UNIVERSIDAD REY JUAN CARLOS

Escuela de Ingeniería de Fuenlabrada

Biomedical Engineering

Course 2023-2024

Lab I: Advanced Medical Image Pre-Processing Methods

MEDICAL IMAGE ANALYSIS

Group 11:

Enrique Almazán Sánchez

Guillermo Ots

Javier Alfonso Villoldo Fernández

OUTLINE

1. Objectives

We aim to test and evaluate different advanced image pre-processing algorithms that have been covered in class, the **Non-Local Means Filtering** and the **Anisotropic Filtering**. They will be explained in their respective sections.

In addition, we will explore and investigate the state-of-the-art in the sector of image filtering methods, thus, increasing our understanding of the available techniques and their applicability to medical image processing.

Each member of the group will work on implementing an advanced filter under one condition: the filter is not part of the regular subject curriculum. We think this is a good approach as each of us will gain knowledge from this research endeavor, as well as from three different new advanced filtering methods instead of only one. The chosen filters are:

- **Bilateral Filter:** implemented by Enrique Almazán
- **TV Chambolle Filter:** implemented by Guillermo Ots
- **Wiener Filter:** implemented by Javier Villoldo

We will apply these algorithms to real clinical images to assess their effectiveness, evaluating the results and assessing how well the algorithms achieve their intended goals, which are noise reduction (denoising the images), and so feature enhancement and artifact removing.

Also, the results obtained will not only be compared to the original images, but also with the results obtained from Smoothing Filters, such as Mean, Median and Gaussian Filters. The required dependencies or libraries for this project are the following ones:

- **NumPy (Numerical Python):** Fundamental library for numerical computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a wide range of mathematical functions to operate on these arrays efficiently.
- **Random:** Standard library used for generating random numbers and performing random operations.
- **Matplotlib's Pyplot Module:** Powerful library for creating static, animated, or interactive visualizations in Python. The pyplot module provides a simple and MATLAB-like interface for creating plots and charts.
- **OpenCV (cv2):** OpenCV (Open-Source Computer Vision Library) is a library dedicated to computer vision and image processing tasks. It offers a wide range of functions for image and video analysis, including image filtering, object detection, and more.
- **scikit-image (skimage):** Image processing library built on top of SciPy and NumPy. It provides tools and algorithms for a wide range of image processing tasks, making it useful for tasks such as feature extraction and image enhancement.
- **SciPy:** Scientific library that builds on NumPy and provides additional functionality for scientific and technical computing. It includes tools for optimization, signal processing, statistical analysis, and more.

2. Contributions

The code regarding the Non-Local Means (NLM) Filter was done by the whole group during the laboratory class. Then, the project was divided as follows.

- Enrique Almazán Sánchez:
  - Explanations regarding NLM filter.
  - Code and explanations regarding Anisotropic diffusion.
  - Code and explanations regarding Bilateral Filter.

- Guillermo Ots:
  - Code and explanations regarding TV Chambolle Filter.

- Javier Alfonso Villoldo Fernández:
  - Code and explanations regarding Anisotropic diffusion.
  - Code and explanations regarding Wiener Filter.

This will add to the posterior report of the project, in which each member will have the same work load, explaining each of its parts, as well as a final comparison between all the filters implemented.

3. Previous issues and Noise Addition

The medical images that are going to be used for testing our implementation of the different advanced filters (named in the 'Objectives' section) are going to be free of noise. Thus, it is necessary to add noise to them in order to later evaluate the results obtained with each filter.

We have created a class, ***'ImageNoiseGenerator'***, designed to encapsulate various image noise generation functions. It provides a convenient way to add different types of noise to input images. The class is instantiated without any specific initialization parameters and have different static methods representing each of the noises we want to study.

It is worth mentioning how the parameters, that will be described below for the generation of different noise types, are those corresponding to the library function and not the ones that are stated in our function, since we have decided to leave some parameters as default and other as freely adjustable.

Thus, the rule to follow throughout the entire report will be that those parameter values that appear in italics as well as in quotes will be the default parameters for the implemented function and the ones used, while the parameter values that appear in bold will be the ones left to adjust the noise adding method or filter function.

a) **Gaussian Noise**: gaussian noise is a type of statistical noise that follows a Gaussian or normal distribution. It is characterized by random variations in pixel values, where most of the variations cluster around the mean value, resulting in a bell-shaped curve.

- **Effects:** it adds subtle variations to an image, simulating the kind of noise often seen in real-world scenarios. It appears as a fine grain or fuzziness, and it can affect both brightness and color.
- **Applications:** Gaussian noise is commonly used for simulating natural variations in data and testing the robustness of image processing algorithms.

def gaussian_noise (image, std_dev):		
random_noise()	image	Input image to which Gaussian noise will be added.
	mode = *'gaussian'*	Gaussian-distributed additive noise.
	clip = *'True'*	Maintain the proper image data range
	var = **std_dev****2	Variance of random distribution
Output: noisy image with added Gaussian Noise		

b) **Salt and Pepper (Impulsive) Noise**: Salt and Pepper noise, introduces random white and black pixels into an image. It mimics sudden, sporadic intensity spikes (salt) and drops (pepper) in pixel values.

- **Effects**: This noise can lead to the appearance of isolated bright and dark spots in an image, resembling grains of salt and pepper. It is often used to simulate sensor defects or random errors in image acquisition.
- **Applications:** is used for testing the robustness of image processing algorithms, particularly those designed for artifact removal.

def salt_and_pepper_noise (image, salt_prob, pepper_prob):

random_noise()	image	Input image to which Salt and Pepper noise will be added.
	mode = *'s&p'*	Replaces random pixels with either 1 or 0.
	amount = **salt_prob** + **pepper_prob**	Proportion of image pixels to replace with noise on range [0, 1]
	salt_vs_pepper = salt_prob / (salt_prob + pepper_prob)	Proportion of salt vs. pepper noise on range [0, 1]
	clip = *'True'*	Maintain the proper image data range
Output: noisy image with added Salt and Pepper noise		

c) **White Noise**: White noise is a type of random signal that contains equal intensity across all frequencies. It is characterized by randomness and unpredictability in signal values, making it appear as a continuous stream of random data.

- **Effects:** In images, white noise manifests as a completely random and grainy pattern, with no discernible structure or repeating elements. It adds a high level of unpredictability to data.
- **Applications:** is often used in signal processing, and image processing for purposes such as masking sensitive information, encryption, or generating random data.

def white_noise (image, noise_amplitude):		
img_as_float() It converts an image, from integer pixel values, into an array with floating-point values.	image	Input image to which White Noise will be added.
np.random.uniform() Samples are uniformly distributed over the half-open interval [low, high), such that any value within the given interval is equally likely to be drawn by.	low = **-noise_amplitude**	Lower boundary of the output interval. All values generated will be greater than or equal to low
	high = **noise_amplitude**	Upper boundary of the output interval. All values generated will be less than or equal to high.
	size = image.shape	Output shape of the noisy image, we maintain original size
np.clip() It limits the image array range within a minimum and maximum value	noisy_image	Output image with White Noise
	a_min = *'0'*	Lower limit of the range into which the values of the noisy image fall

	a_max = *'1'*	Upper limit of the range into which the values of the noisy image fall
Output: noisy image with added White Noise		

d) **Poisson Noise**: Poisson noise arises from the stochastic nature of photon arrivals when capturing images or data. Poisson noise results in variations in pixel intensities, with higher variations in low-intensity regions and lower variations in high-intensity regions.

- **Effects:** it appears as random variations in pixel values, often resembling a "grainy" or "speckled" pattern. It tends to be more pronounced in images with low light levels or low exposure times. Poisson noise can impact image quality by reducing image clarity and making it challenging to discern fine details.
- **Applications:** it is particularly relevant in applications involving low-light imaging, such as fluorescence microscopy, and medical imaging (e.g., positron emission tomography or PET scans).

def poisson_noise(image):		
random_noise()	image	Input image to which Poisson Noise will be added.
	mode = *'poisson'*	Poisson-distributed noise generated from the data.
	clip = *'True'*	Maintain the proper image data range
Output: Noisy image with added Poisson Noise		

e) **Quantization Noise**: Quantization noise occurs when continuous signal values are approximated or rounded to discrete levels during digitization or quantization. It arises due to the finite precision of digital representation.

- **Effects:** it appears as discrepancies between the original continuous signal and the digitized version. It can result in artifacts such as banding or contouring, especially when the quantization levels are coarse.
- **Applications:** Quantization noise is relevant in digital imaging, audio processing, and data compression. It is a critical consideration when selecting the bit depth or precision for digitizing analog signals.

def quantization_noise(image, num_levels):		
np.floor() It rounds down the elements of an array to the nearest integer that is less than or equal to the original value.	image	Input image to which Quantization Noise will be added.
	num_levels	Number of quantization levels. A higher value reduces the quantization noise (it means less discrete values).
np.clip()	noisy_image	Output image with Quantization Noise
	a_min = *'0'*	Lower limit of the range into which the values of the noisy image fall

It limits the image array range within a minimum and maximum value	a_max = *'1'*	Upper limit of the range into which the values of the noisy image fall
Output: noisy image with added Quantization Noise		

4. Implementing Filters

As in the previous sections we have created two different classes to implement the advanced filters (the ones we want to study) and the smooth filters (the ones used for benchmarking and comparison) in an organized and efficient way:

- ***'AdvancedImageFilter'*** class is designed to provide a collection of advanced image denoising and filtering methods. Users can create an instance of this class to access and apply these filters to noisy images. The following libraries are used for the implementation of five different advance filters, two of them being the Non-Local Means Filter and Anisotropic Filter seen in class:

  - **Non-Local Means Filter:** skimage.restoration.denoise_nl_means
  - **Anisotropic Filter:** medpy.filter.smoothing.anisotropic_diffussion
  - **Bilateral Filter:** skimage.restoration.denoise_bilateral
  - **TV Chambolle Filter:** skimag.restoration.denoise_tv_chambolle
  - **Wiener Filter:** skimage.restoration.wiener

  All Advanced Filters will be explained in detail in their respective sections.

- The '***SmoothImageFilter'*** class is designed to provide a collection of image-smoothing filters, including Mean Filter, Median Filter, and Gaussian Filter. It encapsulates the implementation of these filters, allowing users to apply them to images for the purpose of reducing noise or smoothening.

  - **Mean Filter**: is a simple and computationally efficient image processing technique used for spatial smoothing or blurring. It replaces each pixel in the image with the average value of the pixels that the mask covers, which is defined by a rectangular or square kernel. It is particularly good for removing high-frequency noise.

    This filter works in the following way: It uses a kernel (a small window with N number of pixels) that slides over the image. For each pixel in the image, the filter calculates the average of pixel values within the kernel. The pixel value at the center of the kernel is then replaced with this average value. This process is then repeated for every pixel. In the resulting image the noise has been eliminated by smoothing of the image. In turn, spatial resolution decreases and the edges become blurred.

def mean_filter(image, size_filter=5):			
mean_filter	np.ones() Creates a new array of given shape and type, filled with ones.	(size_filter, size_filter)	Shape of the new array (5x5)

	np.power() Raise a value to a given power	size_filter	Size of the mean kernel
		exponent = 2	Power of the function
ndimage.convolve() Performs convolution operations on multi-dimensional arrays, such as images	image	Input image with noise to which mean filtering will be implemented	
	mean_filter	Array of weights corresponding to the mean kernel	
	mode = *'reflect'*	Determines how the input array is extended when the filter overlaps a border. The 'reflect' value means that the input is extended by a padding that reflects about the edge of the last pixel.	
Output: smooth image after Mean denoising			

- **Median Filter**: non-linear image processing technique used for noise reduction and preserving edge details. Unlike the Mean Filter, it replaces each pixel in the image with the median value of its neighborhood and edges are preserved. Therefore, new gray values are not created but instead obtained from the sub-image itself. This filter is particularly useful for removing salt & pepper noise

  The median filter works in the following way: it employs a kernel that slides over the image, similar to the Mean Filter. For each pixel in the image, the filter calculates the median of pixel values within the kernel. The pixel value at the center of the kernel is then replaced with this median value. This process is repeated for every pixel in the image.

def median_filter(image, size=5):		
ndimage.median_filter()	image	Input image with noise to which median filtering will be implemented
	size	The size of the median filter mask
	mode = *'reflect'*	Determines how the input array is extended when the filter overlaps a border. The 'reflect' value means that the input is extended by a padding that reflects about the edge of the last pixel.
Output: smooth image after Median denoising		

- **Gaussian Filter**: linear smoothing filter used to reduce noise and create a blur effect in images. It is based on a Gaussian function, which is a bell-shaped curve. It effectively

reduces Gaussian noise and provides a controlled amount of smoothing. It allows for adjusting the amount of blur by changing the standard deviation of the Gaussian distribution. It essentially works as a mean filter, only in this case a weighted mean kernel is applied over the pixels in the sub-image, The weight of each neighboring pixel is determined by its proximity to the center pixel and follows a Gaussian distribution.

<table>
<tr><td colspan="3">def gaussian_filter (image, sigma=1) :</td></tr>
<tr><td rowspan="3">ndimage.gaussian_filter()</td><td>image</td><td>Input image with noise to which gaussian filtering will be implemented</td></tr>
<tr><td>sigma</td><td>Standard deviation for Gaussian kernel, in this case, equal for both axis since we have a single value.</td></tr>
<tr><td>mode = ‘reflect’</td><td>Determines how the input array is extended when the filter overlaps a border. The ‘reflect’ value means that the input is extended by a padding that reflects about the edge of the last pixel.</td></tr>
<tr><td colspan="3">Output: smooth image after Gaussian denoising</td></tr>
</table>

5. Advanced Filtering Techniques

The same structure will be followed for the proper description of the filters and analysis of their respective results. For each filter we will first provide an explanation of the procedures behind their functioning, then we will include a table showcasing the python functions and parameters used as well as their meaning. Finally for each type of noise filtering technique we will present observations for the resulting images and their comparisons with smoothing filters and other advanced filters.

4.1. Non-Local Means Filtering

The **Non-Local Means (NLM) Filter** is an image denoising technique used to reduce noise in digital images. It operates on the principle that similar patterns or structures in an image should have similar pixel values. Instead of just considering local neighborhoods, the NLM filter looks for similarities across the entire image. While in the mean filter we considered that differences between pixels in a mask were solely due to noise and neglected that this assumption was not met at the edges, resulting in homogenization of the entire image with non-local methods we consider information from the complete image instead. The NLM filters is particularly effective at reducing various types of noise, such as Gaussian noise. It preserves image details and structures, making it suitable for denoising while maintaining image sharpness.

The NLM filter works in the following way: for each pixel in the image, the NLM filter searches for patches of pixels in the image that are similar to the patch centered around the pixel of interest. This similarity is measured using a distance metric, such as Euclidean distance, and a kernel function that weighs the contributions of similar patches. The NLM filter then computes a weighted average of the pixel values within these similar patches, with the weights determined by the kernel function. This weighted average is used to replace the pixel value at the center of the patch. The process is repeated for all pixels in the image, resulting in a denoised image.

<table>
<tr><td colspan="4">def non_local_means(image, patch_size=5, patch_distance=6, h=0.1, fast_mode=False, preserve_range=False):</td></tr>
<tr><td rowspan="5">denoise_nl_means ()</td><td colspan="2">image</td><td>Input image with noise to which NLM filtering will be implemented.</td></tr>
<tr><td rowspan="4">kw = dict()</td><td>patch_size</td><td>Size of patches used for denoising. Smaller patch sizes mean less similarity comparisons will be made, thus it might work better for fine-grained noise, while larger patch sizes mean more comparisons between patches, which may be suitable for smoother types of noise.</td></tr>
<tr><td>patch_distance</td><td>The maximal distance in pixels where the algorithm searches for patches used for denoising.</td></tr>
<tr><td>h</td><td>Cut off distance in grey levels that defines the aggressiveness of the filter. (Detailed explanation below).</td></tr>
<tr><td>fast_mode[1] = ‘False’</td><td>Boolean parameter. If True, the fast version of the NLM filter is used. It</td></tr>
</table>

			provides satisfactory results while reducing computational time in some cases.
		sigma[2] = *sigma_est*	The standard deviation of the (Gaussian) noise. If provided, a more robust computation of patch weights is computed that takes the expected noise variance into account.
		preserve_range[3] = *'False'*	Boolean parameter. If True, it preserves the intensity range of the input image. which maintains the original image's contrast. The choice of this parameter might depend on the specific application.
Output: denoised image after NLM filtering			

The NLM strictly dependent on the patch size and patch distance.

The **'*h*' parameter** is the essential regulator of the NLM filter, then it requires further explaining. Since the '*h*' parameter defines a cut-off distance for which the NLM filter considers patches in the image to be similar to the reference patch, it is understood as a proximity index. This means that the parameter influences the range of similarity between patches in the image.

If the distance between patches exceeds '*h*' they are considered dissimilar and contribute less to the filtering process. In other words, a larger '*h*' value makes the filter more aggressive, since the more permissive one is in accepting patches, causing more denoising (smoother image) at the expense of more blurring.

For example, in the case of Gaussian noise, a good rule of thumb is to choose the 'h' value to be sigma, thus we will use multiples of sigma as 'h' values depending on the noise, reason why we use:

$$h = h \cdot sigma\_est$$

Take into account, that the h's are not the same. The one in the left part of the equation is the parameter for the function '*denoise_nl_means()*', while the one in the right part is the one given by the user.

1: The fast mode parameter will be set to 'False' in order to apply the original version of the NLM which uses a more accurate but computationally demanding algorithm for denoising. In fact, there have been seen significant downsides when using the 'True' value, ergo, using the fast mode, reason why the previous value has been chosen.

2: The sigma parameter which refers to the standard deviation of the noise to deal with will be 'sigma_est'. This variable estimates the noise standard deviation from the noisy image, computed as follows:

$$np.mean(estima\_sigma(noisy\_image))$$

3: Problems in the form of warnings and errors have been encountered regarding the use of the 'preserve_range' parameter. Thus, as it is thought to be a problem related to the version of the notebook, it has been decided to use the default parameter, 'False', of the denoise_nl_means() function.

4.1.1. Implementation with Gaussian Noise

Implementing the NLM filter to denoise an image with added gaussian noise, using the parameters:

Parameter Name	Parameter Value	Comment
patch_size	5	5x5 patch size
patch_distance	6	13x13 search area
h	**[0.8-1]**	Cut-off distance that defines the filter aggressiveness

For these parameters the following denoised image (in comparison with the noisy image and the original one) is obtained:

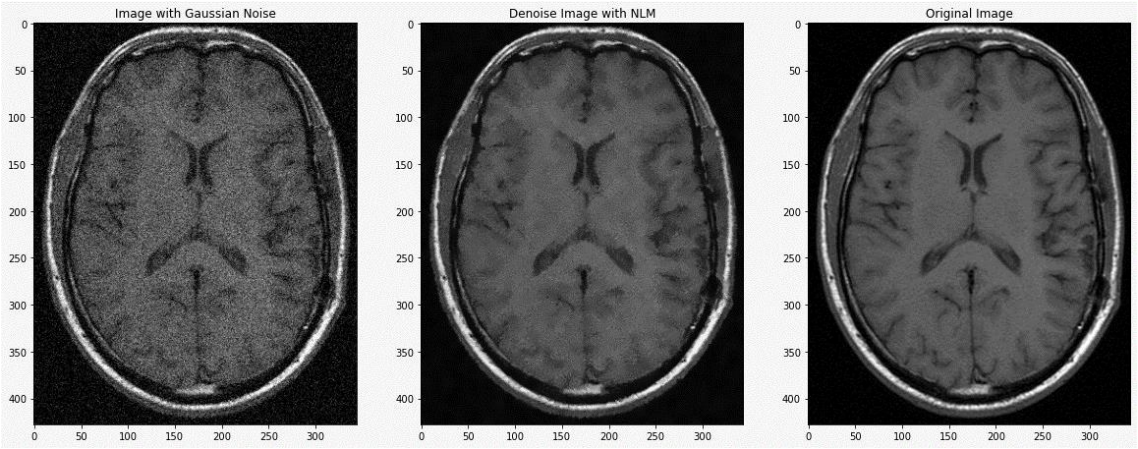

Figure 1: Comparison between noisy image, denoised image and original image for Gaussian noise.

For **patch size** and **patch distance**, values of 5 and 6 have been chosen respectively, as they are a balanced decision. It considers a relatively small local patch and distance to analyze similarities between patches. This means that, as seen in *Figure 1*, the denoising was robust while, at the same time, preserves important image features which in this case are edges.

For better explanation, regarding the patch size, having lower values means having a smaller kernel, thus, less comparison between the pixels of the patches preserving better the details of the image rather than denoising, while higher values mean having a greater kernel, thus, more comparisons between the pixels of the patches denoising better the image rather than preserving fine detail.

For the patch distance something similar happens. With higher values, a higher distance is established for looking for patches, thus having more comparisons making the filter better in conserve the important features of the image than in noise removal, while with lower values, a lower distance is established for looking for patches, thus having less comparisons making the filter better in noise cancelling than in maintaining the important features.

Finally, in relation to the most important parameter of the NLM function, the ***'h'*** value, a range has been computed (see below *Figure 2*). The lower value, 0.8, is the minimum value at which both denoising and edges preserving take place, while the upper value, 1, is the maximum at which this is still accomplished. Thus, the optimal value will be 0.9 but the efficacy of the filter will not be affected if any other value inside of the defined range is chosen.

However, for values outside the range negative consequences has been appreciated as those seen in the patch parameters. When the value of '*h*' is below the minimum of the range it preserves more details at the expense of less denoising because the algorithm is less aggressive as opposed to when the value is above the maximum, in which case the filter will be more aggressive focusing more on denoising, smoother to much image rather than detail preservation.

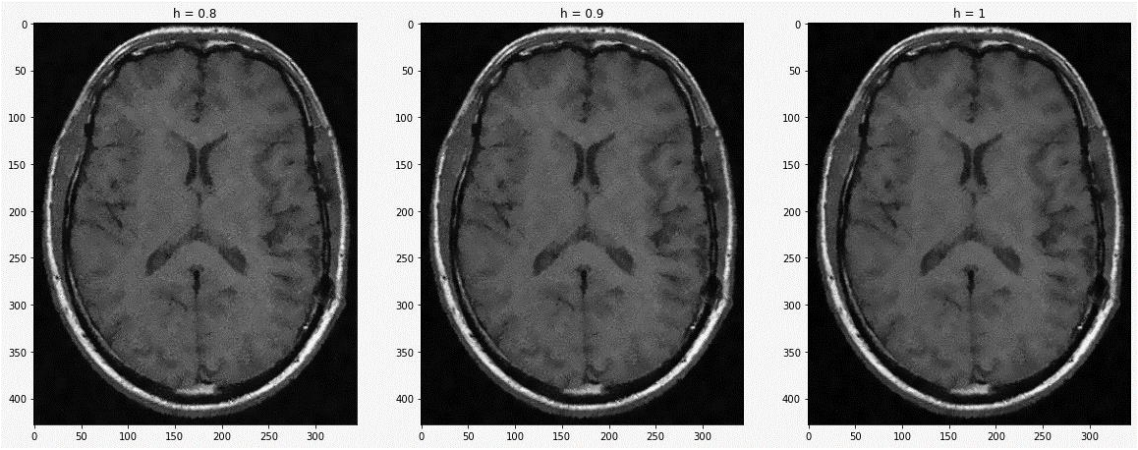

Figure 2: Benchmarking for h parameter, values from 0.8-1 for dealing with Gaussian Noise

To complete the implementation of the NLM Filter for Gaussian Noise, the results obtained from it have been compared with the ones obtained from the smooth filters.

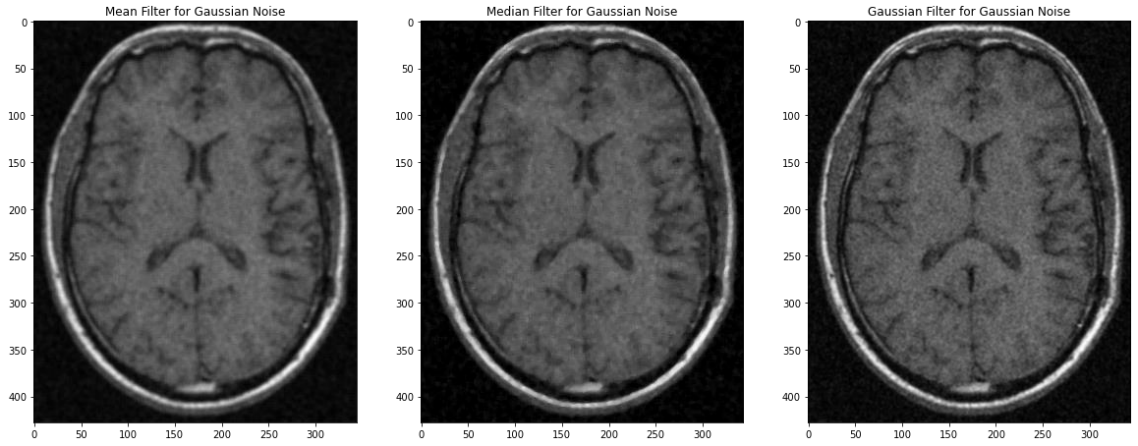

Figure 3: Smoothing filters (Mean, Median and Gaussian) for Gaussian Noise.

Comparing the smoothing filters results viewed in *Figure 3* and those of the NLM (presented in both, *Figure 1* for '$h = 0.9$' and in *Figure 2* for the range of values for the parameter *'h'*), it is obvious that the advanced filter consistently outperforms the traditional smoothing filters. The mean and the median filter return a very smooth and blurry image resulting in the loss of edges, while the Gaussian filter even though it preserves better the fine details it falls short in terms of noise reduction.

Thus, the NLM Filter exhibit improved denoising on par with preservation of, in this case, the edges of the image. This can be explained taking into account the characteristics of both. The NLM applies an adaptive filtering as well as a broader search space when selecting and averaging similar patches from the image while the smoothing filters apply uniform blurring to the image.

4.1.2. Implementation with Salt & Pepper Noise

Implementing the NLM filter to denoise an image with added salt & pepper, using the parameters:

Parameter Name	Parameter Value	Comment
patch_size	3	3x3 patch size
patch_distance	4	9x9 search area
h	**[20, 40]**	cut-off distance that defines the filter aggressiveness

For these parameters the following denoised image (in comparison with the noisy image and the original one) is obtained:

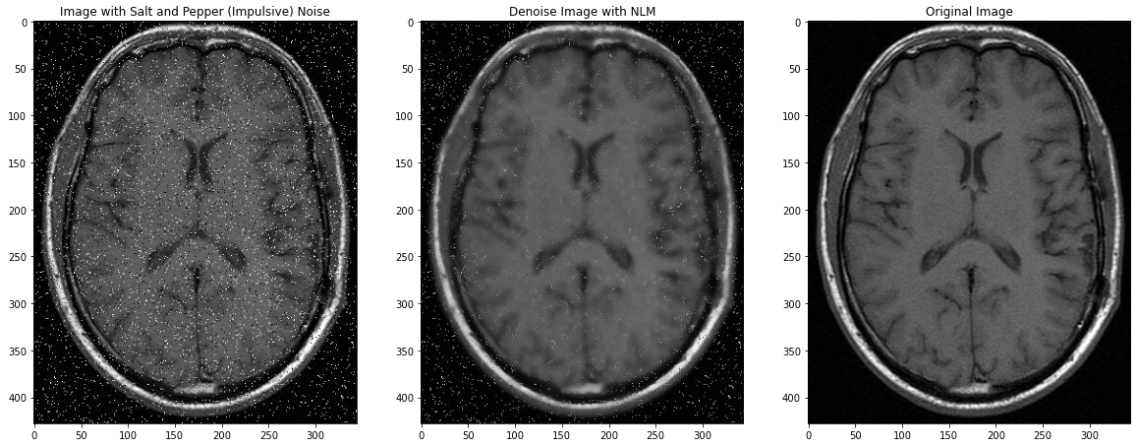

Figure 4: Comparison between noisy image, denoised image and original image for Salt and Pepper (Impulsive) noise.

When dealing with Salt and Pepper (Impulsive) noise, the values for **patch size and patch distance** have been decreased, being the values chosen 3 and 4 respectively, continuing to be a balanced decission. This decrease is due to the intrinsic characteristics of both noises. The Gaussian Noise is defined by a continuous probability distribution having a moderate impact on the similarity between the patches in the image, while the Salt and Pepper (Impulsive) Noise is defined by a random introducation of white (salt) or black (pepper) pixels, having a more pronounced and localized impact.

Thus, for the former noise a larger patch size and patch distance was chosen, capturing a broader range of image and making more comparisons among the pixels of the patches, allowing the NLM filter to consider more significant spatial information for denoising. On the other hand, for the latter noise, a smaller patch size and patch distance was chosen, focusing on the neighboring pixels, which provides better local adaptation. Recalling the previous explanation of what happens when increasing or decreasing each of this parameters, patch size and patch distance, this change when facing each of the noises (Gaussian and Impulsive) makes sense.

Regarding the **parameter *'h'***, it is still the same as before, being a proximity index, defining the aggressiveness of the algorithm. However, it can be seen that the computed range is much greater than for the previous noise. This is due to several aspects, going from a higher standard deviation in the Gaussian (around 0.084) than in the Salt and Pepper Noise (around 0.015) as well as greater values in the patch size and patch distance for the former noise.

Concerning the standard deviation and how the 'h' parameter is computed in the algorithm implemented for the NLM, apart from defining the spread or intensity of the noise, it also regulates the 'h' given by the user. Thus, for the Salt and Pepper Noise a higher 'h' is needed in order to denoise the image.

However, a question arises. Why do the Salt and Pepper Noise will need more aggressiveness with respect the 'h' parameter of the filter than the Gaussian Noise, if it is less spread or intense with respect to the standard deviation? Here is where the patch size and patch distance appear. Recalling that these values were higher in the Gaussian than in the Salt and Pepper noise, for analyzing the localities when dealing with the latter noise, then, to overcome the decrease of denoising regarding these parameters the 'h' needs to be greater, for noise removal.

Thus, when facing Salt and Pepper for analyzing the localities the aggressiveness of the filter should be higher than when facing Gaussian Noise.

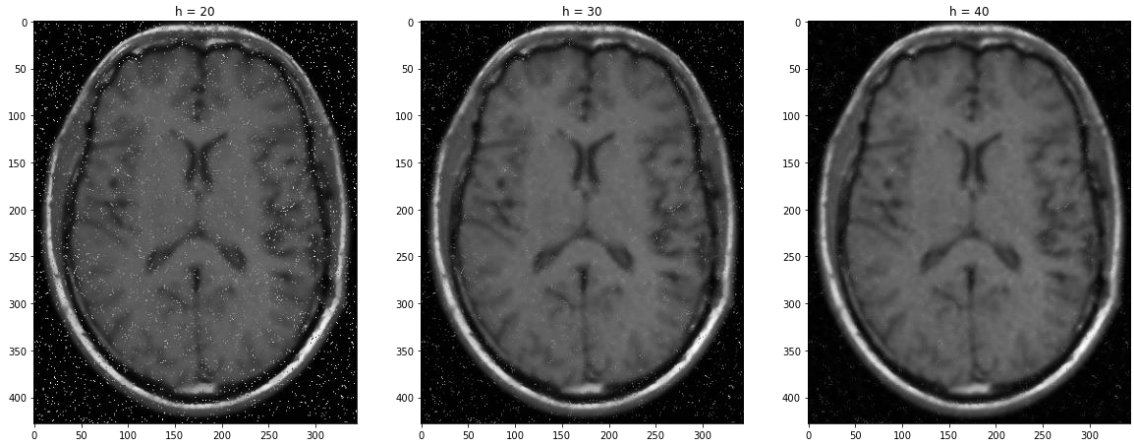

Figure 5: Benchmarking for h parameter, values from 20-40, for Salt and Pepper Noise.

Nonetheless, in this case the range shows different aspects of the parameter 'h'. The minimum value,$h = 20$, shows the case where the NLM Filter focuses to maintain fine details of the image at the expense of removing noise, while the maximum value, $h = 40$, shows the case where it focuses on denoising, but blurring and smoothing to much the image.

Thus, the more efficient value, will be $h = 15$, where the NLM filter tries to compensate both, the amount of blurring and smoothing and the amount of noise removal, even though, there can be an appreciation of loss in image details in *Figure 5*, with respect the one in *Figure 2* when dealing with Gaussian Noise.

To complete the implementation of the NLM Filter for Salt and Pepper (Impulsive) Noise, the results obtained from it have been compared with the ones obtained from the smooth filters.

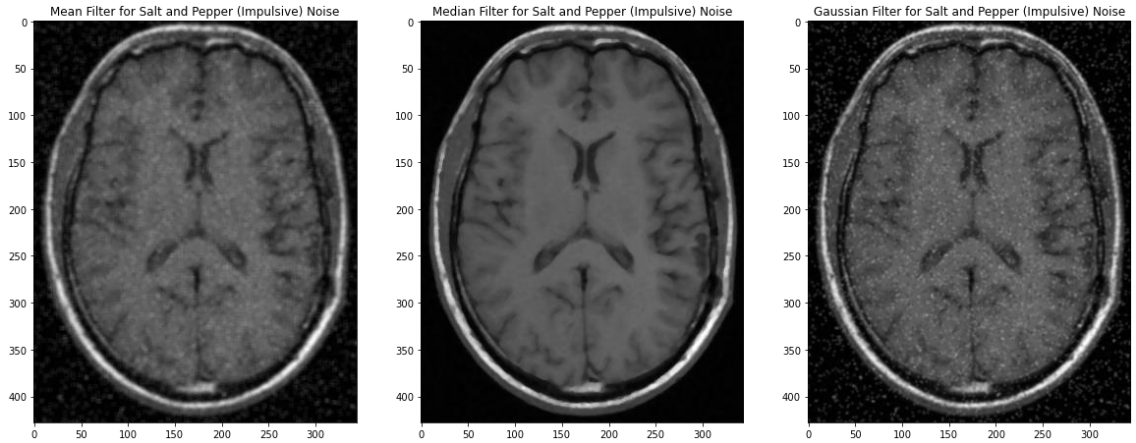

Figure 6: Smoothing filters (Mean, Median and Gaussian) for Salt and Pepper (Impulsive) Noise.

Among the smoothing filters, gaussian and mean filters, provide a similar mediocre output with the gaussian preserving a bit better the edges and giving a less blurred image (this makes sense according to the theory since the gaussian filter is a weighted mean).

The median filter, on the other side, is clearly the most robust against salt and pepper noise, being effective at removing isolated noise pixels. According to theory this also makes sense because salt and pepper noise manifests as isolated noise pixels that are significantly different from their surroundings being considered as outliers. Hence, as the approach of this filter is ideal for removing outliers, is effective at removing these isolated noise pixels by replacing them with a median value from the surrounding.

Even the result given by the NLM filter is not as good as the provided by the median filter. This is due to the fact that the approach followed by the NLM focuses on comparisons between patches giving weights according to the similarities among pixels, lacking the ability of the median filter of removing outliers.

4.1.3. Implementation with Quantization Noise

In this section, the implementation of the NLM filter to denoise an image with added quantization noise, no parameter is presented. This is due to the results shown in *Figure 10*, where it can be seen that the filter does not have any effect on the contaminated image. The variation among the parameters presents no improvements whatsoever, becoming blurrier as the parameters increase without any noise removal.

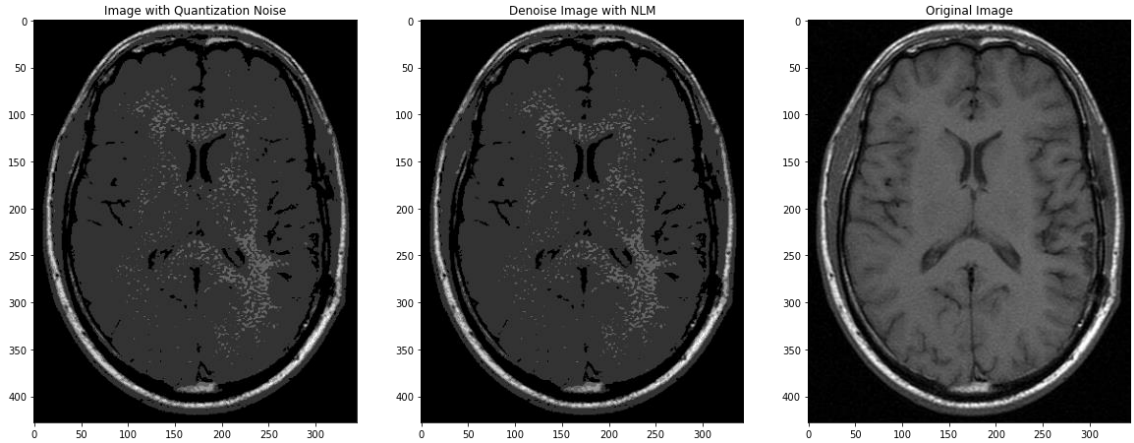

Figure 7: : Comparison between noisy image, denoised image and original image for Quantization noise.

Quantization noise, unlike the previous ones presented, such as Gaussian or Salt and Pepper, is caused by the discretization of pixel values, being challenging to remove using traditional filters, including the one used now, the NLM filter.

Basically, this type of noise results from the limited values used for representing pixel intensities. Thus, when an image is quantized, the number of available gray levels in an image are reduced. As a result, the pixel intensities tend to cluster around these reduced levels, creating a histogram with peaks corresponding to them, meaning that many patches in the image contain similar pixel values, due to the confinement to the quantization levels.

When applying the NLM filter, which relies on patch similarities, the clustering previously mentioned that appears in the histogram in the form of peaks, makes the filter struggle to differentiate between image features and noise structures. In other words, every neighboring patch will contribute almost equally to the NLM filter, as the filter will consider similar all of them compared to the reference patch (centered around the pixel being denoised), being all the weights assigned to each of the pixels of the patches nearly the same.

When the weights are nearly uniform, the filter becomes less discriminative in identifying noise with respect image details, not removing noise effectively. Hence, if every patch contributes equally to the denoising process, the NLM filter will act as a uniform smoothing filter (the results of this type of filter can be seen on the figure below, being similar to the one given by the NLM). Then the averaging pixel values from a large neighborhood around each pixel will lead to significant loss of image features such as edges.

Thus, fine details in images, which are represented by the areas with high variations in pixel values, will become blurred, as these high variations will be diminished due to the quantization.

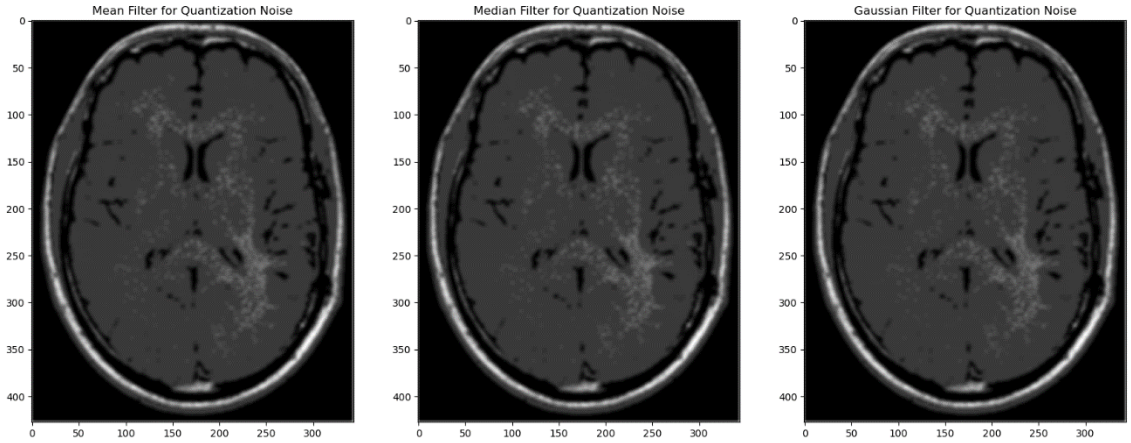

Figure 8: Smoothing filters (Mean, Median and Gaussian) for Quantization Noise.

4.2. Anisotropic Filtering: Perona and Malik

The Anisotropic Filtering is a technique used for image smoothing and noise reduction. Unlike traditional smoothing filters that apply a uniform blurring by diffusing the grey values in all directions, anisotropic filtering adapts its smoothing based on the local image structure, allowing to disregard areas where we are not interested in blurring the image, such as the edges.

The Anisotropic filter works in the following way: it is based on partial differential equations (PDEs) that control the flow of information within an image. It calculates the gradient of pixel values in the image to determine the direction of edges or features. Smoothing is greater within a region (areas with low gradient) than with neighboring regions (areas with high gradient), preserving edge details. The filter iteratively diffuses information across the image while considering local gradients, ultimately reducing noise while preserving important image features.

<table>
<tr><td colspan="4">def anisotropic_filter(image, niter=1, kappa=50, gamma=0.1, option=1):</td></tr>
<tr><td rowspan="5">anisotropic_diffusion ()</td><td colspan="2">image</td><td>Input image with noise to which Anisotropic filtering will be implemented.</td></tr>
<tr><td rowspan="4">kw = dict()</td><td>nitter</td><td>Number of iterations. The more iterations, the stronger the denoising effect.</td></tr>
<tr><td>kappa</td><td>Conduction coefficient that controls conduction as a function of the gradient. A lower value allows small intensity gradients to block conduction, preserving sharp edges. A higher value reduces the influence of intensity gradients on conduction, which may be beneficial for smoothing out low-contrast noise.</td></tr>
<tr><td>gamma</td><td>Speed of diffusion. Lower values result in slower diffusion, which favor edge preservation while higher values will make the diffusion process more isotropic and result in more smoothing across edges.</td></tr>
<tr><td>option = ‘1’</td><td>Option for the diffusion equation:
- 1: Perona Malik diffusion equation, favors high contrast edges over low contrast ones
- 2: Tukey’s biweight function, favors wide regions over smaller ones.
- 3: preserves sharper boundaries and improves the automatic stopping of the diffusion.

option = 1 will be implemented to use Perona and Malik diffusion.</td></tr>
</table>

Output: denoised image after Anisotropic diffusion filtering

4.2.1 Implementation with Gaussian Noise

Implementing the Perona and Malik anisotropic filter to denoise an image with added gaussian noise, using the parameters:

Parameter Name	Parameter Value	Comment
nitter	10	# nº iterations chosen
kappa	20	# conduction coefficient
gamma	0.1	# Speed diffusion chosen

For these parameters the following denoised image (in comparison with the noisy image and the original one) is obtained:

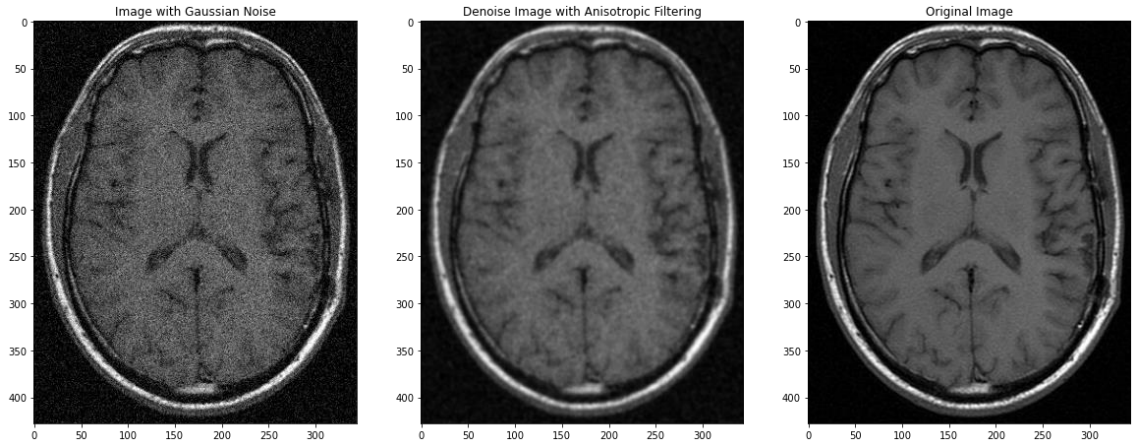

Figure 9: Comparison between noisy image, denoised image and original image for Gaussian noise.

The Perona and Malik anisotropic filter is a powerful tool for noise reduction while preserving edges, but proper parameter adjustment is essential. The number of iterations allows the anisotropic filter to adapt more rigorously to the noise characteristics. A value of 10 has been chosen as an optimal balance between noise reduction and detail preservation. Unlike lower values which denoising effect will be worst, remaining to much noise in the image, and higher values which denoising effect will be too aggressive, making the image too smooth and blurry.

Concerning the diffusion coefficient, kappa, which controls the strength of diffusion, the minimum value suggested in the documentation is used. This means that a moderate amount of noise removal, without being overly aggressive, is suitable for preserving fine details such as the edges, as it can be seen in *Figure 9*. However, if kappa is incremented, the image will become smoother as the filter becomes more aggressive, losing to many details of the image as a higher value would not allow small intensity gradients to block conduction, resulting in more smoothing across edges.

At last, taking into account that a stable value for gamma, the speed of diffusion, is considered to be under 0.25, a value of 0.1 has been chosen which makes the filter more sensitive to edges, thus favoring its preservation. Nonetheless, if higher values will be chosen, then the

diffusion process will approach to an isotropic diffusion, becoming the results more smooth across edges.

Now, by comparing the anisotropic filtered image with the images resulting from smoothing filters for Gaussian noise in *Figure 3* , it can be appreciated how the advanced filter is slightly better in the removal of gaussian noise as traditional smoothing filters.

A great advantage of the anisotropic filter is its versatility for reducing different noise types by adjusting its parameters. However, its main downside is that it can be computationally intensive, hence for cases were a similar outcome to smoothing filters is obtained, it is advisable to opt for them instead of more unnecessarily complex filters, such as the case of Salt and Pepper explained below.

4.2.2 Implementation with Salt & Pepper Noise

Implementing the Perona and Malik anisotropic filter to denoise an image with added salt and pepper noise, using the parameters:

Parameter Name	Parameter Value	Comment
nitter	10	nº iterations chosen
kappa	100	conduction coefficient
gamma	0.1	Speed diffusion chosen

For these parameters the following denoised image (in comparison with the noisy image and the original one) is obtained:

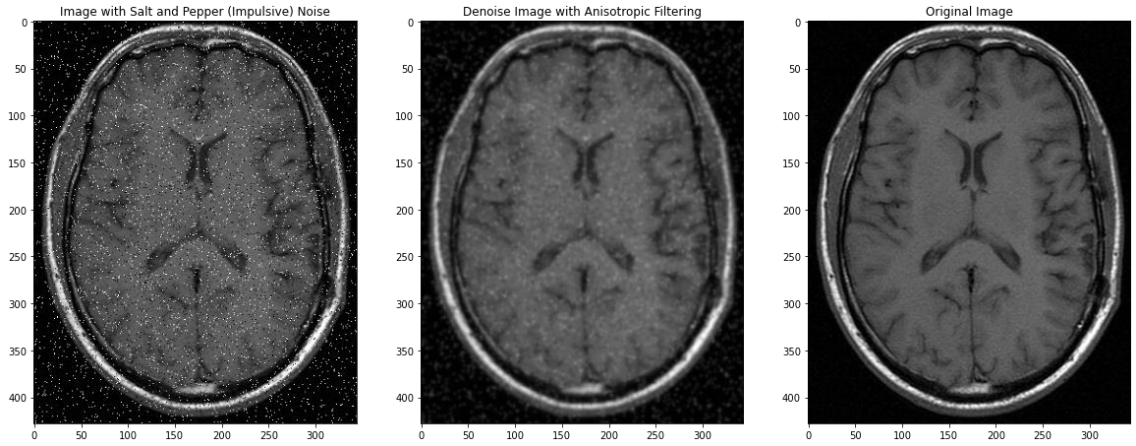

Figure 10: Comparison between noisy image, denoised image and original image for Salt and Pepper (Impulsive) noise.

The Perona and Malik anisotropic filter has proven to be decently effective (this is observed in *Figure 10*) at reducing salt and pepper noise while preserving edges and fine structures. This is thanks to its smoothing adaptability based on the local image structure.

The main variation in parameter values takes place in the kappa conduction coefficient. This has been incremented to reduce the influence of high intensity gradients on conduction because for salt and pepper noise there is predominance of high intensity gradient due to having either white (255) or black (0) pixels. In other words, a greater kappa value increases the diffusion strength, which means more smoothing and more noise reduction at the cost of also smoothing fine details.

The gamma parameter, in this case, needs to remain relatively low so that edges are retained sharply, and smoothness is compensated for, as previously seen for Gaussian noise.

Now, by comparing the anisotropic filtered image with the images resulting from smoothing filters in *Figure 6*, it can be appreciated how the anisotropic filter is still not good enough as the Median filter because it does not focus on removing outliers but instead diffusing based on pixel intensity gradients, as explained in the NLM filter when facing with this type of noise.

4.2.3 Implementation with Quantization Noise

With the results of the anisotropic diffusion for the quantization noise, shown in *Figure 11*, it can be concluded that this filter does not have any effect over quantization noise as seen also in the NLM filter. Added to the fact that adjusting parameters is not of any help and only blurring is achieved.

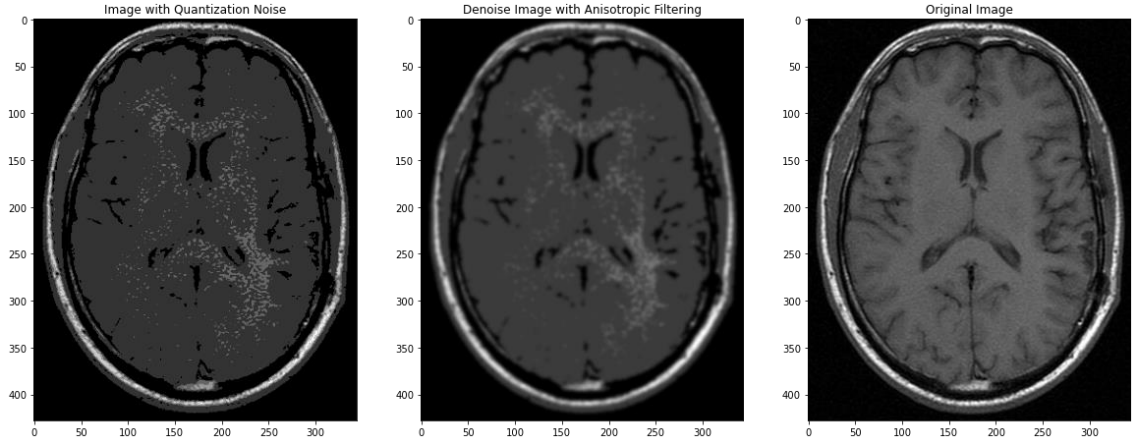

Figure 11: Comparison between noisy image, denoised image and original image for Quantization noise.

The reason for this is further explained in the quantization noise section of the NLM filter.

4.3. Bilateral Filtering

The Bilateral Filter is a nonlinear image processing technique that combines spatial smoothing and range filtering to reduce noise in digital images, by replacing the intensity in each pixel of the image by the weighted average of the neighboring pixels intensity. The weights will not only depend on the pixels Euclidean distance, but also on radiometric differences, as well as they follow a Gaussian distribution. Thus, unlike traditional filters that perform uniform blurring, the Bilateral Filter preserves edges and fine details while reducing noise. It is particularly effective at filtering Gaussian and Impulsive noise.

The Bilateral filter works in the following way: it first applies a spatial filtering (Gaussian Filter). A Gaussian kernel is used to compute the weighted average of pixel values within a local neighborhood. This step smoothes the image based on spatial distances, reducing noise and blurring. Then it performs range filtering (Weighted Averaging) such that for each pixel in the input image, a weighted average is calculated using the intensity similarity (range) among neighboring pixels. The weights consider both spatial and range components, with spatial information from the Gaussian kernel and range information based on intensity differences. Finally, it combines the results of spatial and range filtering filtering are combined (by multiplying them) to obtain the final filtered image.

<table>
<tr><td colspan="4">def bilateral_filter(image, win_size=None, sigma_color=None, sigma_spatial=1, bins=10000):</td></tr>
<tr><td rowspan="5">denoise_bilateral ()</td><td colspan="2">image</td><td>Input image with noise to which Bilateral filtering will be implemented.</td></tr>
<tr><td rowspan="4">kw = dict()</td><td>win_size = ‘None’</td><td>Determines the size of the filtering window. A larger window size will consider a broader neighborhood when filtering and be better fro smoothing. Small windows will preserve detail and remove isolated noise pixels (salt&pepper).</td></tr>
<tr><td>sigma_color</td><td>Standard deviation for gray value / color distance (radiometric similarity). It controls the color similarity weight. A larger value results in averaging of pixels with larger radiometric differences, and a lower preserve more color.</td></tr>
<tr><td>sigma_spatial</td><td>Controls the spatial similarity weight. A larger value results in averaging pixels with larger spatial differences, effectively smoothing over larger areas.</td></tr>
<tr><td>bins= ‘10000’</td><td>Determines the number of discrete values for the Gaussian weights of color filtering. A larger value results in improved accuracy</td></tr>
</table>

		mode = *'reflect'*	Determines how the input array is extended when the filter overlaps a border. The 'reflect' value means that the input is extended by a padding that reflects about the edge of the last pixel.
Output: denoised image after Bilateral filtering			

4.3.1. Implementation with Gaussian Noise

Implementing the Bilateral filter to denoise an image with added Gaussian noise, using the parameters:

Parameter Name	Parameter Value	Comment
sigma_color	0.2	# std for radiometric similarity chosen
sigma_spatial	1	# std for range distance chosen

For these parameters the following denoised image (in comparison with the noisy image and the original one) is obtained:

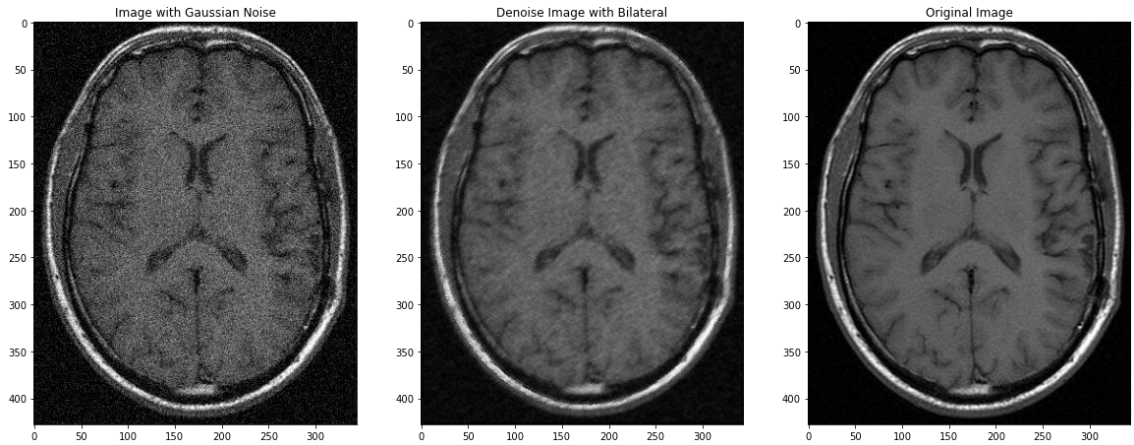

Figure 12: Comparison between noisy image, denoised image and original image for Gaussian noise.

The Bilateral filter is another filtering technique used for denoising, edge-preserving smoothing, and detail enhancement. When a Gaussian noise is added to an image, the Bilateral filter can help reduce the noise while preserving important edges and details.

This advance filter has two main parameters. The sigma_spatial parameter controls spatial extent or neighborhood over which the filter operates.

A smaller sigma_spatial limits the neighborhood to a smaller region around each pixel, preserving detailing without enough smoothing, while a larger sigma_spatial expands the neighborhood, causing more smoothing but also blurring of the edges. The default value given by the function is used, since this is the more balance approach.

Then the sigma_color controls the range of intensity similarity. A small value has been chosen which means that only pixels with very similar intensity values will be considered in the filtering process, hence preserving fine intensity details. This way enough noise is removed without too much blurring.

Now, by comparing the bilateral filtered image with the images resulting from smoothing filters in *Figure 3*, it can be seen that the bilateral filter outputs a similar denoised image as the smoothing filters. According to theory better results were expected because Bilateral filter works by applying a weighted average to each pixel in the image, with the weights determined by two Gaussian functions: one based on spatial distance (sigma_spatial) and the other on intensity difference (sigma_color). However, tuning of this parameter either generate very blurry results or very images that are not properly denoised.

4.3.2. Implementation with Salt & Pepper Noise

Implementing the Bilateral filter to denoise an image with added salt and pepper noise, using the parameters:

Parameter Name	Parameter Value	Comment
sigma_color	0.3	# std for radiometric similarity chosen
sigma_spatial	1	# std for range distance chosen

For these parameters the following denoised image (in comparison with the noisy image and the original one) is obtained:

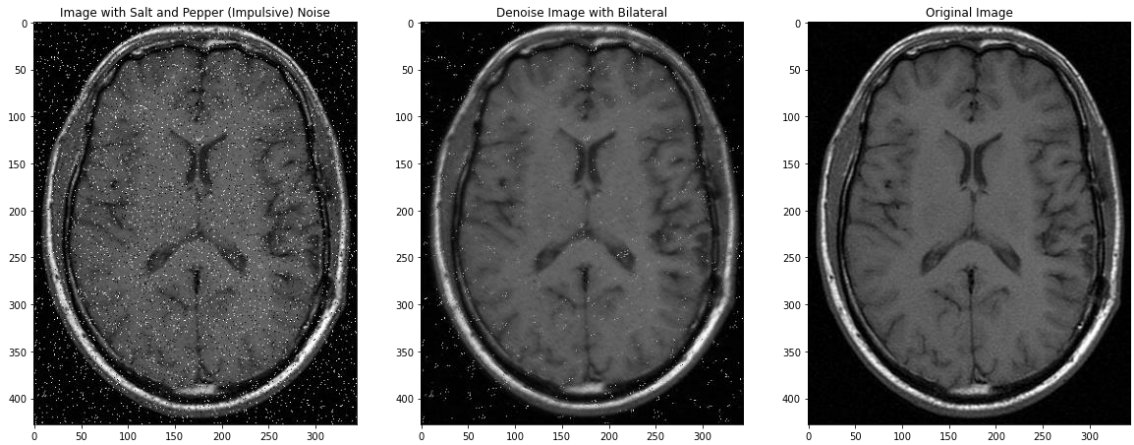

Figure 13: Comparison between noisy image, denoised image and original image for Gaussian noise.

It is believed that a larger sigma-spatial should be used. The reason is that salt and pepper noise consists of isolated noisy pixels that are very different from their neighboring pixels. By using a larger sigma_spatial, the filter to includes more distant pixels in the averaging process. A smaller sigma_spatial would limit the filter to a tighter spatial neighborhood and may not effectively remove the noisy outliers, as it may not include enough non-noisy neighboring pixels to mitigate the effect of the outliers. In spite of this the best resulting image was obtained with a value of 1.

Also, a small sigma_color is thought to be better since it ensures that the filter is sensitive to differences in pixel intensities, which is important for correctly identifying and treating the noisy pixels as outliers. This theoretical deduction was not met with the actual value obtained for the best resulting image which was 0.3.

It should be mentioned that we do not fully understand why our assumptions are not met in practice.

By comparing the theoretical effectiveness of the bilateral with that of traditional smoothing techniques, several conclusions are reached. First traditional smoothing filters work by averaging pixel values within a fixed-size neighborhood while the bilateral allows to control both spatial proximity and intensity similarity when computing the weighted average. Second, this means that the bilateral filter is more adaptable to different scenarios and noise levels. As a result, the filter is more capable of preserving edges and important image details while reducing noise. These assumptions were not able to be

demonstrated since the result provided by the bilateral filter is very similar to smooth filters, as seen in *Figure 6.*

4.3.3. Implementation with Quantization Noise

With the results of the anisotropic diffusion for the quantization noise, shown in *Figure 11*, it can be concluded that this filter does not have any effect over quantization noise as seen also in the NLM filter. Added to the fact that adjusting parameters is not of any help and only blurring is achieved.

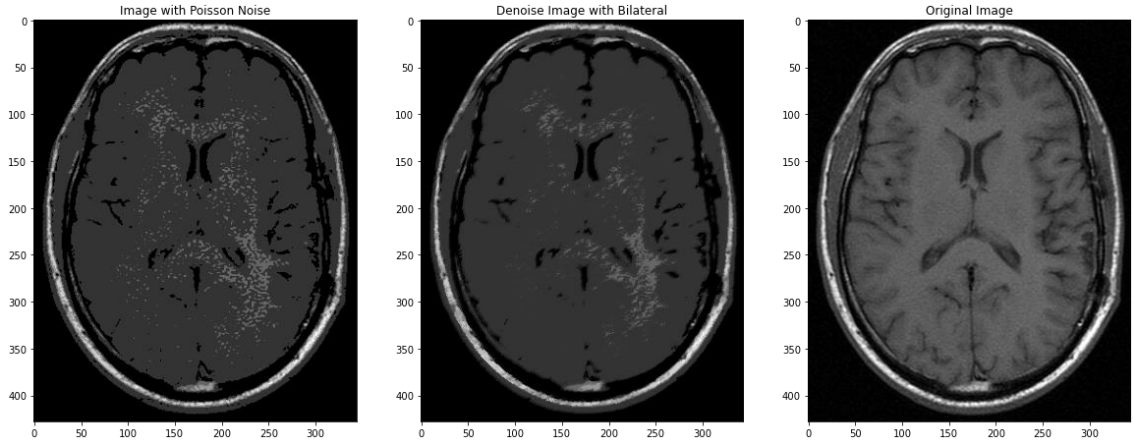

Figure 14: Comparison between noisy image, denoised image and original image for Quantization noise.

The reason for this is further explained in the quantization noise section of the NLM filter.

4.4. Denoise TV Chambolle Filtering

The denoise TV Chambolle filter is a non-linear denoising and regularization filter designed to reduce noise in an image while preserving edges and important image features. This filter is based on the total variation (TV) regularization method and uses the Chambolle algorithm for optimization. It's particularly effective at enhancing diagnostic quality by reducing noise while retaining important structural information.

The TV Chambolle filter works in the following way: it is a type of anisotropic filter based on the concept of total variation, which measures the variation in intensity across an image. Hence the filter operates on the principle of minimizing the total variation of the image, which is high at the edges and low in flat regions. The procedure is similar to that of the Perona and Malik anisotropic filter. In this case we also have an iterative process in which gradients are computed to locate edges and apply smoothing to the correct regions of the image. However now we a different component 'lambda' instead of the previously seen diffusion coefficient. This parameter controls the trade-off between noise reduction and edge-preservation with the aim of minimizing TV of the image.

<table>
<tr><td colspan="4">def denoise_tv_chambolle_filter (image, weight=0.1, eps=0.0002, max_num_iter=200):</td></tr>
<tr><td rowspan="4">denoise_tv_chambolle()</td><td colspan="2">image</td><td>Input image with noise to which TV Chambolle filtering will be implemented.</td></tr>
<tr><td rowspan="3">kw = dict()</td><td>Weight (1/lambda)</td><td>Controls the strength with which the denoise is performed. The higher the weight, the more smoothing, but as a drawback, the image will be more blurred and less detailed.</td></tr>
<tr><td>eps</td><td>Parameter used to control the step size in the optimization algorithm. It's a small positive number that ensures numerical stability during the iterations.</td></tr>
<tr><td>max_num_iter</td><td>Maximal number of iterations used for the optimization.</td></tr>
<tr><td colspan="4">Output: denoised image after TV Chambolle filtering</td></tr>
</table>

4.4.1. Implementation with Gaussian Noise

Implementing the denoise TV Chambolle filter to denoise an image with added gaussian noise, using the parameters:

Parameter Name	Parameter Value	Comment
weight	0.1	Strength of the denoising
eps	0.0002	Step size in the optimization algorithm
max_num_iter	200	Maximal number of iterations

For these parameters the following denoised image (in comparison with the noisy image and the original one) is obtained:

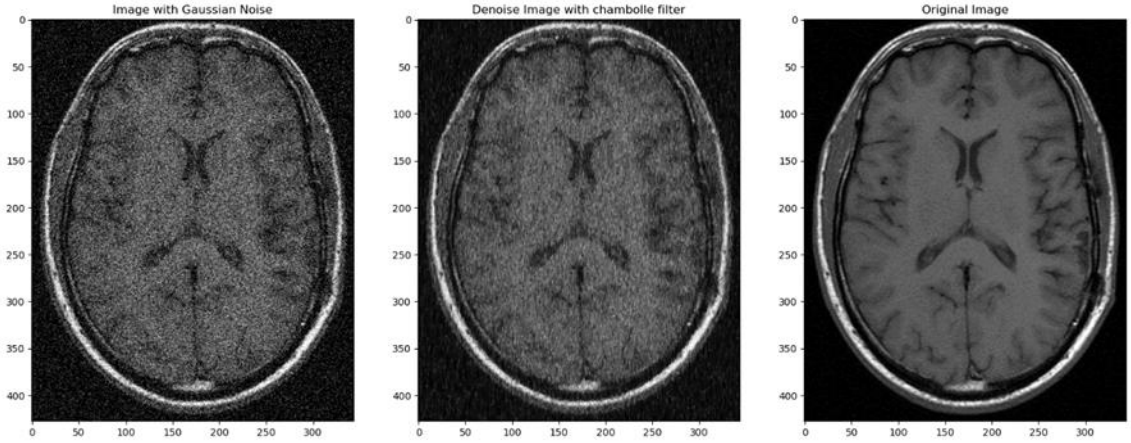

Figure 15: Comparison between noisy image, denoised image and original image for a Gaussian noise.

With a weight of 0.1, the noise is properly removed. This value will not be increased because the image will be blurred without being necessary.

Now, by comparing the filtered image with TV Chambolle with the images resulting from smoothing filters in *Figure 3*, it can be appreciated how the TV Chambolle filter is more effective at preserving edges. The mean and median filter seem better at reducing noise but blur the image considerably more. An in-between result is obtained with the gaussian, with both decent noise reduction and edge preservation.

4.4.2. Implementation with Salt & Pepper Noise

Implementing the denoise TV Chambolle filter to denoise an image with added salt and pepper noise, using the parameters:

Parameter Name	Parameter Value	Comment
weight	0.2	# Strength of the denoising
eps	0.0002	# Step size in the optimization algorithm
max_num_iter	200	# Maximal number of iterations

For these parameters the following denoised image (in comparison with the noisy image and the original one) is obtained:

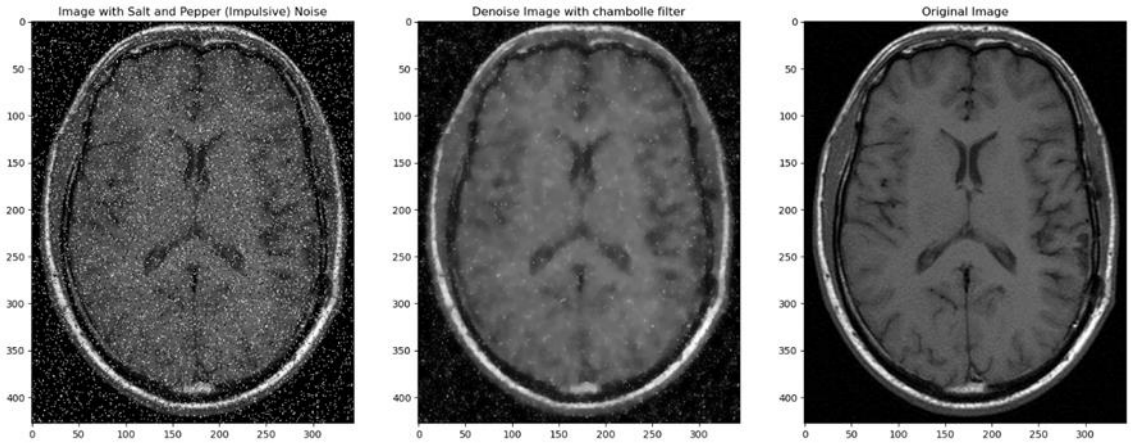

Figure 16: Comparison between noisy image, denoised image and original image for Salt and Pepper (Impulsive) noise.

For salt and pepper noise, we decided to increase the weight to 0.2, since with 0.1, the noise is still present similarly to the noisy image. Hence, by increasing it to 0.2, it is denoised with more strength, but as a drawback the image is more blurred.

Now, by comparing the filtered image with TV Chambolle with the images resulting from smoothing filters in *Figure 6*, it can be appreciated how the TV chambolle filter does a good job at reducing noise while maintain sharp edges, although it does not completely eliminate isolated salt and pepper noise pixels. As for gaussian and mean filters, they provide a similar mediocre output with the gaussian preserving a bit better the edges and giving a less blurred image (this makes sense according to the theory since the gaussian filter is a weighted mean). The median filter, on the other side, is clearly the most robust against salt and pepper noise and is effective at removing isolated noise pixels. According to theory this also makes sense because salt and pepper noise manifests as isolated noise pixels that are significantly different from their surroundings, hence the median filter is effective at removing these isolated pixels by replacing them with a median value from the surrounding, more typical pixel values.

4.4.3. Implementation with Quantization Noise

With the results of the tv Chambolle filter for the quantization noise, shown in *Figure X*, it can be concluded that this filter does not have any effect over quantization noise as seen also in the NLM and anisotropic filter. Added to the fact that adjusting parameters is not of any help and only blurring is achieved.

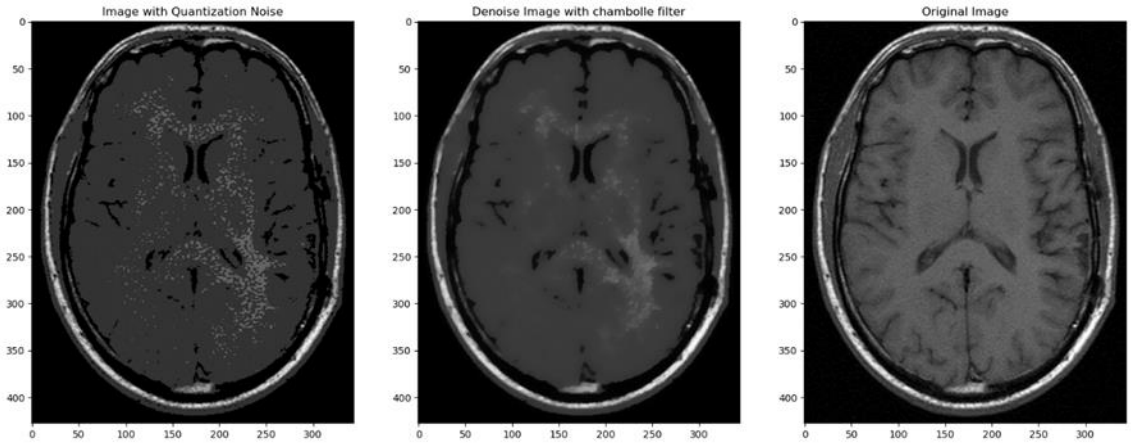

Figure 17: Comparison between noisy image, denoised image and original image for Quantization noise.

The reason for this is further explained in the quantization noise section of the NLM filter.

4.5. Wiener Filtering

The Wiener filtering technique is based on a statistical approach to filter the noise. It is a linear estimation of the original image, allowing to reduce the amount of noise present in an image by comparison with an estimation of the desired noiseless signal. Since it is possible to recover the image by inverse filtering. The Wiener filter executes an optimal trade-off between inverse filtering and noise smoothing. It removes the additive noise and inverts the blurring simultaneously while minimizing the overall Mean Square Error in the operation. It is especially effective at denoising images with gaussian noise and motion blur.

The filter works in the following way: the Wiener filter is particularly useful when the statistical properties of the noise and the original image are known and so its performance depends on the accuracy of those assumptions regarding the statistical properties. The Wiener filter operates in the frequency domain. To estimate the original image, the Power Spectral Density (PSD) of the original image and of the noise is computed. Then, the inverse Fourier transform is applied to convert it back to the spatial domain, resulting in the Wiener filter. Finally, the Wiener Filter is convolved with the image to obtain the estimated original one.

<table>
<tr><td colspan="4">def wiener_deconvolution(image, balance, psf):</td></tr>
<tr><td rowspan="3">wiener ()</td><td colspan="2">image</td><td>Input image with noise to which Wiener filtering will be implemented.</td></tr>
<tr><td rowspan="2">kw = dict()</td><td>psf[5]</td><td>Point Spread Function represents how the image was blurred. It's a kernel that describes the blurring process. The choice of the PSF depends on the characteristics of the blurring and should be estimated.</td></tr>
<tr><td>balance</td><td>Regularization parameter value that tunes the trade-off between noise reduction and detail preservation. A higher value of balance tends to suppress noise more but can result in the loss of fine details.</td></tr>
<tr><td colspan="4">Output: denoised image after Wiener filtering</td></tr>
</table>

5: Due to the complexity and an incomplete understanding of the PSF parameter and the techniques used to properly estimate it we have decided to instead apply two kernels, a gaussian one and a mean kernel, that describe generic types of blurring. We are aware therefore that filtered image results could be improved if it weren't for these limitations.

4.5.1. Implementation with Gaussian Noise

Implementing the Wiener filter to denoise an image with added gaussian noise, using the parameters:

Parameter Name	Parameter Value	Comment
pdf	gaussian_kernel (kernel_size = 3)	# Kernel that describes how the image was blurred
balance	0.1	# Parameter that controls the trade-off between noise reduction and detail preservation

For these parameters the following denoised image (in comparison with the noisy image and the original one) is obtained:

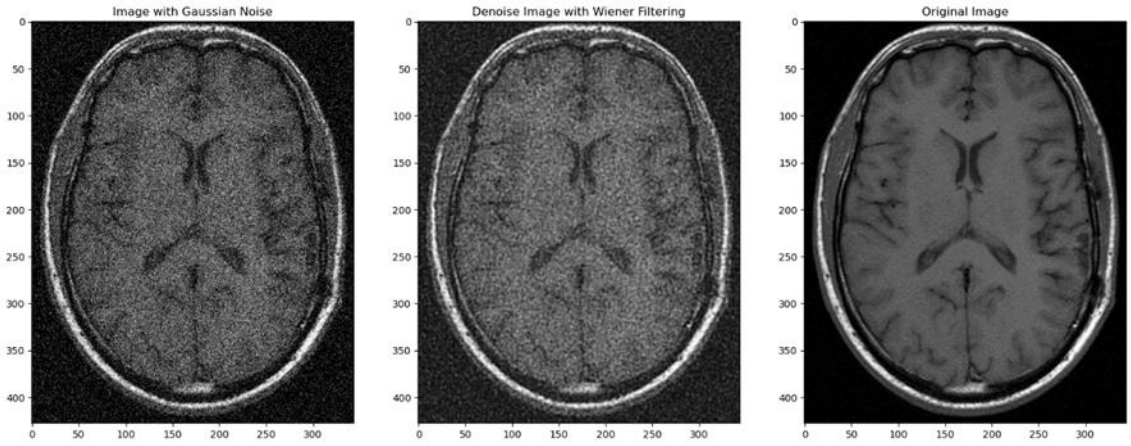

Figure 18: Comparison between noisy image, denoised image and original image for Gaussian noise

After testing with the two different kernels (gaussian and mean) with different sizes and different balance parameters, it has been found that the most effective parameter values are the ones on the table above.

A gaussian kernel was used since it's the one that describes the added noise best. A small kernel size of 3x3 was chosen. Even though a larger kernel size results in more smoothing (because it considers a wider region of pixels in the convolution operation), greater kernel sizes did not actually provide better results because their consequent blurring worsened the image. Additionally, a smaller kernel proved to be better at preserving fine details.

As seen below, a balance of 0.1 is optimal for removing gaussian noise without much detail loss. For a 0.01 value it means that the balance focuses more on preserving fine details than denoising. However, a low balance value can lead to the appearance of a 'flaky' or ringing artifact effect on the image. Since the Wiener filter operates in the frequency domain, in the case where the balance is low, the filter acts more conservatively in suppressing noise and prioritizes preserving high-frequency components in the image. High-frequency components can include noise and subtle variations in pixel values, which can lead to oscillations and ringing artifacts around edges and fine details.

On the other side of the range, a higher value should suppress noise more, but a balance of 0.5 barely suppresses any more noise than a balance of 0.1, though it does cause a noticeable difference in blurring.

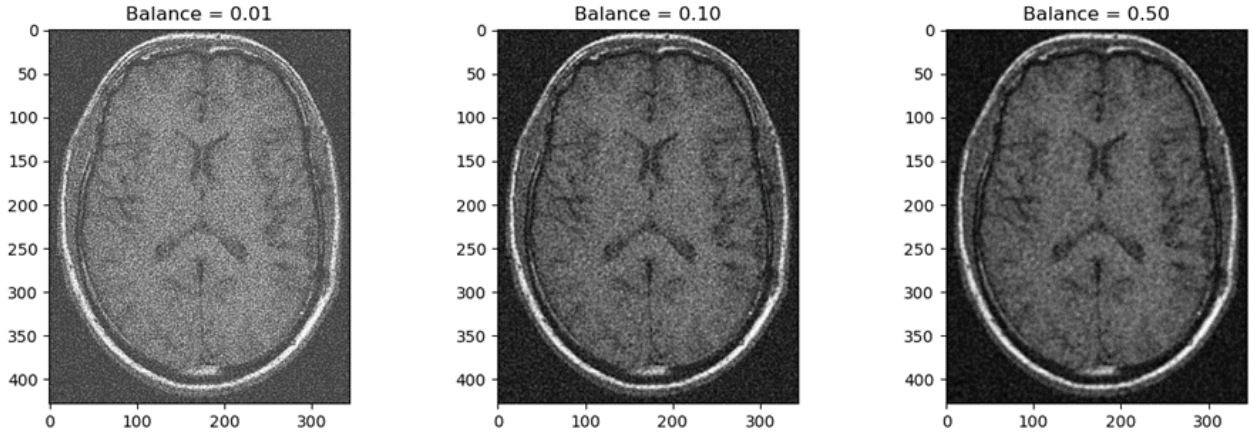

Figure 19: Benchmarking for the balance parameter between 0.01 and 0.5.

Now, by comparing the filtered image with Wiener with the images resulting from smoothing filters in *Figure 3*, it can be appreciated that the Wiener filter is a powerful tool, especially when noise characteristics are known. It effectively reduces Gaussian noise by adaptively adjusting its response based on the noise spectra. It can also be seen that the Wiener filter achieved Gaussian noise reduction without significantly blurring the image. In fact, the Wiener filter achieves a similar denoising result as the Gaussian filter but with better preservation of details (greater spatial resolution).

4.5.2. Implementation with Salt & Pepper Noise

Implementing the Wiener filter to denoise an image with added gaussian noise, using the parameters:

Parameter Name	Parameter Value	Comment
pdf	gaussian_kernel (kernel_size = 2)	# Kernel that describes how the image was blurred
balance	1	# Parameter that controls the trade-off between noise reduction and detail preservation

For these parameters the following denoised image (in comparison with the noisy image and the original one) is obtained:

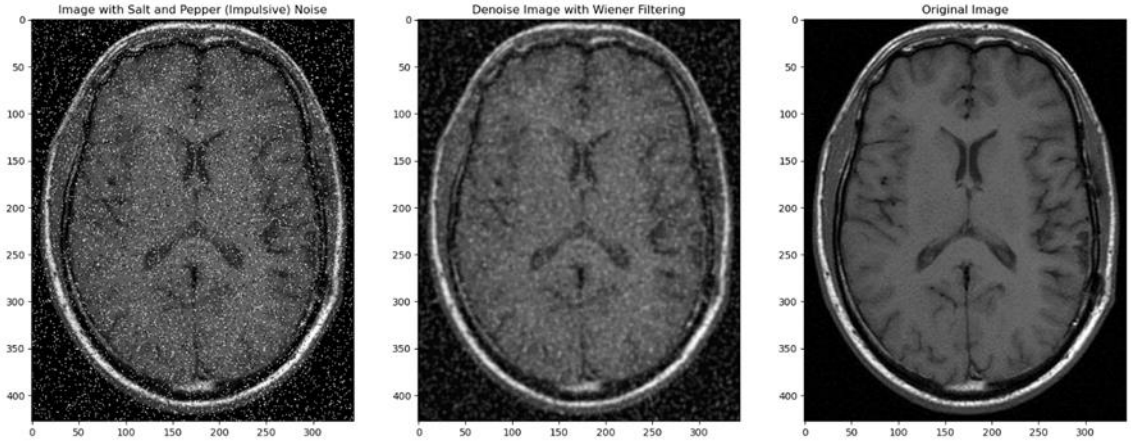

Figure 20: Comparison between noisy image, denoised image and original image for Salt and Pepper (Impulsive) noise

To cover for the lack of a proper estimation of the Point Spread Function (impulse response of the system) that accurately accounts for the degradation of the imagen, a more general gaussian kernel of size 2x2 has been used. This kernel, in this case, worked better than the mean kernel (matrix of 1/25 with size 5x5), since greater kernel sizes added more blurring without improvement in denoising.

The chosen balance value was 1, which provided an intermediate output between a predominantly blurred and denoised image (balance of 10) and a less denoised and sharper image (balance of 0.1)

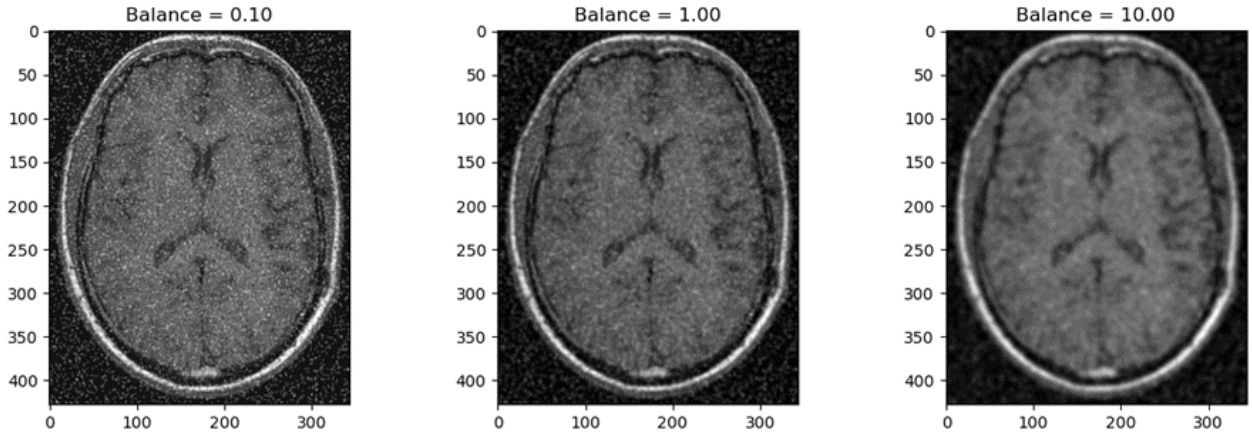

Figure 21: Benchmarking for the balance parameter, from 0.1 to 10.

Now, by comparing the filtered image with Wiener with the images resulting from smoothing filters in *Figure 6*, it can be appreciated that Wiener filter's performance is not satisfactory, compared to other smooth filters (particularly the median one, successful in removing outliers as explain in the NLM filter section) because removing isolated salt and pepper pixels becomes challenging for it, since its performance depends on the accurate estimation of the signal and noise spectra and in this case, it becomes hard to determine given that the randomness of salt and pepper noise that is not subject to a particular noise distribution.

4.5.3. Implementation with Quantization Noise

Implementing the Wiener filter to denoise an image with added quantization noise the following denoised image is obtained:

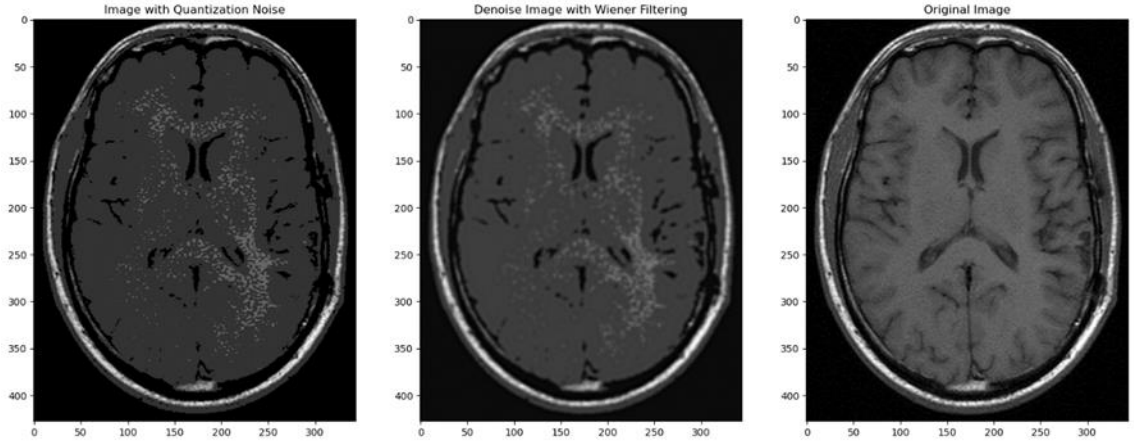

Figure 22: Comparison between noisy image, denoised image and original image for Quantization noise.

Recalling the quantization noise implementation in the NLM filter, similar poor results are also seen in this case. No successful filtering was obtained when the image had quantization noise, as it can also be seen in *Figure x*. Therefore, different parameter values will not be compared in this section. The reason for this is that Quantization noise, unlike other noises presented, is not characterized by a probabilistic approach, it is instead caused by discretization of pixel values (reducing image data by mapping groups of data points to a single point), being challenging for the Wiener filter to remove it, since this filter is based on a statistical approach.

In comparison with traditional smoothing filters, it can also be observed that removing quantization noise as challenging for these filters. This is proved in *Figure 15*.

5. Comparison Between Advanced Filters

In general, all advanced filters have a common advantage, they are able to remove noise while preserving important image features, both great characteristics in the medical image environment. Additionally advanced filters allow more flexibility and adaptability to the characteristics of different images and noises. However, in the next sections it is tried to determine which outputs the best image over 3 different noise types:

5.1. Gaussian Noise

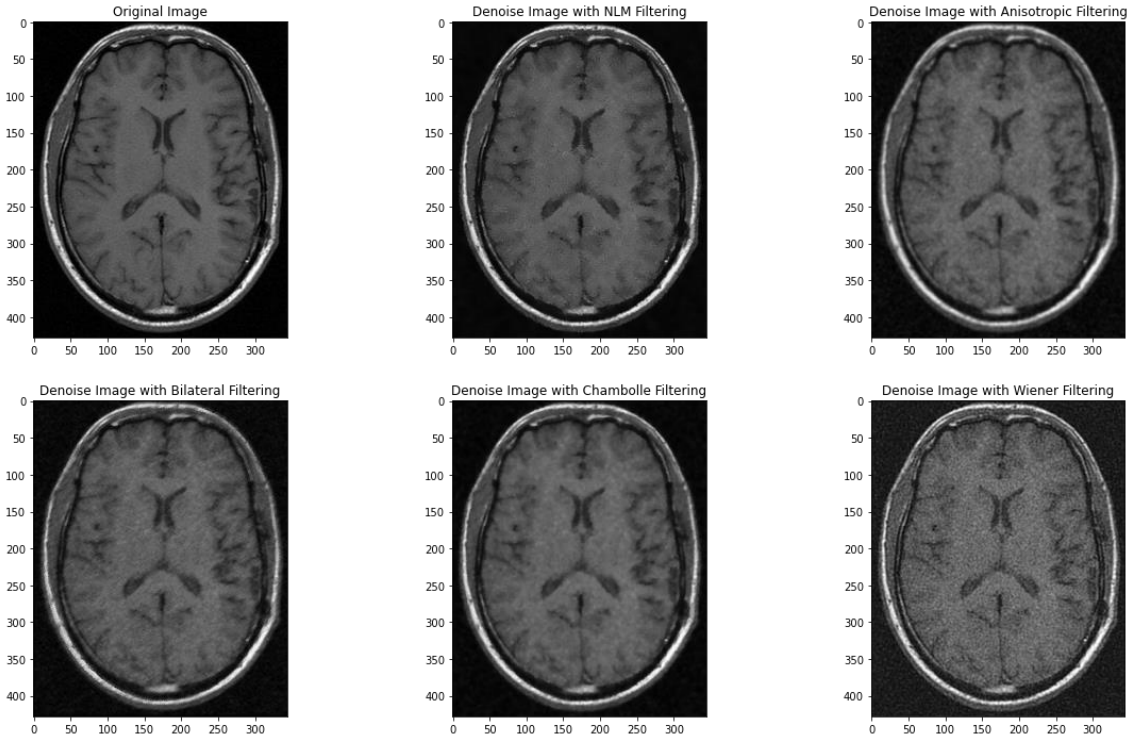

Figure 23: Comparison between advance filters for Gaussian noise.

Regarding the advance filters implemented, for Gaussian noise, it has been seen that both advanced filters studied in class (the NLM and the Perona and Malik) were the ones that gave better results. The reasons for these are explained in the corresponding sections for each of the filters.

It is worth mentioning that the filters studies in class allowed better understanding of them and probably because of this, a more appropriate and consistent application of them.

Gaussian noise however is generally well removed, since is it a very well-defined noise with a distribution which is advantageous for being removed by weighted patch filters as well as frequency filters and variation minimizing filters.

5.2. Salt & Pepper Noise

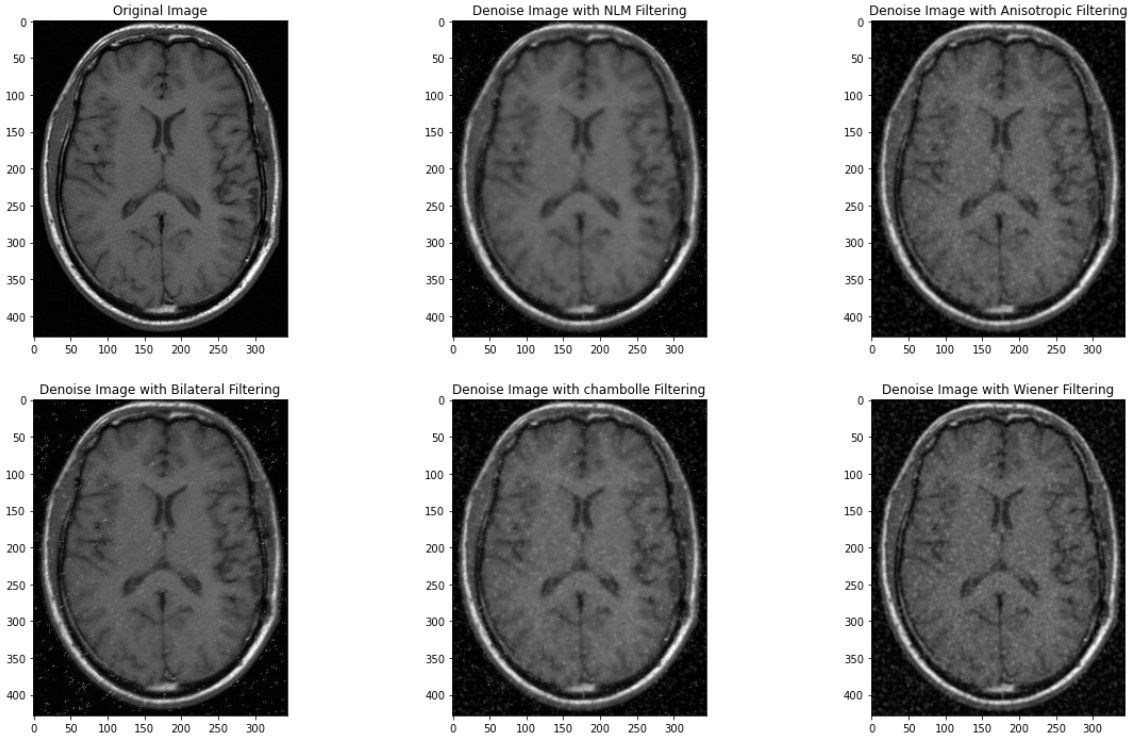

Figure 24: Comparison between advance filters for Salt and Pepper (Impulsive) noise

Regarding the salt and pepper noise, none of the advanced filters have proved to be worse than the median filter at removing this noise. This is due, as explained previously, that they all lack the capacity to remove outliers.

However, it has been seen that even though advance filters do not accomplish a perfect denoising, some of them (NLM and Bilateral) have moderately correct results while the other are slightly worse.

5.3. Quantization Noise

There was no preferable advanced filter for the elimination of quantization noise as justified in the clarification of each filter section and as shown in the Figure X below.

Other, more specialized methods are suggested for the removal of this type of noise, such us:

- Histogram equalization which can help spread out pixel values in the image, potentially reducing the visibility of quantization noise.
- Deep learning models, such as convolutional neural networks (CNNs) are models can be trained to understand and reduce the noise in images effectively.

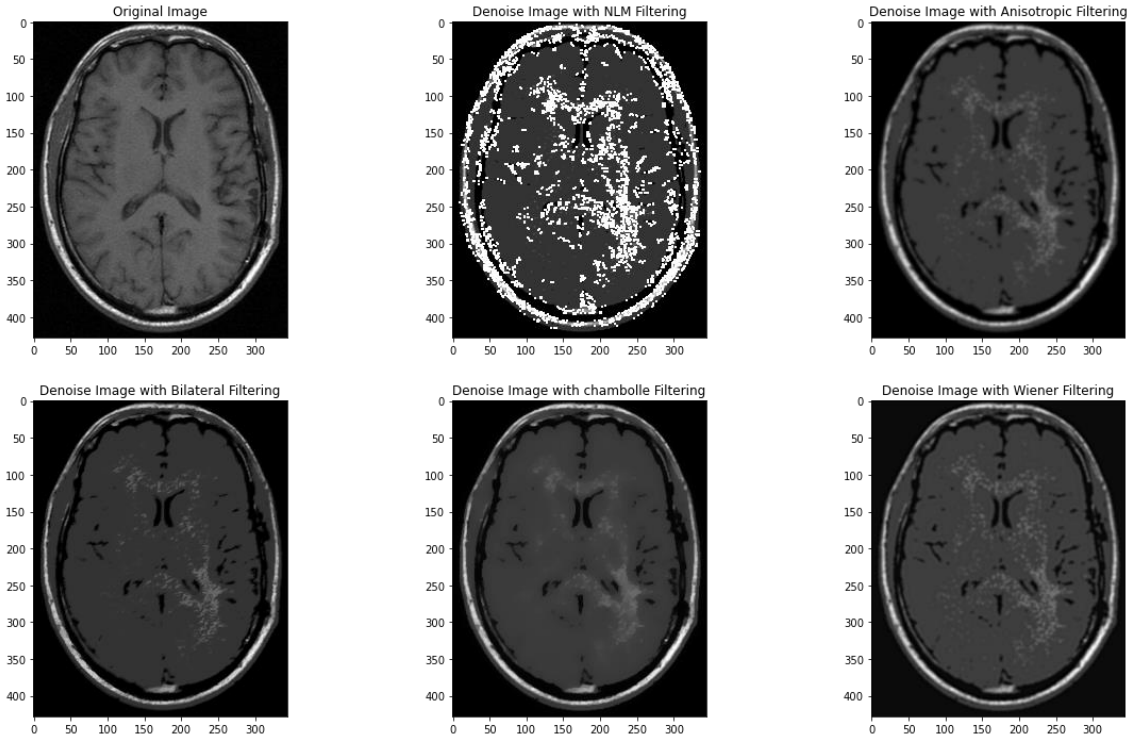

Figure 25: Comparison between advance filters for Salt and Pepper (Impulsive) noise

6. Conclusions

It is evident that not every filter is designed with the same purpose and there is not one filter that solves for the effects of every possible different noise. Image noise and artifacts arise for numerous reasons, a few of them are due to external environment causes (frequency interferences, patient movement, equipment limitations, faults in sensors, etc), randomness with different distributions or digitalization of image.

We have arrived at the conclusion that in real cases, applying several filters (complementing each other) to the same image would be the best approach for overall noise removal in medical images were improving the quality and interpretability of medical images for accurate diagnosis, treatment planning, and medical research is of great importance. In the same line of thought other techniques such as super-resolution techniques could also be introduced to compensate for filters where a large regularization parameter allows to get rid of most of the noise but causes too much blurring.

7. ANNEXES

7.1. ANNEX 1: SPECKLE NOISE

Before the hand in of Lab 1 we have realized that the White noise was properly implemented. A uniformly distributed noise was added to the image, when in reality this Speckle noise and not White noise. Due to time restrictions this noise is not mentioned in the report, however it is present in the code for visualization.

7.2. ANNEX 2: POISSON NOISE

Poisson noise is also implemented in the code but not in the report, this is because due to the nature of the noise, it degraded the image with a low intensity and filtering of the images had little impact on them. The most important conclusion that we were able to deduce from applying this type of noise is regarding the TV Chambolle filter. These is the following: it can be said that the TV Chambolle can handle specially well Poisson noise, since the filter operates on the principle of minimizing the total variation of the image which is a direct neutralizer of the introduced variation in pixel intensities by the Poisson noise.

7.3. ANNEX 3: PROBLEMS REGARDING JUPYTER NOTEBOOK VERSION

For the Anisotropic Diffusion and for the TV Chambolle Filter, some issues appear, that we think is due to the Jupyter Notebook version, from one computer to another.