Elaborazione di Immagini

Denoising Project

Dario Civale - 0622701620 - d.civale1@studenti.unisa.it
Paolo Mansi - 0622701542 - p.mansi5@studenti.unisa.it
Vincenzo Salvati - 0622701550 - v.salvati10@studenti.unisa.it

26/07/2021

0 Introduction

This project required the implementation of three different filters for denoising images: a Linear technique, a Non-Linear technique and a denoising technique in a transformed domain.

0.1 Work division

Therefore, the division of tasks provided:

- as for the Linear technique, Vincenzo Salvati has implemented the Guided Filtering;
- as for the Non-Linear technique, Dario Civale has implemented the Anisotropic Filtering;
- as for the technique in the transformed domain, Paolo Mansi has implemented the filtering through wavelets decomposition.

Finally, all the member carried out jointly the testing and the comparison of all the techniques using images with different types of noise.

0.2 Files Organization

Each Technique is contained in a different package. In particular:

- "*LinearFiltering*" contains the files regarding the Guided Filter;
- "*NonLinearFiltering*" contains the files regarding the Anisotropic Filter;
- "*TranformedFiltering*" contains the files regarding the denoising through wavelet decomposition.

In addition to that, a script named "*Utils.py*" contains all the utilities used throughout the code and a script named "*compare_filters.py*" contains the final testing and the related comparing of results of all the techniques. In particular, the method "*add_noise()*" allows to add four types of noise: Gaussian, Salt and pepper, Poisson and Speckle.

Finally, the "*images*" directory contains all the images used during testing, properly subdivided into "*b&w*" and "*rgb*", according to the number of channels.

1 Linear Filter

It was decided to implement the denoising of a specific image using a guided image which could be different for other kinds of issues, but it's the same image for performing the denoising.

1.1 Algorithm explanation

To carry on that, it is necessary minimize a quadratic function so that it can be possible to obtain two parameters (a and b) useful to build the output filtering. In particular, is useful minimize the noise and the parameter "a" so that not to perfectly follow the noise of the guide image.

Furthermore, each pixel is included in all the overlapping window and, just for that, we need to perform the mean of "a" and "b" for each window and by them it's possible to write an equation that filters the noisy image:

$$filtered\_Image = mean_a * I + mean_b, \text{ where } I \text{ is the guided image}$$

OpenCV give the possibility to use its function which implements the guided filter, but deriving the quadratic function on the parameter "a" and, subsequently, on the parameter "b", it is possible to refer to a formula composed of means and variances:

- ➢ $a = \frac{var^2}{(var^2 + epsilon)}$, where $epsilon$ adjusts how to follow the variations of the guide image, useful for not following and preserving its noise;
- ➢ $b = (1 - a) * mu_k$, where mu_k is the mean of a specific window in the guided Image.

It's easy to understand that when:

- the variance is high: $epsilon << var^2$ so $a = 1$ and $b = 0$. By the equation, the value of that pixel doesn't change and the edge is preserved;
- the variance is low: $epsilon >> var^2$ so $a = 0$ and $b = mu_k$. By the equation, the value of that pixel is the average of the window considered around the pixel (smoothing).

The principal function that are used in "*Guided_filter.py*" are:

- *def* ***guided_filter_OpenCV*** *(I_guid, I_noisy, epsil, nhoodSize)*: implements the guided filter with OpenCV functions, and the parameters used are:
    - *I_guid*: guide Image;
    - *I_noisy*: image to be filtered;
    - *epsil*: the weight that guides how much following the guide image's edges;
    - *nhoodSize*: dimension of the neighbourhood window;
- *def* ***guided_filter*** *(I_guid, I_noisy, epsil, N)*: implements the guided filter with mean and variance, and the parameters used are:
    - *I_guid*: guide Image;
    - *I_noisy*: image to be filtered;
    - *epsil*: the weight that guides how much following the Guide Image's edges;
    - *nhoodSize*: dimension of the neighbourhood window;

- *def* ***compare_images*** *(I_original, I_noisy, noisy, I_filtred, I_filtred_CV)*: plots the four images passed and computes the MSE, PSNR and SSIM indexes.

1.2 Choice of parameters

The choice of the parameters in this filter consist in using appropriately the constant epsilon and the size of neighborhood. Both regularize different situation:

- ➢ *epsilon*: is called regularization parameter which the higher it is, the more the cost function ignore the quick variations of the guided image;
- ➢ *size of Neighbourhood*: is the window that take in consideration pixels around a specific pixel so that is possible apply a specific smooth in based of the variance on that.

To avoid an excessive smooth, in case of low variance, the size of the windows is set to 3. Whereas, to find the correct epsilon, it is realized two scripts of test:

- *"Testing_epsilon.py"*: to evaluate a range of epsilon for each image in which is applied the noise and the mean of the performance indexes is done for **each** image with each noise;
- *"Testong_epsilon_2.py"*: to evaluate a range of epsilon for each image in which is applied the noise, but the mean of the performance indexes is done for **all** the images with each noise.

From the results that have been produced by these tests seems that $epsilon = 0.01$ is the optimum params for each image (black and white or RGB) and for all the noise considered. It means that the output does not ignore completely the guide image's noise.

2 Non Linear Filter

It was decided to realize an Anisotropic Filter evaluating different conductibility functions, and implementing technique to automatically calculate the necessary parameter in order to build the filter.

2.1 Algorithm explanation

That consist in a Non-Linear diffusion based on partial differential equation. This kind of filter permits to realize a smoothing in noisy areas of the images, designed by the algorithm as low variable area and, at the same time, preserving the highly variable areas (possible edges).

The relevant parameters which define the behavior and the anisotropic diffusion are the following:

- Conductibility function: a function g that as the variation increases x along a specific direction has the following behavior:
    - $\lim_{x \to 0} g(x) = 1$ when the diffusion is maximum in uniform regions;
    - $\lim_{x \to \infty} g(x) = 0$ when the diffusion has to arrest in front of the edge (edge stopping condition).
- Gradient threshold: a parameter K used by the conductibility function which represent the boundary between noise and edge. Therefore, the local gradient under the threshold is considered as noise, whereas over that as edge;
- Number of iterations: the procedure can be iterated n times and the more n is high, the more will be done smoothing.

The algorithm relates these parameters in the following formula:

$$Img^n = Img^{n-1} + \frac{1}{\eta} \sum_{0}^{c} g(\delta_i(Img^{n-1}), K) * \delta_i(Img^{n-1})$$

An overestimation of listed parameters could lead to a blurred result, while an underestimation of them would leave the noise on the image. So, we need the right trade-off for anisotropic filtering to be efficient.

The main function dealing with anisotropic filtering has been used in the "*anisotropic.py*":

- def ***anisotropic_denoising*** *(image, iterations = None, kappas = None, option = 1, neighbourhood = 'minimal')*: implement the anisotropic filter and the parameters used are:
    - *image*: greyscale or RGB image;
    - *iterations*: number of iterations;
    - *kappas*: gradient thresholds;
    - *option*: deals with the choice of conductivity function:
        - if option = 1 the exponential equation is chosen;
        - if option = 2 the quadratic equation is chosen;

- *neighbourhood*:
  - if *neighbourhood = 'minimal'* the partial derivatives along {N, S, E, W} are considered;
  - if *neighbourhood = 'maximal'* the partial derivatives along {N, S, E, W, NE, NW, SE, SW} are considered;
- *Single_threshold = False*: if more than one threshold is to be used per channel, otherwise it must be set to True. The maximum number of thresholds used per channel must be equal to the number of iterations used for that channel.

2.2 Choice of parameters

The anisotropic filtering algorithm proposed in the project has algorithms to automatically estimate the threshold and the number of iterations

➢ Algorithms for calculating the gradient threshold:

- Canny noise estimator: The threshold is equal to 90% of the cumulative sum of the histogram of the magnitude in each iteration. In this way the threshold is adapted to the type of image.
- MAD estimator: the threshold is estimated through the Median Deviation Absolute of the modulus of the image gradient:

$$K = 1.4826 * MAD(\nabla \text{ Img} - \text{median}(\nabla \text{ Img}))$$

- Morphological estimator: To estimate the threshold, first a morphological opening and a morphological closing of the image is done, the result of the opening and closing is summed and respectively divided by the total number of pixels in the image, finally these two scalar quantities are subtracted. A 5x5 kernel was chosen as the structuring element *st* for the morphological operations.

$$K = \sum_{i,j \in Img} Img \circ \frac{\text{st}}{\text{row}} * \text{col} - \sum_{i,j \in Img} Img \bullet \frac{st}{row} * col$$

➢ Algorithm for calculating the number of iterations:

The calculated number of iterations depends on the Q-function which represents the quality of the edge.

So, to calculate this function we need to find N number of edges in different areas in the image. For each edge, two regions close to the edge are defined in which 6 pixels per region are located, called interpixels. The equations to find the coordinates of the interpixels, given an edge point with coordinates (x_k, y_k) and the edge direction θ are the following:

$$x_{m,n} = x_k + m * cos(\theta) - n * sin(\theta) \qquad per\ m = \{-1, -1, 1, 2\}, n = \{-1,0,1\}$$

$$y_{m.n} = y_k - m * sin(\theta) - n * cos(\theta) \qquad per\ m = \{-2, -1,1,2\}, n = \{-1,0,1\}$$

Since the intensity of the interpixels is not known, bilinear interpolation was used.

Each edge is now characterised by two sets of inter-pixel intensity values.

$$R_1 = \{f(x_{m,n}, y_{m.n})\} \; per \; m < 0$$

$$R_2 = \{f(x_{m,n}, y_{m,n})\} \; per \; m > 0$$

Now, at each edge point, the quantity Q is calculated:

$$Q = \left| \mu_1 - \mu_2 \right| - \alpha * |\sigma_1 + \sigma_2|$$

where $\mu_1$and μ_2 are respectively the means of the regions R_1 and R_2, and σ1 and σ2 are the standard deviations of the regions respectively R_1 and R_2. The quantity $\left| \mu_1 - \mu_2 \right|$ is an estimation of the intensity of the variation, while the quantity $|\sigma 1 + \sigma 2|$ is an estimation of the intensity of the noise around the edge.

The constant α is used to estimate the effect of the noise on the edges of the original image and it is calculated as:

$$\alpha = 10 * \frac{\sigma}{\mu_0}$$

Where σ is the standard deviation of the noise of the original image and $\mu_0$is the average contrast between all N edges and it's calculated as:

$$\mu 0 = \left(\frac{1}{\mathrm{N}}\right) \sum_{1}^{N} | \mu_1(\mathrm{t}) - \mu_2(t)|$$

In case the noise is low $\alpha = 0$, while in case the noise is high $\alpha = 10$. As noise increases, edge quality is expressed only by the level of noise, whereas as noise decreases, edge quality depends on the amount of noise $| \mu 1 - \mu 2|$.

Finally, to estimate the image quality, the algorithm calculates the following relationship

$$Qm(t) = \left(\frac{1}{\mathrm{N}}\right) \sum_{i=1}^{N} Q_i(t)$$

At each iteration. We stop at time T that maximizes $Qm(t)$

2.3 Example of use

- Automatic use:

anisotropic_denoising *(image)*

In the case where the function only receives the image it directly uses the implemented algorithms to estimate the gradient thresholds and the number of iterations, this applies to both the greyscale and RGB image.

- Manual use of greyscale image using a single threshold:

anisotropic_denoising *(image, 4, 0.5)*

- Manual use of greyscale image using different thresholds at each iteration:

anisotropic_denoising *(image, 3, [0.3, 0.3, 0.2])*

- Manual use of RGB image using a single threshold per channel:

anisotropic_denoising *(image, [4, 5, 6], [0.3, 0.5, 0.1])*

- Manual RGB image use with multiple thresholds per channel:

anisotropic_denoising (image, [3, 3, 3], [[0.1, 0.3, 0.2], [0.4, 0.5, 0.4], [0.1, 0.1, 0.1]])

3 Trasformed Filter

It was decided to implement the denoising firstly performing the wavelet transform of the image and then a thresholding over the coefficients thus obtained.

3.1 Algorithm explanation

Wavelets transform decomposes a function into a set of wavelets, providing localisation both in time and frequency. For each level of the decomposition, 4 different set of coefficients are produced, according to the possible combination of axes on which apply the transform.

Two functions for thresholding the values of the coefficients were then implemented to set to remove the noise present in the higher frequencies, which are less likely to contain much information:

1. The first thresholding function is the Universal Threshold that makes use of a universal threshold calculated as:

$$\lambda = \sigma * \sqrt{2 * log\, n^2}$$

where σ is the standard deviation of the noise and n is the sample size. To estimate the σ standard deviation it is often used the equation:

$$\sigma = \frac{\text{MAD}}{0.6745}$$

Where MAD represents the median of the absolute values of the wavelet coefficients.

In Universal Threshold, all the coefficients lower of the threshold λ are set to 0, and thus removed.

2. A second algorithm for thresholding is the *NeighShrink*[1], which thresholds the wavelet coefficients according to the magnitude of the square sum of all the wavelet coefficients within a neighbourhood window:

$$S_{j,k}^2 = \sum_{(i,j)\in B_{j,k}} d_{i,l}^2$$

where $d_{i,l}$ are the coefficients of the wavelets of interest.

From this we get to the shrinkage factor:

$$\beta_{j,k} = \left(1 - \frac{\lambda^2}{S_{j,k}^2}\right)$$

where λ is the universal threshold. The values lower than B are set to 0.

Finally, the coefficients of the wavelet are thresholded as followed:

$$d_{j,k} = d_{j,k} * \beta_{j,k}$$

[1] G. Y. Chen, T. D. Bui, and A.Krzyzak, “Image Denoising using Neighbouring Wavelet coefficients”

After the soft thresholding of the coefficients, the image is then reconstructed through the inverse wavelet transform.

The interface for using the denoising through wavelets is:

- *def* ***wavelet_denoising*** *(image, wname=DEFAULT_WNAME, levels=None, mode=DEFAULT_MODE, dim_neigh=DEFAULT_LEVEL_NUMBER, show=False).* The parameter required are:
    - *image:* the image to denoise
    - *wname:* the wavelet to use for the decomposition. If not specified, the Default wavelet will be used
    - *levels:* the number of levels of the wavelet decomposition. If not specified, the default number will be used if possible, or else the maximum number allowed. Passing 'max' as argument set the number of levels to the maximum possible.
    - *mode:* the thresholding mode to be used for the denoising. It can be either 'univ' or 'neigh'. If not specified, the neighbouring thresholding will be used.
    - *dim_neigh:* the dimension of the neighbourhood window
    - *show:* a Boolean flag indicating whether or not showing the wavelet decomposition.

3.2 Choice of the parameters

The method illustrated presents 3 degrees of freedom in the actual denoising of an image:

- The wavelet transforms to be used to decompose the image in the time-frequency plane
- The number of levels of the decomposition
- The function used as a threshold

3.2.1 Wavelet

The choice of the wavelet is strongly connected to the image, and overall, to the type of noise that affects the image.

A Mont Carlo simulation was performed on the four types of noise Gaussian, Salt and Pepper, Poisson and Speckle added to the image "*peppers.png*", and the table below shows their results in terms of the SSIM. The best results for each type of noise are highlighted in green, but however for each type of noise the results above the mean of all SSIMs are highlighted in yellow. The table shows that the same wavelet has different results on different noises. While for Gaussian, Poisson, and Speckle noise the "*bior6.8*", the "*coif3*" and "*sym14*" wavelets seems the best choice that is not true regarding the salt and pepper noise, which gives better results with a Daubechies wavelet with a high number of vanishing moments.

The wavelet "*db10*" was chosen because it has averagely good results in all types of noise, even if it's not the best choice for each one of them.

The file "*testing_over_wnames.py*" provides the denoising testing of a single image with a specified noise added, using all the possible discrete wavelets, but keeping the number of levels and the dimension of the neighbourhood as default.

	Gaussian	S&p	Poisson	Speckle
bior3.7	0,82	0,56	0,93	0,92
bior3.9	0,83	0,58	0,93	0,91
bior4.5	0,89	0,5	0,95	0,92
bior5.6	0,85	0,53	0,93	0,91
bior6.8	0,90	0,55	0,94	0,93
coif1	0,87	0,49	0,94	0,92
coif10	0,86	0,61	0,94	0,92
coif11	0,86	0,62	0,94	0,92
coif12	0,86	0,62	0,94	0,92
coif13	0,86	0,63	0,94	0,92
coif14	0,86	0,63	0,94	0,92
coif15	0,86	0,64	0,94	0,92
coif16	0,86	0,64	0,94	0,92
coif17	0,89	0,54	0,92	0,91
coif2	0,88	0,52	0,95	0,93
coif3	0,90	0,55	0,95	0,93
coif4	0,90	0,57	0,94	0,92
coif5	0,90	0,59	0,94	0,92
coif6	0,90	0,6	0,94	0,92
coif7	0,90	0,61	0,94	0,92
coif8	0,90	0,63	0,94	0,92
coif9	0,86	0,61	0,94	0,92
db1	0,87	0,43	0,94	0,92
db10	**0,90**	**0,7**	**0,94**	**0,92**
db11	0,89	0,72	0,93	0,92
db12	0,89	0,71	0,93	0,92
db13	0,89	0,73	0,93	0,91
db14	0,89	0,74	0,93	0,91
db15	0,89	0,75	0,93	0,92
db16	0,89	0,75	0,93	0,92
db17	0,89	0,75	0,93	0,92
db18	0,88	0,76	0,93	0,92
db19	0,89	0,77	0,93	0,91
db2	0,87	0,48	0,94	0,91
db20	0,89	0,77	0,93	0,91
db21	0,88	0,77	0,93	0,92
db22	0,88	0,78	0,93	0,91
db23	0,89	0,78	0,93	0,91
db24	0,88	0,78	0,93	0,91
db25	0,85	0,77	0,92	0,92
db26	0,86	0,77	0,92	0,91
db27	0,85	0,77	0,93	0,92
db28	0,85	0,78	0,93	0,92
db29	0,85	0,77	0,93	0,91
db3	0,88	0,52	0,95	0,92
db30	0,85	0,78	0,93	0,92
db31	0,85	0,77	0,93	0,91
db32	0,85	0,78	0,92	0,91

	Gaussian	S&p	Poisson	Speckle
db33	0,85	0,78	0,92	0,91
db34	0,85	0,78	0,93	0,91
db35	0,85	0,77	0,92	0,91
db36	0,85	0,78	0,93	0,91
db37	0,85	0,78	0,93	0,92
db37	0,85	0,78	0,93	0,91
db4	0,89	0,52	0,94	0,92
db5	0,89	0,56	0,94	0,92
db6	0,90	0,61	0,94	0,92
db7	0,89	0,65	0,94	0,92
db8	0,90	0,64	0,93	0,91
db9	0,89	0,67	0,94	0,92
dmey	0,86	0,63	0,94	0,92
haar	0,87	0,43	0,94	0,92
rbio1,1	0,87	0,43	0,94	0,92
rbio1,3	0,88	0,47	0,94	0,92
rbio1,5	0,88	0,49	0,94	0,92
rbio2,2	0,79	0,52	0,89	0,88
rbio2,4	0,84	0,54	0,93	0,91
rbio2,6	0,86	0,56	0,94	0,92
rbio2,8	0,86	0,57	0,93	0,92
rbio3,1	0,56	0,51	0,63	0,64
rbio3,3	0,74	0,53	0,89	0,89
rbio3,5	0,78	0,53	0,92	0,9
rbio3,7	0,80	0,55	0,91	0,9
rbio3,9	0,80	0,56	0,92	0,91
rbio4,5	0,88	0,53	0,94	0,92
rbio5,6	0,87	0,53	0,94	0,92
rbio6,9	0,90	0,56	0,94	0,92
sym10	0,90	0,58	0,94	0,92
sym11	0,89	0,62	0,94	0,92
sym12	0,90	0,59	0,94	0,92
sym13	0,90	0,62	0,94	0,92
sym14	0,90	0,6	0,94	0,93
sym15	0,90	0,65	0,94	0,92
sym16	0,90	0,61	0,94	0,92
sym17	0,90	0,66	0,94	0,92
sym18	0,90	0,62	0,94	0,92
sym19	0,89	0,66	0,94	0,92
sym2	0,87	0,48	0,94	0,91
sym20	0,90	0,63	0,94	0,92
sym3	0,88	0,52	0,95	0,92
sym4	0,89	0,51	0,94	0,92
sym5	0,89	0,51	0,94	0,92
sym6	0,89	0,53	0,95	0,92
sym7	0,89	0,57	0,94	0,92
sym8	0,90	0,56	0,94	0,92
sym9	0,90	0,56	0,94	0,92

3.2.2 Levels

The file “*testing_over_levels.py*” provide the denoising testing of a single image over a single type of noise. In presence of a Gaussian, Poisson or Speckle noise, a higher level of decomposition returns a higher SSIM. However, with the Salt a pepper noise, the best results appear with a lower level of decomposition, since the noise is not contained at low frequencies, so further decomposition will only erase values already present in the original image.

It was decided to establish a default value of 4 levels of decomposition as it turns out to be averagely good among the different types of noise.

3.2.3 Threshold function

Since the thresholding of the values is what characterize the removal of the noise in this type of filtering, two different thresholding functions where provided. The first one employs simply the “*Universal Thresholding*”, while the other employs the “*NeighShrink*” thresholding.

As for this latter, a parameter that characterize the quality of the denoising is the dimension of the neighbourhood.

The file “*testing_over_neighbourhood_dimensions.py*” provides the testing of the denoising of a single image over the different possible dimensions of the neighbourhood to be considered during the thresholding. The tests were conducted starting with a window of size 2x2 until a window of size 10x10 and the results shows that a window of dimension 3x3 gives the best results in most of the cases.

4 Comparing filters

To sum-up, to compare the filters altogether. It was implemented a script called "*compare_filters.py*" which produce the plot of all filtered image and their performance indexes: MSE, PSNR and SSIM.

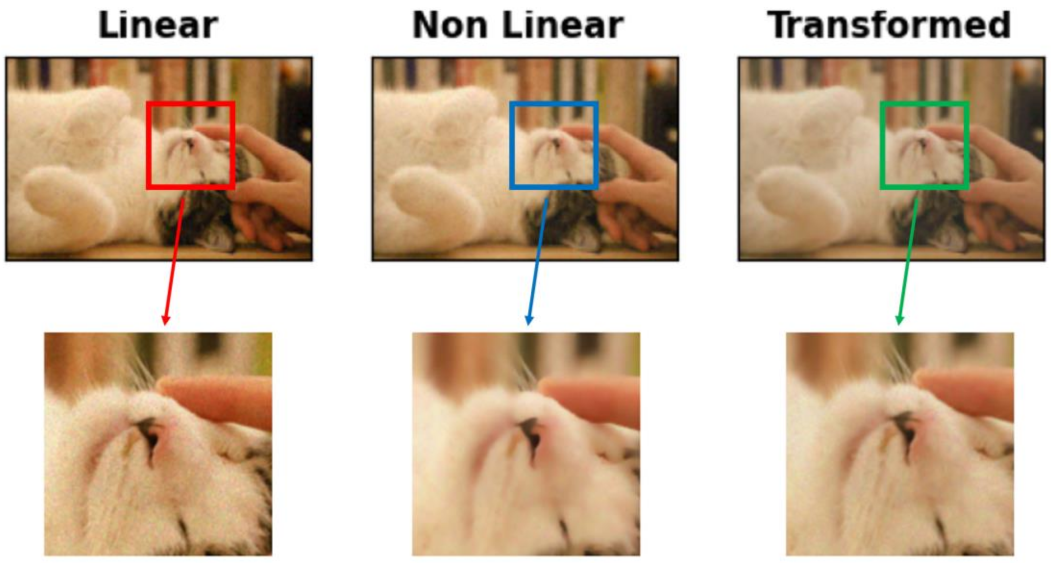
We have used both b&w and RGB image and to follow will be show some plots:

- As for the Gaussian noise, we see that the Non-Linear filter obtains the best performances for both the b&w and RGB image (*Figure 1* and *Figure 2*), followed by the transformed domain filter;
- As for the Salt and Pepper noise, the *Figure 3* and *Figure 4* show that the transformed domain filter achieves the best results in terms of both PSNR and SSIM;
- As for the Poisson noise, the *Figure 5* shows that the transformed domain filter obtains the best result for the b&w image while the Non-Linear filter obtains the best result for the RGB image (*Figure 6*);
- As for the Speckle noise, the Non-Linear filter achieves the best results for both b&w and RGB images (*Figure 7* and *Figure 8*).

The results reported are merely related to the particular distribution of the noise on the image and do not intend to establish the absolute level of quality of the filters on that particular noise.

However, they show a common tendency of quality: the Non-Linear filter give good results, close enough to each other, for Gaussian, Poisson and Speckle noise. Furthermore, among the Non-Linear and the linear filter, the Non-Linear one achieves better results, graphically too, due to the more complex structure of the algorithm, which gives more degrees of freedom too. However, this costs in a major computational complexity.

Finally, as for the salt and pepper noise is concerned, the transformed domain filter obtains better results since the greatest part of noise is present in the high frequency, given the abrupt changes between values, and the transformed domain filter better eliminate high frequency noise.



cat.jpg with Salt and Pepper noise

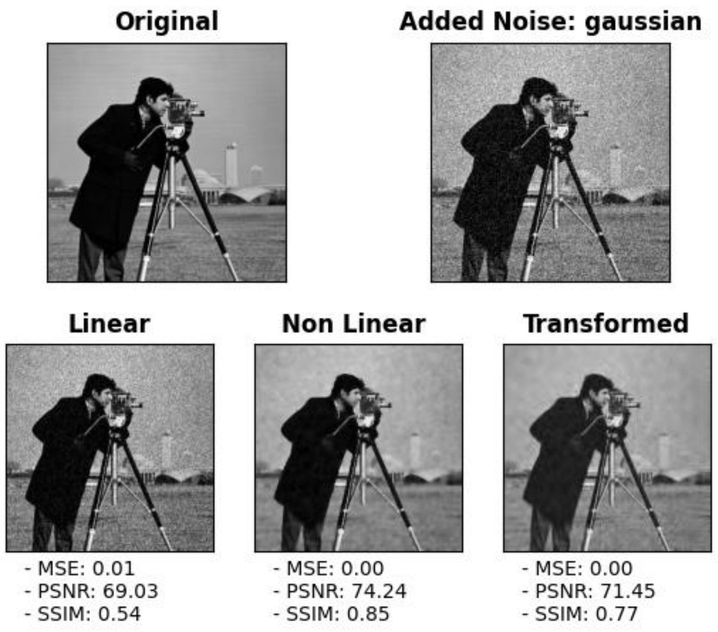


Figure 1: cameramen.tif with Gaussian noise

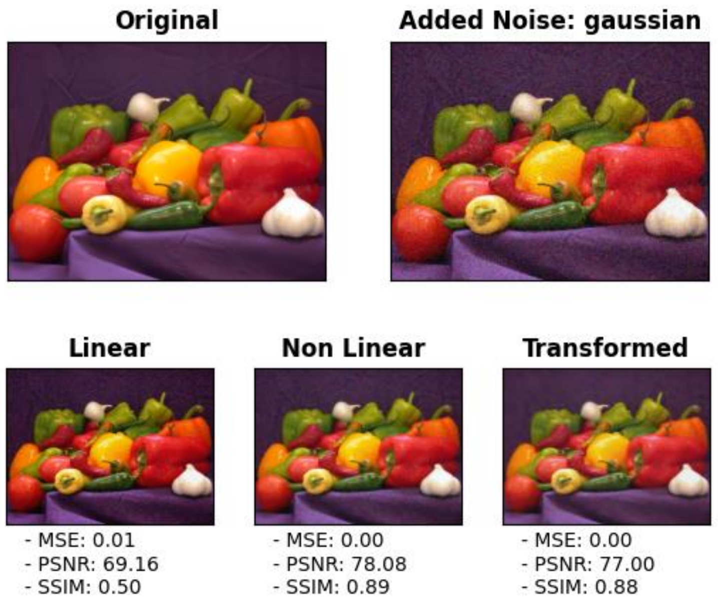


Figure 2: pepper.png with Gaussian noise

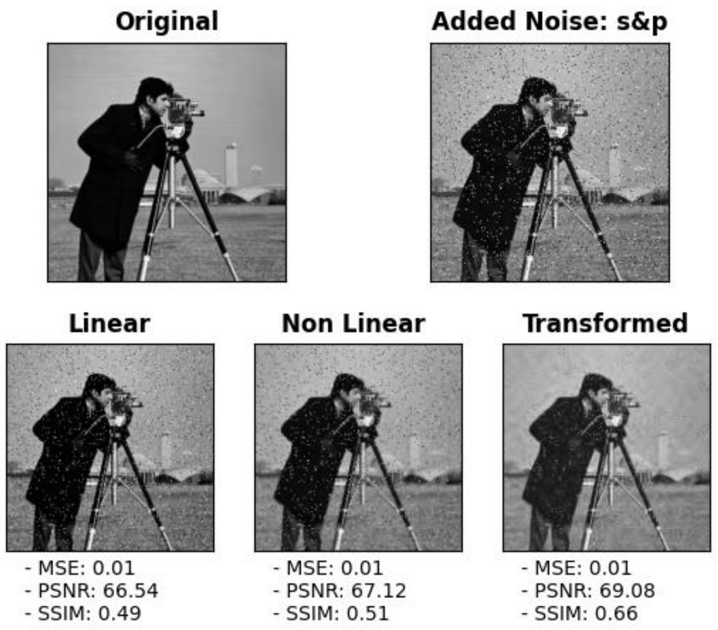


Figure 3: cameramen.tif with Salt and Pepper noise

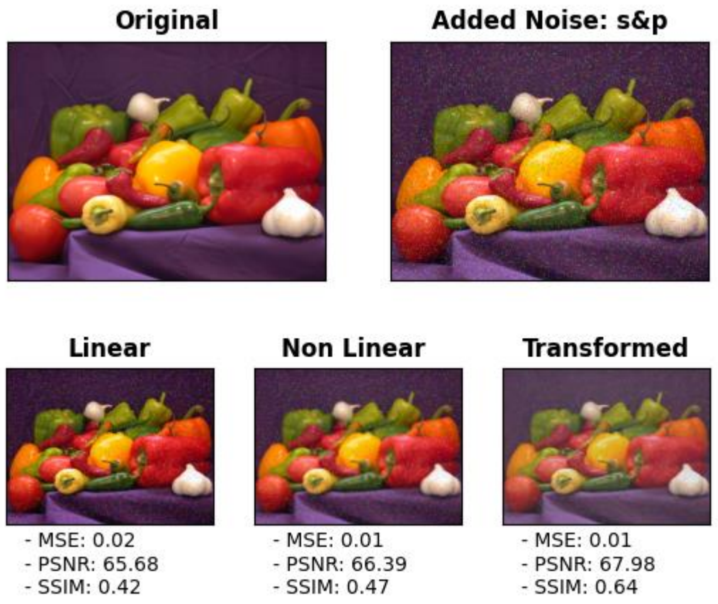


Figure 4: pepper.png with Salt and Pepper noise

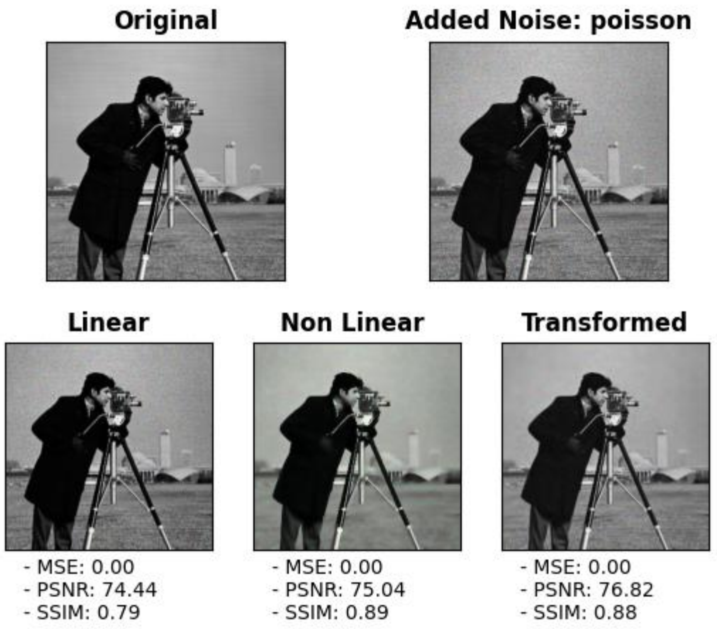


Figure 5: cameramen.tif with Poisson noise

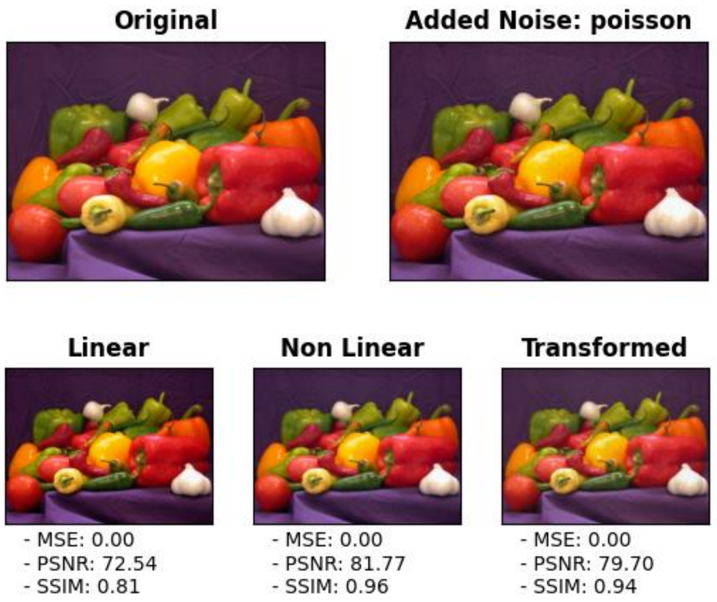


Figure 6: pepper.png with Poisson noise

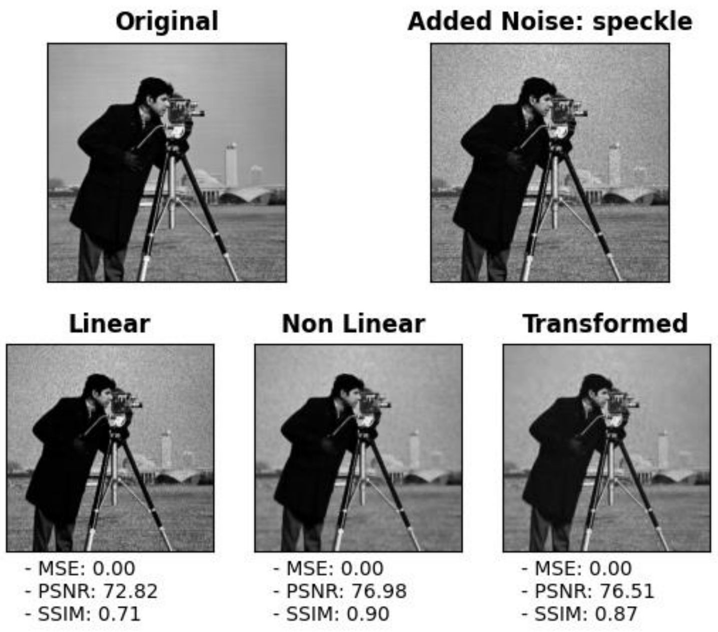


Figure 7: cameramen.tif with Speckle noise

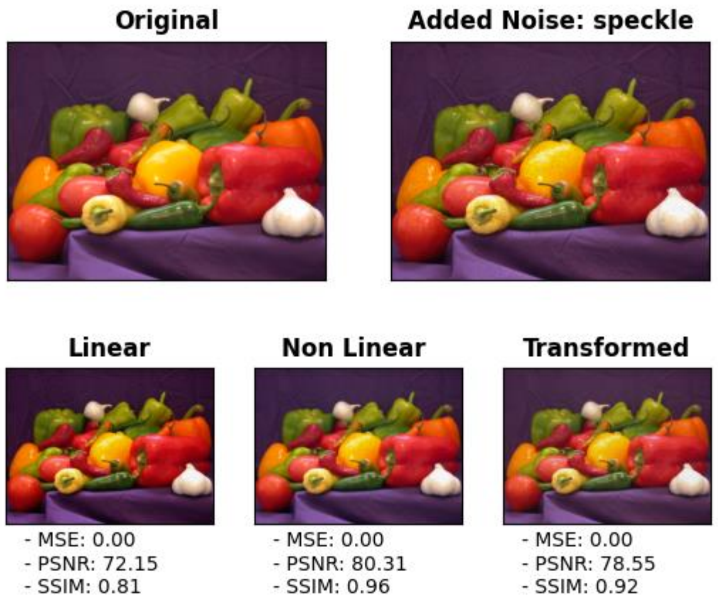


Figure 8: pepper.png with Speckle noise