NSS

Neutrosophic Sets and Systems, Vol. 76, 2025

University of New Mexico

A Python Class for Neutrosophic Morphology

Lorenzo Affè[1,*], Giorgio Nordo[2] and Florentin Smarandache[3]

[1]MIFT Department – Mathematical and Computer Science, Physical Sciences and Earth Sciences University of Messina, 98166 Sant'Agata, Messina, Italy; lorenzo.affe1@gmail.com

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

[3]Mathematics, Physics, and Natural Science Division University of New Mexico 705 Gurley Ave., Gallup, NM 87301, USA; smarand@unm.edu

Abstract. In this work, we introduce a Python class, named `NSmorph`, developed to facilitate image manipulation through neutrosophic morphological operations. This innovative approach extends traditional image processing methods by leveraging the flexibility of neutrosophic logic to handle uncertainty, indeterminacy, and noise in digital images. The class offers implementations of essential morphological operators, such as neutrosophic dilation, erosion, opening, and closing, providing a robust tool for applications where image clarity is often compromised, like medical imaging and surveillance. We detail the class structure and functions and provide multiple examples to demonstrate its practical applications and comparative advantages over classical morphological methods.

Keywords: neutrosophic set; neutrosophic morphology; morphological image processing; Python programming; uncertainty in image analysis.

1. Introduction

In contemporary society, marked by increasing complexity and interconnected systems, classical mathematical theories face significant limitations when applied to real-world scenarios where vagueness, uncertainty, and indeterminacy are predominant. Such challenges are particularly pronounced in the field of image analysis, where accurate data interpretation plays a crucial role in diverse applications ranging from medical diagnostics to security surveillance. Traditional image processing techniques often struggle to handle these uncertainties effectively, necessitating alternative mathematical approaches capable of modeling and managing ambiguity [19].

Mathematical Morphology, originally developed by Matheron and Serra [13, 14], provides a powerful set of tools in nonlinear image analysis, primarily focusing on the geometric structure of images. Classical morphological operators, extensively discussed in Dougherty's works [2,3], have proven effective for tasks like noise filtering, shape extraction, and image segmentation. However, these operators are built on deterministic models of data, which can be limiting in applications where noise or uncertainty inherently affects image quality. Researchers have thus explored various generalizations, including Zadeh's fuzzy sets [20] and Atanassov's intuitionistic fuzzy sets [1], to accommodate situations where precise binary interpretations are insufficient. These theories laid the groundwork for the emergence of neutrosophic sets, introduced by Smarandache [17], which extend the representation of uncertainty by considering degrees of truth, indeterminacy, and falsehood independently, providing a nuanced framework suited to complex, real-world data.

Neutrosophic set theory has been applied successfully in various domains, including graph theory [4], decision-making [6], and soft topological spaces [7], where traditional mathematical approaches fall short. In image analysis, Neutrosophic Morphology applies Smarandache's principles to extend classical morphological operations, providing a framework that captures and preserves information about uncertainty and noise within images. This has proven especially beneficial in applications where data reliability varies significantly, allowing each pixel in an image to possess distinct degrees of membership, non-membership, and indeterminacy [18]. Studies have demonstrated that neutrosophic operators maintain structural transformations without losing important information about the underlying uncertainty, making them particularly effective for medical image analysis [12] and other fields requiring enhanced data fidelity [8, 16].

The importance of computational applications and frameworks developed in Python has significantly grown within neutrosophic studies. These frameworks facilitate practical implementations of neutrosophic concepts, making them more accessible to researchers and practitioners. Recent Python-based tools, such as those by El-Ghareeb [5] and Sleem et al. [16], have laid foundational work for performing neutrosophic operations efficiently. Nordo et al. [9] further extended this effort by developing a comprehensive Python framework specifically tailored for neutrosophic sets and mappings, highlighting the demand for adaptable and open-source tools in this field. Such frameworks not only simplify the computational aspects but also expand the applicability of neutrosophic methods across various disciplines, including image processing, where tools that address uncertainty and indeterminacy are crucial.

Despite these advancements, there remains a need for a comprehensive and user-friendly Python-based solution that can perform neutrosophic morphological operations in image analysis with flexibility and precision. Existing frameworks either focus narrowly on numerical

neutrosophic operations or lack the modular design needed for broader applicability [5, 16]. Recognizing this gap, we propose `NSmorph`, a Python class designed to implement fundamental neutrosophic morphological operators, including neutrosophic dilation, erosion, opening, and closing. These operators extend traditional morphological techniques into the neutrosophic domain, providing a versatile toolkit for image manipulation in environments with prevalent uncertainty, noise, and indeterminacy [11].

The structure of the paper is organized as follows: in Section 2, we delve into the data structure and the specific methods that constitute the `NSmorph` class, detailing each component's functionality. Section 3 presents practical examples that demonstrate the efficacy of the class in various scenarios, along with observations on the performance of neutrosophic operators relative to their classical counterparts. Finally, in Section 4, we provide our conclusions, underscoring the advantages of neutrosophic morphology in image analysis and the potential for future developments in this field [15].

2. The `NSmorph` class

The `NSmorph` class is designed for manipulating neutrosophic images using common morphological operators, including NS-dilation, NS-erosion, NS-opening, and NS-closing. The class constructor can initialize a neutrosophic image from various input types, such as a file path to a PNG or JPEG image, a NumPy array, or an instance of the `NSmorph` class itself. The following code shows how the constructor is implemented:

```
import os.path
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt

class NSmorph:
   def __init__(self, image, radius=0):
      if radius < 0:
      raise ValueError(f"The radius '{radius}' of the neighbourhood cannot be
    negative")
      if type(image) == np.ndarray:
         if image is None:
            raise ValueError("The image is not valid")
         (height, width)=image.shape
         self.__height = height
         self.__width = width
         self.__image_orig = image
         self.__radius = radius
         i_med = np.zeros((height, width), dtype = np.float32)
         for y in range(height):
            for x in range(width):
               md = 0
               n_pixel = 0
               for j in range(y - radius, y + radius + 1):
```

```
24                  for i in range(x - radius, x + radius +1):
25                     if (0<=i<width) and (0<=j<height) :
26                        md += image[j][i]
27                        n_pixel += 1
28               md /= n_pixel
29               i_med[y][x] = md
30         i_med_min = i_med.min()
31         i_med_max = i_med.max()
32         i_med_size = i_med_max - i_med_min
33         delta = np.zeros((height, width), dtype=np.float32)
34         for y in range(height):
35            for x in range(width):
36               delta[y][x] = abs(image[y][x] - i_med[y][x])
37         delta_min = delta.min()
38         delta_max = delta.max()
39         delta_size = delta_max - delta_min
40         ns_image = np.zeros((height, width, 3), dtype = np.float32)
41         for y in range(height):
42            for x in range(width):
43               ns_image[y][x][0] = (i_med[y][x] - i_med_min)/i_med_size if
   i_med_size !=0 else 0
44               ns_image[y][x][1] = (delta[y][x] - delta_min)/delta_size if
   delta_size !=0 else 0
45               ns_image[y][x][2] = 1 - ns_image[y][x][0]
46         self.__ns_image = ns_image
47      elif type(image) == str:
48         tmp_imgns = NSmorph(NSmorph.load(image), radius)
49         self.__ns_image = tmp_imgns.get()
50         self.__image_orig = tmp_imgns.getOrig()
51         self.__height = tmp_imgns.height()
52         self.__width = tmp_imgns.width()
53         self.__radius = radius
54      elif type(image) == NSmorph :
55         self.__ns_image = image.get()
56         self.__image_orig = image.getOrig()
57         self.__height = image.height()
58         self.__width = image.width()
59         self.__radius = image.radius()
60      else:
61         raise ValueError("The first parameter must be a matrix or a string")
```

Lines 1 to 4 import necessary libraries, including OpenCV and NumPy for image handling and `pyplot` for visualization. The constructor is defined in line 7, where the `radius` parameter is checked for validity in lines 8 and 9. The constructor can handle three types of input:

- NumPy array (lines 10–46): If `image` is a NumPy array, the image's height, width, and radius are stored. Then, from lines 18 to 32, the average intensity $\overline{I}$ for each pixel is computed by averaging the values within the kernel centered at each pixel. This forms the first channel of the neutrosophic image. In lines 33 to 39, the `delta` matrix, which measures intensity deviation from the local average (representing the homogeneity function), is computed and stored as the second channel. Finally, lines 40

to 46 construct the neutrosophic image by normalizing the intensity and delta values and calculating the non-membership degree as $1 - \texttt{membership}$.

- File path (string) (lines 47–53): If `image` is a file path, the constructor calls the `load` method to load the image as a grayscale NumPy array. The `NSmorph` instance for this image is then initialized, and the original image dimensions and radius are set.
- Existing `NSmorph` object (lines 54–59): If `image` is an existing `NSmorph` instance, the neutrosophic image and related attributes are directly copied.

Finally, in lines 60 and 61, if the input does not match any valid types, an error is raised.

The `NSmorph` class also includes several methods to retrieve the complete neutrosophic image or its individual components—membership, non-membership, and indeterminacy—or to convert it back to the original binary format. These functions provide flexible access and modification capabilities for each aspect of the neutrosophic image representation.

A particularly useful method is the static method `load(img_path)`, which loads an image from a specified file path `img_path`. This method reads the image in grayscale, thus simplifying conversion to the neutrosophic format. If the provided path is invalid, not found, or inaccessible, the method raises an error, leveraging `os.path.isfile()` to verify the file's existence. This ensures that only valid paths are processed, preventing runtime issues due to incorrect paths.

```
@staticmethod
def load(img_path):
    if img_path is None:
        raise ValueError("The path of the image is not valid")
    if not os.path.isfile(img_path):
        raise FileNotFoundError(f"The file '{img_path}' does not exist or it is not accessible.")
    image = cv.imread(img_path, cv.IMREAD_GRAYSCALE)
    if image is None:
        raise IOError(f"It has not been possible to read the image file '{img_path}'")
    return image
```

The `get()` method gives back the neutrosophic image.

```
def get(self):
    return self.__ns_image
```

Methods are also available for retrieving and assigning specific components of the neutrosophic image:

- Membership (`getM()` and `setM()`): `getM()` returns the membership degree for each pixel, while `setM(x, y, mu)` allows for setting the membership degree at a specific pixel location `(x, y)`.
- Non-membership (`getNM()` and `setNM()`): `getNM()` retrieves the non-membership degree matrix, and `setNM(x, y, omega)` sets the non-membership degree for the specified pixel.
- Indeterminacy (`getI()` and `setI()`): `getI()` returns the indeterminacy degree, while `setI(x, y, sigma)` assigns a new indeterminacy degree to a given pixel.

Each method supports fine-grained control over image processing, allowing users to either view or modify the neutrosophic image's distinct properties individually or as a composite.

```
def getM(self):
    return self.__ns_image[:,:,0]

def getI(self):
    return self.__ns.image[:,:,1]

def getNM(self):
    return self.__ns.image[:,:,2]


def setM(self, x, y, mu):
    self.__ns_image[y][x][0] = mu

def setI(self, x, y, sigma):
    self.__ns_image[y][x][1] = sigma

def setNM(self, x, y, omega):
    self.__ns_image[y][x][2] = omega
```

The `getOrig()` method returns the original grayscale image from which the neutrosophic image was derived. This can be useful for comparison purposes, allowing users to access the unprocessed version of the image alongside its neutrosophic representation. The method is implemented as follows:

```
def getOrig(self):
    return self.__image
```

The `width()` and `height()` methods return the width and height of the image, respectively. These methods are particularly useful when constructing other morphological operations, such as dilation and erosion, as they provide easy access to the dimensions of the image, which is essential for managing boundary conditions and kernel application across the entire image. The implementations are straightforward:

```
def width(self):
    return self.__width

def height(self):
    return self.__height
```

The `radius()` method returns the radius of the neighborhood associated with the uploaded image. This radius defines the size of the local region around each pixel that is used in various neutrosophic morphological operations, such as calculating local intensity averages or applying structuring elements in dilation and erosion. Accessing the radius through this method is particularly useful when the neighborhood size impacts processing or when adapting the morphological operators to different kernel sizes. The implementation is as follows:

```
def radius(self):
    return self.__radius
```

The `getBinary()` method performs thresholding on the current image, producing a binary image based on a threshold value specified by the user. The method returns a binary representation of the image as a NumPy matrix, where rows and columns are indexed from $(0, 0)$ at the top left corner. In this binary matrix, pixel values are set to 0 (black) for intensities below the threshold and 1 (white) for intensities equal to or above the threshold. The method is implemented as follows:

```
def getBinary(self, threshold):
    (ret, bin_image) = cv.threshold(self.__image_orig, threshold, 1, cv.
    THRESH_BINARY)
    return bin_image
```

The `getRepresentation()` method returns a grayscale image that represents an interpolation of the three neutrosophic levels: membership, indeterminacy, and non-membership. This interpolation combines the three levels based on customizable weights provided as optional parameters. By default, these weights prioritize the membership degree, reflecting its primary influence on the pixel intensity in the resulting grayscale image. The following code illustrates this implementation:

```
def getRepresentation(self, weightM=0.85, weightI=0.25, weightNM=-0.1, binary
    =False, limit_value=128):
    img_M = self.getM()
    img_I = self.getI()
    img_NM = self.getNM()
    mat_rap = np.zeros((self.__height, self.__width, 3), dtype = np.float32)
```

```
    for y in range(self.__height):
        for x in range(self.__width):
            mat_rap[y][x][0] = weightM*img_M[y][x] + weightI*img_I[y][x] +
     weightNM*img_NM[y][x]
    img_rap = cv.cvtColor(mat_rap, cv.COLOR_BGR2GRAY)
    if binary == True:
        (ret, img_rap) = cv.threshold(img_rap, limit_value, 1, cv.THRESH_BINARY)
    return img_rap
```

The method signature is as follows:

- `weightM`: Weight assigned to the membership degree. Default is 0.85.
- `weightI`: Weight assigned to the indeterminacy degree. Default is 0.25.
- `weightNM`: Weight assigned to the non-membership degree. Default is -0.1.
- `binary`: Boolean flag to enable binarization of the resulting image based on a threshold.
- `limit_value`: Threshold value used for binarization if `binary` is set to `True`. Default is 128.

The grayscale image is generated by combining the three components, emphasizing areas with higher membership and optionally applying thresholding if `binary` is `True`. If thresholding is active, the method uses `limit_value` to convert the grayscale image to a binary one, mapping values above the threshold to white and those below to black.

This function is particularly useful for visualizing the overall neutrosophic information in a single image, allowing for quick interpretation of complex neutrosophic properties in greyscale or binary format.

2.1. *Neutrosophic dilation*

The `dilation(self, kernel)` method performs the dilation operation on a neutrosophic image, using another neutrosophic image as the structuring element (kernel). The method returns the dilated neutrosophic image and it is implemented as follows:

```
def dilation(self, kernel):
    (height, width) = (self.__height, self.__width)
    (height_k, width_k) = (kernel.height(), kernel.width())

    img_M = self.getM()
    img_I = self.getI()
    img_NM = self.getNM()
    kernel_M = kernel.getM()
    kernel_I = kernel.getI()
    kernel_NM = kernel.getNM()

    mat_generating = np.zeros((height, width, 3), dtype=np.uint8)
    im_empty = cv.cvtColor(mat_generating, cv.COLOR_BGR2GRAY)
```

```
    im_dil = NSmorph(im_empty)

    for y in range(height):
        for x in range(width):

            width_sx = x - width_k//2
            if width_sx<0 :
                width_sx = 0
            width_dx = x + width_k//2 +1
            if width_dx > width - 1 :
                width_dx = width - 1
            height_up = y - height_k//2
            if height_up < 0 :
                height_up = 0
            height_down = y+ height_k//2 + 1
            if height_down > height - 1 :
                height_down = height - 1

            mat_M = img_M[height_up:height_down, width_sx:width_dx]
            mat_I = img_I[height_up:height_down, width_sx:width_dx]
            mat_NM = img_NM[height_up:height_down, width_sx:width_dx]

            (n_rows, n_columns) = mat_M.shape

            width_sx_kernel = width_k//2 - n_columns//2
            if width_sx_kernel > n_columns//2 - 1:
                width_sx_kernel = 0
            width_dx_kernel = width_k//2 + n_columns//2 + 1
            if width_dx_kernel > n_columns//2 + 1:
                width_dx_kernel = n_columns
            height_up_kernel = height_k//2 - n_rows//2
            if height_up_kernel > n_rows//2 - 1:
                height_up_kernel = 0
            height_down_kernel = height_k//2 + n_rows//2 + 1
            if height_down_kernel > n_rows//2 + 1 :
                height_down_kernel = n_rows

            mat_kernelM = kernel_M[height_up_kernel:height_down_kernel,
        width_sx_kernel:width_dx_kernel]
            mat_kernelI = kernel_I[height_up_kernel:height_down_kernel,
        width_sx_kernel:width_dx_kernel]
            mat_kernelNM = kernel_NM[height_up_kernel:height_down_kernel,
        width_sx_kernel:width_dx_kernel]

            minimum_M = np.minimum(mat_M, mat_kernelM)
            minimum_I = np.minimum(mat_I, mat_kernelI)
            maximum_NM = np.maximum(mat_NM, 1 - mat_kernelNM)

            mu = minimum_M.max()
            sigma = minimum_I.max()
            omega = maximum_NM.min()

            im_dil.setM(x, y, mu)
            im_dil.setI(x, y, sigma)
            im_dil.setNM(x, y, omega)
    return im_dil
```

In lines 2 and 3, the dimensions of both the input image and the kernel are stored in separate tuples. Then, in lines 5 to 10, the membership, indeterminacy, and non-membership matrices of the image and kernel are retrieved.

From lines 12 to 14, an empty matrix, `mat_generating`, is created using `np.zeros()` from the NumPy library. This matrix is converted into a grayscale image, `im_empty`, with OpenCV's `cv.cvtColor()` function and serves as the base for creating the dilated neutrosophic image `im_dil`.

Lines 16 and 17 use nested `for` loops to iterate through each pixel of the input image. Within each loop, lines 19 to 30 define the bounds for the image region centered on the current pixel (x, y), ensuring that the extracted region matches the size of the kernel. This is done by adjusting for any out-of-bounds areas along the edges of the image, where the kernel might partially exceed the image boundaries.

In lines 32 to 34, the membership, indeterminacy, and non-membership matrices are extracted for the region defined by these bounds, centered around (x, y). Line 36 then evaluates the actual dimensions of these matrices, accounting for cases where the region is truncated by image boundaries.

Lines 38 to 49 set up further bounds to ensure that the extracted kernel matrices align precisely with the region around (x, y) in the image. This step helps reduce the kernel dimensions to match those of the extracted region when scanning near boundaries.

The actual dilation operation occurs in lines 55 to 61, applying the following neutrosophic morphological formulas:

$$\begin{aligned}
\mu_{A\oplus B}(v) &= \sup_{u\in\mathbb{U}} \min\left\{\mu_A(v+u), \mu_B(u)\right\}, \\
\sigma_{A\oplus B}(v) &= \sup_{u\in\mathbb{U}} \min\left\{\sigma_A(v+u), \sigma_B(u)\right\}, \\
\omega_{A\oplus B}(v) &= \inf_{u\in\mathbb{U}} \max\left\{\omega_A(v+u), 1-\omega_B(u)\right\},
\end{aligned}$$

where $\mu_{A\oplus B}(v)$, $\sigma_{A\oplus B}(v)$, and $\omega_{A\oplus B}(v)$ represent the degrees of membership, indeterminacy, and non-membership, respectively, after dilation.

In lines 63 to 65, the calculated values for membership, indeterminacy, and non-membership are stored in the `im_dil` image at position (x, y). Finally, in line 66, the fully processed dilated neutrosophic image is returned.

2.2. *Neutrosophic erosion*

The `erosion(self, kernel)` method applies the neutrosophic erosion operation to the input image using a specified *kernel*, which is also a neutrosophic image. The method returns the eroded neutrosophic image and is implemented as follows:

```
def erosion(self, kernel):
   (height, width) = (self.__height, self.__width)
   (height_k, width_k) = (kernel.height(), kernel.width())

   img_M = self.getM()
   img_I = self.getI()
   img_NM = self.getNM()
   kernel_M = kernel.getM()
   kernel_I = kernel.getI()
   kernel_NM = kernel.getNM()

   mat_generating = np.zeros((height, width, 3), dtype=np.uint8)
   im_empty = cv.cvtColor(mat_generating, cv.COLOR_BGR2GRAY)
   im_er = NSmorph(im_empty)

   for y in range(height):
      for x in range(width):

         width_sx = x - width_k//2
         if width_sx<0 :
            width_sx = 0
         width_dx = x + width_k//2 +1
         if width_dx > width - 1 :
            width_dx = width - 1
         height_up = y - height_k//2
         if height_up < 0 :
            height_up = 0
         height_down = y+ height_k//2 + 1
         if height_down > height - 1 :
            height_down = height - 1

         mat_M = img_M[height_up:height_down, width_sx:width_dx]
         mat_I = img_I[height_up:height_down, width_sx:width_dx]
         mat_NM = img_NM[height_up:height_down, width_sx:width_dx]

         (n_rows, n_columns) = mat_M.shape

         width_sx_kernel = width_k//2 - n_columns//2
         if width_sx_kernel > n_columns//2 - 1:
            width_sx_kernel = 0
         width_dx_kernel = width_k//2 + n_columns//2 + 1
         if width_dx_kernel > n_columns//2 + 1:
            width_dx_kernel = n_columns
         height_up_kernel = height_k//2 - n_rows//2
         if height_up_kernel > n_rows//2 - 1:
            height_up_kernel = 0
         height_down_kernel = height_k//2 + n_rows//2 + 1
         if height_down_kernel > n_rows//2 + 1 :
            height_down_kernel = n_rows

         mat_kernelM = kernel_M[height_up_kernel:height_down_kernel,
   width_sx_kernel:width_dx_kernel]
         mat_kernelI = kernel_I[height_up_kernel:height_down_kernel,
   width_sx_kernel:width_dx_kernel]
         mat_kernelNM = kernel_NM[height_up_kernel:height_down_kernel,
   width_sx_kernel:width_dx_kernel]
```

```
54
55          maximum_M = np.maximum(mat_M, 1 - mat_kernelM)
56          maximum_I = np.maximum(mat_I, 1 - mat_kernelI)
57          minimum_NM = np.minimum(mat_NM, mat_kernelNM)
58
59          mu = maximum_M.min()
60          sigma = maximum_I.min()
61          omega = minimum_NM.max()
62
63          im_er.setM(x, y, mu)
64          im_er.setI(x, y, sigma)
65          im_er.setNM(x, y, omega)
66      return im_er
```

Similar to the neutrosophic dilation method, the dimensions of both the input image and kernel are stored in tuples (lines 2 and 3), and the membership, indeterminacy, and non-membership matrices of the input image and kernel are assigned to `img_M`, `img_I`, `img_NM` and `kernel_M`, `kernel_I`, `kernel_NM` respectively (lines 5 to 10).

Lines 12 to 14 create an empty matrix, `mat_generating`, which serves as the base for the output eroded image, `im_er`. The matrix is initialized as a grayscale image using `cv.cvtColor()` from the OpenCV library.

Nested `for` loops in lines 16 and 17 iterate over each pixel in the input image. Lines 19 to 30 define the region within the input image centered at each pixel (x, y), ensuring the extracted region matches the kernel size. These boundary checks ensure that when the kernel partially overlaps the image edges, the region adjusts accordingly to prevent out-of-bounds errors.

The membership, indeterminacy, and non-membership matrices for the region around (x, y) are extracted in lines 32 to 34. In line 36, the actual dimensions of these matrices are determined to account for potential edge clipping.

Lines 38 to 49 then adjust the kernel's size to match that of the extracted region, ensuring alignment during the operation.

The erosion operation is applied in lines 55 to 61, using the following formulas for neutrosophic morphological erosion:

$$\mu_{A\ominus B}(v) = \inf_{u\in\mathbb{U}} \max\left\{\mu_A(v+u), 1-\mu_B(u)\right\},$$
$$\sigma_{A\ominus B}(v) = \inf_{u\in\mathbb{U}} \max\left\{\sigma_A(v+u), 1-\sigma_B(u)\right\},$$
$$\omega_{A\ominus B}(v) = \sup_{u\in\mathbb{U}} \min\left\{\omega_A(v+u), \omega_B(u)\right\},$$

where $\mu_{A\ominus B}(v)$, $\sigma_{A\ominus B}(v)$, and $\omega_{A\ominus B}(v)$ represent the degrees of membership, indeterminacy, and non-membership, respectively, for the eroded image.

Finally, in lines 63 to 65, the computed membership, indeterminacy, and non-membership values are saved to `im_er` at (x, y). In line 66, the method returns the fully processed eroded neutrosophic image.

2.3. *Neutrosophic opening and closing*

The `opening(self, kernel)` method performs the neutrosophic opening operation on the image. This operation first applies the erosion operator, followed by the dilation operator, using the same *kernel*, which is also a neutrosophic image, passed as a parameter. The result is a neutrosophic image where small structures are removed, and larger shapes are preserved without altering their boundaries significantly. This method is especially useful for tasks where noise reduction or shape smoothing is required. The implementation is straightforward:

```
def opening(self, kernel):
    return self.erosion(kernel).dilation(kernel)
```

The `closing(self, kernel)` method, in contrast, performs the neutrosophic closing operation. This operation first applies the dilation operator to the image, followed by the erosion operator, using the same *kernel* provided. Closing is typically used to fill small holes and connect adjacent structures within an image without significantly altering their contours, making it useful in image enhancement tasks. The implementation is as follows:

```
def closing(self, kernel):
    return self.dilation(kernel).erosion(kernel)
```

The complete source code for this Python class has been released under the Open Source GNU General Public License version 3.0 (or GPL-3.0) and is freely accessible at the url github.com/lorenzoaffe/nsmorph.

3. **Examples and Observations**

In this section, we demonstrate the `NSmorph` class by using neutrosophic dilation, erosion, opening, and closing operations on an image affected by noise. These examples showcase how each neutrosophic morphological operator interacts with noisy data, specifically observing the effect on the image's brightness, clarity, and feature preservation. Such noise, often introduced as random variations in pixel values, can significantly compromise image quality in real-world applications.

```
from ns_morph import NSmorph
import cv2 as cv
```

```
import numpy as np
from matplotlib import pyplot as plt

path_image = "immagini/j_puntini.jpg"
image = NSmorph.load(path_image)
ns_image = NSmorph(image, 4)

path_kernel = "immagini/kernel_croce.jpg"
kernel = NSmorph.load(path_kernel)
ns_kernel = NSmorph(kernel, 1)

nsdil_image = ns_image.dilation(ns_kernel)

plt.suptitle("Neutrosophic dilation (M=membership, I=indeterminacy, NM=non-
    membership)")

plt.subplot(4,3,1)
plt.imshow(ns_image.getOrig(), cmap='gray')
plt.xticks([])
plt.yticks([])
plt.title("Original")

...

plt.subplot(4,3,11)
plt.imshow(nsdil_image.getRepresentation(), cmap='gray')
plt.xticks([])
plt.yticks([])
plt.title("dilation")

plt.subplot(4,3,12)
plt.imshow(nsdil_image.getRepresentation(binary=True, limit_value=0.04), cmap=
    'gray')
plt.xticks([])
plt.yticks([])
plt.title("binarized dilation")

plt.show()
```

In lines 1 to 4, we import necessary modules and libraries, including `NSmorph`, OpenCV-Python, NumPy, and `pyplot` from Matplotlib. Lines 6 to 8 generate the neutrosophic image from the specified path using the static method `load(img_path)`. Similarly, lines 10 to 12 initialize the kernel. Line 14 performs neutrosophic dilation on the image with the kernel, storing the result in `nsdil_image`. From lines 18 to 36, the images are plotted, showing the original, the dilated, and the binarized dilated images.

Figures 1, 2, 3, and 4 present the results of applying NS-dilation, NS-erosion, NS-opening, and NS-closing respectively. Each figure includes grayscale images of the membership, indeterminacy, and non-membership components, as well as the interpolated neutrosophic image and a binarized version obtained through thresholding.

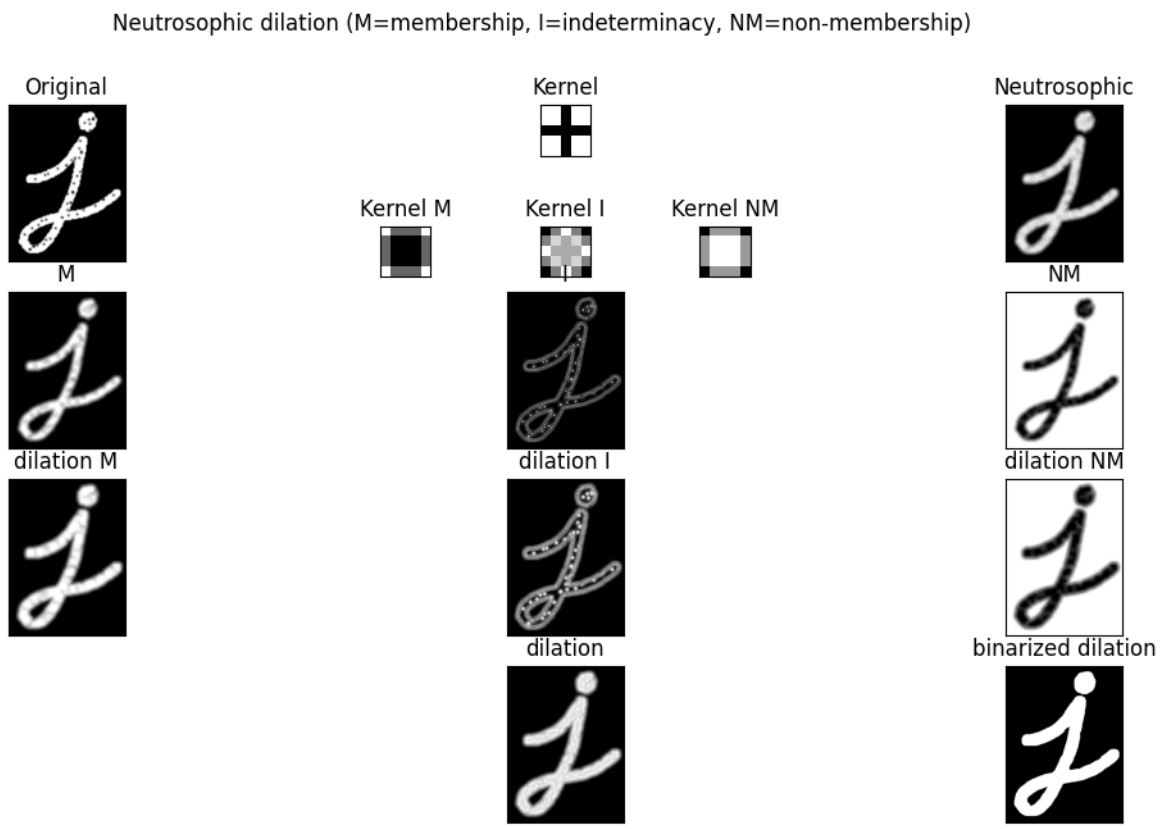

FIGURE 1. Example of the neutrosophic dilation of a neutrosophic image with image radius = 4 and kernel radius = 1

In Figure 1, we observe that the NS-dilation operator effectively smooths the shape of the image while filling in small holes. Notably, the indeterminacy component (I) retains the presence of noise, indicating areas where the operator's action is limited by the uncertainty inherent in the noisy regions.

Figure 2 shows the application of the NS-erosion operator, which reduces protrusions and expands holes. Similar to dilation, erosion captures noise effects, particularly in the membership (M) and non-membership (NM) components. This retention of noise highlights how neutrosophic operators can provide information on uncertainty while performing morphological transformations.

In Figure 3, the NS-opening operator, a sequence of erosion followed by dilation, preserves the indeterminacy and non-membership values in noisy areas but still manages to remove minor structures without disturbing the primary shape of the image. This operator is effective in refining shapes by reducing small-scale noise, emphasizing indeterminacy in regions with higher pixel variability.

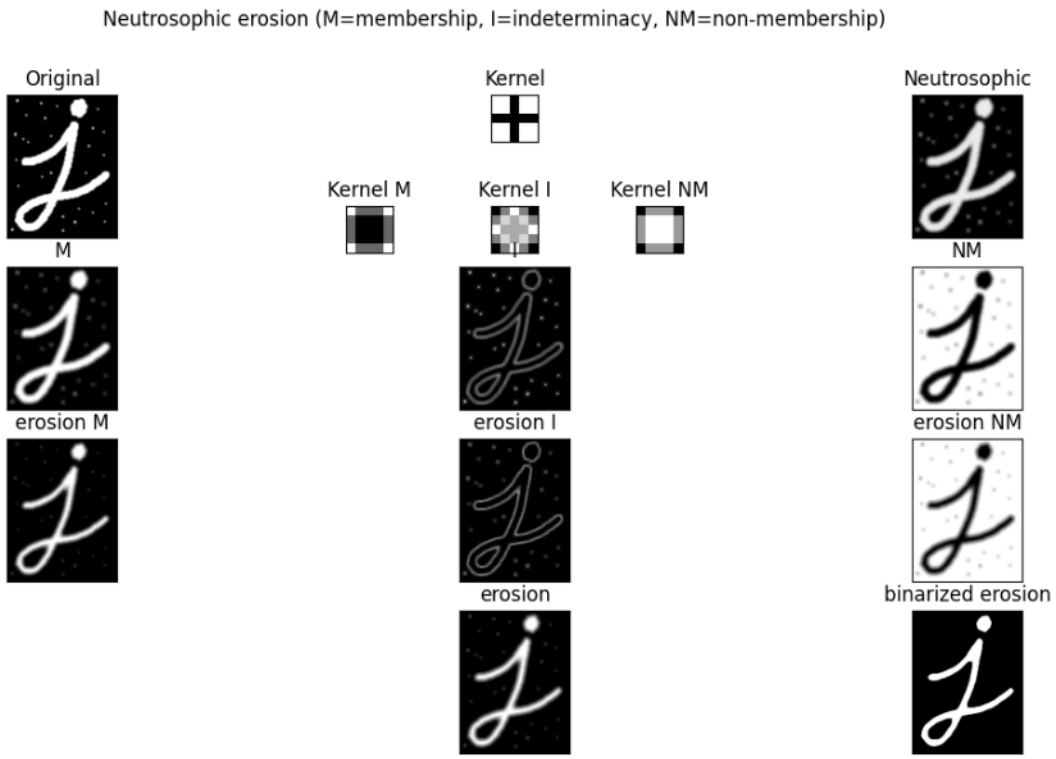

FIGURE 2. Example of the neutrosophic erosion of a neutrosophic image with image radius = 4 and kernel radius = 1

Finally, Figure 4 shows the NS-closing operation, which first applies dilation and then erosion. This operation fills small gaps and connects nearby features, with noise remaining visible in the indeterminacy component. The ability of NS-closing to smooth object boundaries without completely erasing uncertainty demonstrates its utility in applications where preserving ambiguous regions is important.

These examples illustrate the unique advantages of neutrosophic morphological operators, which maintain noise information through indeterminacy while effectively processing image structures. This dual capability of modification and noise retention makes neutrosophic morphology suitable for complex image processing tasks where uncertainty plays a significant role.

4. **Conclusions**

In this paper, we introduced `NSmorph`, a Python class designed for the manipulation of neutrosophic digital images through morphological operators specifically tailored to handle uncertainty and ambiguity. This class implements key neutrosophic operators, including NS-dilation, NS-erosion, NS-opening, and NS-closing, each serving as an extension of classical

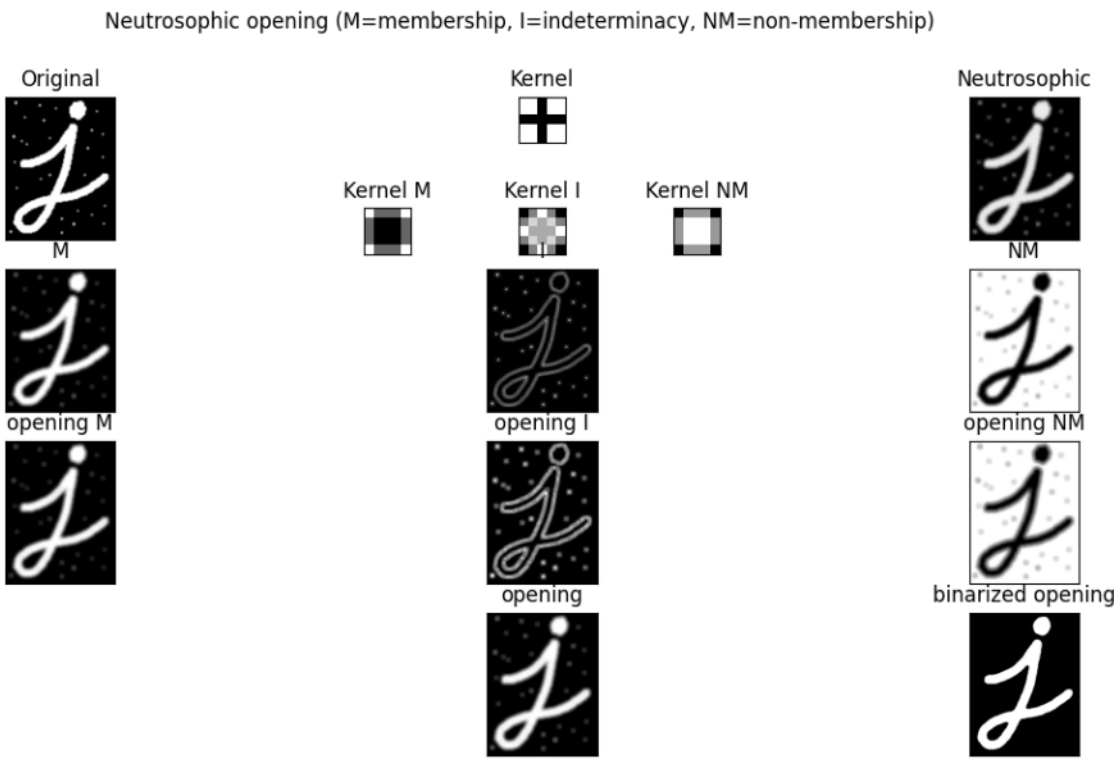

FIGURE 3. Example of the neutrosophic opening of a neutrosophic image with image radius = 4 and kernel radius = 1

morphology in a way that retains the core actions of these transformations while addressing noise and indeterminacy. The design of `NSmorph` includes a flexible constructor and a suite of methods that allow users to access, modify, and visualize the membership, indeterminacy, and non-membership components of an image, adding a layer of control that is essential for applications where uncertain data is prevalent.

The examples provided illustrate how neutrosophic operators differ from traditional morphology, showcasing their capacity to improve image processing outcomes in noisy environments. In particular, the neutrosophic dilation and erosion operations demonstrated a unique capability to maintain information on uncertainty in the indeterminacy component while performing standard morphological tasks, such as smoothing shapes and filling or reducing holes. This dual function of transforming image structures while retaining noise information in the indeterminacy channel provides a substantial advantage for applications that require both enhancement and detailed uncertainty analysis, such as medical imaging, remote sensing, and security surveillance.

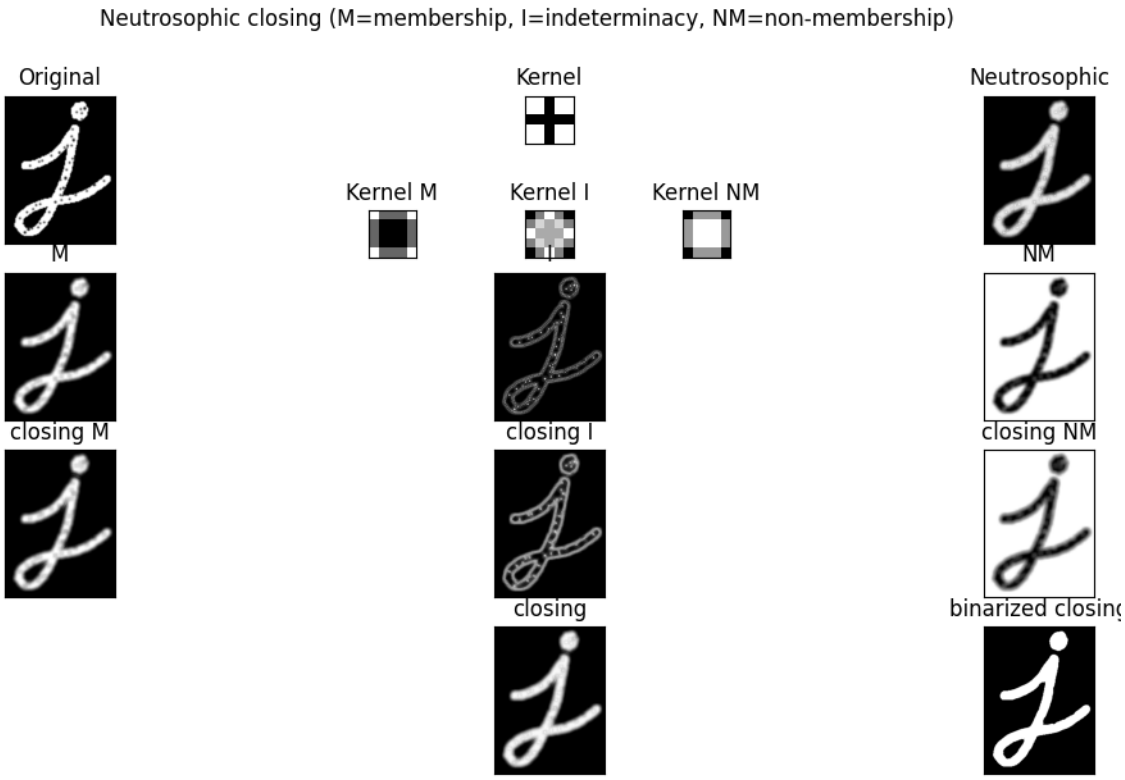

FIGURE 4. Example of the neutrosophic closing of a neutrosophic image with image radius = 4 and kernel radius = 1

Similarly, the neutrosophic opening and closing operations highlighted the class's capability to refine image structures. The NS-opening operator proved effective in reducing small-scale noise while sharpening key features, thus avoiding the loss of critical information. On the other hand, the NS-closing operator showed effectiveness in filling small gaps and connecting nearby structures without compromising the representation of ambiguous areas. These observations underscore how `NSmorph` enables users to adapt morphological operations to complex image processing needs, where standard morphological operators would typically smooth or discard valuable uncertain information.

Overall, the development of `NSmorph` contributes to the field of digital image processing by providing a robust tool for managing indeterminate data, preserving ambiguity, and enhancing the interpretability of images under uncertain conditions. The demonstrated effectiveness of neutrosophic morphological operators in retaining and processing noise information establishes `NSmorph` as a valuable asset for researchers and practitioners who require enhanced flexibility and control over image data affected by noise or uncertainty. Future work may explore

additional neutrosophic operators or adaptations of existing ones to further expand the class's utility across diverse image processing applications.

Acknowledgment

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. Dougherty E.R. An Introduction to Morphological Image Processing. *SPIE Optical Engineering Press*, Bellingham, Washington (USA), 1992.
3. Dougherty E.R., Lotufo R.A. Hands-on Morphological Image Processing. *SPIE Press*, Bellingham, Washington (USA), 2003.
4. Broumi S., Talea M., Bakali A., Smarandache F. Single valued neutrosophic graphs. *Journal of New theory* **10**, pp. 86-101, 2016.
5. El-Ghareeb H.A. Novel Open Source Python Neutrosophic Package. *Neutrosophic Sets and Systems* **25**, pp. 136-160, 2019.
6. 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.
7. 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.
8. Nordo G., Mehmood A., Broumi S. Single Valued Neutrosophic Filters. *International Journal of Neutrosophic Science* **6**, pp. 8-21, 2020.
9. 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.
10. 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.
11. 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.
12. Salama A.A., El-Hafeez S.A., El-Nakeeb A., El-Hassanein Eman M. Neutrosophic Approach for Mathematical Morphology. *Master's thesis*, Port Said University, 2018.
13. Serra J. Image Analysis and Mathematical Morphology. *Academic Press*, London (UK), 1982.
14. Serra J., Soille P. (Eds.) Mathematical Morphology and Its Applications to Image and Signal Processing. *Klwwer Academic Publishers*, Dordrecht (Netherlands), 1994.
15. Shih F.Y. Image processing and Mathematical Morphology: Fundamentals and Applications. *CRC Press*, New York (USA), 2009.
16. 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.
17. Smarandache F. A Unifying Field in Logics. Neutrosophy: Neutrosophic Probability, Set and Logic. Rehoboth: American Research Press, 1999.

18. Wang H., Smarandache F., Zhang Y.Q., Sunderraman R. Single Valued Neutrosophic Sets. *Technical Sciences and Applied Mathematics*, pp. 10-14, 2012.
19. Wilson J.N, Ritter G.X. Handbook of Computer Vision Algorithms in Image Algebra (2nd edition). *CRC Press*, New York (USA), 2000.
20. Zadeh L.A. Fuzzy Sets, *Information and Control* **8** (3), pp. 338-353, 1965.

Received: July 26, 2024. Accepted: Oct 25, 2024