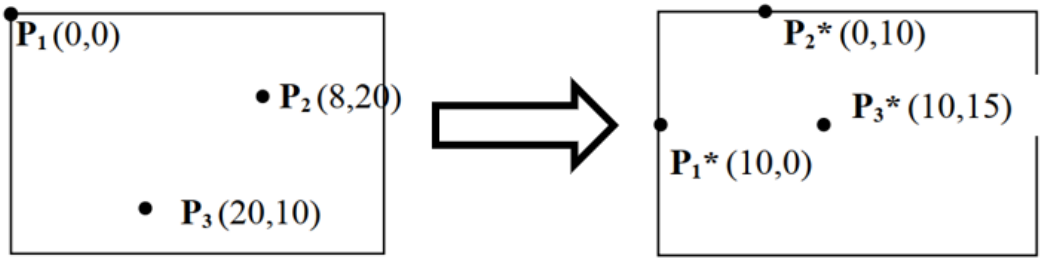
EE431 Intro to Image and Video Processing
Homework 2 - 501504512

PART A

1.



We have transformed 3 points. Therefore, with transformation equations, we can find the parameters of backward and forward affine transformation.

Our main equation:

$$\begin{bmatrix} x' \\ y' \\ w \end{bmatrix} = \begin{bmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

If we solve this matrix for 3 points, we will have 6 unknows and 6 equations.

$$For\ P_1 => \begin{bmatrix} 10 \\ 0 \\ w \end{bmatrix} = \begin{bmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} t_x \\ t_y \\ 1 \end{bmatrix}$$

$$t_x = 10,\ t_y = 0, and\ w = 1$$

$$For\ P_2 => \begin{bmatrix} 0 \\ 10 \\ w \end{bmatrix} = \begin{bmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 8 \\ 20 \\ 1 \end{bmatrix} = \begin{bmatrix} 8a + 20b + 10 \\ 8c + 20d \\ 1 \end{bmatrix}$$

$$For\ P_3 => \begin{bmatrix} 10 \\ 15 \\ w \end{bmatrix} = \begin{bmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 20 \\ 10 \\ 1 \end{bmatrix} = \begin{bmatrix} 20a + 10b + 10 \\ 20c + 10d \\ 1 \end{bmatrix}$$

As a result, to find the parameters, we need to solve this matrix.

$$\left[\begin{array}{cccc|c} 8 & 20 & 0 & 0 & -10 \\ 0 & 0 & 8 & 20 & 10 \\ 20 & 10 & 0 & 0 & 0 \\ 0 & 0 & 20 & 10 & 15 \end{array}\right]$$

Finally, we find our parameters as:

$$a = \frac{5}{16}, b = -\frac{5}{8}, c = -\frac{5}{16}, and\ d = \frac{5}{8}$$

Backward transformation

If we solve this matrix for 3 points, we will have 6 unknows and 6 equations.

$$For\ P_1 => \begin{bmatrix}0\\0\\w\end{bmatrix} = \begin{bmatrix}a & b & t_x\\c & d & t_y\\0 & 0 & 1\end{bmatrix}\begin{bmatrix}10\\0\\1\end{bmatrix} = \begin{bmatrix}10a + t_x\\t_y\\1\end{bmatrix}$$

$$w = 1$$

$$For\ P_2 => \begin{bmatrix}8\\20\\w\end{bmatrix} = \begin{bmatrix}a & b & t_x\\c & d & t_y\\0 & 0 & 1\end{bmatrix}\begin{bmatrix}0\\10\\1\end{bmatrix} = \begin{bmatrix}10b + t_x\\10d + t_y\\1\end{bmatrix}$$

$$For\ P_3 => \begin{bmatrix}20\\10\\w\end{bmatrix} = \begin{bmatrix}a & b & t_x\\c & d & t_y\\0 & 0 & 1\end{bmatrix}\begin{bmatrix}10\\15\\1\end{bmatrix} = \begin{bmatrix}10a + 15b + t_x\\10c + 15d + t_y\\1\end{bmatrix}$$

As a result, to find the parameters, we need to solve the matrix.

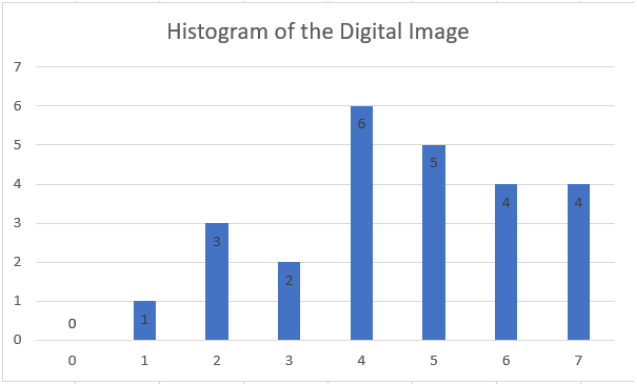
Finally, we find our parameters as:

$$a = \frac{8}{15}, b = \frac{4}{3}, c = -2, d = 2, t_x = -\frac{16}{3}, and\ t_y = 0$$

2.

The digital image

5	7	4	7	5
6	3	6	5	2
4	5	4	7	4
1	2	5	3	6
7	2	4	6	4

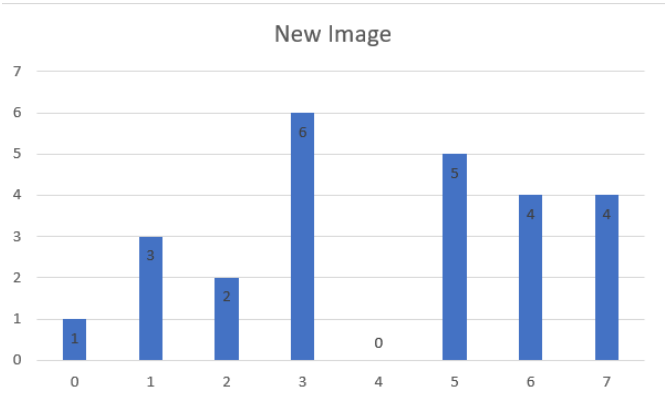


If we analyse and equalize the digital image with assuming its maximum amplitude as 7,

$Value\ r_k$	$Frequency\ n_k$	$Cum.Frequency$	CDF	$Equilized\ Values$
r_0	0	0	0	0 => 0
r_1	1	1	0.04	0.28 => 0
r_2	3	4	0.16	1.12 => 1
r_3	2	6	0.24	1.68 => 2
r_4	6	12	0.48	3.36 => 3
r_5	5	17	0.68	4.76 => 5
r_6	4	21	0.84	6.02 => 6
r_7	4	25	1	7 => 7

New image which is histogram equalized

5	7	3	7	5
6	2	6	5	1
3	5	3	7	3
0	1	5	2	6
7	1	3	6	3



It seems in the histogram that the equilization process pulled the frequent high values to the lower section to converge an equal histogram.
If we try to reequilize the new image, we will see that the result will not change at all beause it has already equilized.

$Value\ r_k$	$Frequency\ n_k$	$Cum.Frequency$	CDF	$Equilized\ Values$
r_0	1	1	0.04	0.28 => 0
r_1	3	4	0.16	1.12 => 1
r_2	2	6	0.24	1.68 => 2
r_3	6	12	0.48	3.36 => 3
r_4	0	12	0.48	3.36 => 3
r_5	5	17	0.68	4.76 => 5
r_6	4	21	0.84	5.88 => 6
r_7	4	25	1	7 => 7

Double histogram equalized values. We did the equalization process twice and we see that it is the same image as histogram equalized once one.

5	7	4	7	5
6	3	6	5	2
4	5	4	7	4
1	2	5	3	6
7	2	4	6	4

⇨

5	7	3	7	5
6	2	6	5	1
3	5	3	7	3
0	1	5	2	6
7	1	3	6	3

⇨

5	7	3	7	5
6	2	6	5	1
3	5	3	7	3
0	1	5	2	6
7	1	3	6	3

3. Edge detection mainly depends on the high change between pixels. But using this method will lead us to observe unexpected edges. For better result another method could be used. This method is Hough transformation. This transformation mainly depends on determining edges by using mathematical equations. For example, for determining a table's edge a simple line equation have been used.

$$y_i = mx_i + c \quad \textbf{(1)}$$

Since the x_i and y_i values are known, this equation can be held on as follows,

$$c = -mx_i + y_i \quad \textbf{(2)}$$

By using different x_i and y_i values, a correct c and m could be found. Every different point will make us to have different lines. Every point that this line contains has vote. We determine the c and m values according to the vote amount. At the intersection of these lines, we will find the correct c and m values, due to the high number of votes. But this method has a handicap, this handicap is that there are no restrictions for c and m, which means that they could have infinitely different values. Instead of using this method for saving memory and other sources, modified version should be used. **(3)**

$$r = x_i \cos(\Theta) + y_i \sin(\Theta)$$

Then if we want to plot the attitude of the r and Θ we will end up with a sinusoidal behaviour. At the intersection of these sinusoidal we will find correct Θ and r values, like in classic transformation. After this part, we can start to detect triangular shapes by using explained method. Modified Hough transformation's handicap is creating lines all along the working space. But we can use this handicap for our sake. A voting like algorithm could be used for extracting triangular shapes. For defining a triangular space with three lines, we need to intersect those lines. These intersection points will be the sharp edges of the triangular shape. Extraction of the triangular shape only could be done by keeping the points that have been placed between sharp edges. Sharp edges will have greater vote than normal parts and unrelated parts won't have votes. By using this method triangular shapes could be detected. Other parts, which has vote but not placed between two sharp edges, will be deleted. So that we will end up with only the triangle.

PART B

In this problem we wanted to create a function for filter operation. For this, we create a function prototype at my_header.h file. Then we generate a c file named median.c which has the prototyped function definition.

```
unsigned char **median_filter(unsigned char **img, int NC, int NR, int size){
```

For median filtering, we need a calculation memory for sorting, and loops for going on pixels. We know we can't apply filter to sides, so we need a padding variable for defining side gap.

```
unsigned char **img_out;
int i, j, k, l, maxval, maxindex, padding;
// create an array for filter memory
int *arr; arr = (int*)malloc(size*size*sizeof(int));
// create an output img
img_out = alloc_img(NC,NR);
// define a padding which prevents segmentation faults
padding = size/2;
// for loops and image coordinates includes padding for dynamic size operations
for(i=padding; i<NR-padding; i++){
    for(j=padding; j<NC-padding; j++){
```

We fill the array with filter applied pixel and their neighbor pixel values. Then we find the median with making zero the max values according to filter size.

```
// fill the array from pixel and its neighbors
for(k=0; k<size; k++){
    for(l=0; l<size; l++){
        arr[size*k+l] = img[i-padding+k][j-padding+l];
    }
}
// to the middle term, zero the max values
for(k=0; k<(size*size)/2+1; k++){
    // find max value
    maxval = -1; maxindex = 0;
    for(l=0; l<size*size; l++){
        if(arr[l] > maxval){
            maxval = arr[l];
            maxindex = l;
        }
    }
    arr[maxindex] = 0;
}
// resulting pixel value
img_out[i][j] = maxval;
```

For using this filter, we generate a median_app.c file which handles the command line arguments, taking image file, and applying median filter given amount of times.

```
if(argc!=4) {printf("\n Usage: median_app size number_of_filters cat5.pgm \n"); exit(-1);
// take command line arguments
size = atoi(argv[1]);
number_of_filters = atoi(argv[2]);
pgm_file = argv[3];
// create img from pgm
img = pgm_file_to_img(pgm_file, &NC, &NR);
// apply filter for wanted number of times
for(i=0; i<number_of_filters; i++){
    img = median_filter(img, NC, NR, size);
}
```

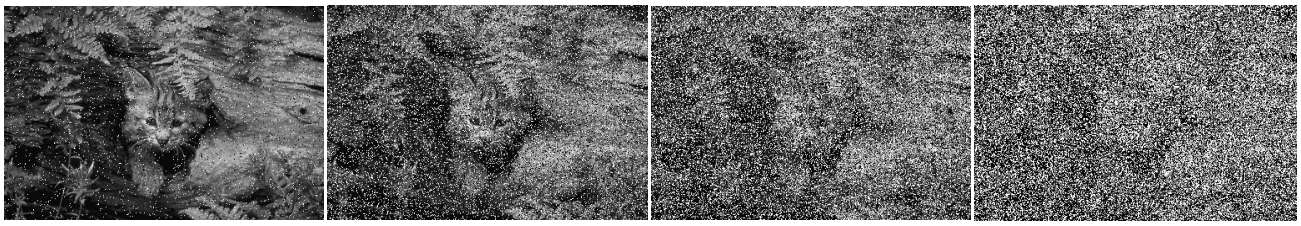
To run our code, we used the following commands to compile and execute;

```
MONSTER@DESKTOP-D50JV7A ~
$ gcc -o medianapp median_app.c median.c img_pro.c

MONSTER@DESKTOP-D50JV7A ~
$ ./medianapp 3 1 cat5.pgm
```

The use of the median filter and comparisons are made on cat images with different salt and pepper noises acquired from: http://www.fit.vutbr.cz/~vasicek/imagedb/

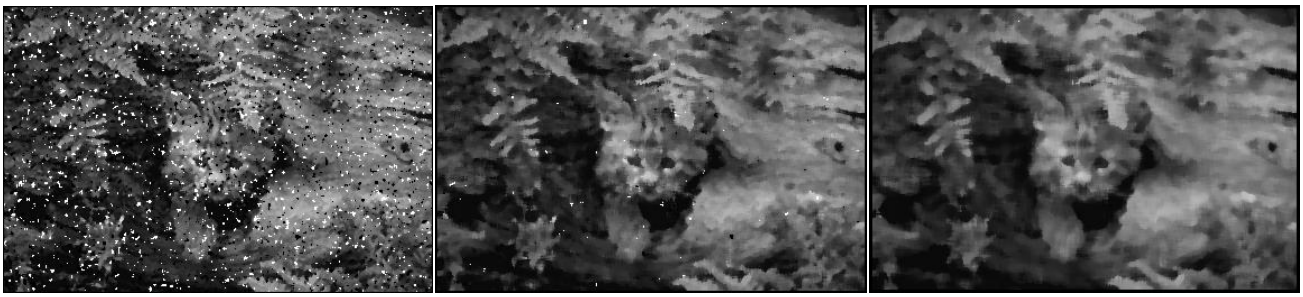
Used images: %5-%25-%45-%65 noise intensity



3x3 filter 1 times to %5-%25-%45 noise intensity

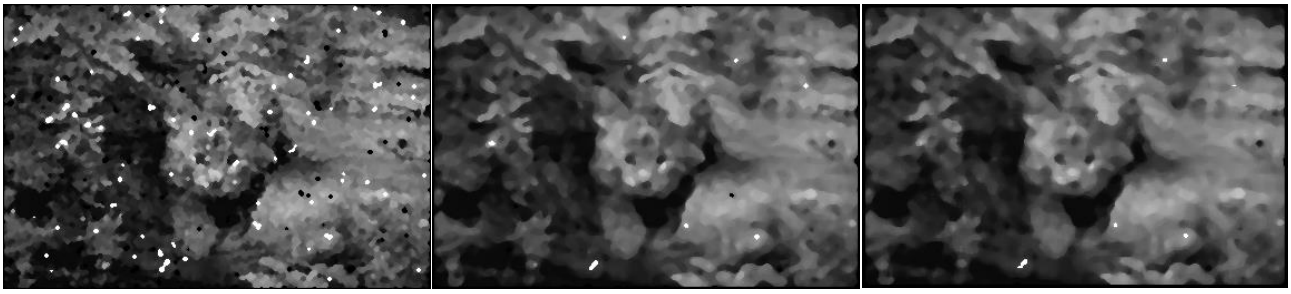
- Same filter gives worse results when increasing noise intensity.

3x3 filter 1 times, 5x5 filter 1 times, 7x7 filter 1 times to %45 noise intensity



- Increasing filter order on same noise intensity gives better results.

3x3 filter 9 times, 5x5 filter 4 times, 7x7 filter 2 times to %65 noise intensity



- Small filters are limited independent of repeating, larger filters blur the image more.