PURDUE UNIVERSITY

ECE 637 Laboratory Exercise 7
Image Restoration

Tong Shen

March 3, 2017

1 MINIMUM MEAN SQUARE ERROR (MMSE) LINEAR FILTERS

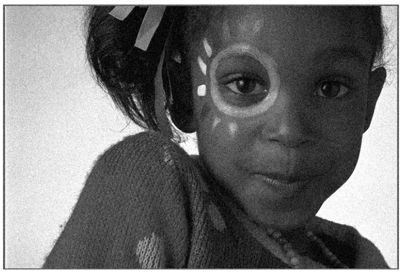
Often filters are designed to minimize the mean squared error between a desired image and the available noisy or distorted image. When the filter is linear, minimum mean squared error (MMSE) filters may be designed using closed form matrix expressions.

For the report, this is the four original images.

(a) *img14g.tif*

(b) *img14bl.tif*

(c) *img14sp.tif*

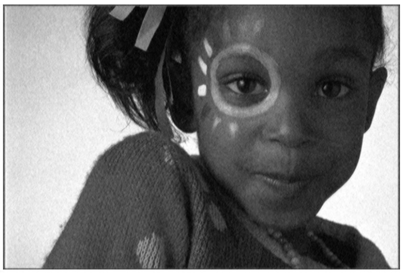


(d) *img14gn.tif*

Figure 1.1: The four original images

And the optimal filtered images are as follows:

(a) optimal filtered blurred images



(b) optimal filtered noisy images

(c) optimal filtered spotted noisy images

Figure 1.2: Optimal filtered images

And for the blurred image, the optimal filtering matrix is:

$$\theta^\star = \begin{bmatrix} 0.3720 & 0.2052 & -0.9682 & 1.0572 & 0.1961 & -1.0020 & 0.9254 \\ -0.0431 & 0.4069 & -1.2219 & -0.0280 & -0.6146 & -1.3229 & 0.4024 \\ -0.3541 & -0.3242 & -0.4810 & 0.3321 & 0.7580 & -0.0871 & -0.7923 \\ 1.1089 & -2.4308 & 1.9317 & 3.7782 & 1.5691 & -0.0701 & 0.0615 \\ 0.3791 & -0.4590 & -1.1045 & 1.2263 & 0.8358 & -1.4710 & 0.3905 \\ -1.0990 & -0.1802 & -0.2944 & 1.0624 & -1.8928 & -1.9628 & 0.8127 \\ 1.1560 & 0.4776 & -1.7439 & 0.6483 & 0.2949 & 0.2604 & 0.3042 \end{bmatrix}$$

For the noisy image, the optimal filtering matrix is:

$$\theta^\star = \begin{bmatrix} 0.0165 & 0.0259 & 0.0044 & 0.0050 & -0.0080 & 0.0302 & -0.0259 \\ -0.0055 & 0.0053 & 0.0355 & 0.0205 & 0.0464 & 0.0091 & 0.0066 \\ -0.0105 & -0.0125 & 0.0674 & 0.0731 & 0.0470 & 0.0290 & -0.0030 \\ -0.0091 & -0.0153 & 0.0476 & 0.2306 & 0.0891 & -0.0175 & 0.0011 \\ -0.0050 & -0.0222 & 0.0423 & 0.1117 & 0.0650 & -0.0118 & 0.0069 \\ -0.0044 & 0.0079 & 0.0307 & 0.0268 & 0.0088 & -0.0063 & 0.0192 \\ -0.0053 & -0.0043 & 0.0154 & 0.0127 & 0.0140 & 0.0183 & 0.0054 \end{bmatrix}$$

For the spotted noisy image, the optimal filtering matrix is:

$$\theta^{\star} = \begin{bmatrix} 0.0080 & 0.0048 & -0.0016 & -0.0050 & 0.0257 & -0.0209 & -0.0185 \\ 0.0017 & -0.0016 & 0.0558 & 0.0267 & 0.0435 & 0.0214 & 0.0196 \\ -0.0010 & 0.0042 & 0.0413 & 0.0968 & 0.0212 & -0.0196 & 0.0199 \\ -0.0014 & -0.0203 & 0.0350 & 0.2652 & 0.1492 & -0.0287 & 0.0083 \\ 0.0252 & 0.0023 & 0.0612 & 0.0965 & 0.0154 & -0.0412 & 0.0233 \\ -0.0099 & -0.0006 & 0.0313 & 0.0497 & 0.0143 & 0.0038 & 0.0131 \\ -0.0407 & 0.0162 & -0.0068 & 0.0100 & 0.0079 & 0.0129 & -0.0110 \end{bmatrix}$$

2 Weighted Median Filtering

A simple median filter is a nonlinear filter which simply replaces each pixel with the median from a set in a window surrounding the pixel. It has the effect of minimizing the absolute prediction error.

The follow two image is the filtered result using the median filter:

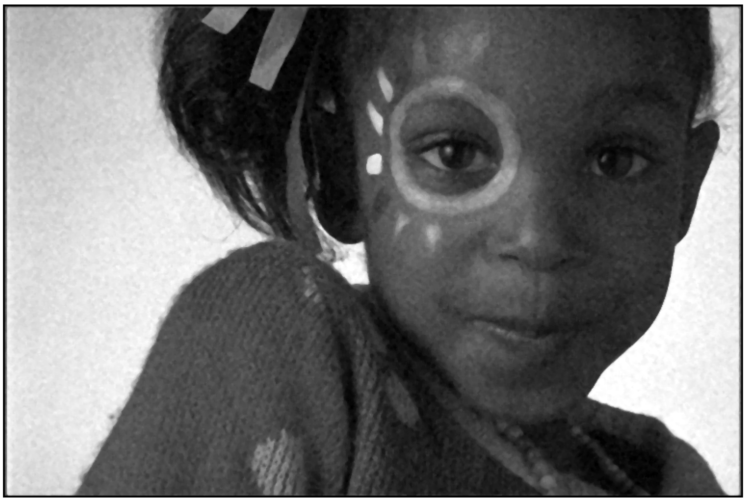


Figure 2.1: The median filtered image of *img14gn.tif*

Figure 2.2: The median filtered image of *img14sp.tif*

2.1 Code Listing

```

#include <math.h>
#include "tiff.h"
#include "allocate.h"
#include "randlib.h"
#include "typeutil.h"

void error(char *name);
int filter(struct TIFF_img input, int x, int y) {
unsigned int image[25];
unsigned int filter[25] = {1,1,1,1,1,1,2,2,2,1,1,2,2,2,1,1,2,2,2,1,1,1,1,1,1 };
unsigned int temp;
unsigned int num = 0;
int location;
for (int i = x - 2; i < x + 3; i++) {
for (int j = y - 2; j < y + 3; j++) {
image[num] = input.mono[i][j];
num++;
}
}
for (int i = 0; i < 24; i++) {
int isSorted = 1;
for (int j = 0; j<24 - i; j++)
{
if (image[j] > image[j + 1])
{
isSorted = 0;
temp = image[j];
```

```
image[j] = image[j + 1];
image[j + 1] = temp;

temp = filter[j];
filter[j] = filter[j + 1];
filter[j + 1] = temp;
}
}
if (isSorted == 1) break;
}
temp = 0;
for (int i = 0; i < 25; i++) {
temp = temp + filter[i];
if (temp >= 17) {
location = i;
break;
}
}
return image[location];
}
int main (int argc, char **argv)
{
FILE *fp;
struct TIFF_img input_img, output_img;
double **img1,**img2;
int i,j;

if ( argc != 2 ) error( argv[0] );

/* open image file */
if ( ( fp = fopen ( argv[1], "rb" ) ) == NULL ) {
fprintf ( stderr, "cannot open file %s\n", argv[1] );
exit ( 1 );
}

/* read image */
if ( read_TIFF ( fp, &input_img ) ) {
fprintf ( stderr, "error reading file %s\n", argv[1] );
exit ( 1 );
}

/* close image file */
fclose ( fp );

/* check the type of image data */
if ( input_img.TIFF_type != 'g' ) {
fprintf ( stderr, "error: image must be grayscale\n" );
exit ( 1 );
}

/* Allocate image of double precision floats */
img1 = (double **)get_img(input_img.width,input_img.height,sizeof(double));
img2 = (double **)get_img(input_img.width,input_img.height,sizeof(double));
```

```
/* copy image to double array */
for ( i = 0; i < input_img.height; i++ )
for ( j = 0; j < input_img.width; j++ ) {
img1[i][j] = input_img.mono[i][j];
}

/* Filter image */
for ( i = 2; i < input_img.height - 2; i++ )
for ( j = 2; j < input_img.width - 2; j++ ) {
img2[i][j] = filter(input_img, i, j);
}

/* set up structure for output image */
get_TIFF(&output_img, input_img.height, input_img.width, 'g');

/* Fill in boundary pixels */
for (i = 0; i < input_img.height; i++)
for (j = 0; j < input_img.width; j++) {
if (i<2||i>input_img.height-3||j<2||j> input_img.width - 3)
{
img2[i][j] = 0;
}
output_img.mono[i][j] = img2[i][j];
}



/* open output image file */
if ( ( fp = fopen ( "output_img.tif", "wb" ) ) == NULL ) {
fprintf ( stderr, "cannot open file green.tif\n");
exit ( 1 );
}

/* write output image */
if ( write_TIFF ( fp, &output_img ) ) {
fprintf ( stderr, "error writing TIFF file %s\n", argv[2] );
exit ( 1 );
}

/* close green image file */
fclose ( fp );

/* de-allocate space which was used for the images */
free_TIFF ( &(input_img) );
free_TIFF ( &(output_img) );

free_img( (void**)img1 );
free_img( (void**)img2 );

return(0);
}

void error(char *name)
{
```

```
exit(1);
}
```