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

Technical Report

EE40054 - Digital Image Processing

University of Bath

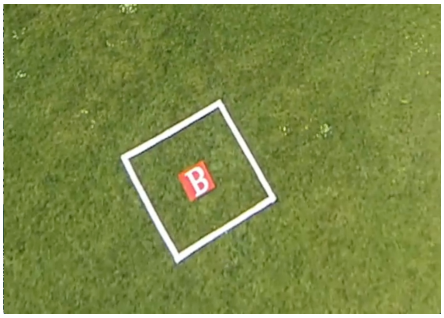
Candidate Number : 10374

1 Introduction

The human vision system is both very sophisticated and highly complicated yet it is not perfect. Noise, the bane of many digital systems but especially image processing, can obfuscate images making them difficult to process by eye when gradients and edges are blurred. Noise, however, typically follows pseudo-random distributions and as such can be counteracted through the use of certain filters. Even in the absence of noise, filters have many uses for enhancing images by artificially exacerbating the gradients and edges that the human vision system has evolved to detect.

2 Sample Images

In the initial assignment brief the two images NZjers.png and foetus.png[1]. For further appraisal DropZone.png was selected (taken from footage of an inflight aUAS [3]) and in order to assess the potential reactions to noise 20% gaussian noise was applied to foetus.png. All of these images are displayed in figure 1.



(a) Sample image from Team Bath Drones

(b) Provided image NZjers1.png

(c) Provided image foetus.png

(d) foetus.png with 20% gaussian noise

Figure 1: The images used for comparing filters

3 Filters

3.1 Comparison of Mean, Median and Gaussian Filters

Comparing the mean filter, the median filter and the gaussian filter for an equivalent window size on the image foetusNoise2 we are able to see the clear differences in the 3 filters. Comparing the gaussian and the mean filters for windows of the same size the mean filter is producing a much greater amount of blurring and edge reduction than the gaussian whereas the gaussian filter has retained a large amount of the noise in the original image. To achieve the same blurring of the edges as seen in the mean filter we have to vastly increase the size of the gaussian window, when we do this we see a lot of the noise disappear from the image with regards to the edges, however inspecting the image itself we see the contrast is still superior to the mean filter.

Lastly we have the median filter for window size 7. Of all of the images presented this has the most clearly defined edges with the skull, cheekbone and the mass within the skull all well defined, we are also able to see that the line surrounding the image is much thinner, being closer to the gaussian for small window size.

3.2 Truncated Mean Filter

By removing outliers within a window before calculating the mean of that window the potential for noise or nearby points of luminescence/darkness adjusting the mean is diminished. Comparing the images in figure 3 we can see how edges are better preserved in the image. Furthermore the shadows on the bottom side of the image and the lighter ridges are much more visible.

3.3 Adaptive Weighted Median

By adjusting the weights for median values based upon the characteristics of the window it is possible to improve the noise response for the median filter on a window by weighting the values within it in accordance with the properties of that window using the equation shown in figure4 [2]. Shown in figure 5 the median and adaptive weighted median filters have both been applied to the image *foetus.png* and from visual inspection of both the output images and their Sobel detected. edges it is clear to see a notable improvement to the clarity of the image.

When these filters are re-run over the image with 20% gaussian noise added to it we see an even greater improvement in the edges visible, the facial structure is clearly defined as and the spinal section of the foetus is notedly separated from the background in comparison to the simple mean filter

4 Edge Detection

In order to more easily assess the information retained within the image after filtering edge detection methods were coded into the image class. Comparing the Sobel and Prewitt methods gave very little detectable difference with regards to the outputted image however due to the edges appearing to have greater visibility, figure 7, the Sobel filter was selected. In order to better visualise the

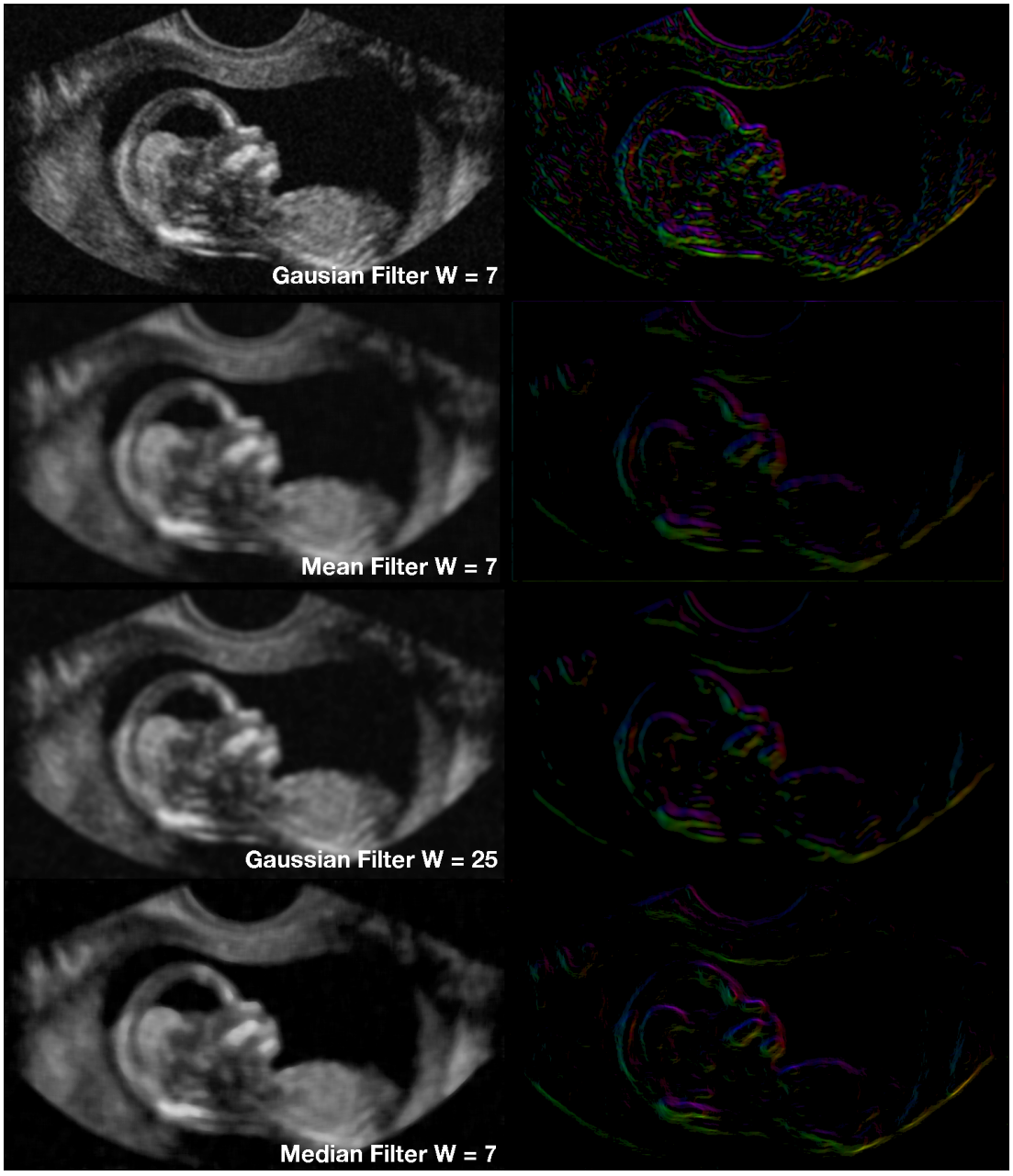


Figure 2: Comparison of a range of filters, minimum gradient = 25

gradients the theta angle generated by the method was encoded as the hue value in an HSV image, with the gradient value itself being assigned as the Value. Saturation was set to 255 across the image to aid in readability. Figure 8 below shows the edge filter applied to the DropZone image after a median filter (w=7) was applied, not only are the outlines of the white square in the image clearly visible but the lines themselves clearly show the direction of their gradients. Unfortunately the initial conversion to grayscale makes the red square very difficult to distinguish from the grassy

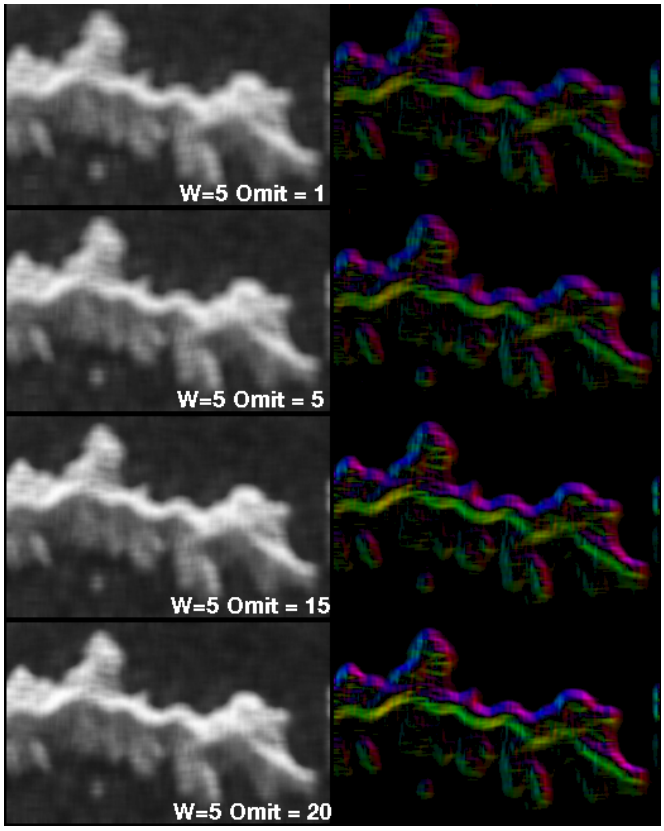


Figure 3: Comparing different amount of values omitted in calculating the mean of a 11x11 image

$$W_{i,j} = W_{mid} - d * c * \frac{\sigma}{\mu}$$

Figure 4: Adaptive Weighting Median, Weighting function

backdrop.

With more noisy images it is important to apply a degree of thresholding to the gradients required to count as an edge. Shown in figure 9 it can be seen that only a low threshold is needed to remove the lower values of noise however

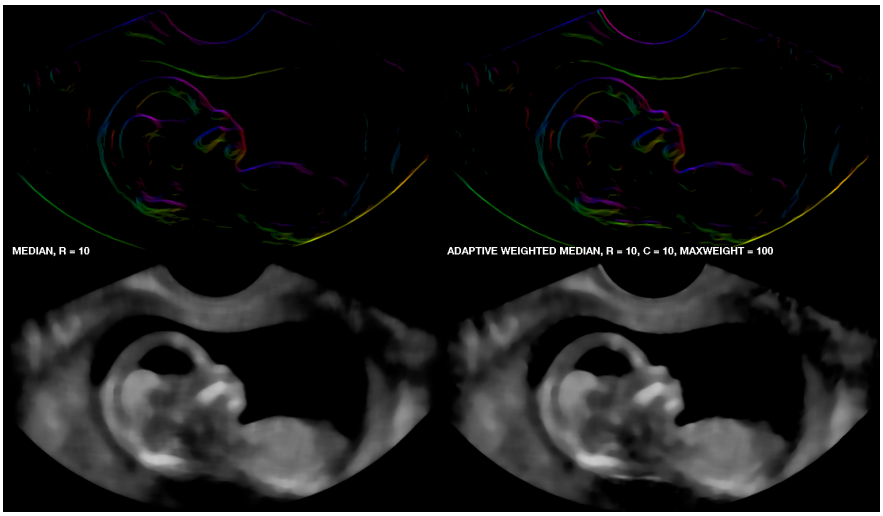


Figure 5: Comparison of the Median and Adaptive weighted Median filters with the same window size on the image: foetus.png

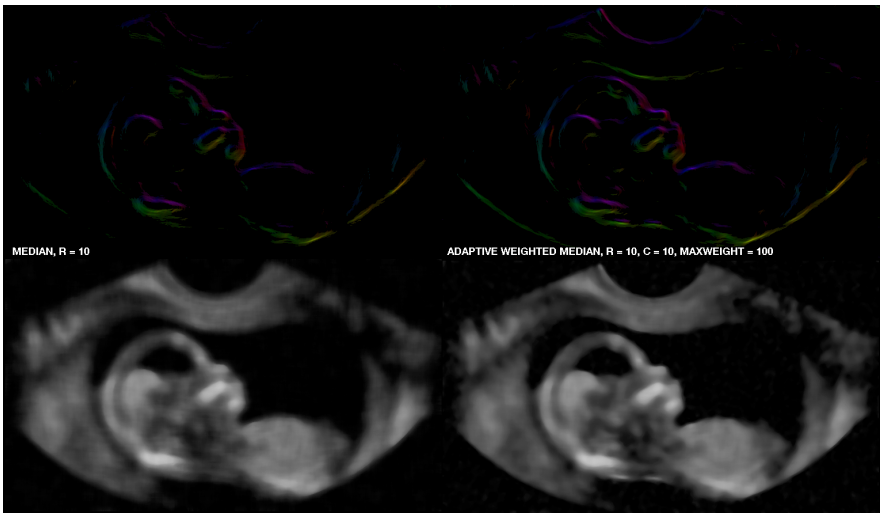
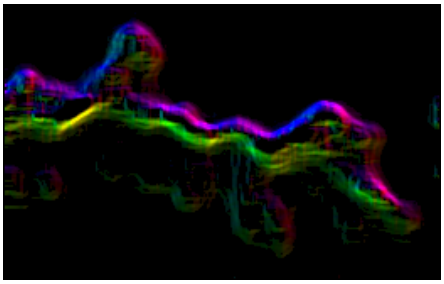


Figure 6: Comparison of the Median and Adaptive weighted Median filters with the same window size on the image: foetus.png with 20% Gaussian Noise

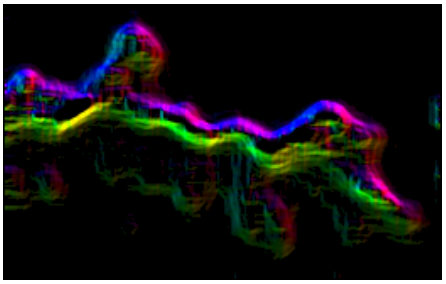
5 Evaluation

From the analysis of the different filters it was determined that the adaptive weighted median gave the most consistently good results for improving the images. For both the initially provided images there is a very clear improvement in the edge detection and readability of the images.

However if, in both cases, we determine that we care most about the edges between the objects in the image and the background then the application of a range highlight filter can help aid separating the objects from the backgrounds. The results of these range highlights can be seen in figure 10 and figure 11.



(a) Prewitt Filter



(b) Sobel Filter

Figure 7: Comparison of Sobel and Prewitt edge filters applied to the image *NZjers.png* after a median filter (w=7)

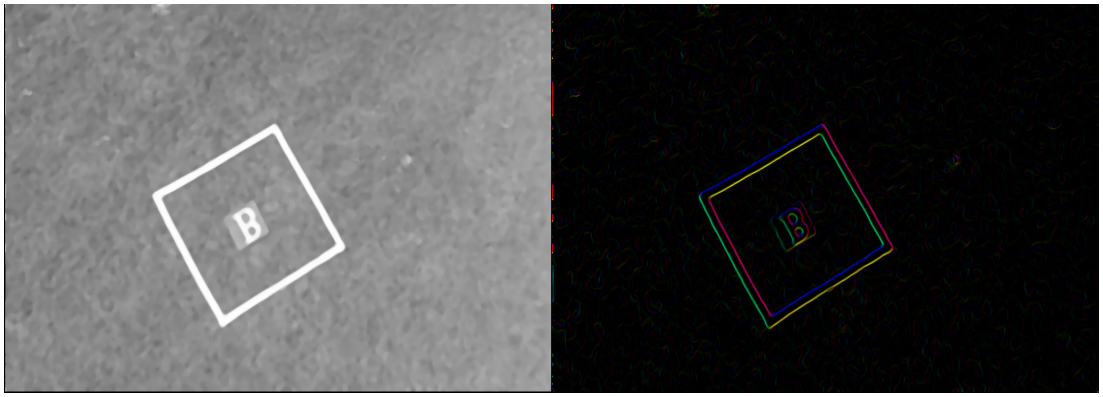


Figure 8: Example of the Sobel edge detector applied after a median filter (w = 7) on the image DropZone.png

6 Image Class

The image class is used to apply filters onto an imported image and to log all of the transforms performed. The methods are detailed in the subsections below.

6.1 Universal Functions

These functions were either used for the initialisation of the image class or for common minor adjustments to the image.

1. **_listFiles(args)** This function lists all png files within the current working directory to aid in loading image files.
2. **_importPic(args)** Using OpenCV's imread function this imports a grayscale copy of the image specified.
3. **_printFileList(_,listOfFiles)** Prints the list of all images found within the current directory.

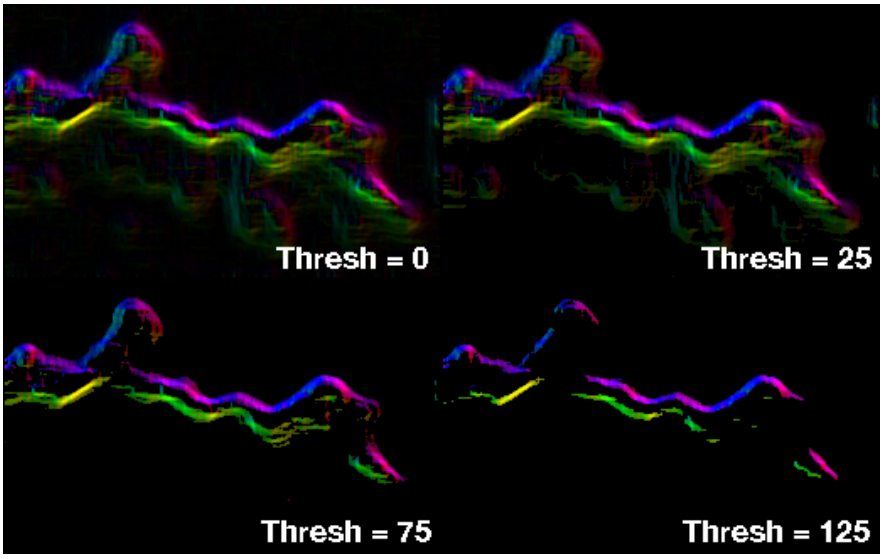


Figure 9: Comparing the effects of minimum gradient thresholds on the Sobel edge detector

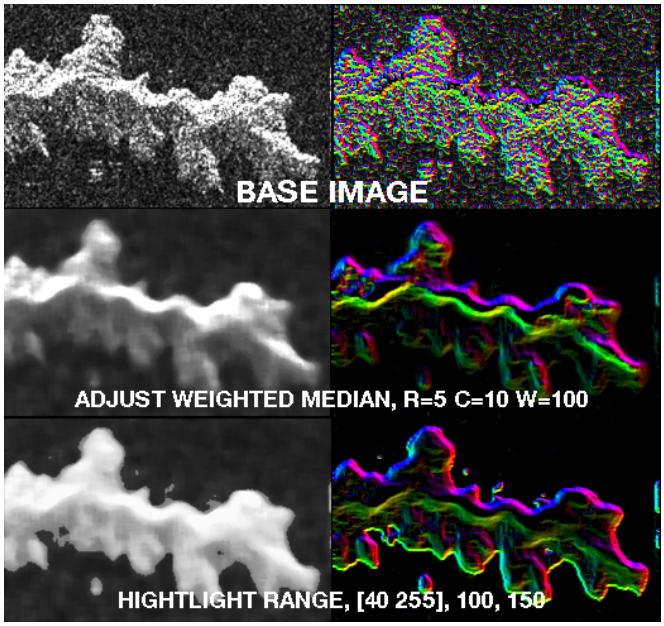


Figure 10: Comparing Stages of image analysis on *NJzers1.png*

4. **_genborders(self,amount)** Function to create a border around the image by mirroring the contents out for a distance of *amount* pixels.

5. **_deborders(self,amount)** Function to remove a border for a distance of *amount* pixels.

6. **createWindows(self, windowSize)** Function that creates an array of all windows within an image of size $2 * windowSize + 1$.

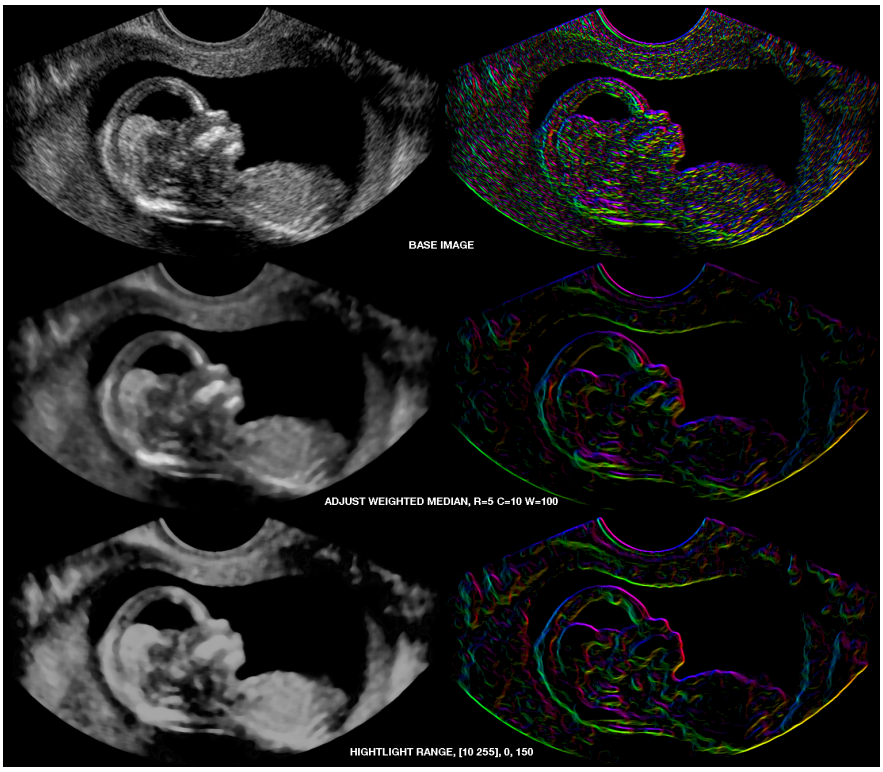


Figure 11: Comparing Stages of image analysis on *foetus.png*

6.2 External Functions

These functions were used for large scale manipulations of the images such as applying masks, updating the image or getting the dimensions of the image.

1. **imageSize(self)** Returns the dimensions of the grayscale image.
2. **uodateImage(self, newImage, transform)** Updates the image in the structure with a new image and appends the transform provided to the list of all applied transforms.
3. **duplicate(self)** Returns a new image with the same image as the original image.
4. **mask(self, mask)** Applies a mask provided to the image
5. **overlay(self,imageOver,mask)** Overlays imageOver ontop of the current image in the structure in accordance to the mask provided.
6. **snrRatio(self)** Small function that prints and returns the current signal to noise ratio of the image.

6.2.1 Output Functions

These functions were used to display the outputs of the processed images or save to files.

1. **showImage(self)** Function to display the current image of a structure on the screen until a key is pressed.

2. **saveImage(self)** Function to save the current image of a structure in the outputs subdirectory. The image name contains all transforms applied to the image in the order they were applied.

3. **histogram(self)** Displays a histogram of the grayscale levels of the current image.

6.3 Point Operator Functions

These functions were all single point operators which worked on a pixel by pixel basis within the image. The code typically looped over each pixel within the image performing operations on each pixel.

1. **POnormalise(self,nMin,nMax)** Function to map the grayscale levels proportionally over the defined range of $nMax - nMin$

2. **POequalise(self,levels,ignore,offset)** More advanced version of image equalisation from the notes. Includes the functionality to add an array of discrete values to be ignored, the number of levels to be equalised to and an offset value to increase all values in the image by.

3. **PObitSlice(self,lMin,lMax)** Function which produces a mask of all values in the image which are within the range specified by $lMin -- > lMax$The output for each pixel is either 0 or 255.

4. **PONegImage(self)** Function to invert the grayscale values of the image. Also works with coloured pictures not covered in this report.

5. **POfft(self)** Function to generate the fast fourier transform of the image.

6. **POfftMag(self)** Function to generate magnitude spectrum of the fast fourier transform of the image.

7. **POifft(self)** Function to generate the inverse fast fourier transform of the image.

6.4 Group Operator Functions

The group operator filters all work through the assessment of a the values within a window of the image. This necessitated the creation of the window class, detailed in section 7, and all functions in this section operate in the same way. Firstly creating an array of all the windows in the image, followed by applying a method to each window in that array.

1. **GOmean(self,windowSize)** Function to replace each pixel with the mean value of the surrounding values dictated by the size of the window.

2. **GOsnr(self,windowSize)** Function to replace each pixel with the signal to noise ratio of the surrounding values dictated by the size of the window.

3. **GOlinearGaussian(self,sigma)** Function to replace each pixel with the mean value of the surrounding values adjusted for distance using a gaussian method, dictated by the size of the sigma used for the gaussian.

4. **GOequalise(self,winSize, levels, ignore, offset)** Function to replace each pixel with the value the pixel equalised with respect to the surrounding values dictated by the size of the window. This filters code has a sys.exit at the start as it takes a long time to run, and as it's not in the notes it validity was uncertain.

5. **GOhysterise(self,windowSize, margin)** Function to remove values from an image if they are not within a specific margin to the greatest value of nearby pixels, dictated by the size of the window.

6.5 Non-Linear Functions

Operating similarly to the group operator filters through the use of window methods in loops the Non-Linear functions differ through using non-linear window method. Unfortunately there was insufficient time to successfully complete the adaptive weighted median however the code is left for posterity.

1. **NLmedian(self,winSize)** Function to replace each pixel with the median of the local window.

2. **NLmeanTrimmed(self,winSize, trim)** Function to replace each pixel with the mean of the local window, trimmed by a chosen value.

3. **NLadveightmedian(self,winSize, midweight, cVal)** Function to replace each pixel with the median calculated using an adaptive weighted method. Within each window the weighting matrix is constructed and then applied when calculating the median.

6.6 Edge Detection Functions

These functions are the implementation of both the Sobel and Prewitt edge detection filters with additional steps to implemented to represent the edge gradient as colour. The colour is centre with a theta angle of 0 corresponding to a hue value of 90.

1. **EDGEgradientsPrewitt(self,min)** Function to detect the edges in an image using the Prewitt filters. The value of min dictates the gradient threshold required for an edge to be counted.

2. **EDGEgradientsSobel(self,min)** Function to detect the edges in an image using the Sobel filters. The value of min dictates the gradient threshold required for an edge to be counted.

6.7 Other Functions

The functions in this section show example functions that can be created through the application of multiple filters within this document, in addition to one bookkeeping function that exists outside of the class definition.

1. **highlightRange(self,lBound,uBound,dcGain,levels)** Function to highlight a range of values, can also be used to remove all values within the two bounds and set them to a single value equal to dcGain.
2. **VisualiseFFT(self)** Function to replace an image in the class with the magnitude spectrum of that image.
3. **__printSpacer__(*args)** Function to print a line of asterix', or to print a line of asterix' and the content of args.

7 Window Class

The window class is used to apply filters to a window within the image being worked on. For brevity, all functions in the window class work by calculating a new value for the window based on the contents of that image and then filling that window with the new value. This value is then read by functions in the image class and used to set the new values.

A References

[1] A. Evans. Noise reduction of synthetic aperture radar (sar) and ultrasound images.

[2] R. C. Gonzalez and R. E. Woods. *Digital Image Processing*, chapter 5, pages 330–335. Pearson, 2016.

[3] R. Thomas. Sensors and vision: Phase 3 - technical report.

B Code

```
#! /usr/bin/env python3
# Module imageImport.py

################################################
#                                              #
#       Python functions for importing image files      #
#                                              #
#       Author: ————————————                   #
#       Username: ————                         #
#                                              #
################################################

# OpenCV used for loading images
from cv2 import *
import cv2


# Modules below used for listing available images in the current directory
from os import getcwd
from os.path import join, split
from glob import glob

# Modules for function termination and writing to file
from sys import exit
import sys

# Maths
import numpy as np
import scipy.signal

# Allow for plotting histogram graphs and 3d graphs http://matplotlib.org/mpl_toolkits/mplot3d/tutorial
    ↪ .html
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Enable creating the windows breaking list connectivity
import copy
from math import*


# Multi-threading for faster operating when utilising windows
from threading import *
```

```
42
43  class Window:
44          """
45  ␣␣␣␣␣␣␣␣Attributes:
46  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣x:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣x␣coordinate␣of␣image␣centre␣point
47  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣y:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣y␣coordinate␣of␣image␣centrepoint
48  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣width:␣␣Width␣of␣the␣window
49  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣content:␣NxN␣array␣containing␣the␣contents␣of␣the␣window
50  ␣␣␣␣␣␣␣␣"""
51          def __init__(self,wholeImage, centerPoint, width):
52                  self.x, self.y = centerPoint
53                  self.content = copy.copy(wholeImage.image[ (self.x-width):(self.x+width+1), (self.y-
                        ↪ width):(self.y+width+1) ])
54                  self.cp = width+1
55                  self.width = (2*width)+1
56
57          def cpVal(self):
58                  # Returns the value of the pixel at the centre of the window
59                  return self.content[self.cp-1,self.cp-1]
60
61          def average(self):
62                  ## Averages the values within a window and then fills the window with the new average
                        ↪ value
63                  averageValue = self.content.sum() / (self.width * self.width)
64
65                  self.content.fill(averageValue)
66
67          def gfilter(self, filter):
68                  # Applies Filter weights provided to the window before calculating mean
69                  averageFilterVal = (self.content * filter).sum() / filter.sum()
70                  self.content.fill(averageFilterVal)
71
72          def equalise(self, levels, ignore, offset):
73                  # Fiter returns the centre points value if equalised over the window
74                  # VERY SLOW & UNTESTED
75                  occurenceMap = np.zeros([256])
76                  greyScaleMap = np.zeros(256,'uint8')
77                  pixelCount=0
78                  for nVal in range(0,255):
79                          if nVal in ignore:
80                                  occurenceMap[nVal] = 0
81                          else:
82                                  occurenceMap[nVal] = (self.content==nVal).sum()
83                                  pixelCount += occurenceMap[nVal]
84                  for n in range(256):
85                          ## Generating incredibly low values for percentage chance (sigma function
                                ↪ missing)
86                          greyScaleMap[n] =  ((np.sum(occurenceMap[0:n]) / pixelCount ) * (levels-1)) +
                                ↪ offset
87                  self.content.fill(greyScaleMap[self.content[self.cp,self.cp]])
88
89          def median(self):
90                  # Filter to return median value of the window
91                  medianValue = np.median(self.content)
92                  self.content.fill(medianValue)
93
94          def snr(self):
95                  # Filter to return the SNR of the window
96                  if not self.content.std() == 0:
97                          val = self.content.mean() / self.content.std()
98                  else:
```

```
                val = 0
            self.content.fill(val)

    def trimmedMean(self,nOmit):
            # Returns the trimmed mean value of the window
            numValues = self.width * self.width
            halfPoint = ceil(numValues/2)

            sortedVal = np.sort(self.content, axis=None, kind='mergesort')
            trimmed = sortedVal[nOmit:-nOmit]
            self.content.fill(trimmed.mean())

    def prewittGX(self):
            # Prewitt GX filter
            Mx = np.array([[-1,0,1], [-1,0,1], [-1,0,1]])
            if self.width >=3:
                    tempWindow = self.content
                    gxWin = np.sum(np.sum(np.multiply(Mx,tempWindow)))
                    return gxWin
            else:
                    return 0

    def prewittGY(self):
            # Prewitt GY filter
            My = np.array([[-1,-1,-1], [0,0,0], [1,1,1]])
            if self.width >=3:
                    tempWindow = self.content
                    gyWin = np.sum(np.sum(np.multiply(My,tempWindow)))
                    return gyWin
            else:
                    return 0

    def sobelGX(self):
            # Sobel GX filter
            Mx = np.array([[-1,0,1], [-2,0,2], [-1,0,1]])
            if self.width >=3:
                    tempWindow = self.content
                    gxWin = np.sum(np.sum(np.multiply(Mx,tempWindow)))
                    return gxWin
            else:
                    return 0

    def sobelGY(self):
            # Sobel GY filter
            My = np.array([[-1,-2,-1], [0,0,0], [1,2,1]])
            if self.width >=3:
                    tempWindow = self.content
                    gyWin = np.sum(np.sum(np.multiply(My,tempWindow)))
                    return gyWin
            else:
                    return 0

    def hysterise(self, margin):
            # Zeros a window if a value nearby is notedly greater than the center value
            centreVal = self.content[self.cp, self.cp]
            maxVal = np.amax(self.content)

            if (centreVal + margin) > maxVal:
                    self.content.fill(maxVal)
            else:
                    self.content.fill(0)
```

```
160
161         def adaptiveMedian(self, c, midweight):
162                 #If the mean is 0 the whole window must be 0
163                 if not (self.content.mean() == 0):
164                         snrInvert = self.content.std() / self.content.mean()
165                         imWidth = int(self.width)
166                         # Generate a blank matrix to populate with the weights
167                         weightings = np.zeros((imWidth, imWidth))
168
169                         # Generate the weights matrix
170                         ## IMPROVEMENT: Should put distance outside of Window function as is constance
                                ↪ matrix
171                         for i in range(imWidth):
172                                 for j in range(imWidth):
173                                         distance = ((self.cp - i -1)**2 + (self.cp - j - 1)**2)**0.5
174                                         weight = int(midweight - (c * distance * snrInvert))
175                                         weight = int(np.amax([0, weight]))
176                                         weightings[i,j] = int(weight)
177
178                         # Flatten both of the arrays into 1dimensional to allow easier sorting
179                         flatIm          = self.content.flatten()
180                         flatWeight      = weightings.flatten()
181
182                         # Return the indices of the content that would correspond to it in sorted
                                ↪ format
183                         sortedIndices = np.argsort(flatIm)
184
185                         # Rearrange both matrices in accordance with the sorted indices
186                         sortedIm = flatIm[sortedIndices]
187                         sortedWeight = flatWeight[sortedIndices]
188
189                         # Replace each weight with the cumulative total of weights up to that index
190                         cumOcc = np.cumsum(sortedWeight)
191
192                         # Calculate the total weights
193                         totalOcc = np.sum(flatWeight)
194
195                         # Find the first index in the cumulative weights greater than half the value of
                                ↪ total weights
196                         index = np.argmax(cumOcc>(totalOcc/2))
197
198                         # Return the median value at that index
199                         medianVal = sortedIm[index]
200
201                 else:
202                         medianVal = 0
203                 self.content.fill(medianVal)
204
205 class Image:
206         # Define a class that contains an image and its associated name
207         'Common class for an image'
208         """
209         Attributes:
210                 name: A string containing the name of the original image
211                 image: An array of the individual pixels of the image
212                 transforms: A list of all performed transforms done on the image
213         """
214         def __init__(self, *args):
215                 # Constructor for the class
216                 # Generate a list of files within the current directory for validation
217                 listOfFiles = self._listFiles()
```

```
218
219             # No transforms have been yet done on the image
220             self.transforms = []
221
222             if (0 == len(args)): # Validate that there is an argument to import, if not generate an
                    ↪ empty image
223                 [self.image, self.name] = [ [], "" ]
224             else: # If a string is provided check that it exists within the current directory
225                 args = args[0]# Select only the first argument provided if it exists
226                 if (args) in listOfFiles: # If filename is present within the current directory
                        ↪ import the file and return it
227                     imageImported = self._importPic(args)
228                     __printSpacer__('Image file ' + args + ' succesfully loaded')
229                     [self.image, self.name] = imageImported, args
230                 elif 'empty' == args:
231                     self.image = []
232                     self.name = 'emptyImage'
233                 else: # If file does not exist within the directory exit the scripts
234                     __printSpacer__(args + ': file not found')
235                     self._printFileList(listOfFiles)
236                     exit()
237
238     ########################################
239     ##                                    ##
240     ## Methods (Internal):          ##
241     ##                                    ##
242     ########################################
243
244     def _listFiles(*args):
245         #Function to list available files for processing in the current directory
246         # Define the current working directory
247         cwd = getcwd()
248         # Create a glob-able parser
249         pngFilter = join(cwd,'*.png')
250         # Extract an array of all pngs in the current working directory
251         pngGlob = glob(pngFilter)
252
253         # Initialise an empty array to contain all valid png image files in the directory
254         pngFiles = []
255
256         # For each image detected in the current working directory remove the directory path
                ↪ and
257         # add file to pngFiles array
258         for image in pngGlob:
259             _ , fileName = split(image)
260             pngFiles.append(fileName)
261
262         # return the populated array of file names
263         return pngFiles
264
265     def _importPic(*args):
266         # Function to import an image corresponding to a pre-validated string
267
268         # OpenCV's function 'imread' brings in a grayscale image
269         # 1 = color image without alpha channel
270         # 0 = grayscale image
271         # -1 = unchanged image including alpha channel
272         imageImported = imread(args[1],0)
273
274         # Return imported image
275         return imageImported
```

```
276
277         def _printFileList(_,listOfFiles):
278                 # Function to print the files within the list
279                 __printSpacer__()
280                 print('No input file selected, please enter one of the following files as a function
                        ↪ input:')
281                 print('')
282                 for entry in listOfFiles:
283                         print(entry)
284                 __printSpacer__()
285
286         def _genBorders(self, amount):
287                 bordered = self.image
288                 for val in range(amount):
289                         bordered = np.vstack([bordered , bordered[-1,:]])
290                         bordered = np.vstack([bordered[0,:] , bordered])
291                         bordered = bordered.transpose()
292                         bordered = np.vstack([bordered , bordered[-1,:]])
293                         bordered = np.vstack([bordered[0,:] , bordered])
294                         bordered = bordered.transpose()
295                 self.updateImage(bordered, 'Bordered')
296                 return bordered
297
298         def _deBorders(self,amount):
299                 imageM, imageN = self.imageSize()
300                 debordered = self.image[amount:(imageM-amount), amount:(imageN-amount)]
301                 # [amount,(imageM - amount)], [amount,(imageN - amount)]
302                 self.updateImage(debordered, 'deBordered')
303
304         def createWindows(self, windowSize):
305                 ## Function to create an array containing NxN sized window structures for the whole
                        ↪ image
306                 imageM, imageN = self.imageSize()
307
308                 windowArray = []
309
310                 for i in range(windowSize,(imageM-windowSize)):
311                         for j in range(windowSize,(imageN-windowSize)):
312                                 windowArray.append( Window(self,[i,j],windowSize))
313
314                 __printSpacer__(str(len(windowArray)) +' windows of size ' + str(windowSize*2 +1) +'x'+
                        ↪ str(windowSize*2 +1)+ ' created from image')
315
316                 return windowArray
317
318
319         ########################################
320         ##                                    ##
321         ## Methods (External):                ##
322         ##                                    ##
323         ########################################
324
325         def imageSize(self):
326                 imageM, imageN = self.image.shape
327                 return imageM, imageN
328
329         def updateImage(self, newImage, transform):
330                 # Update the image after a new transform
331                 self.image = newImage
332                 self.transforms.append(transform)
333
```

```
334         def duplicate(self):
335                 # Function to create a duplicate of the current structure
336                 newStruct = Image('empty')
337                 # Hard assigns to stop altering original
338                 imageM, imageN = self.imageSize()
339                 newImage = np.zeros([imageM, imageN],'uint8')
340                 for i in range(imageM):
341                         for j in range(imageN):
342                                 newImage[i,j] = self.image[i,j]
343                 newStruct.image = newImage
344 
345                 newStruct.name = self.name
346                 newStruct.transforms = self.transforms
347                 return newStruct
348 
349         def mask(self, mask):
350                 # Function to apply a mask provided to the image
351                 self.image = np.minimum(self.image, mask.image)
352 
353         def overlay(self, imageOver, mask):
354                 # Function to overlay two images of the same size using a defined mask
355                 # Apply mask to the base image
356                 self.mask(mask)
357                 # Invert the mask
358                 mask.POnegImage()
359                 # Apply the inverted mask to the image to overlay
360                 imageOver.mask(mask)
361                 # Combine the two masked images
362                 self.image = self.image + imageOver.image
363 
364         def snrRatio(self):
365                 a = self.image.mean() / self.image.std()
366                 print(a)
367                 return a
368 
369         ##########################################
370         ##                                      ##
371         ## Methods (Outputs)                    ##
372         ##                                      ##
373         ##########################################
374         def showImage(self):
375                 # Shows the contained image until a key is pressed
376                 imshow(self.name, self.image)
377                 waitKey(0)
378                 destroyAllWindows()
379 
380         def saveImage(self):
381                 # Function to save an image to file after processing
382                 imwrite("./outputs/"+self.name + "_".join(self.transforms) + ".png", self.image)
383 
384         def histogram(self):
385                 # Function to create a histogram of an image's greyscale levels using the formula:
386                 # p(r) = n / MN
387 
388                 # Find the total number of elements in the image
389                 imageM, imageN = self.imageSize()
390                 imageMN = imageM * imageN *1.0
391                 print(imageMN)
392 
393                 # Initialise an array for counting the occurences of each grey value
394                 nkCount = np.zeros(256)
```

```
395
396                 for i in range(imageM):
397                         for j in range(imageN):
398                                 nkCount[self.image[i,j]] = nkCount[self.image[i,j]] + 1.0
399
400                 # Calculate probability of occurence
401                 probRk = nkCount / imageMN
402
403                 plt.plot(probRk)
404                 plt.xlim(0,255)
405                 plt.show()
406
407         ##########################################
408         ##                                      ##
409         ## Methods (Point Operator Filters)     ##
410         ##                                      ##
411         ##########################################
412         def POnormalise(self, nMin, nMax):
413                 # Function to utilise contrast stretching for a given image using the formula:
414                 # f(x,y) = [f(x,y) - Omin] x (Nmax-Nmin)/(Omax-Omin) + Nmin
415
416                 # Ensure that the boundaries are floats so that the conversion ratio is a float
417                 nMin = float(nMin)
418                 nMax = float(nMax)
419
420                 # Find Omax and Omin Values
421                 oMin = np.amin(self.image)
422                 oMax = np.amax(self.image)
423
424                 # Generate the conversion ratio to reduce the amount of divisions required
425                 conversionRatio = ((nMax - nMin)/(oMax-oMin))
426
427                 # Create an emtpy array to populate with the new image information
428                 imageM, imageN = self.imageSize()
429                 newImage = np.zeros([imageM, imageN],'uint8')
430
431                 for i in range(imageM):
432                         for j in range(imageN):
433                                 newImage[i,j] = int((self.image[i,j] - oMin) * conversionRatio + nMin)
434                 self.updateImage(newImage, 'normalise')
435
436         def POequalise(self,levels,ignore,offset):
437                 """
438 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Inputs:
439 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣self␣␣␣␣=␣I␣dont␣understand␣object␣orientated␣enough␣to␣explain␣why␣self
440 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣levels␣␣=␣the␣number␣of␣unique␣luminence␣levels␣in␣the␣image
441 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ignore␣␣=␣an␣array␣containing␣all␣luminence␣values␣to␣be␣ignored␣for␣the␣current␣array␣
      ↪ processing
442 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"""
443                 # Create an empty array to populate with the new iamge information
444                 imageM, imageN = self.imageSize()
445                 imageMN = imageM * imageN * 1.0
446                 newImage = np.zeros([imageM, imageN],'uint8')
447
448                 # Initialise an array for counting the occurences of each grey value
449                 occurenceMap = np.zeros([256])
450                 greyScaleMap = np.zeros(256,'uint8')
451
452                 pixelCount = 0
453
454                 if ((levels + offset) > 255):
```

```
455                     __printSpacer__('Warning: DC-offset + desired level range exceeds 255, high
                            ↪ values may show as black')
456
457             for nVal in range(0,255):
458                     if nVal in ignore:
459                             occurenceMap[nVal] = 0
460                     else:
461                             occurenceMap[nVal] = (self.image==nVal).sum()
462                             pixelCount += occurenceMap[nVal]
463
464             for n in range(256):
465                     greyScaleMap[n] =  ((np.sum(occurenceMap[0:n]) / pixelCount ) * (levels-1)) +
                            ↪ offset
466
467             for si in range(imageM):
468                     for sj in range(imageN):
469                             newImage[si,sj] = greyScaleMap[self.image[si,sj]]
470             self.updateImage(newImage, 'equaliseHistogram' + '_' + str(levels) + '_' + str(ignore)
                    ↪ + '_' + str(offset))
471
472     def PObitSlice(self, lMin, lMax):
473             # Function to produce the parts of the image within a certain bit range,
474             # Returns a new image structure so that it can be used for generating masks
475             newImageStruct = self.duplicate()
476             newImage = newImageStruct.image
477
478             lowerSubset = newImage < lMin
479             midSubset = (newImage >= lMin) & (newImage <= lMax) # We want to include the values
                    ↪ specified
480             upperSubset = newImage > lMax
481
482             newImage[lowerSubset] = 0
483             newImage[midSubset] = 255
484             newImage[upperSubset] = 0
485
486             newImageStruct.updateImage(newImage,'GreySlice')
487             return newImageStruct
488
489     def POnegImage(self):
490             # Function to invert the colours (or greyscale) of the image
491             initialImage = self.image
492             newImage = 255 - initialImage
493             self.updateImage(newImage, 'Invert')
494
495     def POfft(self):
496             f = np.fft.fft2(self.image)
497             fFhift = np.fft.fftshift(f)
498             self.updateImage(fFhift,'fft')
499
500     def POfftMag(self):
501             # Function to perform a fast fourier transform on an image and subsequently
502             # generate the resulting magnitude spectrum
503             self.POfft()
504             magnitude_spectrum = 20*np.log(np.abs(self.image))
505             self.updateImage(magnitude_spectrum,'Magfft')
506
507     def POifft(self):
508             ifShift = np.fft.ifftshift(self.image)
509             ifft = np.abs(np.fft.ifft2(ifShift))
510             self.updateImage(ifft,'ifft')
511             # Re normalise results to the range of grayscale values
```

```
512             self.POnormalise(0,255)
513
514     ########################################
515     ##                                    ##
516     ## Methods (Group Operator Filters)   ##
517     ##                                    ##
518     ########################################
519
520     def GOmean(self,windowSize):
521             # Create an array of windows
522             windowArray = self.createWindows(windowSize)
523
524             # Create a new image to populate
525             imageM, imageN = self.imageSize()
526             newImage = np.zeros([imageM, imageN],'uint8')
527
528             # Apply the average window function to each window in array
529             for window in windowArray:
530                     window.average()
531                     newImage[window.x, window.y] = window.cpVal()
532             # Update image
533             self.updateImage(newImage, 'mean')
534
535     def GOsnr(self,windowSize):
536
537             windowArray = self.createWindows(windowSize)
538
539             imageM, imageN = self.imageSize()
540             newImage = np.zeros([imageM, imageN],'uint8')
541
542             for window in windowArray:
543                     window.snr()
544                     newImage[window.x, window.y] = window.cpVal()
545
546             self.updateImage(newImage, 'snrCompare_' + str(windowSize))
547
548     def GOlinearGaussian(self, sigma):
549
550             winSize = 2*(3*sigma) + 1
551
552             self._genBorders(winSize+1)
553
554             windowArray = self.createWindows(3*sigma)
555
556             imageM, imageN = self.imageSize()
557             newImage = np.zeros([imageM, imageN],'uint8')
558
559             # Make an empty array to contain the mask
560             gausMask = np.zeros([winSize,winSize])
561
562             # Populate the mask with calulated weightings
563             for i in range(3*sigma+1):
564                     for j in range(3*sigma+1):
565                             gVal = exp( -( pow(3*sigma - i,2) + pow(3*sigma - j,2) ) / ( 2 * sigma
                                ↪  * sigma ) )
566
567                             # Gaussian mask is symettrical around centre so can set multiple values
568                             # at the same time
569                             gausMask[i,j] = gVal
570                             gausMask[i,winSize-j-1] = gVal
571                             gausMask[winSize-i-1,j] = gVal
```

```
572                                 gausMask[winSize-i-1,winSize-j-1] = gVal
573
574             # Apply guasian mask
575             for window in windowArray:
576                     window.gfilter(gausMask)
577                     newImage[window.x, window.y] = window.cpVal()
578
579             self.updateImage(newImage, 'Gaussian')
580             self._deBorders(winSize+1)
581
582     def GOequalise(self, winSize, levels,ignore,offset):
583             sys.exit() ## This code takes an age to run so stop
584
585             self._genBorders(winSize+1)
586
587             imageM, imageN = self.imageSize()
588
589             windowArray = self.createWindows(winSize)
590
591             newImage = np.zeros([imageM, imageN],'uint8')
592
593             for window in windowArray:
594                     window.equalise(levels,ignore,offset)
595                     newImage[window.x, window.y] = window.cpVal()
596
597             self.updateImage(newImage, 'equaliseWindowedHistogram' + '_' + str(levels) + '_' + str(
                    ↪ ignore) + '_' + str(offset))
598
599     def GOhysterise(self,windowSize,margin):
600             # Populate the window array
601             windowArray = self.createWindows(windowSize)
602
603             imageM, imageN = self.imageSize()
604
605             newImage = np.zeros([imageM, imageN],'uint8')
606
607             for window in windowArray:
608                     window.hysterise(margin)
609                     newImage[window.x, window.y] = window.cpVal()
610
611             self.updateImage(newImage, 'hysterise')
612
613     ########################################
614     ##                                    ##
615     ## Methods (Non Linear Filters)       ##
616     ##                                    ##
617     ########################################
618
619     def NLmedian(self, winSize):
620             # Generate borders for the image
621             self._genBorders(winSize+1)
622
623             # Find image dimension
624             imageM, imageN = self.imageSize()
625
626             # Populate the window array
627             windowArray = self.createWindows(winSize)
628
629             newImage = np.zeros([imageM, imageN],'uint8')
630
631             for window in windowArray:
```

```
632                     window.median()
633                     newImage[window.x, window.y] = window.cpVal()
634
635             self.updateImage(newImage, 'Median Filter: r=' + str(winSize))
636             self._deBorders(winSize+1)
637
638     def NLmeanTrimmed(self, winSize,trim):
639             # Generate borders for the image
640             self._genBorders(winSize+1)
641
642             # Find image dimension
643             imageM, imageN = self.imageSize()
644
645             # Populate the window array
646             windowArray = self.createWindows(winSize)
647
648             newImage = np.zeros([imageM, imageN],'uint8')
649
650             if trim > (winSize*winSize):
651                     print('Please enter a trim value less than ' + str(winSize*winSize))
652                     return
653
654             for window in windowArray:
655                     window.trimmedMean(trim)
656                     newImage[window.x, window.y] = window.cpVal()
657
658             self.updateImage(newImage, 'Trimmed Mean Filter: r=' + str(winSize) + 'Omit=' + str(
                    ↪ trim))
659             self._deBorders(winSize+1)
660
661     def NLadweightmedian(self, winSize, midweight, cVal):
662             # Generate borders for the image
663             self._genBorders(winSize+1)
664
665             # Find image dimension
666             imageM, imageN = self.imageSize()
667
668             # Populate the window array
669             windowArray = self.createWindows(winSize)
670
671             newImage = np.zeros([imageM, imageN],'uint8')
672
673             for window in windowArray:
674                     window.adaptiveMedian(cVal,midweight)
675                     newImage[window.x, window.y] = window.cpVal()
676
677             self.updateImage(newImage, 'Adaptive Median Filter: r=' + str(winSize))
678             self._deBorders(winSize+1)
679
680
681     ########################################
682     ##                                    ##
683     ## Methods (Edge Detect Filters)      ##
684     ##                                    ##
685     ########################################
686
687     def EDGEgradientsPrewitt(self, min):
688             self._genBorders(2)
689
690             imageM, imageN = self.imageSize()
691
```

```
692                 windowArray = self.createWindows(1)
693
694                 newImage = np.zeros([imageM, imageN],'uint8')
695                 thetaImage = np.zeros([imageM, imageN],'uint8')
696                 blankImage = np.zeros([imageM, imageN],np.uint8)
697
698                 # For each window run the two directions of the edge detectors
699                 for window in windowArray:
700                         Gx = window.prewittGX()
701                         Gy = window.prewittGY()
702                         G  = sqrt(Gx**2 + Gy**2)
703                         if G > 255:
704                                 G = 255
705                         elif G < min:
706                                 G = 0
707
708                         newImage[window.x, window.y] = G
709                         thetaImage[window.x, window.y] = 90+(90*atan2(Gy, Gx)/np.pi)
710
711                 # Create a RGB version of the image, convert to HSV and replace the H values with that
                        ↪ of the theta values
712                 # Saturation set to max for visibility. Converted back to RGB for display
713                 BGR = np.dstack((newImage, newImage, newImage))
714                 HSV = cv2.cvtColor(BGR, cv2.COLOR_BGR2HSV)
715                 HSV[:,:,0] = thetaImage
716                 HSV[:,:,1] = 255- blankImage
717                 final = cv2.cvtColor(HSV, cv2.COLOR_HSV2BGR)
718
719                 # Update the image with the edge detected image
720                 self.updateImage(final, 'EDGEprewitt')
721
722         def EDGEgradientsSobel(self,min):
723                 self._genBorders(2)
724
725                 imageM, imageN = self.imageSize()
726
727                 windowArray = self.createWindows(1)
728
729                 newImage = np.zeros([imageM, imageN],np.uint8)
730                 thetaImage = np.zeros([imageM, imageN],np.uint8)
731                 blankImage = np.zeros([imageM, imageN],np.uint8)
732
733                 # For each window run the two directions of the edge detectors
734                 for window in windowArray:
735                         Gx = window.sobelGX()
736                         Gy = window.sobelGY()
737                         G  = sqrt(Gx**2 + Gy**2)
738                         if G > 255:
739                                 G = 255
740                         elif G < min:
741                                 G = 0
742
743                         newImage[window.x, window.y] = G
744                         thetaImage[window.x, window.y] = 90+(90*atan2(Gy, Gx)/np.pi)
745
746                 # Create a RGB version of the image, convert to HSV and replace the H values with that
                        ↪ of the theta values
747                 # Saturation set to max for visibility. Converted back to RGB for display
748                 BGR = np.dstack((newImage, newImage, newImage))
749                 HSV = cv2.cvtColor(BGR, cv2.COLOR_BGR2HSV)
750                 HSV[:,:,0] = thetaImage
```

```
751                 HSV[:,:,1] = 255- blankImage
752                 final = cv2.cvtColor(HSV, cv2.COLOR_HSV2BGR)
753 
754                 # Update the image with the edge detected image
755                 self.updateImage(final, 'EDGESobel' +str(min))
756 
757 
758 
759 
760         ##########################################
761         ##                                      ##
762         ## Methods (Filter Applications)        ##
763         ##                                      ##
764         ##########################################
765 
766         def highlightRange(self,lBound,uBound,dcGain,levels):
767                 # Function to take all values in the range [lBound uBound] and remap them
768                 # to the range [dcGain dcGain+levels]
769                 mask = self.PObitSlice(lBound,uBound)
770                 rangePixels = self.duplicate()
771                 rangePixels.mask(mask)
772                 rangePixels.POequalise(levels,[0],dcGain)
773                 mask.PONegImage()
774                 self.overlay(rangePixels,mask)
775 
776         def VisualiseFFT(self):
777                 # Example function to visualise the FFT of an image
778                 self.POfftMag()
779                 self.PONormalise(0,255)
780 
781 
782 def __printSpacer__(*args):
783         # Function created to print a line of asterix, made seperate to make code neater
784         if (0 == len(args)): # If no arguments are included then print an asterix line spacer
785                 print('**************************************************')
786                 print('')
787         else:                           # If arguments are provided then print the argument surrounded
              ↪ by asterix'
788                 for i in range(len(args)):
789                         print('**************************************************')
790                         print('')
791                         print(args[i])
792                         print('')
```