Introduction

This report is part of the CSC8628 course, presents the implementation details, including a simplified algorithm flowchart and functional test results, for three tasks including PGM image reading, wavelet decomposition, and image denoising.

TASK 1: Image reading and displaying

In this task, the *read_pgm()* function is designed to read Portable Gray Map (PGM) files specifically, supporting both P2 and P5 types. P2 image is stored in ASCII decimal number, while P5 image is stored in raw binary format [2]. Consequently, decimal conversion is only required for the P5 image. However, Python's built-in functions can convert raw binary value into decimal value within the range of 0 to 255 by iterating over a byte object [3] which simplifying the process. The simplified flowchart of the function is illustrated below.

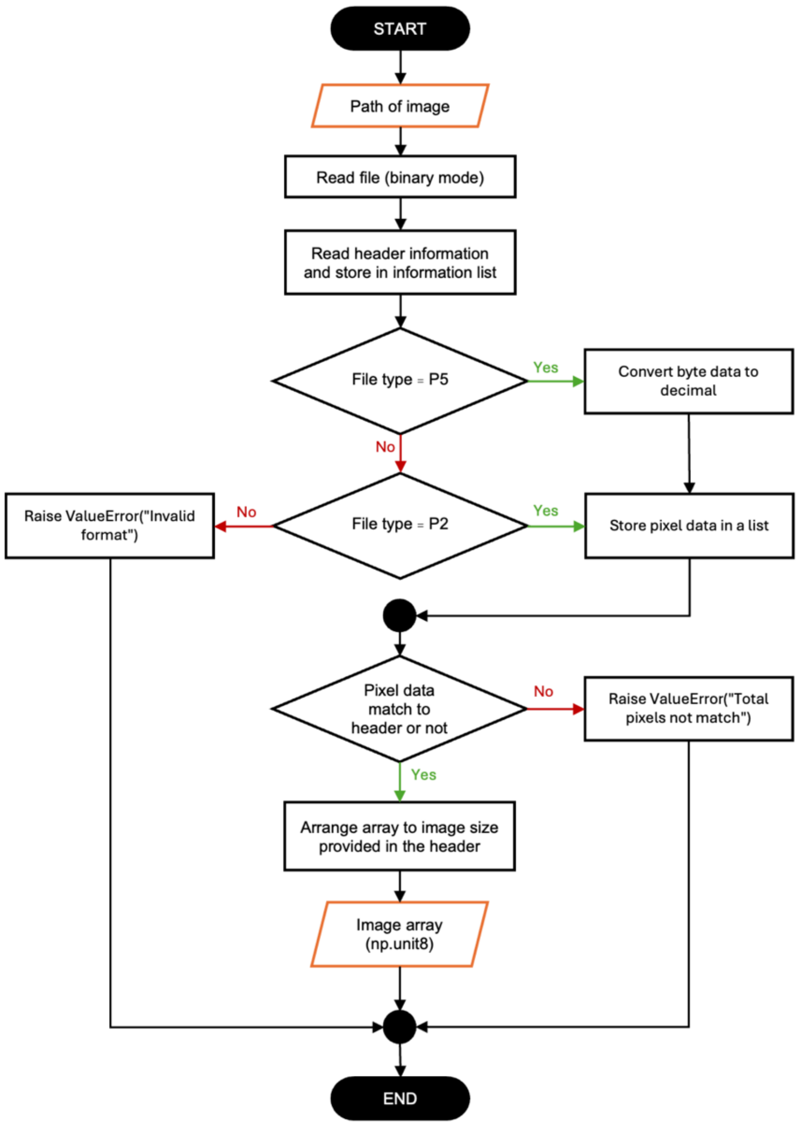

START
Path of image
Read file (binary mode)
Read header information and store in information list
File type = P5
Yes
Convert byte data to decimal
No
File type = P2
Yes
Store pixel data in a list
No
Raise ValueError("Invalid format")
Pixel data match to header or not
No
Raise ValueError("Total pixels not match")
Yes
Arrange array to image size provided in the header
Image array (np.unit8)
END

Figure 1: Simplified flowchart of the *read_pgm()* function

The *read_pgm()* function was tested on three types of PGM images including P2, P5 with headers segmented by spaces, and P5 with headers segmented by tabs. The results demonstrate that *read_pgm()* is successfully process both P2 and P5 file types.

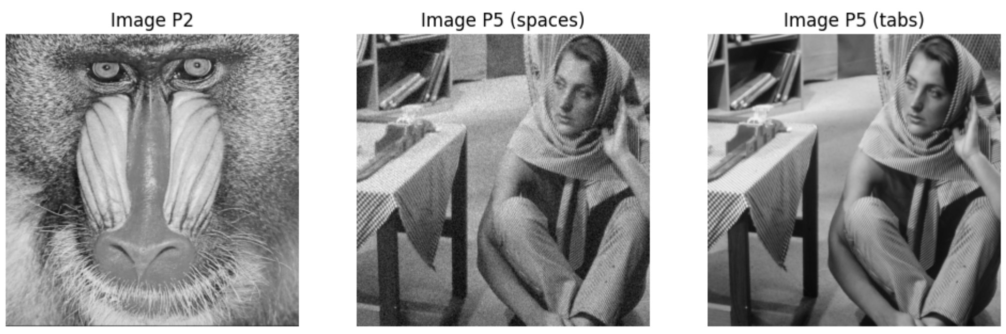


Figure 2: PGM image read by *read_pgm()* function

TASK 2: Wavelet decomposition

Three main functions were developed for this task including convolution, forward discrete wavelet transform, and inverse discrete wavelet transform.

Convolution

The convolution function will calculate the zero-padding for each image based on the kernel size and removes at the end of process differently depending on whether the kernel is flipped. The simplified flowchart of the function is illustrated below.

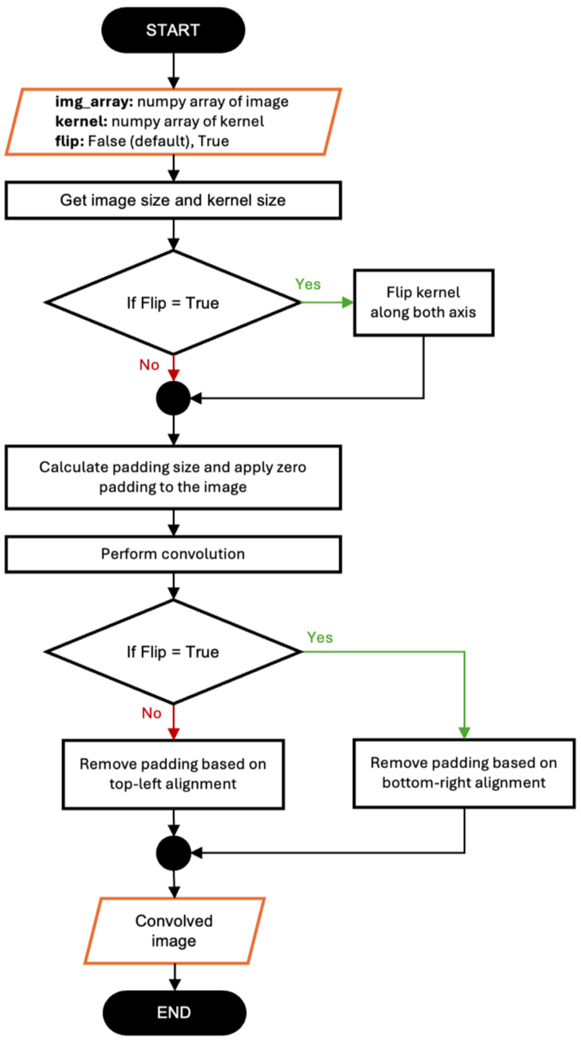

START
img_array: numpy array of image
kernel: numpy array of kernel
flip: False (default), True
Get image size and kernel size
If Flip = True
Yes
Flip kernel along both axis
No
Calculate padding size and apply zero padding to the image
Perform convolution
If Flip = True
Yes
No
Remove padding based on top-left alignment
Remove padding based on bottom-right alignment
Convolved image
END

Figure 3: Simplified flowchart of the *convolution()* function

Forward wavelet transforms

To handle image arrays with odd heights or odd widths, zero padding is applied at the beginning of process to ensure the image dimensions are even. The decomposition is performed iteratively by applying a low-pass kernel ([1/√2, 1/√2]) and a high-pass kernel ([1/√2, 1/−√2]), followed by down sampling, until the desired level is reached. The simplified flowchart is illustrated below.

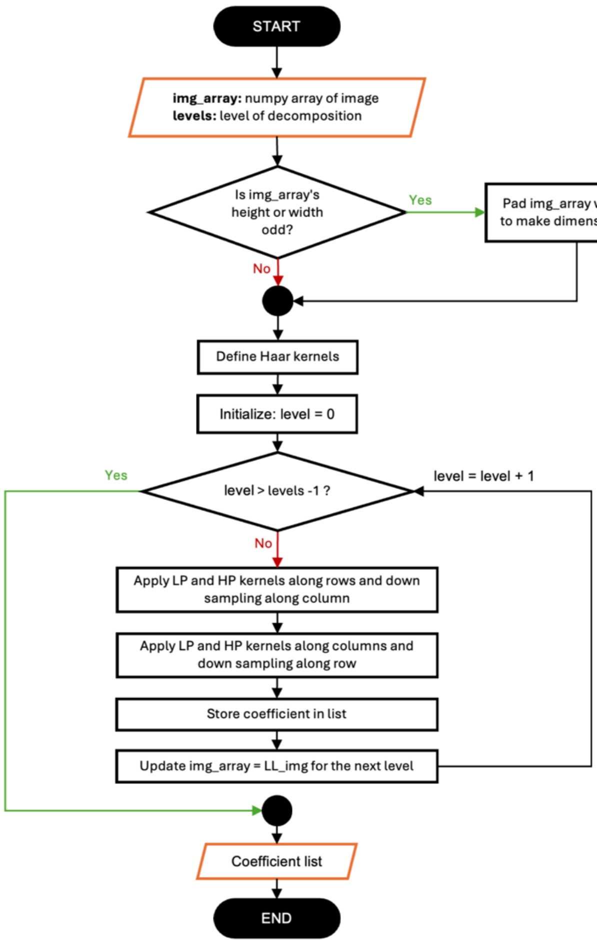

START
img_array: numpy array of image
levels: level of decomposition
Is img_array's height or width odd?
Yes
Pad img_array with zero to make dimension even
No
Define Haar kernels
Initialize: level = 0
Yes
level > levels -1 ?
level = level + 1
No
Apply LP and HP kernels along rows and down sampling along column
Apply LP and HP kernels along columns and down sampling along row
Store coefficient in list
Update img_array = LL_img for the next level
Coefficient list
END

Figure 4: Simplified flowchart of the *FDWT()* function

The example result of the *FDWT()* function is illustrated in the figure below.

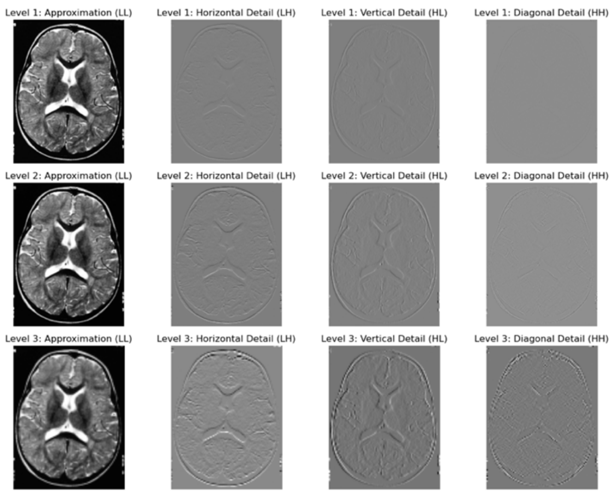

Level 1: Approximation (LL)
Level 1: Horizontal Detail (LH)
Level 1: Vertical Detail (HL)
Level 1: Diagonal Detail (HH)
Level 2: Approximation (LL)
Level 2: Horizontal Detail (LH)
Level 2: Vertical Detail (HL)
Level 2: Diagonal Detail (HH)
Level 3: Approximation (LL)
Level 3: Horizontal Detail (LH)
Level 3: Vertical Detail (HL)
Level 3: Diagonal Detail (HH)

Figure 5: Example result of *FDWT()* function

Inverse wavelet transforms

The input of this function is a list of coefficients calculated by the *FDWT()* function. Reconstruction is performed iteratively, starting from the finest sub-band. This involves up sampling, followed by convolution with the flipped low-pass kernel ([1/√2, 1/√2]) and the flipped high-pass kernel ([1/-√2, 1/√2]). The low and high components are combined to create an approximated image, which serves as the initial LL sub-band for the next level. The reconstruction continues iteratively until all coefficients are processed.

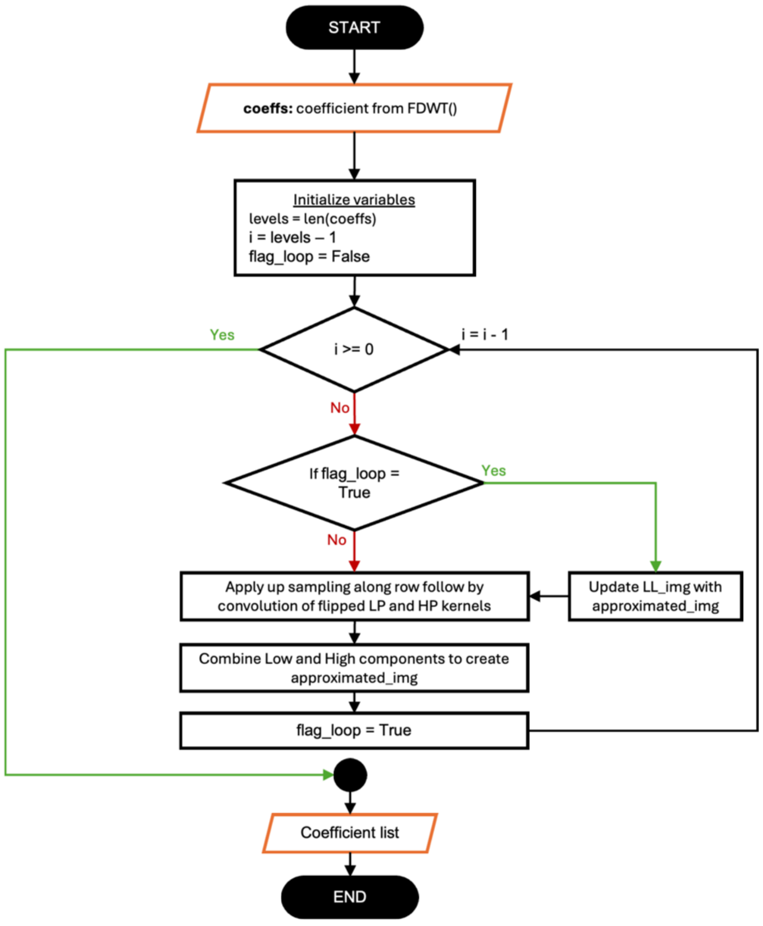

START
coeffs: coefficient from FDWT()
Initialize variables
levels = len(coeffs)
i = levels – 1
flag_loop = False
i >= 0
Yes
i = i - 1
No
If flag_loop = True
Yes
No
Apply up sampling along row follow by convolution of flipped LP and HP kernels
Update LL_img with approximated_img
Combine Low and High components to create approximated_img
flag_loop = True
Coefficient list
END

Figure 6: simplified flowchart of the *IDWT()* function

The example result of the *IDWT()* function is illustrated in the figure below.

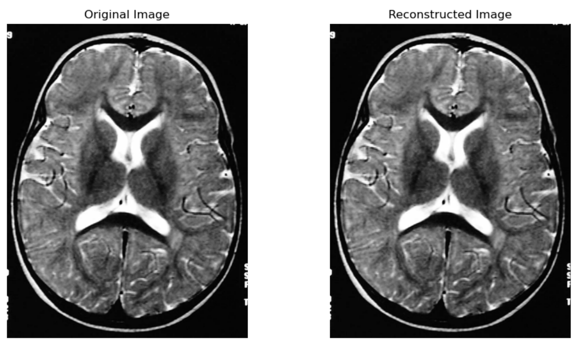


Figure 7: example result of *IDWT()* function

The average mean square error between the original and reconstructed images across the entire dataset is 2.088×10^{-26}, with a minimum error of 1.045×10^{-26} and a maximum error of 4.621×10^{-26}. These results confirm that the forward and inverse wavelet transform functions were executed correctly.

TASK 3: Image denoising

There are numerous methods for image denoising that have been proposed over a decade. Ruikar and Doye demonstrated that Normal Shrink and Sure Shrink are effective in reducing Gaussian and salt-and-pepper noise [4], while Baye Shrink outperforms in Speckle and Poisson noise. Conversely, Dixit and Sharma [5] reveal that the Bayes Shrink method outperforms Visu Shrink and Sure Shrink in reducing Gaussian noise. These implied that distribution and randomness of noise influent the efficiency of noise reduction. Additionally, Dixit and Sharma suggest that denoising in the wavelet domain is more efficient than in the spatial domain [5]. However, the experiment from Jaiswal shows that combination of hard thresholding and filtering given high performance of noise reduction [6]. Therefore, the function in this task aiming to explore the integration of filtering in spatial domain and thresholding in frequency domain to enhance noise reduction capability across various noise types.

Level-dependent thresholding methods, such as BayesShrink, have been reported to exhibit high performance in most studies but low performance in salt-and-pepper noise. Biswas and Om proposed a new soft-thresholding method that calculates threshold based on the universal threshold estimation equation, similar to VisuShrink, but adaptively at each decomposition level [7]. The results show that the new soft-thresholding method outperforms VisuShrink, but there is no evidence to compare it with other methods.

The denoising function in this task is developed based on the combination between level-dependent thresholding estimation from Biswas and Om and median filtering. The flowchart is illustrated in the figure below.

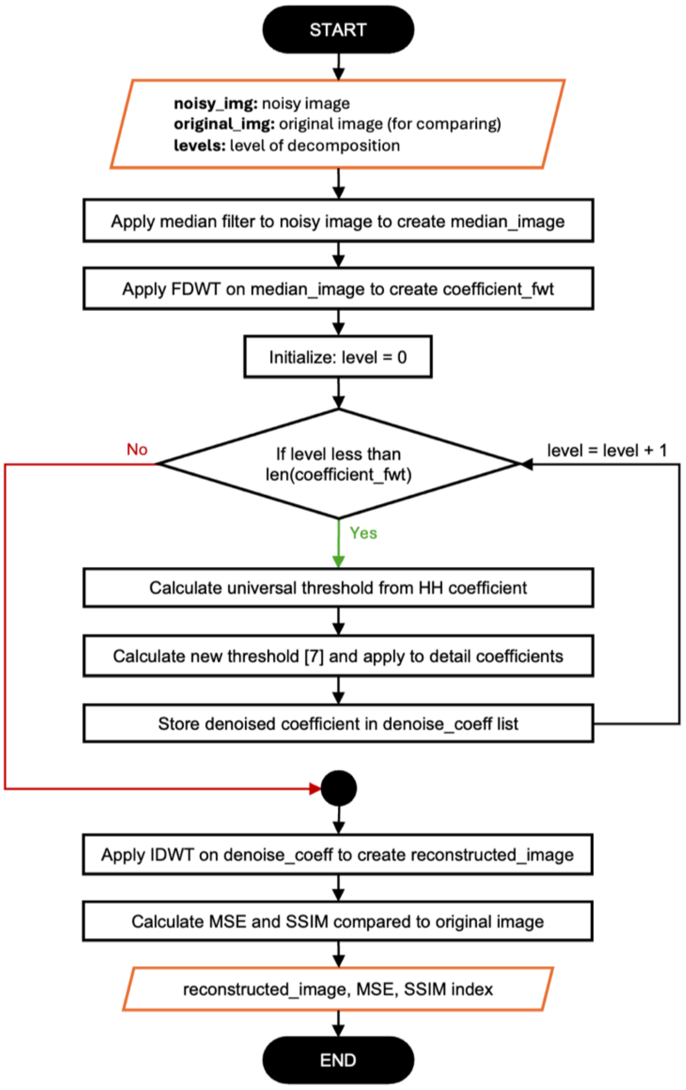

START
noisy_img: noisy image
original_img: original image (for comparing)
levels: level of decomposition
Apply median filter to noisy image to create median_image
Apply FDWT on median_image to create coefficient_fwt
Initialize: level = 0
No
If level less than len(coefficient_fwt)
level = level + 1
Yes
Calculate universal threshold from HH coefficient
Calculate new threshold [7] and apply to detail coefficients
Store denoised coefficient in denoise_coeff list
Apply IDWT on denoise_coeff to create reconstructed_image
Calculate MSE and SSIM compared to original image
reconstructed_image, MSE, SSIM index
END

Figure 8: simplified flowchart of the *IDWT()* function

Four algorithms including Mean filtering, Median filtering, BayeShrink and VisuShrink are compared with the developed algorithm using mean square error (MSE) and structural similarity index (SSIM). The results demonstrate that MSE and SSIM metrics are not strongly correlated. MSE relies on a pixel-by-pixel comparison which ignoring the relationships with neighbouring pixels, while SSIM evaluates luminance, contrast, and structure, making it more aligned with human perception [8]. Therefore, SSIM is primary used to assess reconstruction quality in this task. The mean of MSE and SSIM in each noise images are illustrated below.

Table 1: Evaluation result of each algorithm

Algorithms	**Noise 1**		**Noise 2**		**Noise 3**		**Noise 4**	
	MSE	**SSIM**	**MSE**	**SSIM**	**MSE**	**SSIM**	**MSE**	**SSIM**
Proposed Algorithm	52.98	0.65	61.58	0.58	83.42	0.35	30.60	0.80
Mean Filter	48.41	0.69	55.58	0.65	80.69	0.49	59.50	0.61
Median Filter	48.14	0.68	56.60	0.63	79.92	0.43	32.09	0.77
BayeShrink	50.73	0.73	59.01	0.67	79.61	0.52	6.77	0.39
VisuShrink	69.17	0.56	74.74	0.52	90.48	0.43	44.59	0.39

The table shows that BayesShrink performs best for Noise 1, Noise 2, and Noise 3, while the proposed algorithm demonstrates highest performance for Noise 4. The histogram plots reveal that Noise 1 and Noise 2 exhibit characteristics similar to Gaussian noise with salt-and-pepper noise, whereas Noise 3 displays a flat distribution similar to uniform noise with salt-and-pepper noise. Additionally, the histogram of Noise 4 suggests that the image contains only salt-and-pepper noise. Therefore, BayesShrink, which estimates noise based on a Gaussian distribution, performs better for Noise 1, 2, and 3, while the proposed method, which the combination of level-dependent thresholding and median filtering, perform well with Noise 4.

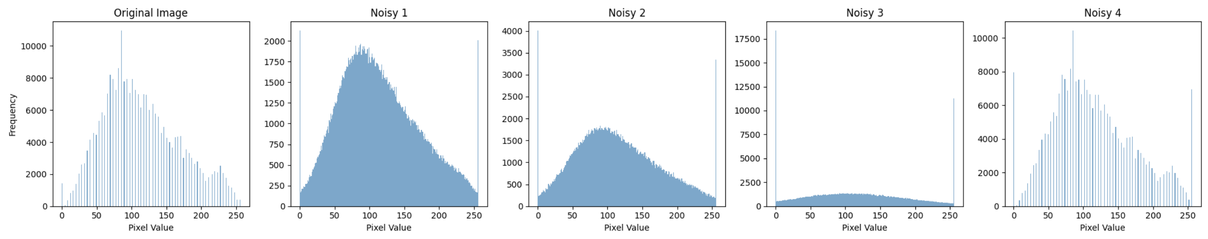
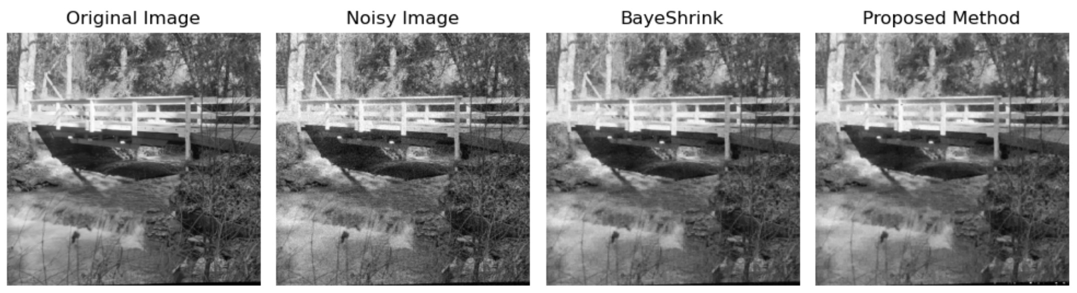


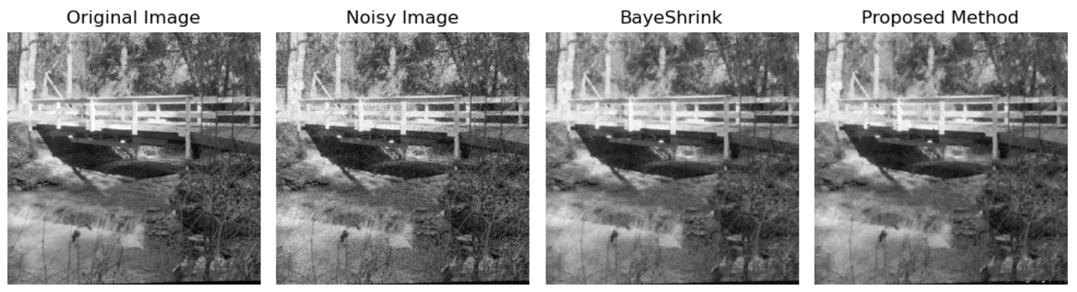
Figure 9: Histograms of the original image and various types of added noise

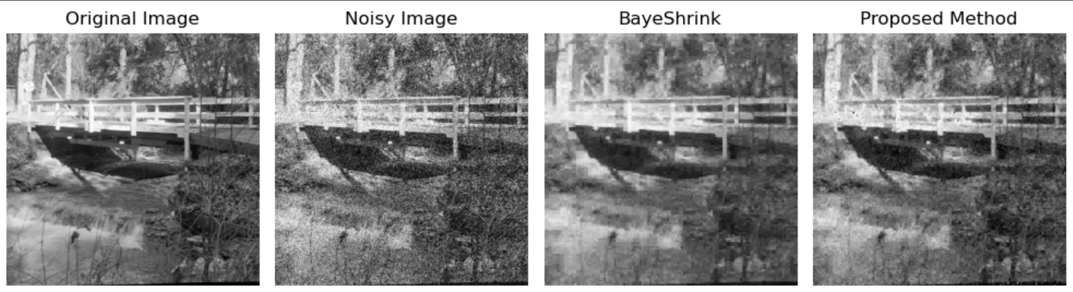
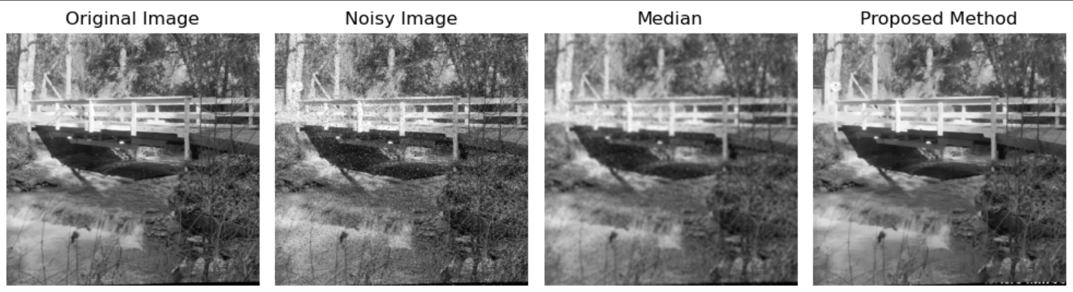
The figures below compare denoised image between BayesShrink thresholding and proposed algorithm.

Noise 1



Noise 2



Noise 3
 Original Image Noisy Image BayeShrink Proposed Method
Noise 4


Conclusion

The mean square error confirms that the forward and inverse discrete wavelet transform functions developed in this assignment perform accurately in decomposing and reconstructing images. However, the processing speed is slower compared to the scikit-image library due to the code structure and programming languages. Additionally, the study highlights that each noise type has unique characteristics, suggesting the need for different denoising methods for each noisy image.

Reference

[1] Rafael Gonzalez and Richard Woods, *Digital Image Processing, Global Edition*, 4th ed. 2017.

[2] J. Poskanzer, “PGM Format Specification.” Accessed: Jan. 02, 2025. [Online]. Available: https://netpbm.sourceforge.net/doc/pgm.html

[3] Python Software Foundation, “Built-in Types — Python 3.13.1 documentation.” Accessed: Jan. 02, 2025. [Online]. Available: https://docs.python.org/3/library/stdtypes.html#bytes-objects

[4] S. Ruikar and D. D. Doye, “Image denoising using wavelet transform,” in *2010 International Conference on Mechanical and Electrical Technology*, IEEE, Sep. 2010, pp. 509–515. doi: 10.1109/ICMET.2010.5598411.

[5] A. Dixit and P. Sharma, “A Comparative Study of Wavelet Thresholding for Image Denoising,” *International Journal of Image, Graphics and Signal Processing*, vol. 6, no. 12, pp. 39–46, Nov. 2014, doi: 10.5815/ijigsp.2014.12.06.

[6] A. Jaiswal, J. Upadhyay, and A. Somkuwar, “Image denoising and quality measurements by using filtering and wavelet based techniques,” *AEU - International Journal of Electronics and Communications*, vol. 68, no. 8, pp. 699–705, 2014, doi: 10.1016/j.aeue.2014.02.003.

[7] M. Biswas and H. Om, “A New Soft-Thresholding Image Denoising Method,” *Procedia Technology*, vol. 6, pp. 10–15, 2012, doi: 10.1016/j.protcy.2012.10.002.

[8] D. P. Kingma and M. Welling, “Auto-encoding variational bayes,” in *2nd International Conference on Learning Representations, ICLR 2014 - Conference Track Proceedings*, International Conference on Learning Representations, ICLR, 2014. doi: 10.61603/ceas.v2i1.33.