Digital Image Processing

EE40054

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

Author:
Alexandros Vlissidis

Lecturer:
Dr. Adrian Evans

November 23, 2016

Contents

1 Introduction

This assignment explores the use of linear, non-linear and morphological filters on various images. These filters can be used to remove impulsive noise from images to prepare them for edge detection and feature extraction. This report focuses on two images, one ultasound image of a foetus, and a SAR image of New Zealand. The functions described in this report assume double images as inputs, in order to make use of MATLAB's statistics functions.

2 Implementation

2.1 Linear Filters

Linear filters use convolution in order to smooth an image. This efffectively removes the noise from an image and improves the result of edge detection algorithms, by convolving the input image with the impulse response of the filter. The negative effects of linear filters is that they also blur the image, thus removing detail, and have poor performance at the presence of impulsive noise. The general formula of linear filters is given in the equation below.

$$y(k) = \frac{\sum_{i=1}^{N} \alpha_i x(i)}{\sum_{i=1}^{N} \alpha_i}, x(i) \in W_x \tag{1}$$

Where $x(i)$ are the input image pixels and α are the weights, which are multplied with the pixel values. The output pixel value is $y(k)$. Equation (1) has been implemented in MATLAB as a generic function that can be used by all linear filtes in order to reduce code duplication. The main loop of this function is shown below.

```
for i = (1+rpad):(rows-rpad)
    for j = (1+cpad):(cols-cpad)
       % Extract the window area
       window = original((i-rpad):(i+rpad), (j-cpad):(j+cpad));
       if wsum ~= 0
            out(i, j) = sum(sum(window.*kernel))/wsum;
       else
           % Dont divide by the sum of the weights
           out(i, j) = abs(sum(sum(window.*kernel)));
       end
    end
end
```

Listing 1: Apply Linear Filter MATLAB Function

This function takes as input the kernel and the original image. It scans the window across the image, convolving the kernel with the window. In this way, this function can be used by any linear filter, as long as they provide an image and a kernel or correct dimensions. However, in some filters the sum of the weights adds up to zero. Hence, there is a check for that case.

2.1.1 Mean Filter

The simplest linear filter is the mean filter. This filter replaces the pixel value of the output image with the average of the window. Effectively, this smoothes the edges, as each pixel value is equally affected from edges, as well as flat areas. The averaging filter can be implemented if the sum of the weights is equal to 1 and each pixel value is divided by the number of pixels in

the window. The equation for a 3x3 filter is shown below.

$$output = \frac{1}{9} \sum_{i=-1}^{1} \sum_{j=-1}^{1} f(x+i, y+j) \tag{2}$$

The kernel has 9 elements, and spans 1 element around the central index. Equation (2) has been implemented in MATLAB.

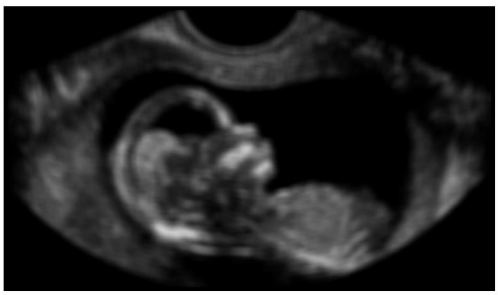
```
function [ out ] = mean_filter( original, size )

weight = 1/(size^2);
kernel = ones(size, size) * weight;

out = apply_linear_filter(original, kernel);

figure; imshow(out)

end
```

Listing 2: Mean Filter MATLAB Function

The code for the mean filter implementation has been simplified a lot by the use of the function described in Listing 1. The mean filter builds a kernel with weights that sum up to one. So each weight must have the value of one over the number of cells in the kernel. The results of this filter are illustrated below.

(a) No smoothing

(b) Kernel size 5x5



(c) Kernel size 13x13

Figure 1: Mean Filter Results

It can be seen from the figures that as the kernel size increases more smoothing is achieved. This is because more of the image is taken into account in the averaging, blurring the edges even more.

2.1.2 Gaussian Filter

The Gaussian filter is a linear filter whose impulse response is a Gaussian function. This means it has a high output at the center and low away from it. This is a smoothing filter as the output value is affected by the neighbouring pixels. In the context of a 2D image, this translates to having high pixel values at the center of the image and low values as you move away from the center of the window. The equation describing a 2D gaussian function is given below.

$$g(x, y) = e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{3}$$

Where x and y are the indices of the elements, relative to the central value in the window and σ is the standard deviation desires, which is essentially the order of the filter. The size of the gaussian mask is set to $2(3\sigma) + 1$. This filter has been implemented as a MATLAB function below.

```
function [ out ] = gaussian(original, sigma)

kernel = make_gaussian_kernel(sigma);
out = apply_linear_filter(original, kernel);

figure; imshow(out)

end
```

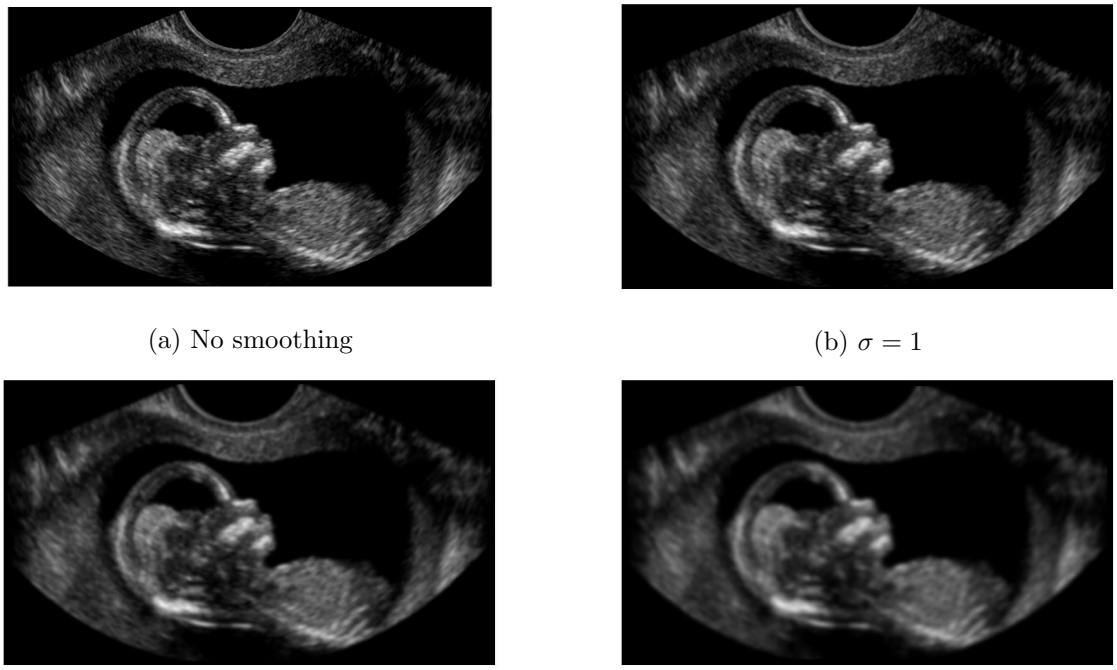
Listing 3: Gaussian Filter MATLAB Function

This code is severely simplified due to the use of the general linear filter function and a factory method to create a gaussian kernel. This is shown below.

```
function [ kernel ] = make_gaussian_kernel( sigma )

% Set the size of the kernel
ksize = 2 * (3 * sigma) + 1;
kernel = zeros(ksize, ksize);

% Calculate the padding
pad = ceil(ksize/2);

% Populate the kernel
for i = 1:ksize
    for j = 1:ksize
       % Translate the indices relative to the central cell
        kernel(i, j) = gauss(i-pad, j-pad, sigma);
    end
end

end

function [ value ] = gauss(x, y, sigma)
    value = exp(-(x^2 + y^2)/(2*sigma^2));
end
```

Listing 4: Apply Linear Filter MATLAB Function

Here the function takes the standad deviation of the filter as an input. From that it can infer the size of the kernel needed. The kernel is populated by using a gaussian function which takes the indices of the cell relative to the central value, as input, as well as the standard deviation. The results of this filter are shown below.



(a) No smoothing

(b) $\sigma = 1$

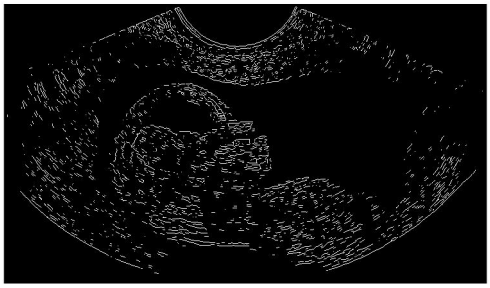
(c) $\sigma = 2$

(d) $\sigma = 3$

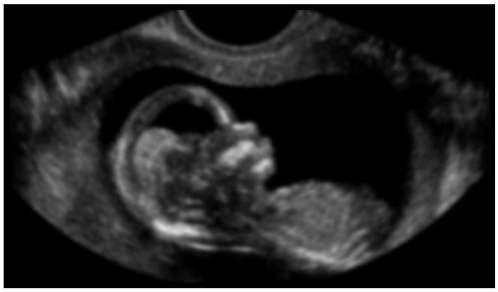
Figure 2: Gaussian Filter Results

The result seem to show what is expected. Since this is a smoothing filter, as the standard deviation increases the image blur is increased, as the kernel size increases and a larger area is taken into account in the convolution. The purpose of applying a smoothing filter is to improve edge detection, since together with the edges, the noise of the image is smoothed as well. The improvement in edge detection is shown below.

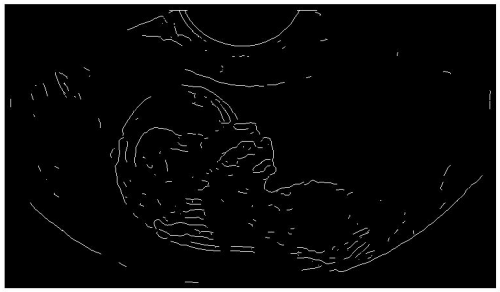
(a) No smoothing



(b) Edge detection with no smoothing



(c) $\sigma = 3$



(d) Edge detection for $\sigma = 3$

Figure 3: Gaussian Filter Results

It is clear from this test that this gaussian smoothing filter has a desirable effect of edge detection. As smoothing is applied, the noise in the image is less prominent, and hence less edges are detected compared to the original image.

2.1.3 Frequency Domain Filtering

All the filters explored until now have been in the spatial domain. We can apply filters in the frequency domain as well. Because convolution in the frequency domain translates to multiplication, it becomes a much simpler operation to handle. Especially, in MATLAB which uses vectorisation to accelerate the results.

In order to apply a filter in the frequency domain, the Fourier Transform of the image must be calculated. This is a compute intensiver process. However, if the dimensions of our image are a power of two, then the much faster FFT algorithm can be applied. In this case, a Butterworth low-pass filter has been implemented. A low-pass filter will effectively cut-off the edges, which are the high frequency components of the image, as the pixel values change rapidly at the edges. The equation of a 2D Butterworth filter in the frequency domain is shown below.

$$H(u,v) = \frac{1}{1 + (\frac{r}{F_c})^{2p}}, u = 0, 1, ..., M-1, v = 0, 1, ..., N-1 \tag{4}$$

Where F_c is the cutoff frequency of the filter. This is the frequency where the filter's response has reached half of its maximum value. The order of the filter p controls the smoothness of the filter. Decrasing the order of the filter increases the smoothing applied. Finally, r is the 2D version of the distance from the image's DC component.

$$r = \sqrt{(u-u_o)^2 + (v-v_o)^2} \tag{5}$$

Where u_o and v_o are the indices at the DC component of the image. This equation corresponds to a rotation of the 1D version of the filter, which generated a circular region where the filter applies. The frequency response of this filter is illustrated below.

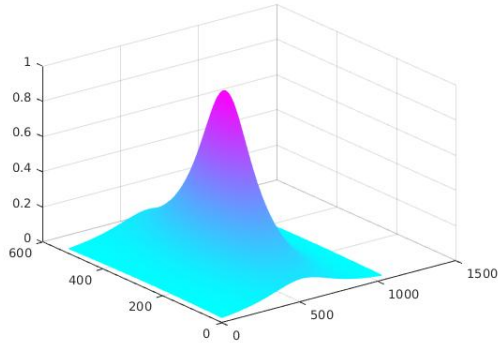


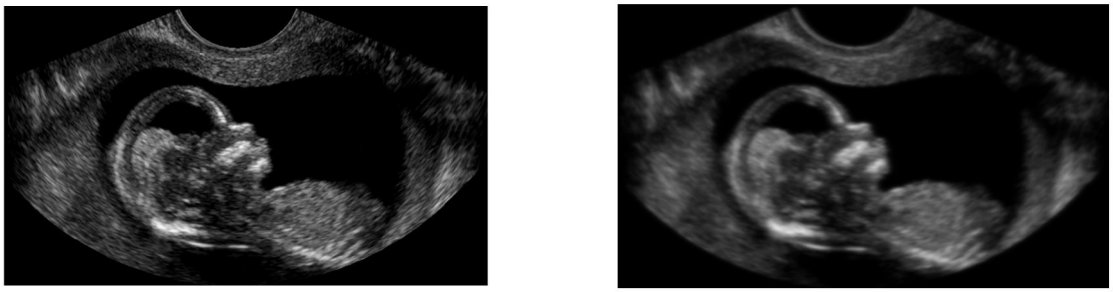
Figure 4: Butterworth Low-Pass Filter Response

This filter has been implemented in MATLAB.

```
function [ out ] = blpf( original, order, cutoff )

% Work out how much we need to pad to apply FFT
[r, c] = size(original);
m = nextpow2(r);
n = nextpow2(c);

% FFT the image and shift it so the zero frequency is aligned
F = fftshift(fft2(original, 2^m, 2^n));

[rows, cols] = size(F);
H = zeros(rows, cols);
rcenter = rows/2;
ccenter = cols/2;

% Build the mask for the FFT domain
for u = 1:rows
    for v = 1:cols
        H(u,v) = 1/(1 + (sqrt(((rcenter-(u-1))^2 + (ccenter-(v-1))^2))/cutoff)
    .^(2*order));
    end
end

% Apply the filter and go back to the spatial domain
G = ifft2(H.*F);
out = sqrt(real(G).^2 + imag(G).^2);

% Crop the result to the initial size
out = out(1:r, 1:c);

figure; imshow(out)
figure; mesh(1:cols, 1:rows, H);
colormap(cool(100))

end
```

Listing 5: Butterworth Low Pass Filter MATLAB Function

As mentioned before in order to apply an FFT algorithm, the size of the image must be a power of 2. Hence the first thing the function does is to work out the size the image needs to be, and passes that to MATLAB's *fft2* function which pads it with zeros to reach that size. Then we must shift the result so that the DC component aligns with the center of the image. This is achieved with MATLAB's *fftshift* function. Then the mask is populated using Equation (4). The much faster multiplication of the mask H and the image F, gives us our result. Then we must translate back to the spatial domain with an inverse FFT function, and crop the image to the initial size. The result of this low pass filter is shown below.



(a) No smoothing

(b) $p = 1$, $F_c = 50$

Figure 5: Butterworth Low-Pass Filter Results

It is obvious that the high frequencies have been cut out and this has blurred the image. However, in order to improve edge detection a high pass filter would be more appropriate, since that would in theory only allow the high frequencies. This can be achieved using the same code, only changing the response of the filter to be $1 - H(u, v)$. The response of a Butterworth high-pass filter is shown below.

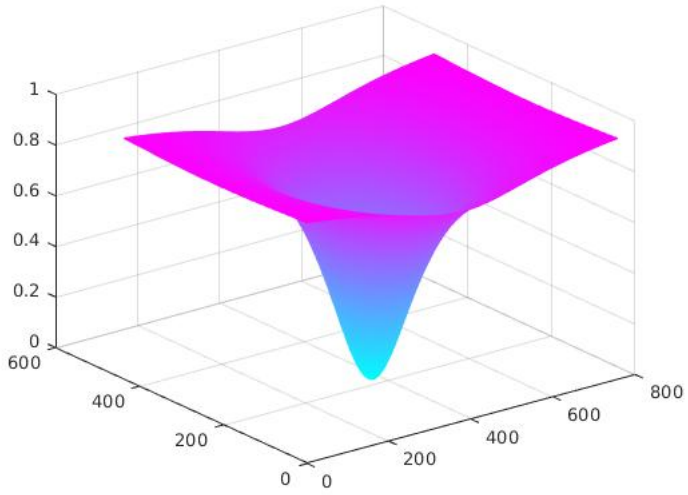
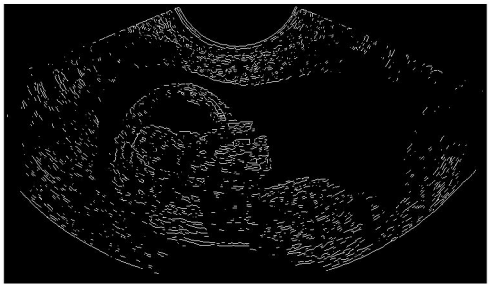


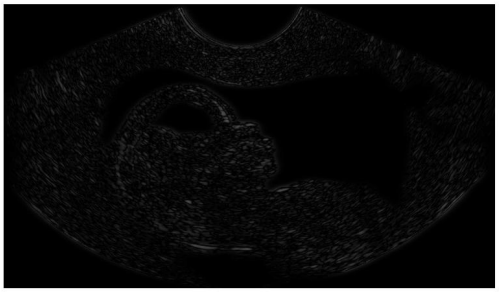
Figure 6: Butterworth High-Pass Filter Response

The edge detection performance of this filter is shown below.

(a) Original



(b) Original Edge Detection



(c) Filtered Image

(d) Filtered Edge Detection

Figure 7: Butterworth High-Pass Filter Edge Detection Results

It can be seen that the edge detection is improved after we apply the high-pass filter, however, still a lot of noise is present and detected as an edge. The Gaussian filter had a better performance in edge detection.

2.1.4 Unsharph Masking Filter

This filter is an adaptive linear filter. It adapts its weights according to the statistics of the local window. As a result this has better performance at the edges of the image. The equation of the filter is shown below.

$$\hat{x} = \bar{x} + k(x - \bar{x}) \tag{6}$$

Where $\hat{x}$ is the output value of the filter, $\bar{x}$ is the mean value of the window, and k is a constant in the range [0, 1], which controls the aggresiveness of the filter. A value of 1 will not smooth at all, and a value of 0 will essentially apply a mean filter. Hence, the value of k is inversely proportional to the SNR, which is equal to $\bar{x}/\sigma$. When k is high, the SNR is low, hence the image is flattened. This has been implemented in MATLAB below.

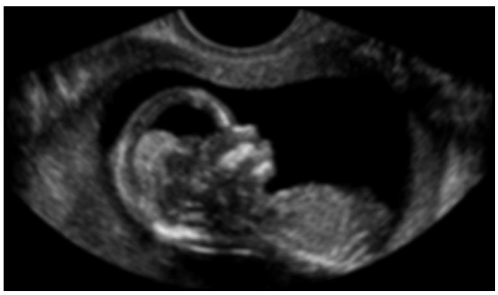
```
function [ out ] = unsharp_masking( original, ksize, k )

[rows, cols] = size(original);
out = zeros(rows, cols);

rpad = (ksize-1)/2;
cpad = (ksize-1)/2;

for i = (1+rpad):(rows-rpad)
    for j = (1+cpad):(cols-cpad)
        % Extract the window area
        window = original((i-rpad):(i+rpad), (j-cpad):(j+cpad));
        m = mean(window(:)); % Calculate the mean

        % Apply the adaptive filter function
        out(i, j) = m + k * (original(i, j) - m);
    end
end

figure; imshow(out)

end
```

Listing 6: Unsharp Masking Filter MATLAB Function

This implementation is quite straightforward, and does not use the general *apply_linear_filter* function, as the output value depends on the local statistics of the window. The results of this filter for a mask size of 9, are shown below.

(a) Original

(b) $k = 0.9$



(c) $k = 0.1$

Figure 8: Unsharp Masking Filter Results

It is obvious that with a value close to 1, no smoothing is performed. However, with $k = 0.1$, the

image is greatly blurred.

2.2 Non-Linear Filters

After exploring a wide range of linear filters, there were a few non-linear filters implemented as well. More specifically, four versions of median filters were implemented. In general, median filters are most effective in removing impulsive noise, while preserving the edges of the image, opposite to linear-filters, which blur the edges and are more effective at removing multiplicative noise. The general formulation for a non-linear filter is shown in the equation below.

$$y(k) = \frac{\sum_{l=1}^{N} \alpha_l x_l(i)}{\sum_{l=1}^{N} \alpha_l}, x_l(i) \in W_x \tag{7}$$

Where $x_l(i)$ is the l^{th} smallest value in the window, and α_l is the weight for the l^{th} position. Hence for this equation to apply the pixel values of the window must be sorted and then the weights must be applied. However, in the non-linear version, the weight do not correspond to a multiplication, but represent the number of times that pixel value must be inserted in the sorted list. After, this is applied, the median of the final list is the outputs value of the filter, $y(k)$. This has been implemented as a generic function, in MATLAB below. This allows for more code re-use between all the median filters.

```
function [ out ] = apply_nonlinear_filter( original, weights, ksize )

[rows, cols] = size(original);
out = zeros(rows, cols);

rpad = (ksize-1)/2;
cpad = (ksize-1)/2;

for i = (1+rpad):(rows-rpad)
    for j = (1+cpad):(cols-cpad)
        % Extract the window
        window = original(i-rpad:i+rpad, j-cpad:j+cpad);

        % Sort the window
        mask = sort(window(1:end));

        list = []; % Empty the final list
        for k = 1:length(weights)
            % Build the temporary list for this value and weight
            buffer = ones(1, weights(k)) * mask(k);

            % Append the temporary list to the final one
            list = [list, buffer];
        end
        out(i, j) = median(list);
    end
end

end
```

Listing 7: Non-Linear Filter MATLAB Function

In this implementation at temporary buffer is built, to hold the number of values needed to be inserted in the final list, which has the size of the weight for that specific pixel value. This can easily be used by any non-linear filter, by just providing a list of the weights.

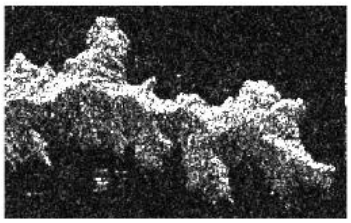
2.2.1 Median Filter

The Median filter is the simplest of the non-linear filters. It replaces the output pixel value with the median of the window. This can be very easily implemented using the *apply_nonlinear_filter* function described in Listing 7. Essentially, this can be achieved if all the weights are zero, except from the middle element. This will eliminate all the values except from the median. This implementation is shown below.

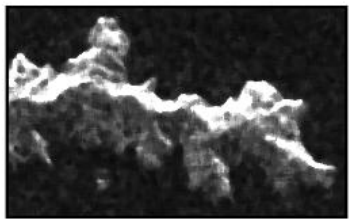
```
function [ out ] = median_filter( original, ksize)

length = ksize^2;

% Everything is 0 except from the middle element
weights = zeros(1, length);
weights(1, (length-1)/2) = 1;

out = apply_nonlinear_filter(original, weights, ksize);

figure; imshow(out)

end
```

Listing 8: Median Filter MATLAB Function

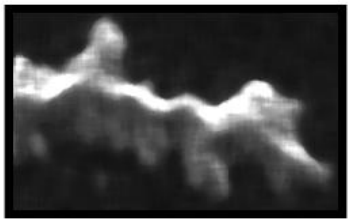
The results of this filter can be shown using the New Zealand SAR image, which contains impulsive (spekle) noise.



(a) Original



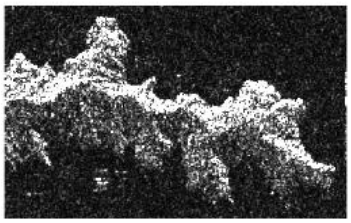
(b) kernel 5x5



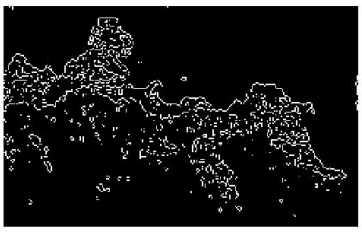
(c) kernel 13x13

Figure 9: Median Filter Results

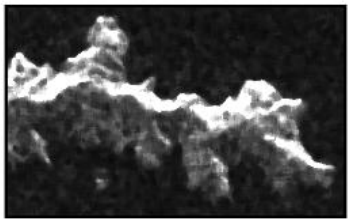
It can be seen that as the size of the kernel increases more smoothing is applied. However, even in the extreme case the edges of the image are preserved and the impulsive noise in the flat regions is completely removed. In theory this will enhance edge detection. This is illustrated below.



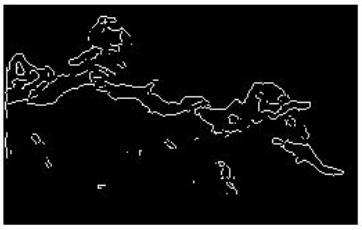
(a) Original



(b) Original edge detection



(c) kernel 5x5



(d) kernel 5x5 edge detection

Figure 10: Median Filter Edge Detection Results

It is clear that with the median filter applied, the outline of the island can be detected much more clearly, excluding the noise in the image.

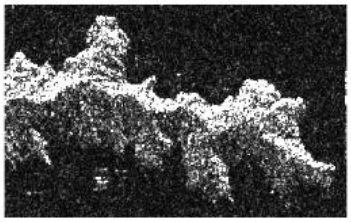
2.2.2 Weighted Median Filter

The weighted median filter is a more general case of the median filter. In fact, the median filter is a special case of the weighted median filter. This gives some flexibility in our filtering. The weights of the filtered can be adjusted to a specific application of the filter. The centre weighted median filter has all its weights set to 1 except from the middle weight which is always greater than 1. This filter constructs the weights vector and passes it to the *apply_linear_filter* function. The code is shown below. This simplifies the implementation.

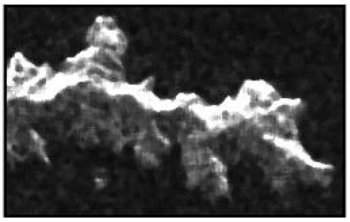
```
function [ out ] = weighted_median( original, ksize )

length = ksize^2;
weights = ones(1, length);
weights(1, (length-1)/2) = 3; % Set the middle weight

out = apply_nonlinear_filter(original, weights, ksize);

figure; imshow(out)

end
```

Listing 9: Median Filter MATLAB Function

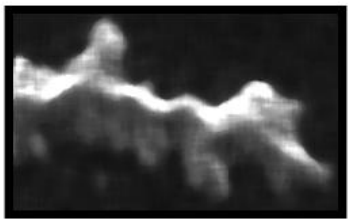
This implements a filter with all the weights set to 1 and the middle weight to 3. The results of this function are illustrated below.



(a) Original



(b) kernel 5x5



(c) kernel 13x13

Figure 11: Center Weighted Median Filter Results

We can see that the center weighted median filter has the same results with a simple median filter. We can then safely assume that it has the same ability to remove impulsive noise and thus improve edge detection algorithms.

2.2.3 Adaptive Weighted Median Filter

In the adaptive weighted median filter, the weight for a specific pixel value in the window depends on the image statistics of the local window area. The formula describing the weight formation for a given window is shown below.

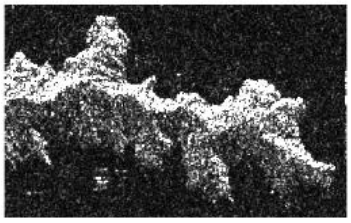
$$w_{i,j} = round(W_{(k+1,k+1)} - \frac{cd\sigma}{\bar{x}}) \tag{8}$$

Where $W_{(k+1,k+1)}$ is the central weight of the window, c is a constant, which controls the aggresiveness of the filter, d is the euclidean distance of the coordinates of the element from the central element in the window, σ is the standard deviation of the window. Then the result is rounded. The implementation is included below.

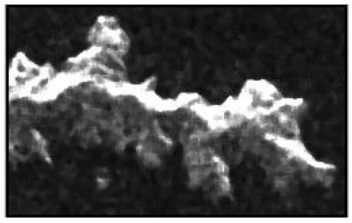
```
function [ out ] = adaptive_weighted_median( original, ksize, cweight, c )

[rows, cols] = size(original);
out = zeros(rows, cols);

rpad = (ksize-1)/2;
cpad = (ksize-1)/2;
d = zeros(ksize, ksize);
mpos = ceil(ksize/2); % Work out the middle matrix element

% Pre-store the distance matrix
for i = 1:ksize
    for j = 1:ksize
        d(i, j) = distance([i j], [mpos mpos]);
    end
end

for i = (1+rpad):(rows-rpad)
    for j = (1+cpad):(cols-cpad)
        list = [];
        window = original(i-rpad:i+rpad, j-cpad:j+cpad);
        window = sort(window(1:end));

        sigma = std(window);
        x = mean(window);

        % if the mean of the window is zero skip this iteration
        if x == 0
            out(i, j) = x;
            continue
        end

        % Compute the weights for this window
        idx = 1;
        for k = 1:ksize
            for q = 1:ksize
                % Calculate weight for this element
                weight = round(cweight - (c * d(k, q) * sigma)/x);

                % Create the new values to insert
                buffer = ones(1, weight) * window(idx);
                idx = idx + 1;

                % Insert new values
                list = [list, buffer];
            end
        end
        out(i, j) = median(list);
    end
end
figure; imshow(out)
end

function  [ d ] = distance(x, y)
% Calculate the euclidean distance between two vectors
d = sqrt(sum((x - y).^2));
end
```

Listing 10: Median Filter MATLAB Function

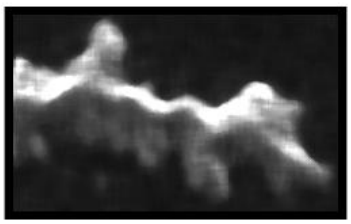
This implementation build a matrix with the distances from the middle element, as they are constant and do not need to be calculated in every loop iteration. This is stored in matrix *d*. Then the window area is extracted and sorted. The next step is to calculate the statistics of the window, but if the mean is zero, the iteration is skipped, as the weight equation would have a division by zero. For each weight a buffer is built, of size *weight* and values of the corresponding pixel in the window. It is much faster to built a buffer and then contatinate with a final list, rather than inserting new values one at a time. In addition, this speeds up the sorting. Since the window is already sorted, we do not have to sort again for the larger final list, as the values will still be in order. The results of this filter are shown below.



(a) Original



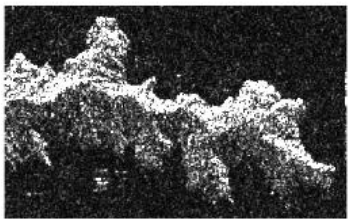
(b) kernel 5x5, $cweight = 100$, $c = 10$



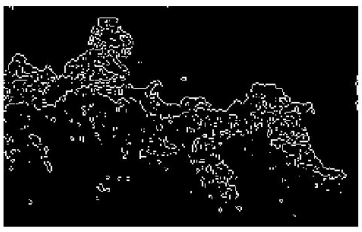
(c) kernel 13x13, $cweight = 100$, $c = 10$

Figure 12: Adaptive Weighted Median Filter Results

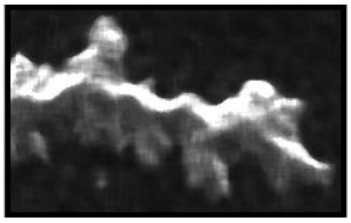
It is obvious that this median filter has a really good performance at removing spekle noise. As the kernel size increase almost all the noise is gone, but the edges are still preserved. Some detail is still lost, however.



(a) Original



(b) Original edge detection



(c) kernel 9x9

(d) kernel 9x9 edge detection

Figure 13: Adaptive Weighted Median Filter Edge Detection Results

Since the median filter has removed most of the spekle noise the edge detection algorithm can more clearly find the edges of the island. The dark side, however, is aparently too dark to be detected, i.e. it has too low intensity to be perceived as an edge. However, the adaptive weighted median filter has improved the edge detection.

2.2.4 Truncated Median Filter

The final median filter implemented is the truncated median filter, or mode filter. This filter tries to approximate the mode of the window, and replace the output image pixel with that value. This filter is based on the fact that the median lies between the mode and the mean.

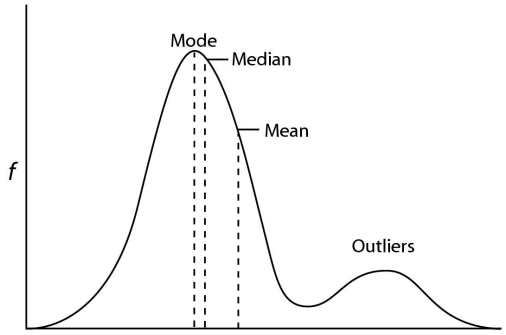


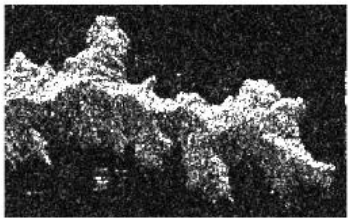
Figure 14: Mean, Median, Mode Order

Because the window has a small number of values in it, there is a high probability that there is not going to be a mode, as all the values will probably be unique. Hence the mode filter tries to approximate the mode by cutting the extreme values, and taking the median of what is left. This has been implemented in MATLAB below.

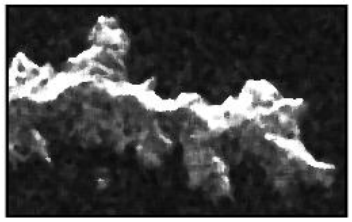
```
function [ out ] = truncated_median( original, ksize )

[rows, cols] = size(original);
out = zeros(rows, cols);

rpad = (ksize-1)/2;
cpad = (ksize-1)/2;

for i = (1+rpad):(rows-rpad)
    for j = (1+cpad):(cols-cpad)
        window = original(i-rpad:i+rpad, j-cpad:j+cpad);
        window = sort(window(1:end));

        med = median(window);

        % Calculate the distances from the median
        upperd = max(window) - med;
        lowerd = med - min(window);

        % Work out the cutoff value
        if upperd > lowerd
            distance = upperd;
        elseif upperd < lowerd
            distance = lowerd;
        end
        % Cut the extreme values
        window = window(abs((window - med)) < distance);

        % The new median must approximate the mode
        out(i, j) = median(window);
    end
end

figure; imshow(out)

end
```

Listing 11: Median Filter MATLAB Function

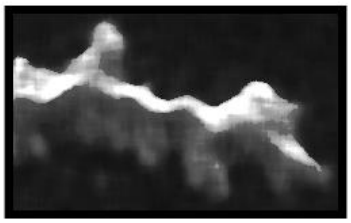
This function calculated the median and then the distance of the median from minimum and maximum values in the window. The smaller distance of the two is kept as a cutoff value. The window values that are kept are the one's whose distance from the median is less that the cutoff distance. The median of the new window must approximate the mode more accurately.



(a) Original



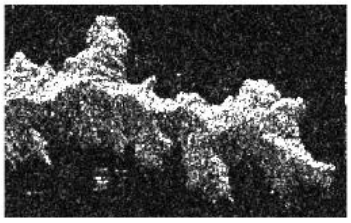
(b) kernel 5x5



(c) kernel 13x13

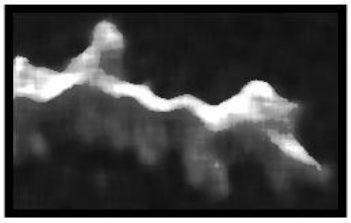
Figure 15: Truncated Median Filter Results

It can be seen that this filter is a bit more agressive at the edges than the median filters. It is able to remove impulsive noise as well as all the median filter. However, the edges appear to be more sharp, even at large mask sizes. We can assume that this will aid edge detection even more.

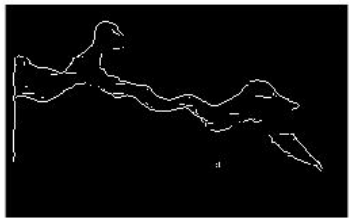


(a) Original

(b) Original edge detection



(c) kernel 13x13



(d) kernel 13x13 edge detection

Figure 16: Truncated Median Filter Edge Detection Results

It is clear that the truncated median filter not only improves the edge detection with respect to the original image, but also it has better performance that the adaptive weighed median filter. Less of the noise is perceived as edges, and generally the edges are more clear. It still cannot

make the edge detection algorithm perceive the dark side of the island as an edge.

2.3 Morphological Filters

Morphological filter are a type of non-linear filters that alter the shapes of objects to enhance edge detection, by removing noise from the image. The morpholohical filter uses a structuring element to select the part of the window to take into account in the filtering. The elements selected with the structuring element are sorted and then a specific position is selected as the output. This position is taken as an input. This filter has been generically implemented in MATLAB, and can be used for both erosion and dilation.

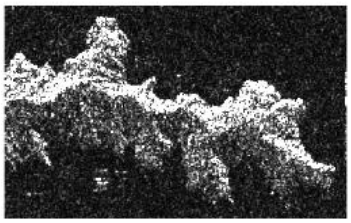
```
function [ out ] = morph( original, se, order )

[rows, cols] = size(original);
[serows, secols] = size(se);

out = zeros(rows, cols);

rpad = (serows-1)/2;
cpad = (secols-1)/2;

for i = (1+rpad):(rows-rpad)
    for j = (1+cpad):(cols-cpad)
        % Extract the window
        window = original(i-rpad:i+rpad, j-cpad:j+cpad);

        % Keep only the elements that matter and sort them
        window = sort(window(se ~= 0));
        out(i, j) = window(order); % Choose the specified position
    end
end
figure; imshow(out)
end
```

Listing 12: Median Filter MATLAB Function

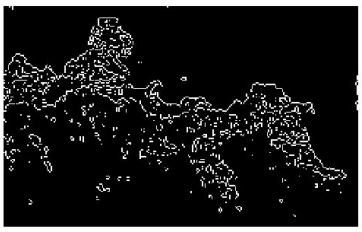
It can be seen that after the window is extracted, the pixel values to be sorted are selected using logical indexing according to the structuring element, which is a simple matlab matrix, with ones in the positions that matter and zeros elsewhere. Hence, this function is generic enough to be used with any structuring element as well as for erosion and dilation. The order of the filter will determine whether erosion or dilation will take place.

2.3.1 Erosion

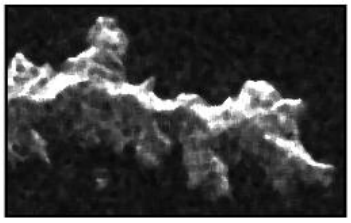
In order to perform erosion, the function described in Listing 12 is used, with a low order value. This means that the output value of the filter will be in the low values of the window. Hence, at the edges, which have high intensity, the shape of an object will shrink. The output image will take the value of the flat regions which have lower pixel values. The erosion function can successfully remove spekle noise from the *nzers1.jpg* image.



(a) Original



(b) Original Edge Detection



(c) Erosion, se = 5x5, order = 10

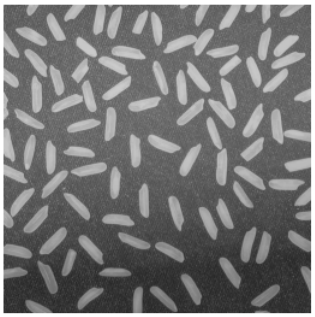
(d) Erosion Edge Detection, se = 5x5, order = 10

Figure 17: Morphological Erosion Results

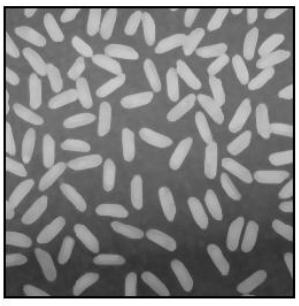
The Erosion filter successfully removed the spekle noise from this image, thus allowing for a much better edge detection that without any filtering. In fact, it performs particularly, well. This is because is resembles the function of a median filter. The difference is that it has a more flexible structuring element structure, which allows to look for specific shapes, and a more flexible output pixel, as it does not have to be the median of the window.

2.3.2 Dilation

Dilation is a filter whose effect if opposite to erosion. It changes the shapes in an image, by expanding its edges. This can help in repairing breaks in images or intrusions. For example, it an image containing a handwritten character, which has breaks, a dilation can repair this hole. This is done by using the same technique as erosion. However, this time, the output value of the filter is selected from the high values of the window. Hence, the edges of objects, which have high values, are expanded into the flat regions, with low pixel values. This can be illustrated using the *rice.png* MATLAB image.



(a) Rice



(b) Rice Dilation, se = 5x5, order = 20

Figure 18: Morphological Dilation Results