NeutrosophyLib

Another ‘Lego’ piece that has the potential to help us to improve DAO Governance and/or develop more empathic AI / Blockchain-based initiatives.

Version 1

Dr. Ranulfo Paiva Sobrinho

Cofounder Satisfied Vagabonds

Cofounder Cambiatus.com and Cofiblocks.com

Ph.D. Economics

July 17, 2023

Welcome to NeutrosophyLib! Pura Vida!!

NeutrosophyLib is a Solidity library (work in progress) developed by Satisfied Vagabonds, from Costa Rica's jungle to the world.

Is Neutrosophy new for you? Don't worry! Read this first, Neutrosophy as a tool to navigate a hybrid convex-concave world! Or, read this brief introduction.

Through NeutrosophyLib we aim to introduce Neutrosophy to the Web3 community because we think it can help us to navigate in a world, accordingly Vitalik Buterin:

"…the world is not entirely convex, but it is not entirely concave either. But the existence of some concave path between any two distant positions A and B is very likely, and if you can find that path then you can often find a synthesis between the two positions that is better than both…"

(https://vitalik.ca/general/2020/11/08/concave.html)

Neutrosophy is a modern and powerful - philosophy, and mathematics - created in the 90s by Prof. Florentin Smarandache[1]. Neutrosophy helps us see the world beyond the lens of polarizing logic. By polarizing logic, we mean:

" ... a logic induces people to categorize something between belonging, or not belonging to a set. Also, it does not allow a person, or group, to express the level of indeterminacy, or, the level of "I don't know if that thing belongs or doesn't belong…" Neutrosophy as a tool to navigate a hybrid convex-concave world

Neutrosophy allows us to express, simultaneously, our opinion regarding something considering:

- our degree of agreement, and,

- our degree of indecision (indeterminacy), and,

- our degree of disagreement.

For example, in a voting process influenced by polarizing logic, we vote FOR or AGAINST something, and if we do not want to express our opinion we ABSTAIN (Figure 1A).

Sometimes we face situations where we agree and disagree with a proposal partially, and, at the same time, we have a doubt regarding it. For example, given a proposal, let us suppose we agree 40%, have doubt (90%) and disagree (10%). In this case, a voting process influenced by Neutrosophy (Figure 1B) allows us to explicit our thoughts clearer and more transparent.

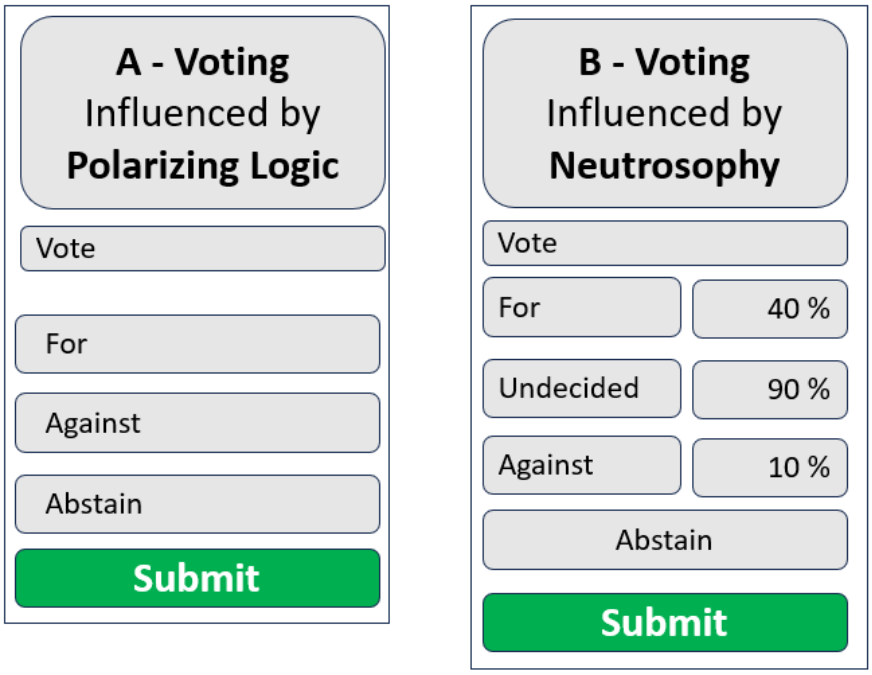

Figure 1 - Voting systems influenced by (A) polarizing logic, and, (B) Neutrosophy

Neutrosophy has three components, in Figure 1B, represented as (For, Undecided, Against) while polarizing logic has two, in Figure 1A, represented as (For, Against). In both cases, the Abstain is present, allowing those who do not want to express their opinion.

Here, we give details related to the neutrosophic operations and how to test them. You can find the code in the following repository: https://github.com/SatisfiedVagabonds/NeutrosophyLib.

These basic operations are like 'Lego' pieces where you connect them and build decentralized applications. We, at Satisfied Vagabonds, are prototyping some collaborative businesses and intend to share them with the Web3 community as soon as possible. Be tuned!

NeutrosophyLib Operations

In the first version of NeutrosophyLib, we handle a set of operations related to SVNS (Single Valued Neutrosophic Set) and SVNN (Single Valued Neutrosophic Number)[2].

We can represent two single-valued neutrosophic numbers, x, and y, as:

$$x = (T_x, I_x, F_x), \text{ where } T_x, I_x, F_x$$

are the degrees of Trueness, Indeterminacy, and Falseness, respectively, of 'x' SVNN.

$$y = (T_y, I_y, F_y), \text{ where } T_y, I_y, F_y$$

are the degrees of Trueness, Indeterminacy, and Falseness, respectively, of 'y' SVNN.

Also, (T_y, I_y, F_y), in mathematical jargon are known, respectively, as truth-membership, indeterminacy-membership, and falsity-membership.

A SVNN is represented as a struct in NeutrosophyLib. See Code 0.

Code 0. A struct to represent an SVNN.

```
library NeutrosophyLib {
    // An struct to represent a single-valued neutrosophic number (SVNN)
    struct SVNN {
        uint256 T; // truth-membership
        uint256 I; // indeterminacy-membership
        uint256 F; // falsity-membership
      }
    }
```

We use the following data from Table 1 to exemplify the Neutrosophy operations.

Table 1 – A single-valued neutrosophic set (**voter**) with its three SVNNs.

Voter	**T** = For	**I** = Undecided	**F** = Against
Helia	0.90	0.20	0.00
Marcio	0.50	0.70	0.10
Joana	0.30	0.90	0.10

For more details, read Neutrosophy as a tool to navigate a hybrid convex-concave world[3]

1. IsContained operation

Given two SVNNs, x, and y. To check if x is contained in y, the following properties must be attended [4]:

$$T_x =< T_y,$$
$$I_x >= I_y,$$
$$F_x >= F_y$$

In other words, the algorithm can be represented by:

If ($T_x =< T_y$) AND ($I_x >= I_y$) AND ($F_x >= F_y$) then TRUE else FALSE.

Example:

Consider Table 1, being Helia (x) and Joana (y) two SVNN, respectively, belonging to an SVNS (voters), represented as:

SVNS(voters) = { Helia (0.9, 0.2, 0.0), Marcio (0.5, 0.7, 0.1), Joana (0.3, 0.9, 0.1) }

Check if SVNN(x) is contained in SVNN(y). See code 1.

Code 1 – Testing the function IsContained.

```
contract TestNeutrosophyLib is Test {
    // Define a function to test if a SVNN is contained in another one
    function testIsContained() public {
        NeutrosophyLib.SVNN memory x = NeutrosophyLib.SVNN(
            0.9 * 1e4,
            0.2 * 1e4,
            0.0 * 1e4
        ); // Helia (0.9, 0.2, 0.0)
        NeutrosophyLib.SVNN memory y = NeutrosophyLib.SVNN(
            0.3 * 1e4,
            0.9 * 1e4,
            0.1 * 1e4
        ); // Joana (0.3, 0.9, 0.1)

        // Check SVNN
        NeutrosophyLib.checkSVNN(x);
         NeutrosophyLib.checkSVNN(y);

        // Call the isContained function and store the result
        bool result = NeutrosophyLib.isContained(x, y);

        // Assert that the result is correct
```

```
        assertEq(result, false); // result must be False

In the terminal write:
      forge test --match-test testIsContained -vvv
```

1.1. EQUAL operation

Given two SVNNs, x, and y. To check if x is equal to y, the following condition must happen [4]:

x is contained in y, and, y is contained in x.

Example:

Consider Table 1, being Helia (x) and Marcio (w) two SVNN, respectively, belonging to an SVNS (voters), represented as:

SVNS(voters) = { Helia (0.9, 0.2, 0.0), Marcio (0.5, 0.7, 0.1), Joana (0.3, 0.9, 0.1) }

Check if 'x' and 'w' are equal. The result is False.

Code 1.1 – Testing the function isEqual.

```
function testIsEqual() public {
        NeutrosophyLib.SVNN memory x = NeutrosophyLib.SVNN(
            0.9 * 1e4,
            0.2 * 1e4,
            0.0 * 1e4
        ); // Helia (0.9, 0.2, 0.0)
        NeutrosophyLib.SVNN memory w = NeutrosophyLib.SVNN(
            0.5 * 1e4,
            0.7 * 1e4,
            0.1 * 1e4
        ); // Marcio (0.5, 0.7, 0.1)
        // Check SVNN
        NeutrosophyLib.checkSVNN(x);
        NeutrosophyLib.checkSVNN(w);

        // Call the union function and store the result
        bool result = NeutrosophyLib.isEqual(x, w);

        // Assert that the result is correct
        assertEq(result, false); // The result must be False
    }
In the terminal write:
```

```
forge test --match-test testIsEqual -vvv
```

2. UNION operation

The Union of two SVNNs (x, y) is also an SVNN (z) whose elements are calculated [4]:

$$T_z = \max(T_x, T_y)$$
$$I_z = \max(I_x, I_y)$$
$$F_z = \min(F_x, F_y)$$

$$x \cup z = z,$$
$$z = (T_z, I_z, F_z)$$

Example:

Consider Table 1, being Helia (x) and Marcio (w) two SVNN, respectively, belonging to an SVNS (voters), represented as:

SVNS(voters) = { Helia (0.9, 0.2, 0.0), Marcio (0.5, 0.7, 0.1), Joana (0.3, 0.9, 0.1) }

Compute the Union of 'x' and 'w'. The result is: (0.9, 0.7, 0.0)

Code 2 – Testing the function Union.

```
function testUnion() public {
        NeutrosophyLib.SVNN memory x = NeutrosophyLib.SVNN(
            0.9 * 1e4,
            0.2 * 1e4,
            0.0 * 1e4
        ); // Helia (0.9, 0.2, 0.0)
        NeutrosophyLib.SVNN memory w = NeutrosophyLib.SVNN(
            0.5 * 1e4,
            0.7 * 1e4,
            0.1 * 1e4
        ); // Marcio (0.5, 0.7, 0.1)

        // Check SVNN
        NeutrosophyLib.checkSVNN(x);
        NeutrosophyLib.checkSVNN(w);

        // Call the union function and store the result
        NeutrosophyLib.SVNN memory result = NeutrosophyLib.union(x, w);

        // Assert that the result is correct
        assertEq(result.T, 0.9 * 1e4); // T should be max(0.9, 0.5) = 0.9
        assertEq(result.I, 0.7 * 1e4); // I should be max(0.2, 0.7) = 0.7
```

```
        assertEq(result.F, 0.0 * 1e4); // F should be min(0.0, 0.1) = 0.0
    }

In the terminal write:
      forge test --match-test testIsContained -vvv
```

3. INTERSECTION operation

The Intersection of two SVNNs (x, y) is also an SVNN (z) whose elements are calculated [4]:

$$T_z = \min (T_x, T_y)$$
$$I_z = \min (I_x, I_y)$$
$$F_z = \max (F_x, F_y)$$

$$x \cap y = z,$$
$$z = (T_z, I_z, F_z)$$

Example:

Consider Table 1, being Helia (x) and Marcio (w) two SVNN, respectively, belonging to an SVNS (voters), represented as:

SVNS(voters) = { Helia (0.9, 0.2, 0.0), Marcio (0.5, 0.7, 0.1), Joana (0.3, 0.9, 0.1) }

Compute the Intersection of 'x' and 'w'. The result is: (0.5, 0.2, 0.1)

Code 3 – Testing the function Intersection

```
function testIntersection() public {
        // Create two SVNNs to test with
        NeutrosophyLib.SVNN memory x = NeutrosophyLib.SVNN(
            0.9 * 1e4,
            0.2 * 1e4,
            0.0 * 1e4
        ); // Helia (0.9, 0.2, 0.0)
        NeutrosophyLib.SVNN memory w = NeutrosophyLib.SVNN(
            0.5 * 1e4,
            0.7 * 1e4,
            0.1 * 1e4
        ); // Marcio (0.5, 0.7, 0.1)

        // Check SVNN
        NeutrosophyLib.checkSVNN(x);
        NeutrosophyLib.checkSVNN(w);

        // Call the union function and store the result
```

```
        NeutrosophyLib.SVNN memory result = NeutrosophyLib.intersection(x,
w);

        // Assert that the result is correct
        assertEq(result.T, 0.5 * 1e4); // T should be max(0.9, 0.5) = 0.5
        assertEq(result.I, 0.2 * 1e4); // I should be max(0.2, 0.7) = 0.2
        assertEq(result.F, 0.1 * 1e4); // F should be min(0.0, 0.1) = 0.1
    }

In the terminal write:
    forge test --match-test testIntersection -vvvv
```

4. DIFFERENCE operation

The Difference between two SVNNs (x, y) is also an SVNN (z) whose elements are [4]:

$$T_z = \min (T_x, F_y)$$
$$I_z = \min (I_x, 1 - I_y)$$
$$F_z = \max (F_x, T_y)$$

$$x - y = z,$$
$$z = (T_z, I_z, F_z)$$

Example:

Consider Table 1, being Helia (x) and Marcio (w) two SVNN, respectively, belonging to an SVNS (voters), represented as:

SVNS(voters) = { Helia (0.9, 0.2, 0.0), Marcio (0.5, 0.7, 0.1), Joana (0.3, 0.9, 0.1) }

Compute the Difference between 'x' and 'w'. The result is: (0.1, 0.2, 0.5)

Code 4 – Testing the function Difference

```
function testDifference() public {
        // Create two SVNNs to test with
        NeutrosophyLib.SVNN memory x = NeutrosophyLib.SVNN(
            0.9 * 1e4,
            0.2 * 1e4,
            0.0 * 1e4
        ); // Helia (0.9, 0.2, 0.0)
        NeutrosophyLib.SVNN memory w = NeutrosophyLib.SVNN(
            0.5 * 1e4,
            0.7 * 1e4,
            0.1 * 1e4
        ); // Marcio (0.5, 0.7, 0.1)

        // Check SVNN
```

```
        NeutrosophyLib.checkSVNN(x);
        NeutrosophyLib.checkSVNN(w);

        // Call the union function and store the result
        NeutrosophyLib.SVNN memory result = NeutrosophyLib.difference(x, w);

        // Assert that the result is correct
        assertEq(result.T, 0.1 * 1e4); // T = min(x.T, w.F) = min(0.9, 0.1)
= 0.1
        assertEq(result.I, 0.2 * 1e4); // I = min(x.I, 1-b.I) = min(0.2,
0.9) = 0.2
        assertEq(result.F, 0.5 * 1e4); // F = max(x.F, w.T)= max(0.0, 0.5) =
0.5
    }

In the terminal write:
      forge test --match-test testDifference -vvv
```

5. COMPLEMENT operation

Given a SVNN, x, its complement, ‘ x^c ’, is another SVNN, z, which values are given by [4]:

$$x^c = (F_x , 1 - I_x , T_x)$$

Example:

Consider Table 1, being Helia (x), an SVNN, represented as Helia (0.9, 0.2, 0.0), its complement, ‘ x^c ’, is equal to (0,0, 0.8, 0.9).

Code 5 -

```
// Define a function to test the complement function of the NeutrosophyLib
    function testComplement() public {
        NeutrosophyLib.SVNN memory x = NeutrosophyLib.SVNN(
            0.9 * 1e4,
            0.2 * 1e4,
            0.0 * 1e4
        ); // Helia (0.9, 0.2, 0.0)

        // Check SVNN
        NeutrosophyLib.checkSVNN(x);

        // Call the Complement nion function and store the result
        NeutrosophyLib.SVNN memory result = NeutrosophyLib.complement(x);
```

```
        // Assert that the result is correct
        assertEq(result.T, 0.0 * 1e4); // T should be 0
        assertEq(result.I, 0.8 * 1e4); // I should be 0.80
        assertEq(result.F, 0.9 * 1e4); // F should be 0.9
    }

In the terminal write:
      forge test --match-test testComplement -vvvv
```

6. ADD_SVNN operation

The addition of two SVNNs (x, y) results in another SVNN z, z = x + y, whose elements are calculated [4]:

$$T_z = (T_x + T_y - T_x * T_y),$$
$$I_z = I_x * I_y,$$
$$F_z = F_x * F_y$$

$$x + y = z,$$
$$z = (T_z, I_z, F_z)$$

Example:

Consider Table 1, being Helia (x) and Marcio (w) two SVNN, respectively, belonging to an SVNS (voters), represented as:

SVNS(voters) = { Helia (0.9, 0.2, 0.0), Marcio (0.5, 0.7, 0.1), Joana (0.3, 0.9, 0.1) }

The result of Adding 'x' and 'w' is: ((1.4 – 0.45), (0.2*0.7), (0.0 * 0.5)) = (0.95, 0.14, 0.0)

Code 6 – Testing the function AddSVNN

```
function testAddSVNN() public {
        // Create two SVNNs to test with
        NeutrosophyLib.SVNN memory x = NeutrosophyLib.SVNN(
            0.9 * 1e4,
            0.2 * 1e4,
            0.0 * 1e4
        ); // Helia (0.9, 0.2, 0.0)
        NeutrosophyLib.SVNN memory w = NeutrosophyLib.SVNN(
            0.5 * 1e4,
            0.7 * 1e4,
            0.1 * 1e4
        ); // Marcio (0.5, 0.7, 0.1)

        // Check SVNN
```

```
        NeutrosophyLib.checkSVNN(x);
        NeutrosophyLib.checkSVNN(w);

        // Call the union function and store the result
        NeutrosophyLib.SVNN memory result = NeutrosophyLib.addSVNN(x, w);

        // Assert that the result is correct
        assertEq(result.T, 0.95 * 1e4); // T = (Tx + Ty - Tx * Ty) =((1.4 -
0.45), (0.2*0.7), (0.0 * 0.5)) = 0.955
        assertEq(result.I, 0.14 * 1e4); // I = Ix * Iy = 0.14
        assertEq(result.F, 0.0 * 1e4); // F = Fx * Fy = 0.0
    }
In the terminal write:
      forge test --match-test testAddSVNN -vvv
```

7. MULTIPLY_SVNN operation

The multiplication of two SVNNs (x, y) results in another SVNN z, z = x * y, whose elements are calculated [4]:

$$\mathbf{T_z} = T_x * T_y,$$
$$\mathbf{I_z} = I_x + I_y - I_x * I_y,$$
$$\mathbf{F_z} = F_x + F_y - F_x * F_y$$

$$x * y = z,$$
$$z = (\mathbf{T_z}, \mathbf{I_z}, \mathbf{F_z})$$

Example:

Consider Table 1, being Helia (x) and Marcio (w) two SVNN, respectively, belonging to an SVNS (voters), represented as:

SVNS(voters) = { Helia (0.9, 0.2, 0.0), Marcio (0.5, 0.7, 0.1), Joana (0.3, 0.9, 0.1) }

The result of Multiplying 'x' and 'w' is:

((0.9 * 0.5) , (0.2+0.7 – 0.2*0.7), (0.0+0.1 – 0.0*0.1)) = (0.45, 0.76, 0.1)

Code 7

```
function testMultiplySVNN() public {
        // Create two SVNNs to test with
        NeutrosophyLib.SVNN memory x = NeutrosophyLib.SVNN(
            0.9 * 1e4,
            0.2 * 1e4,
            0.0 * 1e4
        ); // Helia (0.9, 0.2, 0.0)
        NeutrosophyLib.SVNN memory w = NeutrosophyLib.SVNN(
```

```
            0.5 * 1e4,
            0.7 * 1e4,
            0.1 * 1e4
        ); // Marcio (0.5, 0.7, 0.1)

        // Check SVNN
        NeutrosophyLib.checkSVNN(x);
        NeutrosophyLib.checkSVNN(w);

        // Call the union function and store the result
        NeutrosophyLib.SVNN memory result = NeutrosophyLib.multiplySVNN(x,
w);

        // Assert that the result is correct
        assertEq(result.T, 0.45 * 1e4); // T = (Tx * Ty) =(0.9 * 0.5) = 0.45
        assertEq(result.I, 0.76 * 1e4); // I = Ix+Iy - Ix*Iy = 0.2+0.7 –
0.2*0.7 = 0.76
        assertEq(result.F, 0.10 * 1e4); // F = Fx+Fy - Fx * Fy = 0.0+0.1 –
0.0*0.1 = 0.1
    }

In the terminal write:
      forge test --match-test testMultiplySVNN -vvv
```

8. SCORE operation

The score of a SVNN, y, is a real number ∈ [0, 1]. The equation to score an SVNN is given by [4]:

$$Score(y) = (2 + T_y – I_y – F_y) / 3,$$

where Score ∈ [0, 1]

Example:

Consider Table 1, being Helia (x) and Marcio (w) two SVNN, respectively, belonging to an SVNS (voters), represented as:

SVNS(voters) = { Helia (0.9, 0.2, 0.0), Marcio (0.5, 0.7, 0.1), Joana (0.3, 0.9, 0.1) }

The Score of the SVNN, x, is equal to ((2 + 0.9 – 0.2 – 0.0) / 3) = 0.9

Code 8 – Testing the Score function.

```
function testScoreSVNN() public {
        // Create two SVNNs to test with
        NeutrosophyLib.SVNN memory x = NeutrosophyLib.SVNN(
```

```
            0.9 * 1e4,
            0.2 * 1e4,
            0.0 * 1e4
        ); // Helia (0.9, 0.2, 0.0)

        // Call the union function and store the result
        uint256 result = NeutrosophyLib.ScoreSVNN(x);

        // Check SVNN
        NeutrosophyLib.checkSVNN(x);

        // Assert that the result is correct
        // Scr(a) = (2 + a.T - a.I - a.F) / 3
        assertEq(result, 0.9 * 1e4); // result = 0.9
    }

In the terminal write:
      forge test --match-test testScoreSVNN -vvv
```

9. ACCURACY operation

The accuracy of a SVNN, z, is a real number ∈ [-1, 1], given by [4]:

$$A(z) = T_z - I_z,$$

where $A(z) \in [\ -1, 1\ ]$

Example:

Consider Table 1, being Helia (x) and Marcio (w) two SVNN, respectively, belonging to an SVNS (voters), represented as:

SVNS(voters) = { Helia (0.9, 0.2, 0.0), Marcio (0.5, 0.7, 0.1), Joana (0.3, 0.9, 0.1) }

The Accuracy of the SVNN, x, is equal to (0.9 – 0.2) = 0.7

Code 9 – Testing the Accuracy function.

```
function testAccuracySVNN() public {
        // Create an SVNN to test with
        NeutrosophyLib.SVNN memory x = NeutrosophyLib.SVNN(
            0.9 * 1e4,
            0.2 * 1e4,
            0.0 * 1e4
        ); // Helia (0.9, 0.2, 0.0)

        // Call the union function and store the result
```

```
        uint256 result = NeutrosophyLib.AccuracySVNN(x);

        // Check SVNN
        NeutrosophyLib.checkSVNN(x);

        // Assert that the result is correct
        // Accur(x) = ( x.T - x.I )
        assertEq(result, 0.7 * 1e4); // Accur = x.T - x.I = 0.9 - 0.2 = 0.7
    }
In the terminal write:
      forge test --match-test testAccuracySVNN -vvv
```

10. DENEUTROSOPHY operation

Given an SVNN, x, you can transform it in a real number ∈ [0, 1], through the Deneutrosophy operation. The result, dNeut, is given by[4]:

$$dNeut = 1 - sqrt(((1 - T_x)^2 + (I_x)^2 + (F_x)^2) / 3)$$

Example:

Consider Table 1, being Helia (x) and Marcio (w) two SVNN, respectively, belonging to an SVNS (voters), represented as:

SVNS(voters) = { Helia (0.9, 0.2, 0.0), Marcio (0.5, 0.7, 0.1), Joana (0.3, 0.9, 0.1) }

The result of Deneutrosophy an SVNN, x, is equal to:

$$dNeut(x) = 1 - sqrt(((1 - 0.9)^2 + (0.2)^2 + (0.0)^2) / 3) = 0.8712$$

Code 10 – Testing the function Deneutrosophy.

```
function testDeneutrosophySVNN() public {
        // Create an SVNN to test with
        NeutrosophyLib.SVNN memory x = NeutrosophyLib.SVNN(
            0.9 * 1e4,
            0.2 * 1e4,
            0.0 * 1e4
        ); // Helia (0.9, 0.2, 0.0)

        // Check SVNN
        NeutrosophyLib.checkSVNN(x);

        // Call the union function and store the result
        uint256 result = NeutrosophyLib.DeneutrosophySVNN(x);

        // Assert that the result is correct
```

```
        // dNeutr = (1e4 - sqrt((1*e4 - a.T)^2 + a.I^2 + a.F^2)/(3*1e4)
        assertEq(result, 0.8712 * 1e4);
    }

In the terminal write:
      forge test --match-test testDeneutrosophySVNN -vvv
```

Coming soon

As you noted, we introduced the ‘Lego’ pieces (neutrosophic operations). We intend to share as soon as possible, real case applications using these and other operations.

References

[1] Smarandache, F. 1999. A unifying field in Logics: Neutrosophic Logic. In Philosophy. American Research Press, 1-141

[2] Wang, H.; Smarandache, F.; Zhang, Y.; and Sunderraman, R. 2010. Single valued SVNS sets, Multisp Multistruct 4: 410–413.

[3] https://medium.com/@ranulfo17/neutrosophy-as-a-tool-to-navigate-a-hybrid-convex-concave-world-50a3f1c9b3ff

[4] Lu, Z.; YE, J. 2017. Single-valued neutrosophic hybrid arithmetic and geometric aggregation operators and their decision-making method. Information 8, 3.