Lab 2. Filtering with ITK

UNIVERSIDAD DE GRANADA

Autor: Víctor Torres d e la Torre

Task01.
Write a program that reads an image from a file and displays the result of applying the following filters to the input image: RescaleIntensityImageFilter, ShiftScaleImageFilter, and NormalizeImageFilter. Try this program with the input image BrainProtonDensitySlice256x256.png. You may read the corresponding source code of each filter in 'ITK_DIR/Examples/Filtering' to get a clue about how to program this task.

Para poder hacer un análisis más en profundidad de los 4 algoritmos de filtrado que se piden en este ejercicio haremos cada uno de estos por separado, dando diferentes valores a sus parámetros y comparando los resultados.

01.1 RescaleIntensityImageFilter:

Antes de comenzar con este filtro haremos una descripción del mismo para poder entender que hace el algoritmo y como lo podemos manipular a través de sus parámetros o atributos.

RescaleIntensityImageFilter hace un escalado lineal de los valores de pixel de forma que todos los valores de pixel de toda la imagen quedarán en un rango de valores definidos por el usuario. Este es un proceso típico para forzar el rango dinámico de la imagen para ajustarla a una escala particular.

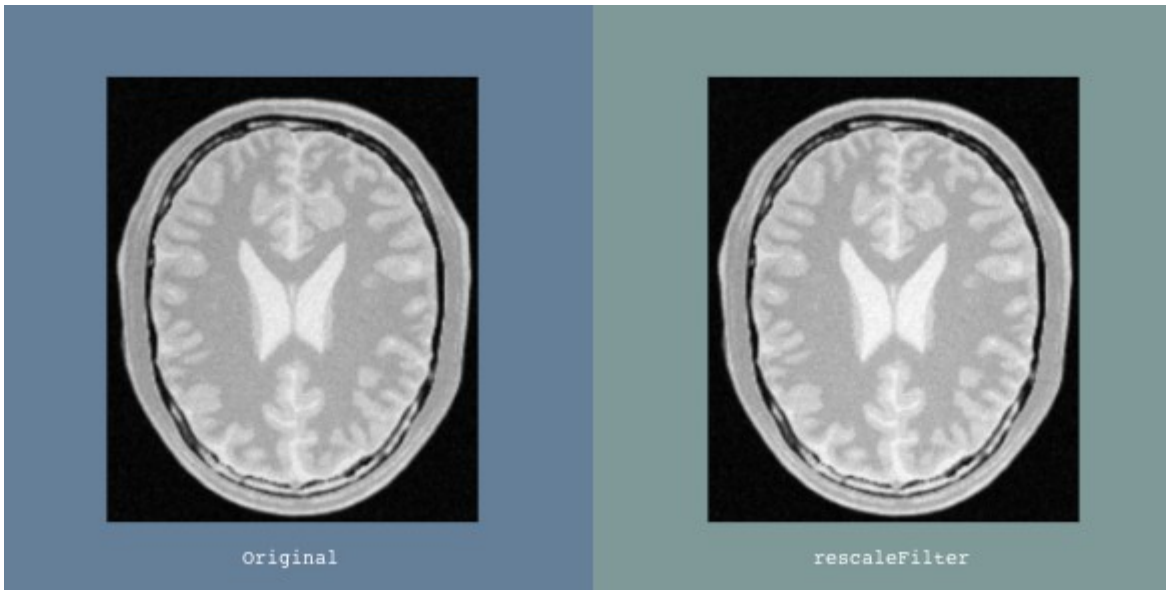
Para dar este rango de valores, ITK nos proporciona dos métodos

```
SetOutputMinimum(x1);
```

```
SetOutputMaximum(x2);
```

Una vez entendemos el funcionamiento del algoritmo vamos a ver como, a pesar de que el algoritmo no va a tener ningún efecto en la visualización de la imagen, sus valores de pixel van a cambiar en función a la formula que aplica dicho algoritmo. Para comprobar esto hemos creado un iterador que recorre la imagen de salida y otro que recorre la imagen de entrada, de esta forma, podremos ver los valores de entrada y salida y así ver que todo ha funcionado perfectamente.

Visualización con y sin filtro (sin diferencias):



Tras imprimir los valores de los pixel en la entrada y la salida, además de imprimir los valores máximos de entrada y salida, vamos a coger un pixel para hacer el cálculo y ver que todo ha funcionado correctamente:

```
in -> 4 out -> 11.4754
maxIn -> 244
minIn -> 0
maxOut -> 100
minOut -> 10
```

$$outputPixel = (inputPixel - inpMin) \times \frac{(outMax - outMin)}{(inpMax - inpMin)} + outMin$$

11.4754 = (4-0) x (100-10)/(244-0) + 10

Como podemos comprobar la igualdad se cumple y por lo tanto el algoritmo ha funcionado correctamente.

01.2 ShiftScaleImageFilter:

Antes de comenzar con este filtro haremos una descripción del mismo para poder entender que hace el algoritmo y como lo podemos manipular a través de sus parámetros o atributos.

ShiftScaleImageFilter también aplica una transformación lineal a las intensidades de la imagen de entrada, pero el usuario especifica la transformación en forma de un factor multiplicador y un valor adicional.

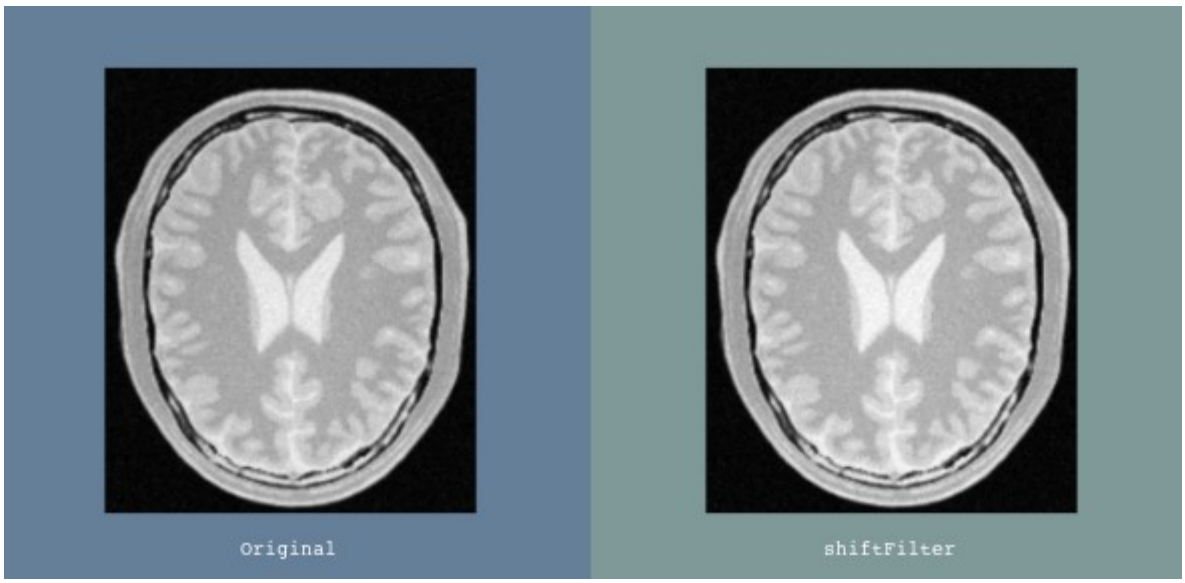
Para hacer esta transformación lineal, ITK nos proporciona dos métodos

```
SetScale(x1);
```

```
SetShift(x2);
```

Una vez entendemos el funcionamiento del algoritmo vamos a ver como, a pesar de que el algoritmo no va a tener ningún efecto en la visualización de la imagen, sus valores de pixel van a cambiar en función a la formula que aplica dicho algoritmo. Para comprobar esto hemos creado un iterador que recorre la imagen de salida y otro que recorre la imagen de entrada, de esta forma, podremos ver los valores de entrada y salida y así ver que todo ha funcionado perfectamente.

Visualización con y sin filtro (sin diferencias):



Tras imprimir los valores de los pixel en la entrada y la salida, además de imprimir los valores máximos de entrada y salida, vamos a coger un pixel para hacer el cálculo y ver que todo ha funcionado correctamente:

```
in -> 4 out -> 34.8
maxIn -> 244
minIn -> 0
maxOut -> 322.8
minOut -> 30
```

$$outputPixel = (inputPixel + Shift) \times Scale$$

Teniendo en cuenta que los valores de shift y scale son 25 y 1.2 respectivamente:

34.8 = (4+25) x 1.2

Como podemos comprobar la igualdad se cumple y por lo tanto el algoritmo ha funcionado correctamente.

01.3 NormalizeImageFilterFilter:

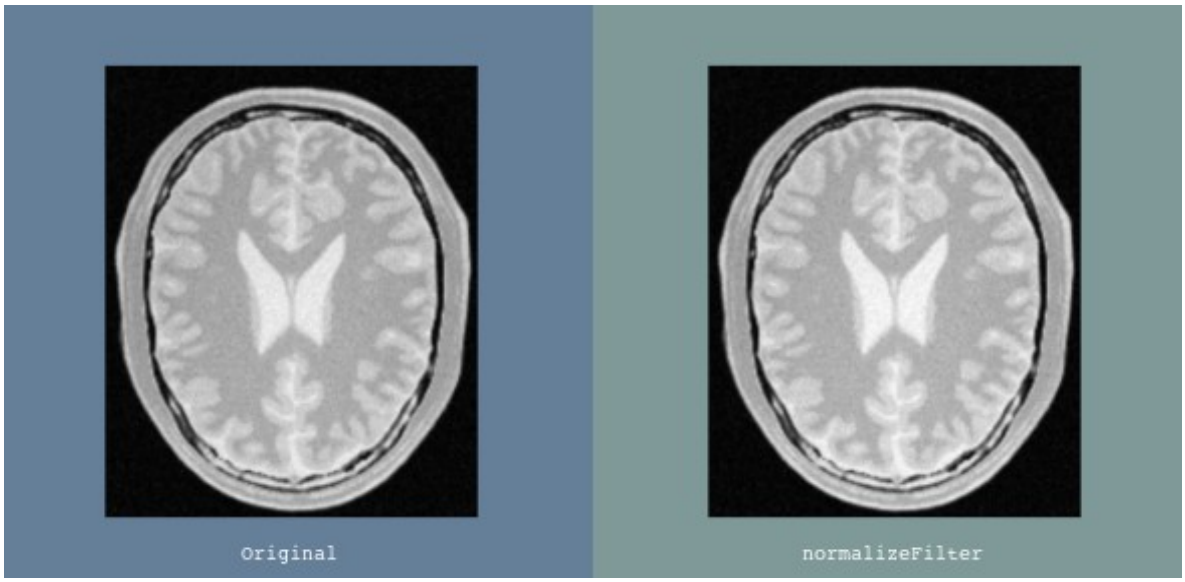
Antes de comenzar con este filtro haremos una descripción del mismo para poder entender que hace el algoritmo y como lo podemos manipular a través de sus parámetros o atributos.

Los parámetros de la transformación lineal aplicada por NormalizeImageFilter se calculan internamente de modo que la distribución estadística de los niveles de gris en la imagen de salida tenga media cero y una variación de uno.

Como se ha explicado previamente, este algoritmo no precisa de parámetros cuyo control recaiga en el usuario.

Una vez entendemos el funcionamiento del algoritmo vamos a ver como, a pesar de que el algoritmo no va a tener ningún efecto en la visualización de la imagen, sus valores de pixel van a cambiar en función a la formula que aplica dicho algoritmo. Para comprobar esto hemos creado un iterador que recorre la imagen de salida y otro que recorre la imagen de entrada, de esta forma, podremos ver los valores de entrada y salida y así ver que todo ha funcionado perfectamente.

Visualización con y sin filtro (sin diferencias):



Tras imprimir los valores de los pixel en la entrada y la salida, además de imprimir los valores máximos de entrada y salida, vamos a coger un pixel para hacer el cálculo y ver que todo ha funcionado correctamente:

```
in -> 4 out -> -1.46875
media -> 123.241
varianza -> 6591.49
```

$$outPixel = \frac{inPixel - mean}{\sqrt{variance}}$$

1.46875 = 4 - 123.241/sqrt(6591.49)

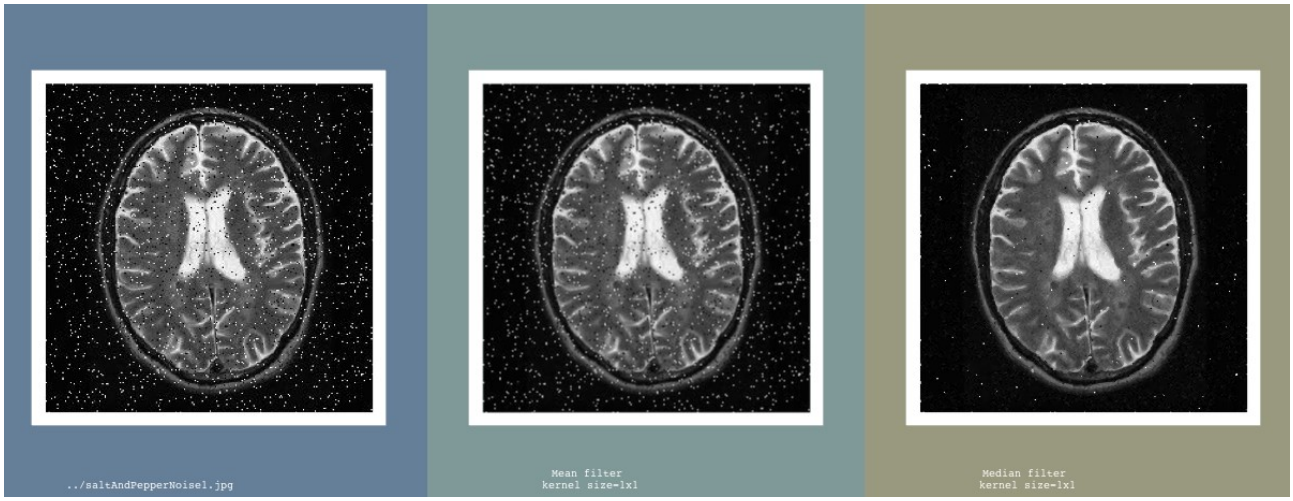
Como podemos comprobar la igualdad se cumple y por lo tanto el algoritmo ha funcionado correctamente.

Task02.
Write a program that reads an image from a file and displays the result of applying two mean filters with neighborhood size of 3 × 3 and 5 × 5, respectively; and two median filters with the same neighborhood sizes. Try the program with the following images: saltAndPepperNoise1.jpg, saltAndPepperNoise2.jpg, and gaussianNoise.jpg. You may read the source code MeanImageFilter.cxx and MedianImageFilter.cxx placed at 'ITK_DIR/Examples/Filtering' to understand the use of both filters.

Para entender como funcionan los algoritmos que se proponen en el ejercicio y poder sacar conclusiones relevantes al respecto, se van a visualizar las combinaciones de filtros y tamaños de vecindad de formas diferentes, lo que nos permitirá un análisis más exhaustivo.

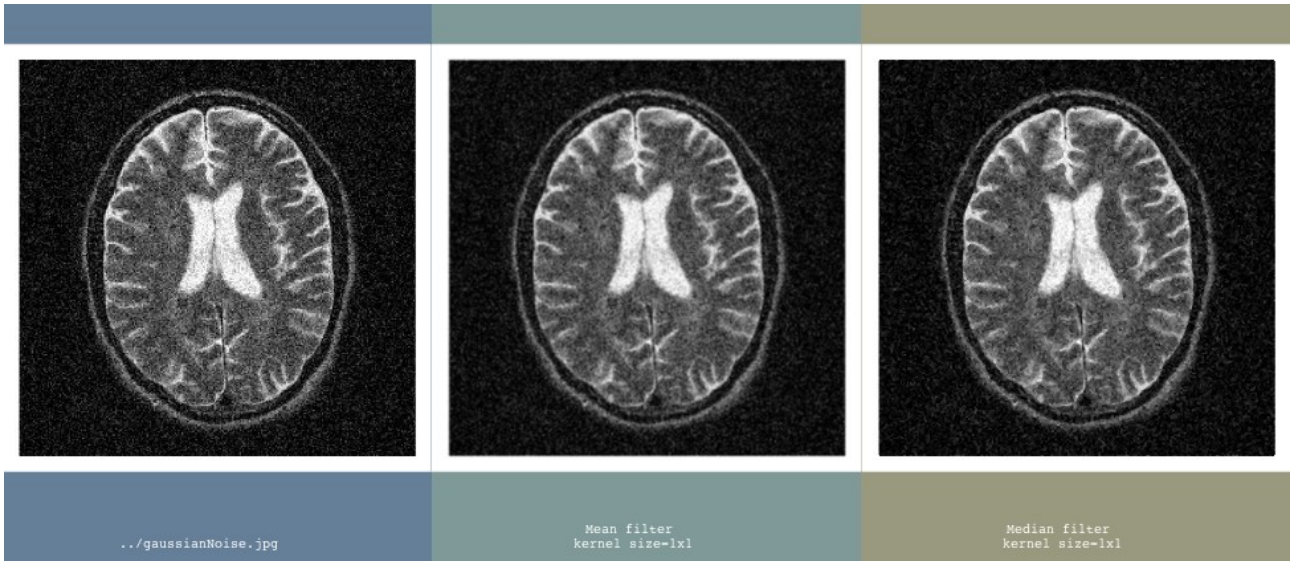
02.1 Experimento 1 Vecindad 3x3:

02.1.1 Imagen 1. saltAndPepperNoise1.jpg



02.1.2 Imagen 2. saltAndPepperNoise2.jpg

02.1.3 Imagen 3. gaussianNoise.jpg.jpg



En este experimento vamos a analizar el comportamiento de ambos algoritmos con un tamaño de vecindad de 3x3 en las tres imágenes propuestas en el enunciado del ejercicio.

Como hemos hablado en clase, el ruido de tipo salt and pepper se caracteriza principalmente por cubrir de forma dispersa toda la imagen con una serie de píxeles blancos y negros. Si entendemos como funcionan los algoritmos que estamos ejecutando, teniendo en cuenta que los valores de los pixel serán el resultado de la media/mediana del valor original de los pixeles considerados como vecinos, al ser un ruido que consiste en pixeles aislados y dispersos los que provocan el efecto, este tipo de algoritmo va a tener una gran eficacia en su eliminación.

Podemos observar en las imágenes de los apartados 02.1.1 y 02.1.2 lo descrito en el párrafo anterior. En cuanto a la diferencia entre ambos algoritmos en estas dos imágenes, parece evidente que el algoritmo basado en la mediana elimina más ruido que el basado en la media, esto es debido a que mientras con el valor medio, un pixel que era ruido blanco sobre fondo negro puede tomar con una alta probabilidad un tono de gris, en el caso de la mediana esto no puede ocurrir ya que, el valor de nuestro pixel de ejemplo solo tendrá, en este caso, la posibilidad de tomar valores que se encuentren en el conjunto de valores de los vecinos de dicho pixel.

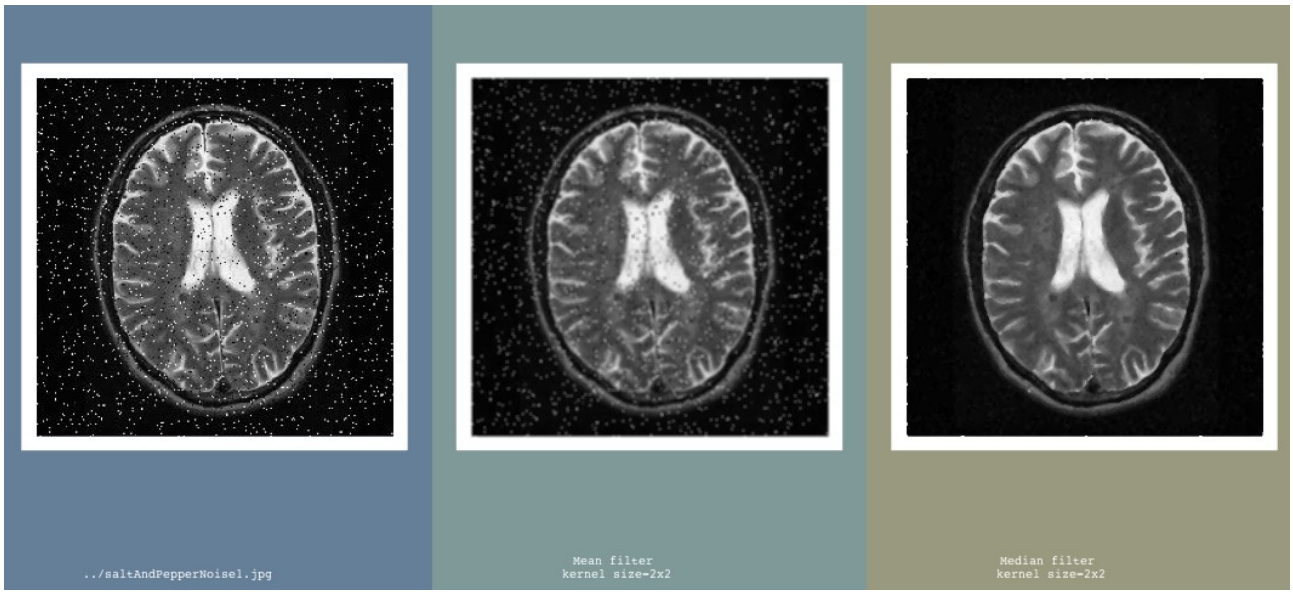
¿Qué ha ocurrido al aplicar estos filtros sobre una imagen con ruido gaussiano?

En este caso el ruido no está distribuido de una manera dispersa y aleatoria si no que sigue una distribución gaussiana, de modo que la probabilidad de que un pixel vecino sea también producto de este ruido es muy alta. Es justo este motivo el que hace que ambos algoritmos se comporten peor con este tipo de ruido que con el de salt and pepper. Teniendo en cuenta esto, cabe destacar que en este caso el algoritmo basado en la mediana, por el mismo motivo que en los párrafos anteriores, funciona algo mejor aunque de manera mucho menos notable que en el resto de imágenes con ruido de tipo salt and pepper.

02.2 Experimento 1 Vecindad 5x5:

Este experimento nos servirá para hacer el efecto del filtro más evidente y poder corroborar la información detallada en el experimento anterior.

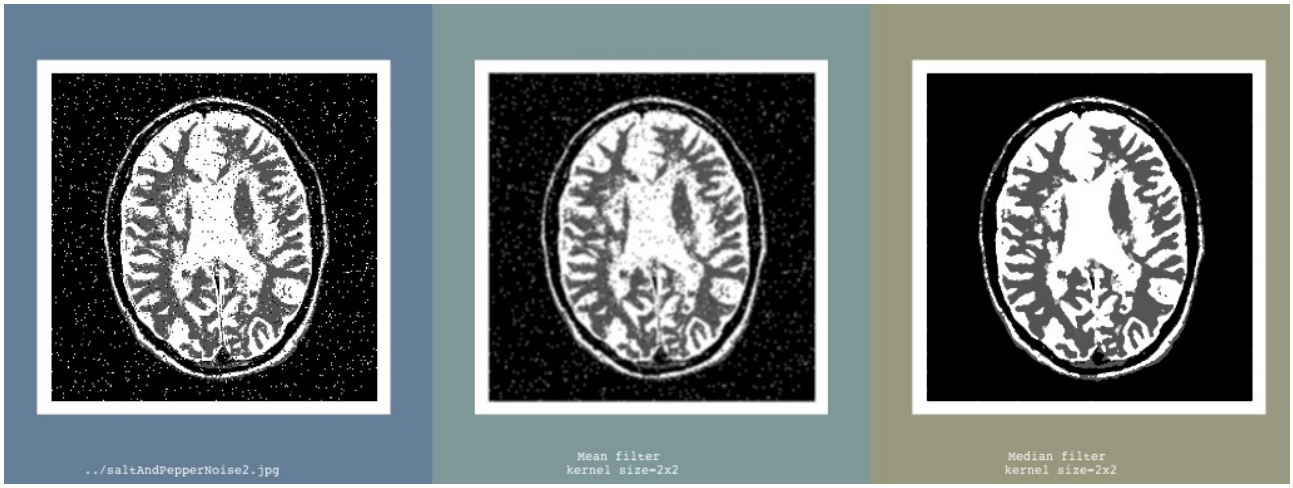
02.2.1 Imagen 1. saltAndPepperNoise1.jpg



Resulta muy evidente que en el filtro que se basa en la media, al incrementar el tamaño de vecindad, si ponemos de ejemplo un pixel blanco dentro del fondo negro, el tono de gris ahora, con respecto al experimento anterior, se acerca más al negro pasando más desapercibido, a pesar de esto, la sensación visual sigue siendo de ruido aunque más atenuado.

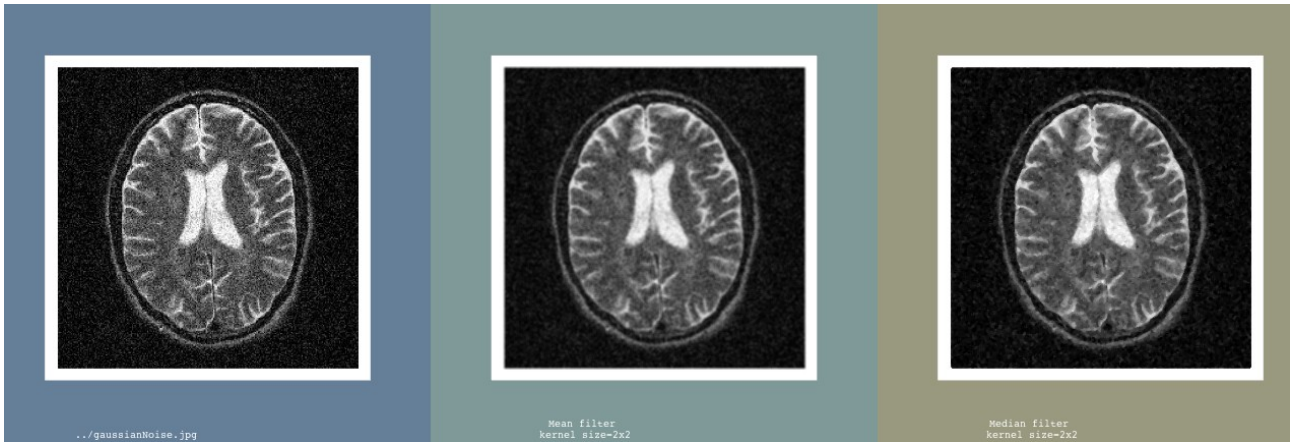
En el caso del filtro basado en la mediana, al aumentar el tamaño de vecindad, hemos aumentado las posibilidades de que el valor que corresponde a la mediana dentro de los valores de los vecinos, en el ejemplo anterior, sea un valor negro. Esto hace que el ruido desaparezca por completo en prácticamente toda la imagen. Si observamos detalladamente, en los bordes quedan algunos pixeles que no se han podido corregir, esto es debido a que los pixeles situados en los bordes, por lógica, tienen menos cantidad de vecinos y esto hace que la probabilidad de la que hablábamos antes disminuya.

02.2.2 Imagen 2. saltAndPepperNoise2.jpg



Ocurre igual que en el apartado anterior por lo que no merece mayor explicación.

02.2.3 Imagen 3. gaussianNoise.jpg.jpg



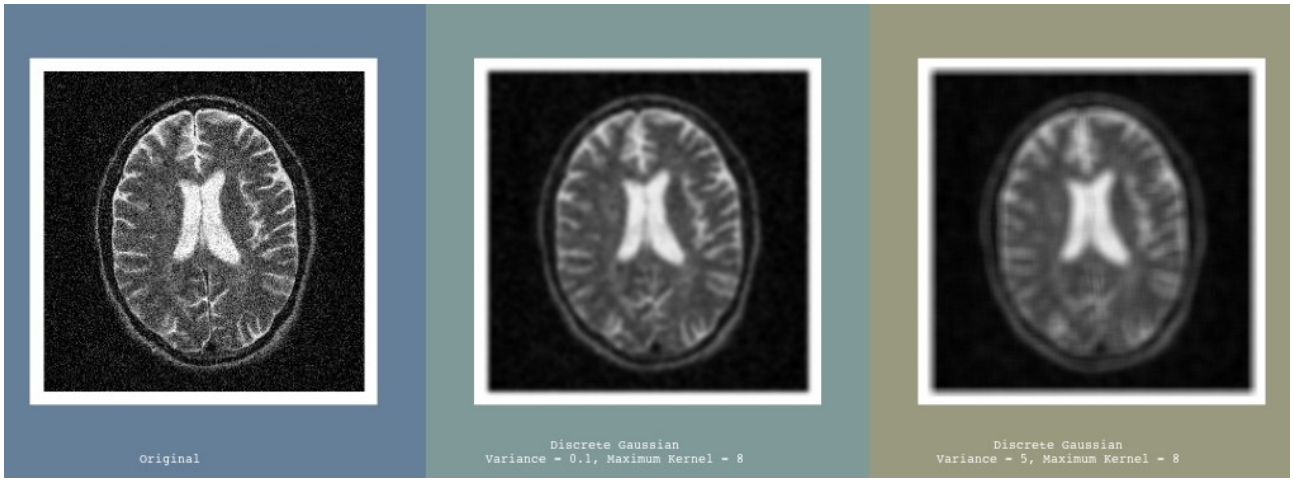
En este último caso vemos un mejora considerable en ambos casos aunque, ninguno de los dos filtros, ha sido capaz de eliminar el ruido por completo.

Task03.
Write a program that reads an image from a file and displays the result of applying the three Gaussian filters presented above. Try the program with the following images: saltAndPepperNoise1.jpg, saltAndPepperNoise2.jpg, and gaussianNoise.jpg. Compare the resulting images with the images from Task 2.

03.1 DiscreteGaussianImageFilter

Como se indica en la práctica y la documentación de ITK podemos tocar dos parámetros clave en este filtro que son la varianza y el tamaño máximo de kernel. Veamos como afectan a la imagen cada uno de ellos.

Para hacer notorio el efecto de la Varianza se han sacado dos imágenes con el mismo valor de kernel máximo y diferentes varianzas, con valores muy lejanos, para ver el efecto que produce:



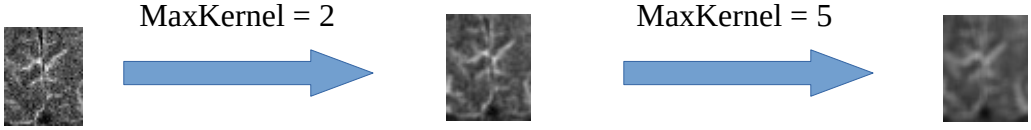
Podemos observar que cuando ponemos una varianza muy pequeña se puede percibir algo más de ruido aunque casi imperceptible, la definición de las lineas queda mas clara. Hemos perdido bastante detalle debido al tamaño máximo de kernel que es bastante alto.

En cuanto a una varianza demasiado alta no tenemos ninguna ganancia clara, si embargo, hemos difuminado la imagen mucho más que en el caso anterior.

Veamos ahora que ocurre cuando jugamos con el tamaño de kernel manteniendo un valor de varianza.

En los tres casos que hemos decidido utilizar para este experimento vemos que al poner un tamaño máximo de kernel no conseguimos eliminar totalmente el ruido aunque mantenemos de una forma bastante clara la definición de la imagen, mejorando dicho ruido. Apenas perdemos detalle teniendo una ganancia significativa en corrección de imagen.

En el segundo caso parece que hemos conseguido eliminar prácticamente por completo el ruido, aunque podemos observar ya una pérdida de detalle importante con respecto al primer caso:



Como vemos, la linea de la sección que presentamos pierde definición de una forma muy notable al crecer el kernel máximo pero también, al suavizar la imagen, elimina más ruido.

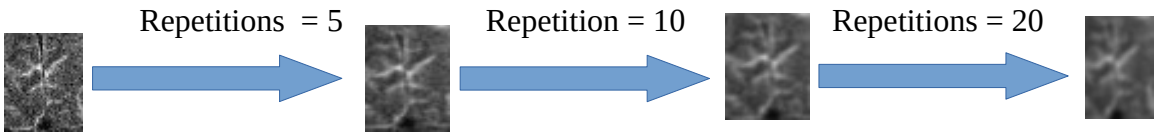
En el caso de un kernel máximo a 10 vemos que el ruido ha desaparecido por completo pero la imagen ha perdido demasiada información y ha quedado con mucho menos detalle.

En definitiva el uso de este filtro requiere de un equilibrio entre la perdida de información y la eliminación del ruido que podemos conseguir ajustando bien los parámetros disponibles.

03.2 BinomialBlurImageFilter

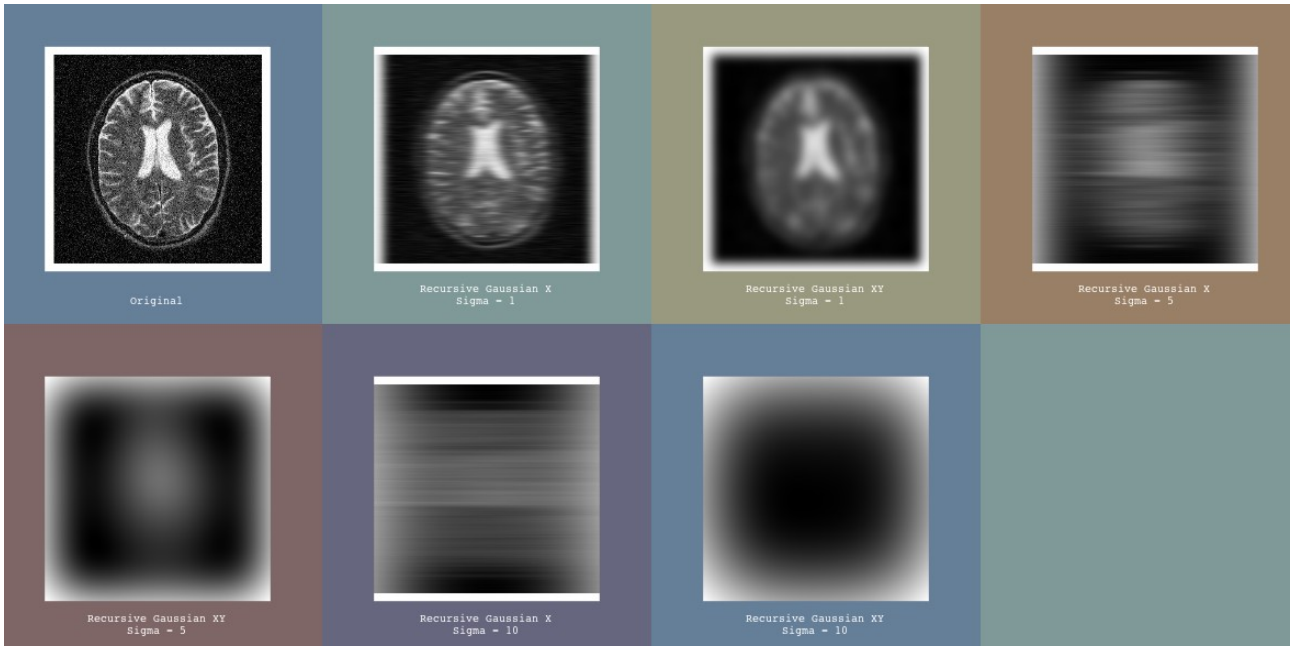
Sabemos que este filtro consiste en calcula un promedio de vecino más cercano a lo largo de cada dimensión de forma iterativa, el número de iteraciones será controlado por el usuario con el parámetro repetitions. Vemos como afecta este parámetro:

Como era de esperar, de forma similar al caso 03.2, tenemos que a mayor número de iteraciones más difuminación de la imagen, lo que implica que perdamos definición y mejoremos el ruido presente en la imagen. Del mismo modo que el caso anterior se debe buscar un equilibrio, veamos el detalle que usamos en el apartado anterior para ver como se pierde el detalle:



03.3 RecursiveGaussianImageFilter

Sabemos que este filtro consiste en resolver el problema relacionado con el ancho de la discretización de kernel gaussiano cuando la desviación estándar es grande, esto lo podrá controlar el usuario dando un valor al parámetro sigma. Vemos como afecta este parámetro:



Se ha programado el filtro en dos partes, de forma que primero actuamos sobre la dimensión X y después sobre la dimensión Y, de esta forma agilizamos el proceso de filtrado y podemos ver como primero se aplica una mascara en X y después en XY.

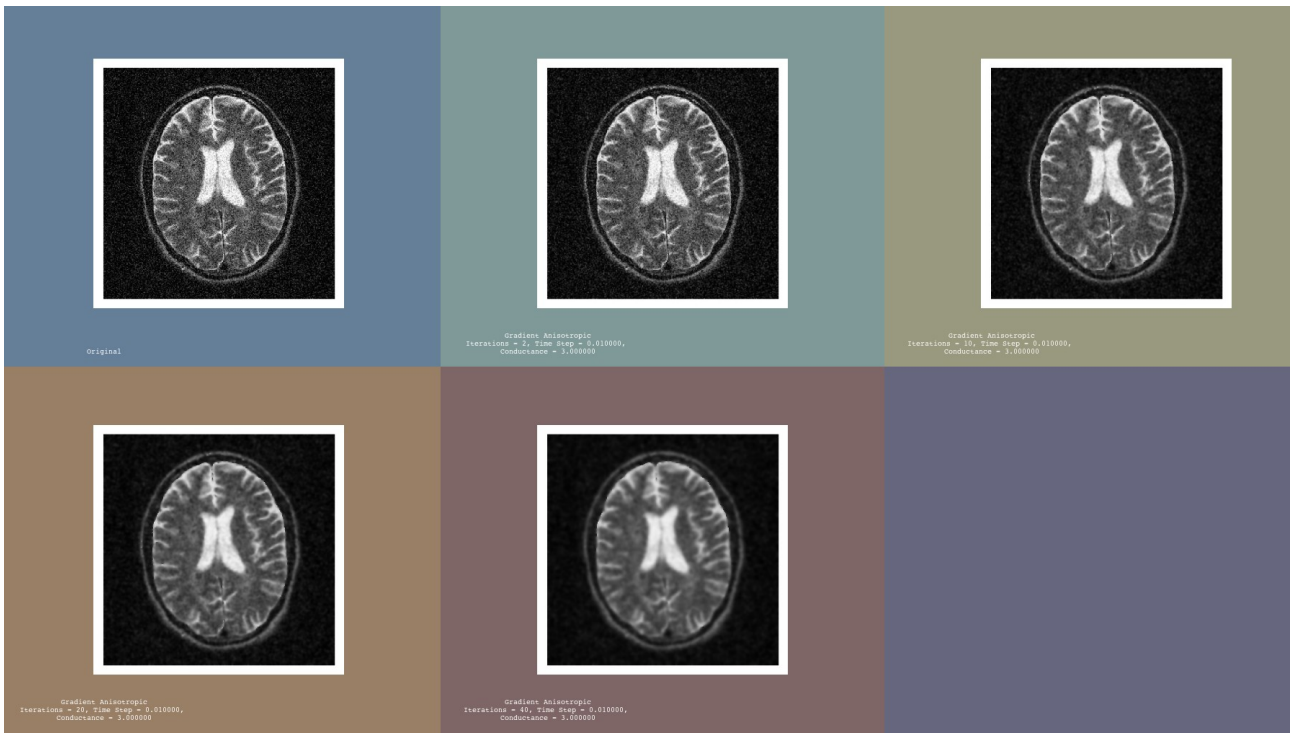
En cuanto al efecto que obtenemos, como se puede ver en la imagen superior, estamos ante un suavizado muy potente, a medida que aumentamos el valor de sigma vamos perdiendo definición hasta quedar en una imagen en la que es imposible distinguir ninguna forma que se corresponda con la imagen original.

Task04.
Write a program that reads an image from a file and displays the result of applying the three edge-preserving filters presented above. Try the program with the following images: saltAndPepperNoise1.jpg , saltAndPepperNoise2.jpg , and gaussianNoise.jpg . Compare the resulting images with the images from tasks Task 2 y Task 3.

04.1 GradientAnisotropicDiffusionImageFilter

Este algoritmo nos permite manipular tres parámetros clave para su utilización, vamos a estudiar como influye en la visualización de la imagen cada uno de ellos.

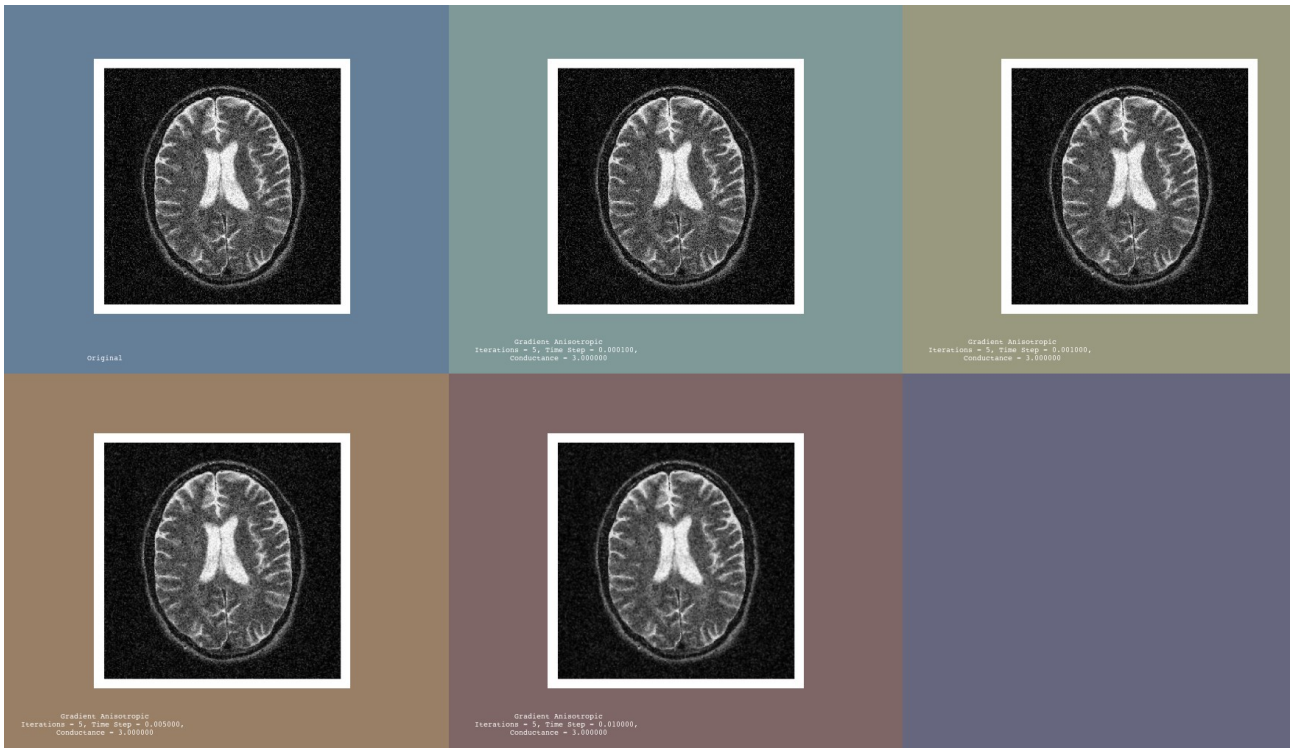
04.1.1 gradientAnisotropicNumberOfIterations



Los valores de este parámetro de izquierda a derecha son [2,10,20,40], la primera imagen se corresponde con la imagen original y el resto de parámetros se mantienen constantes.

Como se puede observar, a medida que el valor de este parámetro crece vamos difuminando la imagen, se mantienen bastante bien los bordes y parece que difuminamos el contenido eliminando el ruido. En los últimos casos los detalles en bordes empiezan a perderse pero de una forma mucho más sutil que el resto de filtros aplicados hasta el momento.

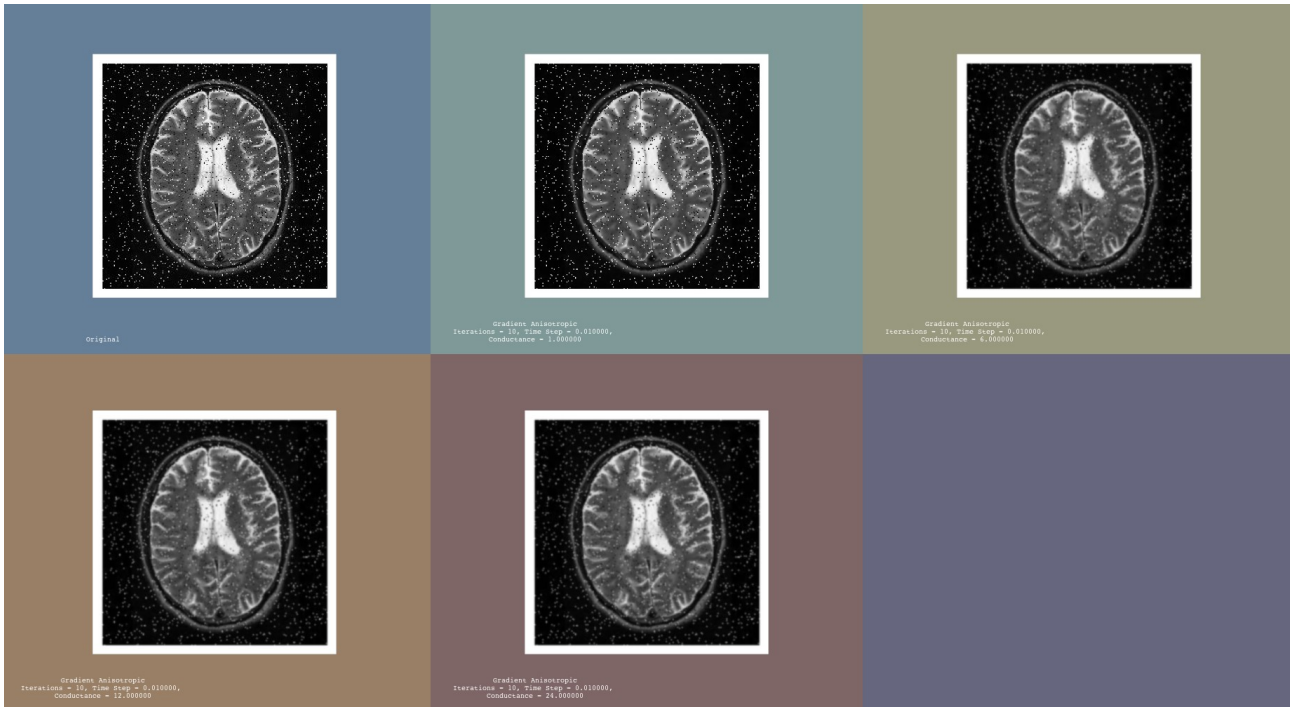
04.1.2 gradientAnisotropicTimeStep



Los valores de este parámetro de izquierda a derecha son [0.0001,0.001,0.005,0.01], la primera imagen se corresponde con la imagen original y el resto de parámetros se mantienen constantes.

Como se puede observar, a medida que el valor de este parámetro crece vamos marcando los bordes y eliminando ruido gaussiano. En los últimos casos los detalles en bordes empiezan a perderse pero de una forma mucho más sutil que el resto de filtros aplicados hasta el momento.

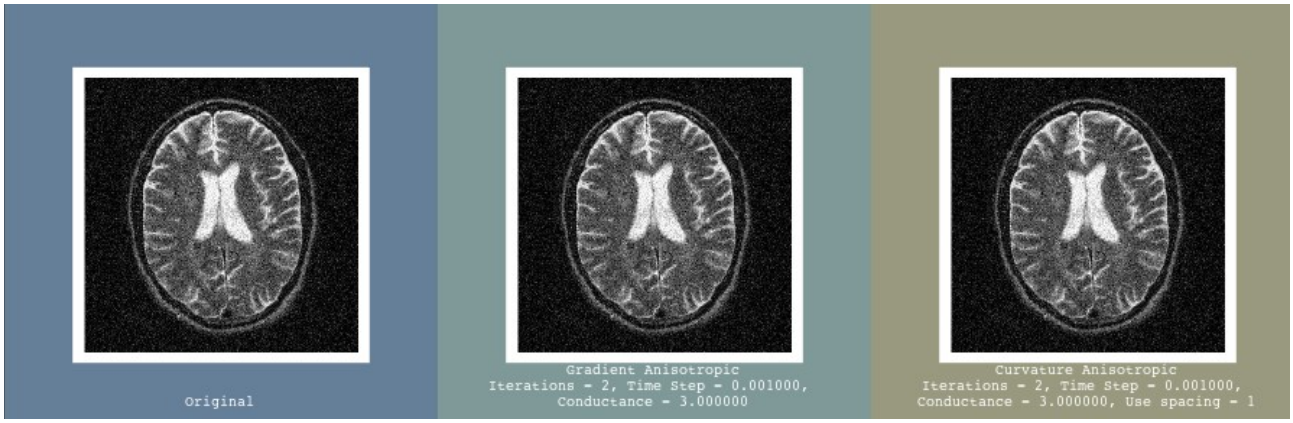
04.1.3 gradientAnistropicConductance



En este caso se ha tomado la decisión de usar una imagen con ruido del tipo salt and pepper porque resulta más ilustrativo. Podemos observar que los valores que va tomando la variable en las diferentes visualizaciones son [1,6,12,24]. Es evidente que a medida que va aumentando la conductancia se consigue difuminar el ruido sin perder detalle en los bordes.

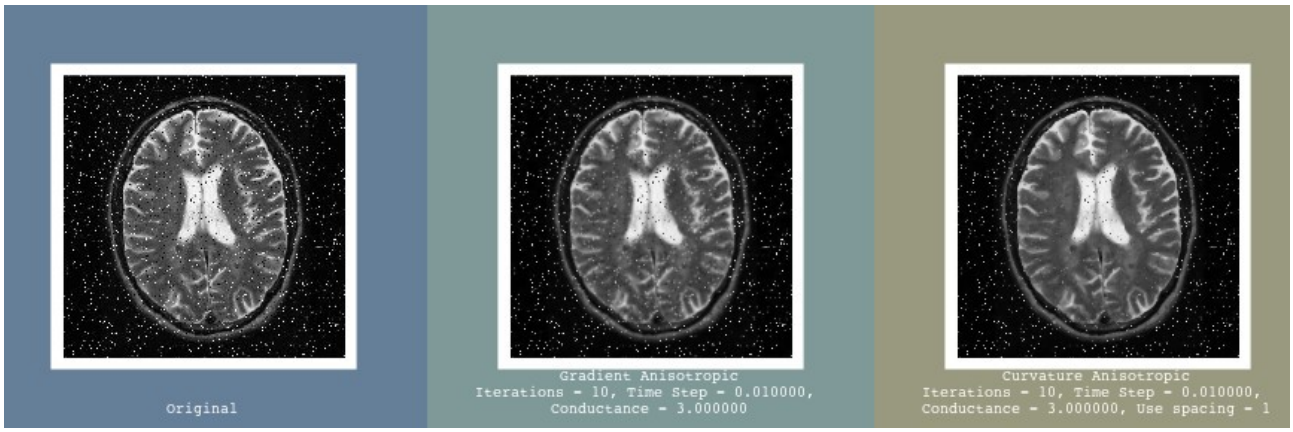
04.2 GradientAnisotropicDiffusionImageFilter

Ya hemos visto como afectan los parámetros por separado en el ejercicio anterior y no parece de mucho interés hacer el mismo proceso en este caso. Se considera por lo tanto que resulta más interesante intentar ver la diferencia entre el filtro anterior y este con los mismos valores en los parámetros:



Aunque no de forma muy evidente, parece que se puede apreciar cierto aumento en el contraste de bordes en la última imagen con respecto a la del filtro anterior.

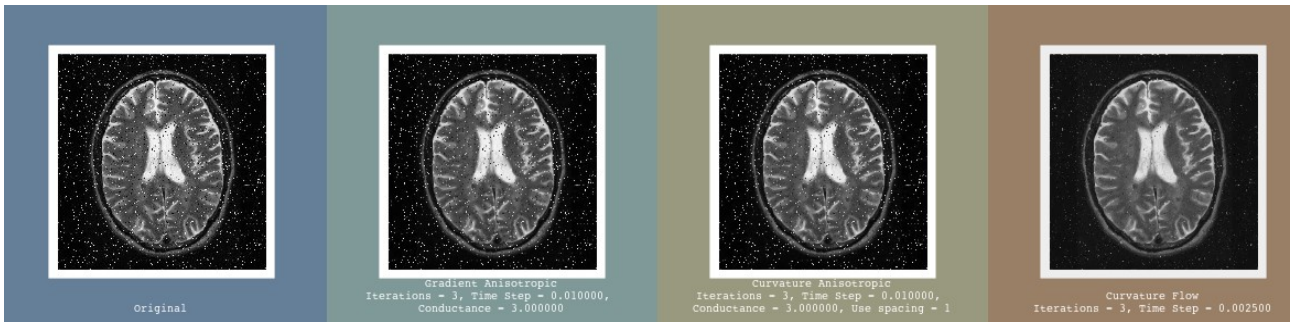
¿Qué ocurrirá con una imagen con ruido de tipo salt and pepper?



En este caso si se ve claramente como este último filtro no solo elimina más ruido si no que, además, da una mejor definición de bordes.

04.3 CurvatureFlowImageFilter

En este caso volvemos a coincidir en que será más ilustrativo comparar los tres filtros de este task y ver como se comporta cada algoritmo con parámetros similares.



Parece evidente la capacidad de este último filtro para eliminar ruido con solo dos iteraciones es bastante mayor a los otros tres. Los bordes quedan perfectamente marcados, el ruido es mínimo y la

definición en bordes es muy alta. Hemos conseguido con este filtro eliminar ruido con una pérdida de información mínima.