Digital Image Processing:

Noise Reduction of Synthetic Aperture Radar (SAR) and Ultrasound Images

Thursday 28th November 2019 Candidate Number: 12627

Table of Contents

Table of Figures

1 Introduction

Often, images generated for scientifict, geological and medical purposes can be found to be noisy and unclear. For the features in these images to be analysed, be it by a human's visual system or by using a feature extraction algorhithms, the noise within the image needs to be reduced to allow features within them to be made more prominent.

By having a look at the images, Figure 1 and Figure 2, image enhancement techniques can start to be applied to reduce their noise.

Figure 1: SAR image of land in the Marlborough Sounds, New Zealand, 'NZjers1.png' [4]

Figure 2: Ultrasound image of a foetus, 'foetus.png' [4]

About Python implementation

In this project, the choice of programming language was 'Python 3'. This was for learning reasons as well as the possibility for testing the filters produced with the powerful 'OpenCV' library, referred to as 'cv2' in the code, made for image processing and computer vision applications.

The array manipulation library of choice was 'Numpy' due to its widespread use and comprehensive documentation. Along with this, the 'Math' library was useful for general mathematical expressions and values such as 'π' or 'e'. The 'Datetime' libaray was also used a bit for timing some of the filters that were created.

Necessary Libraries to run DIPFilters.py

```
import cv2
import numpy as np
import math
from datetime import datetime
```

At the bottom of the script contains an '`if __name__ == "__main__":`' section where the filter functions are implemented and tested.

In this section, the image can be read using the '`cv2.imread`':

```
    #Reading image files for image processing
    image = cv2.imread('NZjers1.png',  cv2.IMREAD_GRAYSCALE )
```

Later in the section the filter of choise can be tested by commenting or uncommenting the sections that need testing.

Commented Median Filter, this will not run the median filter

```
    # ##-- MEDIAN FILTER -- NON LINEAR--##
    # n = 9
    # imageMed = medianFilter(image, n)
    # print('median complete')
    # cv2.imshow('median image', imageMed)
```

Uncommented Median Filter, this will run the median filter

```
    ##-- MEDIAN FILTER -- NON LINEAR--##
    n = 9
    imageMed = medianFilter(image, n)
    print('median complete')
    cv2.imshow('median image', imageMed)
```

2 Image Enhancement [1]

There are different of image enhancement techniques that can be applied to images. These are broken down into 2 main methods, 'Point Operators' and 'Group Operators'.

2.1 Point Operators

Point operators tend to be simple implementation of image enhancement techniques. They determine the change in the pixel value by comparing itself to all the other pixels in the image and not buy comparing it to the neighborhood around it. The only point operator that is applied within this report is the Histogram Normalisation operation which extends an image from a limited grayscale range to a the full greyscale range.

2.2 Group Operators

Group operators change the value of pixels by comparing it to its neighbouring pixels, termed the region of interest (ROI). This is normally done by convoluting a window with the ROI to determine the final value of that pixel and by averaging or carefully selecting terms within the convoluted array, an output value can be generated to replace the given pixel.

The different group operators can be categorised as 'Linear Filters' or 'Non-Linear Filters'.

2.2.1 Linear Filters

Linear Filters are a simple type of filter that takes the region of interest and convolutes it with the desired window. An example of a linear filter can be seen in Figure 3 where the yellow pixel is the pixel in the ROI to be changed.

ROI				Mean				Output
50	80	75		1/9	1/9	1/9		
43	2	43	*	1/9	1/9	1/9	=	48
55	23	61		1/9	1/9	1/9		

Figure 3: Example of Linear Filtering (Mean Filter)

Linear filters have the benefit of being quite simple to apply and perform quite quickly. However, the image they tend to produce a more blurred output than that of non linear filters. This is because for every region of interest they perform the same and thus do not have the intelligence to filter out pixels that may be outliers in the region, ie noise. So an

amount of noise will always be present in the image and the points that are not noise will always be slightly blurred as well.

The different Linear Filters that will be explored include:

1- Mean Filter
2- Gaussian Filter

2.2.2 Non-Linear Filters

Non Linear Filters have a very large range of different types of implementations and thus some are quite simple and quick to run and other are comlex and long to run. Typtically, they cause the ROI window to be modified in some way such as reorganising it then evaluating the reordered ROI. Most of the Non-Linear filters that have been explored in this report are linked with calculating some kind of median figure. An example of a non linear filter can be seen in Figure 4 where the yellow pixel is the pixel in the ROI to be changed.

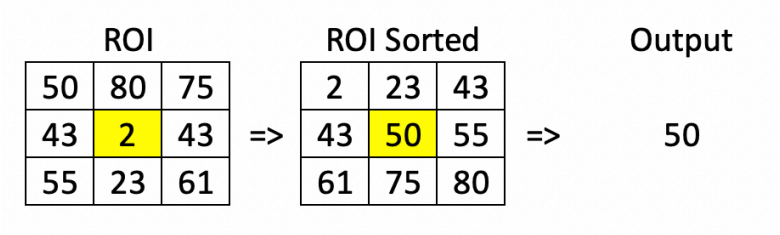

Figure 4: Example of Non- Linear Filtering (Median Filter)

The different Non Linear Filters that will be explored include:

1- Median Filter
2- Weighted Median Filter
3- Truncated Median Filter
4- Adaptive Median Filter
5- Trimmed Mean Filter

2.3 Analysis Techiniques

The analysis techniques will include visually analysing the image to see if it is pleasing to look at along with edge detection techniques to see how a computer would perceive it. These will allow evaluation of the filtering techniques to see what parameters produces the best images.

The edge detection technique being used is the Canny Edge Detector for its accuracy and speed compared to many other edge detectors. [2]

Canny Edge Detector Implementation:

```
edgeOrig = cv2.Canny(image, threshold1, threshold2)
```

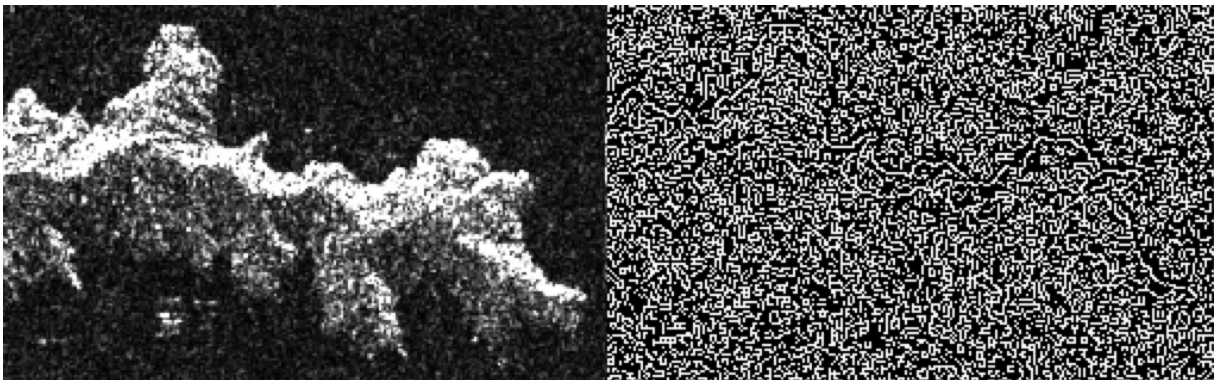

Figure 5: Canny edge detection of the NZ Land

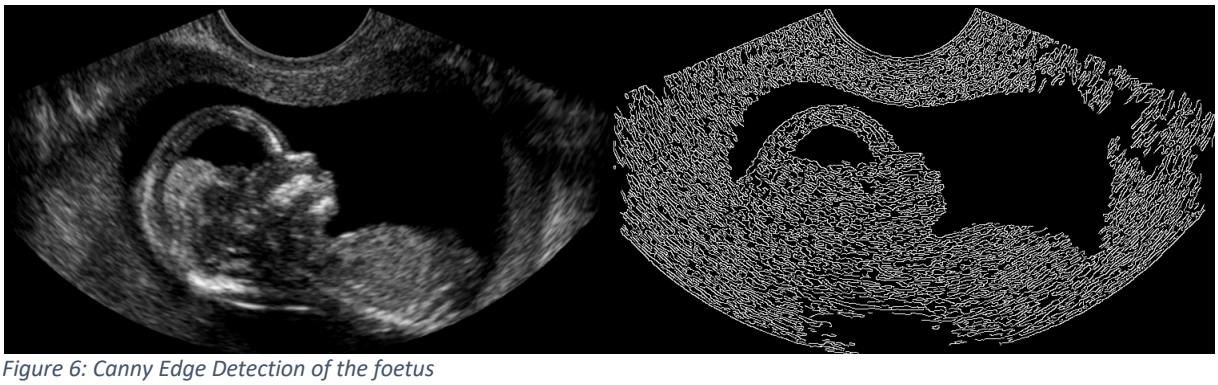

Figure 6: Canny Edge Detection of the foetus

Figures Figure 5 and Figure 6 will be the baseline edge detection that the filters will be compared against. It is easy to notice that the noise in the image causes the edge detector to pick up far more edges than it should in the image. The filters will be applied to mitigate the false edges caused by noise in the image.

3 Image Padding

A function was created to apply image padding to the input image. Image padding is the process of adding a few extra layers at each end of the image. The amound of padding determined by the width of the filtering window, ie if the filtering window was 3x3, there would be one layer of padding, if the window was 9x9, there would be four layers of padding.

Padding is applied to make sure that the filters have something to compare against on the first arrays of horizontal and/or vertical pixels. The accuracy of these values are not very important as the edges of the window is not where the interesting sections of the image are.

The imagePadding function created replicates the border of the image by the rounded down value of half the window size. The cv2.copyMakeBorder function is then used to copy and replicate the edge values to its opposite padded position. [3]

imagePadding Function Code:

```
def imagePaddingFunc(image, window):
    #Image padding function to extend the outer edges of the image
    #to the same values as the first few pixels along the edges

    #defining the height and width of image and filtering window
    wW,  wH = window.shape[:2]

    #padding the  edges of the image to allow processing to
    #start at the beginning of the image (needs to be int)
    padding = ((wW -1) // 2)

    paddedImage = cv2.copyMakeBorder(image, padding, padding, padding, padding,
cv2.BORDER_REPLICATE)

    #initialising the output image matrix
    output = np.zeros((imH, imW), dtype="float32")

    return output, padding, paddedImage
```

The filtering window size will be applied from $(x, y - padding)$ positions to $(x, y + padding)$ positions hence why padding appeards as an output of the function. This will first be seen in the slideMean function introduced within the Linear Filters section.

4 Linear Filters

In this section, the code that has been used to genenerate the different linear filters will be explained, as well as an analysis of the images that are produced.

4.1 Mean Filter Function

A mean filter is a simplistic method to apply smoothing across an image. It reduces the intensity of a given pixel value by bringing it closer to the the value of the pixels surrounding it.

Equation of the mean along and down a ROI window: [1]

$$\sum_i \sum_j w_{i,j} = 1$$

By applying this mean equation to an nxn filtering window, a mean for a ROI can be evaluated and outputted to the pixel position. An example of how this works has already been shown in Figure 3. This process repeats for every pixel along and down the image.

The meanFilter function below generates the window to be applied for mean filtering then sends the window to the slideMean function which convolutes the averaging window with the ROI window.

meanFilter Function Code:

```
def meanFilter(image, n):
    #creating a window of matrix nxn
    typeWindow = np.ones((n,n), dtype=np.int8)

    #creating a scalar to divide te overall mean window to bering intensity
    #back into the relevant range of 0 to 255
    elem = typeWindow.sum()

    #Image Padding
    output, padding, paddedImage = imagePaddingFunc(image, typeWindow)

    #apply slide mean function to find the mean of every ROI along the image
    output  = slideMean(output,paddedImage, typeWindow, padding, imH, imW, elem)

    # return the output image
    output = np.array(output, dtype="uint8")

    return output
```

The slideMean function applies the filtering procedure for all the x and y axis values.

slideMean Function Code:

```
def slideMean (output, paddedImage, typeWindow, padding, imH, imW, elem):
    #For sliding down Y in the image as all of X completes
    # this is applies for filters that use a genereric mean filterin style

    for y in np.arange(padding, imH+padding):
        #For sliding across X in the image
        for x in np.arange(padding, imW +padding):

            #getting the Region of Interest of an image by creating the window
            # to apply the filter to
            roi = paddedImage[y-padding:y+padding+1, x-padding:x+padding+1]

            #applying the element wise multiplication of the filter to the image
            weightWindow = (roi * typeWindow)

            #adding  all the elements up and dividing my nuber of elemets to find
            #the mean
            mean = weightWindow.sum()/elem

            output[y - padding, x - padding] = mean

    output = np.array(output, dtype="uint8")
    return  output
```

Now these functions can be implements to generate mean filtered images. For different window sizes to see how the output looks. This can be seen in Figure 7.

As the value of the window size increases, the image becomes considerably more blurry, this is because the noise is not being removed but is more smeared across the image. This calse causes the true edges to bleed and thus generates a blurry image.
However, it can be seen that the edge detector has removed a lot of the false edges in the background but has still retained a lot of false edges on the land in the image.

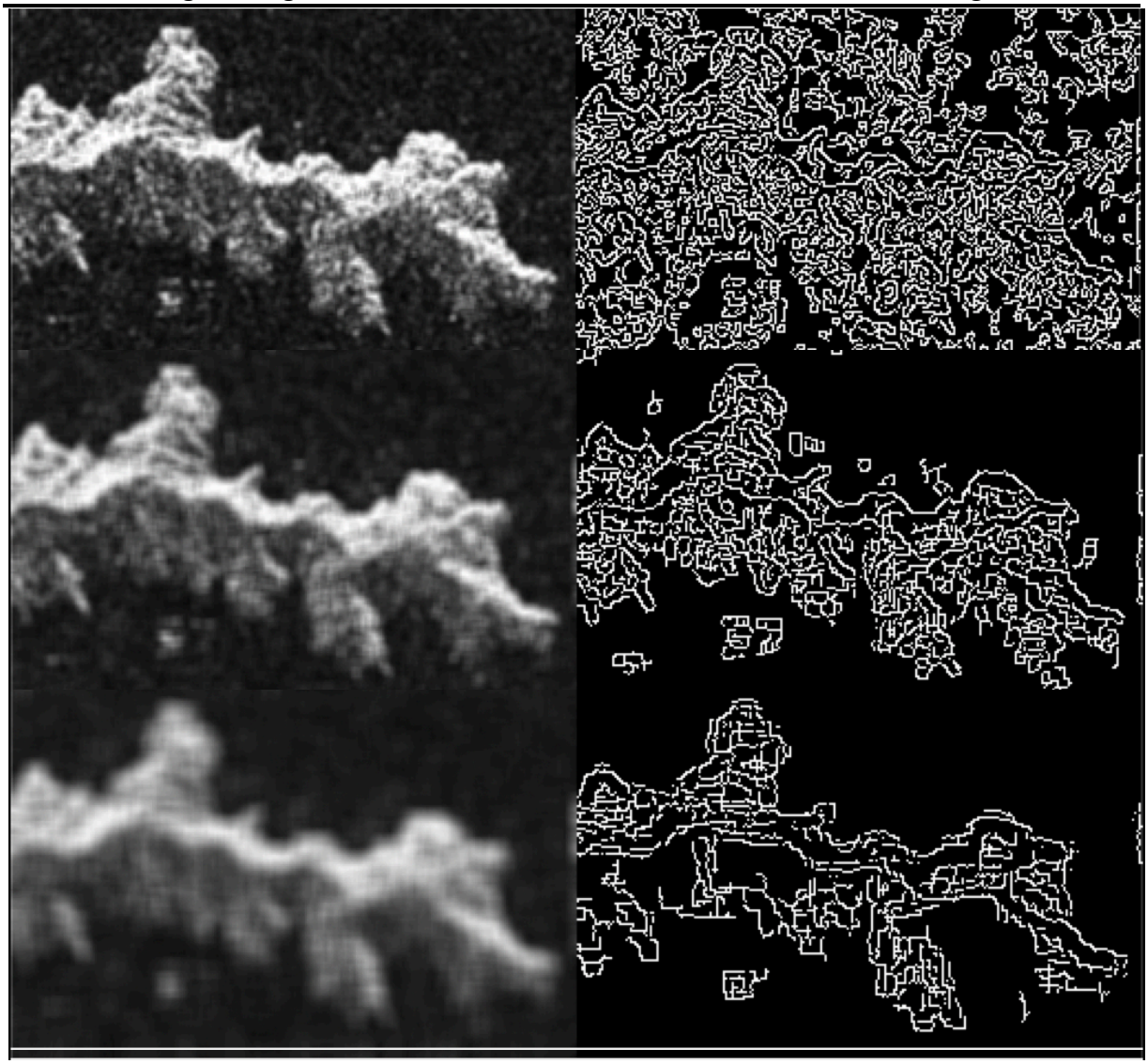

Figure 7: Mean Filtering, from top to bottom n = 3, n = 5, n= 9

The mean filter is a simple and quick filter to operate, however the image outputs of the filter do not provide good images for either visual analysis or edge detection. This should not be a primary filter used for image analysis.

4.2 Gaussinan Mean Filter

A gaussian filter is a type of lowpass linear filter is that is is used as a standard for image smoothing. It works based on the gasussian function: [4]

$$g(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Where the position of x and y in the equation are the distance from the centre of the filtering window.

Gaussian Window Generation Code:

```
#making the gaussian window
    for j in range (0,n):
        for i  in range (0,n):
            typeWindow[j][i] = (1/elem) * np.exp( -0.5*( (midpos-(i))**2 + (midpos-(j))**2)/(sigma**2))
```

It was found that the output of this image had a variation of intensity depending on the kernel size used as well as the σ value that was used. Due to this Histogram normalisation was also implemented to bring back the full greyscale range into the image.

histogramNorm Function Code:

```
def histogramNorm(output, offset, padding, imH, imW):
    # rescale the output image to make sure the range of the image extends
    # to a gull greyscale range of [0, 255]

    oMax = np.amax(output)  #max value of o/p image
    oMin = np.amin(output)  #min value of o/p image
    for y in np.arange(padding, imH+padding):
        for x in np.arange(padding, imW +padding):
            output[y-padding, x-padding] = (output[y-padding, x-padding]-oMin)*255/(oMax-oMin) + offset
    return output
```

The overall code of a gaussian filter is very similar to that of a mean filter, the additional sections of this include generating the gaussian window and histogram normalisation of the output image.

gaussianFilter Function Code:

```
def gaussianFilter(image, n, sigma):
    #similar to a mean filter but uses a gaussian mask where the centre values
    #have higher intensity the outer values

    typeWindow = np.ones((n,n), dtype=np.float)
    output, padding, paddedImage = imagePaddingFunc(image, typeWindow)
    midpos = math.ceil(n/2) - 1
    elem = 2 * np.pi * (sigma**2)

    #making the gaussian window
    for j in range (0,n):
        for i  in range (0,n):
            typeWindow[j][i] = (1/elem) * np.exp( -0.5*( (midpos-(i))**2 + (midpos-
(j))**2)/(sigma**2))

    output  = slideMean(output, paddedImage, typeWindow ,padding, imH, imW, 1)

    #applying histogram normalisation, as sigma increases, overall image darkens
    output = histogramNorm(output, 0)

    #return the output image
    return output
```

A gaussiean filter does a great job of reducing the noise in the background of the image and also allows the true edges to be very well preserved, unlike the mean filter. This is due to the filtering window having a variation of values, where the centre point is the most prominement and as it moves away from this the the value is reduced. This change in value is determined by the variable σ.

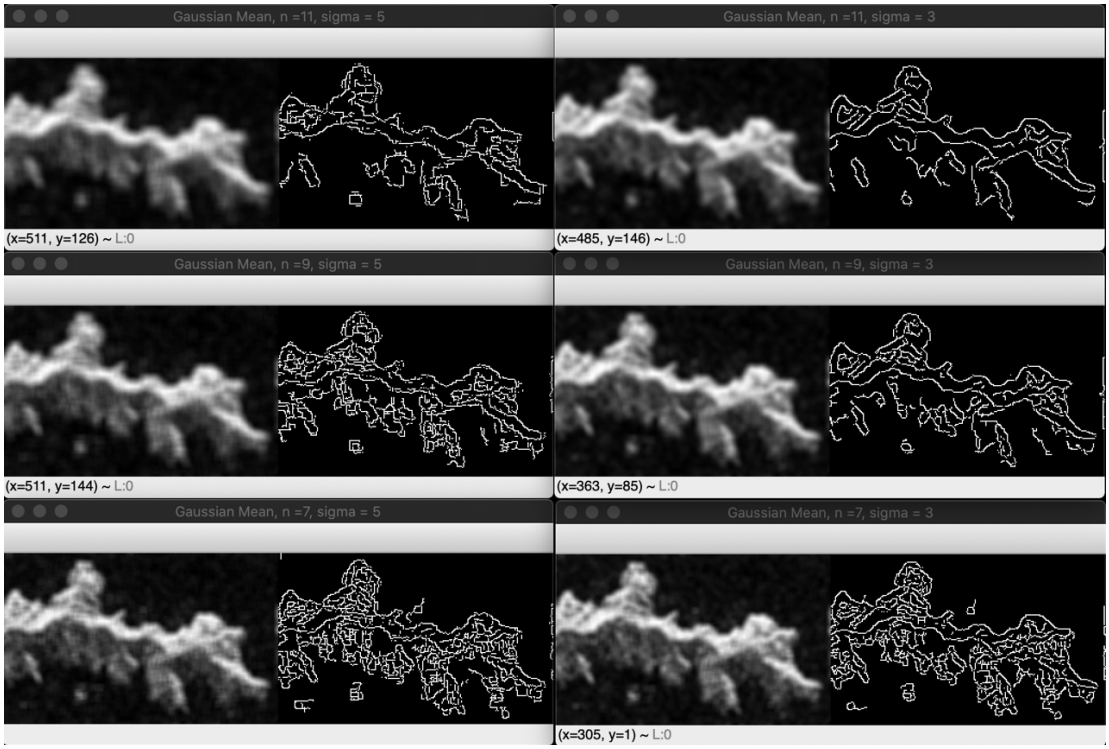

Figure 8: Gaussian Filter for a range of values of n and sigma σ

Through extensive testing, which can be found in Appendix A it was found that as σ increase for a given window size, background noise was usually reduced but forground noise seemed to increase, as indicated by the canny edge detector, by striking a balance between n and σ, the canny edge detector could find all the prominent feature of the image with and fitler away the less prominent ones. These were found to be $n = 9$ and $\sigma = 6.5$.

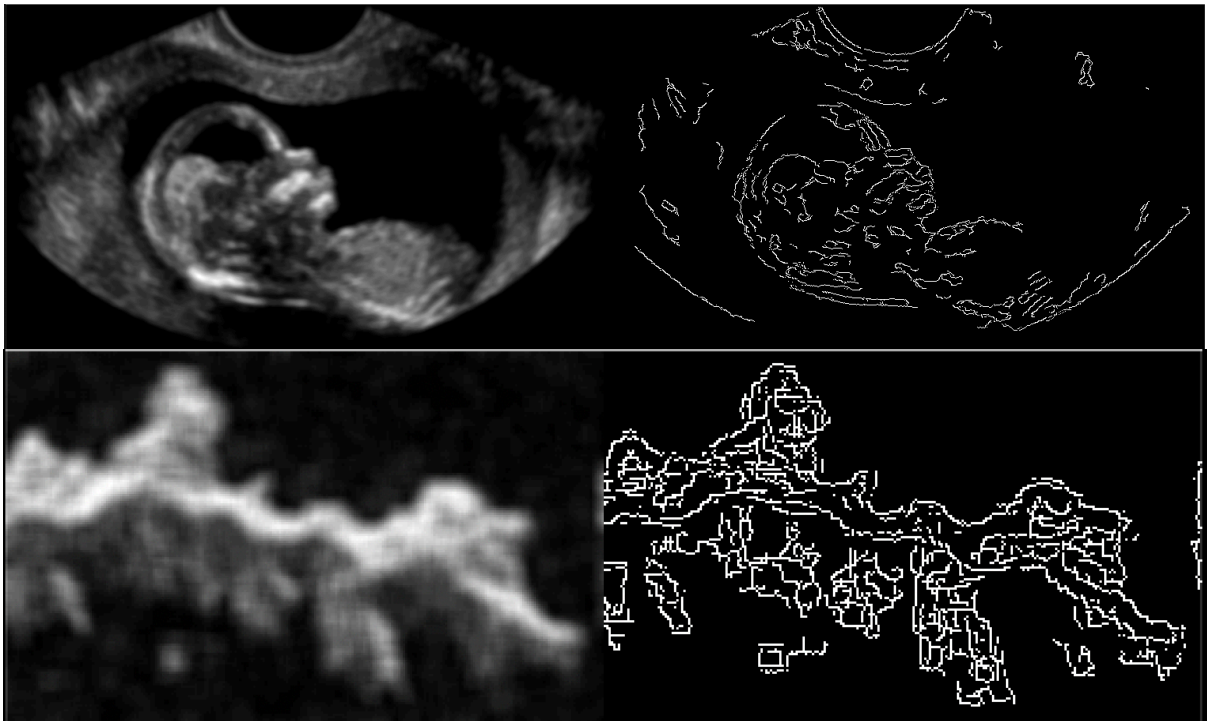

Figure 9: Gaussian Filter on the foetus and Land in NZ respectively, n = 9, sigma = 6.5

However it can be seen that for the Land image, although the edge detection is okay, the image appears quite blurry visually, this is due to the smaller size of this image with a larger n value.

The gaussian filter performs a little better than the mean filter, and can be more fine tuned to output a type of image clarity that a user would prefer for either visual or edge detection analysis, however the overall performance of this filter is still not great as larger window sizes cause a lot of image smearing and larger σ values introduce unwanted noise. A low intensity gaussian filter can be an good initial filter across an image to reduce noise but should not be relied on for generating clearer overall images.

5 Non-Linear Function

In this section, the code that has been used to genenerate the different non-linear filters will be explained as well as an analysis of the images that are produced.

5.1 Median Filter [1]

A median filter sorts the values of an nxn ROI around a pixel from the smallest to the largest value. The filter then finds the value of the central position within the ROI. This is the value that is then outputted into the current pixel positions.

medianFilter Function Code:

```
def medianFilter(image,n):
    typeWindow = np.ones((n,n), dtype=np.int8)
    output, padding, paddedImage = imagePaddingFunc(image, typeWindow)
    output  = slideMedian(output,paddedImage, typeWindow, padding, imH, imW)

    # return the output image
    output = np.array(output, dtype="uint8")
    return output
```

slideMedian Function Code:

```
def slideMedian(output, paddedImage, typeWindow, padding, imH, imW):
    #For sliding across X and down Y in the image for all pixels
    # this is applies for filters that use a genereric median filtering style

    for y in np.arange(padding, imH+padding):
        #For sliding across X in the image
        for x in np.arange(padding, imW +padding):
            #getting the Region of Interest of an image by creating the window
            # to apply the filter to
            roi = paddedImage[y-padding:y+padding+1, x-padding:x+padding+1]

            #Sort from lowest to highest value in the array
            #Clipping and additional convolution is for the weighted median
            roiSorted = np.clip(np.sort(roi,axis=None)*typeWindow.flatten(),0,255)

            median = math.ceil(len(roiSorted)/2) - 1

            output[y - padding, x - padding] = roiSorted[median]
    output = np.array(output, dtype="uint8")
    return output
```

Now the median filter can be applied to see how it affects the input image.

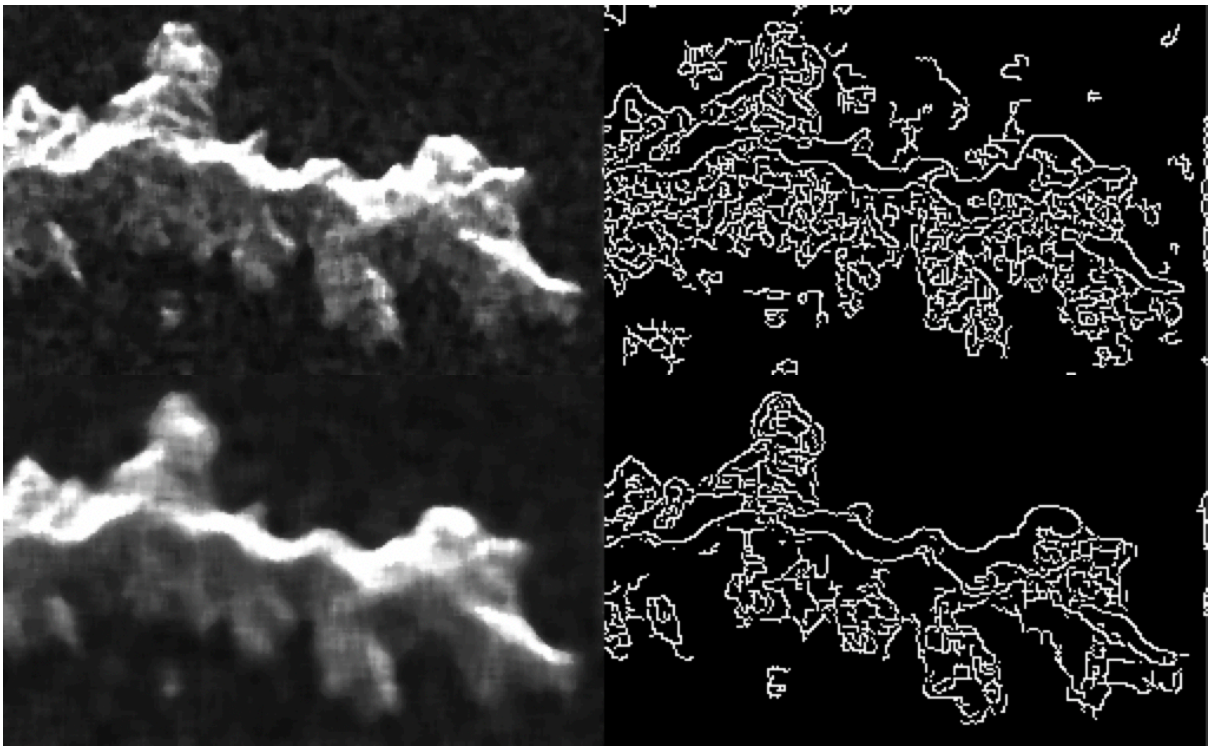

Figure 10: Median Filtering, from top to bottom n = 5, n= 9

For a lower value of n a lot of noise is still retained in the image but features of the land are also retained making it a visually appealing image but the edge detection picks up a lot of areas that are noisy. At higher values of n the the features on the land begin to face away and the darker sections of the foreground begins to blend into the background as well. This image is slightly better for edge detection, however the bottom of the land has a lot of missing edges.

The median does a good job of smoothing the image whilst retaining the prominent edges of the image without taking a considerable amount of computing power. A compromise between features and noise is made when selecting the window size.

5.2 Weighted Median Filter [1]

Applies similar logic to the regular median filter, however, once the median value is found, it is multiplied by a weight.

weightedMedian Function Code: (difference from the medianFilter Function is highlighted)

```
def weightedMedianFilter(image,n, W):
    typeWindow = np.ones((n,n), dtype=np.int8)
    typeWindow[math.floor(n/2)][math.floor(n/2)] = W
    output, padding, paddedImage = imagePaddingFunc(image, typeWindow)
    output  = slideMedian(output,paddedImage, typeWindow, padding, imH, imW)
    output = histogramNorm(output, 0, padding, imH, imW)

    # return the output image
    output = np.array(output, dtype="uint8")
    return output
```

Clipping in slideMedian function:

```
roiSorted = np.clip(np.sort(roi, axis=None) * typeWindow.flatten(),0,255)
```

This is clipped to prevent the value of the median reaching above 255 to retain the uint8 formatting.

Now the median filter can be applied to see how it affects the input image.

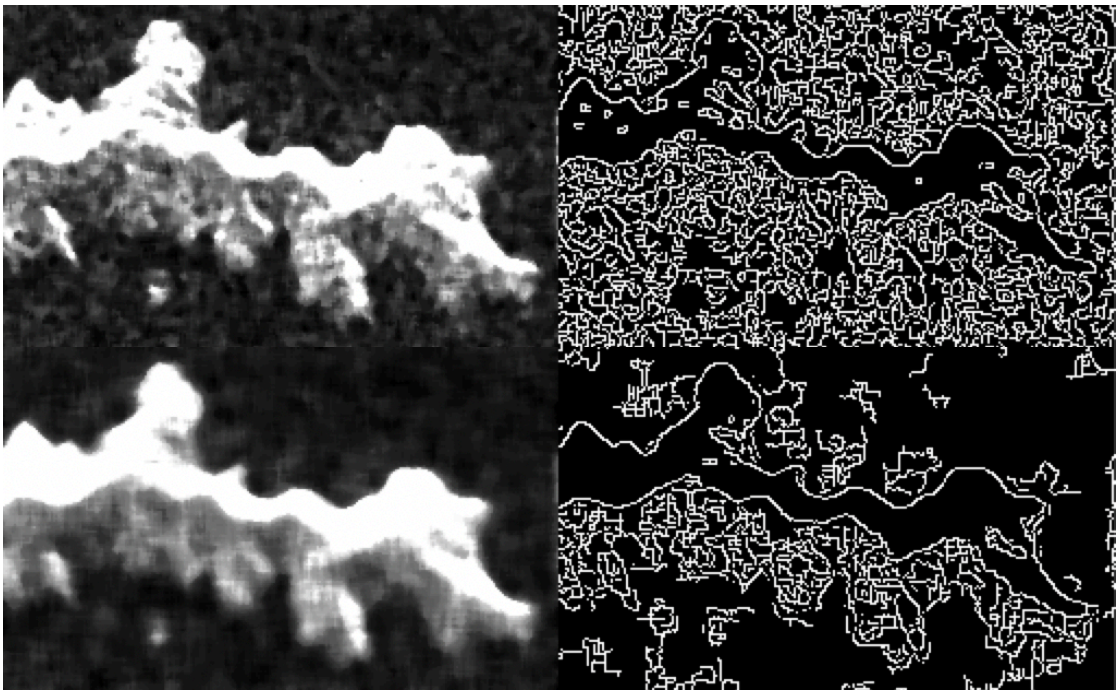

Figure 11: Weighted Median Filter W = 2, from top to bottom n = 5, n= 9, on NZjers1

It can be seen in Figure 11 that the weighting causes the details to begin to blow out. This it not ideal for an image with lots of noise in the background. However in an image with low background noise, such as the foetus imge, this can be very beneficial as the portion of the foetus can be blown out a lot making it much easier to detect on the canny edge detector.

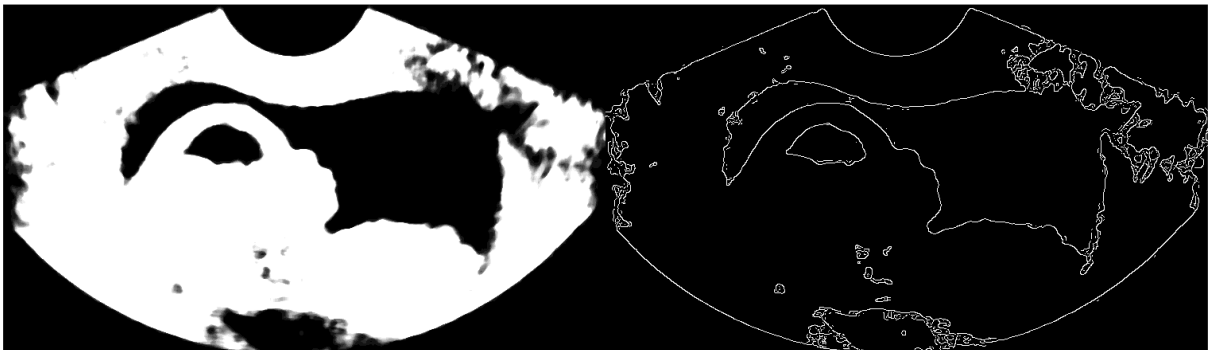

Figure 12: Weighted Median Filter, W = 20, n= 9, on foetus

Figure 12 shows a very clear outline of the foetus. This however loses the details of some of the prominent features such as the brain. This may or may not be useful depending on the purpose of the edge detection.

The weighted mean filter can be a beneficial tool as it provides exposure to the median value of image, however the tradeoff is the nose also experiences these effects. In terms of computing performance, this filter will perform almost identically to the median filter.

5.3 Truncated Median [1]

The truncated median filter compares the difference in value from the beginning of the sorted ROI to the median to the end of the sorted ROI to the median. The value found to be larger will have its end removed and the process will repeat until a single value is evaluated.

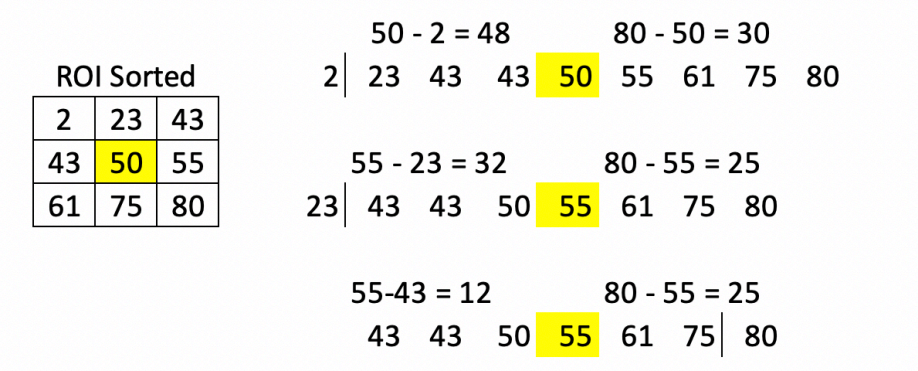

Figure 13: Example of the truncation process of the ROI

This process helps remove any outliers (ie, noise) in an image allowing the filtered image to be filtered correctly. This also means that edges are well preserved , and has the effect of 'crispening' the edges because it will prefer to select values either side of an edge due to the edge point being treated at an outlier for any points either side of it, thus this sharphens and enhancing the edge position.

Extract of the truncation loop from the truncatedMedianFilter Function:

```
#Difference of first half and second half
upperDifference = roiSorted[len(roiSorted)-1]-roiSorted[median]
lowerDifference = roiSorted[median]-roiSorted[0]

i = n*n
#Truncation Loop
while i > 0:
    i = i-1
    if upperDifference > lowerDifference:
        #delete highest value in the array
        roiSorted = np.delete(roiSorted,len(roiSorted)-1, 0)

        #set new values
        median = math.ceil(len(roiSorted)/2)
        upperDifference = roiSorted[len(roiSorted)-1]-roiSorted[median]
        lowerDifference = roiSorted[median]-roiSorted[0]

    elif lowerDifference > upperDifference:
        #delete lowest value in the array
```

```
        roiSorted = np.delete(roiSorted,0, 0)

        #set new values of median, upperDifference and lowerDifference
        median = math.ceil(len(roiSorted)/2)
        try:  #to avoid errors if there are 2 elements left
            upperDifference = roiSorted[len(roiSorted)-1]-roiSorted[median]
        except:
            median = 0
            break
```

The truncated median filter can be applied to see how it affects the input image.

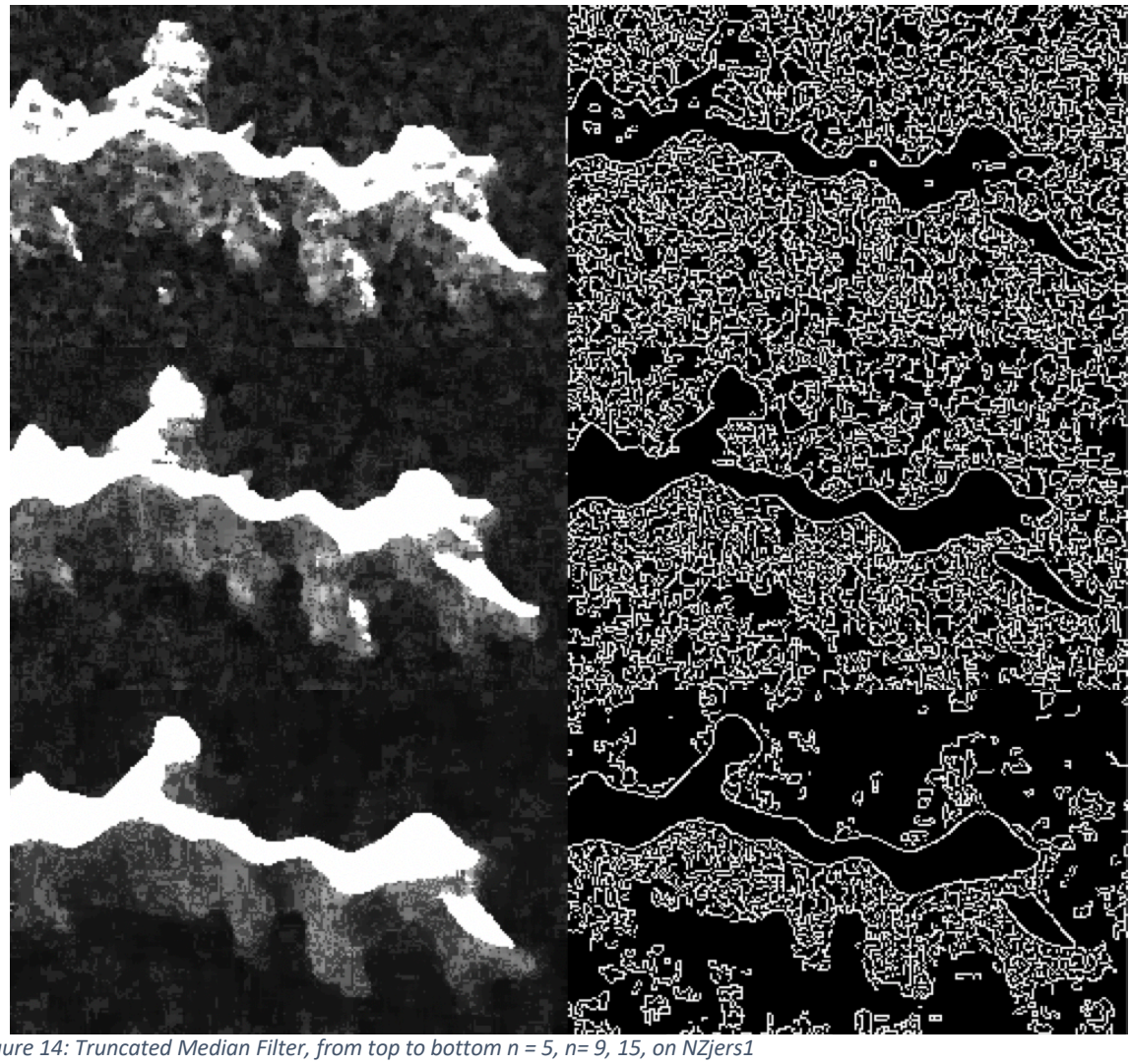

Figure 14: Truncated Median Filter, from top to bottom n = 5, n= 9, 15, on NZjers1

As the value of n increases the image loses a lot of detail, however the enges are sharpened. This filter might work well if the image was first filtered using some kind of linear filter to

reduce the intensity of the noise. This would bring back the sharpness of the features so long as the value of n is not too high.

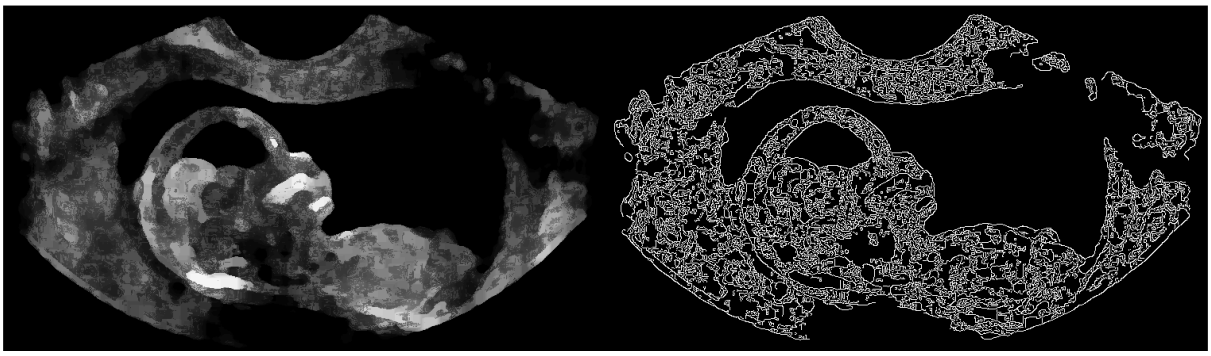

Figure 15: Truncated Median Filter n = 15, on foetus

It is interesting to see how a value of $n = 15$ causes the foetus image more clearly show the internal features such as the brain. This is a much more visually appealing and clear image for a human to analyse compared to that of the original foetus image. However the canny edge detection doesn't work very well here and may require other filters to be applies to mitigate the noisiness of the image.

One of the main issues with this filter is its performance spceed. In testing, it's been found to sometime produce images after several minutes of computation, depending on the window size.

Visually the output images are nice to look at the the edges are very well defined, the larger the value of the the more intense these defined edges become but this also begins to remove many features of the images. This is very prominent in the NZjers1 image as it is quite small so it can been seen that it deteriorats quickly. However in the image of the foetus, since the image is quite large, edges of different features are very well defined and would be very easy to visually analyse.

The edge detection however is extremely poor with this filter as it does not do a good job eliminating noise. In conjuction with other filters, such as the gaussian filter for example, this filter has the potential to overcome this problem and produce very well defined edges.

5.4 Adaptive Median [1]

The final of the Median Variants of filtering if the Adaptive Weighted Median. This uses a wighting window that changes depending on the position of the ROI within the image.

The equation that governs this weighting window w is:

$$w_{i,j} = \left[W_{(K+1,K+1)} - \frac{cd\sigma}{\bar{x}} \right] \qquad \text{Where } d = \begin{bmatrix} \sqrt{2} & 1 & \sqrt{2} \\ 1 & 0 & 1 \\ \sqrt{2} & 1 & \sqrt{2} \end{bmatrix}$$

c = constant
d = Euclidian distance from centre of mask
σ = standard deviation of the ROI window
$\bar{x}$ = mean of the ROI window

Using this filter, as edges are approached, the signal to noise ratio is reduced, and the central weighting values increases this is because the of $\frac{\sigma}{\bar{x}}$ will increase at these regions and cause w to reduce their elemental value the further away they way they are from the central element of the window, this then mean the filtering effects at the edges are reduced. The opposite applied at flat regions and thus the filtering effects are increased in.

The application of this adaptive weighted median required several additional steps to get functioning properly so the code to have it be functional will be broken down into segements.

Creating a W and a d window for a given nxn window

```
#Setting the central position of the adaptive  weight
    midpos = math.ceil(n/2)-1
    #Central Weight
    W = W*np.ones((n,n), dtype=np.int8)

    #Setting d values  (distance from the centre of the mask)
    d = np.ones((n,n), dtype=np.float)
    for j in range (0,n):
            for i  in range (0,n):
                d[j][i] = np.sqrt(((midpos-(i))**2 + (midpos-(j))**2))
```

Application of the Adaptive Median Matrix for every pixel in the image

```
for y in np.arange(padding, imH+padding):
        #For sliding across X in the image
        for x in np.arange(padding, imW +padding):
            #getting the Region of Interest
            roi = paddedImage[y-padding:y+padding+1, x-padding:x+padding+1]
```

Finding the value of mean ($\bar{x}$) and std (σ) and thus the adaptive weighted window (w)

```
            #getting mean xbar and std deviation
            mean = roi.sum()/(n*n)
            std  = np.std(roi)

            #adaptive weight window
            weightWindow = np.array(W - c*d*std/mean, dtype= np.float)

            #if values are less than 0 set to zero to prevent negative weights
            weightWindow = weightWindow.clip(0)
```

Sorting the order of the weights based on the ascending values of the ROI index

```
            #flattening ROI and Wighting
            roiFlattened = roi.flatten()
            weightWindowFlattened = weightWindow.flatten()
            #finding the index of the lowest to highest value in the ROI
            roiSortedIndex = np.argsort(roiFlattened)[::]

            #flattening out the ROI
            roiSorted = np.sort(roiFlattened, axis=0)

            #arranging the Weight Window according to the index values
            weightWindowFlattenedSorted = np.empty(len(weightWindowFlattened)-1,
dtype=np.int8) #initialising
            for i in range (0,len(weightWindowFlattened)-1):
                weightWindowFlattenedSorted[i] =
weightWindowFlattened[roiSortedIndex[i]]
```

By using the median of the sorted weights, the adaptive wighted median index can be found.

```
            #finding the sorted Weights median by (sum of the weights /2)rounded up
            medianWeightWindow = math.ceil(weightWindowFlattenedSorted.sum()/2)

            findAdaptiveMedian = int(0)  #initialising for adaptive median value
            indexAdaptiveMedian = int(0) #initialising for adaptive median value

            #finding the index of the adaptive weighted median value
            while findAdaptiveMedian < medianWeightWindow:
                findAdaptiveMedian =  findAdaptiveMedian +
weightWindowFlattenedSorted[indexAdaptiveMedian]
                indexAdaptiveMedian = indexAdaptiveMedian  +1
```

Finally, the value of the adaptive wighted median is then found byfinding the its corresponding index value within the sorted ROI.

```
            #finding the adaptive weighted median
            adaptiveWeightedMedian = roiSorted[indexAdaptiveMedian]

            output[y - padding, x - padding] = adaptiveWeightedMedian
```

Extensive testing was done on the NZjers1 image with these filters to see the effect of chaning the value of n and c to produce the best results. The results for these can be found in Appendix A.

Visually, the best value of n appeard to be when $n = 7$, lower than this there was still a considerable amound of noise in the image, and any larger would just produce a blurry looking image. The value of c that seemed the best was when $c = 25$ higher values of c began to introduce new noise into the image and lower values of c generated bleeding egdes in the image.

For edge detection a similar result was evaluated, where the best detection occurred around the area where $c = 25$, at higher values of c there was better edge detection around the island, however, the issue is that new noise being introduced meant that flase positives were becoming more and more prominent in the image.

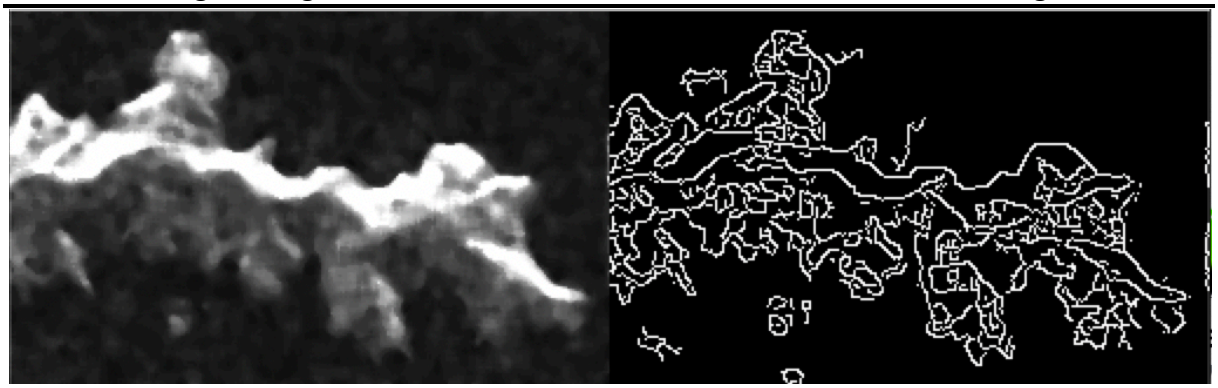

Figure 16: Adaptive Weighted Median where n = 7, c = 25, NZjers1

When applied to the foetus image however, it was found that very different values of c were needed to generate reasonable results. The optimal value of c was found to be $c = 10$ with $n = 9$ as the filtering window size.

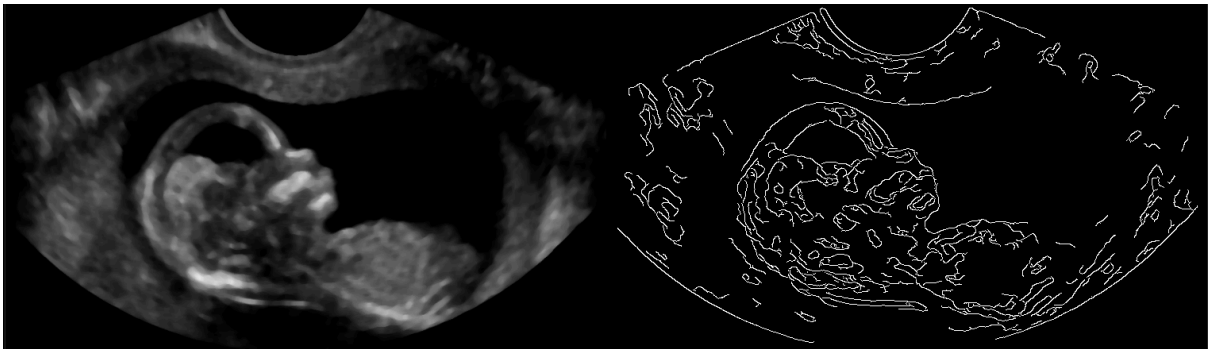

Figure 17: Adaptive Weighted Median where n = 9, c = 10, foetus

The images produced by the adaptive weighted median filter are visually easy to analyse and with the right parameters can generate quite good images to be used for edge detection. Computationally, this process is less intensive than the truncated median filter, however, this still does take considerably more time to evaluate than the other filters that were looked into.

5.5 Trimmed Mean Filter [1]

The trimmed mean filter is different to all the other non-linear filters in that it does not use medians to evaluate the value of the output pixel and is in fact much simpler.

It takes the values in the ROI window and sorts them, then removes a number of elements, set by the user, from either side before evaluating the mean value to be outputted into the pixel.

trimmedMean Function Code:

```
def trimmedMean(image, n, trimVal):
    #creating a window (aka kernel)
    typeWindow = np.ones((n,n), dtype=np.int8)
    element = typeWindow.sum()
    output, padding, paddedImage = imagePaddingFunc(image, typeWindow)

    for y in np.arange(padding, imH+padding):
        for x in np.arange(padding, imW +padding):
            roi = paddedImage[y-padding:y+padding+1, x-padding:x+padding+1]
            #applying the element wise multiplication of the filter to the image
            k = (roi * typeWindow)
            elem = element
            #sorting window (kernel) into a long single axis
            windowSorted = np.sort(k, axis=None)

            #Trimming the mean
            for i in range (0,trimVal):
                windowSorted = np.delete(windowSorted,0,0)
                windowSorted = np.delete(windowSorted,len(windowSorted)-1,0)
                elem = elem-2

            #adding  all the elements up and dividing my nuber of elemets to find
            #the mean
            mean = k.sum()/elem
            if mean < 0:
                mean = 0
            if mean > 255:
                mean  =  255

            output[y - padding, x - padding] = mean
    # return the output image
    output = np.array(output, dtype="uint8")
    return output
```

In testing, although this filter performs better than a mean Filter, it was found to perform quite badly compared to any of the other filters. The testing results can be found in Appendix A. Visually, the best image that was produced was generated at $n = 5$ with $trimVal = 3$.

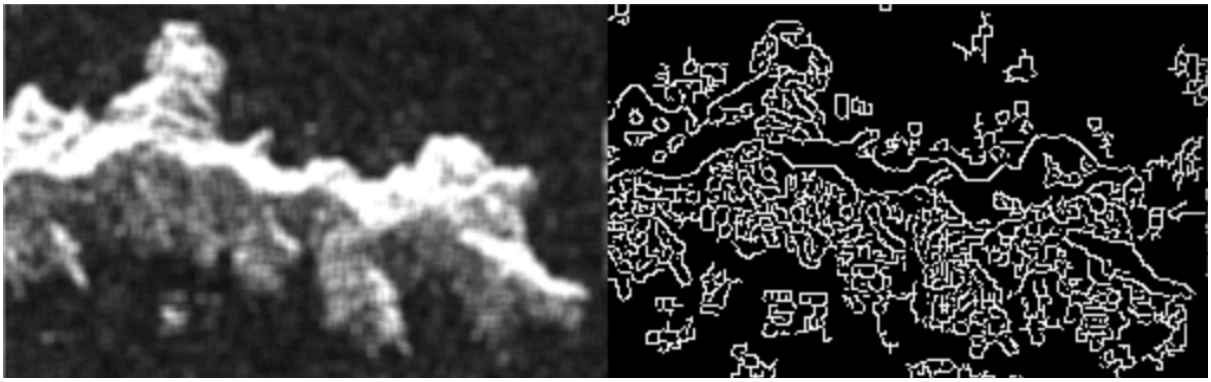

Figure 18: Trimmed mean filteter, n = 5, trimVal = 3, NZjers1

For edge detection, it was found the best performing values were at $n = 9$ with $trimVal = 3$. As at this poine noise around the land was mitigated but the lower features of the land begin to blend into the background.

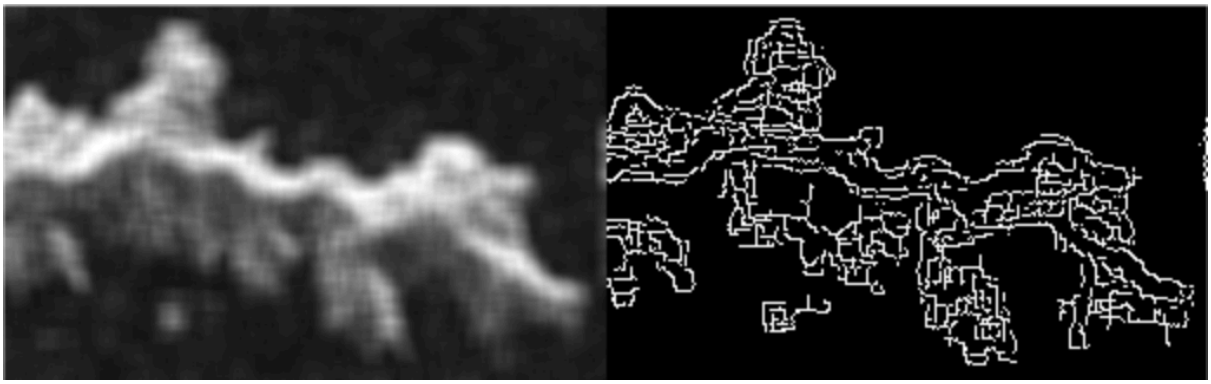

Figure 19: Trimmed mean filteter, n = 5, trimVal = 3, NZjers1

This filter offers better performance than the mean filter, but it can quickly be established that the method does not really produce very good results visually or for edge detection and would be better to use some of the other filters.

6 Conclusions

By looking the results of the different filters that have been programmed, some of them provide better results than the others both in terms of visual analysis as well as edge detection analysis for computational applications.

Of the two linear filters that were generated, the gaussian filter produces much better results than the mean filter. This is due to the gaussian distribution of the weighting window. It provides a quick method for reducing noise while retaining the edges of features, better the mean filter would.

Several non-linear filters were generated. The meadian filter is the fundamental filter for many of the other non-linear filters that were produced, the advantage of this filter was its ability to retain edges better than the linear filters whilst also smoothening the images at flat regions. Of the other filters, the output of the truncated median filter was the most visually appealing to look at due to the filter's ability to crispen edges but the image retained and reproduced noisiness at higher window sizes, this can be mitigated by passing a linear filter ontop of the image afterwards. The biggest issue with this filter was how slow it would operate, particularly at higher values of window size where images could take minutes to finish filtering. The adaptive weighted median filter also did a good job of retaining edges whilst also smoothening out flat regions, it can do this because its weighting changes depending on the region it is in making it filter more aggressivly in flat regions and less aggressively in edged regions.

Although none of the filters indivisually provide the best solution to generate the best visual and edge detection analysis, together, some of these filters can be powerful to find the relevant features of interest.

7 Bibliography

[1] D. A. Evans, “Image Enhancement - Lecture 4,” University of Bath, Bath, 2019.

[2] S. P. A. W. a. E. W. R. Fisher, “Canny Edge Detector,” 2003. [Online]. Available: https://homepages.inf.ed.ac.uk/rbf/HIPR2/canny.htm.

[3] A. Rosebrock, “Convolutions with OpenCV and Python,” pyimagesearch, 2016. [Online]. Available: https://www.pyimagesearch.com/2016/07/25/convolutions-with-opencv-and-python/.

[4] S. P. A. W. a. E. W. R. Fisher, “Gaussian Smoothing,” 2003. [Online]. Available: https://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm.

[5] D. A. Evans, “Noise Reduction of Synthetic Aperture Radar (SAR) and Ultrasound Images,” University of Bath, Bath, 2019.

8 Appendix A: Testing Results

8.1 Gaussian Image Results

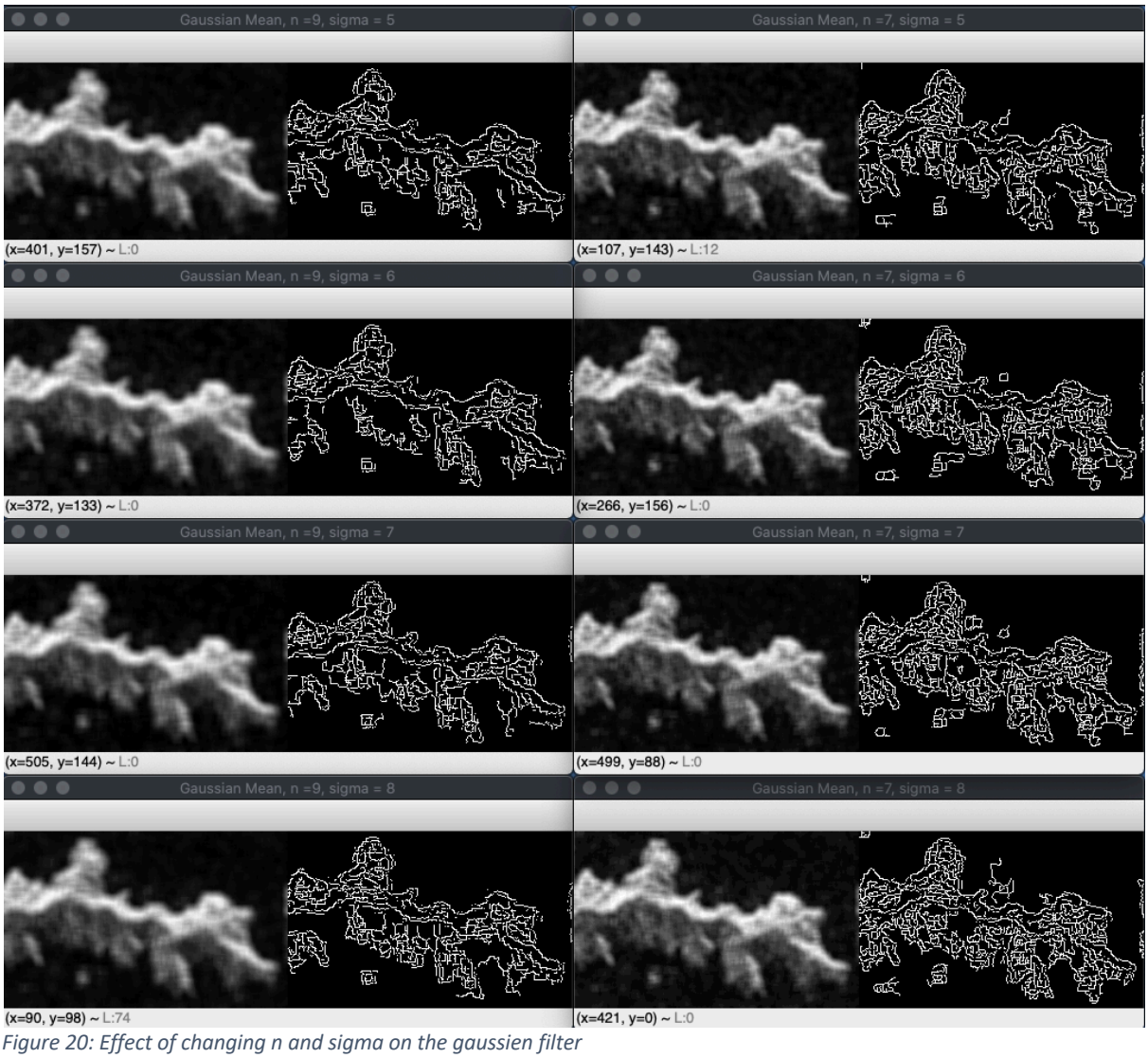

Figure 20: Effect of changing n and sigma on the gaussien filter

8.2 Adaptive Weighted Median Filter Results

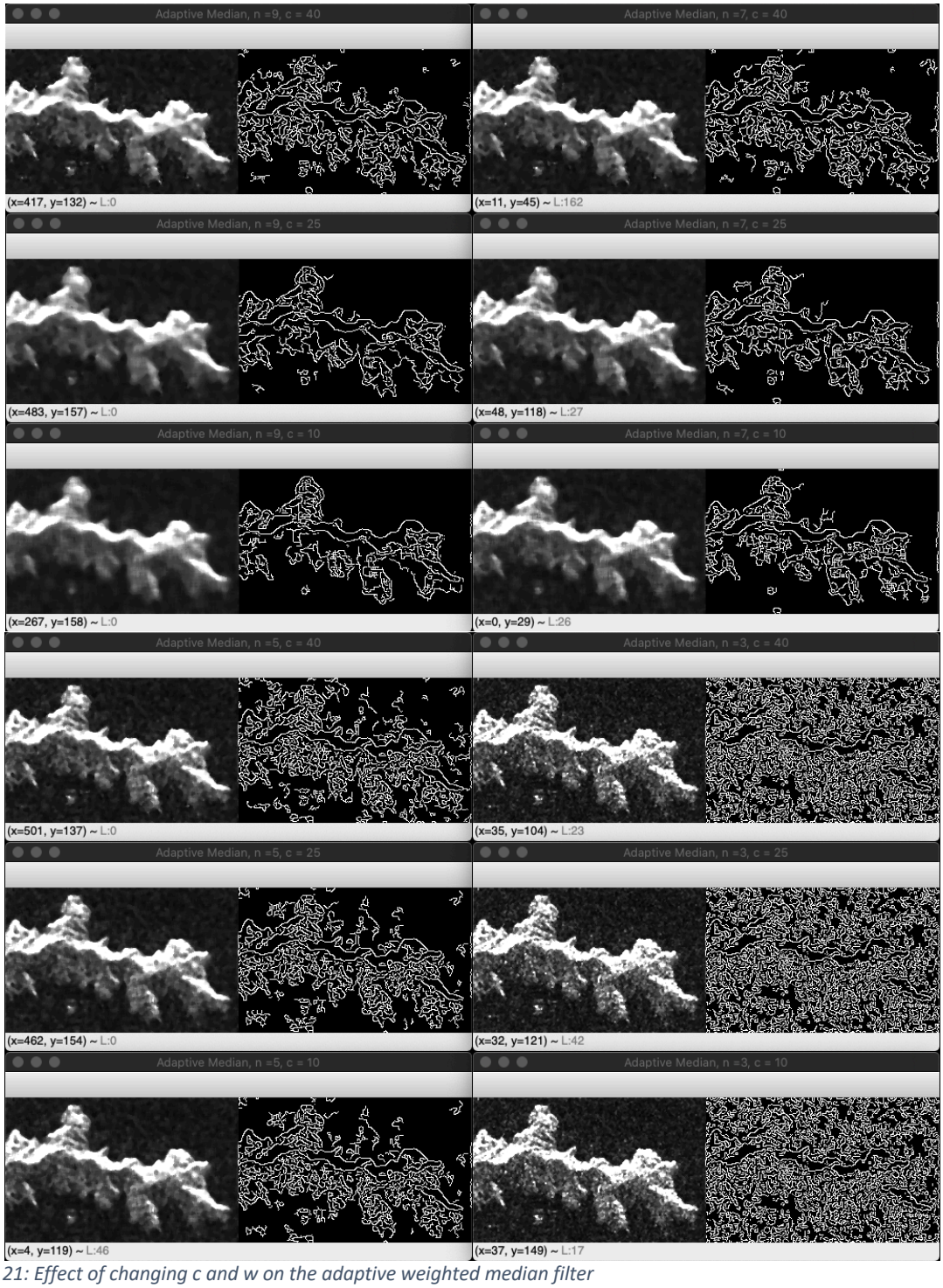

Figure 21: Effect of changing c and w on the adaptive weighted median filter

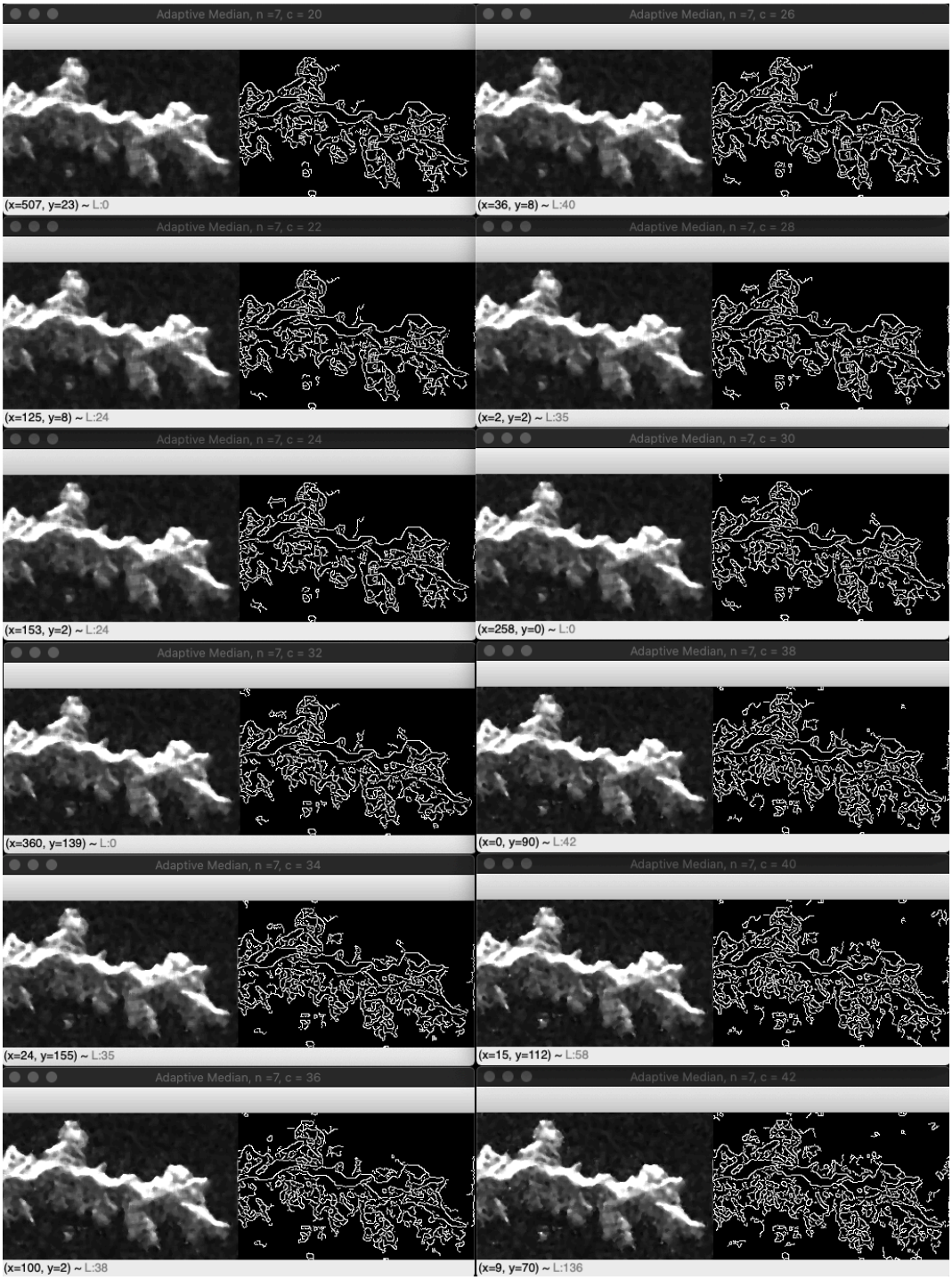

Figure 22: Effects of changing the value of c while n = 7 on the adaptive weighted median

8.3 Trimmed Median Filter Results

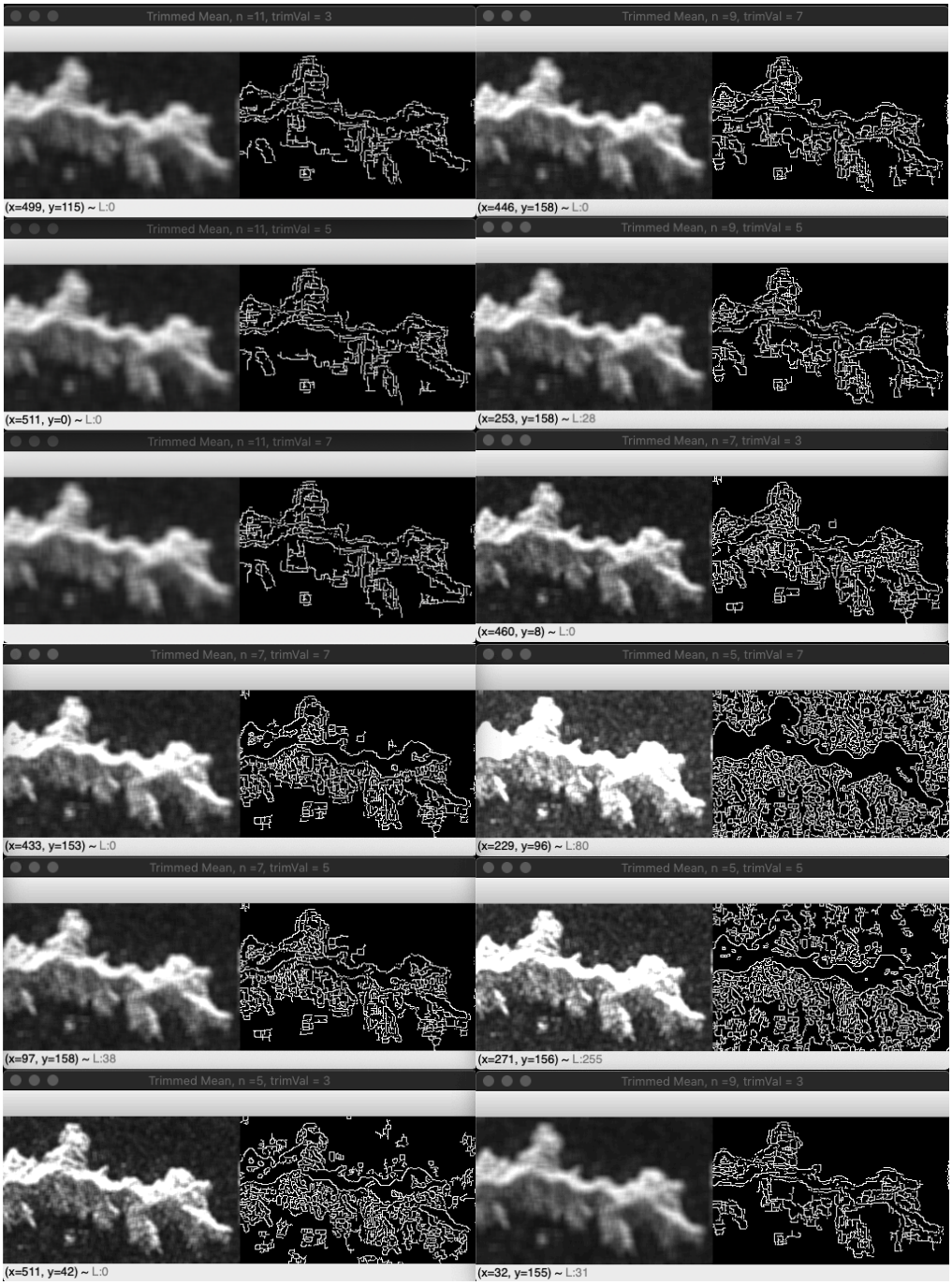

Figure 23: Effect of changing n and trimVal on the Trimmed Mean Filter

9 Appendix B: Full Python script

DIPFilters.py

```
import cv2
import numpy as np
import math
from datetime import datetime

def imagePaddingFunc(image, window):
    #Image padding function to extend the outer edges of the image
    #to the same values as the first few pixels along the edges

    #defining the height and width of image and filtering window
    wW,  wH = window.shape[:2]

    #padding the  edges of the image to allow processing to
    #start at the beginning of the image (needs to be int)
    padding = ((wW -1) // 2)

    paddedImage = cv2.copyMakeBorder(image, padding, padding, padding, padding,
cv2.BORDER_REPLICATE)

    #initialising the output image matrix
    output = np.zeros((imH, imW), dtype="float32")

    return output, padding, paddedImage

def slideMean (output, paddedImage, typeWindow, padding, imH, imW, elem):
    #For sliding down Y in the image as all of X completes
    # this is applies for filters that use a genereric mean filterin style

    for y in np.arange(padding, imH+padding):
        #For sliding across X in the image
        for x in np.arange(padding, imW +padding):

            #getting the Region of Interest of an image by creating the window
            # to apply the filter to
            roi = paddedImage[y-padding:y+padding+1, x-padding:x+padding+1]

            #applying the element wise multiplication of the filter to the image
            weightWindow = (roi * typeWindow)

            #adding  all the elements up and dividing my nuber of elemets to find
the mean
            mean = weightWindow.sum()/elem
```

```
            output[y - padding, x - padding] = mean

    output = np.array(output, dtype="uint8")
    return  output

def slideMedian(output, paddedImage, typeWindow, padding, imH, imW):
    #For sliding across X and down Y in the image for all pixels
    # this is applies for filters that use a genereric median filterin style

    for y in np.arange(padding, imH+padding):
        #For sliding across X in the image
        for x in np.arange(padding, imW +padding):
            #getting the Region of Interest of an image by creating the window
            # to apply the filter to
            roi = paddedImage[y-padding:y+padding+1, x-padding:x+padding+1]

            #Sort from lowest to highest value in the array
            #Clipping and additional convolution is for the weighted median
            roiSorted = np.clip(np.sort(roi, axis=None) *
typeWindow.flatten(),0,255)

            median = math.ceil(len(roiSorted)/2) - 1

            output[y - padding, x - padding] = roiSorted[median]
    output = np.array(output, dtype="uint8")
    return output

def histogramNorm(output, offset, padding, imH, imW):
    # rescale the output image to make sure the range of the image extends
    # to a full greyscale range of [0, 255]

    oMax = np.amax(output)  #max value of o/p image
    oMin = np.amin(output)  #min value of o/p image
    for y in np.arange(padding, imH+padding):
        for x in np.arange(padding, imW +padding):
            output[y-padding, x-padding] = (output[y-padding, x-padding]-
oMin)*255/(oMax-oMin) + offset
    return output

##Linear Functions##
def meanFilter(image, n):
    #creating a window of matrix nxn
    typeWindow = np.ones((n,n), dtype=np.int8)
```

```
    #creating a scalar to divide te overall mean window to bering intensity
    #back into the relevant range of 0 to 255
    elem = typeWindow.sum()

    #Image Padding
    output, padding, paddedImage = imagePaddingFunc(image, typeWindow)

    #apply slide mean function to find the mean of every ROI along the image
    output  = slideMean(output,paddedImage, typeWindow, padding, imH, imW, elem)

    # return the output image
    output = np.array(output, dtype="uint8")

    return output

def gaussianFilter(image, n, sigma):
    #similar to a mean filter but uses a gaussian mask where the centre values
    #have higher intensity the outer values

    typeWindow = np.ones((n,n), dtype=np.float)
    output, padding, paddedImage = imagePaddingFunc(image, typeWindow)
    midpos = math.ceil(n/2) - 1
    elem = 2 * np.pi * (sigma**2)

    #making the gaussian window
    for j in range (0,n):
        for i  in range (0,n):
            typeWindow[j][i] = (1/elem) * np.exp( -0.5*( (midpos-(i))**2 + (midpos-
(j))**2)/(sigma**2))

    output  = slideMean(output, paddedImage, typeWindow ,padding, imH, imW, 1)

    #applying histogram normalisation, as sigma increases, overall image darkens
    output = histogramNorm(output, 0, padding, imH, imW)

    #return the output image
    return output

##Non Linear Functions##
def trimmedMean(image, n, trimVal):
    #creating a window (aka kernel)
    typeWindow = np.ones((n,n), dtype=np.int8)
    element = typeWindow.sum()

    output, padding, paddedImage = imagePaddingFunc(image, typeWindow)
```

```
    for y in np.arange(padding, imH+padding):
        for x in np.arange(padding, imW +padding):
            roi = paddedImage[y-padding:y+padding+1, x-padding:x+padding+1]

            #applying the element wise multiplication of the filter to the image
            k = (roi * typeWindow)
            elem = element

            #sorting window (kernel) into a long single axis
            windowSorted = np.sort(k, axis=None)

            #Trimming the mean
            for i in range (0,trimVal):
                windowSorted = np.delete(windowSorted,0,0)
                windowSorted = np.delete(windowSorted,len(windowSorted)-1,0)
                elem = elem-2

            #adding  all the elements up and dividing my nuber of elemets to find
the mean
            mean = k.sum()/elem
            if mean < 0:
                mean = 0
            if mean > 255:
                mean  =  255

            output[y - padding, x - padding] = mean

    # return the output image
    output = np.array(output, dtype="uint8")
    return output

def medianFilter(image,n):
    typeWindow = np.ones((n,n), dtype=np.int8)
    output, padding, paddedImage = imagePaddingFunc(image, typeWindow)
    output  = slideMedian(output,paddedImage, typeWindow, padding, imH, imW)

    # return the output image
    output = np.array(output, dtype="uint8")
    return output

def weightedMedianFilter(image,n, W):
    typeWindow = np.ones((n,n), dtype=np.int8)
    typeWindow[math.floor(n/2)][math.floor(n/2)] = W
    output, padding, paddedImage = imagePaddingFunc(image, typeWindow)
```

```
    output  = slideMedian(output,paddedImage, typeWindow, padding, imH, imW)

    output = histogramNorm(output, 0, padding, imH, imW)

    # return the output image
    output = np.array(output, dtype="uint8")
    return output

def truncatedMedianFilter(image,n):
    window = np.ones((n,n), dtype=np.int8)
    output, padding, paddedImage = imagePaddingFunc(image, window)

    #For sliding down Y in the image as all of X completes
    for y in np.arange(padding, imH+padding):
        #For sliding across X in the image
        for x in np.arange(padding, imW +padding):

            #getting the Region of Interest of an image by creating the window
            # to apply the filter to
            roi = paddedImage[y-padding:y+padding+1, x-padding:x+padding+1]

            #Sort from lowest to highest value in the array
            roiSorted = np.sort(roi, axis=None)
            median = math.ceil(len(roiSorted)/2)

            #Difference of first half and second half
            upperDifference = roiSorted[len(roiSorted)-1]-roiSorted[median]
            lowerDifference = roiSorted[median]-roiSorted[0]

            i = n*n
            #Truncation Loop
            while i > 0:
                i = i-1
                if upperDifference > lowerDifference:
                    #delete highest value in the array
                    roiSorted = np.delete(roiSorted,len(roiSorted)-1, 0)

                    #set new values
                    median = math.ceil(len(roiSorted)/2)
                    upperDifference = roiSorted[len(roiSorted)-1]-roiSorted[median]
                    lowerDifference = roiSorted[median]-roiSorted[0]

                elif lowerDifference > upperDifference:
                    #delete lowest value in the array
```

```
                    roiSorted = np.delete(roiSorted,0, 0)

                    #set new values of median, upperDifference and lowerDifference
                    median = math.ceil(len(roiSorted)/2)
                    try:  #to avoid errors if there are 2 elements left
                        upperDifference = roiSorted[len(roiSorted)-1]-
roiSorted[median]
                    except:
                        median = 0
                        break
            output[y - padding, x - padding] = roiSorted[median]

    # return the output image
    output = np.array(output, dtype="uint8")
    return output

def adaptiveMedianFilter(image,n, W, c):
    #adaptiveMedianFilter(image, 3, 100, 10, 3)

    #Setting the central position of the adaptive  weight
    midpos = math.ceil(n/2)-1
    #Central Weight
    W = W*np.ones((n,n), dtype=np.int8)

    #Setting d values  (distance from the centre of the mask)
    d = np.ones((n,n), dtype=np.float)
    for j in range (0,n):
            for i  in range (0,n):
                d[j][i] = np.sqrt(((midpos-(i))**2 + (midpos-(j))**2))

    #appling padding
    output, padding, paddedImage = imagePaddingFunc(image, W)

    for y in np.arange(padding, imH+padding):
        #For sliding across X in the image
        for x in np.arange(padding, imW +padding):
            #getting the Region of Interest
            roi = paddedImage[y-padding:y+padding+1, x-padding:x+padding+1]

            #getting mean xbar and std deviation
            mean = roi.sum()/(n*n)
            if mean == 0:
                output[y - padding, x - padding] = 0
            else:
                std  = np.std(roi)
```

```
                #adaptive weight window
                weightWindow = np.array(W - c*d*std/mean, dtype= np.float)

                #if values are less than 0 set to zero
                weightWindow = weightWindow.clip(0)

                #flattening ROI and Wighting
                roiFlattened = roi.flatten()

                weightWindowFlattened = weightWindow.flatten()

                #finding the index of the lowest to highest value in the ROI
                roiSortedIndex = np.argsort(roiFlattened)[::]

                #flattening out the ROI
                roiSorted = np.sort(roiFlattened, axis=0)

                #arranging the Weight Window according to the index values
                weightWindowFlattenedSorted = np.zeros(len(weightWindowFlattened)-
1, dtype=np.int8) #initialising
                for i in range (0,len(weightWindowFlattened)-1):
                    weightWindowFlattenedSorted[i] =
weightWindowFlattened[roiSortedIndex[i]]

                #finding the sorted Weights median by (sum of the weights
/2)rounded up
                medianWeightWindow = math.ceil(weightWindowFlattenedSorted.sum()/2)

                findAdaptiveMedian = int(0)  #initialising for adaptive median
value
                indexAdaptiveMedian = int(0) #initialising for adaptive median
value


                #finding the index of the adaptive weighted median value
                while findAdaptiveMedian < medianWeightWindow:
                    findAdaptiveMedian =  findAdaptiveMedian +
weightWindowFlattenedSorted[indexAdaptiveMedian]
                    indexAdaptiveMedian = indexAdaptiveMedian  +1

                #finding the adaptive weighted median
                adaptiveWeightedMedian = roiSorted[indexAdaptiveMedian]

                output[y - padding, x - padding] = adaptiveWeightedMedian
```

```
    # return the output image
    output = np.array(output, dtype="uint8")
    return output

def weightedRankFilter(img, n):
    typeWindow = np.ones((n,n), dtype=np.int8)
    output, padding, paddedImage = imagePaddingFunc(image, typeWindow)

    #For sliding down Y in the image as all of X completes
    for y in np.arange(padding, imH+padding):
        #For sliding across X in the image
        for x in np.arange(padding, imW +padding):
            #getting the Region of Interest of an image by creating the window
            # to apply the filter to
            roi = paddedImage[y-padding:y+padding+1, x-padding:x+padding+1]

            #Sort from lowest to highest value in the array
            roiSorted = np.sort(roi, axis=None)

            median = math.ceil(len(roiSorted)/2) - 1

            output[y - padding, x - padding] = roiSorted[median]

    # return the output image
    output = np.array(output, dtype="uint8")
    return output

if __name__ == "__main__":
    #Reading image files for image processing
    image = cv2.imread('foetus.png',  cv2.IMREAD_GRAYSCALE )
    imH, imW = image.shape[:2]

    #Canny Edge Detection Threshold Values
    threshold1 = 30
    threshold2 =  100

    ############ APPLYING THE FILTERS ############
    ##--ORIGINAL IMAGE--##
    # cv2.imshow('original image', image)
    # print('original image')
    # edgeOrig = cv2.Canny(image, threshold1, threshold2)
    # catOrig = cv2.hconcat([image,edgeOrig])
    # cv2.imshow('Original', catOrig)
```

```
    ##-- MEAN FILTER -- LINEAR --##
    # n = 3
    # meanFilter = meanFilter(image, n)
    # # cv2.imshow('mean image', meanFilter)
    # print('mean complete')
    # edgeMean = cv2.Canny(meanFilter, threshold1, threshold2)
    # catMean = cv2.hconcat([meanFilter,edgeMean])
    # cv2.imshow('Mean, n =' + str(n), catMean)

    # #-- MEAN GAUSSIAN FILTER--LINEAR --##
    # n  = 5
    # sigma = 10
    # # for sigma in range(5,9,1):
    # #     for n in range (7,10,2):
    # gaussFilter = gaussianFilter(image, n,sigma)
    # # cv2.imshow('filtered image - gaussian', gaussFilter)
    # print('gaussian complete')
    # edgeGauss = cv2.Canny(gaussFilter, threshold1, threshold2)
    # catGauss = cv2.hconcat([gaussFilter,edgeGauss])
    # cv2.imshow('Gaussian Mean, n =' + str(n) +', sigma = '+ str(sigma), catGauss)

    ##-- MEDIAN FILTER -- NON LINEAR--##
    # n = 9
    # imageMed = medianFilter(image, n)
    # print('median complete')
    # cv2.imshow('median image', imageMed)
    # edgeMed = cv2.Canny(imageMed, threshold1, threshold2)
    # # catMed = cv2.hconcat([imageMed,edgeMed])
    # # cv2.imshow('Median, n =' + str(n), catMed)

    # # # # ##-- WEIGHTED MEDIAN FILTER -- NON LINEAR--##
    # n =11
    # W = 2
    # imageWmed = weightedMedianFilter(image, n, W)
    # print('weighted median complete')
    # # cv2.imshow('wighted median image', imageWmed)
    # edgeWmed = cv2.Canny(imageWmed, threshold1, threshold2)
    # catWMed = cv2.hconcat([imageWmed,edgeWmed])
    # cv2.imshow('Wighted Median', catWMed)
```

```
    # ##-- TRIMMED MEAN FILTER -- NON LINEAR--##
    # n =11
    # trimVal = 5
    # imageTmean = trimmedMean(image, n, trimVal)
    # print('trimmed mean complete')
    ## cv2.imshow('trimmed mean image', trimmedMean)
    # edgeTmean = cv2.Canny(imageTmean, threshold1, threshold2)
    # catTmean = cv2.hconcat([imageTmean,edgeTmean])
    # cv2.imshow('Trimmed Mean, n =' + str(n) +', trimVal = '+ str(trimVal),
catTmean)

    # ##-- RANK MEDIAN FILTER -- NON LINEAR--##
    # n = 9
    # imageRmed = weightedRankFilter(image,n)
    # print('ranked median complete')
    # # cv2.imshow('wighted rank image', imageRmed)
    # edgeRmed = cv2.Canny(imageRmed, threshold1, threshold2)
    # catTmed = cv2.hconcat([imageRmed,edgeRmed])
    # cv2.imshow('Ranked Median', catTmed)

    # ##-- TRUNCATED MEDIAN FILTER -- NON LINEAR --##
    # n = 15
    # t0 = datetime.now()
    # truncMed = truncatedMedianFilter(image, n)
    # print('truncated median took '+str(datetime.now()-t0)+' to complete ' )
    # # cv2.imshow('truncatedMedianFilter image - Linear', truncatedMedianFilter)
    # edgeTruncMed = cv2.Canny(truncMed, threshold1, threshold2)
    # catTruncMed = cv2.hconcat([truncMed,edgeTruncMed])
    # cv2.imshow('Truncated Median', catTruncMed)

    ##-- ADAPTIVE MEDIAN FILTER -- NON LINEAR --##
    # n = 9
    # W = 100
    # c = 25
    # adapMed = adaptiveMedianFilter(image, n, W, c)
    # print('adaptiveMedianFilter complete')
    ## cv2.imshow('Truncated Median', catAdapMed)
    # edgeAdapMed = cv2.Canny(adapMed, threshold1, threshold2)
    # catAdapMed = cv2.hconcat([adapMed,edgeAdapMed])
    # cv2.imshow('Adaptive Median, n =' + str(n) +', c = '+ str(c), catAdapMed)
```