4.6 TrapezoidalFuzzyNumber class

This is primarily a container class to hold information about fuzzy numbers with trapezoidal shape, storing their key points as discussed in chapter 3.2.2 .

Constructor methods and Flags

```
class TrapezoidalFuzzyNumber:
    _CRISP_STRATEGY: str = None
    _IMPLEMENTED_CRISP_STRATEGIES = ["interval-x2x3", "fully-available",
"fully-unavailable"]

    def __init__(self, x1: float, x2: float, x3: float, x4: float):
        self.x1 = float(x1)
        self.x2 = float(x2)
        self.x3 = float(x3)
        self.x4 = float(x4)

    @staticmethod
    def from_list(terms: list):
        return TrapezoidalFuzzyNumber(terms[0], terms[1], terms[2], terms[3])

    @staticmethod
    def from_list_triangular_with_errorfactor(terms: list):
        x1 = round(terms[0] / terms[1], __PRECISION__)
        x2 = x3 = terms[0]
        x4 = round(terms[0] * terms[1], __PRECISION__)
        return TrapezoidalFuzzyNumber(x1, x2, x3, x4)

    @staticmethod
    def from_list_triangular(terms: list):
        return TrapezoidalFuzzyNumber(terms[0], terms[1], terms[1], terms[2])
```

Snippet 18: Source code for the TrapezoidalFuzzyNumber class constructor methods, docstrings and error checking code has been omitted for the sake of brevity

The class gives several constructor methods. The **from_list** creates the fuzzy number from a list of it's keypoints. The method **from_list_triangular** constructs the class from the key points of a triangular fuzzy number. The **from_list_triangular_errorfactor** method implements the point median value and error factor notation used in [7][13].

The class variable **_CRISP_STRATEGY** if set to something other then **None** will use a defuzzification strategy to create a crisp output for both alpha-cuts and logical operations. The list of implemented strategies is held in the class variable **_IMPLEMENTED_CRISP_STRATEGIES** and the name's need to be unique.

Defuzzification strategies

```
def apply_crisp_strategy(self):
    if self._CRISP_STRATEGY is None:
        return self

    elif self._CRISP_STRATEGY == "fully-available":
        return TrapezoidalFuzzyNumber(0, 0, 0, 0)

    elif self._CRISP_STRATEGY == "fully-unavailable":
        return TrapezoidalFuzzyNumber(1, 1, 1, 1)

    elif self._CRISP_STRATEGY == "interval-x2x3":
        return TrapezoidalFuzzyNumber(self.x2, self.x2, self.x3, self.x3)

def set_crispness_to_fully_avalable(self):
    self._CRISP_STRATEGY = "fully-available"

def set_crispness_to_fully_unavalable(self):
    self._CRISP_STRATEGY = "fully-unavailable"

def set_crispness_to_interval_x2x3(self):
    self._CRISP_STRATEGY = "interval-x2x3"

def reset_crispness(self):
    self._CRISP_STRATEGY = None
```

Snippet 19: Source code for the TrapezoidalFuzzyNumber class defuzzyfication methods, docstrings and error checking code has been omitted for the sake of brevity

Defuzzification is needed during analysis to determine both Fuzzy Importance and Fuzzy Uncertainty Importance Measures, by returning a new instance of the class after applying the strategy it is trivial to integrate new strategies into this method. The following strategies that the analysis depended on were implemented (from chapter 4.4):

fully-available creates a point-wise number at 0 probability, FIM calculation $q_i = 0$

fully-unavailable creates a point-wise number at 1 probability, needed for FIM calculation $q_i = 1$

interval-x2x3 creates a square shaped fuzzy number, needed to calculate FUIM

Adding more strategy types is done by a new **elif** branch that implements the code itself, then adding the new unique name to the class variable **_IMPLEMENTED_CRISP_STRATEGY** as a list entry.

Convenience methods are also provided to change and reset the defuzzification strategy to make usage of it more readable and avoid errors in changing the **_CRISP_STRATEGY** variable directly.

Logical operations

```
@staticmethod
def logical_and(terms: list):
    terms = [t.apply_crisp_strategy() for t in terms]

    x1 = ProbabilityTools.logical_and([t.x1 for t in terms])
    x2 = ProbabilityTools.logical_and([t.x2 for t in terms])
    x3 = ProbabilityTools.logical_and([t.x3 for t in terms])
    x4 = ProbabilityTools.logical_and([t.x4 for t in terms])

    return TrapezoidalFuzzyNumber(x1, x2, x3, x4)

@staticmethod
def logical_or(terms: list):
    terms = [t.apply_crisp_strategy() for t in terms]

    x1 = ProbabilityTools.logical_or([t.x1 for t in terms])
    x2 = ProbabilityTools.logical_or([t.x2 for t in terms])
    x3 = ProbabilityTools.logical_or([t.x3 for t in terms])
    x4 = ProbabilityTools.logical_or([t.x4 for t in terms])

    return TrapezoidalFuzzyNumber(x1, x2, x3, x4)
```

Snippet 20: Source code for the TrapezoidalFuzzyNumber class logic operation functions, docstrings and error checking code has been omitted for the sake of brevity

To preform logical AND and OR operations on TrapezoidalFuzzyNumber objects two convenience methods are provided to do it point-wise for each of their key points. Note that before it is performed each of the supplied objects will be processed by their de-fuzzification method **apply_crisp_trategy** which will return the de-fuzzified number if there was a strategy defined for it, or the unmodified object if not.

Alpha cut method

```
def alphacut(self, alphaLevel: float):
    if not self._CRISP_STRATEGY is None:
        self = self.apply_crisp_strategy()

    lower = round(self.x1 + alphaLevel * (self.x2 - self.x1), __PRECISION__)
    upper = round(self.x4 - alphaLevel * (self.x4 - self.x3), __PRECISION__)

    return AlphaLevelInterval(lower, upper, alphaLevel)
```

Snippet 21: Source code for the TrapezoidalFuzzyNumber class logic operation functions, docstrings and error checking code has been omitted for the sake of brevity

The alpha-cut was performed by applying the crisp strategy first if any for de-fuzzification, then using the method outlined in chapter 4.2 we apply the alpha cut, returning a new instance of **AlphaLevelInterval** to hold the result.

6.4.7 FuzzyFaultTree class

Initialization, Loading and Saving

Saving and Loading happens in a JSON file with the structure is as outlined in chapter 6.3 .

File operations with Python happen by directly opening a file with the built-in method **open** that takes two arguments, the path to the file and the mode which is a string (default 'r'). However the files need to be manually closed at the end of reads or writes. To that end, the **with** keyword helps to create a context manager, which automatically closes the opened file once the indented code block finishes running or errors out, while the **as** keyword is used as an assignment to a new name.

In example, the following two syntaxes are equivalent in this use case:

open(file) as f

f = open(file)

```
class FuzzyFaultTree:
    _MAX_SEARCH_DEPTH = 1000
    _METADATA_STRUCTURE = {"version": 'str',
                           "base-event-shape": ["trapezoidal", "triangular", "triangular-errorfactor"]
                           }
    _LOGIC_GATE_STRUCTURE = {"type": ["and", "or"],
                             "inputs": 'list'
                             }

    def __init__(self, baseEventShape: str = 'trapezoidal', topEventGateType: str = 'and'):
        self.metadata = {'version': __version__, 'base-event-shape': baseEventShape}
        self.baseEvents = {}
        self.logicGates = {'top-event': {'type': topEventGateType, 'inputs': []}}

    @staticmethod
    def load_from_file(loadPath : str):
        treeDict = None
        with open(loadPath) as f:
            treeDict = json.load(f)

        if not self.version_check(treeDict['metadata']['version']):
            raise FaultTreeLoadError()

        metadata = treeDict['metadata']
        baseEvents = treeDict['base-events']

        for name, event in baseEvents.items():
            if metadata['base-event-shape'] == "trapezoidal":
                baseEvents[name] = TrapezoidalFuzzyNumber.from_list(event)
            elif metadata['base-event-shape'] == "triangular-errorfactor":
                baseEvents[name] =
TrapezoidalFuzzyNumber.from_list_triangular_with_errorfactor(event)
            elif metadata['base-event-shape'] == "triangular":
                baseEvents[name] = TrapezoidalFuzzyNumber.from_list_triangular(event)

        logicGates = treeDict['logic-gates']

        fft = FuzzyFaultTree()
        fft.metadata = metadata
        fft.baseEvents = baseEvents
        fft.logicGates = logicGates

        return fft

    def save_to_file(self, savePath: str):
        md = self.metadata.copy()
        be = {n: self.baseEvents[n].to_list() for n in self.baseEvents.keys()}
        lg = self.logicGates.copy()

        with open(savePath, 'w') as f:
            json.dump({'metadata': md, 'base-events': be, 'logic-gates': lg}, f, indent=4)
```

Snippet 22: Source code for the FuzzyFaultTree class constructor methods, docstrings and error checking code has been omitted for the sake of brevity

Version Checking

```
    @staticmethod
    def version_check(version: str):
        minVersionParts = __MINIMUM_VERSION__.split('.')
        thisVersionParts = version.split('.')
        maxVersionParts = __version__.split('.')

        versionNumberLength = max([len(vnum) for vnum in minVersionParts + maxVersionParts
+ thisVersionParts])

        minVersion = ''.join([part.ljust(versionNumberLength, '0')
for part in minVersionParts])
        thisVersion = ''.join([part.ljust(versionNumberLength, '0')
for part in thisVersionParts])
        maxVersion = ''.join([part.ljust(versionNumberLength, '0')
for part in maxVersionParts])

        return minVersion <= thisVersion <= maxVersion
```

Snippet 23: Source code for the FuzzyFaultTree class Version checking method, docstrings and error checking code has been omitted for the sake of brevity

Version checking is for filtering out incompatible versions when the program code has changed. If new methods were added for instance the **__version__** module-level variable needs to be changed to reflect some new functionality. However some changes can be breaking, such as when the input file structure is changed, in which case **__MINIMUM_VERSION__** needs updating to reflect the incompatibilities.

Version checking is done my simple string comparisons. The versions are first split into their constituent parts that are separated by a dot, then each part is measured for the number of characters it contains which is then used to pad each part to that length, by adding zeros to it's left side. This step is important for the comparisons to function properly.

Due to how string comparisons work in python, it accepts both letters and numbers in the versions.

It is worth noting that during the calculation of **versionNumberLength** the addition (+) sign is used to add lists together, in Python this is supported and the result of the operation will a new list that contains all elements of the added together lists. This acts similarly to appending each element from each list to an empty list.

Internal calculations

```
   def _calculate_dependent_logic_gates(self, gateName: str):
      search = [i for i in self.logicGates[gateName]["inputs"] if i in self.logicGates]
      dependancies = search.copy()

      for _ in range(self._MAX_SEARCH_DEPTH):
         if search == []:
            return dependancies
         found = [i for i in self.logicGates[search[0]]["inputs"] if i in self.logicGates]

         dependancies.extend(found)
         search.extend(found)
         search.pop(0)
      else:
         raise FaultTreeError(f"Could not calculate dependencies of '{logicGate}' :
MAX_SEARCH_DEPTH reached without success.")

   def _calculate_computation_order(self):
      processed = list(self.baseEvents.keys())
      computeOrder = [list(self.baseEvents.keys())]
      allEventKeys = []
      allEventKeys.extend(self.baseEvents.keys())
      allEventKeys.extend(self.logicGates.keys())

      for _ in range(self._MAX_SEARCH_DEPTH):
         if len(processed) == len(allEventKeys):
            break

         currentComputeOrder = []
         currentProcessed = []

         for name, gate in self.logicGates.items():
            if name in processed:
               continue
            elif not all([i in allEventKeys for i in gate['inputs']]):
               raise ComputationOrderingError(f"Gate '{name}' has inputs that are neither another
gate or a base event")

            elif all([i in processed for i in gate['inputs']]):
               currentProcessed.append(name)
               currentComputeOrder.append(name)
               continue

         computeOrder.append(currentComputeOrder)
         processed.extend(currentProcessed)
      else:
         raise ComputationOrderError("Maximum iteration count reached.")

      return computeOrder
```

Snippet 24: Source code for the FuzzyFaultTree internal calculation methods, docstrings and error checking code has been omitted for the sake of brevity

These methods are internal to the workings of the FuzzyFaultTree class and not strictly available outside of it, however it is not a private method and as such it can be accessed externally should it prove useful or needed.

The **_calculate_computation_order** method determines the order of operations for logic gate calculations. It can also be used to calculate which logic gates can be computed in parallel to one another. This is an iterative approach that first adds the base events to the list as they do not depend on anything, then in a continuous loop that runs at maximum **_MAX_SEARCH_DEPTH** number of times we add the events whose inputs have all been added to the compute order in a previous iteration.

The **_calculate_dependent_logic_gates** method searches all logic gates to find which one depend on a parent logic gate. This is used in the API when deleting a logic gate to also remove all other logic gates that it has as inputs down the tree. This is an iterative method that runs at most **_MAX_SEARCH_DEPTH** number of times before it errors out, as it did not find the answer in that time.

Top Event Standard Approximation

```
    def calculate_top_event(self, useMinCutSets: bool = False):
        if useMinCutSets:
            self = self.calculate_minimum_cut_sets(newTree=True)

        computeOrder = self._calculate_computation_order()
        computed = {}

        for depth, layerEvents in enumerate(computeOrder):
            for event in layerEvents:
                if depth == 0:
                    computed[event] = self.baseEvents[event]
                elif self.logicGates[event]['type'] == "and":
                    computed[event] = TrapezoidalFuzzyNumber.logical_and([computed[e]
for e in self.logicGates[event]['inputs']])
                elif self.logicGates[event]['type'] == "or":
                    computed[event] = TrapezoidalFuzzyNumber.logical_or([computed[e]
for e in self.logicGates[event]['inputs']])
                else:
                    raise Exception("logic gate neither 'and' nor 'or' ")

        return computed['top-event']
```

Snippet 25: Source code for the FuzzyFaultTree top event calculation method, docstrings and error checking code has been omitted for the sake of brevity

The top event fuzzy number is calculated point-wise as discussed in chapter 4.3 .

The method is normally operating on the fault tree, but the flag **useMinCutSets** if set to true will calculate the minimum cut sets of the fault tree, create a new FuzzyFaultTree

for it and replace **self** with this new tree, what this does is it allows the method to remain the same but operate on a different object (the same fault tree with the minimum cut sets for it's logic gates).

The method first has to calculate the order in which to calculate the logic gates and does so with the **_calculate_computation_order** method, the results of which is then used in a nested loop to calculate each logic gate output, stored in the **computed** dictionary.

Top Event Alpha-cut Calculation

```
    def calculate_top_event_alphacut(self, alphaLevel: float, useMinCutSets: bool = False):
        if useMinCutSets:
            self = self.calculate_minimum_cut_sets(newTree=True)

        computeOrder = self._calculate_computation_order()
        computedCuts = {}

        for layer in computeOrder:
            for event in layer:
                if event in self.baseEvents:
                    computedCuts[event] = self.baseEvents[event].alphacut(alphaLevel)
                else:
                    gate = self.logicGates[event]
                    if gate['type'] == "and":
                        computedCuts[event] = AlphaLevelInterval.logical_and([computedCuts[k]
for k in gate['inputs']])
                    elif gate['type'] == "or":
                        computedCuts[event] = AlphaLevelInterval.logical_or([computedCuts[k]
for k in gate['inputs']])
                    else:
                        raise Exception("gate type is neither 'and' , 'or' ")

        return computedCuts['top-event']
```

Snippet 26: Source code for the FuzzyFaultTree class top event alphacut method, docstrings and error checking code has been omitted for the sake of brevity

The alpha-cut method for calculating the top event is done in the same manner as **calculate_top_event** method works, with the key difference being that it calculates the base event's alpha-cut, which is then propagated up the tree point-wise instead of the fuzzy number representing the base event.

The flag **useMinCutSets** if set will calculate the top event using a fuzzy fault tree that has it's logic gates replaced by the minimum cut sets, working similarly as outlined in the method **calculate_top_event** .

Minimum Cut Sets Calculation

```
    def calculate_minimum_cut_sets(self, newTree: bool = False, analyzedGate='top-event',
_topCall = True):
        gateType = self.logicGates[analyzedGate]["type"]
        inputs = self.logicGates[analyzedGate]["inputs"]

        visited = []

        for i in inputs:
            if i in self.baseEvents:
                visited.append([i])
            else:
                visited.append(self.calculate_minimum_cut_sets(analyzedGate=i, _topCall = False))

        if gateType == "and":
            ret = list(itertools.product(*visited))
        elif gateType == "or":
            ret = list(itertools.chain(*visited))

        if not _topCall:
            answer = ret
        else:
            def flatten(L, outerLayer = False):
                if not (isinstance(L, list) or isinstance(L, tuple)):
                    return [L]
                else:
                    if outerLayer:
                        return list(itertools.chain([flatten(x) for x in L]))
                    else:
                        return list(itertools.chain(*[flatten(x) for x in L]))

            flt = flatten(ret, outerLayer=True)

            if not newTree:
                answer = flt
            else:
                ft = FuzzyFaultTree(self.metadata["base-event-shape"], "or")
                ft.baseEvents = self.baseEvents.copy()
                ft.metadata = self.metadata.copy()

                for i, inputs in enumerate(flt):
                    ft.add_logic_gate('top-event', f'MCS-{i+1}', 'and', inputs)

                answer = ft

        return answer
```

Snippet 27: Source code for the FuzzyFaultTree minimum cut set method, docstrings and error checking code has been omitted for the sake of brevity

The minimum cut sets have been discussed in chapter 2.1 and chapter 3.1.2 . This method uses a recursive depth-first graph walking algorithm to compute them. The **_topCall** flag is specifically used to to separate the outside call from recursive calls, **it is highly recommended to not use it**.

First, the method walks through the fuzzy fault tree depth-first, starting at the top-event and working down the tree through logic gates. Encountering a logic gate, it calls itself again recursively until it encounters a logic gate whose inputs are all base events. Then it performs one of two things depending on the logic gate's type.

The **OR** operation is the simplest, chaining the inputs together into a single list, as any of them occurring will cause the intermediate event (output of the logic gate) to occur. In example, the logic gate has base events **D, F, K, J** then walking through the gate it first creates a list of lists like so:

[[D], [F], [K, J]]

Then unpacks the list (the ***variable** syntax) to a series of inputs to **itertools.chain** function separately, which strings them into a single list: **[D, F, [K, J]]** .

To illustrate unpacking behavior in Python the following two bits of code illustrations are functionally equivalent:

function(*[[A], [B, C]])

function([A], [B, C])

The **AND** operation however, requires a different approach. Here the **itertools.product** built-in function is used which gives the Cartesian product of all iterables passed in as arguments. Using the above example, taking the list of lists **[[D], [F], [K, J]]** it takes the cartesian product, which in this case will be **[(D, F, K), (D, F, J)]** .

Once these are done, the result will be a deeply nested list, in example:

[A, [B, [C, D] , [E, [F]], G], H]

This needs to be converted into a 2 dimensional list from the n-dimensional deeply nested list, essentially it needs to be flattened. This is accomplished by the **flatten_list** recursive function to the necessary depth, in this case 2. The result of which is as follows:

[A, [B, C, D, E, F, G], H]

Note that the **flatten** function is defined inside another function. This is permitted in Python and causes little performance loss, as it is only defined and used once during program execution. However it is not a general purpose function nor is it used anywhere else.

Finally the **newTree** flag changes the return format from a list of base event names to a new fuzzy fault tree with the same data, but the logic gates replaced by the minimum cut sets in the SOP (sum of products) representation.

Fuzzy Importance Measure

```
    def calculate_fim(self, alphaLevels=[0.0], useMinCutSets: bool = False):
        if useMinCutSets:
            self = self.calculate_minimum_cut_sets(newTree=True)

        fim = {}

        for eventName in self.baseEvents:
            currentFim = 0
            for level in alphaLevels:
                qi = []

                self.baseEvents[eventName].set_crispness_to_fully_avalable()
                qi.append(self.calculate_top_event_alphacut(level))

                self.baseEvents[eventName].set_crispness_to_fully_unavalable()
                qi.append(self.calculate_top_event_alphacut(level))

                self.baseEvents[eventName].reset_crispness()

                currentFim += round(math.sqrt((qi[0].lower - qi[1].lower)**2 +
(qi[0].upper - qi[1].upper)**2), __PRECISION__)

            fim[eventName] = round(currentFim, __PRECISION__)

        fimRanks = {n: {"rank": None, "value": v} for n, v in fim.items()}

        lastNumber = None
        offset = 1
        for rank, name in enumerate([k for k, _ in sorted(fim.items(), key=lambda item: item[1],
reverse=True)]):
            if fimRanks[name]["value"] == lastNumber: offset -= 1
            fimRanks[name]["rank"] = rank + offset
            lastNumber = fimRanks[name]["value"]

        return fimRanks
```

Snippet 28: Source code for the FuzzyFaultTree class FIM method, docstrings and error checking code has been omitted for the sake of brevity

This method implements the Fuzzy Importance Measure (FIM) calculation discussed in chapter 4.4 .

The flag **useMinCutSets** if set will use a fuzzy fault tree that has it's logic gates replaced by the minimum cut sets during its calculation, working similarly as outlined in the method **calculate_top_event** .

The method has a nested for loop, which calculates the FIM number for each event by calculating the top event alpha-cut when the base event is set to be crisp and be fully available, then fully unavailable. It then takes the distance between the two alpha-cuts and accumulates the result as repeats the calculation for each specified alpha-level.

Once all base events have the FIM calculated, it then needs to be ranked. Ranking is done by sorting the FIM numbers in descending order, however as the results are stored in a dictionary it is tricky to accomplish.

The code " **fimRanks = {n: {"rank": None, "value": v} for n, v in fim.items()}** " is a list comprehension that can is used to construct a new dictionary to hold not just the value, but also the ranking for each base event.

To sort the entries in a dictionary, the built-in method **sorted** is used, which takes in an iterable and sorts it, optionally this function can take another function by which to do the sort in the key-word argument **key** , which is in this case is a lambda function " **lambda item: item[1]** " that is an unnamed function that takes in a list and returns the second element. To use this approach **sorted** requires a list, to create a list of key-value pairs from a dictionary the **dictionary.items()** method can be used. In order to get the resultant list in descending order the **reverse** key-word argument needs to be set to **True** .

Finally, we iterate through this sorted list, ignoring the value and only taking the name, using a list comprehension that creates a list of names, out of the key-value pairs. To get the ranks the built-in method **enumerate** is used to return the index of elements in a list along side the list element, which is essentially the rank.

The **lastNumber** and **offset** variables help to bin numbers together so that when the ranking loop has two or more of the same number show up it assigns the same rank to all equal numbers. This is highly dependant on the **__PRECISION__** attribute, unlike in the **TREEZZY2** code which seems to work on its own numerical precision and only the display is rounded.

Fuzzy Uncertainty Importance Measure

```
    def calculate_fuim(self, alphaLevels=[0.0], defuzzingMethod='interval-x2x3', useMinCutSets:
bool = False):
        if useMinCutSets:
            self = self.calculate_minimum_cut_sets(newTree=True)

        fuim = {}
        q = [self.calculate_top_event_alphacut(level) for level in alphaLevels]

        for eventName in self.baseEvents:
            self.baseEvents[eventName]._CRISP_STRATEGY = defuzzingMethod
            qi = [self.calculate_top_event_alphacut(level) for level in alphaLevels]
            self.baseEvents[eventName].reset_crispness()

            euclideanDistances = [math.sqrt((q[x].lower - qi[x].lower)**2 +
(q[x].upper - qi[x].upper)**2) for x in range(len(alphaLevels))]
            fuim[eventName] = round(sum(euclideanDistances), __PRECISION__)

        fuimRanks = {n: {"rank": None, "value": v} for n, v in fuim.items()}

        lastNumber = None
        offset = 1
        for rank, name in enumerate([k for k, _ in sorted(fuim.items(), key=lambda item: item[1],
reverse=True)]):
            if fuimRanks[name]["value"] == lastNumber: offset -= 1
            fuimRanks[name]["rank"] = rank + offset
            lastNumber = fuimRanks[name]["value"]

        return fuimRanks
```

Snippet 29: Source code for the FuzzyFaultTree class FUIM method, docstrings and error checking code has been omitted for the sake of brevity

This method implements the Fuzzy Uncertainty Importance Measure (FIUM) calculation discussed in chapter 4.4 .

This method is functionally identical to the **calculate_fim** method with the only difference is that the top event alphacut is calculated normally, then each base event is defuzzified with a certain strategy passed in with the **defuzzingMethod** argument. The distance between the regular top event alpha-cut and the top event alpha-cut when the base event is a crisp number, gives the FUIM number.

Note that because no paper mentioned what strategy is used for defuzzification to get the crisp number used to calculate FUIM, this is an optional argument for later development in this area.

Fault Tree manipulation API

```
def api_add_logic_gate(self, parentGate: str, name: str, gateType: str, inputs: list = []):
    if not name in self.logicGates[parentGate]["inputs"]:
        self.logicGates[parentGate]["inputs"].append(name)

    self.logicGates[name] = {"type": gateType, "inputs": inputs}

def api_remove_logic_gate(self, gateName: str):
    for gate in self.logicGates:
        if gateName in self.logicGates[gate]["inputs"]:
            self.logicGates[gate]["inputs"].remove(gateName)

    dependancies = [gateName]
    dependancies.extend(self._calculate_dependent_logic_gates(gateName))
    for d in dependancies:
        del self.logicGates[d]

def api_add_base_event(self, name: str, fuzzyNumber):
    self.baseEvents[name] = fuzzyNumber

def api_remove_base_event(self, name: str):
    for gate in self.logicGates:
        try:
            self.logicGates[gate]["inputs"].remove(name)
        except ValueError:
            pass

    del self.baseEvents[name]

def api_assign_base_event_to_gate_input(self, eventName: str, gateName: str):
    if not eventName in self.logicGates[gateName]["inputs"]:
        self.logicGates[gateName]["inputs"].append(eventName)

def api_remove_input_from_gate(self, inputName: str, gateName: str):
    if inputName in self.baseEvents:
        try:
            self.logicGates[gateName]["inputs"].remove(inputName)
        except ValueError:
            pass

    elif inputName in self.logicGates:
        self.api_remove_logic_gate(inputName)
```

Snippet 30: Source code for the FuzzyFaultTree class API methods, docstrings and error checking code has been omitted for the sake of brevity

These methods implement an API (application programming interface) for interacting with the fuzzy fault tree, mainly for adding and manipulating base events and logic gates. This helps embedded applications as well as front end development. All of the methods start with **api_** to distinguish them from other methods.

The **api_add_logic_gate** simply adds a logic gate to the fault tree, a required argument for it other then it's **name** is the **parentGate** which is the name of the logic gate it will be added to as an input, optionally it's inputs can be specified with the **inputs** key word argument. The new logic gate's name has to be a unique in the tree (both base events and logic gates) .

The **api_remove_logic_gate** method removes the specified logic gate by it's **name** from all inputs, then it searches the tree for all logic gates that it depended on and deletes them as well to keep the logic gate section of the fault tree fully connected.

The **api_add_base_event** method simply adds a new base event, without connecting it to logic gate inputs.

The **api_remove_base_event** method deletes the base event from the tree, searching through all logic gate inputs it also removes all occurrences.

The **api_assign_base_event_to** method adds the specified base event to the input of the specified logic gate, ignoring the operation without error if the logic gate already had it in it's inputs.

The **api_remove_input_from_gate** method removes the specified event from the specified logic gate's inputs, ignoring the operation without error if the logic gate did not have an input with that name. The method checks if the event to be removed from the inputs is a base event or a logic gate. If the event is a base event it simply removes it from the logic gate inputs, if it is another logic gate it calls the **api_remove_logic_gate** method to properly remove it.

6.5 Command Line Interface

The interface for the standalone operation of the software is developed to be command-line (or terminal) based. Executing the software source file requires Python 3.6+ with the optional dependency of the **matplotlib** package when the visualization is used. The program can be called from the console as such:

python fuzzyftapy.py <OPTIONS>

where **<OPTION>** is replaced by the command line options. Multiple options can be used together unless indicated otherwise.

The following options are available at the time of writing.

-h or **--help** : print out the help for the software detailing each option

-i or **--input-file** : Specifies the input JSON file that has the fault tree. (cannot be used with -GUI) example usage:

python fuzzyftapy.py -i "C:/the folder/input.json"

-gui : Launches the GUI frontend. (NOT IMPLEMENTED)

-l or **--alpha-levels** : Makes the calculations use the specified alpha levels. Cannot be used with the **-nl** option. Example usage:

python fuzzyftapy.py -i "C:/the folder/input.json" -l 0.0 0.2 0.4 0.8 1.0

-nl or **--number-of-alpha-levels** : Makes calculations use a list of alpha levels equally divided between 0 and 1 a number of times supplied. (minimum of 2). Cannot be used with the **-l** option. Example usage:

python fuzzyftapy.py -i "C:/the folder/input.json" -nl 7

-o or **--output-file** : Save results to a new JSON file. (You are responsible for the file extension.). If this is not specified the results will be printed to the console. Example usage:

python fuzzyftapy.py -i "C:/the folder/input.json" -o "C:/the folder/output.json"

-teap or **--calc-top-approx** : Calculate the top event using standard approximation. This calculates the key points. Example usage:

python fuzzyftapy.py -i "C:/the folder/input.json" -teap

-tecut or **--calc-top-cuts** : Calculate the top event with the alpha-cut method. Example usage:

python fuzzyftapy.py -i "C:/the folder/input.json" -tecut

-mcs or **--minimum-cut-sets** : Calculate the minimum cut sets of the fault tree. (for the results). Example usage:

python fuzzyftapy.py -i "C:/the folder/input.json" -mcs

-umcs or **--use-minimum-cut-sets** : Signal the top event and importance measure calculations to use the minimum cut set representation of the tree. Example usage:

python fuzzyftapy.py -i "C:/the folder/input.json" -umcs

-im or **--calc-importance** : Calculate the importance measures FIM and FUIM of base events using the alpha cut method. Example usage:

python fuzzyftapy.py -i "C:/the folder/input.json" -im

-p or **--precision** : Sets the precision to the specified number of digits. Example Usage:

python fuzzyftapy.py -i "C:/the folder/input.json" -p 6

-v or **--visualize** : Visualize the computation results using 'matplotlib'. (requires matplotlib as dependency). Note that this option requires either **-teap** or **-tecut** for visualizing the top event and **-im** to visualize the FIM and FUIM ranks. Example usage:

python fuzzyftapy.py -i "C:/the folder/input.json" -teap -im -v

-t or **--time-execution** : Prints rough estimates of program execution time in seconds to the console (Does not save in file). This only calculates execution time for other options such as **-teap** , **-tecut** , **-im** , **-mcs** or **-umcs** . Example usage:

python fuzzyftapy.py -i "C:/the folder/input.json" -mcs -t

--licence : Print out the copiryght notice. NOTE: Using this ignores all other flags. Example usage:

python fuzzyftapy.py --licence

7 TESTING AND COMPARISON

7.1 Methodology for Testing and Comparisons

To test the developed program the **TREEZZY2** computer program will be used as a base-line and all tested Fault Trees will be reproduced in both the developed program and the **TREEZZY2** computer software.

The input event's fuzzy probabilities will be given in a table and the resulting top event fuzzy number, FIM and FUIM will also be compared in a table and the tree structure will be illustrated. All computation will be done with 6 digit precision wherever possible, 10 digits when it is necessary to avoid rounding related differences.

The main points of comparisons are as follows:

- Top Event Standard Approximation
- Fuzzy Importance Measure (FIM) and its ranking
- Fuzzy Uncertainty Importance Measure (FUIM) and its ranking
- Top Event Alpha Cut approach (at 10 alpha levels between [0, 1] to cross-check with TREEZY2 graph)
- Minimum Cut Sets

Note that all numbers and event trees are just fictional to test and compare the results of both programs given a diverse set of inputs.

The exact flags used for each result of the **fuzzyftapy** software developed and tested and as it appears in the results comparison tables is as follows:

fuzzyftapy: **python fuzzyftapy.py -i input.json -teap -tecut -im -p 6 -l 0 1**

fuzzyftapy with mcs:
python fuzzyftapy.py -i input.json -teap -tecut -im -umcs -p 6 -l 0 1

fuzzyftapy alpha = 0: **python fuzzyftapy.py -i input.json -teap -tecut -im -p 6 -l 0**

In special cases when the numbers are too small the decimal precision is increased to 10 digits both in TREEZZY2 and in fuzzyftapy with the **-p 10** flag.

The drawing of the graph for the top event is done with the **-nl 10** flag to replace the exact alpha levels passed in (with **-l 0 1**) in order to get the same drawing as in TREEZZY2, which uses 10 alpha levels by default.

7.2 Test 1

7.2.1 Input Data

Base Event Name	x1	x2	x3	x4
e1	0.15684	0.15998	0.161789	0.171034
e2	0.05684	0.08143	0.099785	0.120013
e3	0.31498	0.31649	0.319564	0.319719
e4	0.123456	0.135847	0.164891	0.235881
e5	0.005698	0.016234	0.121041	0.123052
e6	0.015893	0.019753	0.023495	0.032479
e7	0.002358	0.007985	0.123798	0.145287
e8	0.079521	0.099002	0.179811	0.498552
e9	0.225889	0.2522	0.314412	0.337008

Table 2: Trapezoidal Base Event keypoints for Test 1

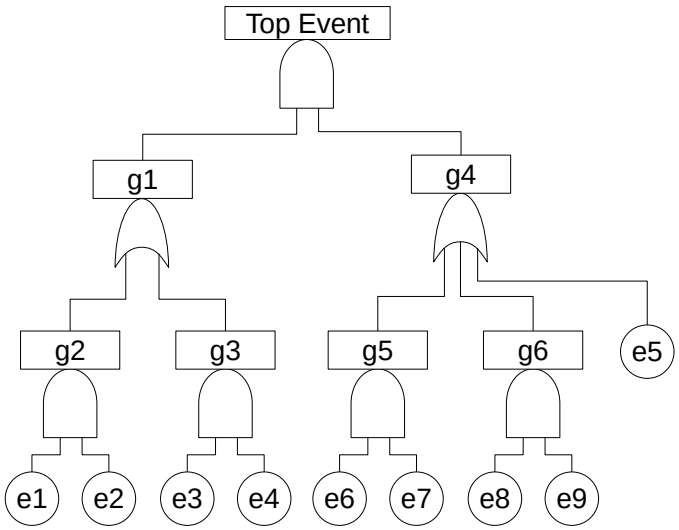

Figure 10: Fault Tree drawing for Test 1

7.2.2 Results

Top Event Apprximate	x1	x2	x3	x4
TREEZZY2	0.001120	0.002271	0.011772	0.025848
fuzzyftapy	0.001120	0.002271	0.011771	0.025848
fuzzyftapy with mcs	0.001133	0.002314	0.012374	0.028110

Table 3: Trapezoidal Top Event keypoints using Standard Approximation for Test 1

Event Name	FIM				FIM ranking			
	TREZZY2	fuzzyftapy	fuzzyftapy with mcs	fuzzyftapy alpha = 0.0	TREZZY2	fuzzyftapy	fuzzyftapy with mcs	fuzzyftapy alpha = 0.0
e1	0.0344	0.047086	0.052506	0.030412	6	7	7	6
e2	0.048837	0.070715	0.078457	0.043449	4	4	4	4
e3	0.068129	0.091954	0.098197	0.063333	3	3	3	3
e4	0.091758	0.14199	0.150297	0.086069	2	2	2	2
e5	0.092834	0.174749	0.191715	0.091001	1	1	1	1
e6	0.013522	0.017001	0.02194	0.010008	8	8	8	8
e7	0.003028	0.004046	0.005064	0.002355	9	9	9	9
e8	0.031764	0.052983	0.059138	0.02974	7	6	6	7
e9	0.046867	0.053244	0.060459	0.041246	5	5	5	5

Table 4: FIM numbers and ranks for base events in Test 1

Event Name	FUIM				FUIM ranking			
	TREZZY2	fuzzyftapy	fuzzyftapy with mcs	fuzzyftapy alpha = 0.0	TREZZY2	fuzzyftapy	fuzzyftapy with mcs	fuzzyftapy alpha = 0.0
e1	0.00032	0.00028	0.000321	0.00028	6	6	6	6
e2	0.001483	0.000879	0.001003	0.000879	5	4	4	4
e3	0.00013	1.00E-05	1.10E-05	1.00E-05	9	9	9	9
e4	0.007144	0.006088	0.00661	0.006088	2	2	2	2
e5	0.005239	0.000515	0.000536	0.000515	3	5	5	5
e6	0.000137	8.90E-05	0.000123	8.90E-05	8	7	7	7
e7	0.000233	4.80E-05	6.50E-05	4.80E-05	7	8	8	8
e8	0.009999	0.008851	0.010145	0.008851	1	1	1	1
e9	0.002551	0.000933	0.001067	0.000933	4	3	3	3

Table 5: FUIM numbers and ranks for base events in Test 1

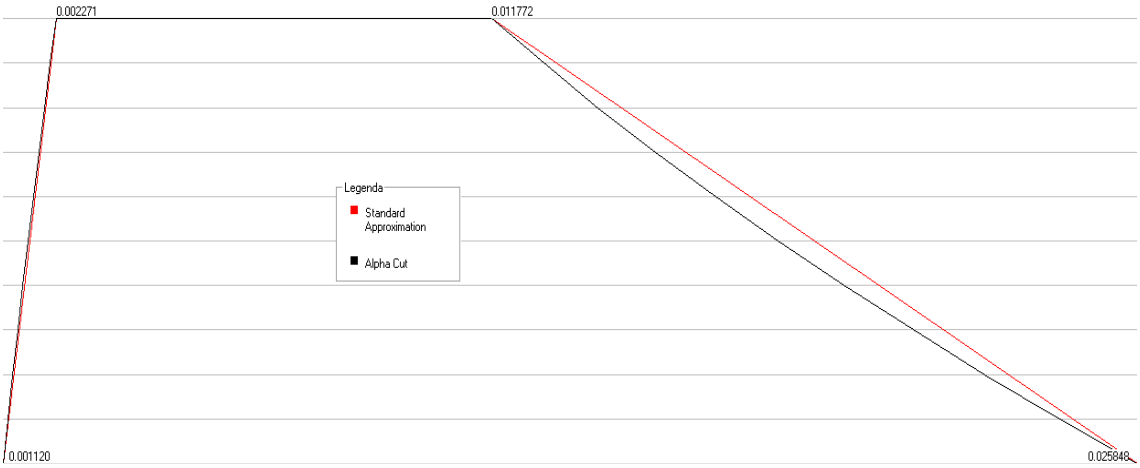

Figure 11: Diagram of Top Event shape using Standard Approximation and Alpha Cut method from the TREEZY2 software for Test 1

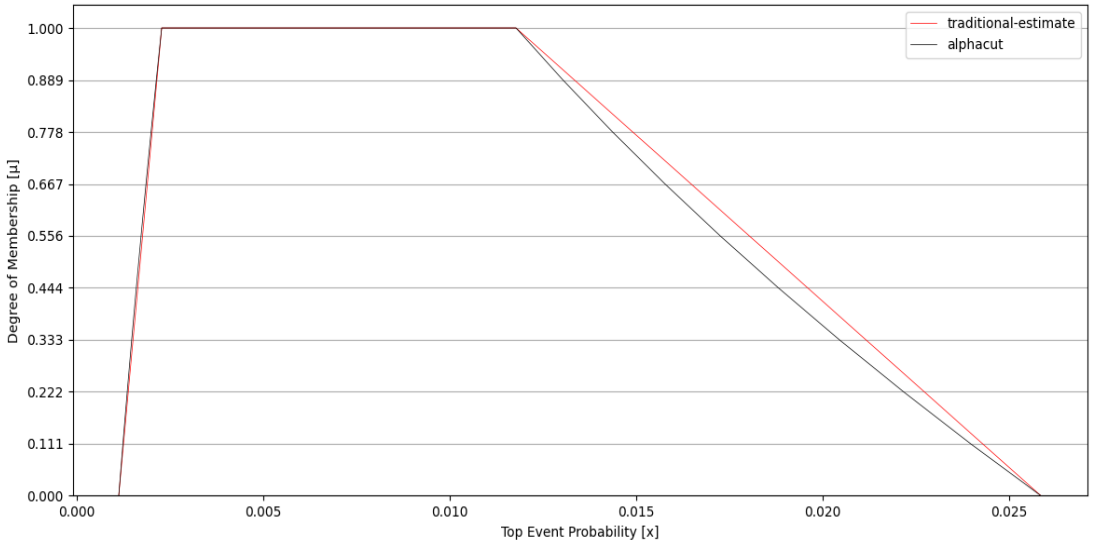

Figure 12: Diagram of Top Event shape using Standard Approximation and Alpha Cut method from the fuzzyftapy software for Test 1

```
{
        “mcs-TREEZZY2”: [     ["e1", "e2", "e5"], ["e3", "e4", "e5"],
                              ["e1", "e2", "e6", "e7"], ["e1", "e2", "e8", "e9"],
                              ["e3", "e4", "e8", "e9"], ["e3", "e4", "e6", "e7"]
                        ]
        ”mcs-FUZZYFTAPY”: [ ["e1", "e2", "e5"], ["e3", "e4", "e5"],
                              ["e1", "e2", "e6", "e7"], ["e1", "e2", "e8", "e9"],
                              ["e3", "e4", "e8", "e9"], ["e3", "e4", "e6", "e7"]
                            ]
}
```

Snippet 31: Test 1 Minimum cut set results for FUZZYFTAPY and TREEZZY2, format (JSON) and order has been altered for readability

7.2.3 Comparison

The Top Event keypoints using Standard Approximation shown in Table 3 is compared between TREEZZY2 and FUZZYFTAPY , which shows that all are exactly the same except for **x2** which seems to be a rounding error. Using minimum cut sets to calculate the top event in FUZZYFTAPY produce overall higher estimates for each key point of the top event.

Both the Fuzzy Importance and Uncertainty Importance measures produce different numerical results as seen in Table 4 and Table 5 , however the FIM ranking was similar in prediction by both software, with all variations of parameters tested in FUZZYFTAPY, with only **e1** and **e7** switching places. The FUIM ranks however featured minor deviations between the two software, while all options of FUZZYFTAPY produced the same Result.

The FIM results using different alpha levels to perform their calculations show a clear difference with higher number of alpha cuts producing higher overall measures. This is expected behavior as the formula adds together the measure at each alpha level, but TREEZZY2 has no such option suggesting a different algorithm is used in it.

The Figure 11 and Figure 12 show the plots of the alpha cut method for top event calculation. As TREEZZY2 does not feature any known numerical output only a graph, it can only be compared in shape, which show that the alpha cut approximation follow a similar inflection.

Finally minimum cut sets seen in Snippet 31 shows that both software calculated the exact same minimum cut sets.

7.3 Test 2

7.3.1 Input Data

Base Event Name	x1	x2	x3	x4
e1	0.015531	0.025998	0.298410	0.310005
e2	0.110035	0.153320	0.154220	0.192235
e3	0.095513	0.155200	0.155200	0.160000
e4	0.200563	0.210056	0.219535	0.225753
e5	0.005698	0.016234	0.121041	0.123052

Table 6: Trapezoidal Base Event keypoints for Test 2

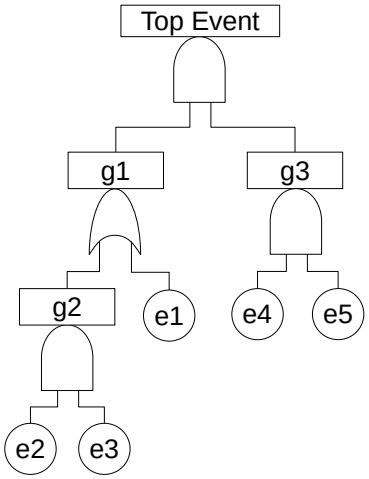

Figure 13: Fault Tree drawing for Test 2

7.3.2 Results

Top Event Approximate	x1	x2	x3	x4
TREEZZY2	0.000030	0.000168	0.008376	0.009201
fuzzyftapy	0.000030	0.000168	0.008376	0.009201
fuzzyftapy with mcs	0.000030	0.000170	0.008561	0.009459

Table 7: Trapezoidal Top Event keypoints using Standard Approximation for Test 2

Event Name	FIM				FIM ranking			
	TREZZY2	fuzzyftapy	fuzzyftapy with mcs	fuzzyftapy alpha = 0.0	TREZZY2	fuzzyftapy	fuzzyftapy with mcs	fuzzyftapy alpha = 0.0
e1	0.027756	0.053099	0.054553	0.026949	3	3	3	3
e2	0.004406	0.006006	0.008533	0.003068	5	5	5	5
e3	0.005294	0.006606	0.009394	0.003686	4	4	4	4
e4	0.041787	0.078918	0.080708	0.040758	2	2	2	2
e5	0.076442	0.144921	0.147811	0.074956	1	1	1	1

Table 8: FIM numbers and ranks for base events in Test 2

Event Name	FUIM				FUIM ranking			
	TREZZY2	fuzzyftapy	fuzzyftapy with mcs	fuzzyftapy alpha = 0.0	TREZZY2	fuzzyftapy	fuzzyftapy with mcs	fuzzyftapy alpha = 0.0
e1	0.004098	0.000312	0.000322	0.000312	2	1	1	1
e2	0.000173	0.000116	0.000168	0.000116	4	4	3	4
e3	0.000069	0.000019	0.000026	0.000019	5	5	5	5
e4	0.004790	0.000253	0.000260	0.000253	3	2	2	2
e5	0.004279	0.000159	0.000164	0.000159	1	3	4	3

Table 9: FUIM numbers and ranks for base events in Test 2

Figure 14: Diagram of Top Event shape using Standard Approximation and Alpha Cut method from the TREEZY2 software for Test 2

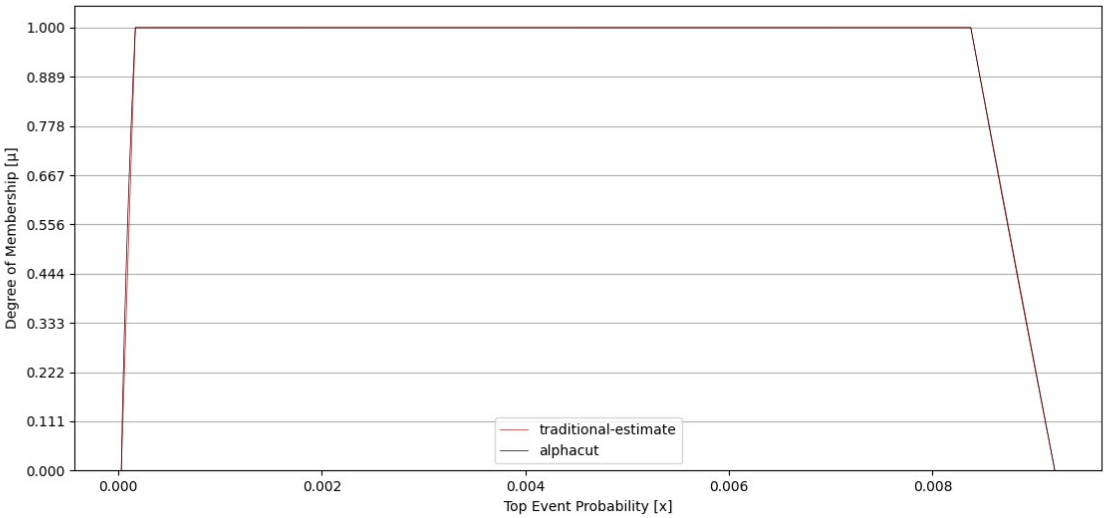

Figure 15: Diagram of Top Event shape using Standard Approximation and Alpha Cut method from the fuzzyftapy software for Test 2

```
{
        “mcs-TREEZZY2”: [ ["e1", "e4", "e5"], ["e2", "e3", "e4", "e5"] ],
        ”mcs-FUZZYFTAPY”: [ ["e1", "e4", "e5"], ["e2", "e3", "e4", "e5"] ]
}
```

Snippet 32: Test 2 Minimum cut set results for FUZZYFTAPY and TREEZZY2, format (JSON) and order has been altered for readability

7.3.3 Comparison

The Top Event keypoints using Standard Approximation shown in Table 7 reveals that both TREEZZY2 and FUZZYFTAPY reach the same conclusion exactly. Using the minimum cut sets for calculating this in FUZZYFTAPY produce overall higher estimates for each key point of the top event.

Both the Fuzzy Importance and Uncertainty Importance measures produce different numerical results as seen in Table 8 and Table 9 , however the FIM ranking was exactly as predicted by both software, with all variations of parameters tested in FUZZYFTAPY. The FUIM ranks however featured minor deviations, namely base events **e1** and **e5** has had their ranks swapped when compared to TREEZZY2 in most cases, using the minimum cut sets in FUZZYFTAPY has also made base event **e2** deviate as well.

While the top event graphs depicting both standard approximation and alphacuts between the two software in Figure 14 and Figure 15 show a graph that has too little in the way detail to analyze as TREEZZY2 does not allow us to zoom in to the graph, it nonetheless shows similar inflections (the black line). Though this result is inconclusive as the feature sizes are far too small.

Finally minimum cut sets seen in Snippet 32 shows that both software calculated the exact same minimum cut sets.

7.4 Test 3

7.4.1 Input Data

Base Event Name	x1	x2	x4
e1	0.019985	0.035122	0.042238
e2	0.059841	0.061238	0.071002
e3	0.156991	0.235120	0.272563
e4	0.100251	0.120000	0.156847
e5	0.110000	0.130000	0.130000
e6	0.051235	0.068945	0.071462
e7	0.156234	0.195312	0.256843
e8	0.258952	0.279856	0.315689

Table 10: Triangular Base Event keypoints for Test 3

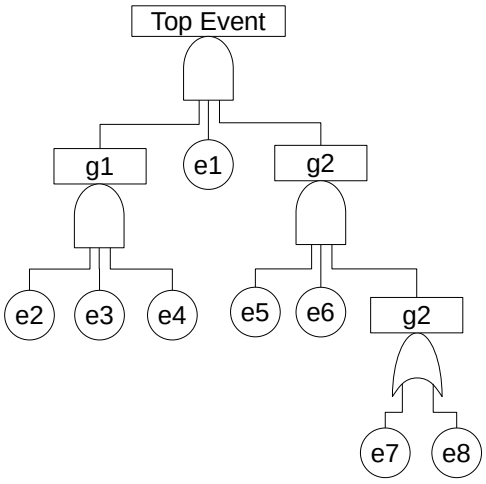

Figure 16: Fault Tree drawing for Test 3

7.4.2 Results

Top Event Approximate	x1	x2	x4
TREEZZY2	3.98E-08	2.29E-07	5.85E-07
fuzzyftapy	3.98E-08	2.29E-07	5.85E-07
fuzzyftapy with mcs	4.41E-08	2.58E-07	6.82E-07

Table 11: Triangular Top Event keypoints using Standard Approximation for Test 3

Event Name	FIM				FIM ranking			
	TREZZY2	fuzzyftapy	fuzzyftapy with mcs	fuzzyftapy alpha = 0.0	TREZZY2	fuzzyftapy	fuzzyftapy with mcs	fuzzyftapy alpha = 0.0
e1	1.61447E-05	2.32097E-05	2.67007E-05	1.40003E-05	1	1	1	1
e2	9.60430E-06	1.35526E-05	1.56009E-05	8.27080E-06	2	2	2	2
e3	2.50190E-06	3.53820E-06	4.07210E-06	2.16250E-06	6	6	6	6
e4	4.34770E-06	6.44830E-06	7.41550E-06	3.75300E-06	5	5	5	5
e5	5.24560E-06	7.00520E-06	8.07250E-06	4.51720E-06	4	4	4	4
e6	9.54240E-06	1.29191E-05	1.48822E-05	8.22770E-06	3	3	3	3
e7	1.19110E-06	1.37280E-06	1.96500E-06	8.18900E-07	8	7	7	8
e8	1.19110E-06	1.50870E-06	1.96500E-06	8.89700E-07	7	7	8	7

Table 12: FIM numbers and ranks for base events in Test 3

Event Name	FUIM				FUIM ranking			
	TREZZY2	fuzzyftapy	fuzzyftapy with mcs	fuzzyftapy alpha = 0.0	TREZZY2	fuzzyftapy	fuzzyftapy with mcs	fuzzyftapy alpha = 0.0
e1	1.34300E-07	1.03100E-07	1.19500E-07	1.03100E-07	2	2	2	2
e2	8.17000E-08	8.04000E-08	9.38000E-08	8.04000E-08	4	4	4	4
e3	1.08900E-07	8.28000E-08	9.61000E-08	8.28000E-08	3	3	3	3
e4	1.49000E-07	1.37700E-07	1.60400E-07	1.37700E-07	1	1	1	1
e5	1.57000E-08	7.20000E-09	8.00000E-09	7.20000E-09	8	8	8	8
e6	4.58000E-08	2.47000E-08	2.84000E-08	2.47000E-08	6	7	7	7
e7	6.93000E-08	5.02000E-08	7.34000E-08	5.02000E-08	5	5	5	5
e8	4.00000E-08	3.18000E-08	4.28000E-08	3.18000E-08	7	6	6	6

Table 13: FUIM numbers and ranks for base events in Test 3

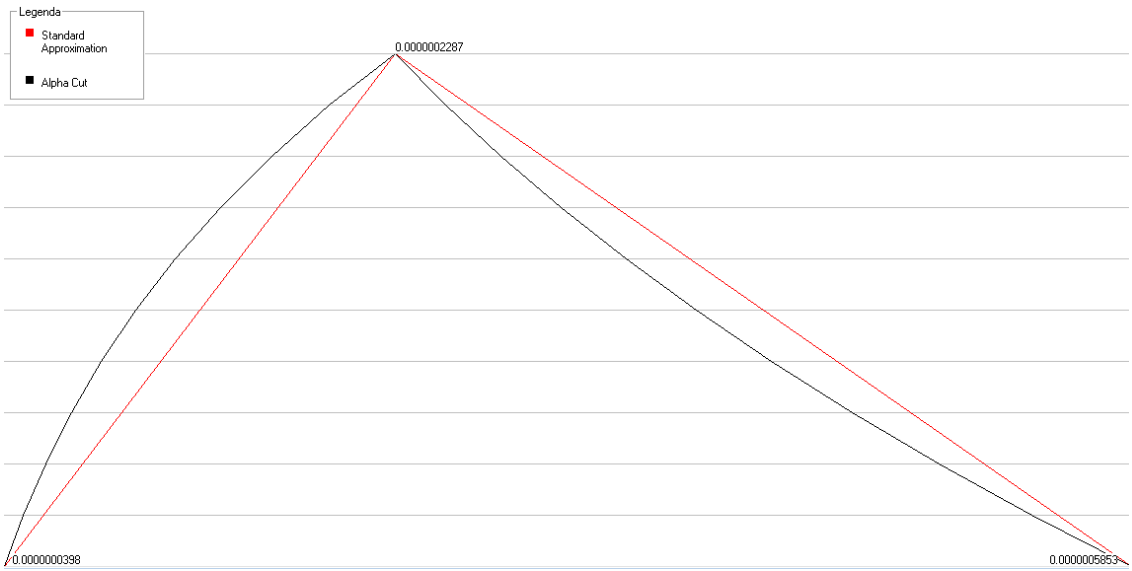

Figure 17: Diagram of Top Event shape using Standard Approximation and Alpha Cut method from the TREEZY2 software for Test 3

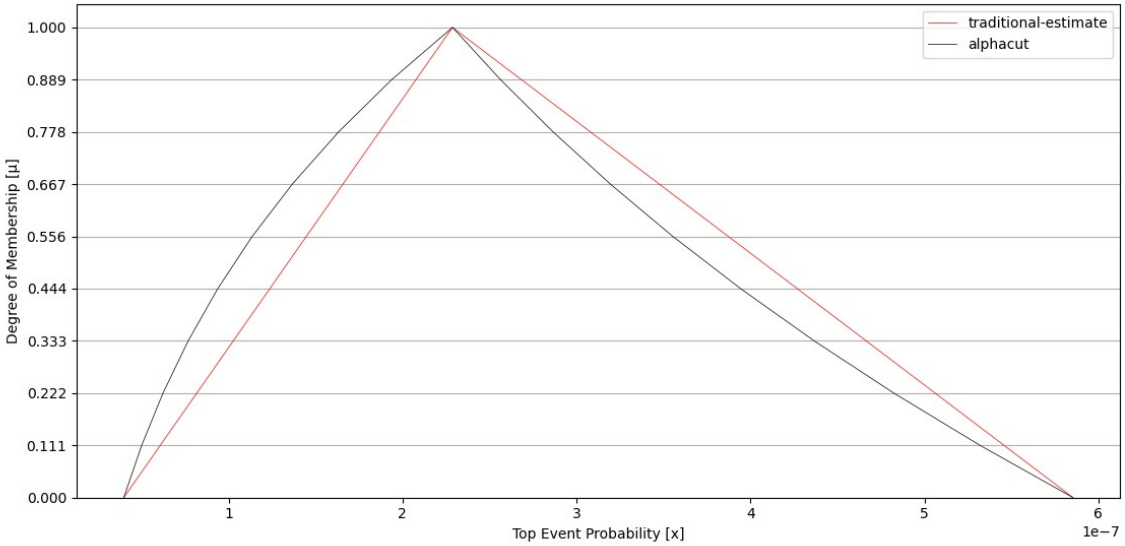

Figure 18: Diagram of Top Event shape using Standard Approximation and Alpha Cut method from the fuzzyftapy software for Test 3

```
{
        “mcs-TREEZY2”: [
                                [“e2”, “e3”, “e4”, “e5”, “e6”, “e7”, “e1”],
                                ["e2", "e3", "e4", "e5", "e6", "e8", "e1"]
                        ],
        ”mcs-FUZZYFTAPY”: [
                                ["e2", "e3", "e4", "e5", "e6", "e7", "e1"],
                                ["e2", "e3", "e4", "e5", "e6", "e8", "e1"]
                            ]
}
```

Snippet 33: Test 3 Minimum cut set results for FUZZYFTAPY and TREEZZY2, format and order has been altered for readability

7.4.3 Comparison

The top event key points shown in Table 11 exactly match between the TREEZZY2 and FUZZYFTAPY predictions. Once more using minimum cut sets in the approximation causes the estimate to be a higher estimate.

The FIM and FUIM numbers in Table 12 and Table 13 differ greatly between the TREEZZY2 and FUZZYFTAPY predictions. The FIM rankings meanwhile show the same predictions with the exception of events **e7** and **e8** which differ between options in FUZZYFTAPY, while FUIM ranks follow a similar pattern with **e6** and **e8** differing between the two software but remaining overall the same regardless of options.

The Top Event Alpha Cut graphs shown on Figure 17 and Figure 18 appear to be matching in shape exactly between the two software.

Finally minimum cut sets seen in Snippet 33 shows that both software calculated the exact same minimum cut sets.

7.5 Test 4

7.5.1 Input Data

Base Event Name	x1	x2	x3	x4
e1	0.100	0.110	0.122	0.200
e2	0.215	0.294	0.300	0.351
e3	0.051	0.119	0.122	0.371
e4	0.311	0.354	0.485	0.499
e5	0.170	0.230	0.289	0.325
e6	0.220	0.300	0.300	0.339
e7	0.108	0.114	0.121	0.222
e8	0.219	0.291	0.316	0.451
e9	0.051	0.115	0.124	0.358
e10	0.350	0.351	0.480	0.496
e11	0.171	0.230	0.284	0.371
e12	0.229	0.351	0.351	0.371
e13	0.110	0.118	0.122	0.205
e14	0.210	0.295	0.310	0.400
e15	0.051	0.113	0.127	0.328
e16	0.301	0.350	0.489	0.490
e17	0.179	0.233	0.289	0.352
e18	0.221	0.302	0.310	0.330
e19	0.081	0.115	0.123	0.211
e20	0.210	0.295	0.316	0.401
e21	0.054	0.117	0.125	0.313
e22	0.311	0.351	0.481	0.490
e23	0.107	0.232	0.287	0.308
e24	0.221	0.234	0.330	0.337

Table 14: Trapezoidal Base Event keypoints for Test 4

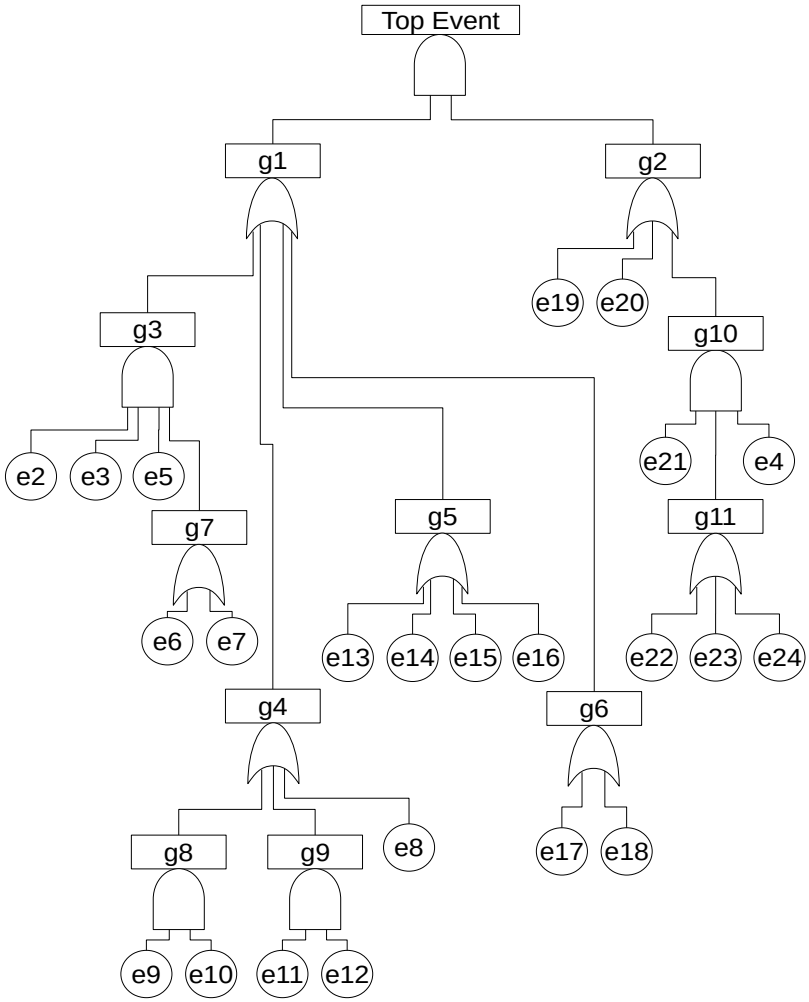

Figure 19: Fault Tree drawing for Test 4

7.5.2 Results

Top Event Approximate	x1	x2	x3	x4
TREEZZY2	0.021874	0.037964	0.048164	0.113625
fuzzyftapy	0.021874	0.037964	0.048164	0.113626
fuzzyftapy with mcs	0.039942	0.085578	0.123440	0.369692

Table 15: Trapezoidal Top Event keypoints using Standard Approximation for Test 4

Event Name	FIM				FIM ranking			
	TREZZY2	fuzzyftapy	fuzzyftapy with mcs	fuzzyftapy alpha = 0.0	TREZZY2	fuzzyftapy	fuzzyftapy with mcs	fuzzyftapy alpha = 0.0
e1	0.117836	1.133165	1.850544	0.608784	1	1	1	1
e2	0.000392	0.000266	0.007674	0.000190	22	22	22	22
e3	0.001424	0.000380	0.008709	0.000191	19	20	20	21
e4	0.011010	0.029818	0.166286	0.022106	7	5	12	5
e5	0.000503	0.000293	0.008317	0.000205	21	21	21	20
e6	0.000190	0.000159	0.004887	0.000107	24	23	23	23
e7	0.000380	0.000132	0.004885	0.000091	23	24	24	24
e8	0.014188	0.019092	0.175885	0.009765	6	7	6	7
e9	0.010674	0.005684	0.080702	0.002907	8	16	13	16
e10	0.002819	0.002225	0.044452	0.001413	18	19	19	19
e11	0.004093	0.004512	0.062214	0.002001	14	17	17	17
e12	0.001314	0.003550	0.055838	0.001747	20	18	18	18
e13	0.007450	0.015390	0.171337	0.007982	12	12	11	12
e14	0.009893	0.018738	0.175072	0.009410	9	8	7	8
e15	0.019149	0.015414	0.173193	0.008016	5	11	10	11
e16	0.007978	0.021909	0.177062	0.010774	11	6	5	6
e17	0.009139	0.017711	0.174179	0.008950	10	10	8	10
e18	0.003365	0.018594	0.174000	0.009208	16	9	9	9
e19	0.031769	0.218600	0.616996	0.119428	4	3	3	3
e20	0.036793	0.278676	0.666225	0.152712	3	2	2	2
e21	0.036967	0.064303	0.324788	0.036329	2	4	4	4
e22	0.004522	0.009072	0.073078	0.006623	13	13	14	13
e23	0.003676	0.006775	0.072099	0.004883	15	15	16	15
e24	0.003205	0.007066	0.072256	0.005103	17	14	15	14

Table 16: FIM numbers and ranks for base events in Test 4

Event Name	FUIM				FUIM ranking			
	TREZZY2	fuzzyftapy	futtyftapy with mcs	fuzzyftapy alpha = 0.0	TREZZY2	fuzzyftapy	futtyftapy with mcs	fuzzyftapy alpha = 0.0
e1	0.910042	0.044368	0.125199	0.044368	1	1	1	1
e2	0.006726	0.000010	0.000344	0.000010	20	21	22	21
e3	0.006365	0.000045	0.001589	0.000045	21	18	14	18
e4	0.130400	0.000317	0.001836	0.000317	5	13	13	13
e5	0.007262	0.000008	0.000263	0.000008	19	22	23	23
e6	0.004211	0.000005	0.000165	0.000005	22	23	24	24
e7	0.004208	0.000010	0.000426	0.000010	23	21	20	22
e8	0.101540	0.000963	0.014030	0.000963	7	6	6	6
e9	0.049476	0.000469	0.011750	0.000469	16	10	7	10
e10	0.035967	0.000023	0.000575	0.000023	18	20	18	20
e11	0.037012	0.000147	0.003265	0.000147	17	14	11	14
e12	0.037012	0.000137	0.000958	0.000137	17	15	17	15
e13	0.097603	0.000336	0.008392	0.000336	12	12	9	12
e14	0.100706	0.000816	0.009526	0.000816	8	7	8	7
e15	0.099545	0.001028	0.020660	0.001028	11	5	5	5
e16	0.102184	0.000432	0.001430	0.000432	6	11	15	11
e17	0.099930	0.000509	0.006602	0.000509	9	9	10	9
e18	0.099577	0.000648	0.003108	0.000648	10	8	12	8
e19	0.321984	0.009266	0.033645	0.009266	3	3	4	3
e20	0.360316	0.012981	0.034461	0.012981	2	2	3	2
e21	0.196490	0.006634	0.040243	0.006634	4	4	2	4
e22	0.057281	0.000066	0.000526	0.000066	13	17	19	17
e23	0.056342	0.000120	0.001231	0.000120	15	16	16	16
e24	0.056490	0.000037	0.000402	0.000037	14	19	21	19

Table 17: FUIM numbers and ranks for base events in Test 4

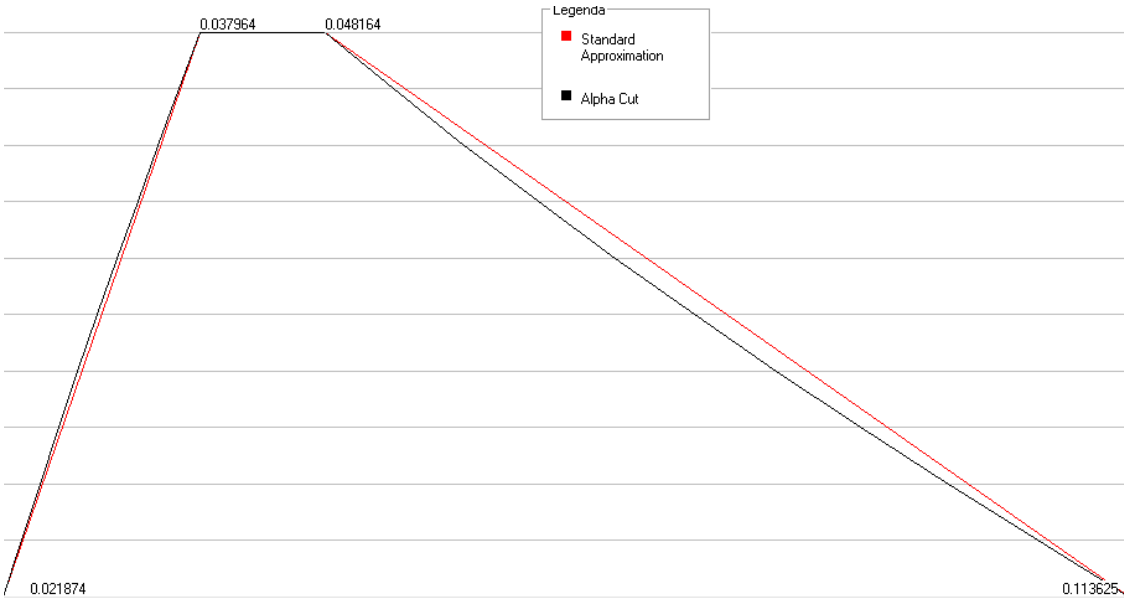

Figure 20: Diagram of Top Event shape using Standard Approximation and Alpha Cut method from the TREEZY2 software for Test 4

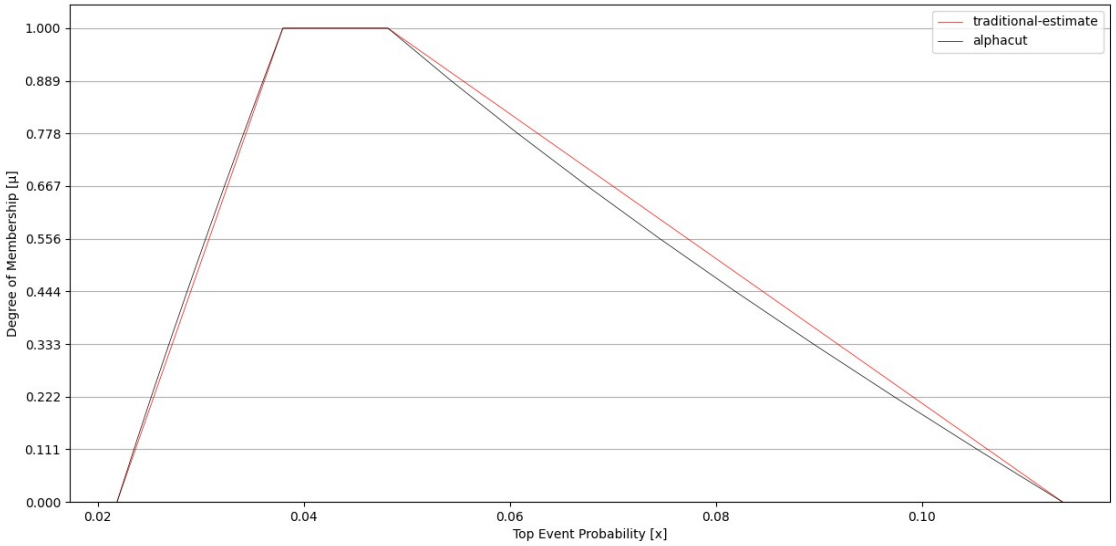

Figure 21: Diagram of Top Event shape using Standard Approximation and Alpha Cut method from the fuzzyftapy software for Test 4

```
{ "mcs": [ ["e1", "e8", "e19"], ["e1", "e8", "e20"],
   ["e1", "e13", "e19"], ["e1", "e13", "e20"], ["e1", "e14", "e19"],
   ["e1", "e14", "e20"], ["e1", "e15", "e19"], ["e1", "e15", "e20"],
   ["e1", "e16", "e19"], ["e1", "e16", "e20"], ["e1", "e17", "e19"],
   ["e1", "e17", "e20"], ["e1", "e18", "e19"], ["e1", "e18", "e20"],
   ["e1", "e11", "e12", "e19"], ["e1", "e11", "e12", "e20"],
   ["e1", "e9", "e10", "e19"], ["e1", "e9", "e10", "e20"],
   ["e1", "e15", "e21", "e23", "e4"], ["e1", "e15", "e21", "e24", "e4"],
   ["e1", "e13", "e21", "e23", "e4"], ["e1", "e13", "e21", "e24", "e4"],
   ["e1", "e16", "e21", "e22", "e4"], ["e1", "e16", "e21", "e23", "e4"],
   ["e1", "e16", "e21", "e24", "e4"], ["e1", "e8", "e21", "e24", "e4"],
   ["e1", "e8", "e21", "e22", "e4"], ["e1", "e17", "e21", "e22", "e4"],
   ["e1", "e17", "e21", "e23", "e4"], ["e1", "e17", "e21", "e24", "e4"],
   ["e1", "e14", "e21", "e22", "e4"], ["e1", "e14", "e21", "e23", "e4"],
   ["e1", "e18", "e21", "e22", "e4"], ["e1", "e18", "e21", "e23", "e4"],
   ["e1", "e18", "e21", "e24", "e4"], ["e1", "e14", "e21", "e24", "e4"],
   ["e1", "e8", "e21", "e23", "e4"], ["e1", "e13", "e21", "e22", "e4"],
   ["e1", "e15", "e21", "e22", "e4"], ["e1", "e11", "e12", "e21", "e24", "e4"],
   ["e1", "e11", "e12", "e21", "e22", "e4"], ["e1", "e11", "e12", "e21", "e23", "e4"],
   ["e1", "e9", "e10", "e21", "e22", "e4"], ["e1", "e9", "e10", "e21", "e23", "e4"],
   ["e1", "e9", "e10", "e21", "e24", "e4"], ["e1", "e2", "e3", "e5", "e7", "e19"],
   ["e1", "e2", "e3", "e5", "e7", "e20"], ["e1", "e2", "e3", "e5", "e6", "e19"],
   ["e1", "e2", "e3", "e5", "e6", "e20"],
   ["e1", "e2", "e3", "e5", "e7", "e21", "e24", "e4"],
   ["e1", "e2", "e3", "e5", "e7", "e21", "e22", "e4"],
   ["e1", "e2", "e3", "e5", "e7", "e21", "e23", "e4"],
   ["e1", "e2", "e3", "e5", "e6", "e21", "e22", "e4"],
   ["e1", "e2", "e3", "e5", "e6", "e21", "e23", "e4"],
   ["e1", "e2", "e3", "e5", "e6", "e21", "e24", "e4"] ]}
```


Snippet 34: Test 4 Minimum cut set results for TREEZZY2 in JSON format, FUZZYFTAPY produced the same result but is left out for brevity

7.5.3 Comparison

The top event key points shown in Table 15 exactly match between the TREEZZY2 and FUZZYFTAPY predictions. Once more using minimum cut sets in the approximation causes the estimate to be a higher estimate.

The FIM and FUIM numbers in Table 16 and Table 17 differ greatly between the TREEZZY2 and FUZZYFTAPY predictions, most notably the fact that base event **e1** FIM number in TREEZZY2 is an order of magnitude lower then in FUZZYFTAPY. While FUZZYFTAPY can produce numbers higher then 1 during FIM and FUIM calculation the TREEZZY2 software does not. The rankings are similar once more with major differences in rankings between the two software and some minor differences between the FUZZYFTAPY options.

The Top Event Alpha Cut graphs shown on Figure 20 and Figure 21 appear to be matching in shape exactly between the two software.

The minimum cut sets are shown in Snippet 34 were the same for FUZZYFTAPY software as well, but it was left out for the sake of brevity.

7.6 Execution time

	test 1	test 1 using MCS	test 4	test 4 using MCS
top event approximate	0.000173 s	0.000197 s	0.000269 s	0.001297 s
top event alphacut	0.000351 s	0.000503 s	0.000640 s	0.002705 s
FIM	0.003500 s	0.003834 s	0.020930 s	0.068483 s
FUIM	0.005197 s	0.005707 s	0.030690 s	0.101906 s
MCS	0.000053 s	0.000066 s	0.000236 s	0.000000 s
total	0.009274 s	0.010307 s	0.052529 s	0.174391 s

Table 18: Single run execution times for FUZZYFTAPY using test datasets

The TREEZZY2 software could not be comparatively measured as during usage and testing it was running in a virtual machine, however during test4 calculations determining the minimum cut set took over a minute, and even standard approximation took seconds.

In contrast the FUZZYFTAPY software, even in the more complex test4 dataset, took a fraction of a second to complete all its calculations. These numbers are highly hardware dependent and should be evaluated on a case-by-case basis. Furthermore these are single run numbers, a more comprehensive test would average many runs on the same dataset. This shows the potential to use in real-time applications after further optimizations took place.

It is worth noting that these numbers were obtained on Python 3.9+ using a Ryzen 1700x processor.

8 POTENTIAL APPLICATIONS OF FFTA IN SELF-DRIVING CARS

As the automobile industry is moving towards fully autonomous self-driving vehicles and new cars sold to consumers have some level of autonomy, be it in the form of cruise control, lane keeping or semi-autonomous driving, it is becoming increasingly important to quantify and asses the safety and risks of self-driving. Since self-driving cars are relatively few and were introduced into traffic a few years ago there are relatively few crashes to draw accurate statistics about their safety from. Not only that, but even after a crash has occurred an investigation into the cause of it has to be made to determine wether the autonomous vehicle was at fault (and which component there of) or a human error has been made by the driver of the vehicle, or another participant of the crash.

There has been some application of Fuzzy Logic in parts of the autonomous systems in self-driving vehicles and fewer research papers discussing fault-tree applications in risk assessment of autonomous vehicle components, namely [11]. And just one found that discussed using fault-tree's for self-driving vehicle Neural Network (AI) robustness estimation in extreme conditions, namely [12].

While few papers applied Fuzzy Fault Tree Analysis successful in certain applications that had high levels of uncertainty to estimate failure probabilities [1][2][3], no paper found had applied the approach to self-driving vehicles.

As such, there is few references to draw inspiration from and an in-depth analysis into the subject is beyond the scope of this paper. Although there are a few places where this can be applied.

One such field is in the risk assessment of self driving car component failure. Components in this case can be both hardware (sensors, electronics, communication) or software (sign recognition, obstacle detection, etc.).

Another such field is determining the learnability of intersections by neural networks. Each intersection would be analyzed in many aspects, such as the frequency of crashes, number of lanes, pedestrian traffic, turning lanes, etc. , after which the performance of neural networks can be evaluated at each intersection and critical intersections can be identified by key measures, such as FIM and FUIM ranks to focus efforts on gathering data and improving neural network model performances at those critical intersections.

9 CONCLUSION

A fuzzy methodology applied to traditional fault tree techniques can help model and quantify uncertainties in the top event approximation, helping engineers determine which base events should be investigated to reduce the top event occurrence with the fuzzy importance measure (FIM) while also showing which base events contribute the most to the overall system uncertainty with the fuzzy uncertainty importance measure (FUIM).

This thesis provided an overview in the field of fuzzy fault tree analysis and a brief summary in the Python programming language to aid understanding and further development in this field.

Taking the TREEZZY2 software as a base a new software called FUZZYFTAPY was developed with a permissive license and as much documentation in this thesis and it's source code form as necessary, using as few dependencies as possible, making a command-line interface for it to make it embeddable in other software as an internal or external dependency.

Testing concluded that FUZZYFTAPY produces the same top event using standard approximation, similar top event using the alpha cut approach and the exact same minimal cut sets as the TREEZZY2 software. However the FIM and FUIM numbers were very different, suggesting that TREEZZY2 uses a different method of computation then the literature. The FIM and FUIM ranks on the other hand were similar in both software which is the important measure, though differences warrant further testing.

There was no literature found on the application of fuzzy fault tree analysis in self-driving vehicles, and few papers discussing fault tree analysis, making this a new avenue of research.

Further testing and validation needs to be done on the developed software to ensure the results are correct, optimization needs to be done to improve computation performance and new approaches of de-fuzzification need to be examined to improve the FUIM measure, as the literature did not discuss this topic in any length. With applications in the fields of self-driving vehicles evaluated.

The created software and thesis work translated in English and Hungarian is hosted under the following Github repository:

https://github.com/AzureDVBB/fuzzyftapy

References

[1] VESELY, William E., et al. *Fault tree handbook*. Nuclear Regulatory Commission Washington DC, 1981, pp. 34 36 58-59 79 93-97 174-180.

[2] ZHENG, Xiaoxia, et al. Drive system reliability analysis of wind turbine based on fuzzy fault tree. In: *2016 35th Chinese Control Conference (CCC)*. IEEE, 2016. p. 6761-6765, pp. 1-2.

[3] CASAMIRRA, Maddalena, et al. Safety analyses of potential exposure in medical irradiation plants by Fuzzy Fault Tree. 2014, pp. 1-2 4.

[4] Fuzzy Set Theory, KRUSE, Rudolf; MOEWES, Christian. Fuzzy Systems. 2012, [viewed date: 11 December 2021]. Available From: <http://fuzzy.cs.ovgu.de/wiki/uploads/Lehre.FS1213/fs2012_ch02_fst.pdf>

[5] PAHL, Peter J.; DAMRATH, Rudolf. *Mathematical foundations of computational engineering: a handbook*. Springer Science & Business Media, 2001. pp. 15

[6] ZADEH, Lotfi A. Fuzzy sets and information granularity. *Advances in fuzzy set theory and applications*, 1979, 11: 3-18.

[7] GUIMARÃES, Antonio CF; EBECKEN, Nelson FF. FuzzyFTA: a fuzzy fault tree system for uncertainty analysis. *Annals of Nuclear Energy*, 1999, 26.6: 523-532. pp. 2-4

[8] BELLINI, Salvatore; CASAMIRRA, Maddalena; CASTIGLIA, Francesco. TREEZZY2, a Fuzzy Logic Computer Code for Fault Tree and Event Tree Analyses. In: *Probabilistic Safety Assessment and Management*. Springer, London, 2004. p. 3577-3582.

[9] Python Software Foundation, [viewed date: 11 December 2021]. Available From: <https://www.python.org/doc/essays/blurb/>

[10] VANROSSUM, Guido; DRAKE, Fred L. *The python language reference*. Amsterdam, Netherlands: Python Software Foundation, 2010.

[11] DAS, Plaban. *Risk analysis of autonomous vehicle and its safety impact on mixed traffic stream*. Rowan University, 2018.

[12] ZHANG, Jin, et al. Analyzing Influence of Robustness of Neural Networks on the Safety of Autonomous Vehicles. In: *31th European Safety and Reliability Conference (Forthcoming)*. 2021.

[13] SURESH, P. V.; BABAR, A. K.; RAJ, V. Venkat. Uncertainty in fault tree analysis: A fuzzy approach. *Fuzzy sets and Systems*, 1996, 83.2: 135-141.