Introduction

Today, noise reduction in wildlife images is a challenging task because images of wildlife have mixed noise and it is not possible to completely determine the type and intensity of noise inside those images and fix them.

Almost all of the existing algorithms in this context remove noise that is added manually to the image. While, as mentioned above, the noises of wildlife images are mixed and unpredictable.

In this paper, we propose a novel method for reducing mixed noise in wildlife images using the sparse error representation method.

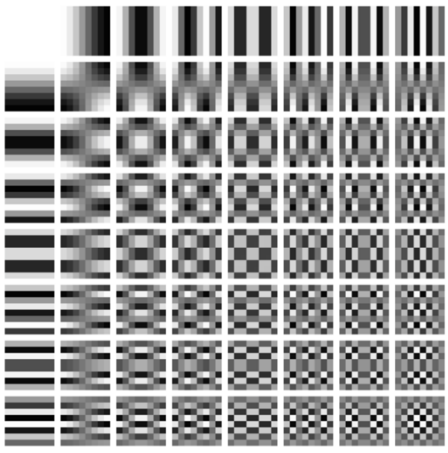
Used ideas to reduce mixed noise:

1 - Converting the original image into two sub-images, including high-frequency and low-frequency sub-images

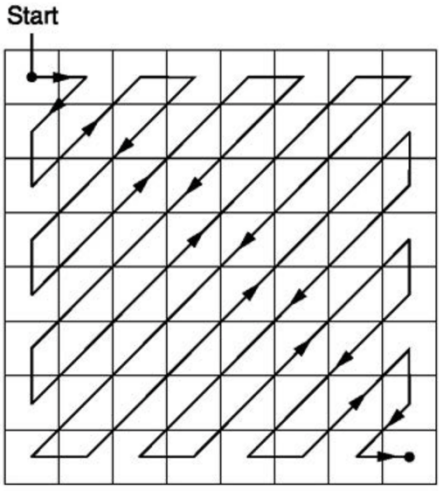
The low-frequency sub-image includes the generality of the main image.
The high-frequency sub-image includes image edges and mixed noise.

This operation is performed with the help of 2D-DCT frequency conversion. First, the original images are divided into 8 x 8 blocks without overlapping.

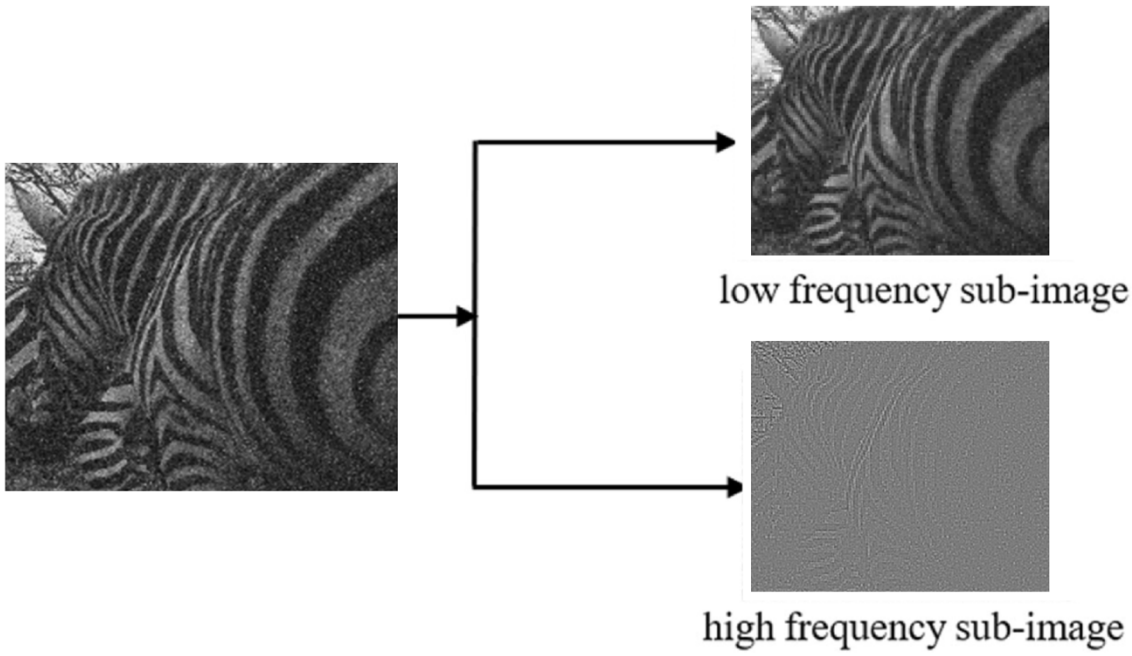
Each of these blocks is taken to the frequency space using 2D-DCT transformation, and then, as mentioned in the article, the first house of the 2D-DCT transformation matrix is kept as the generalities of the image using the Zigzag-Sorting approach, which includes its average brightness and set the value of the rest of the houses to zero. After doing this, we have blocks containing low-frequency images. Then we put these blocks together to recover the original image that includes the low-frequency sub-image.



DCT-8*8

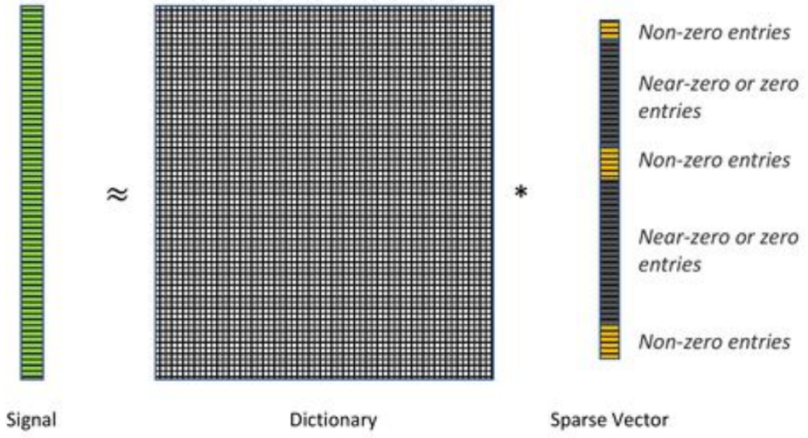


Then, after subtracting the original image from the image including low frequencies, the high-frequency sub-image including mixed noise and image edges is obtained.



2 - Using the approach of finding and training the best dictionary and the best Sparse coefficients, we try to retrieve the high-frequency sub-image. In the recovery done, the obtained image is without any noise and the edges of the image are.

$$x \approx D * a$$



In the presented error function, lp-l1 is used to learn the dictionary of the image edge pattern and at the same time to maintain the sparsity of the recovery coefficients.
As the proximal alternating linearization minimization approach has been used to obtain and find the best state of recovery coefficients. Because the final error function is a non-convex and non-smooth function.

3- After implementing everything mentioned in the article and retrieving the high-frequency sub-image from the trained dictionary (including patterns from the edges of the image) and the best retrieving coefficients, we will have:

High-frequency sub-image + low-frequency sub-image = noise-free image including the generality of the original image along with noise-free edges

Functions used to implement the paper:

1- Entering the necessary libraries

```
import numpy as np
import cv2
from glob import glob
import matplotlib.pyplot as plt
import csv
from scipy.fftpack import dct, idct
from scipy.signal import wiener
from scipy.ndimage import median_filter
from skimage.metrics import structural_similarity as ssim
import torch

# PSNR
from math import log10, sqrt

# Proximal alternating linearilization minimization
#https://pyproximal.readthedocs.io/en/stable/api/generated/pyproximal.optimization.palm.PALM.html#pyproximal.optim
ization.palm.PALM

from pyproximal.optimization.palm import PALM
from pyproximal.utils.bilinear import BilinearOperator
from pyproximal import L1
```

We enter the necessary libraries to implement the functions into the project environment.

2- Function to calculate the value of the PSNR metric

```
def PSNR(original, denoised):
    mse = np.mean((original - denoised) ** 2)
    if (mse == 0):  # MSE is zero means no noise is present in the signal
.       # Therefore PSNR have no importance.
        return 100
    max_pixel = 255.0
    psnr = 20 * log10(max_pixel / sqrt(mse))
    return psnr
```

3- function to calculate the value of the SSIM metric

```python
def calculate_ssim(original, denoised):
    if not original.shape == denoised.shape:
        raise ValueError('Input images must have the same
dimensions.')
    if original.ndim == 2:
        return ssim(original, denoised)
    elif original.ndim == 3:
        if original.shape[2] == 3:
            ssims = []
            for i in range(3):
                ssims.append(ssim(original, denoised))
            return np.array(ssims).mean()
        elif original.shape[2] == 1:
            return ssim(np.squeeze(original), np.squeeze(denoised))
    else:
        raise ValueError('Wrong input image dimensions.')
```

4- Functions to transform the image from space to frequency space

```python
def dct2(block):
    return dct(dct(block.T).T)

def idct2(block):
    return
idct(idct(block.T).T)
```

5- Function to transform the original image into 8×8 blocks.

```python
def get_blocks(image, block_size=8):

    patches = [image[i:i + 8, j:j + 8] for i in range(0, image.shape[0], 8) for j in range(0, image.shape[1], 8)]

    # patches = []
    # for i in range(0, image.shape[0], 8):
    #     for j in range(0, image.shape[1], 8):
    #         patch = image[i:i + 8, j:j + 8]
    #         patches.append(patch)

    return np.array(patches)
```

6- Function to obtain the original image from many blocks

```python
def get_image(patches, h, w):

    image = np.zeros((h, w))
    c = 0
    for i in range(0, image.shape[0], 8):
        for j in range(0, image.shape[1], 8):
            image[i:i + 8, j:j + 8] = patches[c]
            c += 1

    return image
```

7- Functions to convert block from space to frequency space

```python
def get_patch_dct(patches):
    return [dct2(patch) for patch in patches]


def get_patch_idct(patches):
    return [idct2(patch) for patch in patches]
```

8- Function to extract the low-frequency sub-image from the original image

```python
def generate_low_frequency_image(image):
    patches = get_blocks(image)
    patches = get_patch_dct(patches)

    low_freq = []
    for p in patches:
        p[1:] = 0
        p[:, 1:] = 0
        low_freq.append(p)

    low_patches = get_patch_idct(low_freq)
    low_image = get_image(low_patches, image.shape[0], image.shape[1])
    low_image = (low_image - low_image.min()) / (low_image.max() - low_image.min())

    return low_image
```

First, we divide the original image into 8×8 blocks.

Then we apply the 2D-DCT transformation on the 8×8 blocks.

Then we keep the first element of the DCT conversion as a low-frequency element and consider the rest of the elements to be zero so that the brightness of each pixel inside each block is converted to the average brightness of that block.

Then we put the blocks that have become low-frequency blocks together to form the final image which is the low-frequency sub-image.

Then we normalize the final image and return it to the output.

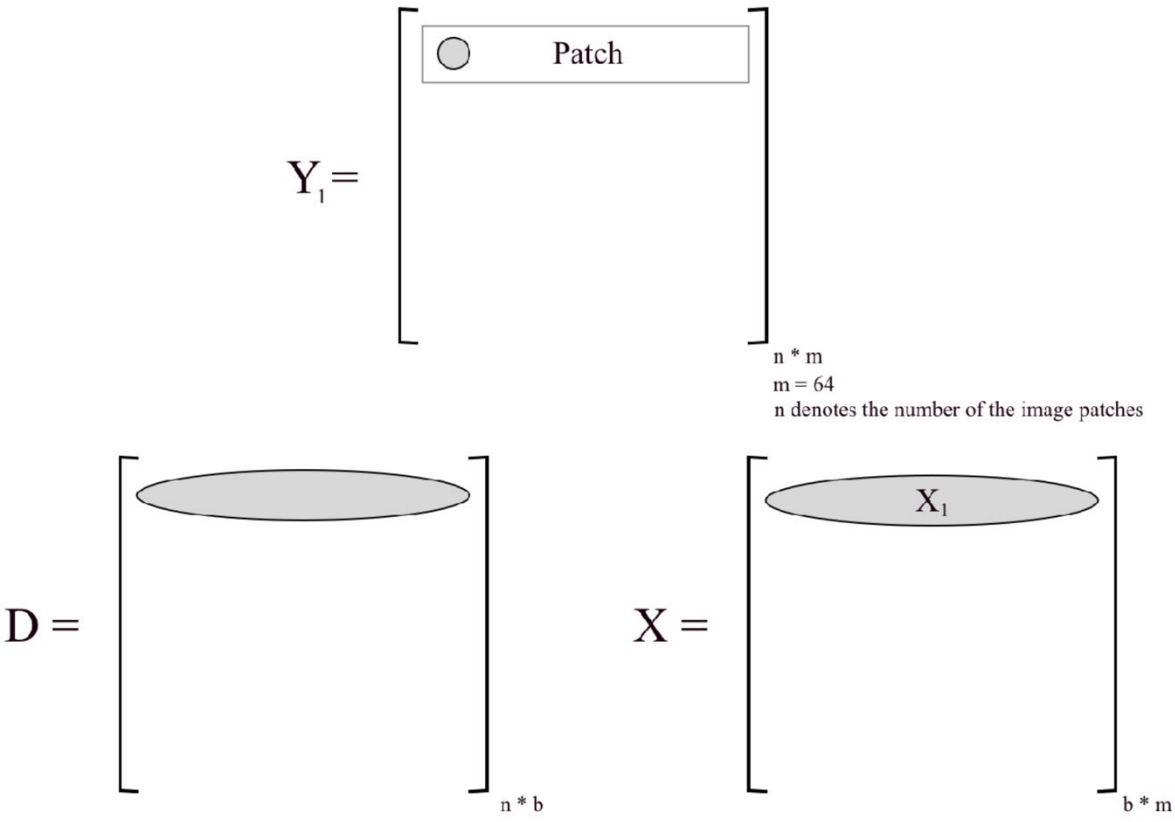
9- The final cost function

$$NH_y = \underset{\{NH_i\},\{DH_i\}}{argmin} \sum_{i=1}^{K} \left(h_\mu(YH_i - DH_i NH_i) + \lambda_i \| MH_i - NH_i \|_1 \right) \qquad (9)$$

The formula of the obtained error function is implemented in the following function:

```python
def minimize_error_function_9(YHi, DHi, MHi, NHi0, lam, p, mu, lr, iter):

    def h_mu(x, p, mu):
        return torch.sum((x ** 2 + mu) ** (p / 2))

    # YHi.shape = (8, 8) -> (64)
    # DHi.shape(k, 8, 8) -> (k * 64)
    # NHi.shape = (n * k)

    NHi = torch.autograd.Variable(NHi0, requires_grad=True)
    optimizer = torch.optim.Adam([NHi], lr=lr)
    history = []
    for i in range(iter):
        f = YHi - torch.matmul(NHi, DHi)
        f = h_mu(f, p, mu)
        f += lam * torch.sum(torch.abs(MHi - NHi))

        optimizer.zero_grad()
        f.backward()
        optimizer.step()
        history.append(f.item())

    # plt.plot(history)
    # plt.show()

    return NHi.detach().numpy(), f.item()
```

The dimensions considered for the dictionary including the image edge pattern and its matrix of coefficients are provided in the following code and image:



10- Function for dictionary construction to learn the pattern of image edge

```python
def construct_initial_dictionary(high_images, high_images_median, k=10, d=100):

    patches = []
    for image in high_images:
        patches += list(get_blocks(image))
    patches = np.array(patches)
    patches = patches.reshape(-1, 64)

    patches_median = []
    for image in high_images_median:
        patches_median += list(get_blocks(image))
    patches_median = np.array(patches_median)
    patches_median = patches_median.reshape(-1, 64)

    dictionaries = [patches[np.random.permutation(patches.shape[0])[:d]] for _ in range(k)]

    # dictionaries = []
    # for _ in range(k):
    #     di = patches[np.random.permutation(patches.shape[0])[:d]]
    #     dictionaries.append(di)

    return dictionaries, patches, patches_median
```

This function takes the original image and the resulting image of the median filter in the input and produces a certain number of k dictionaries to find the best result from these dictionaries.

11- Function to train coefficients or Sparse Representation

```python
def calculate_sparse_representation(dictionaries, MH, NH0, patches, lam, p, mu, lr, iter):

    sparse_representations = []
    best_val = np.inf
    best_idx = 0

    for i in range(len(dictionaries)):
        ret = minimize_error_function_9(torch.from_numpy(patches).float(),
                                        torch.from_numpy(dictionaries[i]).float(),
                                        torch.from_numpy(MH[i]).float(),
                                        torch.from_numpy(NH0[i]).float(),
                                        lam, p, mu, lr, iter)
        sparse_representation, value = ret
        if best_val > value:
            best_idx = i
            best_val = value
        sparse_representations.append(sparse_representation)

    return sparse_representations, best_idx
```

12- Added class to determine the sizes of dictionaries and sparse representation.

```python
class approx_dictionary_h(BilinearOperator):
    # x = NHi / MHi
    # y = DHi
    def __init__(self, YH, MHi, MDi, p, mu):
        super().__init__()
        self.YH = YH
        self.x = MHi
        self.y = MDi
        self.p = p
        self.mu = mu

    def h_mu(self, t, p, mu):
        return np.sum((t ** 2 + mu) ** (p / 2))

    def __call__(self, x, y):
        self.updatex(x)
        self.updatey(y)
        res = self.h_mu(self.YH - np.dot(self.x, self.y), self.p, self.mu)
        return res

    def gradx(self, x):
        # d/dMHi (h(Yi - MDi.MHi)) = d/dx h(x) * MDi
        # d/dx h(x) = p * x * (x^2 + mu) ^ (p / 2 - 1)
        # d/dMHi (h(Yi - MDi.MHi)) = p * x * (x^2 + mu) ^ (p / 2 - 1) * MDi
        f = self.YH - np.dot(x.reshape(self.x.shape), self.y)
        dMHi = -np.dot(self.p * f * (f ** 2 + self.mu) ** (self.p / 2 - 1), self.y.T)
        return dMHi

    def grady(self, y):
        # d/dMHi (h(Yi - MDi.MHi)) = d/dx h(x) * MDi
        # d/dx h(x) = p * x * (x^2 + mu) ^ (p / 2 - 1)
        # d/dMHi (h(Yi - MDi.MHi)) = p * x * (x^2 + mu) ^ (p / 2 - 1) * MDi
        f = self.YH - np.dot(self.x, y.reshape(self.y.shape))
        dMDi = -np.dot(self.x.T, self.p * f * (f ** 2 + self.mu) ** (self.p / 2 - 1))
        return dMDi

    def lx(self, x):
        # lx = d^2 / dx^2 H(w) , YH = Y.W
        # f = YH - X.Y, YH = Y.W => f = 0
        # d^2 / dx^2 H(w) = 2 * (p/2 - 1) * p * Y * f^2 * (f^2 + mu) ^ (P/2 - 2) + p * y^2 * (f^2 + mu)^(p/2 -
1)
        # lx = p * y^2 * mu^(p/2 - 1)
        return self.p * np.max(self.y ** 2) * mu ** (p / 2 - 1)

    def ly(self, y):
        # ly = d^2 / dy^2 H(w) , YH = W.X
        # f = YH - X.Y, YH = W.X => f = 0
        # d^2 / dy^2 H(w) = 2 * (p/2 - 1) * p * X * f^2 * (f^2 + mu) ^ (P/2 - 2) + p * X^2 * (f^2 + mu)^(p/2 -
1)
        # ly = p * X^2 * mu^(p/2 - 1)
        return self.p * np.max(self.x ** 2) * mu ** (p / 2 - 1)

    def updatex(self, x):
        self.x = x

    def updatey(self, y):
        self.y = y
```

13- Function for optimizing and updating the dictionary and sparse Representation

Compute $\{(MH_i, MD_i)\}$ by the equations (12) and (13);

```python
def calculate_prox_update(Yi, MDi, MHi, p, mu, gamma1, gamma2, lam):

    # H(x, y) -> function of MHi and MDi
    # proxf(x) -> function of only MHi
    # proxg(y) -> function of only MDi

    H = approx_dictionary_h(Yi, MHi, MDi, p, mu)

    proxf = L1(lam)
    proxg = None

    MHi, MDi = PALM(H, proxf, proxg, MHi, MDi, gamma1, gamma2, niter=50, show=True)
    MHi, MDi = PALM(H, proxf, proxg, MHi, MDi, gamma1 / 2, gamma2 / 2, niter=50, show=True)
    MHi, MDi = PALM(H, proxf, proxg, MHi, MDi, gamma1 / 4, gamma2 / 4, niter=50, show=True)


    return MHi, MDi
```

Optimization, as mentioned in the paper, is carried out using the PALM library and method.

14- function for calculating the stop time of the training process

Compute (A^i_{MH}, A^i_{MD}) by the equations (14) and (15);

```python
def calculate_stop_condition(Yi, MHi, MHi1, MDi, MDi1, p, mu):

    H1 = approx_dictionary_h(Yi, MHi, MDi, p, mu)
    ci = H1.lx(MHi)
    di = H1.ly(MDi)

    H2 = approx_dictionary_h(Yi, MHi1, MDi1, p, mu)

    AMH = ci * (MHi - MHi1) + H2.gradx(MHi1) - H1.gradx(MHi)
    AMD = di * (MDi - MDi1) + H2.grady(MDi1) - H1.grady(MDi)

    return AMH, AMD
```

15- Project main function

```
results = []
for image in glob('data/*'):

  #Section-1

    k = 1  # number of dictionaries
    d = 600  # number of patterns in each dictionary
    epsilon = 1e-3  # optimization stop criteria
    lam = 1e-8  # L1 regularization
    p = 0.9  # power
    mu = 0.5
    gamma1 = 10000
    gamma2 = 10

    #Section-2

    img = cv2.imread(image, 0)
    new_size = (int(img.shape[1] / 16) * 8, int(img.shape[0] / 16) * 8)
    img = cv2.resize(img, new_size)
    img = (img - img.min()) / (img.max() - img.min()) * 255

    rows, cols = img.shape
    noise = np.ones_like(img) * 0.2 * (img.max() - img.min())
    rng = np.random.default_rng()
    noise[rng.random(size=noise.shape) > 0.5] *= -1
    img_noise = img + noise
    img_noise = (img_noise - img_noise.min()) / (img_noise.max() - img_noise.min()) * 255

    #Section-3

    low_image = generate_low_frequency_image(img_noise)

    high_image = img_noise - low_image

    #Section-4

    high_image_median = median_filter(high_image, 3)
```

Section 1

First, we set the hyperparameters specified for the algorithm.

Section 2

We enter the main image into the project.
We change the original image so that its length and width are divisible by 8. Because finally, we have to divide it into 8×8 blocks.
Then we create a matrix of mixed noise and add it to the original image.

Section 3

We obtain a low-frequency sub-image using a given function.
Then we get the high-frequency sub-image by subtracting the original image from the low-frequency image.

Section 4

We save the high-frequency sub-image using the median filter in the corresponding variable.

```
#Section-5

DH, patches, patches_median = construct_initial_dictionary([high_image], [high_image_median], k, d)
init_zeros = np.zeros((len(DH), len(patches), DH[0].shape[0]))

#Section-6

MH, _ = calculate_sparse_representation(DH, init_zeros, init_zeros, patches, lam, p, mu, 0.001, 100)
NH, _ = calculate_sparse_representation(DH, MH, MH, patches_median, lam, p, mu, 0.0005, 100)

#Section-7

Amh = np.array([0])
Amd = np.array([0])
for j in range(len(DH)):
    while max(np.abs(Amh).max(), np.abs(Amd).max()) < epsilon:
        # MH[j] = minimize_error_function_12(patches, DH[j], MH[j], p, ci, mu)
        # DH[j] = minimize_error_function_13(patches, DH[j], MH[j], p, ci, mu)
        MH_new, DH_new = calculate_prox_update(patches, DH[j], MH[j], p, mu, gamma1, gamma2, lam)
        Amh, Amd = calculate_stop_condition(patches, MH[j], MH_new, DH[j], DH_new, p, mu)
        MH[j], DH[j] = MH_new, DH_new
        print(np.abs(Amh).max(), np.abs(Amd).max())

#Section-8

NH, best_idx = calculate_sparse_representation(DH, MH, NH, patches_median, lam, p, mu, 0.0001, 100)

high_image = np.dot(NH[best_idx], DH[best_idx])
high_image = get_image(high_image.reshape(-1, 8, 8), img.shape[0], img.shape[1])
high_image = (high_image - high_image.min()) / (high_image.max() - high_image.min())

#Section-9

low_image = wiener(low_image, (16, 16))

img_result = cv2.addWeighted(low_image, 1, high_image, 1, 0.0)
img_result = (img_result - img_result.min()) / (img_result.max() - img_result.min()) * 255

#Section-10

SSIM_noisy_value = calculate_ssim(img, img_noise)
PSNR_noisy_value = PSNR(img, img_noise)

SSIM_result_value = calculate_ssim(img, img_result)
PSNR_result_value = PSNR(img, img_result)
```

Section 5

We create the required dictionaries with the corresponding function.

Section 6

Sparse Representation matrices named MHi and NHi are created and initialized as mentioned in the article using the corresponding functions.

Section 7
We implement this part of the article:

While $\max_i \left|(A^i_{MH}, A^i_{MD})\right| \leq \epsilon$ do
Compute $\{(MH_i, MD_i)\}$ by the equations (12) and (13);
Compute (A^i_{MH}, A^i_{MD}) by the equations (14) and (15);

Section 8

We consider the best MHi and NHi matrices, and obtain the high-frequency sub-image as follows:

High-frequency Sub Image = Dictionary * Sparse Representation

Section 9

As mentioned in the article, we apply the Wiener filter on the low-frequency sub-image.

Then, we add the obtained high-frequency sub-image with the low-frequency sub-image on which the Wiener filter is applied to obtain the final de-noised image.

Section 10
We obtain the desired metrics on the de-noised image.

```
#Section-11

result = {'Image': image, 'PSNR': PSNR_result_value, 'SSIM': SSIM_result_value}
results.append(result)

fields = ['Image', 'PSNR', 'SSIM']

with open('Results.csv', 'w', newline='') as file:
    writer = csv.DictWriter(file, fieldnames=fields)

    writer.writeheader()

    writer.writerows(results)

#Section-12

plt.figure(figsize=(10., 10.))
plt.subplot(1, 2, 1)
plt.title(f'Noisy image: PSNR = {PSNR_noisy_value:.2f}, SSIM = {SSIM_noisy_value:.2f}')
plt.imshow(img_noise, cmap="gray")

plt.subplot(1, 2, 2)
plt.title(f'Result: PSNR = {PSNR_result_value:.2f}, SSIM = {SSIM_result_value:.2f}')
plt.imshow(img_result, cmap="gray")
plt.show()

file.close()
```

Section 11

The results in an excel file are saved as CSV files.

Section 12

We put the image containing the mixed noise and the de-noised image together and show it in the output.
We also show PSNR and SSIM measurement metrics for each one.

Article output images:
The following images are presented as the output images of the algorithm in the paper: