Automatic Interpretation of Synthetic Aperture Radar (SAR) and Ultrasound Images

ELEC0054

Digital Image Processing

Chris Cox
I declare all material contained within this report and the source code associated with it, is my own work, except where otherwise stated.

Signed………………………..   Date……………
Contents

Contents .....................................................................................................................3
Introduction ................................................................................................................4
Image Filtering ..........................................................................................................4
  Mean Filter .........................................................................................................4
  Basic Median Filter ..........................................................................................5
  Weighted Median Filter ....................................................................................8
  Adaptive weighted median filter .......................................................................9
  Laplacian of a Gaussian .....................................................................................12
Edge Detection for heart.pgm image .....................................................................13
  Prewitt Templates ............................................................................................13
  Sobel Templates ...............................................................................................15
  Laplacian Operator ..........................................................................................17
Edge Detection for nzjers.pgm .............................................................................18
Appendix .................................................................................................................19

List of Figures

Figure 1 Square Median Window ............................................................................5
Figure 2 Vertical/horizontal line-preserving neighbourhood for median filter ..........7
Figure 3 Possible mask weightings .........................................................................9
Figure 4 Gaussian hump with mean(0,0) and $\sigma=1$ ..........................................12
Figure 5 2D Gaussian distribution equation ............................................................12
Figure 6 Laplacian of a Gaussian mask for $\sigma=1.0$ .............................................12
Figure 7 Prewitt template arrangement ..................................................................14
Figure 8 Sobel filter template arrangement .............................................................16
Figure 9 Laplacian operator mask for 8 neighbourhoods .......................................17
Introduction

This coursework is part of the Digital Signal Processing course (ELEC0054). The aim is to automatically identify the boundaries within the images, nzjers.pgm and heart.pgm, using an edge detection method. To achieve this the task has been split into two stages: Image Filtering and Edge Detection. The associated code is written in the C programming language and is also included with the filter descriptions. For completeness, the entire source code for the project can be found in the appendix. The report will look in turn at each of the filters implemented and discuss their effects on the images.

Image Filtering

For the image filtering section I will concentrate on showing the results gained using the nzjers.pgm image, as speckle noise is very evident. Also the size of the image allows the results to be displayed more accurately.

Mean Filter

Initially we will look at the results gained from using a mean filter. The mean filter is one of the crudest types of image filtering techniques available. The effect of this filter type is to remove detail and noise from an image. A 3 by 3 window is taken from the original image and all the pixel values are summed and then divided by the number of values (i.e. 9). This mean value then replaces the value of the centre of the 3 by 3 window in the new image.

The code fragment for this particular type of filter is below:

```c
/*----------------------------------------------
*Function:filter_mean
*Inputs:None
*Dependencies:None
*Description: Simple 3by3 mean filter, which sums contents of window, the divides
* by number of values (i.e. 9). The calculated mean is the placed into
* the second image.
*----------------------------------------------*/
void filter_mean(void)
{
  for (i=2;i< (num_cols -2);i++)
    for (j=2;j< (num_rows -2);j++)
      image2[i][j]=((image1[i-1][j-1]+image1[i][j-1]+image1[i+1][j-1]+
                     image1[i-1][j]+image1[i][j]+image1[i+1][j]+
                     image1[i-1][j+1]+image1[i][j-1]+image1[i+1][j-1])/9);
}
```

The resulting image from this technique is:-
It is clearly visible that the noise has been removed from the image, but there has also been a loss of detail (blurring). This is to be expected, as any edges existing within the original image will have been averaged. An image with a large number (high frequency) of edges would be almost unrecognisable after applying this technique.

**Basic Median Filter**

Median filtering is a smoothing method that reduces the blurring of edges. The idea is to replace the current point in the image by the median of the brightness in its neighbourhood.¹

Initially a square window was used as shown in Figure 1. The window is placed over the original image. The median of the pixel values is taken and this value is then placed into position “E” in the created image. The window then moves to the next pixel location and the process is repeated.

\[
\text{Median} = \text{Median}\{A, B, C, D, E, F, G, H, I\}
\]

\[
\frac{2}{2}
\]

*Figure 1 Square Median Window*

The code fragment for this filter is below:

---

Function: filter_median
Inputs: None
Dependencies: None
Description: Simple 3by3 median filter, using a square neighbourhood window.
Median is value replaces original in created image

void filter_median(void)
{
    int num_in_array=9;/*gives length of array*/
    int cnt;/*index for test option*/

    for (i=2;i< (num_rows -2);i++)
        for (j=2;j< (num_cols -2);j++)
        {
            int window_values[9] = {image1[i-1][j-1], image1[i][j-1], image1[i+1][j-1],
                                    image1[i-1][j], image1[i][j], image1[i+1][j],
                                    image1[i-1][j+1], image1[i][j-1], image1[i+1][j-1]};/*square shaped mask*/
            if (debug_test==1) {/*test option*/
                printf("Unsorted array../
                for(cnt=0; cnt<num_in_array; cnt++) {
                    printf("%d, ", window_values[cnt]);
                }
            }
            image2[i][j]=sort(window_values, num_in_array); /*sorts array and creates new pixel value*/
        }

The above code utilises a “sort” function, the code for this is below. It is necessary to
arrange the values in size order so that the median can be collected from the n\textsuperscript{th}/2
value. This function will accept an array of any length, which will be useful in future
filter processes, although longer arrays will result in an increased computational load.

Function: sort
Inputs: array[], n
Dependencies: None
Returns: int median
Description: Sorting mechanism returning the median, run for each window. Works
with array of any length

int sort(int array[], int n)
{
    int this, next, temp, cnt, median;/*local variables*/
    for (this=0; this<n; this++)/*index marker for array sorting*/
        for (next=this+1; next<n; next++)/*index marker for array sorting*/
            if (array[this]>array[next]){
                temp=array[this];/*arranges values, smallest at beginning*/
                array[this]=array[next];/*largest at end*/
                array[next]=temp;/*sorted value put in array*/
            }
    for(cnt=0; cnt<n; cnt++) {
        /*Now check median postn is whole*/
        if ((n % 2)==0){
            median=array[n/2];/*if no remainder*/
        else {
            temp=((n/2)-0.5);
            temp=(array[temp] + array[temp+1]);
            median=temp/2;/*remainder so take average of two hits*/
        }
    return median;/*return median for this window*/
}
Once the array is ordered the median is collected. For a window with an uneven number of values, the average of the two values located at the median (mid-way) point is taken.

The results gained from the filter with the nzers.pgm image are as follows:-

There is a distinct difference between the two images. The original image has been subjected to speckle noise, and this has been removed in the filtered result. However there has also been a considerable amount of blurring and loss of edge definition. This is a major disadvantage of median filtering using a rectangular neighbourhood (window). An improvement can be made if another shape for the neighbourhood is used. By using a cross shaped neighbourhood, horizontal and vertical line preservation is possible.

This modified neighbourhood pattern produces a result, which contains sharper line definition, and still successfully removes the speckle noise. However there is still significant blurring of the image as a whole.
Weighted Median Filter

The weighted median filter allows the median of a neighbourhood to be taken but with more influence given to certain pixel values. This allows the image to be despeckled, but with a lesser amount of blurring.

The weight value depicts how many times the pixel value is entered into the array, and hence how much it will affect the final median value. For example, a weighting mask with a centre value of 5 and all other weights zero, would result in just the centre pixel value being taken into consideration. The code for this is as below:

```c
/*-----------------------------------------------
*Function:filter_weighted_median
*Inputs:None
*Dependencies:None
*Description: Weighted median filter. Weights are set by \"weight_mask\" Can be
*    any but must sum to less than 90. Results then sorted and median
*    found.
*-----------------------------------------------*/
void filter_weighted_median(void)
{
    int weight_mask[3][3]={ {3, 2, 1},
                           {2, 2, 1},
                           {1, 1, 1}};/*Filter weights*/
    int postn=0;/*set array addressing to zero for initial condition*/
    int wght_mult=0;
    int result_array[90];/*weights must not add to more than 90!*/
    int mask_x, mask_y;
    printf("weighted median filter kernal started \n");
    for (i=1;i < (num_rows -1);i++)
        for (j=1;j < (num_cols -1);j++){
            /*Searches through each pixel*/
            for (mask_y=-1; mask_y<2; mask_y++)
                for (mask_x=-1; mask_x<2; mask_x++)
                { /*+1 to center*/
                    while (wght_mult < weight_mask[mask_y+1][mask_x+1])
                        /*this loop run for n times depending on weight value (n)*/
                        result_array[postn]=image1[(i+mask_x)][(j+mask_y)];
                        /*Weighted pixel values put in array to be sorted*/
                        postn++;
                        wght_mult++;
                }
            wght_mult=0;
            /*now get corresponding pixel value*/
            image2[i][j]=sort(result_array,postn);/*sorting*/
            postn=0; /*re-set array position for next pixel value*/
        }
}

Image 6 was obtained using a centre weight of 5, that is the centre pixel value is weighted five times that of the neighbourhood ones, which are weighted by only one.
Whilst the noise has been removed the image still displays some blurring. Many varying arrangements of the mask weightings are possible, and a few are detailed below:

(a) \begin{array}{ccc}
1 & 1 & 1 \\
1 & 5 & 1 \\
1 & 1 & 1
\end{array}

(b) \begin{array}{ccc}
2 & 1 & 2 \\
1 & 3 & 1 \\
2 & 1 & 2
\end{array}

(c) \begin{array}{ccc}
3 & 2 & 1 \\
2 & 2 & 1 \\
1 & 1 & 1
\end{array}

\textbf{Figure 3 Possible mask weightings}

The results below show masks (b) and (c) used.

\textbf{Image 9 Result using weighting (b)} \hspace{1cm} \textbf{Image 10 Result using weighting (c)}

These results show how a small alteration on the weight mask can have a considerable effect on the final result.

\section*{Adaptive weighted median filter}

Whilst the weighted median filter allows the user to select weightings for the image, these remain constant for the whole image, regardless of the pixel values for each sampled window. The adaptive weighted median filter however changes the weight depending on the image statistics (mean and variance) and the position in the mask, for each set of window values.

These are related by the following equation:

\[ w_{i,j} = \left[ w_{(K+1,K+1)} - \frac{cd\sigma^2}{x} \right] \]

where: \( w_{ij} \) are the new filter weights
\( W_{(K+1,K+1)} \) is the central weighting constant

c is a constant to vary the influence of the equation on the weight values

\( \bar{x} \) is the mean in mask

\( \sigma^2 \) is the variance in mask

d is the distance from the centre of the mask.

The code featuring this equation is shown below. The sort function is used again to
arrange the array and hence calculate the median. Calculations with large values of
the centre weighting carry a significant computational load. The distance values, d,
are stored in an array for simplicity. Once the new weights have been calculated the
process is exactly the same as the weighted median filter.

```c
/*---------------------------------------------
 * Function: filter_adap_weighted_median 
 * Inputs: None 
 * Dependencies: None 
 * Description: An adaptive weighted median filter, for each pixel value calculates
 * window \((3*3)\) values for mean, variance, distance and computes a
 * weight mask depending on results 
 *---------------------------------------------*/
void filter_adap_weighted_median(void)
{

double dist_mask[3][3]= {{1.414, 1, 1.414},
                         {1, 0, 1},
                         {1.414, 1, 1.414}};

int weight_mask[3][3]; /*to be calculated*/
int result_array[1000]; /*weights must not add to more than 90*/
int postn=0; /*set array addressing to zero for initial condition*/
int wgth_sum=0;
float variance, mean, mean_sqrd; /*allow for fractions*/
int w_center=100; /*center weight setting*/
int constant=4; /*normalisation constant*/

/*as usual look at each pixel value at a time*/
for (i=1; i < (num_rows -1); i++)
for (j=1; j < (num_cols -1); j++)
{

/*get values from image*/
sum=(image1[i-1][j-1]+image1[i][j-1]+image1[i+1][j-1]+image1[i-1][j]+image1[i][j]+image1[i+1][j]+image1[i-1][j+1]+image1[i][j+1]+image1[i+1][j+1]);
mean=(sum/9);
mean_sqrd=(mean * mean);
/*calculate the variance-(hard coded)*/
sqrd_sum=((image1[i-1][j-1] * image1[i-1][j-1])+
          (image1[i][j-1] * image1[i][j-1]) 
          + (image1[i+1][j-1] * image1[i+1][j-1])+
          (image1[i-1][j] * image1[i-1][j])
          + (image1[i][j] * image1[i][j])
          + (image1[i+1][j] * image1[i+1][j])
          + (image1[i-1][j+1] * image1[i-1][j+1])
          + (image1[i][j+1] * image1[i][j+1])
          + (image1[i+1][j+1] * image1[i+1][j+1]));
variance=((sqrd_sum - (9*mean_sqrd))/9);

if (debug_test==1){printf("sum=%d sum2d=%d mean=%f mean_sqrd=%f
", sum, sqrd_sum, mean, mean_sqrd, variance);}

/*now calculate the weight mask*/
for (move_i=0; move_i < 3; move_i++)
for (move_j=0; move_j < 3; move_j++)
{

/*the equation*/

    tmp_value=(w_center-((constant * dist_mask[move_i][move_j] *

```
if (tmp_value < 0) {
    weight_mask[move_i][move_j]=0;
} else {
    weight_mask[move_i][move_j]=tmp_value;
}
if (debug_test==2){printf("%d ",weight_mask[move_i][move_j]);}
if (debug_test==2){printf("n");}
if (debug_test==2){printf("Filter weight mask has been created\n");}
/*++++++++++++++++++++++++ now treat as weighted median filter++++++++*/
/*So for each pixel*/
for (mask_y=-1; mask_y<2; mask_y++)
    for (mask_x=-1; mask_x<2; mask_x++)
        {wght_mult = 0;}
    while (wght_mult < weight_mask[mask_y+1][mask_x+1])/*+1 to center*/
        /*this loop for run n times depending on weight value (n)*/
            result_array[postn]=image1[(i+mask_x)][(j+mask_y)];
            /*Weighted pixel values put in array to be sorted*/
            postn++;
            wght_mult++;}
    wght_mult=0;
/*now get corresponding pixel value*/
if (debug_test==1){printf("    -->Length is %d at location[%d][%d]\n", postn, i, j);
    image2[i][j]=sort(result_array,postn);/*sorting*/
    postn=0; /*re-set array position for next pixel value*/
    }
}
The results show the effect on varying the constant, which relates to the degree of weight variance and hence the level of adaptability.
Laplacian of a Gaussian

The Laplacian of a Gaussian operator is a 2-D convolution operator used to blur images and remove detail and noise (similar to a mean filter). The kernel used represents the shape of a Gaussian hump (Figure 4). This technique is used to “clean” an image before edge detection methods (such as the Canny Edge Detector) are attempted.

The idea of Gaussian smoothing is to use this 2-D distribution as a “point-spread” function. This is achieved by convolution. The .pgm image is stored as a collection of discrete pixels. Therefore it is necessary to produce an approximation to the Gaussian function in order to perform the convolution. The distribution is effectively zero at three standard deviations from the mean.

\[ G(x, y) = \frac{1}{2\pi\sigma^2}e^{-\frac{x^2+y^2}{2\sigma^2}} \]

Figure 5 2D Gaussian distribution equation

This can be approximated into the following mask value integers for a standard deviation of 1.0.

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>7</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>26</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>26</td>
<td>41</td>
<td>26</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>26</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>7</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 6 Laplacian of a Gaussian mask for \( \sigma = 1.0 \)

---

2 Figure courtesy of http://www.dai.ed.ac.uk/HIPR2/
The code for this filter is below:

```c
void filter_gaus_log(void)
{
    int constant=21;
    int a, b, x, y;
    int gaus_mask[5][5]={{1, 4, 7, 4,1}, /*mask for set stdv 1.0*/
                        {4,16,26,16,4},
                        {7,26,41,26,7},
                        {4,16,26,16,4},
                        {1, 4, 7, 4,1}};
    /*write special case for 5*5 convolution*/
    for (x=0;x< (num_rows -6);x++)
        for (y=0;y< (num_cols -6);y++) {
            /*for each pixel values*/
            image2[x][y]=0;/*flush out just in case*/
            for (a=0; a<5; a++) {/*for each mask value*/
                for (b=0; b<5; b++) {/*for each mask value*/
                    /*convolution */
                    image2[x][y] += (image1[x+b][y+a] * gaus_mask[b][a]);
                }/*convolution */
            }/*for each pixel values*/
            image2[x][y]=(image2[x][y] / constant);/*to normalise*/
        }
}
```

This mask produced the following blurring effects on the image.

![Image 15 Original](image15.png) ![Image 16 Result of LoG filter (σ = 1.0)](image16.png)

**Edge Detection for heart.pgm image**

Now we shall look at some edge detection techniques for the `heart.pgm` image.

**Prewitt Templates**

The Prewitt operator can split an image into the edges that exist within the image.
Each template gives the magnitude of the edges in each direction. The values then need to be squared and then rooted to obtain the intensity magnitude at each point. The code for the Prewitt template is below. The process requires a convolution in order to obtain the results and this function is also shown.

```c
/*------------------------------
*Function:filter_prewitt
*Inputs:
*Dependencies:None
*Description:Basic Prewitt Filter
*------------------------------*/
void filter_prewitt(void)
{
    /*initialise filter weights for prewitt*/
    int x_mask[3][3]={
        {-1, 0, 1},
        {-1, 0, 1},
        {-1, 0, 1}};

    int y_mask[3][3] = {
        { 1, 1, 1},
        { 0, 0, 0},
        {-1,-1,-1}};

    int threshold_value=180; /*this gives the threshold level*/

    int x_mask_mag, y_mask_mag;
    for (i=2; i< (num_rows -2); i++)
        for (j=2; j< (num_cols -2); j++)  {
            /*for each pixel value*/
            x_mask_mag = convolve(i, j, x_mask);
            y_mask_mag = convolve(i, j, y_mask);
            /*Returns the results of the two masks*/
            if (sqrt((x_mask_mag*x_mask_mag)+(y_mask_mag*y_mask_mag)) > threshold_value)
                {image2[i][j]=0;} /*sets to white if bigger than threshold*/
            else
                {image2[i][j]=255;} /*sets to black if less than threshold*/
        }
}
```

This uses the convolve function:-

```c
/*------------------------------
*Function:convolve
*Inputs:int x, int y, int mask[3][3]
*Dependencies:None
*Description:Convolution Kernal for use with a 3*3 mask
*------------------------------*/
int convolve(int x, int y, int mask[3][3])
{
    int x_mask, y_mask, value=0; /*local variables*/
    for(x_mask=-1; x_mask<=1; x_mask++)
        for(y_mask=-1; y_mask<=1; y_mask++)  {
            /*cumulatively adds the result*/
            value += image1[x+x_mask][y+y_mask] * mask[x_mask+1][y_mask+1];
        }
    return value;
}
```

The results gained by using this filter can be seen below:-
It is clear to see the effect of decreasing the threshold on the images, and also how edges in the image that have been detected result in a black pixel value.

**Sobel Templates**

The Sobel filter is another method of edge detection and very similar to the Prewitt template. This filter utilises another couple of templates, offering both smoothing and dithering. This is one of the most popular filters for edge detection. The templates are as follows:-
The code for the Sobel filter is below:

```c
void filter_sobel(void)
{
    /*initialise filter weights*/
    int x_mask[3][3]=
    { { -1, 0, 1},
      { -2, 0, 2},
      { -1, 0, 1}};

    int y_mask[3][3]=
    { {  1, 2, 1},
      {  0, 0, 0},
      { -1,-2,-1}};

    int threshold_value=180;/*this gives the threshold level*/
    int x_mask_mag, y_mask_mag;/*position index variables*/
    for (i=2;i< (num_rows -2);i++)
        for (j=2;j< (num_cols -2);j++)
        {
            x_mask_mag = convolve(i, j, x_mask);
            y_mask_mag = convolve(i, j, y_mask);
            /*Returns the results of the two masks*/
            if (sqrt((x_mask_mag*x_mask_mag)+(y_mask_mag*y_mask_mag)) > threshold_value)
                {image2[i][j]=0;}//sets to black
            else
                {image2[i][j]=255;}//sets to white
        }
}
```

Using the original image `heart.pgm`, the results below were obtained for varying threshold levels.
Laplacian Operator

The Laplacian operator is another method for detecting edges. This process uses a single mask with 8 neighbourhoods. The sum of the coefficients must equal zero. The mask is ordered as follows:-

\[
\begin{array}{ccc}
-1 & -1 & -1 \\
-1 & -8 & -1 \\
-1 & -1 & -1 \\
\end{array}
\]

Figure 9 Laplacian operator mask for 8 neighbourhoods

The mask values are convolved with those from the image window to produce the resulting modified pixel value. The code is as follows:-

```c
/*------------------------------------------
*Function:filter_laplacian
*Inputs:
*Dependencies:None
*Description:Basic laplacian Filter, 8 neighbourhoods.
*-------------------------------------------*/
void filter_laplacian(void)
{
    /*initialise filter weights for prewitt*/
    int x_mask[3][3]=
    { { -1, -1, -1},
      { -1,  8, -1},
      { -1, -1, -1 }};

    int threshold_value=40;/*this gives the threshold level*/
    int x_mask_mag;

    for (i=2;i< (num_rows -2);i++)
        for (j=2;j< (num_cols -2);j++)  {
            /*for each pixel value*/
            x_mask_mag = convolve(i, j, x_mask);
            /*Returns the results of the mask*/
            if (x_mask_mag > threshold_value)
                {image2[i][j]=0;}/*set to white*/
            else
```
This mask is seldom used in practise due to the 2\textsuperscript{nd} order derivative being very sensitive to noise, and sometimes the filter responds doubly to edges in the image. Also edge direction cannot be obtained from this method. Image 25 shows the result.

![Image 26 Laplacian operator](image)

As can be seen the image edges have been picked out, although the filter is very sensitive to the noise within the image and hence the result is poorly defined.

**Edge Detection for nzjers.pgm**

The same techniques applied to the *heart.pgm* image can also be applied to the *nzjers.pgm* image. The results are shown below. Firstly the image was passed through an adaptive weighted median filter with centre weighting 100 and a weighting constant of 4. This removed the speckle noise. Image 27 then shows the result of a Sobel transform, and Image 28 a Prewitt transform.

![Image 29 Original](image)

![Image 30 Adaptive weighted median (w cent=100, cons=4)](image)
The edges can be seen, although for the “top” side of the image these are defined with greater clarity. Further processing would be required to link the edges together and hence separate the landmass from the sea.

Appendix

The next few pages contain the complete C source code used for the generation of the images in the report (including those used for file handling).