Visual Computing Lab 2

CHAKKA Karthik Subramanyam

CHATTERJEE Jit

TP2 - Local Filtering and Histograms

Exercise 1: Filtering

1.a) Write a c program that reads a PGM image and compute the image smoothed with the following binomial filter:

$$b_{2,2}(m,n) = \frac{1}{16} \begin{matrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{matrix}$$

Try on boat_noise.pgm

1.b) Add the option to smooth n times as well as the option to smooth with a filter of dimension 5x5 (binomial coefficients can be obtained from the pascal triangle).

1.c) Implement a median filter and compare it with the binomial filters.

1.a) Introduction:

```
A Binomial filter is a type of low pass gaussian filter which provides
 an efficient and simple means for suppression of noise (unwanted
 signals) in images where only few filter coefficients (binomial
 coefficients) are required. The binomial filter corresponds to the rows
 of Pascal’s triangle:
```

$$\begin{array}{ccccccccccc} & & & & & 1 & & & & & \\ & & & & 1 & & 1 & & & & \\ & & & 1 & & 2 & & 1 & & & \\ & & 1 & & 3 & & 3 & & 1 & & \\ & 1 & & 4 & & 6 & & 4 & & 1 & \\ 1 & & 5 & & 10 & & 10 & & 5 & & 1 \\ & & & & & \vdots & & & & & \end{array}$$

```
Two-dimensional binomial filters can be generated by using two one-
dimensional binomial filters in a separable fashion, for example:
```

$$B_2 = (1/4)\begin{bmatrix}1 & 2 & 1\end{bmatrix} \otimes (1/4)\begin{bmatrix}1\\2\\1\end{bmatrix} = (1/16)\begin{bmatrix}1 & 2 & 1\\2 & 4 & 2\\1 & 2 & 1\end{bmatrix}$$

```
A Median filter is a type of non-linear filter which removes speckles
 from images. The median filter considers each pixel in the image in
 turn and looks at its nearby pixels to decide whether or not it is
 representative of its neighbors. It replaces the pixel value with the
 median of neighboring pixel values. The median is calculated by first
 sorting all the pixel values from the surrounding neighborhood into
 numerical order and then replacing the pixel being considered with the
 middle pixel value.
```

Goal:

```
To generate Binomial Filter and Median Filter and apply it on a pgm
 image for image smoothing and compare both.
```

Procedure:

```
Binomial filtering or smoothing consists in replacing a pixel value
 (considered as the central pixel value) by a weighted combination of
 the binomial values in a local neighborhood of the pixel. This linear
 operation is the discrete convolution of the image with a mask.
```

```
In the spatial domain:
```

$$f[m,n] = h \circ g = \sum_{k,l} g[m-k, n-l]\, h[k,l]$$

```
where g is the intensity function in the original image, f is the
 intensity function in the filtered image and h the convolution mask.
```

Steps:

1. Generate Pascal Numbers

```
Based on the size of the binomial filter generate the pascal numbers
 for the Binomial coefficients and save them to a 1D memory buffer.
```

2. Generate Binomial Filter

```
Using the Pascal numbers generate a matrix (Binomial Filter).
```

3. Generate the Division Factor

```
Summation of all the values of the filter matrix will give us the
 Division Factor (for 3X3 Binomial Filter its 1/16).
```

4. Reverse Filter Matrix

```
This step involves flipping of the filter matrix along rows followed
 by a flip along its columns.
```

5. Read pgm image

```
We need to read the pgm image and store the pixel values into a 1D
 memory buffer.
```

6. Convolution for each pixel

```
For convolution, we need to iterate over the memory buffer which
 contains the grayscale (pgm) image and take one pixel (considered
 as the center pixel) with its neighbouring pixels then do the
 multiplication with each element of the corresponding binomial
 coefficients of the filter matrix (Reverse Filter Matrix). Then
 summation of the new pixels and finally dividing by the division
 factor (1/16) and thus replacing the value. For the edges in the
 image matrix, the neighbouring pixels which are not present in the
 original matrix are considered as 0.
```

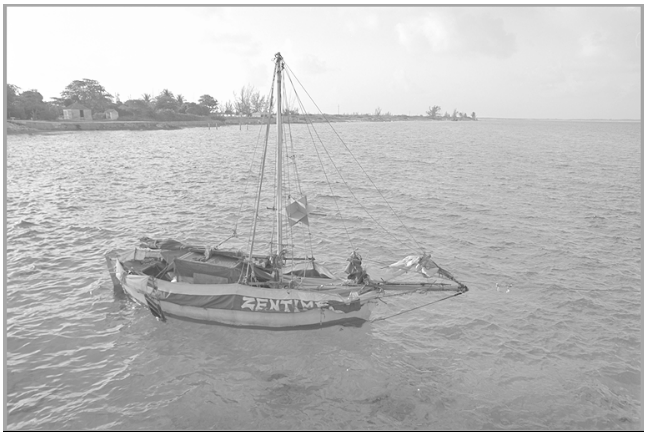
a. Iterate over the image pixels stored in 1D Memory buffer
b. Iterate over the reverse filter matrix store in 1D Memory buffer
c. Multiply each element of step 1 with each element of step 2 y[((i * c) + j)] = (x[(i * c) + j] * h[(i * c) + j])
d. Summation of the values: sum += m[(i * SIZE) + j]
e. Dividing the values by (1/16)
f. Convolution:

```
for (i = 0 to rows) do
    for (j = 0 to columns) do
        Memory_Buffer( (i * col) + j) )=Convolutionforeach pixel
```

Code Snippet:

```
void convol(pix *dstBuff, pix *srcBuff, size_t row, size_t col, pix *h,
 pixprod divFactor){
    pix *revH = allocFilterMem(sizeof (pix));
    revFilter(revH, h);
    for (size_t i = 0; i < row; ++i) {
        for (size_t j = 0; j < col; ++j) {
            *(dstBuff + ( (i * col) + j) ) = convXY(srcBuff, i, j, row,
 col, revH, divFactor);
        }
    }
    free(revH);
    return;
}
```

Input Image:



Input image without noise (Fig 1.1)

Output Image:

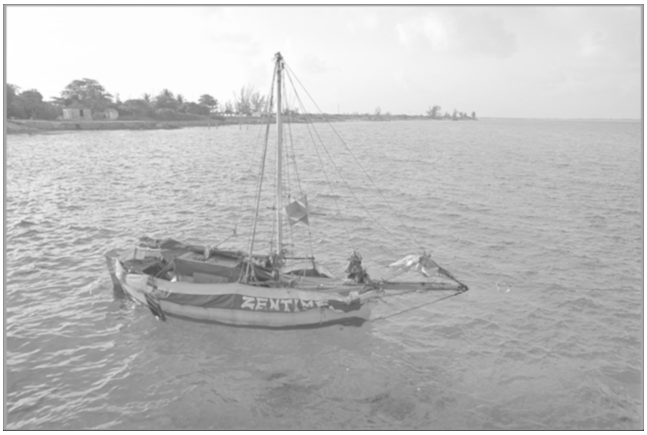
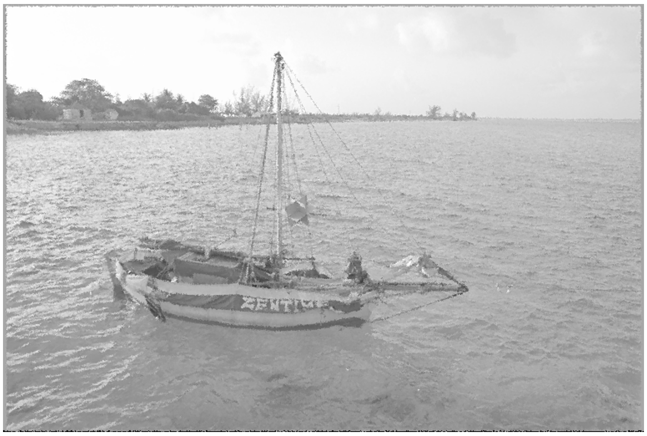


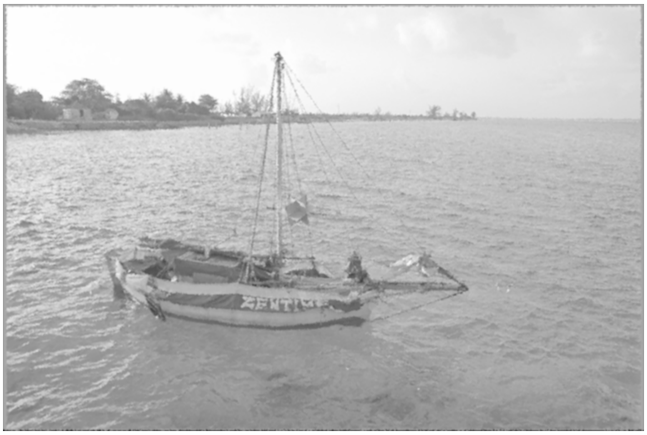
Image with blurring effect after applying 3X3 Binomial Filter (Fig 1.2)

Input Image:



```
Input image with noise (Fig 1.3)
```

Output Image:



```
Image with smoothing effect after applying 3X3 Binomial Filter
(Fig 1.4)
```

1.b) Similarly, to smooth n times using a **Binomial filter** of **size 5X5**, we need to use the convoluted output matrix and pass it again for convolution as input n times.

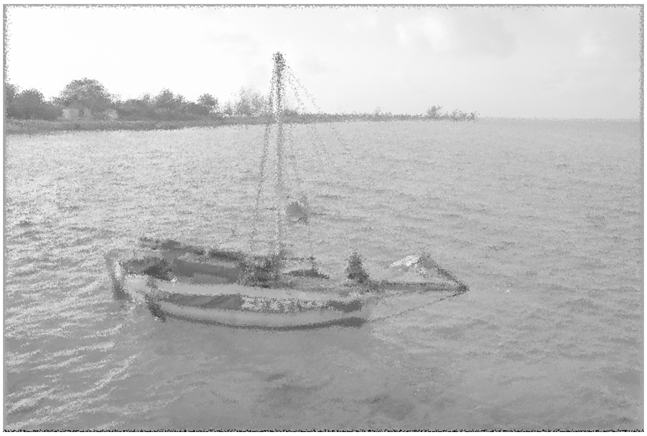
Steps:

1. Generate the 5X5 Binomial Filter using pascal triangle.
2. Apply the filter on the image and the output will be feedback to the input for further convolutions.
3. Convolution for n times:
4. Suppose we convolute 5 times (n=5):
5. Convolution of the image pixels with the 5X5 filter
6. Save the output of step 1 to a temp Memory Buffer
7. Put the output of step 1 as input for the next step and convolute with 5X5 filter and save it to the temp Memory Buffer.
8. Repeat for 5 times (n=5).

Code Snippet:

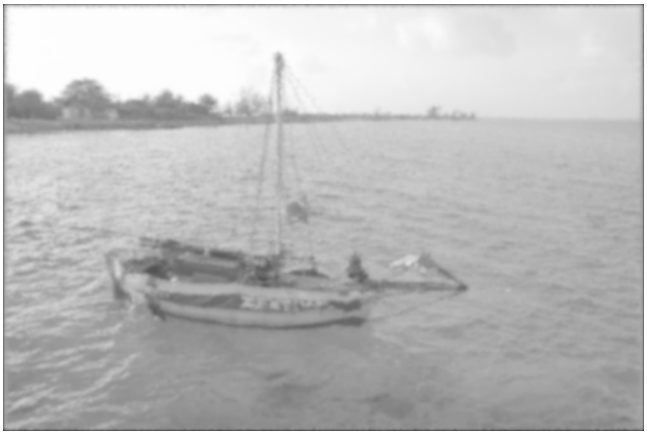
```
void convolN(pix *dstBuff, pix *srcBuff, size_t row, size_t col, pix *h,
 pixprod divFactor, size_t n){
    chkMatrixValidity(srcBuff, row, col);
    NULL_PTR_CHK(dstBuff);
    ABOVE_ZERO_CHK(divFactor);
    ABOVE_ZERO_CHK(n);
    SAME_PTR_CHK(dstBuff, srcBuff);

    pix *tmpBuff = allocMatMem(row, col, sizeof (pix));
    cpyMatrix(tmpBuff, srcBuff, row, col);
    for (size_t i = 0; i < n; ++i) {
        convol(dstBuff, tmpBuff, row, col, h, divFactor);
        cpyMatrix(tmpBuff, dstBuff, row, col);
    }
    free(tmpBuff);
    return;
}
```

Input Image:



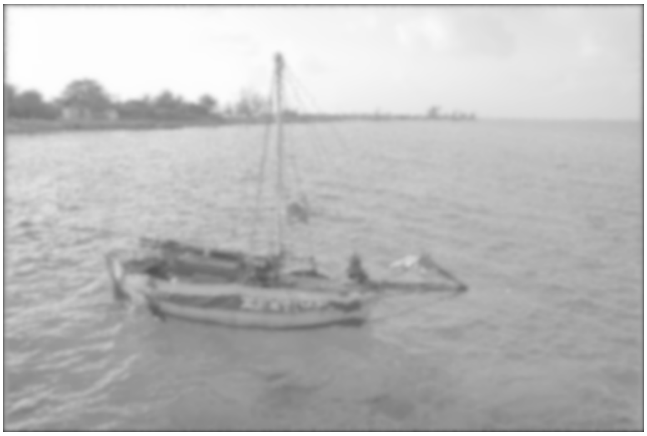
```
Input image with noise (Fig 1.5)
```

Output Image:



```
   Image with smoothing effect after applying 5X5 Binomial Filter 3
times
                              (Fig 1.6)
```

Output Image:



```
   Image with smoothing effect after applying 5X5 Binomial Filter 5
 times
                              (Fig 1.7)
```

```
Fig 1.7 clearly depicts that applying 5X5 Binomial Filter 5 times has
 more smoothening effect than applying 5X5 Binomial Filter 3 times (Fig
 1.6) on the same Input Image (Fig 1.5). Thus, increasing the blurring
 effect.
```

1.c) Median Filter: Median filtering is a nonlinear process useful in reducing impulsive, or salt-and-pepper noise.

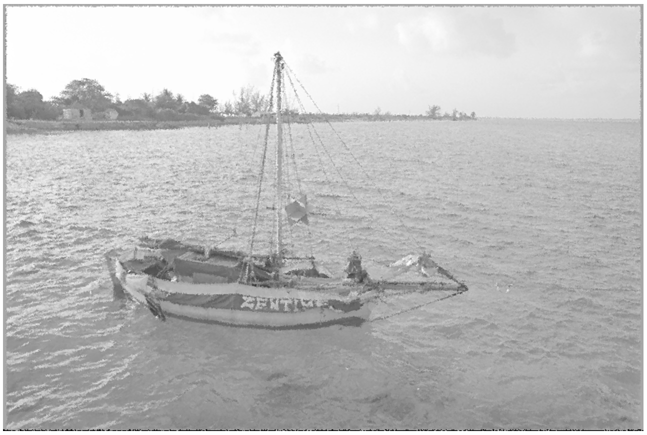
Steps:

1. Store the pixel values of input image in a 1D memory buffer.
2. For each pixel value take all the neighbour pixel value including that cell and for the corner cases consider the neighbour pixels as 0 and store them in a 1D memory buffer.
3. Sort the 1D memory buffer and find the median value.
4. Then the median value is used to store output image pixel intensity.

Code Snippet:

```
void median(pix *dstBuff, pix *srcBuff, size_t row, size_t col){
    for (size_t i = 0; i < row; ++i) {
        for (size_t j = 0; j < col; ++j) {
            *(dstBuff + ( (i * col) + j) ) = medianXY(srcBuff, i, j,
 row, col);
        }
    }
    return;
}
```

Input Image:



```
Input image with less noise (Fig 1.8)
```

Output Image:

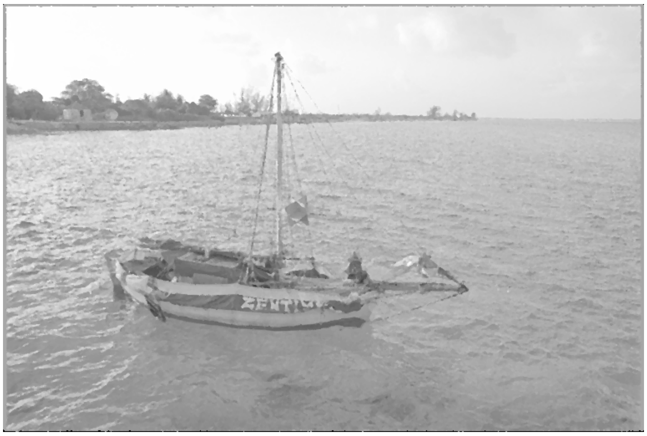
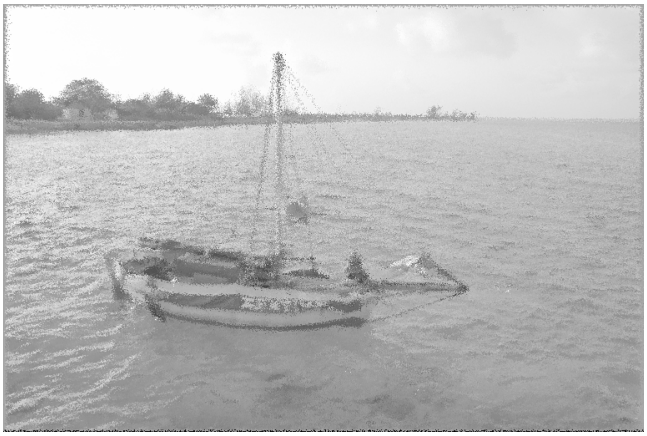


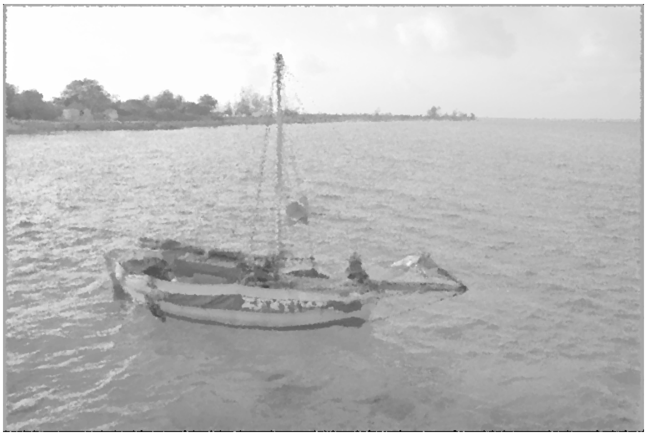
Image with smoothing effect after applying 3X3 Median Filter
(Fig 1.9)

Input Image:



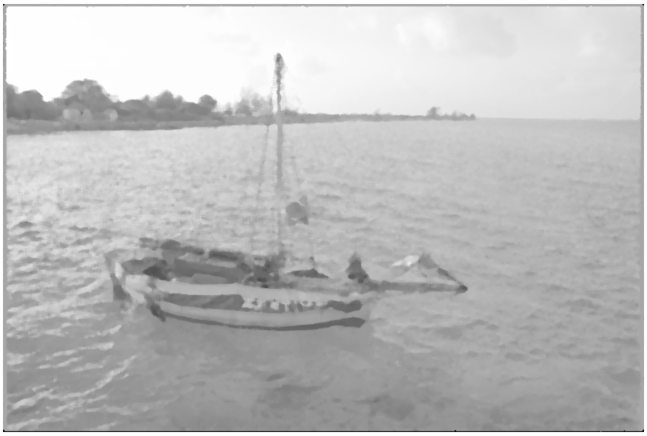
Input image with more noise (Fig 1.10)

Output Image:



```
   Image with smoothing effect after applying 3X3 Median Filter
                            (Fig 1.11)
```

Output Image:



```
   Image with smoothing effect after applying 5X5 Median Filter (Fig
1.12)
```

Results:

```
Binomial filter weigh pixels as a bell-curve (Gaussian) around the
 center pixel. This means that farther pixels get lower weights. As a
 result, near-by pixels have bigger influence on the smoothed rather
 than more distant ones. Median filter is best and simplest when it
 comes to removing outliers and hence noise of salt paper types.
```

Exercise 2 : Histograms

2.a) Add to the previous program a function that computes the intensity histogram. This histogram will be represented by an array of dimension the maximal number of grayscale intensities (i.e. 255).

2.b) Modify the program so that it computes the image transformed with a histogram stretching.

2.c) Modify the program so that it computes the image transformed with a histogram equalization.

Introduction:

Histogram of an image shows frequency of pixels intesity values. So, a histogram for a grayscale image with intensity values in the range

$$I(u,v) \in [0, K-1]$$

would contain exact K entries

For example, 8-bit grayscale image, K = 2^8 = 256. Each histogram entry is defined as: h(i) = number of pixels with intensity / for all 0<i<K. For example, h(125) = number of pixels with intensity equal to 125.

Histogram stretching is a process which involves the application of a transformation function to each pixel of an image in order to redistribute the information of the histogram toward the extremes of a grey level range.

$$g(x,y) = \frac{f(x,y) - fmin}{fmax - fmin} * 2^{bpp}$$

where f(x,y) denotes the value of each pixel intensity, bpp = 8, fmax = max pixel intensity and fmin = min pixel intensity of the image.

Histogram equalization is a technique for adjusting image intensities to enhance contrast. For equalizing the histogram, we need to compute the histogram and then normalize it into a probability distribution. For normalization, we just need to divide the frequency of each pixel intesity value by the total number of pixels present in the image. It makes the resulting histogram flat.

Goal:

```
To implement a histogram intensity function with histogram stretching
 and histogram equalization.
```

Procedure:

2.a) Steps to implement Histogram intensity function:

1. Iterate over the rows of the image matrix.
2. Iterate over the columns of the image matrix.
3. Count the number of intensities for each pixel intensity of the image

Pseudocode:

```
1)Build_Histogram()
2)  for(i=0 to row) do
3)      for(j=0 to col) do
4)          Histogram[Memory Buffer[(i * col) + j]]++
```

Code Snippet:

```
pixprod *buildHist(pix *buf, size_t row, size_t col){
    chkMatrixValidity(buf, row, col);
    pixprod *hist = malloc(sizeof (pixprod) * (MAX_PIX_VAL + 1));
    memset(hist, 0, sizeof (pixprod) * (MAX_PIX_VAL + 1));
    for (size_t i = 0; i < row; ++i) {
        for (size_t j = 0; j < col; ++j) {
            hist[buf[(i * col) + j]]++;
        }
    }
    return hist;
}
```

Image:

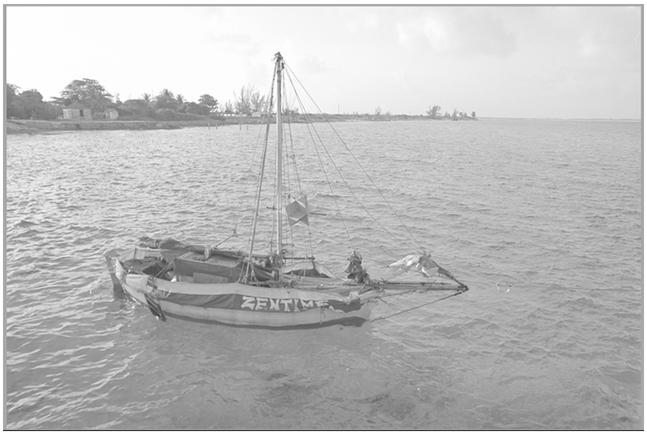
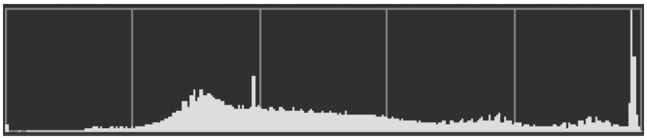


Image of a boat (Fig 2.1)

Histogram:



Histogram of Fig 2.1
(Fig 2.2)

Image:

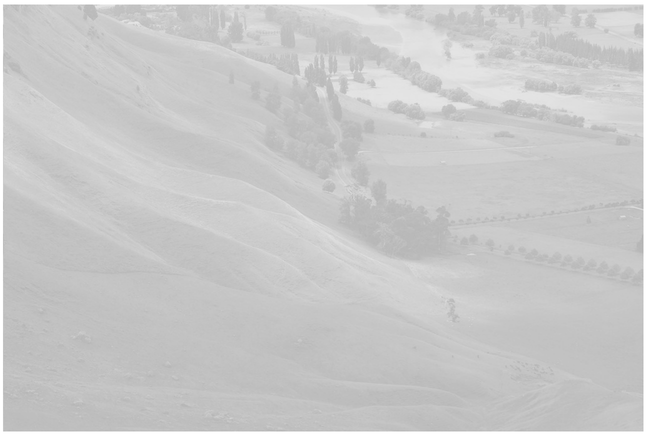
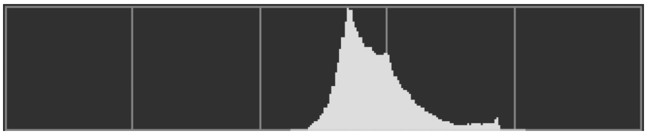


Image of a landscape(Fig 2.3) [Image Source: Wikipedia]

Histogram:



Histogram of Fig 2.3 (Fig 2.4)

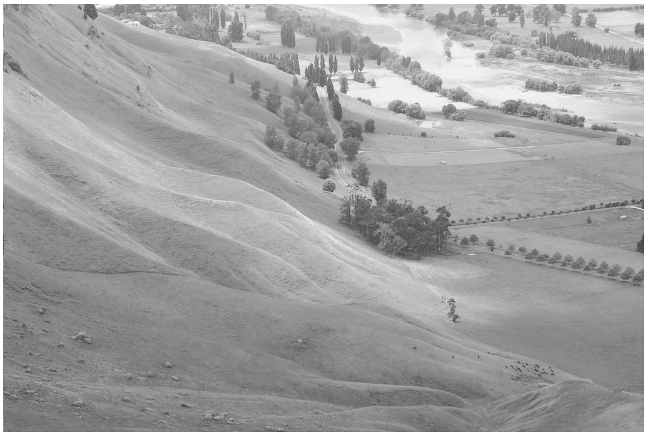
2.b) Steps to implement Histogram Stretching:

```
Input: Current Pixel Intensity = PI, Max Intensity of the Image = IMax,
 Min Intensity of the Image = IMin, Max Pixel Value = 255

1) Build_Histogram_Stretching()
2) DI = IMax-IMin
3) for (i=0 to row) do
4)    for (j=0 to col) do
5)        Val = ((PI-IMin)*255)/DI
6)        Memory Buffer[ (i * col) + j ] = keepInRange(Val);


7) keepInRange()
8)      Checks the value (Val) between 0 to 255 to keep within the range
```

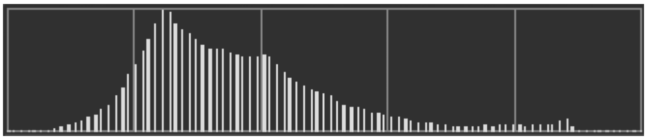
Code Snippet:

```
void stretchHist(pix *dstBuf, pix *srcBuf, size_t row, size_t col){
    chkMatrixValidity(srcBuf, row, col);
    NULL_PTR_CHK(dstBuf);
    SAME_PTR_CHK(dstBuf, srcBuf);

    pixprod *hist = buildHist(srcBuf, row, col);
    pix iMin = findMinIndex(hist, sizeof (pix) * MAX_PIX_VAL);
    pix iMax = findMaxIndex(hist, sizeof (pix) * MAX_PIX_VAL);
    pix iDel = iMax - iMin;
    for (size_t i = 0; i < row; ++i) {
        for (size_t j = 0; j < col; ++j) {
            pix iCur = srcBuf[ (i * col) + j ];
            signed int val = (int)( ( (iCur - iMin) * MAX_PIX_VAL ) /
 (double) iDel );
            dstBuf[ (i * col) + j ] = keepInRange(val);
        }
    }
    free(hist);
}
```

Histogram stretching:



Applying Histogram stretching on Fig 2.3
(Fig 2.5)



Corresponding Histogram of Fig 2.5 after Histogram stretching
(Fig 2.6)

Fig 2.4 and Fig 2.6 shows that after Histogram stretching, the overall shape of the histogram remains same.

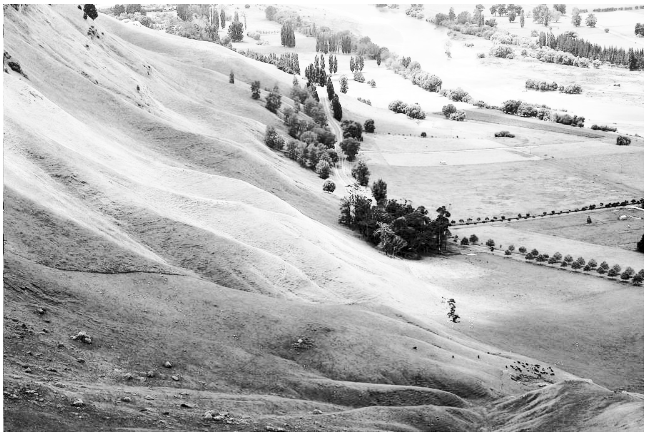
2.c) Steps to implement Histogram Equalization:

```
Input: Max Pixel Value = 255, sum = sum of the pixel intensities of
 the image (Cumulative distribution function), DBf = 1D Memory Buffer
 containing the image matrix.

1) Build_Histogram_Equalization()
2) for (i=0 to row) do
3)     for (j=0 to col) do
4)         DBf + ((i * col) + j)) = (sum * 255) / (row * col))
```

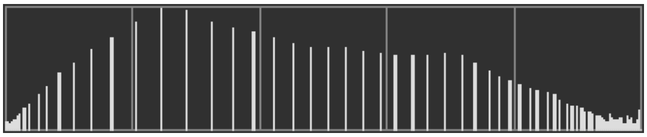
Code Snippet:

```
mfiltersum cdfBuff(pixprod *buff, size_t limit){
    NULL_PTR_CHK(buff);

    mfiltersum sum = 0;
    for (size_t i = 0; i <= limit; ++i) {
        sum += buff[i];
    }
    return sum;
}

pix histEqXY(pixprod *histBuf, pix *imgBuf, size_t x, size_t y, size_t
 row, size_t col ){
    mfiltersum sum = cdfBuff(histBuf, imgBuf[(x * col) + y]);
    mfiltersum eqXY = (mfiltersum) ( ( (double)(sum * MAX_PIX_VAL) ) /
 (row * col) );
    return keepInRange((int)eqXY);
}
```

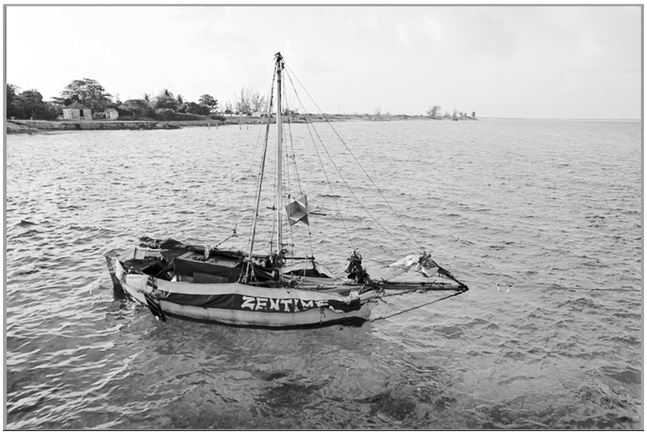
Histogram equalization:



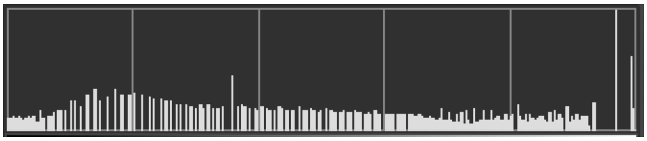
Applying Histogram equalization on Fig 2.3
(Fig 2.7)



Corresponding Histogram of Fig 2.7 after Histogram equalization
(Fig 2.8)



Applying Histogram equalization on Fig 2.1
(Fig 2.9)



Corresponding Histogram of Fig 2.9 after Histogram equalization
(Fig 2.10)

Comparing, Fig 2.4 with Fig 2.8 and Fig 2.2 with Fig 2.10 clearly indicates after Histogram equalization, the shape of the histogram changes.

Results:

```
Histograms help detect image acquisition issues. Histogram stretching
 increases the contrast of the image. Histogram equalization is also
 used to enhance contrast of the image. But, there is one important
 thing to be note here that during histogram equalization the overall
 shape of the histogram changes, where as in histogram stretching the
 overall shape of histogram remains same.
```

Conclusion:

```
    Hence, we can conclude Binomial filters are simple, linear
 and efficient structures based on the binomial coefficients for
 implementing Gaussian filtering where as a Median filter is a non-
linear filter particularly good for removing impulsive noise. On the
 other side, both Histogram stretching and Histogram equalization are
 Image Enhancement techniques. Both depends on transformation functions
 to increase the contrast of the images. The main difference is the
 selection of transformation function.
```