

ECE 637 Lab1 Report

Ruijie Song

Jan. 29. 2021

Section 3 Report:

1.

Handwritten derivation of the analytical expression for H . The derivation starts with the definition of $h(m, n)$ as a 2D rectangular pulse, followed by the double summation for the 2D DTFT. It then separates the variables m and n into two separate geometric series, each of which is simplified using the formula for the sum of a finite geometric series. The final result is the product of two sinc-like functions.

$$\begin{aligned} \textcircled{1} \quad h(m, n) &= \begin{cases} 1/81 & |m| \leq 4 \text{ and } |n| \leq 4 \\ 0 & \text{otherwise} \end{cases} \\ H(e^{ju}, e^{jv}) &= \sum_{n=-\infty}^{\infty} \sum_{m=-\infty}^{\infty} h(m, n) e^{-j(u m + v n)} \\ &= \sum_{n=-4}^4 \sum_{m=-4}^4 \frac{1}{81} e^{-j(u m + v n)} \\ \text{Since } h(m, n) &\text{ is separable,} \\ \therefore H &= \sum_{n=-4}^4 \frac{1}{9} e^{-jv n} \sum_{m=-4}^4 \frac{1}{9} e^{-j u m} = \sum_{n=0}^8 \frac{1}{9} e^{-jv(n-4)} \sum_{m=0}^8 \frac{1}{9} e^{-j u (m-4)} \\ &= \frac{1}{9} e^{jv4} \sum_{n=0}^8 e^{-jv n} \cdot \frac{1}{9} e^{ju4} \sum_{m=0}^8 e^{-j u m} \\ &= \frac{1}{81} e^{jv4} \frac{1 - (e^{-jv})^9}{1 - e^{-jv}} \cdot e^{ju4} \frac{1 - (e^{-ju})^9}{1 - e^{-ju}} \end{aligned}$$

Figure 1. A derivation of the analytical expression for H

2.

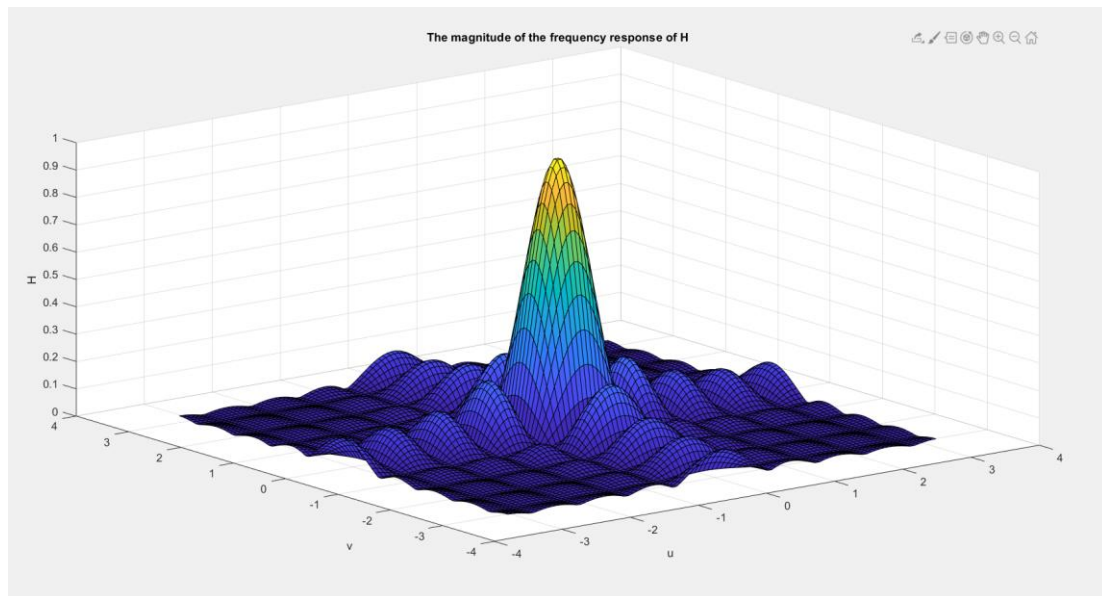


Figure 2. A plot of $|H|$

3.



Figure 3. img03.tif



Figure 4. color.tif



Figure 5. green.tif

4.



Figure 6. The filtered color image

Section 4 Report:

1. & 2.

$$\begin{aligned}
 4 \quad h(m, n) &= \begin{cases} 1/25 & |m| \leq 2, |n| \leq 2 \\ 0 & \text{otherwise} \end{cases} & \sum_{n=0}^{K-1} p^n &= \frac{1-p^K}{1-p} \\
 g(m, n) &= s(m, n) + \lambda (s(m, n) - h(m, n)) \\
 \textcircled{1} \quad H(e^{ju}, e^{jv}) &= \sum_{n=-\infty}^{\infty} \sum_{m=-\infty}^{\infty} h(m, n) e^{-j(um+vn)} \\
 &= \frac{1}{25} \sum_{n=-2}^2 \sum_{m=-2}^2 e^{-j(um+vn)} \\
 &= \frac{1}{5} \sum_{n=-2}^2 e^{-jvn} \cdot \frac{1}{5} \sum_{m=-2}^2 e^{-jum} = \frac{1}{5} \sum_{n=0}^4 e^{-jv(n-2)} \cdot \frac{1}{5} \sum_{m=0}^4 e^{-ju(m-2)} \\
 &= \frac{1}{5} e^{jv2} \sum_{n=0}^4 e^{-jvn} \cdot \frac{1}{5} e^{ju2} \sum_{m=0}^4 e^{-jum} \\
 &= \frac{1}{25} e^{jv2} \frac{1-(e^{-jv})^5}{1-e^{-jv}} \cdot e^{ju2} \frac{1-(e^{-ju})^5}{1-e^{-ju}} \\
 \textcircled{2} \quad \underline{s(x, y)} &= \underline{s(m, n)} = \underline{s(m) s(n)} & s(m) &\longleftrightarrow 1 \\
 &\quad \quad \quad \uparrow \\
 &\quad \quad \quad 1 \\
 \text{DSFT}(s(m, n)) &= \text{DSFT}(s(m) s(n)) = \text{DTFT}(s(m)) \cdot \text{DTFT}(s(n)) \\
 &= 1 \\
 \therefore G &= 1 + \lambda(1 - H)
 \end{aligned}$$

Figure 7. A derivation of the analytical expression for H & G

3.

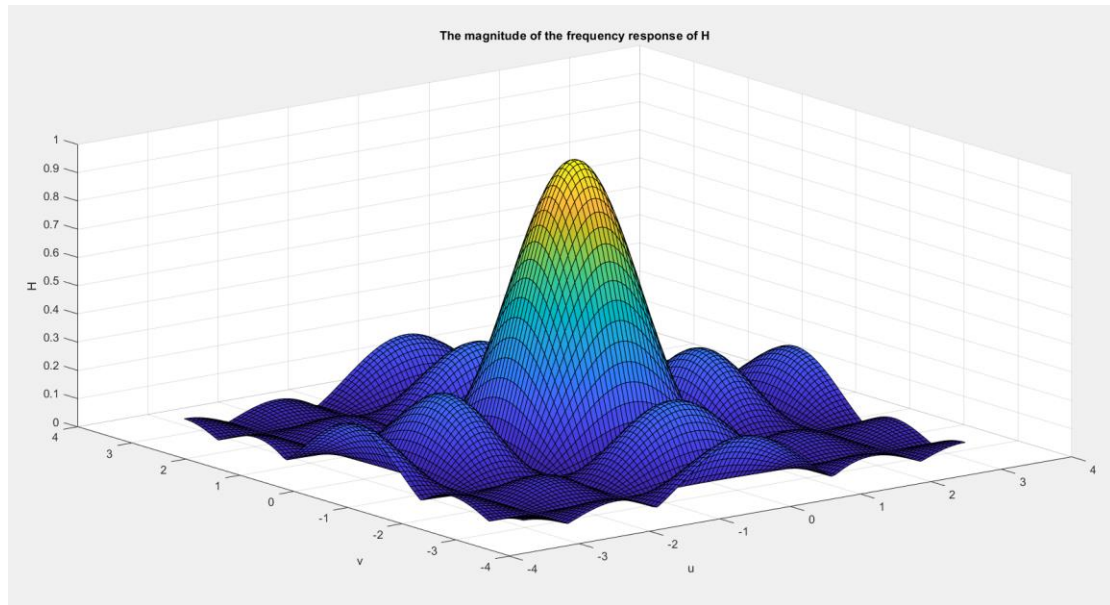


Figure 8. A plot of $|H|$

4.

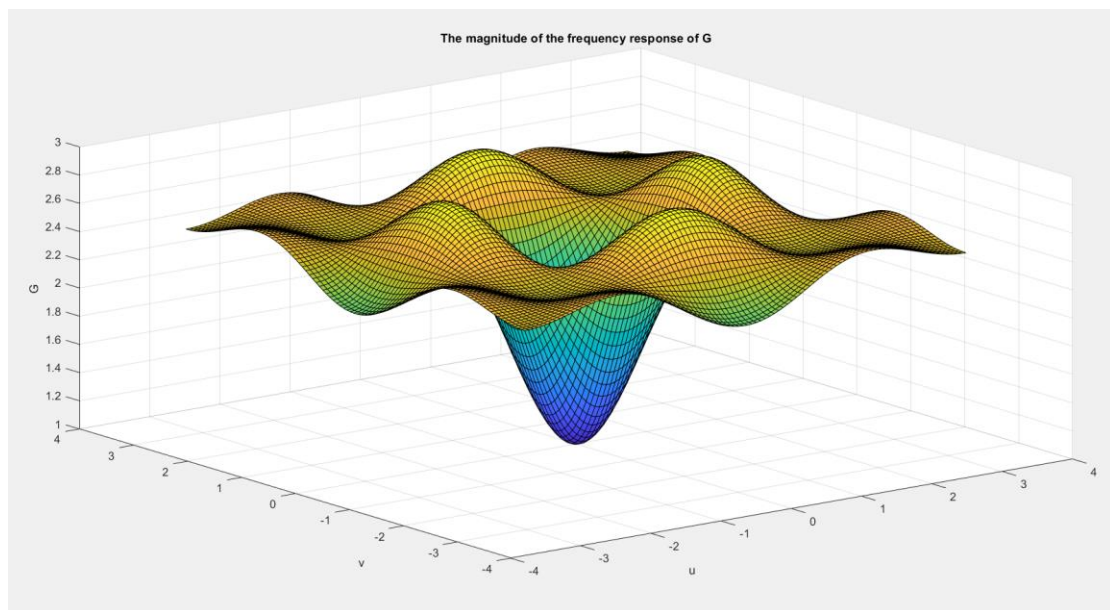


Figure 9. A plot of $|G|$

5.



Figure 10. imgblur.tif.

6.



Figure 11. The output sharpened color image for $\lambda = 1.5$

5. IIR Filter

1.

$$\begin{aligned}
 5. \quad y(m, n) &= 0.01x(m, n) + 0.9(y(m-1, n) + y(m, n-1)) - 0.81y(m-1, n-1) \\
 \cancel{y(m, n)} &\longleftrightarrow \cancel{Y(z_1, z_2)} & \cancel{x(m, n)} &\longleftrightarrow \cancel{X(z_1, z_2)} \\
 y(m, n) &\longleftrightarrow Y(e^{ju}, e^{jv}) & y(m-1, n) &\longleftrightarrow e^{-ju} Y(e^{ju}, e^{jv}) \\
 y(m, n-1) &\longleftrightarrow Y(e^{ju}, e^{jv}) \cdot e^{-jv} & y(m-1, n-1) &\longleftrightarrow e^{-ju} e^{-jv} Y(e^{ju}, e^{jv}) \\
 H = \frac{Y}{X} & & Y &= 0.01X + 0.9(e^{-ju}Y + e^{-jv}Y) - 0.81Y \cdot e^{-ju} e^{-jv} \\
 & & &= 0.01X + (0.9e^{-ju} + 0.9e^{-jv})Y - 0.81e^{-ju} e^{-jv} Y \\
 0.01X &= Y + 0.81e^{-jv} e^{-ju} Y - (0.9e^{-ju} + 0.9e^{-jv})Y \\
 &= Y(1 + 0.81e^{-jv} e^{-ju} - (0.9e^{-ju} + 0.9e^{-jv})) \\
 H = \frac{Y}{X} &= \frac{0.01}{1 + 0.81e^{-jv} e^{-ju} - (0.9e^{-ju} + 0.9e^{-jv})}
 \end{aligned}$$

Figure 12. A derivation of the analytical expression for H

2.

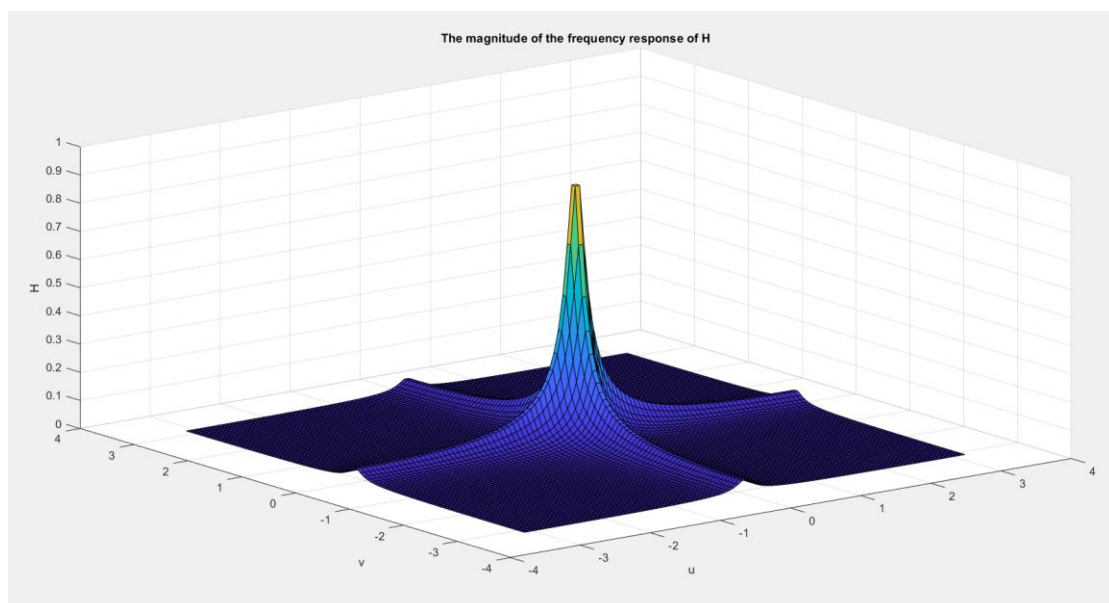


Figure 13. A plot of |H|

3.

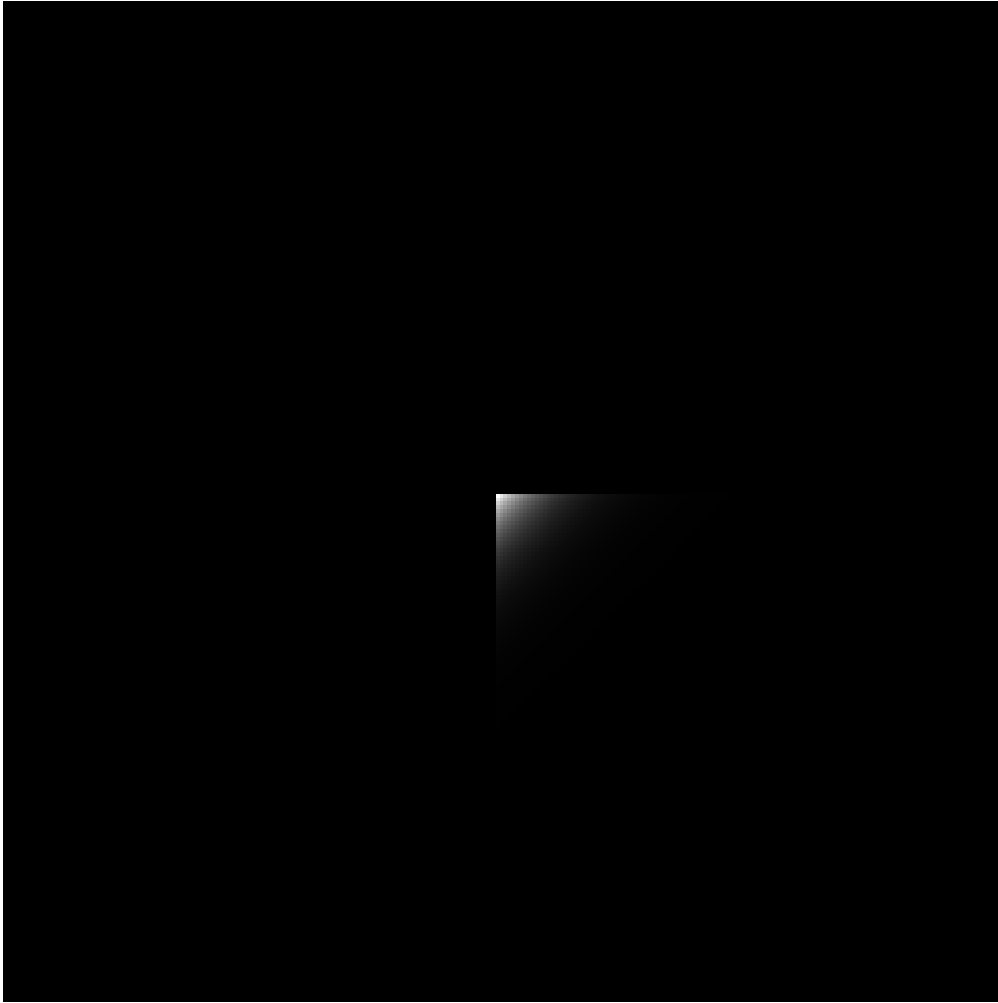


Figure 14. An image of the point spread function

4.



Figure 15. The filtered output color image.

```

1  //-----3 FIR Low Pass Filter-----
2  #include <math.h>
3  #include "tiff.h"
4  #include "allocate.h"
5  #include "randlib.h"
6  #include "typeutil.h"
7
8  void error(char *name);
9
10 int main (int argc, char **argv)
11 {
12     FILE *fp;
13     struct TIFF_img input_img, green_img, red_img, blue_img, color_img;
14     double **img1, **imgr, **imgb, **img2, **img3, **img4;
15     int32_t i, j, pixelg, ii, jj, pixelr, pixelb;
16
17     if ( argc != 2 ) error( argv[0] );
18
19     /* open image file */
20     if ( ( fp = fopen ( argv[1], "rb" ) ) == NULL ) {
21         fprintf ( stderr, "cannot open file %s\n", argv[1] );
22         exit ( 1 );
23     }
24
25     /* read image */
26     if ( read_TIFF ( fp, &input_img ) ) {
27         fprintf ( stderr, "error reading file %s\n", argv[1] );
28         exit ( 1 );
29     }
30
31     /* close image file */
32     fclose ( fp );
33
34     /* check the type of image data */
35     if ( input_img.TIFF_type != 'c' ) {
36         fprintf ( stderr, "error: image must be 24-bit color\n" );
37         exit ( 1 );
38     }
39
40     /* Allocate image of double precision floats */
41     img1 = (double **)get_img(input_img.width, input_img.height, sizeof(double));
42     imgr = (double **)get_img(input_img.width, input_img.height, sizeof(double));
43     imgb = (double **)get_img(input_img.width, input_img.height, sizeof(double));
44     img2 = (double **)get_img(input_img.width, input_img.height, sizeof(double));
45     img3 = (double **)get_img(input_img.width, input_img.height, sizeof(double));
46     img4 = (double **)get_img(input_img.width, input_img.height, sizeof(double));
47
48     /* Initialize the img arrays */
49     for ( i = 0; i < input_img.height; i++ )
50     for ( j = 0; j < input_img.width; j++ ) {
51         img1[i][j] = 0;
52         img2[i][j] = 0;
53     }
54
55     /* copy green, red & blue component to double array */
56     for ( i = 0; i < input_img.height; i++ )
57     for ( j = 0; j < input_img.width; j++ ) {
58         img1[i][j] = input_img.color[1][i][j];
59         imgr[i][j] = input_img.color[0][i][j];
60

```



```
61     imgb[i][j] = input_img.color[2][i][j];
62 }
63
64
65 /* Filter image with the FIR Low Pass Filter */
66 for ( i = 4; i < input_img.height-4; i++ )
67 for ( j = 4; j < input_img.width-4; j++ ) {
68     // img2[i][j] = (img1[i][j-1] + img1[i][j] + img1[i][j+1])/3.0;
69     for ( ii = -4; ii <= 4; ii++ )
70     for ( jj = -4; jj <= 4; jj++ ) {
71         img2[i][j] = img2[i][j] + img1[i+ii][j+jj]/81;
72         img3[i][j] = img3[i][j] + imgr[i+ii][j+jj]/81;
73         img4[i][j] = img4[i][j] + imgb[i+ii][j+jj]/81;
74     }
75 }
76
77 /* Fill in boundary pixels */
78
79 // for ( i = 0; i < input_img.height; i++ ) {
80 //     img2[i][0] = 0;
81 //     img2[i][input_img.width-1] = 0;
82 // }
83
84 for ( i = 0; i < 4; i++ )
85 for ( j = 0; j < 4; j++ ) {
86     for ( ii = -1*i; ii <= 4; ii++ )
87     for ( jj = -1*i; jj <= 4; jj++ ) {
88         img2[i][j] = img2[i][j] + img1[i+ii][j+jj]/81;
89         img3[i][j] = img3[i][j] + imgr[i+ii][j+jj]/81;
90         img4[i][j] = img4[i][j] + imgb[i+ii][j+jj]/81;
91     }
92 }
93
94 for ( i = input_img.height-4; i < input_img.height; i++ )
95 for ( j = input_img.width-4; j < input_img.width; j++ ) {
96     for ( ii = -4; ii < input_img.height-i; ii++ )
97     for ( jj = -4; jj < input_img.width-j; jj++ ) {
98         img2[i][j] = img2[i][j] + img1[i+ii][j+jj]/81;
99         img3[i][j] = img3[i][j] + imgr[i+ii][j+jj]/81;
100        img4[i][j] = img4[i][j] + imgb[i+ii][j+jj]/81;
101    }
102 }
103
104 for ( i = 0; i < 4; i++ )
105 for ( j = input_img.width-4; j < input_img.width; j++ ) {
106     for ( ii = -1*i; ii <= 4; ii++ )
107     for ( jj = -4; jj < input_img.width-j; jj++ ) {
108         img2[i][j] = img2[i][j] + img1[i+ii][j+jj]/81;
109         img3[i][j] = img3[i][j] + imgr[i+ii][j+jj]/81;
110         img4[i][j] = img4[i][j] + imgb[i+ii][j+jj]/81;
111     }
112 }
113
114 for ( i = input_img.height-4; i < input_img.height; i++ )
115 for ( j = 0; j < 4; j++ ) {
116     for ( ii = -4; ii < input_img.height-i; ii++ )
117     for ( jj = -1*i; jj <= 4; jj++ ) {
118         img2[i][j] = img2[i][j] + img1[i+ii][j+jj]/81;
119         img3[i][j] = img3[i][j] + imgr[i+ii][j+jj]/81;
120         img4[i][j] = img4[i][j] + imgb[i+ii][j+jj]/81;
```

```
121     }
122 }
123
124
125 // /* Set seed for random noise generator */
126 // srand2(1);
127
128 // /* Add noise to image */
129 // for ( i = 0; i < input_img.height; i++ )
130 // for ( j = 1; j < input_img.width-1; j++ ) {
131 //     img2[i][j] += 32*normal();
132 // }
133
134 /* set up structure for output achromatic image */
135 /* to allocate a full color image use type 'c' */
136 get_TIFF ( &green_img, input_img.height, input_img.width, 'g' );
137 get_TIFF ( &red_img, input_img.height, input_img.width, 'g' );
138 get_TIFF ( &blue_img, input_img.height, input_img.width, 'g' );
139
140 /* set up structure for output color image */
141 /* Note that the type is 'c' rather than 'g' */
142 get_TIFF ( &color_img, input_img.height, input_img.width, 'c' );
143
144 /* copy green, red & blue component to new images */
145 for ( i = 0; i < input_img.height; i++ )
146 for ( j = 0; j < input_img.width; j++ ) {
147     pixelg = (int32_t)img2[i][j];
148     pixelr = (int32_t)img3[i][j];
149     pixelb = (int32_t)img4[i][j];
150
151     if(pixelg>255) {
152         green_img.mono[i][j] = 255;
153     }
154     else {
155         if(pixelg<0) green_img.mono[i][j] = 0;
156         else green_img.mono[i][j] = pixelg;
157     }
158
159     if(pixelr>255) {
160         red_img.mono[i][j] = 255;
161     }
162     else {
163         if(pixelr<0) red_img.mono[i][j] = 0;
164         else red_img.mono[i][j] = pixelr;
165     }
166
167     if(pixelb>255) {
168         blue_img.mono[i][j] = 255;
169     }
170     else {
171         if(pixelb<0) blue_img.mono[i][j] = 0;
172         else blue_img.mono[i][j] = pixelb;
173     }
174 }
175
176 // /* Illustration: constructing a sample color image -- interchanging the red and
177 // green components from the input color image */
178 // for ( i = 0; i < input_img.height; i++ )
179 //     for ( j = 0; j < input_img.width; j++ ) {
180 //         color_img.color[0][i][j] = input_img.color[1][i][j];
```

```
180 //          color_img.color[1][i][j] = input_img.color[0][i][j];
181 //          color_img.color[2][i][j] = input_img.color[2][i][j];
182 //      }
183
184 /* Illustration: constructing a sample color image -- put 3 image (green,red,blue)
into 1 image */
185 for ( i = 0; i < input_img.height; i++ )
186     for ( j = 0; j < input_img.width; j++ ) {
187         color_img.color[0][i][j] = red_img.mono[i][j];
188         color_img.color[1][i][j] = green_img.mono[i][j];
189         color_img.color[2][i][j] = blue_img.mono[i][j];
190     }
191
192 // /* open green image file */
193 // if ( ( fp = fopen ( "green.tif", "wb" ) ) == NULL ) {
194 //     fprintf ( stderr, "cannot open file green.tif\n");
195 //     exit ( 1 );
196 // }
197
198 // /* write green image */
199 // if ( write_TIFF ( fp, &green_img ) ) {
200 //     fprintf ( stderr, "error writing TIFF file %s\n", argv[2] );
201 //     exit ( 1 );
202 // }
203
204 // /* close green image file */
205 // fclose ( fp );
206
207
208 /* open color image file */
209 if ( ( fp = fopen ( "color.tif", "wb" ) ) == NULL ) {
210     fprintf ( stderr, "cannot open file color.tif\n");
211     exit ( 1 );
212 }
213
214 /* write color image */
215 if ( write_TIFF ( fp, &color_img ) ) {
216     fprintf ( stderr, "error writing TIFF file %s\n", argv[2] );
217     exit ( 1 );
218 }
219
220 /* close color image file */
221 fclose ( fp );
222
223 /* de-allocate space which was used for the images */
224 free_TIFF ( &(input_img) );
225 free_TIFF ( &(green_img) );
226 free_TIFF ( &(red_img) );
227 free_TIFF ( &(blue_img) );
228 free_TIFF ( &(color_img) );
229
230 free_img( (void**)img1 );
231 free_img( (void**)img2 );
232 free_img( (void**)img3 );
233 free_img( (void**)img4 );
234 free_img( (void**)imgr );
235 free_img( (void**)imgb );
236
237 return(0);
238 }
```

```
239
240 void error(char *name)
241 {
242     printf("usage: %s image.tiff \n\n",name);
243     printf("this program reads in a 24-bit color TIFF image.\n");
244     printf("It then horizontally filters the green component, adds noise,\n");
245     printf("and writes out the result as an 8-bit image\n");
246     printf("with the name 'green.tiff'.\n");
247     printf("It also generates an 8-bit color image,\n");
248     printf("that swaps red and green components from the input image");
249     exit(1);
250 }
251
252
```



```

1  //-----4 FIR Sharpening Filter-----
2  #include <math.h>
3  #include "tiff.h"
4  #include "allocate.h"
5  #include "randlib.h"
6  #include "typeutil.h"
7
8  void error(char *name);
9
10 int main (int argc, char **argv)
11 {
12     FILE *fp;
13     struct TIFF_img input_img, green_img, red_img, blue_img, color_img;
14     double **img1, **imgr, **imgb, **img2, **img3, **img4, rho;
15     int32_t i, j, pixelg, ii, jj, pixelr, pixelb;
16
17     /* accepts a command line argument specifying the value of rho */
18     scanf("%lf", &rho);
19
20     if ( argc != 2 ) error( argv[0] );
21
22     /* open image file */
23     if ( ( fp = fopen ( argv[1], "rb" ) ) == NULL ) {
24         fprintf ( stderr, "cannot open file %s\n", argv[1] );
25         exit ( 1 );
26     }
27
28     /* read image */
29     if ( read_TIFF ( fp, &input_img ) ) {
30         fprintf ( stderr, "error reading file %s\n", argv[1] );
31         exit ( 1 );
32     }
33
34     /* close image file */
35     fclose ( fp );
36
37     /* check the type of image data */
38     if ( input_img.TIFF_type != 'c' ) {
39         fprintf ( stderr, "error: image must be 24-bit color\n" );
40         exit ( 1 );
41     }
42
43     /* Allocate image of double precision floats */
44     img1 = (double **)get_img(input_img.width, input_img.height, sizeof(double));
45     imgr = (double **)get_img(input_img.width, input_img.height, sizeof(double));
46     imgb = (double **)get_img(input_img.width, input_img.height, sizeof(double));
47     img2 = (double **)get_img(input_img.width, input_img.height, sizeof(double));
48     img3 = (double **)get_img(input_img.width, input_img.height, sizeof(double));
49     img4 = (double **)get_img(input_img.width, input_img.height, sizeof(double));
50
51     /* Initialize the img arrays */
52     for ( i = 0; i < input_img.height; i++ )
53         for ( j = 0; j < input_img.width; j++ ) {
54             img1[i][j] = 0;
55             img2[i][j] = 0;
56         }
57
58
59     /* copy green, red & blue component to double array */
60     for ( i = 0; i < input_img.height; i++ )

```

```

61 for ( j = 0; j < input_img.width; j++ ) {
62     img1[i][j] = input_img.color[1][i][j];
63     imgr[i][j] = input_img.color[0][i][j];
64     imgb[i][j] = input_img.color[2][i][j];
65 }
66
67
68 /* Filter image with the F FIR Sharpening Filter */
69 for ( i = 2; i < input_img.height-2; i++ )
70 for ( j = 2; j < input_img.width-2; j++ ) {
71     // img2[i][j] = (img1[i][j-1] + img1[i][j] + img1[i][j+1])/3.0;
72     for ( ii = -2; ii <= 2; ii++ )
73     for ( jj = -2; jj <= 2; jj++ ) {
74         if (ii == 0 && jj == 0) {
75             img2[i][j] = img2[i][j] + (1+rho)*img1[i+ii][j+jj];
76             img3[i][j] = img3[i][j] + (1+rho)*imgr[i+ii][j+jj];
77             img4[i][j] = img4[i][j] + (1+rho)*imgb[i+ii][j+jj];
78         }
79         img2[i][j] = img2[i][j] - (rho*1/25)*img1[i+ii][j+jj];
80         img3[i][j] = img3[i][j] - (rho*1/25)*imgr[i+ii][j+jj];
81         img4[i][j] = img4[i][j] - (rho*1/25)*imgb[i+ii][j+jj];
82     }
83 }
84
85 /* Fill in boundary pixels */
86
87 // for ( i = 0; i < input_img.height; i++ ) {
88 //     img2[i][0] = 0;
89 //     img2[i][input_img.width-1] = 0;
90 // }
91
92 for ( i = 0; i < 2; i++ )
93 for ( j = 0; j < 2; j++ ) {
94     for ( ii = -1*i; ii <= 2; ii++ )
95     for ( jj = -1*i; jj <= 2; jj++ ) {
96         if (ii == 0 && jj == 0) {
97             img2[i][j] = img2[i][j] + (1+rho)*img1[i+ii][j+jj];
98             img3[i][j] = img3[i][j] + (1+rho)*imgr[i+ii][j+jj];
99             img4[i][j] = img4[i][j] + (1+rho)*imgb[i+ii][j+jj];
100         }
101         img2[i][j] = img2[i][j] - (rho*1/25)*img1[i+ii][j+jj];
102         img3[i][j] = img3[i][j] - (rho*1/25)*imgr[i+ii][j+jj];
103         img4[i][j] = img4[i][j] - (rho*1/25)*imgb[i+ii][j+jj];
104     }
105 }
106
107 for ( i = input_img.height-2; i < input_img.height; i++ )
108 for ( j = input_img.width-2; j < input_img.width; j++ ) {
109     for ( ii = -2; ii < input_img.height-i; ii++ )
110     for ( jj = -2; jj < input_img.width-j; jj++ ) {
111         if (ii == 0 && jj == 0) {
112             img2[i][j] = img2[i][j] + (1+rho)*img1[i+ii][j+jj];
113             img3[i][j] = img3[i][j] + (1+rho)*imgr[i+ii][j+jj];
114             img4[i][j] = img4[i][j] + (1+rho)*imgb[i+ii][j+jj];
115         }
116         img2[i][j] = img2[i][j] - (rho*1/25)*img1[i+ii][j+jj];
117         img3[i][j] = img3[i][j] - (rho*1/25)*imgr[i+ii][j+jj];
118         img4[i][j] = img4[i][j] - (rho*1/25)*imgb[i+ii][j+jj];
119     }
120 }

```

```

121
122 for ( i = 0; i < 2; i++ )
123 for ( j = input_img.width-2; j < input_img.width; j++ ) {
124     for ( ii = -1*i; ii <= 2; ii++ )
125     for ( jj = -2; jj < input_img.width-j; jj++ ) {
126         if (ii == 0 && jj == 0) {
127             img2[i][j] = img2[i][j] + (1+rho)*img1[i+ii][j+jj];
128             img3[i][j] = img3[i][j] + (1+rho)*imgr[i+ii][j+jj];
129             img4[i][j] = img4[i][j] + (1+rho)*imgb[i+ii][j+jj];
130         }
131         img2[i][j] = img2[i][j] - (rho*1/25)*img1[i+ii][j+jj];
132         img3[i][j] = img3[i][j] - (rho*1/25)*imgr[i+ii][j+jj];
133         img4[i][j] = img4[i][j] - (rho*1/25)*imgb[i+ii][j+jj];
134     }
135 }
136
137 for ( i = input_img.height-2; i < input_img.height; i++ )
138 for ( j = 0; j < 2; j++ ) {
139     for ( ii = -2; ii < input_img.height-i; ii++ )
140     for ( jj = -1*i; jj <= 2; jj++ ) {
141         if (ii == 0 && jj == 0) {
142             img2[i][j] = img2[i][j] + (1+rho)*img1[i+ii][j+jj];
143             img3[i][j] = img3[i][j] + (1+rho)*imgr[i+ii][j+jj];
144             img4[i][j] = img4[i][j] + (1+rho)*imgb[i+ii][j+jj];
145         }
146         img2[i][j] = img2[i][j] - (rho*1/25)*img1[i+ii][j+jj];
147         img3[i][j] = img3[i][j] - (rho*1/25)*imgr[i+ii][j+jj];
148         img4[i][j] = img4[i][j] - (rho*1/25)*imgb[i+ii][j+jj];
149     }
150 }
151
152
153 // /* Set seed for random noise generator */
154 // srand2(1);
155
156 // /* Add noise to image */
157 // for ( i = 0; i < input_img.height; i++ )
158 // for ( j = 1; j < input_img.width-1; j++ ) {
159 //     img2[i][j] += 32*normal();
160 // }
161
162 /* set up structure for output achromatic image */
163 /* to allocate a full color image use type 'c' */
164 get_TIFF ( &green_img, input_img.height, input_img.width, 'g' );
165 get_TIFF ( &red_img, input_img.height, input_img.width, 'g' );
166 get_TIFF ( &blue_img, input_img.height, input_img.width, 'g' );
167
168 /* set up structure for output color image */
169 /* Note that the type is 'c' rather than 'g' */
170 get_TIFF ( &color_img, input_img.height, input_img.width, 'c' );
171
172 /* copy green, red & blue component to new images */
173 for ( i = 0; i < input_img.height; i++ )
174 for ( j = 0; j < input_img.width; j++ ) {
175     pixelg = (int32_t)img2[i][j];
176     pixelr = (int32_t)img3[i][j];
177     pixelb = (int32_t)img4[i][j];
178
179     if(pixelg>255) {
180         green_img.mono[i][j] = 255;

```

```
181     }
182     else {
183         if(pixelg<0) green_img.mono[i][j] = 0;
184         else green_img.mono[i][j] = pixelg;
185     }
186
187     if(pixelr>255) {
188         red_img.mono[i][j] = 255;
189     }
190     else {
191         if(pixelr<0) red_img.mono[i][j] = 0;
192         else red_img.mono[i][j] = pixelr;
193     }
194
195     if(pixelb>255) {
196         blue_img.mono[i][j] = 255;
197     }
198     else {
199         if(pixelb<0) blue_img.mono[i][j] = 0;
200         else blue_img.mono[i][j] = pixelb;
201     }
202 }
203
204 // /* Illustration: constructing a sample color image -- interchanging the red and
green components from the input color image */
205 // for ( i = 0; i < input_img.height; i++ )
206 //     for ( j = 0; j < input_img.width; j++ ) {
207 //         color_img.color[0][i][j] = input_img.color[1][i][j];
208 //         color_img.color[1][i][j] = input_img.color[0][i][j];
209 //         color_img.color[2][i][j] = input_img.color[2][i][j];
210 //     }
211
212 /* Illustration: constructing a sample color image -- put 3 image (green,red,blue)
into 1 image */
213 for ( i = 0; i < input_img.height; i++ )
214     for ( j = 0; j < input_img.width; j++ ) {
215         color_img.color[0][i][j] = red_img.mono[i][j];
216         color_img.color[1][i][j] = green_img.mono[i][j];
217         color_img.color[2][i][j] = blue_img.mono[i][j];
218     }
219
220 // /* open green image file */
221 // if ( ( fp = fopen ( "green.tif", "wb" ) ) == NULL ) {
222 //     fprintf ( stderr, "cannot open file green.tif\n");
223 //     exit ( 1 );
224 // }
225
226 // /* write green image */
227 // if ( write_TIFF ( fp, &green_img ) ) {
228 //     fprintf ( stderr, "error writing TIFF file %s\n", argv[2] );
229 //     exit ( 1 );
230 // }
231
232 // /* close green image file */
233 // fclose ( fp );
234
235
236 /* open color image file */
237 if ( ( fp = fopen ( "color.tif", "wb" ) ) == NULL ) {
238     fprintf ( stderr, "cannot open file color.tif\n");
```



```
239     exit ( 1 );
240 }
241
242 /* write color image */
243 if ( write_TIFF ( fp, &color_img ) ) {
244     fprintf ( stderr, "error writing TIFF file %s\n", argv[2] );
245     exit ( 1 );
246 }
247
248 /* close color image file */
249 fclose ( fp );
250
251 /* de-allocate space which was used for the images */
252 free_TIFF ( &(input_img) );
253 free_TIFF ( &(green_img) );
254 free_TIFF ( &(red_img) );
255 free_TIFF ( &(blue_img) );
256 free_TIFF ( &(color_img) );
257
258 free_img( (void**)img1 );
259 free_img( (void**)img2 );
260 free_img( (void**)img3 );
261 free_img( (void**)img4 );
262 free_img( (void**)img_r );
263 free_img( (void**)img_b );
264
265 return(0);
266 }
267
268 void error(char *name)
269 {
270     printf("usage: %s image.tiff \n\n",name);
271     printf("this program reads in a 24-bit color TIFF image.\n");
272     printf("It then horizontally filters the green component, adds noise,\n");
273     printf("and writes out the result as an 8-bit image\n");
274     printf("with the name 'green.tiff'.\n");
275     printf("It also generates an 8-bit color image,\n");
276     printf("that swaps red and green components from the input image");
277     exit(1);
278 }
279
280
```

```

1  //-----5 IIR Filter-----
2  #include <math.h>
3  #include "tiff.h"
4  #include "allocate.h"
5  #include "randlib.h"
6  #include "typeutil.h"
7
8  void error(char *name);
9
10 int main (int argc, char **argv)
11 {
12     FILE *fp;
13     struct TIFF_img input_img, green_img, red_img, blue_img, color_img;
14     double **img1, **imgr, **imgb, **img2, **img3, **img4;
15     int32_t i, j, pixelg, pixelr, pixelb;
16
17     /* accepts a command line argument specifying the value of rho */
18     // scanf("%lf", &rho);
19
20     if ( argc != 2 ) error( argv[0] );
21
22     /* open image file */
23     if ( ( fp = fopen ( argv[1], "rb" ) ) == NULL ) {
24         fprintf ( stderr, "cannot open file %s\n", argv[1] );
25         exit ( 1 );
26     }
27
28     /* read image */
29     if ( read_TIFF ( fp, &input_img ) ) {
30         fprintf ( stderr, "error reading file %s\n", argv[1] );
31         exit ( 1 );
32     }
33
34     /* close image file */
35     fclose ( fp );
36
37     /* check the type of image data */
38     if ( input_img.TIFF_type != 'c' ) {
39         fprintf ( stderr, "error: image must be 24-bit color\n" );
40         exit ( 1 );
41     }
42
43     /* Allocate image of double precision floats */
44     img1 = (double **)get_img(input_img.width, input_img.height, sizeof(double));
45     imgr = (double **)get_img(input_img.width, input_img.height, sizeof(double));
46     imgb = (double **)get_img(input_img.width, input_img.height, sizeof(double));
47     img2 = (double **)get_img(input_img.width, input_img.height, sizeof(double));
48     img3 = (double **)get_img(input_img.width, input_img.height, sizeof(double));
49     img4 = (double **)get_img(input_img.width, input_img.height, sizeof(double));
50
51     // /* Initialize the img arrays */
52     // for ( i = 0; i < input_img.height; i++ )
53     // for ( j = 0; j < input_img.width; j++ ) {
54     //     img1[i][j] = 0;
55     //     img2[i][j] = 0;
56     // }
57
58
59     /* copy green, red & blue component to double array */
60     for ( i = 0; i < input_img.height; i++ )

```

```

61  for ( j = 0; j < input_img.width; j++ ) {
62      img1[i][j] = input_img.color[1][i][j];
63      imgr[i][j] = input_img.color[0][i][j];
64      imgb[i][j] = input_img.color[2][i][j];
65  }
66
67
68  /* Filter image with the IIR Filter */
69  for ( i = 0; i < input_img.height; i++ )
70  for ( j = 0; j < input_img.width; j++ ) {
71      // img2[i][j] = (img1[i][j-1] + img1[i][j] + img1[i][j+1])/3.0;
72      img2[i][j] = 0.01*img1[i][j];
73      img3[i][j] = 0.01*imgr[i][j];
74      img4[i][j] = 0.01*imgb[i][j];
75      if (i>0) {
76          img2[i][j] = img2[i][j] + 0.9*img2[i-1][j];
77          img3[i][j] = img3[i][j] + 0.9*img3[i-1][j];
78          img4[i][j] = img4[i][j] + 0.9*img4[i-1][j];
79      }
80      if (j>0) {
81          img2[i][j] = img2[i][j] + 0.9*img2[i][j-1];
82          img3[i][j] = img3[i][j] + 0.9*img3[i][j-1];
83          img4[i][j] = img4[i][j] + 0.9*img4[i][j-1];
84      }
85      if (i>0 && j>0) {
86          img2[i][j] = img2[i][j] - 0.81*img2[i-1][j-1];
87          img3[i][j] = img3[i][j] - 0.81*img3[i-1][j-1];
88          img4[i][j] = img4[i][j] - 0.81*img4[i-1][j-1];
89      }
90  }
91
92  /* Fill in boundary pixels */
93
94  // for ( i = 0; i < input_img.height; i++ ) {
95  //     img2[i][0] = 0;
96  //     img2[i][input_img.width-1] = 0;
97  // }
98
99  // /* Set seed for random noise generator */
100 // srand2(1);
101
102 // /* Add noise to image */
103 // for ( i = 0; i < input_img.height; i++ )
104 // for ( j = 1; j < input_img.width-1; j++ ) {
105 //     img2[i][j] += 32*normal();
106 // }
107
108 /* set up structure for output achromatic image */
109 /* to allocate a full color image use type 'c' */
110 get_TIFF ( &green_img, input_img.height, input_img.width, 'g' );
111 get_TIFF ( &red_img, input_img.height, input_img.width, 'g' );
112 get_TIFF ( &blue_img, input_img.height, input_img.width, 'g' );
113
114 /* set up structure for output color image */
115 /* Note that the type is 'c' rather than 'g' */
116 get_TIFF ( &color_img, input_img.height, input_img.width, 'c' );
117
118 /* copy green, red & blue component to new images */
119 for ( i = 0; i < input_img.height; i++ )
120 for ( j = 0; j < input_img.width; j++ ) {

```

```
121 pixelg = (int32_t)img2[i][j];
122 pixelr = (int32_t)img3[i][j];
123 pixelb = (int32_t)img4[i][j];
124
125 if(pixelg>255) {
126     green_img.mono[i][j] = 255;
127 }
128 else {
129     if(pixelg<0) green_img.mono[i][j] = 0;
130     else green_img.mono[i][j] = pixelg;
131 }
132
133 if(pixelr>255) {
134     red_img.mono[i][j] = 255;
135 }
136 else {
137     if(pixelr<0) red_img.mono[i][j] = 0;
138     else red_img.mono[i][j] = pixelr;
139 }
140
141 if(pixelb>255) {
142     blue_img.mono[i][j] = 255;
143 }
144 else {
145     if(pixelb<0) blue_img.mono[i][j] = 0;
146     else blue_img.mono[i][j] = pixelb;
147 }
148 }
149
150 // /* Illustration: constructing a sample color image -- interchanging the red and
green components from the input color image */
151 // for ( i = 0; i < input_img.height; i++ )
152 //     for ( j = 0; j < input_img.width; j++ ) {
153 //         color_img.color[0][i][j] = input_img.color[1][i][j];
154 //         color_img.color[1][i][j] = input_img.color[0][i][j];
155 //         color_img.color[2][i][j] = input_img.color[2][i][j];
156 //     }
157
158 /* Illustration: constructing a sample color image -- put 3 image (green,red,blue)
into 1 image */
159 for ( i = 0; i < input_img.height; i++ )
160     for ( j = 0; j < input_img.width; j++ ) {
161         color_img.color[0][i][j] = red_img.mono[i][j];
162         color_img.color[1][i][j] = green_img.mono[i][j];
163         color_img.color[2][i][j] = blue_img.mono[i][j];
164     }
165
166 // /* open green image file */
167 // if ( ( fp = fopen ( "green.tif", "wb" ) ) == NULL ) {
168 //     fprintf ( stderr, "cannot open file green.tif\n");
169 //     exit ( 1 );
170 // }
171
172 // /* write green image */
173 // if ( write_TIFF ( fp, &green_img ) ) {
174 //     fprintf ( stderr, "error writing TIFF file %s\n", argv[2] );
175 //     exit ( 1 );
176 // }
177
178 // /* close green image file */
```



```
179 // fclose ( fp );
180
181
182 /* open color image file */
183 if ( ( fp = fopen ( "color.tif", "wb" ) ) == NULL ) {
184     fprintf ( stderr, "cannot open file color.tif\n");
185     exit ( 1 );
186 }
187
188 /* write color image */
189 if ( write_TIFF ( fp, &color_img ) ) {
190     fprintf ( stderr, "error writing TIFF file %s\n", argv[2] );
191     exit ( 1 );
192 }
193
194 /* close color image file */
195 fclose ( fp );
196
197 /* de-allocate space which was used for the images */
198 free_TIFF ( &(input_img) );
199 free_TIFF ( &(green_img) );
200 free_TIFF ( &(red_img) );
201 free_TIFF ( &(blue_img) );
202 free_TIFF ( &(color_img) );
203
204 free_img( (void**)img1 );
205 free_img( (void**)img2 );
206 free_img( (void**)img3 );
207 free_img( (void**)img4 );
208 free_img( (void**)img_r );
209 free_img( (void**)img_b );
210
211 return(0);
212 }
213
214 void error(char *name)
215 {
216     printf("usage: %s image.tiff \n\n",name);
217     printf("this program reads in a 24-bit color TIFF image.\n");
218     printf("It then horizontally filters the green component, adds noise,\n");
219     printf("and writes out the result as an 8-bit image\n");
220     printf("with the name 'green.tiff'.\n");
221     printf("It also generates an 8-bit color image,\n");
222     printf("that swaps red and green components from the input image");
223     exit(1);
224 }
225
226
```