NSS

Neutrosophic Sets and Systems, Vol. 78, 2025

University of New Mexico

A Python Framework Enhancement for Neutrosophic Topologies

Giorgio Nordo[1,*], Saeid Jafari[2] and Maikel Yelandi Leyva Vázquez[3]

[1]MIFT Department – Mathematical and Computer Science, Physical Sciences and Earth Sciences University of Messina, 98166 Sant'Agata, Messina, Italy; giorgio.nordo@unime.it

[2]Mathematical and Physical Science Foundation, 4200 Slagelse, Denmark; jafaripersia@gmail.com

[3] Salesian Polytechnic University of Guayaquil, Guyas, Ecuador; mleyvaz@gmail.com

[*]Correspondence: giorgio.nordo@unime.it

Abstract. This paper introduces an extension to the Python Neutrosophic Sets (PYNS) framework, originally detailed in [13], with the addition of the `NSfamily` class for constructing and manipulating neutrosophic topologies. Building on existing classes like `NSuniverse` and `NSset`, the `NSfamily` class enables the definition and testing of neutrosophic families as basis and sub-basis for neutrosophic topological spaces. This extension provides tools for verifying closure properties under union and intersection, and for determining whether a given family constitutes a neutrosophic topology. Through implemented algorithms, the framework automates the generation of topologies from families of neutrosophic sets, offering an efficient tool for advancing research in neutrosophic topology. Practical applications are demonstrated with detailed examples, showcasing how this class enhances the scope and flexibility of neutrosophic modeling within the PYNS framework.

Keywords: Neutrosophic topology, Neutrosophic set, Python framework, Neutrosophic family, Neutrosophic topological basis, Neutrosophic topological sub-basis.

2020 Mathematics Subject Classification: 03E72, 54A40, 54D99, 68W99.

---

1. Introduction

The theory of neutrosophic sets, introduced by Smarandache in 1999 [25], generalizes both fuzzy sets [29] and Atanassov's intuitionistic fuzzy sets [1] by providing three distinct parameters—truth, indeterminacy, and falsity—for each element. This generalization offers a more flexible and nuanced model for representing uncertainty and incomplete information, making it applicable to a wide range of fields, including statistics [26] and image processing [30]. In addition, single-valued neutrosophic sets have been explored to represent degrees of truth, indeterminacy, and falsity within a simplified framework [28], with applications in graph theory and decision making [2, 12].

In recent years, neutrosophic topology has emerged as an extension of classical topological concepts to the neutrosophic domain. Key developments include the exploration of connectedness and stratification in neutrosophic topological spaces by Broumi et al. [3] and the investigation of separation axioms by Dey and Ray [4]. Additionally, foundational studies by Gallego Lupiáñez [6, 7, 16] have laid the groundwork for applying neutrosophic theory to topology, creating opportunities to model complex relationships characterized by uncertainty.

The practical application of neutrosophic sets has been greatly enhanced by the development of computational tools. El-Ghareeb introduced an open-source Python package to handle neutrosophic data [5], and Sleem et al. developed the PyIVNS tool for interval-valued neutrosophic operations [24]. In addition, Topal et al. created a Python tool for implementing operations on bipolar neutrosophic matrices, useful in applications requiring complex matrix computations [27]. Furthermore, Saranya et al. developed a C# application specifically designed to handle neutrosophic $g\alpha$-closed sets, expanding the accessibility of neutrosophic topology across different programming environments [23]. However, these tools are limited in scope and focus primarily on specific neutrosophic operations rather than providing a comprehensive and interactive framework for dealing with symbolic representations of neutrosophic sets.

The Python Neutrosophic Sets (PYNS) framework, developed by Nordo et al. [13], addresses this limitation by offering a robust, modular library for the representation and manipulation of neutrosophic sets and mappings. PYNS enables the modeling of various neutrosophic structures, such as single-valued neutrosophic mappings [9] and neutrosophic soft topological spaces [10, 11]. However, while PYNS supports essential topological operations, such as basis and sub-basis construction, it lacks a generalized framework for systematically defining and verifying neutrosophic topologies.

In this paper, we introduce the `NSfamily` class as an extension of the PYNS framework, aimed at facilitating the construction, manipulation, and verification of neutrosophic topologies. This new class draws upon theoretical work by Ozturk [14] and Salama et al. [18,20–22] on neutrosophic sets and topological structures, offering a more integrated approach for defining families of neutrosophic sets as basis and sub-basis. The `NSfamily` class supports automated generation of neutrosophic topologies, including operations to verify closure under union and intersection, providing an efficient and scalable solution for neutrosophic topological modeling.

Furthermore, `NSfamily` addresses a broad range of computational requirements in neutrosophic topology, including the ability to handle interval-valued data and complex neutrosophic structures. This flexibility makes it suitable for both theoretical research and practical applications, as demonstrated in Rabuni and Balaman's approach to neutrosophic soft topologies using Python [15], and in applications of single-valued neutrosophic ideals by Saber et al. [17].

Our contribution extends the PYNS framework with a cohesive and efficient tool for neutrosophic topological modeling, addressing the computational needs of researchers and expanding the possibilities of neutrosophic analysis in complex domains.

The structure of the paper is as follows. In Section 2, we review fundamental definitions and properties in neutrosophic topology, including closure and continuity. Section 3 details the `NSfamily` class, highlighting the core methods and their applications. Finally, we provide practical examples that illustrate the enhanced capabilities of PYNS with the `NSfamily` extension, discussing the potential impact of this work on future research in neutrosophic topology.

2. **Preliminaries**

This section provides a comprehensive overview of neutrosophic sets, neutrosophic topology, and the structure of the Python Neutrosophic Sets (PYNS) framework. This background is necessary to understand the design and functionality of the new `NSfamily` class, which extends PYNS for advanced neutrosophic topological modeling.

2.1. *Neutrosophic Sets*

The concept of neutrosophic sets, introduced by Smarandache [25], generalizes classical, fuzzy [29], and intuitionistic fuzzy sets [1] by assigning each element three independent degrees: truth, indeterminacy, and non-membership, each within the hyperreal interval $]0^-, 1^+[$ of the nonstandard real numbers. A simpler variant, the single-valued neutrosophic set [28], uses the standard interval $[0, 1]$, making it more accessible and practical for scientific and engineering applications.

Definition 2.1. [28] *Let $\mathbb{U}$ be a universal set and $A \subseteq \mathbb{U}$. A single-valued neutrosophic set (abbreviated SVN-set) over $\mathbb{U}$, denoted by $\widetilde{A} = \langle \mathbb{U}, \mu_A, \sigma_A, \omega_A \rangle$, is defined as:*

$$\widetilde{A} = \{(u, \mu_A(u), \sigma_A(u), \omega_A(u)) : u \in \mathbb{U}\}$$

where $\mu_A : \mathbb{U} \to I$, $\sigma_A : \mathbb{U} \to I$, and $\omega_A : \mathbb{U} \to I$ are the membership, indeterminacy, and non-membership functions of A, respectively, and $I = [0, 1]$ denotes the real unit interval. For every $u \in \mathbb{U}$, $\mu_A(u)$, $\sigma_A(u)$, and $\omega_A(u)$ are called the degree of membership, degree of indeterminacy, and degree of non-membership of u, respectively.

Definition 2.2. [25, 28] *Let $\widetilde{A} = \langle \mathbb{U}, \mu_A, \sigma_A, \omega_A \rangle$ and $\widetilde{B} = \langle \mathbb{U}, \mu_B, \sigma_B, \omega_B \rangle$ be two SVN-sets over the universe set $\mathbb{U}$, we say that:*

- *$\widetilde{A}$ is a neutrosophic subset (or simply a subset) of $\widetilde{B}$ and we write $\widetilde{A} \Subset \widetilde{B}$ if, for every $u \in \mathbb{U}$, it results $\mu_A(u) \leq \mu_B(u)$, $\sigma_A(u) \leq \sigma_B(u)$ and $\omega_A(u) \geq \omega_B(u)$. We also say* *that $\widetilde{A}$ is contained in $\widetilde{B}$ or that $\widetilde{B}$ contains $\widetilde{A}$ and we write $\widetilde{B} \Supset \widetilde{A}$ to denote that $\widetilde{B}$ is a neutrosophic superset of $\widetilde{A}$.*

- *$\widetilde{A}$ is a neutrosophically equal (or simply equal) to $\widetilde{B}$ and we write $\widetilde{A} \equiv \widetilde{B}$ if $\widetilde{A} \Subset \widetilde{B}$ and $\widetilde{B} \Subset \widetilde{A}$.*

Notation 2.3. *Let $\mathbb{U}$ be a set, $I = [0, 1]$ the unit interval of the real numbers, for every $r \in I$, with $\underline{r}$ we denote the constant mapping $\underline{r} : \mathbb{U} \to I$ defined by $\underline{r}(u) = r$, for every $u \in \mathbb{U}$.*

Definition 2.4. [28] *Given a universe set $\mathbb{U}$:*

- *the SVN-set $\langle \mathbb{U}, \underline{0}, \underline{0}, \underline{1} \rangle$ is said to be the neutrosophic empty set over $\mathbb{U}$ and it is denoted by $\widetilde{\emptyset}$, or more precisely by $\widetilde{\emptyset}_{\mathbb{U}}$ in case it is necessary to specify the corresponding universe set*
- *the SVN-set $\langle \mathbb{U}, \underline{1}, \underline{1}, \underline{0} \rangle$ is said to be the neutrosophic absolute set over $\mathbb{U}$ and it is denoted by $\widetilde{\mathbb{U}}$.*

On neutrosophic sets, union, intersection and complement operations can be defined.

Definition 2.5. [19] *Let $\left\{\widetilde{A}_\alpha\right\}_{\alpha\in\Lambda}$ be a family of SVN-sets $\widetilde{A}_\alpha = \langle \mathbb{U}, \mu_{A_\alpha}, \sigma_{A_\alpha}, \omega_{A_\alpha} \rangle$ over a common universe set $\mathbb{U}$, then:*

- *the neutrosophic union, denoted by $\bigcup\limits_{\alpha\in\Lambda} \widetilde{A}_\alpha$, is the neutrosophic set $\widetilde{A} = \langle \mathbb{U}, \mu_A, \sigma_A, \omega_A \rangle$ with $\mu_A = \bigvee\limits_{\alpha\in\Lambda} \mu_{A_\alpha}$, $\sigma_A = \bigvee\limits_{\alpha\in\Lambda} \sigma_{A_\alpha}$, and $\omega_A = \bigwedge\limits_{\alpha\in\Lambda} \omega_{A_\alpha}$. In particular, the neutrosophic union of two SVN-sets $\widetilde{A} = \langle \mathbb{U}, \mu_A, \sigma_A, \omega_A \rangle$ and $\widetilde{B} = \langle \mathbb{U}, \mu_B, \sigma_B, \omega_B \rangle$, denoted by $\widetilde{A} \Cup \widetilde{B}$, is the neutrosophic set defined by $\langle \mathbb{U}, \mu_A \vee \mu_B, \sigma_A \vee \sigma_B, \omega_A \wedge \omega_B \rangle$*
- *the neutrosophic intersection, denoted by $\bigcap\limits_{\alpha\in\Lambda} \widetilde{A}_\alpha$, is the neutrosophic set $\widetilde{A} = \langle \mathbb{U}, \mu_A, \sigma_A, \omega_A \rangle$ with $\mu_A = \bigwedge\limits_{\alpha\in\Lambda} \mu_{A_\alpha}$, $\sigma_A = \bigwedge\limits_{\alpha\in\Lambda} \sigma_{A_\alpha}$, and $\omega_A = \bigvee\limits_{\alpha\in\Lambda} \omega_{A_\alpha}$. In particular, the neutrosophic intersection of two SVN-sets $\widetilde{A} = \langle \mathbb{U}, \mu_A, \sigma_A, \omega_A \rangle$ and $\widetilde{B} = \langle \mathbb{U}, \mu_B, \sigma_B, \omega_B \rangle$, denoted by $\widetilde{A} \Cap \widetilde{B}$, is the neutrosophic set defined by $\langle \mathbb{U}, \mu_A \wedge \mu_B, \sigma_A \wedge \sigma_B, \omega_A \vee \omega_B \rangle$*

where $\wedge$ and $\vee$ denote the minimum and the maximum, respectively.

Definition 2.6. [25,28] *The neutrosophic complement of a SVN-set $\widetilde{A} = \langle \mathbb{U}, \mu_A, \sigma_A, \omega_A \rangle$ over $\mathbb{U}$, denoted by $\widetilde{A}^{\complement}$, is given by: $\widetilde{A}^{\complement} = \langle \mathbb{U}, \omega_A, \underline{1} - \sigma_A, \mu_A \rangle$.*

2.2. *Neutrosophic Topology*

Neutrosophic topology generalizes classical topology by defining open and closed sets through neutrosophic set theory, enabling topologies where openness, closedness, and indeterminacy are graded. According to Salama et al [18], Serkan Karatas et al [8] and Gallego

Lupiáñez [6, 7], a neutrosophic topology is a family of neutrosophic sets which contains the empty neutrosophic set, the absolute neutrosophic set and is closed with respect to the neutrosophic union and the finite neutrosophic intersection.

Definition 2.7. *[6, 18] Let $\mathscr{T}$ be a family of neutrosophic sets over the same universe set $\mathbb{U}$, we say that $\mathscr{T}$ is a neutrosophic topology if the following four conditions hold:*

(1) $\widetilde{\emptyset} \in \mathscr{T}$

(2) $\widetilde{\mathbb{U}} \in \mathscr{T}$

(3) $\forall \mathscr{A} \subseteq \mathscr{T}, \bigcup\!\!\!\!\bigcup \mathscr{A} \in \mathscr{T}$

(4) $\forall \widetilde{U}, \widetilde{V} \in \mathscr{T}, \widetilde{U} \Cap \widetilde{V} \in \mathscr{T}$

and when this occurs we also say that the pair $(\mathbb{U}, \mathscr{T})$ constitutes a neutrosophic topological space, while the elements of $\mathscr{T}$ are named neutrosophic open sets.

Although, the authors are not aware of papers in which notions such as bases and subbasis of neutrosophic topologies are introduced and treated, by simple analogy with classical topological spaces, the following general definitions and basic properties can be introduced.

Definition 2.8. *Given two nutrosophic topologies $\mathscr{T}_1$ and τ_2 over same universe set $\mathbb{U}$, we say that $\mathscr{T}_1$ is coarser (weaker or smaller) than $\mathscr{T}_2$, or equivalently, that $\mathscr{T}_2$ is finer (stronger or larger) than $\mathscr{T}_1$ if $\mathscr{T}_1 \subseteq \mathscr{T}_2$.*

So, the inclusion relation $\subseteq$ is a partial order on the set of all neutrosophic topologies over a universe set $\mathbb{U}$.

Proposition 2.9. *The intersection $\bigcap\limits_{\alpha\in\Lambda} \mathscr{T}_\alpha$ of any family $\{\mathscr{T}_\alpha\}_{\alpha\in\Lambda}$ of neutrosophic topologies over a common universe set $\mathbb{U}$ is itself a neutrosophic topology on $\mathbb{U}$.*

Proposition 2.10. *Given a family $\mathscr{S}$ of neutrosophic subsets over a common universe set $\mathbb{U}$, there exists and is unique the smallest (minimal) neutrosophic topology $\mathscr{T}(\mathscr{S})$ on $\mathbb{U}$ containing $\mathscr{S}$.*

Definition 2.11. *Let $\mathscr{S}$ be a family of neutrosophic subsets over a common universe set $\mathbb{U}$, the neutrosophic topology $\mathscr{T}(\mathscr{S})$ of Proposition 2.10 is called the neutrosophic topology generated by the family $\mathscr{S}$.*

Proposition 2.12. *If $\mathscr{S}_1$ and $\mathscr{S}_2$ are two families of neutrosophic sets on the same universe set $\mathbb{U}$ such that $\mathscr{S}_1 \subseteq \mathscr{S}_2$ then the corresponding generated neutrosophic topologies satisfy $\mathscr{T}(\mathscr{S}_1) \subseteq \mathscr{T}(\mathscr{S}_2)$.*

Definition 2.13. *Given a neutrosophic topological space $(\mathbb{U}, \mathscr{T})$, a family $\mathscr{B} \subseteq \mathscr{T}$ is said to be a neutrosophic basis for the neutrosophic topology $\mathscr{T}$ if every neutrosophic open set in $\mathscr{T}$*

can be expressed as a neutrosophic union of a subfamily of $\mathscr{B}$*, i.e., if for each* $\widetilde{U} \in \mathscr{T}$*, there exists* $\mathscr{A} \subseteq \mathscr{B}$ *such that* $\widetilde{U} = \bigsqcup \mathscr{A}$*.*

Notation 2.14. *Given a family* $\mathscr{S}$ *of neutrosophic sets over a common universe set* $\mathbb{U}$*, let us denote by* $\mathscr{B}(\mathscr{S})$ *(or, equivalently, by* $\mathscr{S}^*$*) the family of all its finite neutrosophic intersections, with the addition of the absolute neutrosophic set, i.e:*

$$\mathscr{B}(\mathscr{S}) = \left\{ \bigsqcap_{i=1}^{n} \widetilde{S}_i : n \in \mathbb{N}, \widetilde{S}_i \in \mathscr{S} \right\} \Cup \left\{ \widetilde{\mathbb{U}} \right\}.$$

Proposition 2.15. *If* $\mathscr{S}_1$ *and* $\mathscr{S}_2$ *are two families of neutrosophic sets on the same universe set* $\mathbb{U}$ *such that* $\mathscr{S}_1 \subseteq \mathscr{S}_2$ *then the corresponding family of all their finite neutrosophic intersections satisfy* $\mathscr{B}(\mathscr{S}_1) \subseteq \mathscr{B}(\mathscr{S}_2)$*.*

Proposition 2.16. *The neutrosophic topology* $\mathscr{T}(\mathscr{S})$ *generated by a family of neutrosophic sets* $\mathscr{S}$ *over the same universe set* $\mathbb{U}$ *coincides with the topology* $\mathscr{T}(\mathscr{B}(\mathscr{S}))$ *generated by the family of all finite neutrosophic intersections* $\mathscr{B}(\mathscr{S})$ *of* $\mathscr{S}$*, i.e., we have:*

$$\mathscr{T}(\mathscr{S}) = \mathscr{T}(\mathscr{B}(\mathscr{S}))$$

and the family $\mathscr{B}(\mathscr{S})$ *of all finite neutrosophic intersections forms a neutrosophic basis for* $\mathscr{T}(\mathscr{S})$*.*

Definition 2.17. *In the situation described by Proposition 2.16, the* $\mathscr{S}$ *family is said to be a neutrosophic sub-basis (or a neutrosophic pre-base) of the* $\mathscr{T}(\mathscr{S})$ *topology.*

2.3. *The PYNS Framework*

The Python Neutrosophic Sets (PYNS) framework, developed by Nordo et al. [13], supports essential components for neutrosophic modeling. It is composed of three primary classes: `NSuniverse`, `NSset` and `NSmapping` designed to manage neutrosophic universes, individual neutrosophic sets and functions between neutrosophic sets, respectively.

The PYNS framework includes three primary classes that enable the representation and manipulation of neutrosophic structures in Python. These are:

- `NSuniverse` that defines a universe set over which neutrosophic sets are established. It supports initialization with various formats, such as lists, tuples, and strings. Key methods include:
  - `get()` - Returns the elements of the universe set
  - `cardinality()` - Returns the total count of elements
  - `isSubset(unv)` - Checks if the current universe set is a subset of another specified universe set.

- `NSset` which represents individual neutrosophic sets, where each element in the universe has associated degrees of membership, indeterminacy, and non-membership. Primary methods include:
  - `NSunion(nset)` - Computes the union of the current neutrosophic set with another specified set
  - `NSintersection(nset)` - Finds the intersection of the current set with another
  - `NScomplement()` - Generates the complement of the neutrosophic set, flipping the membership, indeterminacy, and non-membership values
  - `NSdifference(nset)` - Calculates the neutrosophic difference between the current set and another
  - `isNSdisjoint(nset)` - Verifies if the current set is disjoint from the given neutrosophic set.
- `NSmapping` that is designed to manage mappings between two neutrosophic universe sets, each instance of this class holds a dictionary to represent the mapping relations. It includes methods such as:
  - `getDomain()` - Retrieves the domain of the mapping as an `NSuniverse` object
  - `getCodomain()` - Returns the codomain as an `NSuniverse` object
  - `getMap()` - Provides access to the dictionary of element-value pairs defining the mapping
  - `setValue(u, v)` - Assigns a codomain element v to a domain element u
  - `getValue(u)` - Returns the mapping value of a given element u from the domain
  - `getFibre(v)` - Identifies all elements in the domain that map to a specified codomain value v, effectively finding the fibre of v
  - `NSimage(nset)` - Computes the neutrosophic image of a given neutrosophic set
  - `NScounterimage(nset)` - Determines the neutrosophic counterimage for the specified neutrosophic set.

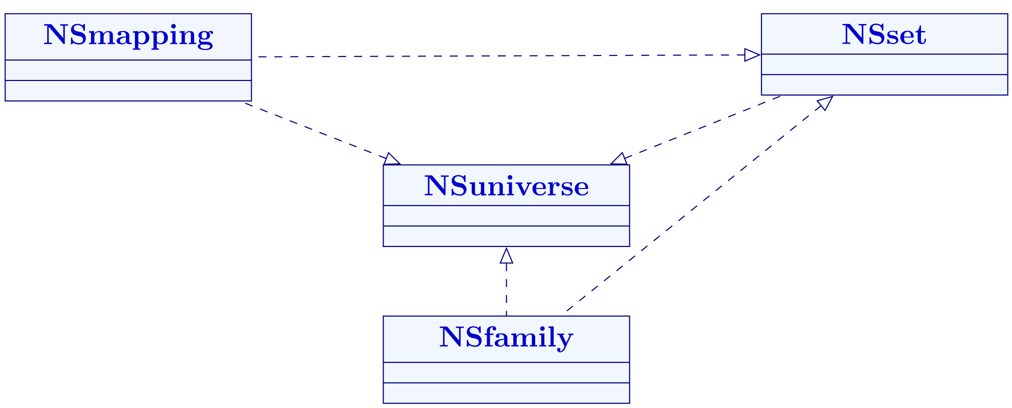
In summary, the `PYNS` framework, as developed by Nordo et al. [13], provides a robust foundation for neutrosophic modeling in Python, structured around the core classes `NSuniverse`, `NSset`, and `NSmapping`. With approximately 1500 lines of code, `PYNS` supports intuitive operations on neutrosophic universes, individual sets, and mappings. Key functionalities include fundamental operations such as neutrosophic union, intersection, difference, and complement, as well as the computation of images and counterimages through mappings, all of which contribute to a flexible toolset for handling neutrosophic structures across various types of universes.

The modular design of `PYNS` not only supports interactive use, ideal for experimentation and the exploration of neutrosophic properties, but also facilitates integration into larger Python projects due to its adaptable architecture and comprehensive, example-rich documentation. This ease of use and flexibility makes the PYNS framework a valuable open source tool for exploring the properties of neutrosophic sets, with applications spanning diverse areas of research that benefit from neutrosophic set theory.

Despite its comprehensive capabilities, PYNS currently lacks a mechanism for systematically managing collections of neutrosophic sets that could serve as bases or sub-bases in topological constructions. The ability to handle such collections is crucial for building and verifying topological structures in neutrosophic settings, which remain essential for advanced applications and theoretical explorations in neutrosophic topology. This limitation is addressed by extending the PYNS framework with the NSfamily class, presented in this paper which provides the necessary functionality to manage and interact with families of neutrosophic sets and neutrosophic topologies.

3. The NSfamily Class

The NSfamily class enhances the PYNS framework by enabling the construction and management of neutrosophic families, which serve as bases or sub-bases for topology generation. Inspired by Ozturk's theoretical work on neutrosophic topology [14] and Salama et al.'s object-oriented design principles [20,21], NSfamily provides a unified interface for creating topologies and verifying closure properties within neutrosophic environments. By adding NSfamily to the other three classes NSuniverse, NSset and NSmapping above described, the framework PYNS now supports topological operations on families of neutrosophic sets within the same universe, bridging a key gap in the framework and enabling systematic topology construction. The UML diagram below illustrates the relationships among the classes, with dashed arrows indicating "uses" relationships.



The NSfamily class manages families of neutrosophic sets stored as lists of objects of the class NSset sharing the same universe set and can be shortly described by means of its main properties and methods in the following UML class diagram.

NSfamily
__neutrosophicfamily : list of NSset
__init__(*args) : constructor with generic argument storeName() : save the name of the family as a property of the object itself setUniverse() : sets the universe of the neutrosophic sets family cardinality() : returns the number of elements of the neutrosophic family isSubset(nsfamily) : checks if is contained in that passed as parameter isSuperset(nsfamily) : checks if contains that passed as parameter union(nsfamily) : returns the union with another one intersection(nsfamily) : returns the intersection with another one isDisjoint(nsfamily) : checks if is disjoint with another one difference(nsfamily) : returns the set difference with another one complement() : returns the family of neutrosophically complemented neutrosophic sets getNSBase() : returns the neutrosophic topological base containing the current family getNSTopologyByBase() : returns the neutrosophic topology from a neutrosophic base getNSTopologyBySubBase() : returns the neutrosophic topology from the given family isNeutrosophicTopology() : checks if the family forms a neutrosophic topology

The `NSfamily` constructor initializes a family of neutrosophic sets over a shared universe. The initialization process is detailed in the following algorithm:

Constructor method of the class `NSfamily`

```
Function __init__(args):
    Initialize the private property neutrosophicfamily as an empty list
    Get the length of args
    if length = 0 then
        Set universe to None
    else if length = 1 then
        Let elem be the first and only element in args
        if elem is an instance of NSset then
            Add elem to neutrosophicfamily and set universe to the universe of elem
        else if elem is a list or tuple then
            foreach e in elem do
                if e is not in neutrosophicfamily then
                    Set the name of e and add it to neutrosophicfamily
            if neutrosophicfamily is not empty then
                Set universe to the universe of the first element in neutrosophicfamily
            else
                Set universe to None
    else if length > 1 then
        foreach e in args do
            if e is not in neutrosophicfamily then
                Set the name of e and add it to neutrosophicfamily
        Set universe to the universe of the first element in args
    Store universe, neutrosophicfamily as private properties
    Set the private property name to None
```

The Python code corresponding to the constructor method of the `NSfamily` class is given below.

```
from .ns_universe import NSuniverse
from .ns_set import NSset
from .ns_util import NSreplace,NSstringToDict,NSisExtDict,nameToBB,isBB
import inspect
from itertools import combinations
from functools import reduce
from time import time

class NSfamily:

    def __init__(self, *args):
        neutrosophicfamily = list()
        length = len(args)
        if length == 0:
            universe = None
        elif length == 1:
            elem = args[0]
            if type(elem) == NSset:
                neutrosophicfamily = [elem]
                universe = elem.getUniverse()
            elif type(elem) in [list ,tuple]:
                for e in elem:
                    if e not in neutrosophicfamily:
                        e.setName(e.getName())
                        neutrosophicfamily.append(e)
                if len(neutrosophicfamily) > 0:
                    universe = neutrosophicfamily[0].getUniverse()
                else:
                    universe = None
        elif length > 1:
            for e in args:
                if e not in neutrosophicfamily:
                    e.setName(e.getName())
                    neutrosophicfamily.append(e)
            universe = args[0].getUniverse()
        self.__universe = universe
        self.__neutrosophicfamily = neutrosophicfamily
        self.__name = None
```

The `NSfamily` constructor offers flexible initialization to handle various cases in creating neutrosophic set families, managing arguments through the `*args` parameter. This enables creation from individual sets, collections, or copies of existing `NSfamily` objects:

- empty family: `NSfamily()` creates an empty family with no universe, allowing elements to be added incrementally
- single neutrosophic set: `NSfamily(ns_set)` initializes a family with one set, inheriting its universe

- list or tuple of sets: `NSfamily([ns_set1, ns_set2, ...])` or `NSfamily((ns_set1, ns_set2, ...))` accepts a collection, avoiding duplicates and setting the universe based on the first set
- multiple sets as arguments: `NSfamily(ns_set1, ns_set2, ...)` creates a family from the specified sets, with the universe derived from the first neutrosophic set, ensuring no duplicate entries
- existing `NSfamily` object: `NSfamily(existing_family)` copies an existing family, creating an independent object with the same universe.

This versatility allows the `NSfamily` class to support a broad range of initial configurations, thereby facilitating the manipulation and management of neutrosophic set families.

In order to conveniently display families of neutrosophic sets in both simplified and tabular text formats while providing a comprehensive representation, the special methods `__str__()` and `__format__()` have been implemented. These methods allow customization of the output format, especially when using the **print** function with f-strings for displaying `NSfamily` objects. The `__format__()` method supports the following format specifiers:

s: : simple format (default), offering a straightforward textual representation

t: : tabular format, presenting data in a structured and neatly aligned manner

l: : includes a label with the `NSfamily` object's name, if available

x: : extended format, providing additional spacing in the first column to accommodate longer names or labels.

```
def __str__(self, tabularFormat=False, label=False, extended=False):
    labelname = ""
    if label and self.__name is not None:
        labelname = f"{self.__name} = "
    if not self.__neutrosophicfamily:
        return labelname + "\u2205"
    indentation = " "*(len(labelname) + 2)
    if not tabularFormat:
        if extended:
            items = [a.__str__(tabularFormat, True, extended)
                     for a in self.__neutrosophicfamily]
            formatted_lines=[", ".join(items[i:i + 2]) for i in range(0,len(items),2)]
            return labelname+"{ "+f",\n{indentation}".join(formatted_lines)+" }\n"
        else:
            return labelname + "{ " + ", ".join(
                [a.__str__(tabularFormat, True, extended)
                 for a in self.__neutrosophicfamily]) + " }\n"
    res = labelname + "{ " + ", ".join(
        [a.__str__(tabularFormat, True, extended)
         for a in self.__neutrosophicfamily]) + " }\n"
    return res
```

```python
def __format__(self, spec):
    label = "l" in spec
    extended = "x" in spec
    if "t" in spec:
        result = self.__str__(tabularFormat=True, label=label, extended=extended
    )
    else:
        result = self.__str__(tabularFormat=False, label=label, extended=
    extended)
    return result
```

Some basic methods facilitate the management of neutrosophic family properties. The `storeName()` method captures the object's name from the local context, while `getName()` retrieves it if stored. `getUniverse()` returns the associated `NSuniverse` of the neutrosophic family, and `setUniverse()` sets it, ensuring it is a valid `NSuniverse` instance. Finally, `cardinality()` provides the count of neutrosophic sets within the family. Their code is given below.

```python
def storeName(self):
    frame = inspect.currentframe().f_back
    local_vars = frame.f_locals
    var_name = next((nm for nm, vl in local_vars.items()
                        if vl is self), None)
    self.__name = var_name

def getName(self):
    return self.__name

def getUniverse(self):
    return self.__universe

def setUniverse(self, universe):
    if type(universe) != NSuniverse:
        raise ValueError("The parameter's type must be NSuniverse.")
    self.__universe = universe

def cardinality(self):
    return len(self.__neutrosophicfamily)
```

Let us illustrate what just said with an example of code executed interactively in the Python console.

```python
>>> from NS.pyns.ns_universe import NSuniverse
>>> from NS.pyns.ns_set import NSset
>>> from NS.pyns.ns_family import NSfamily
>>> U = NSuniverse("a,b,c")
>>> A1 = NSset(U, "(0.4,0.4,0.3), (0.1,0.1,0.1), (0.2,0.2,0.2)")
>>> A2 = NSset(U, "(0.1,0.2,0.9), (0.9,0.1,0.3), (0.5,0.3,0.4)")
>>> A1.storeName()
>>> A2.storeName()
>>> E = NSset.EMPTY(U)
>>> L = NSfamily(E, A1, A2)
```

```
>>> L.storeName()
>>> print(f"family {L:lx}")
family L = { A1 = < a/(0.4,0.4,0.3), b/(0.1,0.1,0.1), c/(0.2,0.2,0.2) >,
      A2 = < a/(0.1,0.2,0.9), b/(0.9,0.1,0.3), c/(0.5,0.3,0.4) > }
```

The following methods provide essential operations for managing and comparing neutrosophic families. These methods allow for determining subset and superset relationships, computing unions and intersections of families, checking for disjoint sets, and calculating differences and complements.

The `isSubset()` method verifies if the current neutrosophic family is contained within another, while the `__le__()` method overloads the <= operator to provide a more intuitive comparison.

```
def isSubset(self, nsfamily):
    if type(nsfamily) != NSfamily:
        raise ValueError("the parameter is not a neutrosophic family")
    if self.getUniverse() != nsfamily.getUniverse():
        raise ValueError("the two neutrosophic families cannot be defined on different universe sets")
    else:
        result = True
        for e in self.__neutrosophicfamily:
            if e not in nsfamily.__neutrosophicfamily:
                result = False
                break
        return result

def __le__(self, nsfamily):
    if type(nsfamily) != NSfamily:
        raise ValueError("the second argument is not a neutrosophic family")
    return self.isSubset(nsfamily)
```

Similarly, the `isSuperset()` method checks if the current neutrosophic family contains another, with the `__ge__()` method overloading the >= operator to simplify this comparison.

```
def isSuperset(self, nsfamily):
    if type(nsfamily) != NSfamily:
        raise ValueError("the parameter is not a neutrosophic family")
    if self.getUniverse() != nsfamily.getUniverse():
        raise ValueError("the two neutrosophic families cannot be defined on different universe sets")
    return nsfamily.isSubset(self)

def __ge__(self, nsfamily):
    if type(nsfamily) != NSfamily:
        raise ValueError("the second argument is not a neutrosophic family")
    return self.isSuperset(nsfamily)
```

Equality and inequality between families are determined by overloading the comparison operators == and =! through the special methods `__eq__()` and `__ne__()`.

```
def __eq__(self, nsfamily):
    if type(nsfamily) != NSfamily:
        raise ValueError("the second argument is not a neutrosophic family")
    if self.getUniverse() != nsfamily.getUniverse():
        raise ValueError("the two neutrosophic families cannot be defined on different universe sets")
    equal = self.isSubset(nsfamily) and nsfamily.isSubset(self)
    return equal

def __ne__(self, nsfamily):
    if type(nsfamily) != NSfamily:
        raise ValueError("the second argument is not a neutrosophic family")
    if self.getUniverse() != nsfamily.getUniverse():
        raise ValueError("the two neutrosophic families cannot be defined on different universe sets")
    different = not (self == nsfamily)
    return different
```

The `union()` method, along with its operator + overloaded by `__add__()`, combines two neutrosophic families into one, eliminating duplicates and ensuring they share the same universe.

```
def union(self, nsfamily):
    if not isinstance(nsfamily, NSfamily):
        raise ValueError("The parameter is not a neutrosophic family.")
    if self.getUniverse() != nsfamily.getUniverse():
        raise ValueError("The two neutrosophic families cannot be defined
            on different universes.")
    combined_sets = self.__neutrosophicfamily +
            [s for s in nsfamily.__neutrosophicfamily
                    if s not in self.__neutrosophicfamily]
    return NSfamily(combined_sets)

def __add__(self, nsfamily):
    if not isinstance(nsfamily, NSfamily):
        raise ValueError("the second argument is not a neutrosophic family")
    return self.union(nsfamily)
```

The `intersection()` method identifies common elements between two families, facilitated by the & operator via `__and__()`, while `isDisjoint()` checks for disjointness.

```
def intersection(self, nsfamily):
    if not isinstance(nsfamily, NSfamily):
        raise ValueError("The parameter is not a neutrosophic family.")
    if self.getUniverse() != nsfamily.getUniverse():
        raise ValueError("The two neutrosophic families cannot be defined on different universes.")
```

```python
    common_sets = [s for s in self.__neutrosophicfamily
                   if s in nsfamily.__neutrosophicfamily]
    return NSfamily(common_sets)

def __and__(self, nsfamily):
    if not isinstance(nsfamily, NSfamily):
        raise ValueError("the second argument is not a neutrosophic family")
    return self.intersection(nsfamily)

def isDisjoint(self, nsfamily):
    nsemptyfamily = NSfamily()
    nsemptyfamily.setUniverse(self.getUniverse())
    intersez = self.intersection(nsfamily)
    disjoint = intersez.cardinality() == 0
    return disjoint
```

The `difference()` method and its − operator via `__sub__()` return the elements unique to the first family, and `complement()` along with ˜ (overloaded by the special method `__invert__()`) yields the family consisting of the neutrosophic complements of the neutrosophic sets of the original family.

```python
def difference(self, nsfamily):
    if not isinstance(nsfamily, NSfamily):
        raise ValueError("The parameter is not a neutrosophic family.")
    if self.getUniverse() != nsfamily.getUniverse():
        raise ValueError("The two neutrosophic families cannot be defined on different universes.")
    unique_sets = [s for s in self.__neutrosophicfamily
                       if s not in nsfamily.__neutrosophicfamily]
    return NSfamily(unique_sets)

def __sub__(self, nsfamily):
    if not isinstance(nsfamily, NSfamily):
        raise ValueError("The second argument is not a neutrosophic family")
    return self.difference(nsfamily)

def complement(self):
    complementary_sets = [ns_set.NScomplement()
                              for ns_set in self.__neutrosophicfamily]
    return NSfamily(complementary_sets)

def __invert__(self):
    return self.complement()
```

From Proposition 2.16 and Definition 2.17 we know that any family of neutrosophic sets over a same universe set is a neutrosophic sub-basis for the neutrosophic topology $\mathscr{T}(\mathscr{S})$ generated by $\mathscr{S}$ which has the family $\mathscr{B}(\mathscr{S})$ of all finite neutrosophic intersections of $\mathscr{S}$ as neutrosophic basis.

The method `getNSBase` returns a neutrosophic topological basis from the current neutrosophic family regarded as a sub-basis by generating all its possible neutrosophic intersections.

Method getNSBase

```
Function getNSBase():
    Let subbase be self.__neutrosophicfamily
    Initialize an empty list base
    for i ← 1 to len(subbase) do
        foreach combination in combinations(subbase, i) do
            Compute intersection by applying NSintersection on all elements in
             combination
            if intersection_name is not empty then
                Set the name of intersection to intersection_name
            foreach s in subbase do
                if intersection equals s then
                    Set the name of intersection to name of s
                    break
            Add intersection to base
    Convert base to an NSfamily object
    Set the universe of base to self.getUniverse()
    return base
```

The Python code corresponding to the method getNSBase is given below.

```python
def getNSBase(self):
    subbase = self.__neutrosophicfamily
    base = list()
    for i in range(1, len(subbase) + 1):
        for combin in combinations(subbase, i):
            intersez = reduce(lambda x, y: x.NSintersection(y), combin)
            names = [s.getName() for s in combin if s.getName()]
            intersez_name = " ⋒ ".join(names) if names else None
            if intersez_name:
                intersez.setName(intersez_name)
            for s in subbase:
                if intersez == s:
                    intersez.setName(s.getName())
                    break
            base.append(intersez)
    base = NSfamily(base)
    base.setUniverse(self.getUniverse())
    return base
```

As an example, we show how the method described above can be used in the interactive mode by means of the Python console:

```python
>>> from pyns.ns_universe import NSuniverse
>>> from pyns.ns_set import NSset
>>> from pyns.ns_family import NSfamily
>>> U = NSuniverse("a,b,c")
>>> A1 = NSset(U, "(0.4,0.4,0.3), (0.1,0.1,0.1), (0.2,0.2,0.2)")
>>> A2 = NSset(U, "(0.1,0.2,0.9), (0.9,0.1,0.3), (0.5,0.3,0.4)")
>>> A1.storeName()
>>> A2.storeName()
```

```
>>> L = NSfamily(A1, A2)
>>> L.storeName()
>>> print(f"the neutrosophic family is {L:lt}")
 the neutrosophic family is L = {
 A1       |    membership    |  indeterminacy | non-membership |
------------------------------------------------------------
 a        | 0.4              | 0.4              | 0.3              |
 b        | 0.1              | 0.1              | 0.1              |
 c        | 0.2              | 0.2              | 0.2              |
------------------------------------------------------------
,
 A2       |    membership    |  indeterminacy | non-membership |
------------------------------------------------------------
 a        | 0.1              | 0.2              | 0.9              |
 b        | 0.9              | 0.1              | 0.3              |
 c        | 0.5              | 0.3              | 0.4              |
------------------------------------------------------------
>>> B = L.getNSBase()
>>> B.storeName()
>>> print(f"basis is {B:lx}")
basis is B = { A1 = < a/(0.4,0.4,0.3), b/(0.1,0.1,0.1), c/(0.2,0.2,0.2) >,
      A2 = < a/(0.1,0.2,0.9), b/(0.9,0.1,0.3), c/(0.5,0.3,0.4) >,
      A1 ⋒ A2 = < a/(0.1,0.2,0.9), b/(0.1,0.1,0.3), c/(0.2,0.2,0.4) > }
```

From Proposition 2.16 and Definition 2.13, we know that a base for a topology is a collection of sets from which every open set can be represented as a union of elements of the base. Specifically, in the context of neutrosophic topology, a neutrosophic base provides the foundation for generating all possible neutrosophic unions, which form the topology.
The method `getNSTopologyByBase` returns the neutrosophic topology derived from the current base. This is accomplished by generating all possible finite neutrosophic unions of the base sets, thus constructing the full topology.

Method `getNSTopologyByBase`

```
Function getNSTopologyByBase():
    Let base be self.__neutrosophicfamily
    Initialize an empty list topology
    Create an empty neutrosophic set and add it to topology
    for i ← 1 to len(base) do
        foreach combination in combinations(base, i) do
            Compute union by applying NSunion on all elements in combination
            Construct the union_name based on the combination's names
            Assign union_name to union, if applicable
            if union matches any base element then
                Update union's name accordingly
            Add union to topology
    Create an absolute neutrosophic set from the universe and add it to topology
    Convert topology to an NSfamily object
    Set the universe of topology to self.getUniverse()
    return topology
```

The Python code corresponding to the method `getNSTopologyByBase` is given below.

```python
def getNSTopologyByBase(self):
    base = self.__neutrosophicfamily
    topology = list()
    universe = self.__universe
    empty = NSset.EMPTY(universe)
    empty.setName("\u2205\u0303")
    topology.append(empty)
    for i in range(1, len(base) + 1):
        for combin in combinations(base, i):
            union = reduce(lambda x, y: x.NSunion(y), combin)
            names = [s.getName() for s in combin if s.getName()]
            union_name = " ⋓ ".join(f"({name})" if "⋒" in name else name
                                    for name in names)
            if union_name:
                union.setName(union_name)
            for b in base:
                if union == b:
                    union.setName(b.getName())
                    break
            topology.append(union)
    absolute = NSset.ABSOLUTE(universe)
    universe_name = universe.getName()
    absolute.setName(nameToBB(universe_name) if universe_name
                                 and not isBB(universe_name) else universe_name)
    topology.append(absolute)
    topology = NSfamily(topology)
    topology.setUniverse(self.getUniverse())
    return topology
```

The getNSTopologyBySubBase method constructs the neutrosophic topology derived from a subbase by computing all possible neutrosophic unions of intersections. It first generates the neutrosophic base using the getNSBase() method and then forms the complete topology by applying getNSTopologyByBase() on this base. This process ensures the creation of a full neutrosophic topology from any given family of neutrosophic sets. The Python code corresponding to this method is shown below.

```python
def getNSTopologyBySubBase(self):
    nsbase = self.getNSBase()
    nstopology = nsbase.getNSTopologyByBase()
    return nstopology
```

The isNeutrosophicTopology method verifies whether a neutrosophic family satisfies the axioms of a neutrosophic topology. It ensures the presence of both the empty set and the universal set in the family, and checks closure properties under union and intersection operations.

```python
def isNeutrosophicTopology(self):
    family = self.__neutrosophicfamily
    universe = self.__universe
    empty = NSset.EMPTY(universe)
    if empty not in family:
        return False
    absolute = NSset.ABSOLUTE(universe)
    if absolute not in family:
        return False
    if not self.NSunionClosed():
        return False
    if not self.NSintersectionClosed():
        return False
    return True
```

The isNeutrosophicTopology method relies on NSunionClosed and NSintersectionClosed, both of which delegate the closure verification to the private method __checkClosure. This method efficiently manages the verification process by iterating over all possible combinations of sets and applying the specified operation (union or intersection) provided as a lambda function. By centralizing the core logic of closure checks, __checkClosure ensures consistency and efficiency in determining whether the neutrosophic family satisfies the closure properties essential for a neutrosophic topology.

```python
def __checkClosure(self, operation, operation_name):
    family = self.__neutrosophicfamily
    l = len(family)
    for i in range(2, l + 1):
        for combin in combinations(family, i):
            result = reduce(operation, combin)
            if result not in family:
                return False
    return True


def NSunionClosed(self):
    return self.__checkClosure(lambda x,y: x.NSunion(y),"union")

def NSintersectionClosed(self:
    return self.__checkClosure(lambda x,y: x.NSintersection(y),"intersection")
```

The following code example executed interactively in the Python console illustrates the use of the above described methods.

```python
>>> from pyns.ns_universe import NSuniverse
>>> from pyns.ns_set import NSset
>>> from pyns.ns_family import NSfamily
>>> U = NSuniverse("1,2,3")
>>> B1 = NSset(U, "(0.2,0.4,0.3), (0.6,0.1,0.1), (0.4,0.6,0.3)")
>>> B2 = NSset(U, "(0.3,0.2,0.9), (0.6,0.5,0.3), (0.2,0.3,0.8)")
>>> B1.storeName()
```

```
>>> B2.storeName()
>>> S = NSfamily(B1, B2)
>>> T = S.getNSTopologyBySubBase()
>>> T.storeName()
>>> print(f"topology has cardinality {T.cardinality()} and is:\n {T:tlx}")
topology has cardinality 6 and is:
 T = {
 ∅̃             |   membership    |  indeterminacy | non-membership |
------------------------------------------------------------------
 1             | 0               | 0              | 1              |
 2             | 0               | 0              | 1              |
 3             | 0               | 0              | 1              |
------------------------------------------------------------------
,
 B1            |   membership    |  indeterminacy | non-membership |
------------------------------------------------------------------
 1             | 0.2             | 0.4            | 0.3            |
 2             | 0.6             | 0.1            | 0.1            |
 3             | 0.4             | 0.6            | 0.3            |
------------------------------------------------------------------
,
 B2            |   membership    |  indeterminacy | non-membership |
------------------------------------------------------------------
 1             | 0.3             | 0.2            | 0.9            |
 2             | 0.6             | 0.5            | 0.3            |
 3             | 0.2             | 0.3            | 0.8            |
------------------------------------------------------------------
,
 B1 ⋒ B2       |   membership    |  indeterminacy | non-membership |
------------------------------------------------------------------
 1             | 0.2             | 0.2            | 0.9            |
 2             | 0.6             | 0.1            | 0.3            |
 3             | 0.2             | 0.3            | 0.8            |
------------------------------------------------------------------
,
 B1 ⋓ B2       |   membership    |  indeterminacy | non-membership |
------------------------------------------------------------------
 1             | 0.3             | 0.4            | 0.3            |
 2             | 0.6             | 0.5            | 0.1            |
 3             | 0.4             | 0.6            | 0.3            |
------------------------------------------------------------------
,
 𝕌̃             |   membership    |  indeterminacy | non-membership |
------------------------------------------------------------------
 1             | 1               | 1              | 0              |
 2             | 1               | 1              | 0              |
 3             | 1               | 1              | 0              |
------------------------------------------------------------------
 }
>>> T.isNeutrosophicTopology()
True
```

4. Conclusions

In this paper, we introduced an extension to the Python Neutrosophic Sets (PYNS) framework with the development of the `NSfamily` class. This addition allows for the efficient construction, manipulation, and analysis of neutrosophic families and their roles as bases and sub-bases in neutrosophic topological spaces. Leveraging existing PYNS classes, `NSuniverse` and `NSset`, the new class provides robust methods for evaluating closure properties, unions, intersections, and complements, which are essential operations in topological studies.

The theoretical exploration underscored the adaptability of neutrosophic sets in addressing indeterminate and inconsistent information, a feature that differentiates them from classical set theories. The practical implementations demonstrated through detailed examples reveal the computational efficiency and flexibility of the PYNS framework, particularly in generating complete neutrosophic topologies from defined families.

Future research will aim to broaden the applicability of neutrosophic topologies, especially in decision-making scenarios characterized by high uncertainty and indeterminacy. Further enhancements to the PYNS framework are also planned to optimize performance and extend support for more complex neutrosophic operations.

This work lays a foundational framework for the integration of neutrosophic topological methods into various scientific and engineering disciplines. By providing a comprehensive toolset for the exploration and application of neutrosophic concepts, it opens new avenues for research where traditional approaches may not suffice.

The source code for the PYNS framework, including the `NSfamily` class introduced in this paper, is available at `https://github.com/giorgionordo/PythonNeutrosophicTopologies`. This repository, licensed under GPL 3.0, serves as a valuable resource for researchers and practitioners, enabling them to explore, utilize, and extend neutrosophic topologies in their projects.

Acknowledgements This research was supported by Gruppo Nazionale per le Strutture Algebriche, Geometriche e le loro Applicazioni (G.N.S.A.G.A.) of Istituto Nazionale di Alta Matematica (INdAM) "F. Severi", Italy.

References

[1] Atanassov K.T. Intuitionistic Fuzzy Sets. *Fuzzy Sets and Systems* **20** (1), pp. 87-96, 1986.

[2] Broumi S., Talea M., Bakali A., Smarandache F. Single valued neutrosophic graphs. *Journal of New theory* **10**, pp. 86-101, 2016.

[3] Broumi S., Smarandache F., Bakali A. Connectedness and Stratification of Single-Valued Neutrosophic Topological Spaces. *Neutrosophic Sets and Systems*, **36**, pp. 77-84, 2020.

[4] Dey S., Ray G.C., Separation Axioms in Neutrosophic Topological Spaces. *Neutrosophic Systems with Applications* **2**, 2023.

[5] El-Ghareeb H.A. Novel Open Source Python Neutrosophic Package. *Neutrosophic Sets and Systems* **25**, pp. 136-160, 2019.

[6] Gallego Lupiáñez F., On Neutrosophic Topology. *Kybernetes*, 2008.

[7] Gallego Lupiáñez F., On Neutrosophic Sets and Topology. *Procedia Computer Science* **120** (1), pp. 975–982, 2017.

[8] Karatas S., Kuru C. Neutrosophic Topology. *Neutrosophic Sets and Systems* **13**, pp. 90-95, 2016.

[9] Latreche A., Barkat O., Milles S., Ismail F. Single valued neutrosophic mappings defined by single valued neutrosophic relations with applications. *Neutrosophic Sets and Systems* **32**, pp. 203-220, 2020.

[10] Mehmood A., Nadeem F., Nordo G., Zamir M., Park C.,Kalsoom H., Jabeen S., Khan M.Y. Generalized neutrosophic separation axioms in neutrosophic soft topological spaces. *Neutrosophic Sets and Systems* **32** (1), pp. 38–51, 2020.

[11] Mehmood A., Nadeem F., Park C., Nordo G., Kalsoom H., Rahim Khan M., Abbas N. Neutrosophic soft alpha-open set in neutrosophic soft topological spaces. *Journal of Algorithms and Computation* **52**, pp. 37–66, 2020.

[12] Mondal K, Pramanik S. Neutrosophic Decision Making Model of School Choice. *Neutrosophic Sets and Systems* **7** (1), pp. 62-68, 2015.

[13] Nordo G., Jafari S., Mehmood A., Basumatary B. A Python Framework for Neutrosophic Sets and Mappings. *Neutrosophic Sets and Systems* **65**, pp. 199-236, 2024.

[14] Ozturk T.Y., Some Structures on Neutrosophic Topological Spaces. *Applied Mathematics and Nonlinear Sciences*, pp. 1–12, 2020.

[15] Mershia Rabuni J.J., Balaman N., Computation of Neutrosophic Soft Topology using Python. *Neutrosophic Sets and Systems* **60**, pp. 548-569, 2023.

[16] Ray G.C., Dey S. Neutrosophic point and its neighbourhood structure *Neutrosophic Sets and Systems* **43**, pp. 1-13, 2021.

[17] Saber Y., Alsharari F., Smarandache F. On Single-Valued Neutrosophic Ideals in Şostak Sense. *Symmetry* **12** (2):193, 2020.

[18] Salama A.A., Alblowi S., Neutrosophic Set and Neutrosophic Topological Spaces. *International Journal of Mathematical Archive*, 2012.

[19] Salama A.A., Alagamy H. Neutrosophic Filters. *International Journal of Computer Science Engineering and Information Technologt Research* **3** (1), pp. 307-312, 2013.

[20] Salama A., El-Ghareeb H.A.. Manie A.M., Smarandache F. Introduction to develop some software programs for dealing with neutrosophic sets. *Neutrosophic Sets and Systems* **3**, pp. 51-52, 2019.

[21] Salama A., Abd el-Fattah M., El-Ghareeb H.A.. Manie A.M. Design and Implementation of Neutrosophic Data Operations Using Object Oriented Programming. *International Journal of Computer Application* **4** (5), pp. 163-175, 2014.

[22] Salama A.A., Smarandache F., Kromov V. Neutrosophic Closed Set and Neutrosophic Continuous Functions. *Neutrosophic Sets and Systems* **4**, pp. 4-8, 2014.

[23] Saranya S., Vigneshwaran M., Jafari S. C# Application to Deal with Neutrosophic $g\alpha$-Closed Sets In Neutrosophic Topology. *Application and Applied Mathematics* **15** (1), pp. 226-239, 2020.

[24] Sleem A., Abdel-Baset M., El-henawy I. PyIVNS: A python based tool for Interval-valued neutrosophic operations and normalization. *SoftwareX* **12**, pp. 1-7, 2020.

[25] Smarandache F. A Unifying Field in Logics. Neutrosophy: Neutrosophic Probability, Set and Logic. Rehoboth: American Research Press, 1999.

[26] Smarandache F. Introduction to Neutrosophic Statistics. USA: Sitech & Education Publishing, 2014.

[27] Topal S., Broumi S., Bakali A., Talea M., Smarandache F. A Python Tool for Implementations on Bipolar Neutrosophic Matrices. *Neutrosophic Sets and Systems* **28**, pp. 138-161, 2019.

[28] Wang H., Smarandache F., Zhang Y.Q., Sunderraman R. Single Valued Neutrosophic Sets. *Technical Sciences and Applied Mathematics*, pp. 10-14, 2012.

[29] Zadeh L.A. Fuzzy Sets, *Information and Control* **8** (3), pp. 338-353, 1965.

[30] Zhang M., Zhang L. Cheng H.D. A neutrosophic approach to image segmentation based on watershed method. *Signal Processing* **90**, pp. 1510-1517, 2010.